Column #116 December 2004 by Jon Williams:

BASIC Stamp Accessories Made Easier

Not long after the BASIC Stamp started a revolution in small microcontrollers, Scott Edwards started what turned into a cottage industry: serial accessories. Thanks to the new – and free! – SX/B compiler from Parallax, you too can join the Serial Accessory club ... and do so much more.

If you were around this time last year you may remember my absolute glee at the return of the BS1 – via programming support with the BASIC Stamp IDE that runs in Windows. I've had a great time this past year, and like our project of a year ago I have created a few serial accessory devices using the BS1 as the host controller.

While this works and is fine for one-off experiments, it's not really practical from a cost standpoint, especially when one has a good idea that is marketable to a large group of users. Well, I've got good news that will top your list of stocking stuffers this holiday season: Parallax has released a free BASIC language compiler for the SX microcontroller. Yes, that's right; free.

I can hear you now, "Crimony, have they lost their minds?" Of course not. But as you've seen in recent months, Parallax has made getting started with the SX micro easier by reducing the cost of the SX-Key programming tool. By adding SX/B to the SX-Key software the SX-Key becomes an even better bargain, and now even easier to use.

From Problem Solver to Product

Before I get into the details, let me give you a bit of history, and make sure that I manage your expectations. The SX/B compiler started out as a tool to help remove some of the drudgery of getting an assembly language program started. As such, it didn't support many of the BASIC language instructions we're all used to. But it worked so well that the team responsible for putting it all together decided it would be worth pushing ahead, moving toward a full-fledged – albeit small – compiler. So over the next several months they added features that moved SX/B from simple a helper program to one that would serve the professional engineer as well as the student or hobbyist who is attempting to make the move from high-level to low-level coding.

Let me start by explaining what SX/B is and isn't. SX/B is a straight, in-line compiler that converts BASIC (very much like BS1) syntax to SX assembly code. Being an inline compiler, no attempt to optimize program space is made – this is left to the programmer (and not hard to do). Why not optimize? Well, optimizing compilers are very complicated (and generally have big price tags), and the output is not really suitable to be modified by the programmer. You can do great things with an optimizing compiler, but what you really can't do (easily) is learn from it.

And that's one of Parallax's primary goals for SX/B: to help people learn how to code in assembly language by seeing the assembly code output from the compiler. This is possible because every line of code is translated into a block of assembly instructions, and the original BASIC code is inserted into the assembly listing as a comment. This allows you to see what's happening and, when you're so inclined, to modify the assembly code before downloading to the SX.

SX/B = PBASIC?

Well, sort of. One of the things that I think many people will find when they look at the output from the SX/B compiler is that it takes a lot of code to do what seems like a simple thing. What many will conclude, I believe, is that the BASIC Stamp does far more work under the hood than was ever imagined.

To that end, the SX/B language is somewhat PBASIC compatible; the goal was to get it fairly close to BS1 syntax with a few additions (like SHIFTOUT and SHIFTIN) and changes to simplify compiler design so that it could be used as an effective learning tool. With that in mind, complex functions like SEROUT and SERIN accept only single output/input parameters. If, for example, you want to send or receive a string of bytes you must do so in a loop.

And let me be very clear on this point: the SX/B compiler does not excuse the programmer from understanding the architecture and behavior of the SX micro. Don't let that worry you though, if you've been around BASIC Stamps or other micros for any length of time the SX will not be hard to pick up.

Follow the Line

Okay, I could fill the magazine with theoretical chat, but that's not our style, it is? Let's get right to our project, a serial line follower module that you can use with your BASIC Stamp-powered robots. And before I forget, let me give credit where it's due: the sensor design I'm presenting here isn't mine. It was designed by a very nice guy named James Vroman. I met James when he was living in Dallas and actively involved in the Dallas Personal Robotics Group (www.dprg.org). One his robots, JavaBot (java as in coffee can), uses an array of sensors to follow a line. The same array is used on the Parallax line follower module, and in our project here.

Figure 116.1 shows the schematic for the line sensor array. Each element is composed of a QRB1114 reflective object sensor. Each sensor holds an IR LED and an IR phototransistor. When the LED is activated and the light reflected from a nearby surface, the current flowing through the transistor is affected -- the greater the IR reflection, the greater the current flow.

Notice that the collector of each transistor is connected to a 10K pull-up, and that junction is labeled CMP. What happens is that the 10K resistor and the (activated) transistor form a voltage divider, with the output of that divider fed into a comparator circuit. By using a variable voltage into the other side of the comparator we can set the reflection threshold for the sensor array. This allows us to "tune" the circuit for ambient light.

Okay, to activate and monitor the sensor array we will use, of course, an SX micro; in this project we'll use the SX28 (note that SX/B will work with the SX18, SX20, and SX28). Using the SX28 gives us plenty of IO pins so that we can have a large sensor array. The large array lets us follow wide lines, or have greater resolution and control when following thin lines.

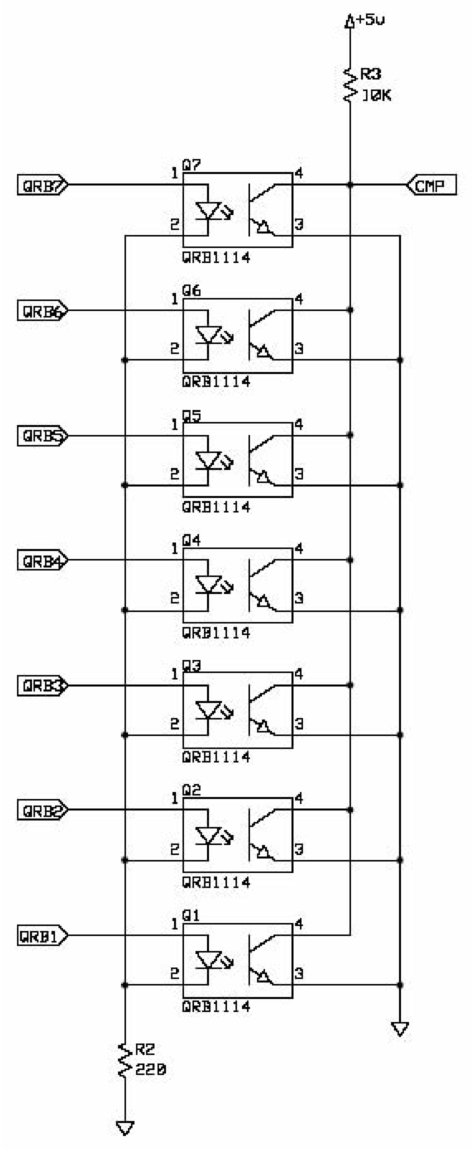


Figure 116.1: QRB1114 Sensor Array Schematic

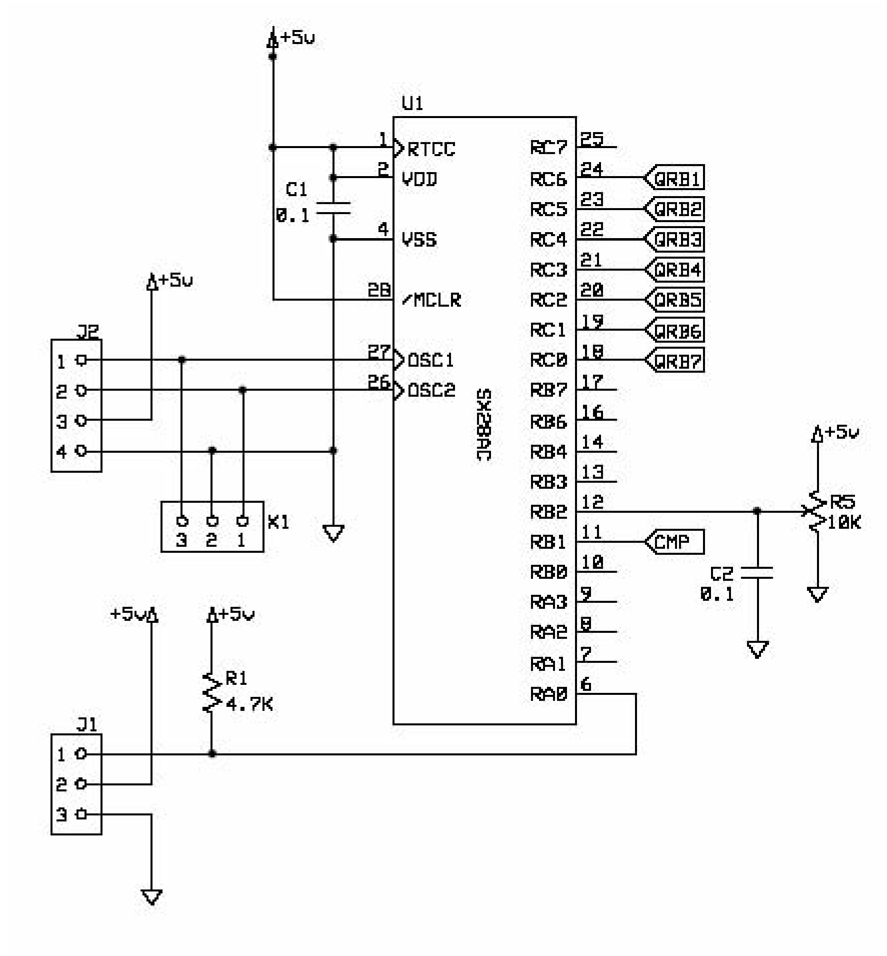


Figure 116.2: SX28 Line Follower Controller Schematic

Figure 116.2 shows the schematic for our SX28 controller circuit. As you can see, it's actually quite simple. Pin RA.0 is our serial IO connection. It is pulled-up through 4.7K to make it compatible with the Parallax AppMod serial protocol which allows for two-way communications over the same line. The pins on port RC will be configured as outputs to activate the sensor array. Finally, pins RB.1 and RB.2 are used to monitor and analyze the output of the sensor array. You see, the SX has a built-in comparator that can be enabled with code. The inputs of the comparator happen to be RB.1 and RB.2.

If you do decide to build the board (complete schematic and board layout is part of this month's download), make sure that you do indeed use a socket for X1. This is the socket for the ceramic resonator. If you remove the resonator then you can run the program through the SX-Key in single step mode for debugging. You cannot debug SX code with the resonator in place. And by using a socket, you can select whatever frequency you choose. Just be aware that my simple design borrows power from an external source (i.e., the Boe-Bot) and the faster the SX runs the more current it will consume.

The Code that Follows the Line

Before we actually jump into the code, let's talk requirements. First, we want the module to be compatible with the Parallax AppMod serial protocol. What this means is an open-baudmode, bidirectional serial link. If you're a little fuzzy on the concept of "open" baud modes, let me try to clear things up. When using an open mode, the BASIC Stamp will drive the serial output pin one way or the other (depending on mode, true of inverted), and rely on a pull-up or pull-down to take care of the other pin state. In our case, we're going to use true mode, which means the serial line rests at a high state, and an active bit is a low. What we have to do, then, is pull the line to Vdd through a resistor. When a "1" bit is transmitted the line will be pulled low by the SX or the BASIC Stamp. For a "0" bit the serial output pin is made Hi-Z (input state) and the pull-up takes care of the rest.

Why go through all this? Well, what this does is let us connect a bunch of devices to the same line and if more that one go active at the same time there is no danger of a short. If, for example, the BASIC Stamp and the SX both pull the line low at the same time there will be a data collision for sure, but no short circuit since they are at the same state. But if one pulled the line low while the other was trying to drive the line high ... we could end up with blue smoke.

Now that the protocol is settled, let's talk features. How about querying the device for a firmware number? I think that's a good idea, especially if we create a product for sell; we can allow the user to identify the firmware version of that device. Since we're building a line follower, all that's left is to return the line bits. Just to simplify things for the BASIC Stamp, let's create two line functions: one that returns the bits when the line is white on a black field, the other that returns the bits when the line is black on a white field. Both functions will return a "1" when the sensor element has detected the line.

Time to jump in. Within the SX/B program there must exist a directive called PROGRAM that tells the code where to begin after the start-up code (IO and variables initialization) is executed. If an ISR (interrupt service routine) is declared, the directive must come after that. In our case, the directive is PROGRAM Start, so the code will begin its execution at the label Start.

You're probably wondering why the subroutines are placed ahead of the main code. This has to do with the SX architecture. The entry point of a subroutine must be in the first half (256 words) of the code page where it lives. The SX28 has four code pages of 512 words (2K total). This program is small and everything fits into page 0. For larger programs we can create a "jump table" and move subroutine code to another page – but that's beyond the scope of this article. The SX/B help file gives plenty of good examples to show how to handle larger programs.

Back to Start. The first instruction you'll see is ADDRESS $100. The ADDRESS directive forces the code to be placed at a certain location. By forcing the start code to this address, the assembler will complain if our subroutines run too long. This wasn't the case here, but is still a good idea to use if you're going to put your subroutines on page 0.

The next few lines setup the IO structure. On the SX28 we have three ports (RA, RB, and RC) for a total of 20 IO pins (RA has only four pins). On ports A and B we have a bunch of unused pins, and it's not a good idea to leave these floating. What we can do, then, is activate the weak pull-ups on the unused pins to pull then into a known state. What you have probably noticed is that a 0 in the respective bit activates the pull-up. When using the TRIS registers (equivalent to the BASIC Stamp DIRS), a 0 causes the respective pin to be an output – this is just the opposite of how we program the BASIC Stamp and we must remain mindful of this fact when programming in SX/B.

In short, our setup section sets makes all unused pins inputs and enables the pull-ups on those pins. The reason that RB.0 is made an output when it is not connected to anything is that it will hold the state of the comparator, so the comparator result will make it high or low and allow us to read that bit as part of our line scan.

With the I/O pins initialized it's time to start – start waiting, that is. This device is a serial slave and speaks only when spoken too. The front end of the code (that begins at Main) waits for the proper message header. In our case that header will be "!LF" (for line follower). Here's the code that waits for the message header:

```
Main:
  GOSUB RX_Byte, @char
  IF char <> "!" THEN Main
  GOSUB RX_Byte, @char
  IF char <> "L" THEN Main
  GOSUB RX_Byte, @char
  IF char <> "F" THEN Main
```

The "!" character is a legacy thing from the AppMod serial protocol. In some AppMods, the "!" is used to set detect and set the baud rate. We can't do that (auto-baud detect) without resorting to assembly language, so we've fixed our baud rate to 9600. This is about as fast as we can go when using a 4 MHz clock.

As I stated earlier, many of the functions in SX/B work differently than their BASIC Stamp counterparts. SERIN, for example, will wait for a byte – one byte, and we have to do our own filtering. And also remember that each time we have SERIN in our program it gets translated into a set of assembly instructions. If we have a bunch of SERINs, we could quickly chew up our programming space. To save space, we can put SERIN into a subroutine, and even give it the ability to work with parameters. Here's that routine:

```
RX_Byte:
  rtnAddr = __PARAM1
  SERIN Sio, Baud, temp1
  PUT rtnAddr, temp1
  RETURN
```

On entering the subroutine we make a copy of an internal SX/B variable called __PARAM1. This variable holds the first parameter sent to the subroutine. Now go back and look at the code at Main. Notice how we call the subroutine with a variable, and in front of that variable is the "@" character. What the "@" character does is tell the compiler to send the address of the variable instead of its value. This is very powerful, because it lets create a subroutine that can affect any variable we send to it using this technique.

Back in our RX_Byte subroutine we then use SERIN to receive the byte and return it to the variable that was passed by using the PUT function. PUT takes a location and value as arguments; in this case we are putting the variable received by SERIN into the address that was passed to the subroutine.

Once we've received a byte we compare it to the list of valid characters in our header string. If there is ever a mismatch, the code is forced back to the beginning. Once we have received "!", "S", and "L" in that order we will wait for one more byte that will be the command to process.

For our line follower there are three valid command bytes: "V" (return version), "B" (return black line bits), and "W" (return white line bits). Let's start with "V." In fact, let's show how to get the BASIC Stamp to request and wait for the version code from our line follower:

```
  SEROUT Sio, Baud, ["!LFV"]
  SERIN Sio, Baud, [STR version\3]
```

Pretty simple, right? It is – but there's something we need to be aware of when designing serial accessories for the BASIC Stamp. Even though the Stamp does a SERIN right after the SEROUT, it is still much slower than the SX. What this means is that the SX has to allow time for the BASIC Stamp to get ready before sending any information back. You'll see that in just a second.

```
Check_V:
  IF char <> "V" THEN Check_B
  GOSUB Delay, 250, 4
  idx = 0

Next_Char:
  READ Rev_Code + idx, char
  INC idx
  IF char = 0 THEN Main
  GOSUB TX_Byte, char
  GOTO Next_Char
```

When we do receive a "V" command code the IF-THEN line will fail and the program will drop through to a call to another subroutine named Delay. This is another case where program space is conserved by placing high-level (lots of assembly code) functions into subroutines so that the high-level functions are compiled in one place. Let's have a look at the Delay subroutine:

```
Delay:
  temp1 = __PARAM1
  temp2 = __PARAM2
  PAUSEUS temp1 * temp2
  RETURN
```

This subroutine is expecting two parameters; a delay value and a multiplier. You may be wondering why we have to save __PARAM1 and __PARAM2. The reason is that these variables will be used when the PAUSEUS (pause in microseconds) subroutine gets compiled; so if we didn't save the parameters that get passed they would ultimately be clobbered.

Since we don't need particularly long delays in this program, and there is a time when very short delays are needed, we're using PAUSEUS. We're also using a syntax variation that lets the ultimate delay be the product of the two values passed to it. With this syntax we could create delays up to 65 milliseconds. By passing 250 and 4 to the Delay subroutine we create a delay of one millisecond; this is plenty of time for the BASIC Stamp to load its SERIN routine and be ready for what we send back to it.

Now that the BASIC Stamp is ready, we can transmit the three-character version string. The string itself is stored in a DATA table, very much like we would do with a BASIC Stamp. The difference between the SX and the BASIC Stamp is that SX tables are read-only; we cannot rewrite them at run time.

The transmission code is pretty easy: it grabs a character from the string (table) and if it's not zero, it sends it to the BASIC Stamp. Here's the transmit subroutine code:

```
TX_Byte:
  temp1 = __PARAM1
  SEROUT Sio, Baud, temp1
  RETURN
```

By now the structure should be fairly obvious: we make a copy of the parameter (variable) that gets passed and then send it out. In this case, though, we don't use "@" so what we end up passing is the value of the variable.

The last major step is reading the line sensor array. If the command byte is "B" or "W" we will read the sensor array with this code:

```
Get_Line:
  rtnAddr = __PARAM1                    ' save return address
  CMP_B = 0                             ' enable comparator
  temp1 = %00000000
  FOR idx = 0 TO 6
    Sensor = 1 << idx
    \ MOV __PARAM1, #250
    \ DJNZ __PARAM1, $
    temp1 = temp1 << 1
    temp1.0 = CmpOut
  NEXT idx
  Sensor = %00000000
  PUT rtnAddr, temp1
  CMP_B = $FF
  RETURN
```

As with the RX_Byte routine, we're going to pass an address parameter to Get_Line. When we're done that address (hence its variable) will hold the current line sensor value. On entering the routine we activate the SX comparator by clearing bits 7 and 6 of CMP_B (comparator setup register). By doing this, the comparator is enabled and its result output is routed to RB.0 (aliased as CmpOut).

The bulk of this routine is a loop that activates a single sensor in the array. After the sensor is activated there is a short delay to allow things to settle. One of the (many) neat things about the SX/B compiler is that we can insert assembly code when we want to. In this case we'll pop in two lines that will ultimately result in a 250 microsecond delay. Yes, we could have used our Delay subroutine, but why not stretch a little bit and have some fun? After the delay we will rotate our temporary line bits variable and then place the comparator output into bit zero of it. By going through this loop seven times we end up with a byte that holds the value of the line sensor array. Keep in mind that the value is going to be affected by the setting of the pot connect to RB.2. The pot is used to set the sensitivity of the sensor elements, so we're able to tune the circuit for ambient lighting.

When the loop is complete we finish by making sure that all the sensors are off, we move the scan result to the passed variable, then we shutdown the comparator to conserve as much power as possible.

The design of the sensor array and the comparator inputs will return a "1" bit when the sensor is over a highly reflective surface. Well, what happens when we have a black line on a white surface? It's actually pretty easy to deal with. Let's have a look:

```
Check_B:
  IF char <> "B" THEN Check_W
  GOSUB Delay, 250, 4
  GOSUB Get_Line, @lnBits
  lnBits = lnBits XOR %01111111
  GOSUB TX_Byte, lnBits
  GOTO Main
```

After retrieving the line sensor value we can invert the bits using XOR, then we send it off to the BASIC Stamp. Once the line value has been transmitted the code returns to the top of the program and waits for another command sequence.

For those that want to build this serial line follower project I've included the schematic and PCB layout in ExpressPCB (www.expresspcb.com) format. Please understand that using these files is at your own risk. I'm certainly no PCB designer – heck, I'm barely a programmer. Please check everything carefully before you make an order. I actually found an error in my first layout that I corrected with a bit of PCB surgery. That error has been removed from the project files that you can download. Figure 116.3 shows my prototype board, ready to mount on the bottom a Boe-Bot. What, no Boe-Bot? Well, RadioShack® carries them now so a robot may be available in your neighborhood as you read this. Figure 4 shows the output of the BASIC Stamp test program that can be used to calibrate the sensor pot.

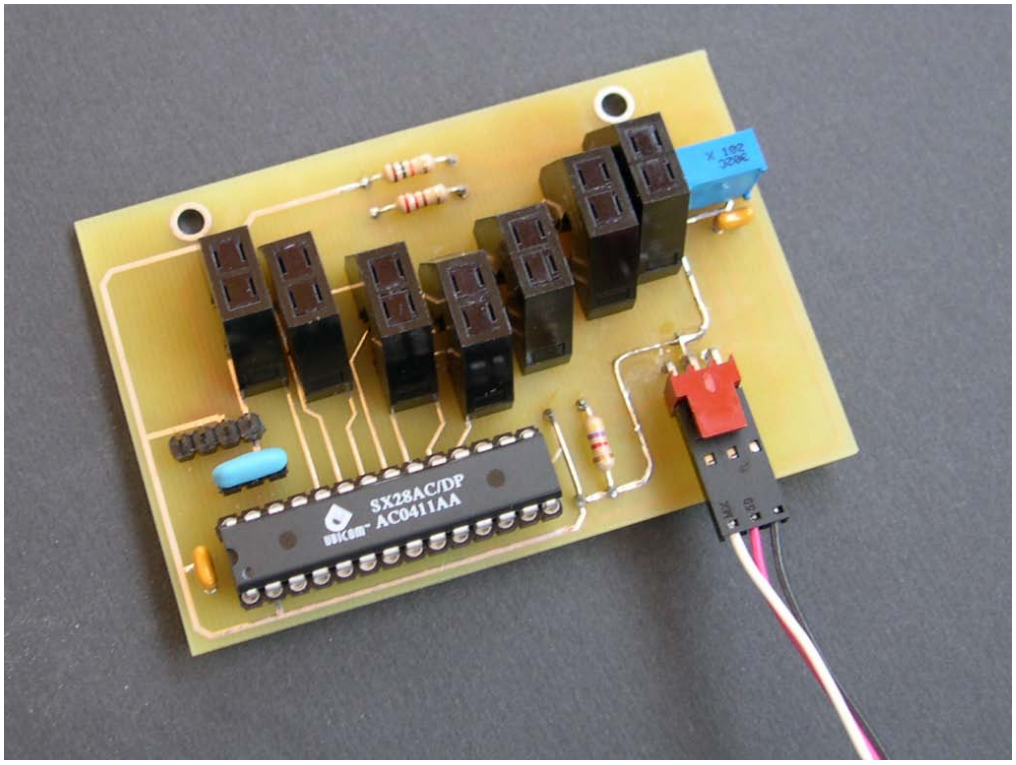
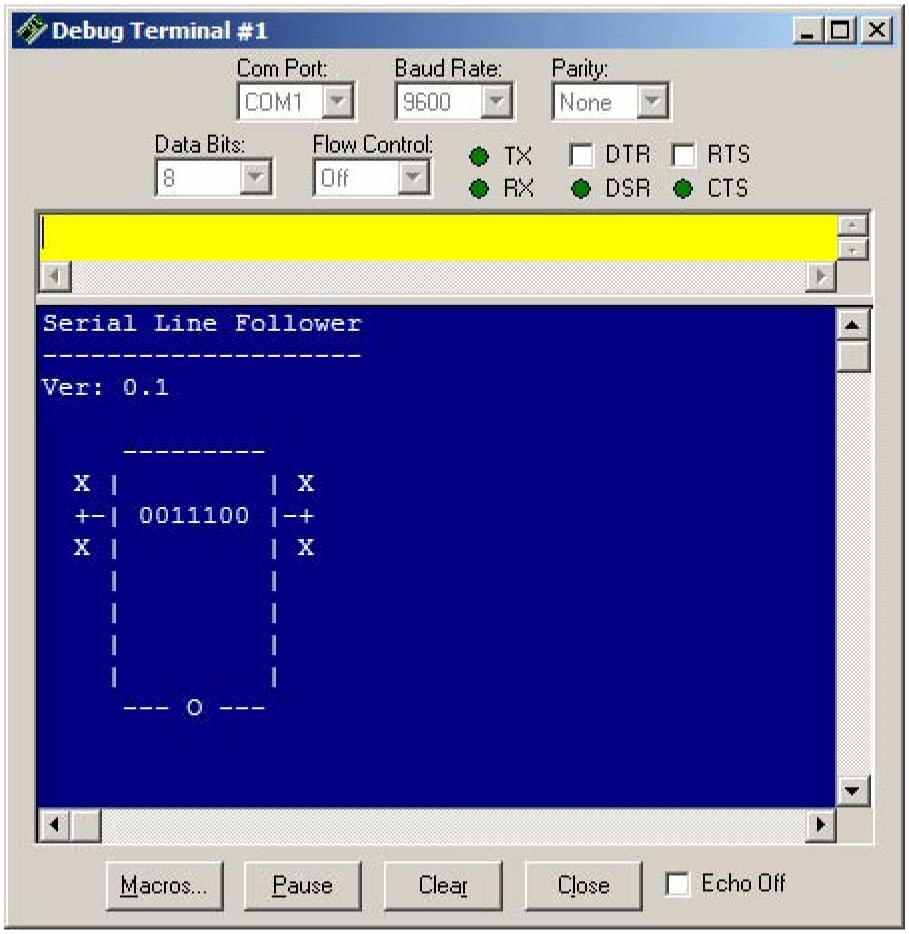


Figure 116.3: Line Follower PCB (board by ExpressPCB)



Com Port: COM1
Baud Rate: 9600
Parity: None
Data Bits: 8
Flow Control: Off
DTR
RTS

```
Serial Line Follower
--------------------
Ver: 0.1

     ---------
  X |         | X
  +-| 0011100 |-+
  X |         | X
    |         |
    |         |
    |         |
    |         |
     --- O ---
```

Figure 116.4: Output from BASIC Stamp test program

There's More ... A Lot More

Wow, I'm out of breath – and out of space. Let me assure you that SX/B is a lot of fun to play with and with a bit of patience and study, you'll be as confident at it as you are with the BASIC Stamp. You're probably wondering how you get the SX/B compiler. The answer is as simple as downloading the SX-Key software from Parallax; SX/B is built right in. Of course, you'll need an SX-Key to program the SX chips, and some sort of programming board. Parallax has a small development board called the SX Tech Board, and if you're feeling really industrious you could even build your own.

Parallax has a couple great books on the SX, and more coming. You can download Al Williams' book, Exploring the SX Microcontroller with Assembly and BASIC Programming, and Günther Daubach's book, Programming the SX Microcontroller – A Complete Guide is available in a bound volume. There is a version of the SX-Key starter kit that includes both these books, the SX-Key programming tool, and the SX Tech Board – it's a great way to get started with the SX. As for the Internet, James Newton's SX List (www.sxlist.com) is full of useful information and tips on programming the SX micro. Check it out.

As I close, please accept my sincere wishes for a happy and peaceful holiday season. And until next year (which is just a month away!), Happy Stamping.

```
' =========================================================================
'
'   File...... SERIAL_LF.SXB
'   Purpose... Serial Line Follower module for robots
'   Author.... Jon Williams -- Parallax, Inc.
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 10 OCT 2004
'
' =========================================================================


' -------------------------------------------------------------------------
' Program Description
' -------------------------------------------------------------------------
'
' Serial Line Follower module for BOE-Bots
'
' The module uses an open baudmode serial connection and the Parallax
' AppMod protocol structure.
'
' Valid serial commands from host:
'
' "!LFV" -- returns 3-byte version string (e.g., "0.1")
' "!LFB" -- returns sensor bits, black line on white field
' "!LFW" -- returns sensor bits, while line on black field


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

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


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

Sio             VAR     RA.0                    ' serial connection
Sensor          VAR     RC                      ' sensor control pins
CmpOut          VAR     RB.0                    ' comparitor output


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

Baud            CON     "OT9600"                ' open, true, 9600 baud
```

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

char            VAR     Byte                    ' serial char in and out
lnBits          VAR     Byte                    ' line sensor bits
idx             VAR     Byte                    ' loop counter

rtnAddr         VAR     Byte                    ' return address parameter
temp1           VAR     Byte
temp2           VAR     Byte


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

Rev_Code:
  DATA  "0.1", 0


' -------------------------------------------------------------------------
' Subroutines
' -------------------------------------------------------------------------

' Wait for and receive a byte from the serial connection
' -- Use: GOSUB RX_Byte, @theByte
' -- serial input byte is placed in 'theByte'

RX_Byte:
  rtnAddr = __PARAM1                            ' save return address
  SERIN Sio, Baud, temp1                        ' get serial byte
  PUT rtnAddr, temp1                            ' put in return address
  RETURN


' Transmit a byte through the serial connection
' -- Use: GOSUB TX_Byte, theByte
' -- transmits 'theByte'

TX_Byte:
  temp1 = __PARAM1                              ' save serial byte
  SEROUT Sio, Baud, temp1                       ' transmit it
  RETURN


' Insert program delay
' -- Use: GOSUB Delay, delayVal, multiplier
' -- holds program 'delayVal' * 'multiplier' microseconds
```

```
Delay:
  temp1 = __PARAM1                             ' grab parameters
  temp2 = __PARAM2
  PAUSEUS temp1 * temp2                        ' do the delay
  RETURN


' Reads the line sensor bits
' -- Use: GOSUB Get_Line, @theByte
' -- line bits are placed in 'theByte'

Get_Line:
  rtnAddr = __PARAM1                           ' save return address
  CMP_B = 0                                    ' enable comparator
  temp1 = %00000000                            ' clear line bits
  FOR idx = 0 TO 6                             ' loop through 7 sensors
    Sensor = 1 << idx                          ' activate element
    \ MOV __PARAM1, #250                       ' 250 us delay @ 4 MHz
    \ DJNZ __PARAM1, $
    temp1 = temp1 << 1                         ' setup output var
    temp1.0 = CmpOut                           ' get new bit
  NEXT idx
  Sensor = %00000000                           ' deactivate all elements
  PUT rtnAddr, temp1                           ' retun value to sender
  CMP_B = $FF                                  ' shutdown comparitor
  RETURN



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

Start:
  ADDRESS $100                                 ' prevent subs overrun
  PLP_A = %0001                                ' activate pull-ups, 1 - 3
  PLP_B = %00000111                            ' activate pull-ups, 3 - 7
  TRIS_B = %11111110                           ' RB.0 is an output
  Sensor = %00000000                           ' clear sensor
  TRIS_C = %00000000                           ' make pins outputs

Main:
  GOSUB RX_Byte, @char
  IF char <> "!" THEN Main                     ' wait for "!"
  GOSUB RX_Byte, @char
  IF char <> "L" THEN Main                     ' wait for "L"
  GOSUB RX_Byte, @char
  IF char <> "F" THEN Main                     ' wait for "F"

Get_Command:
  GOSUB RX_Byte, @char                         ' wait on a character

Check_V:                                       ' version
  IF char <> "V" THEN Check_B
  GOSUB Delay, 250, 4                          ' let BASIC Stamp get ready
  idx = 0                                      ' reset index

Next_Char:
  READ Rev_Code + idx, char                    ' read character
  INC idx                                      ' update index
  IF char = 0 THEN Main                               ' if 0, we're done
  GOSUB TX_Byte, char                          ' transmit the character
  GOTO Next_Char                               ' repeat until done


Check_B:                                       ' black line
  IF char <> "B" THEN Check_W
  GOSUB Delay, 250, 4                          ' let BASIC Stamp get ready
  GOSUB Get_Line, @lnBits                      ' get line bits
  lnBits = lnBits XOR %01111111                ' make line bit = 1
  GOSUB TX_Byte, lnBits                        ' send line bits
  GOTO Main

Check_W:                                       ' white line
  IF char <> "W" THEN Check_X
  GOSUB Delay, 250, 4                          ' let BASIC Stamp get ready
  GOSUB Get_Line, @lnBits                      ' get line bits
  GOSUB TX_Byte, lnBits                        ' send line bits
  GOTO Main

Check_X:
  '
  ' for future expansion
  '
  GOTO Main
```

```
' =========================================================================
'
'   File...... SLF_TEST.BS2
'   Purpose... Serial Line Follower Test
'   Author.... Jon Williams -- Parallax, Inc.
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 10 OCT 2004
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =========================================================================


' -----[ Program Description ]---------------------------------------------
'
' Simple test program for the Serial Line Follower module


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


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

Sio             PIN     15                      ' use BOE servo connector


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

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

Inverted        CON     $4000
Open            CON     $8000

Baud            CON     Open | T9600            ' open for AppMod protocol
```

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

version         VAR     Byte(3)
response        VAR     Byte


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


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

Reset:
  PAUSE 100
  DEBUG CLS,
        "Serial Line Follower", CR,
        "--------------------", CR,
        "Ver: "

Get_Version:
  SEROUT Sio, Baud, ["!LFV"]                    ' request version
  SERIN Sio, Baud, 1000, Reset, [STR version\3] ' receive version string
  DEBUG STR version


Robot_Screen:
  DEBUG CRSRXY, 0, 4,
        "     ---------    ", CR,
        "  X |         | X ", CR,
        "  +-| xxxxxxx |-+ ", CR,
        "  X |         | X ", CR,
        "    |         |   ", CR,
        "    |         |   ", CR,
        "    |         |   ", CR,
        "    |         |   ", CR,
        "     ---( )---    ", CR


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

Main:
  SEROUT Sio, Baud, ["!LFW"]                    ' request white line bits
  SERIN Sio, Baud, 1000, Main, [response]       ' wait for response
  DEBUG CRSRXY, 6, 6, BIN7 response             ' display on screen
  PAUSE 200
  GOTO Main

  END

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