Column #142, March 2007 by Jon Williams:

Livin' Life on the SX28

It occurred to the other day that I've been programming in one form of BASIC or another for over 25 years now… wow, that seems like a long time. I taught myself to program on the venerable Timex-Sinclair 1000, my first "real" computer, which I purchased in the fall of 1981. One of my favorite TS-1000 programs was a version of Conway's Game of Life, a simple artificial life simulation. I used to start the program before work and was always excited to come home and see if the "colony" was still evolving, had reached a state of equilibrium, or had just died. Honestly, I was always saddened when the latter event occurred – imagine being saddened by the "death" of a simulated cell colony… welcome to my wackiness!

Conway's Game of Life (CGoL) is a very simple program, and though it's been around since the 70's, it is still considered an important learning tool. I was telling my friend, Ryan Clarke, a professor at the University of Advancing Technology in Phoenix about this project and he told me that there are at least two courses on their campus that use CGoL as part of the curriculum. That's the thing about CGoL; it's simple, it's elegant, and yet it has implications in so many fields from basic gaming to advanced robotics.

In case you've never seen CGoL, it works like this: a rectangular grid serves as the home of a digital cell colony. A set of rules are applied that cause the colony to evolve from generation-to-generation.

Ultimately, the colony with either:

1) Die (no living cells)
2) Live in static equilibrium (no cells change)
3) Live in dynamic equilibrium (cells change in a repeating pattern)

The rules that drive inter-generation change are simple, and are based on the number of living "neighbors" that surround each cell.

4) With one or fewer neighbors, the cell dies (of loneliness)
5) With two neighbors, there is no change in the cell state
6) With three neighbors, the cell lives
7) With four neighbors, the cell dies (of over-crowding)

For me, there are few more compelling programs than Conway's Game of Life. My rediscovery (running Java versions online) of CGoL caused me to wonder if I could translate it to the SX. It was easy on the TS-1000 (or other "big" PC), but the SX28 (using SX/B) doesn't support multi-dimensional arrays and that's a requirement to manage the cell colony grid.

I decided to give it a shot for two reasons: First, it would just be plain fun and would allow me to incorporate some electronics into one of Joshua's (my youngest brother) paintings. Second, it would give me a reason to build a platform to experiment with discrete LED multiplexing. In fact, I could build a very generic circuit that could is, essentially, a mini game console and CGoL would be the first demo. So that's what I did.

The circuit is easy, and by using the SX28 the logical size of the grid is 8x8; this allows us to use the pins on RB to control the LED cathodes and the pins on RC to control the LED anodes. This leaves the pins on RA available for button inputs; again, the circuit generic and can be used for a whole host of experiments. Figures 142.1 (processor and buttons) and 142.2 (LED matrix) show the schematic.

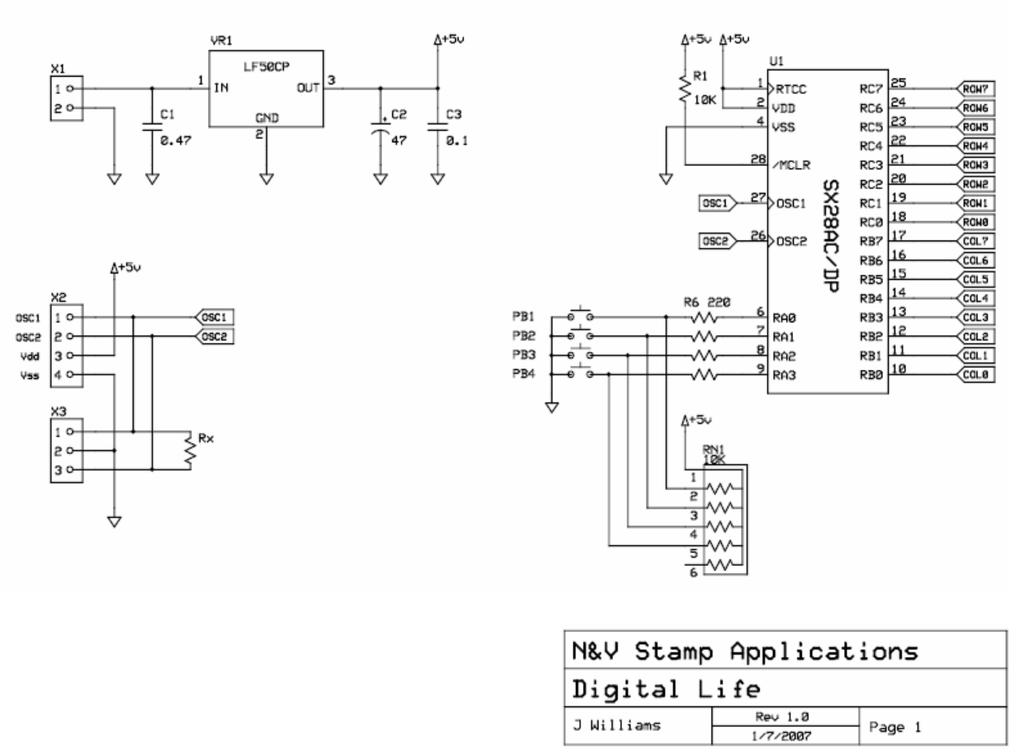


Figure 142.1: Digital Life Schematic Page 1

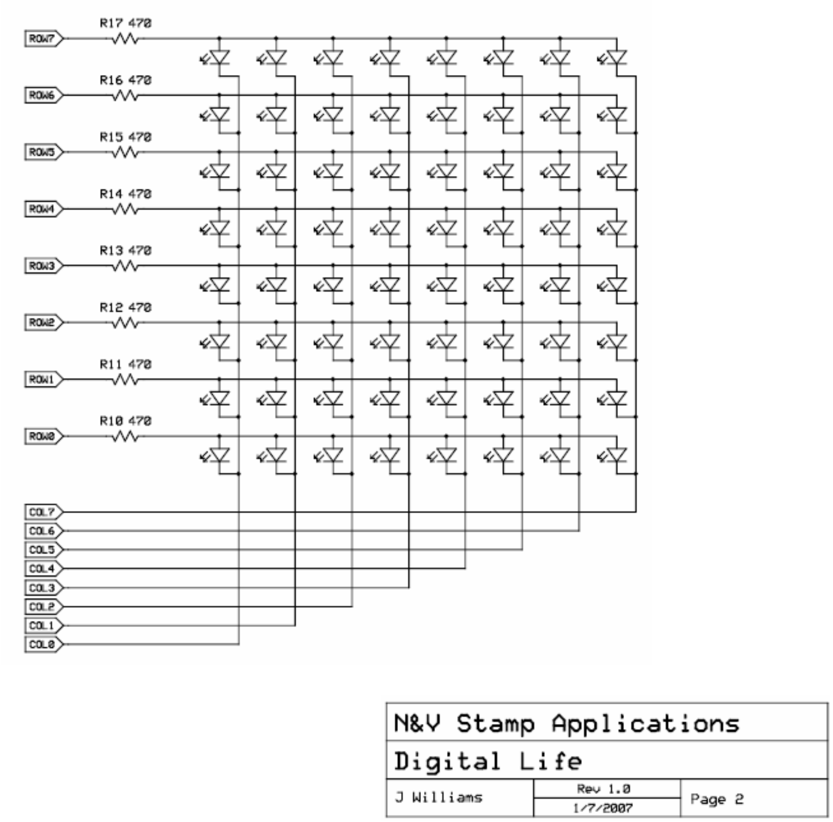


Figure 142.2: Digital Life Schematic Page 2

Now, I'm pretty good with a soldering iron, but there was no way in Heaven or on Earth that I was going to connect the processor and 64 discrete LEDs using point-to-point wiring. If you choose to go that route, you're a braver soul than me. As with the Pinewood Derby Lane Timer we made in January, I entered the circuit in ExpressSCH and then created the board in ExpressPCB.

I may have made this statement before, but I think it's worth repeating: DO NOT – under any circumstances – be tempted to skip past ExpressSCH and go right to ExpressPCB. It's not that I layout a lot of boards, but I had tried ExpressPCB way back before ExpressSCH was part of the package, and while the PCB layout program is very nice and easy to work with, the value to connecting to a schematic [netlist] as an aid to the PCB layout cannot be overstated.

Of course, for this project – should you like it as is – you don't have to worry about that as I've already done the layout work (which took about eight hours). But… if you decide to make a change, copy and modify the schematic first, then open and copy the PCB file, finally linking it to the new schematic. Make you PCB changes from there allowing ExpressPCB to tell you what connects to what. Please trust me on this as there is nothing more frustrating than spending time on a nice, neat PCB layout, only to find that when it gets back from the board house there's a self-created error.

For circuit components, I tend to order from Mouser. When I lived in Dallas I had the opportunity to visit their facilities and it is really a first-class operation. Their prices are good, too. Of course, vendors like Digi-Key and Jameco also provide great products and service. I just want to let you know that the schematic file that you can download as part of this article includes Mouser part numbers. There is nothing exotic, though, and you should be able to get the components anywhere.

Construction is easy – it's really just a big solder job. As always, I start with the "low lying" components (e.g., resistors) and work my way up to the taller components like the power-supply cap and the power connector. I started by soldering in everything except the LEDs. Despite my confidence in the schematic and the board, I certainly wasn't going to spend the time to solder in 64 discrete LEDs only to find I had screwed up. With everything but the LEDs in place I connected power and download a little test program to poll and display the status of the switch inputs (I used the Debug window for this). Guess what? – I actually had a duff SX (one pin on RA, anyway).

After I knew the power supply and buttons worked, the next step was the LEDs. Being a cautious guy, however, I soldered them in eight at a time and then ran a quick test program to make sure that those in the board were working. In the end, everything worked perfectly and it was time to start on the Game of Life program.

Creating Digital Life

In order to use the 8x8 LED matrix as a display for the game, it needs constant (periodic) refreshing – a logical choice is to use an interrupt. To keep things easy, I decided on a one millisecond interrupt period; there is nothing magic about that value except that it's a convenient way to enable fairly precise delays.

Wait a minute, what about PAUSE? Well, remember that when we activate periodic interrupts any timing sensitive instructions will be adversely affected. So, you’ll see that there is no PAUSE instruction used in the program, and yet there is a way to do delays with millisecond (+0/-1) resolution.

Let’s have a look at the interrupt code.

```
INT_HANDLER:
  Anodes = %00000000
  READ Col_Mask + col, Cathodes
  Anodes = dispBuf(col)
  INC col
  IF col = 8 THEN
    col = 0
  ENDIF

Update_Timer:
  IF ms > 0 THEN
    DEC ms
  ENDIF

LFSR:
  IF seed = 0 THEN
    seed = 24
  ENDIF
  ASM
    MOV  W, #$1D
    CLRB C
    RL   seed
    SNB  C
    XOR  seed, W
  ENDASM

ISR_Exit:
  RETURNINT
```

As you can see, the ISR code is divided into three distinct elements: display update, timer update, and random value update. First things first. The bits to be displayed are kept in an array called dispBuf(); with eight bytes this gives us a 64-bit (8x8) array for the colony. The orientation of the LEDs on the board is designed to match Cartesian coordinates, that is, the lower left LED corresponds to dispBuf(0), bit 0, and the upper right LED corresponds to dispBuf(7), bit 7.

The display update starts by clearing the anode outputs and then reading the column mask from a DATA table (using the current column value). I like the table approach versus creating a mask by bit shifting; it seems more obvious and I think it adds a bit of flexibility.

With the column selected, the anodes are read from dispBuf(col); at this point, the column is being displayed (until the next ISR call). Then the column pointer is incremented and wrapped back to zero once it passes the 7th column. Note that the variable, col, should not be manipulated outside the ISR.

The second section updates another dedicated ISR variable called ms. This variable is a word (16 bits) so that we can create delays up to 65,535 milliseconds. Through each pass of the ISR this variable is checked for being non-zero; when it is it gets decremented. We'll see how to use this value in place of PAUSE in just a bit.

Finally, there is a section called LFSR (which stands for linear feedback shift register). In this program it is used to randomize the third dedicated ISR variable called seed. When I first started the program I used the built-in RANDOM function but found that the results weren't visually pleasing. So I went out to James Newton's SX List (www.sxlist.com, an excellent resource) and found an 8-bit LFSR routine that gave me the visual results I was looking for.

You might wonder why this is embedded in the ISR. Of course, I could have created a traditional function but I thought it would be nice to have a running random number. As you can imagine, I work with a lot of folks that are new to BASIC Stamps and the SX and the interesting thing is that many of them believe that the RANDOM function is a "background" process that runs all the time. Well, in this case it is. We simply need to copy the value of seed whenever we want an 8-bit random number.

Scrollin', Scrollin', Scrollin'…

As one of the possible uses for the 8x8 LED matrix is a scrolling display, let's add that to the front end of the game program to make things a bit snazzy. Sticking with the K.I.S.S. principle, we'll store the scrolling banner in a big DATA table and simply loop thought it, the effect is an 8x8 window sliding over the banner as shown in Figure 142.3.

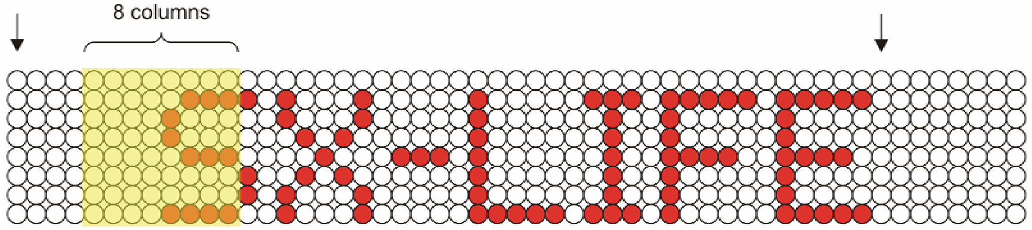


Figure 142.3: Scrolling Banner Map

Note that there are eight blank columns on either end of the banner text; the front-end blanks let the banner scroll on to the display; the back-end blanks push it off.

The arrows above the figure indicate the starting and ending columns for the main portion of the loop. An inner loop will iterate from that starting point out seven additional columns to fill the display buffer. Here's the code:

```
Start:
  ' scrolling banner
  FOR tmpB1 = 0 TO 45
    tmpB2 = tmpB1
    FOR idxCol = 0 TO 7
      READINC Banner + tmpB2, dispBuf(idxCol)
    NEXT
    DELAY 75
  NEXT
```

The outer (scrolling) loop is controlled by tmpB1. A copy is made in tmpB2 that will be used as an offset for the READINC function. The inner loop, controlled by idxCol, runs eight times to fill the eight columns of the display with values from the DATA table. The nice thing about the READINC function is that it automatically updates the offset variable (tmpB2) for us. Once the display buffer is filled we need to insert a short delay to control the column-to-column scrolling speed.

Here's the delay subroutine that replaces the use of PAUSE in this program.

```
DELAY:
  IF __PARAMCNT = 1 THEN
    ms = __PARAM1
  ELSE
    ms = __WPARAM12
  ENDIF
  DO
    ' wait for timer to expire
  LOOP UNTIL ms = 0
  RETURN
```

Pretty simple, isn't it? The subroutine is setup to allow a byte or word to be passed to it. That value gets loaded into variable ms and then a DO-LOOP holds the program right where it is until ms is zero. Remember, ms is being decremented every millisecond in the ISR when it's greater than zero. This is a good bit of code for your SX/B library, especially as you delve more deeply into interrupts.

Framed!

We've just seen one style of animation, how about another – something akin to cell animation in a cartoon. We can do this kind of animation by storing the frames in a DATA table. For frames that are going to run in order, as we will do here, the code is assisted by lining up the frames end-to-end. Figure 142.4 shows a simple four-frame sequence that will run after the scrolling banner moves out of the display.

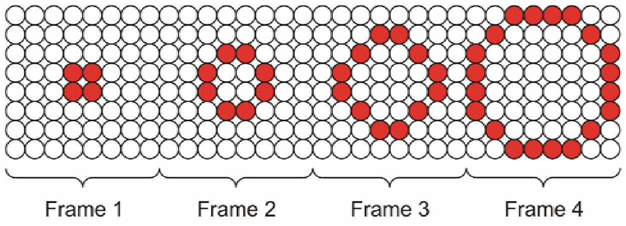


Figure 142.4: Maps for Animation Frames

```
Frames_Animation:
  FOR tmpB1 = 0 TO 24 STEP 8
    tmpB2 = tmpB1
    FOR idxCol = 0 TO 7
      READINC Frame1 + tmpB2, dispBuf(idxCol)
    NEXT
    DELAY 100
  NEXT
```

It's clear that the code is identical to the scrolling animation except that the outer loop steps eight columns (one frame) each time through, and the base pointer starts at Frame1 instead of Banner.

Now that the fanfare is complete we can get into the meat of the Game of Life program. At the start of the main program loop a question mark will be displayed and then the buttons scanned.

```
Main:
  FOR idxCol = 0 TO 7
    READ Q_Mark + idxCol, dispBuf(idxCol)
  NEXT

User_Select:
  DO
```

```
    btns = SCAN_BUTTONS
  LOOP UNTIL btns <> %0000

Here’s the routine the scans and debounces the buttons.

SCAN_BUTTONS:
  tmpB1 = %00000000
  FOR tmpB2 = 1 TO 5
    tmpB1 = tmpB1 | BtnPort
    DELAY 10
  NEXT
  tmpB1 = tmpB1 ^ %11111111
  tmpB1 = tmpB1 & %00001111
  RETURN tmpB1
```

The buttons are configured as active-low inputs to the SX so the subroutine starts by clearing the result variable, tmpB1. It then runs a short loop with a 10 millisecond pad between scans. With active-low buttons, a short release (bounce) will cause the input to go high (because of the pull-up) and the 1 bit will get OR’d into the result; this will stay there through the entire scan cycle.

At the end the scan result gets inverted to make the buttons look active-high and the unused inputs are stripped away. The design of this function ensures that a button must be down and stay down for 50 milliseconds to call it a good press. Using the loop to check the switch state at short intervals helps eliminate contact bounce and noise.

With the switches scanned and debounced the program can check for and process valid “press” events. The first button will cause the cell matrix to be randomly populated.

```
Randomize_Cells:
  IF btns = B_RAND THEN
    FOR idxCol = 0 TO 7
      dispBuf(idxCol) = seed
      DELAY 5
    NEXT
    DELAY 50
    GOTO User_Select
  ENDIF
```

Here you can see the use of the system random value, seed. Note that there is a short delay in the middle of the cell-populating loop; this lets the LFSR code in the ISR run a few times between calls. A short delay is also added after the loop just to hold the display a bit if the randomizing button is held down.

The next two buttons load fixed colony patterns from DATA tables. The first pattern loads “blinkers” that will oscillate in a state of dynamic equilibrium. The second pattern is called a “glider.” It will move from the lower left corner to the upper right corner, ultimately achieving a state of static equilibrium (a living colony that does not change from one generation to the next).

```
Load_Pattern1:
  IF btns = B_PAT1 THEN
    RELEASE
    FOR idxCol = 0 TO 7
      READ Pattern1 + idxCol, dispBuf(idxCol)
    NEXT
    GOTO User_Select
  ENDIF

Load_Pattern2:
  IF btns = B_PAT2 THEN
    RELEASE
    FOR idxCol = 0 TO 7
      READ Pattern2 + idxCol, dispBuf(idxCol)
    NEXT
    GOTO User_Select
  ENDIF
```

As it stands now the program only has two fixed patterns in memory. If you want to add more, then change the code to keep track of a pattern pointer and use the PB2 and PB3 buttons to increment or decrement that pointer before loading the pattern.

The last button launches the game with generation zero being whatever the display is current showing – including the initial question mark prompt. This section also handles getting back to the button scanning if more than one button was pressed.

```
Run_Simulation:
  IF btns = B_RUN THEN
    RELEASE
    GOTO Its_Alive
  ELSE
    GOTO User_Select
  ENDIF

Within the button handlers there is a subroutine employed called RELEASE.
This is used to hold the program until the buttons are cleared.

RELEASE:
  DO
    tmpB1 = SCAN_BUTTONS
  LOOP UNTIL tmpB1 = %0000
```

```
  RETURN
```

As you can see, this routine uses a work variable (tmpB1) so the result of our last button scan (btns) is not affected.

And now we get to the nitty-gritty. The code at Its_Alive is what runs the game logic. What this section does is iterate through all of the cells of the display buffer, counting the neighbors for each. The rule set is applied and the result are written to a secondary buffer called newGen(). We can’t operate directly on the display buffer as this would change the colony mid generation and the results would not accurately reflect the rules. Once all of the cells in the display buffer have been scanned and analyzed, the newGen() buffer is copied to the display. After a scan of the keys and short delay the whole process starts over.

```
Its_Alive:
  FOR idxCol = 0 TO 7
    FOR idxRow = 0 TO 7
      COUNT_NEIGHBORS
      IF neighbors <= 1 THEN
        ' alone... dies
        newGen(idxCol) = ↵
          CLR_BIT newGen(idxCol), idxRow
      ENDIF
      IF neighbors = 2 THEN
        ' no change
        cell = GET_BIT dispBuf(idxCol), idxRow
        newGen(idxCol) = ↵
          PUT_BIT newGen(idxCol), idxRow, cell
      ENDIF
      IF neighbors = 3 THEN
        ' lives!
        newGen(idxCol) = ↵
          SET_BIT newGen(idxCol), idxRow
      ENDIF
      IF neighbors >= 4 THEN
        ' crowded... dies
        newGen(idxCol) = ↵
          CLR_BIT newGen(idxCol), idxRow
      ENDIF
    NEXT
  NEXT

  FOR idxCol = 0 TO 7
    dispBuf(idxCol) = newGen(idxCol)
  NEXT

  DELAY 200
  btns = SCAN_BUTTONS
```

```
  IF btns = %0000 THEN Its_Alive
  RELEASE
  GOTO Main
```

I moved the code for COUNT_NEIGHBORS out of the main loop because it was just very big and bulky. I tried to figure out some elegant way to do the testing, but in the end found that it was simply best to use a bit of blunt force. It's long so I won't show the whole thing here, but what you'll see when you download the full listing is that COUNT_NEIGHHBORS has eight sections that look like this:

```
  chkCol = idxCol - 1
  chkRow = idxRow - 1
  cell = GET_CELL
  neighbors = neighbors + cell
```

You see, each cell has eight possible neighbors – but not all cells; the corner cells, for example, only have three neighbors. To deal with this I created a routine called GET_CELL which is really just a wrapper for GET_BIT. The code in GET_CELL ensures that we don't try to ask for a bit that exceeds the bounds of the array.

```
GET_CELL:
  tmpB1 = 0
  IF chkCol >= 0 THEN
    IF chkCol <= 7 THEN
      IF chkRow >= 0 THEN
        IF chkRow <= 7 THEN
          tmpB1 = ↵
            GET_BIT dispBuf(chkCol), chkRow
        ENDIF
      ENDIF
    ENDIF
  ENDIF
  RETURN tmpB1
```

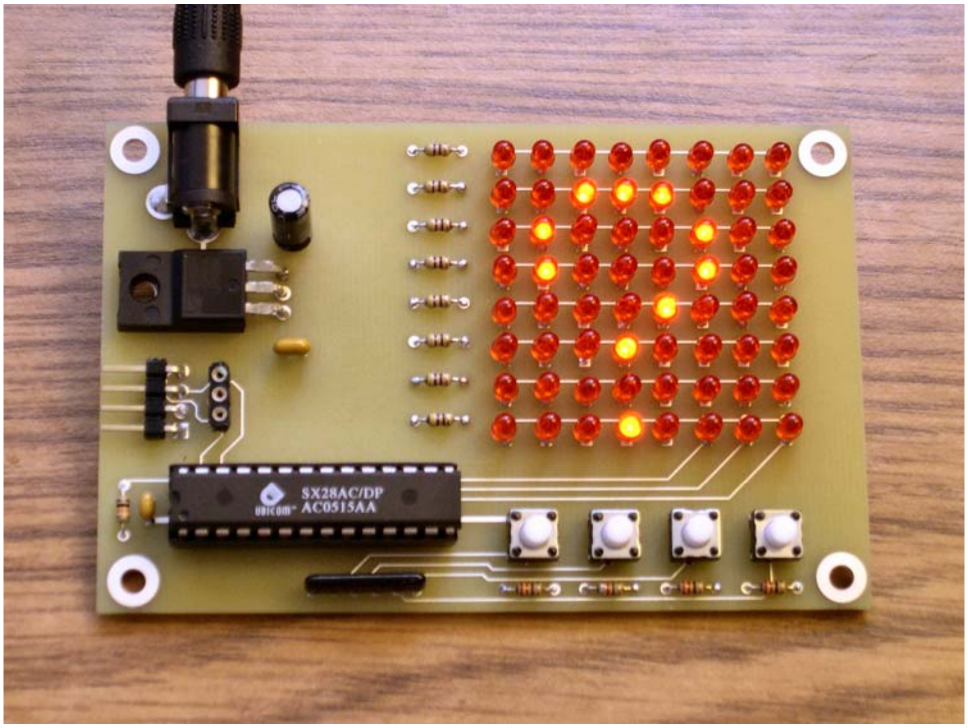
Those of us that have been using the BS2 family for a long time are well aware of and enjoy the use of the .LOWBIT() modifier of variables – this does not exist in SX/B. Well, not as part of the standard language, so we just have to add it (or something like it) ourselves.

To get .LOWBIT() functionality actually requires three separate functions; they're actually very simple and provide a bit more flexibility than .LOWBIT(). These functions expect a byte and return a byte; this lets us send the result to any variable we choose, including to the variable who's value was passed as a parameter to the function.

```
GET_BIT:
  tmpB1 = __PARAM1
```

```
  tmpB2 = __PARAM2

  tmpB2 = 1 << tmpB2
  tmpB1 = tmpB1 & tmpB2
  IF tmpB1 > 0 THEN
    tmpB1 = 1
  ENDIF
  RETURN tmpB1

SET_BIT:
  tmpB1 = __PARAM1
  tmpB2 = __PARAM2

  tmpB2 = 1 << tmpB2
  tmpB1 = tmpB1 | tmpB2
  RETURN tmpB1

CLR_BIT:
  tmpB1 = __PARAM1
  tmpB2 = __PARAM2

  tmpB2 = 1 << tmpB2
  tmpB2 = tmpB2 ^ %11111111
  tmpB1 = tmpB1 & tmpB2
  RETURN tmpB1
```

All three functions take the byte parameter into tmpB1 and the position value into tmpB2. The position value is turned into a bit mask for that position. For bit checking or setting, the mask is left as is; for bit clearing the mask gets inverted. The functions only work on bytes, but could easily be modified to work with words.



Let The Games Begin!

And there you have it – a simple digital game console and enough framework functions to create a wide variety of low-resolution games. I wonder… what could you create with this neat little platform? It's easy to get jaded by all the resources we have with PC games and even hand-held units with color LCDs; can you create something compelling and entertaining with a simple 8x8 LED matrix and four pushbuttons? I'm betting you can, if you'll simply put your mind to it and let your imagination run wild.

Until next time… Happy Stamping!

Resources:

http://www.sxlist.com
http://en.wikipedia.org/wiki/Conway's_Game_of_Life

Project Bill of Materials		
Designator	**Value**	**Source**
C1	0.47	Mouser 80-C320C474M5U
C2	47	Mouser 647-UVR1C470MDD
C3	0.1	Mouser 80-C315C104M5U
D1-D64	LED	Mouser 859-LTL-4222N
PB1-PB4		Mouser 612-TL59F160Q
R1	10K	Mouser 299-10K-RC
R6-R9	220	Mouser 299-220-RC
R10-R17	470	Mouser 299-470-RC
RN1	10K	Mouser 264-10K-RC
Rx	(optional)	
U1	SX28AC/DP	Parallax SX28AC/DP-G
	socket	Mouser 571-3902619
VR1	LF50CP	Mouser 511-LF50CP
X1	2.1 mm	Mouser 806-KLDX-0202-A
X2	for SX-Key	Mouser 517-5111TG
X3	for resonator	Mouser 506-510-AG91D
PCB		From ExpressPCB.com

Source Code

```
'
'   File...... Life.SXB
'   Purpose... Digital Life Program
'   Author.... Jon Williams
'              Copyright (c) 2007 Jon Williams
'              Some Rights Reserved
'              -- see http://creativecommons.org/licenses/by/2.5/
'   E-mail.... jwilliams@efx-tek.com
'   Started...
'   Updated... 18 JAN 2007
'
' =========================================================================


' -------------------------------------------------------------------------
' Program Description
' -------------------------------------------------------------------------
'
' Conway's Game of Life simulation.
' -- see: http://en.wikipedia.org/wiki/Conway's_Game_of_Life
'
' PB1 - Randomize population
```

```
' PB2 - load pattern 1
' PB3 - load pattern 2
' PB4 - Run
'
' Pressing any button while the simulation is running will stop it and
' cause the program to return to a "?" prompt.


' -------------------------------------------------------------------------
' Conditional Compilation Symbols
' -------------------------------------------------------------------------


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

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX, BOR42
FREQ            4_000_000
ID              "LIFE"


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

Anodes          PIN     RC  OUTPUT
Cathodes        PIN     RB  OUTPUT
BtnPort         PIN     RA


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

Yes             CON     1
No              CON     0

B_RAND          CON     %0001
B_PAT1          CON     %0010
B_PAT2          CON     %0100
B_RUN           CON     %1000

Alive           CON     1
Dead            CON     0


' -------------------------------------------------------------------------
' Variables
' -------------------------------------------------------------------------
```

```
dispBuf        VAR     Byte (8)               ' LED display buffer
newGen         VAR     Byte (8)               ' new generation buffer

col            VAR     Byte                   ' display column (ISR)
ms             VAR     Word                   ' for delay timing (ISR)
seed           VAR     Byte                   ' running random value (ISR)

btns           VAR     Byte
btnRand        VAR     btns.0
btnPat1        VAR     btns.1
btnPat2        VAR     btns.2
btnRun         VAR     btns.3

idxCol         VAR     Byte
idxRow         VAR     Byte
neighbors      VAR     Byte
chkRow         VAR     Byte
chkCol         VAR     Byte
cell           VAR     Byte

tmpB1          VAR     Byte                   ' work variables
tmpB2          VAR     Byte
tmpB3          VAR     Byte


' -------------------------------------------------------------------------
   INTERRUPT 1000                             ' run every millisecond
' -------------------------------------------------------------------------

   GOTO INT_HANDLER


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


' -------------------------------------------------------------------------
' Subroutine Declarations
' -------------------------------------------------------------------------

DELAY          SUB     1, 2                   ' delay in 1 ms units
RELEASE        SUB     0                      ' wait for button release
COUNT_NEIGHBORS        SUB     0                     ' counts cell neighbors

SCAN_BUTTONS   FUNC    1, 0                   ' scan buttons on RA
GET_BIT        FUNC    1, 2                   ' get bit value
SET_BIT        FUNC    1, 2                   ' sets a bit in a byte
CLR_BIT        FUNC    1, 2                   ' clears a bit in a byte
PUT_BIT        FUNC    1, 3                   ' writes bitVal into byte
GET_CELL       FUNC    1, 0                   ' get cell status
```

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

Start:
  ' scrolling banner
  FOR tmpB1 = 0 TO 45                           ' columns to scroll
    tmpB2 = tmpB1                               ' copy of left column pos
    FOR idxCol = 0 TO 7                                 ' fill character buffer
      READINC Banner + tmpB2, dispBuf(idxCol)        ' load character column
    NEXT
    DELAY 75
  NEXT

Frames_Animation:
  FOR tmpB1 = 0 TO 24 STEP 8                    ' step through frames
    tmpB2 = tmpB1                               ' copy of left column pos
    FOR idxCol = 0 TO 7                                 ' fill character buffer
      READINC Frame1 + tmpB2, dispBuf(idxCol)        ' load character column
    NEXT
    DELAY 100
  NEXT

Main:
  FOR idxCol = 0 TO 7                           ' show ?
    READ Q_Mark + idxCol, dispBuf(idxCol)
  NEXT

User_Select:
  DO
    btns = SCAN_BUTTONS
  LOOP UNTIL btns <> %0000                      ' wait for button press

Randomize_Cells:
  IF btns = B_RAND THEN
    FOR idxCol = 0 TO 7
      dispBuf(idxCol) = seed                    ' randomize this column
      DELAY 5                                   ' let rand gen run
    NEXT
    DELAY 50
    GOTO User_Select
  ENDIF

Load_Pattern1:                                  ' load blinkers
  IF btns = B_PAT1 THEN
    RELEASE
    FOR idxCol = 0 TO 7
      READ Pattern1 + idxCol, dispBuf(idxCol)
    NEXT
```

```
    GOTO User_Select
  ENDIF

Load_Pattern2:                                ' load glider
  IF btns = B_PAT2 THEN
    RELEASE
    FOR idxCol = 0 TO 7
      READ Pattern2 + idxCol, dispBuf(idxCol)
    NEXT
    GOTO User_Select
  ENDIF

Run_Simulation:
  IF btns = B_RUN THEN
    RELEASE
    GOTO Its_Alive
  ELSE
    GOTO User_Select
  ENDIF

Its_Alive:
  FOR idxCol = 0 TO 7
    FOR idxRow = 0 TO 7
      COUNT_NEIGHBORS
      IF neighbors <= 1 THEN
        ' alone... dies
        newGen(idxCol) = CLR_BIT newGen(idxCol), idxRow
      ENDIF
      IF neighbors = 2 THEN
        ' no change
        cell = GET_BIT dispBuf(idxCol), idxRow
        newGen(idxCol) = PUT_BIT newGen(idxCol), idxRow, cell
      ENDIF
      IF neighbors = 3 THEN
        ' lives!
        newGen(idxCol) = SET_BIT newGen(idxCol), idxRow
      ENDIF
      IF neighbors >= 4 THEN
        ' crowded... dies
        newGen(idxCol) = CLR_BIT newGen(idxCol), idxRow
      ENDIF
    NEXT
  NEXT

  FOR idxCol = 0 TO 7                         ' update display
    dispBuf(idxCol) = newGen(idxCol)
  NEXT

  DELAY 200                                   ' inter-generation timing
  btns = SCAN_BUTTONS                         ' (plus 50 ms for scan)
  IF btns = %0000 THEN Its_Alive              ' keep going if no button
```

```
  RELEASE
  GOTO Main


' -------------------------------------------------------------------------
' Subroutine Code
' -------------------------------------------------------------------------

' Interrupt handler

INT_HANDLER:
  Anodes = %00000000                          ' clear display
  READ Col_Mask + col, Cathodes               ' enable column
  Anodes = dispBuf(col)                       ' output LEDs for column
  INC col                                     ' point to next column
  IF col = 8 THEN                             ' reached last column?
    col = 0                                   '   yes, reset
  ENDIF

Update_Timer:
  IF ms > 0 THEN                              ' delay timer running?
    DEC ms                                    '   yes, decrement
  ENDIF

LFSR:                                         ' randomize "seed"
  IF seed = 0 THEN
    seed = 24
  ENDIF
  ASM
    MOV  W, #$1D
    CLRB C
    RL   seed
    SNB  C
    XOR  seed, W
  ENDASM

ISR_Exit:
  RETURNINT

' -------------------------------------------------------------------------

' Use: DELAY msec

DELAY:
  IF __PARAMCNT = 1 THEN
    ms = __PARAM1                             ' get byte parameter
  ELSE
    ms = __WPARAM12                           ' get word parameter
  ENDIF
  DO
    ' wait for timer to expire
```

```
  LOOP UNTIL ms = 0
  RETURN

' -------------------------------------------------------------------------

' Use: result = SCAN_BUTTONS
' -- scans active-low buttons; returns 1 for pressed button
' -- routine consumes about 50 milliseconds

SCAN_BUTTONS:
  tmpB1 = %00000000                          ' assume all pressed
  FOR tmpB2 = 1 TO 5
    tmpB1 = tmpB1 | BtnPort                  ' scan port
    DELAY 10
  NEXT
  tmpB1 = tmpB1 ^ %11111111                  ' invert; 1 = pressed
  tmpB1 = tmpB1 & %00001111                  ' clear unused bits
  RETURN tmpB1

' -------------------------------------------------------------------------

RELEASE:
  DO
    tmpB1 = SCAN_BUTTONS
  LOOP UNTIL tmpB1 = %0000
  RETURN

' -------------------------------------------------------------------------

' Use: result = GET_BIT value, position
' -- returns 1 or 0

GET_BIT:
  tmpB1 = __PARAM1                           ' save value
  tmpB2 = __PARAM2                           ' save position

  tmpB2 = 1 << tmpB2                         ' create mask
  tmpB1 = tmpB1 & tmpB2                      ' isolate bit
  IF tmpB1 > 0 THEN
    tmpB1 = 1
  ENDIF
  RETURN tmpB1

' -------------------------------------------------------------------------

' Use: result = SET_BIT value, position
' -- returns copy of value with position bit set

SET_BIT:
  tmpB1 = __PARAM1                           ' save value
  tmpB2 = __PARAM2                           ' save position
```

```
  tmpB2 = 1 << tmpB2                            ' create mask
  tmpB1 = tmpB1 | tmpB2                         ' set the bit
  RETURN tmpB1

' -------------------------------------------------------------------------

' Use: result = CLR_BIT value, position
' -- returns value with position bit cleared

CLR_BIT:
  tmpB1 = __PARAM1                              ' save value
  tmpB2 = __PARAM2                              ' save position

  tmpB2 = 1 << tmpB2                            ' create mask
  tmpB2 = tmpB2 ^ %11111111                     ' invert the mask
  tmpB1 = tmpB1 & tmpB2                         ' clear the bit
  RETURN tmpB1

' -------------------------------------------------------------------------

' Use: result = PUT_BIT value, position, bitVal
' -- writes bitVal to value.position

PUT_BIT:
  tmpB1 = __PARAM1                              ' save value
  tmpB2 = __PARAM2                              ' save position
  tmpB3 = __PARAM3.0                            ' save bit value

  tmpB2 = 1 << tmpB2                            ' create mask
  IF tmpB3 = 1 THEN
    tmpB1 = tmpB1 | tmpB2                       ' set the bit
  ELSE
    tmpB2 = tmpB2 ^ %11111111                   ' invert the mask
    tmpB1 = tmpB1 & tmpB2                       ' clear the bit
  ENDIF
  RETURN tmpB1

' -------------------------------------------------------------------------

' Use: COUNT_NEIGHBORS
' -- counts live neighbors of cell in dispBuf
' -- location of cell in idxCol/idxRow

COUNT_NEIGHBORS:
  neighbors = 0                                 ' reset neighbors count

  chkCol = idxCol - 1                           ' SW
  chkRow = idxRow - 1
  cell = GET_CELL
  neighbors = neighbors + cell
```

```
  chkCol = idxCol - 1                         ' W
  chkRow = idxRow + 0
  cell = GET_CELL
  neighbors = neighbors + cell

  chkCol = idxCol - 1                         ' NW
  chkRow = idxRow + 1
  cell = GET_CELL
  neighbors = neighbors + cell

  chkCol = idxCol + 0                         ' N
  chkRow = idxRow + 1
  cell = GET_CELL
  neighbors = neighbors + cell

  chkCol = idxCol + 1                         ' NE
  chkRow = idxRow + 1
  cell = GET_CELL
  neighbors = neighbors + cell

  chkCol = idxCol + 1                         ' E
  chkRow = idxRow + 0
  cell = GET_CELL
  neighbors = neighbors + cell

  chkCol = idxCol + 1                         ' SE
  chkRow = idxRow - 1
  cell = GET_CELL
  neighbors = neighbors + cell

  chkCol = idxCol + 0                         ' S
  chkRow = idxRow - 1
  cell = GET_CELL
  neighbors = neighbors + cell

  RETURN

' -------------------------------------------------------------------------

' Use: result = GET_CELL
' -- returns value of cell (0 or 1) from dispBuf array
' -- uses bound checking for "chkCol" and "chkRow"

GET_CELL:
  tmpB1 = 0
  IF chkCol >= 0 THEN
    IF chkCol <= 7 THEN
      IF chkRow >= 0 THEN
        IF chkRow <= 7 THEN
          tmpB1 = GET_BIT dispBuf(chkCol), chkRow
```

```
        ENDIF
      ENDIF
    ENDIF
  ENDIF
  RETURN tmpB1


' =========================================================================
' User Data
' =========================================================================

Col_Mask:
  DATA  %11111110                               ' for common cathode LEDs
  DATA  %11111101
  DATA  %11111011
  DATA  %11110111
  DATA  %11101111
  DATA  %11011111
  DATA  %10111111
  DATA  %01111111

Banner:
' col    76543210
  DATA  %00000000                               ' pre-string pad
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000

Ltr_S:
  DATA  %00110001
  DATA  %01001001
  DATA  %01001001
  DATA  %01001001
  DATA  %01000110

  DATA  %00000000

Ltr_X:
  DATA  %01100011
  DATA  %00010100
  DATA  %00001000
  DATA  %00010100
  DATA  %01100011

  DATA  %00000000

Dash:
```

```
  DATA  %00001000
  DATA  %00001000
  DATA  %00001000

  DATA  %00000000

Ltr_L:
  DATA %01111111
  DATA %00000001
  DATA  %00000001
  DATA  %00000001
  DATA %00000001

  DATA  %00000000

Ltr_I:
  DATA %01000001
  DATA  %01111111
  DATA  %01000001

  DATA  %00000000

Ltr_F:
  DATA %01111111
  DATA  %01001000
  DATA  %01001000
  DATA  %01001000
  DATA  %01000000

  DATA  %00000000

Ltr_E:
  DATA %01111111
  DATA  %01001001
  DATA  %01001001
  DATA  %01001001
  DATA  %01000001

Pad2:
  DATA  %00000000                            ' post string pad
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000

' Animation frames

Frame1:
```

```
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000
  DATA  %00011000
  DATA  %00011000
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000

Frame2:
  DATA  %00000000
  DATA  %00000000
  DATA  %00011000
  DATA  %00100100
  DATA  %00100100
  DATA  %00011000
  DATA  %00000000
  DATA  %00000000

Frame3:
  DATA  %00000000
  DATA  %00111100
  DATA  %01000010
  DATA  %01000010
  DATA  %01000010
  DATA  %01000010
  DATA  %00111100
  DATA  %00000000

Frame4:
  DATA  %00111100
  DATA  %01000010
  DATA  %10000001
  DATA  %10000001
  DATA  %10000001
  DATA  %10000001
  DATA  %01000010
  DATA  %00111100

Q_Mark:
  DATA  %00000000
  DATA  %00110000
  DATA  %01000000
  DATA  %01000101
  DATA  %01001000
  DATA  %00110000
  DATA  %00000000
  DATA  %00000000

Pattern1:
  DATA  %01000000                              ' blinkers
```

```
  DATA  %01000000
  DATA  %01000100
  DATA  %00000110
  DATA  %00000110
  DATA  %00000010
  DATA  %11100000
  DATA  %00000000

Pattern2:
  DATA  %00000100                             ' glider
  DATA  %00000101
  DATA  %00000110
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000
  DATA  %00000000
```