Column #104 December 2003 by Jon Williams:

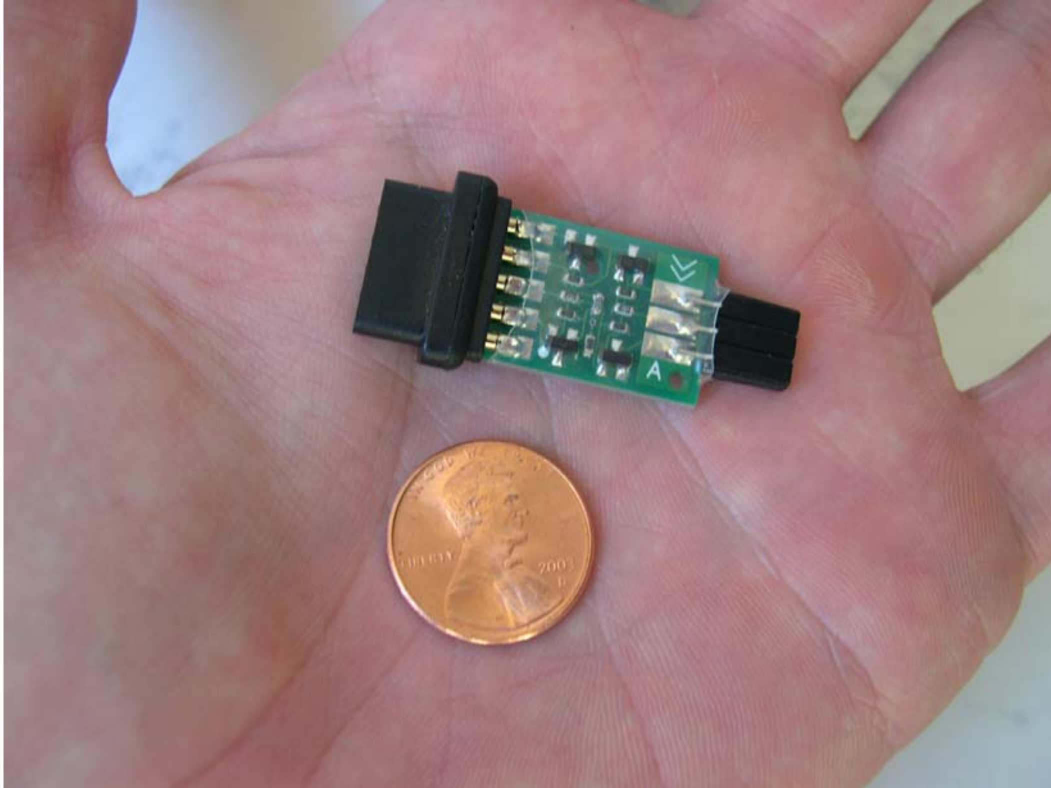
The BS1 is Back, Baby!

Did I ever tell you about my first BASIC Stamp? No? Okay, then, the story goes like this: I was scheduled for surgery on my skull (not my brain as many have claimed would have been a better use of the doctors' time...) and decided to take a couple weeks off work while I recovered. The thing is, I'm a fidgety guy and don't like sitting still for very long, so I decided to find something to occupy myself with during the recovery.

I had been seeing the advertisements for the BASIC Stamp on the back of Nuts & Volts for a few months, but kept thinking to myself, "It's too good to be true...." In the end, though, I decided that I could spend a hundred dollars on the BS1 Starter Kit – if it wasn't any good then, oh well, I tossed away a hundred bucks.

A few days later the box arrives from Parallax. I open it up, check it out, hook it up, and start playing. It was about six or seven in the evening when I started. I didn't get to bed until 5:00 AM the next day (I called in sick). That was almost 10 years ago and I think it is safe to say that I've worked with or played with BASIC Stamps every day since.

Figure 104.1: BS1 Serial Windows Programming Adapter



On my trip out to the California office in October I was handed a gift; a small gift, but one with big promise: the BS1 serial port adapter. I connected it to my shiny new laptop computer (running XP Pro), started up the new editor and download a program to an old BS1 Rev. D module – the very same module that I had purchased and worked with all those years ago. Dare I say the same goofy smile I had way back when crossed my face that day in the office.

Parallel? Serial? Huh?...

Okay, I know what a few of you Stamp old-timers are thinking... "Wait a minute there, buddy, the BS1 programs through the parallel port – so what's the story with a SERIAL adapter?" The truth is the BS1 actually programs serially at 4800 baud. What happens is that the DOS BS1 editor bit-bangs serial data through a couple of lines on the parallel port. Why? Because the parallel port works at TTL (5v) levels so that simplifies the design of the module. If you look at the BS1 schematic you'll find there is a direct connection between the programming port and the PIC interpreter.

The serial adapter takes care of the RS-232 to TTL level conversion required by the BS1. But here's the trick: laptop computers usually don't come anywhere near the RS-232 limits, in fact the frequently fall very short (though within the specification). The engineering staff at Parallax came up with a very simple circuit that will handle swings through the entire RS-232 range, making the adapter work with virtually any PC serial port. Yeah baby, the BS1 is back!

The photo in Figure 104.1 shows the adapter – literally putting BS1 programming through Windows in the palm of your hand. If you're adventurous and want to build your own, feel free to download the schematic from the Parallax web site.

What Can You Do With Eight Pins?

Way back in the early days – when I was really getting into the BS1 – I can remember one of my technical buddies asking me what I could do with an eight-pin microcontroller (technically, the BS1 has more than eight pins, just eight IO). In his mind, it just wasn't enough. Interesting how things change, isn't it? It seems like every major microcontroller manufacturer in the world is offers an eight-pin micro – and two of those pins get used for the power supply!

And to answer the question at the head of this section, we can do quite a lot with just eight pins. Also remember that with a bit of code and some standard IO chips like the 74HC595 and 74HC165, we can create as much IO as we need.

What we're going to do this month is create a little serial slave device using a BS1. Now to be perfectly honest, this isn't an especially practical project. I've never claimed that everything I present in my column is practical – my purpose is teaching and inspiring.

Because of the BS1's limited IO structure, serial devices became very popular. I believe it's fair to say that Scott Edwards started the serial device (for micros) movement with his "Stamp Stretcher." In my book he started a whole cottage industry (and is still a leader in it). So, as a tribute to Scott, we'll create a serial display – albeit a very simple one. Our project will be a serial seven-segment digit. Okay, okay, after you pick yourself up and stop laughing, give it a read through. I told you this wasn't going to be terribly practical; the goal is to show you how to make your own serial accessory device.

Device on a Wire

One of the first things I noticed about those big ads that ran on the back of Nuts & Volts is that the BS1 could do serial input or output on any pin. This is a very cool feature for talking to other Stamps, or even a PC. Better yet, the BS1 could do different serial baud rates and modes. It's the latter part we're interested in here, the various modes.

Of particular interest are the "open" baud modes. When using one of these modes, the Stamp only drives the serial line in one direction, either to ground in True mode, to Vdd in Inverted mode. The other state of the serial line is controlled by a pull-up or pull-down resistor; the Stamp "opens" the pin [makes it an input] to let the resistor set the state of the serial line.

This comes into play when we want to drop more than one slave device on the serial line and expect that they can talk back to the master. Since the master and slaves can only drive the serial line in one direction (the same direction for all), there is no possibility of an electrical conflict. Of course, this is not new or strange to most of you. This strategy is used on the Dallas/Maxim 1-Wire® bus (a specialized serial bus) and the Philips I2C™ bus (a synchronous serial bus).

So that our project is compatible with Parallax AppMod accessory modules, we will use an open-drain setup: the serial bus will be pulled high through a 4.7k resistor. The master and slaves will pull the bus line low for a "1" bit. Figure 104.2 shows the schematic for our BS1 slave.

BS1 In Review

Before we smash through the code, let's look back at the BS1 since it differs quite a lot from the Stamp 2; especially lately with the release of PBASIC 2.5.

- The BS1 has only 14 total bytes of variable space

- The BS1 uses three variable types: Bit, Byte, and Word
- Bit-level access is only allowed on variables B0 and B1 (W0)
- The programmer assigns variable location by aliasing internal variables
- PBASIC 1.0 is very terse; not as flexible or feature-rich as PBASIC 2.5

Now, don't let that last point throw you – the BS1 is a very capable little beast (it started a microcontroller revolution after all) and with a bit of practice and planning, you will be surprised at what you can pull off. The key is to let go of any smug suppositions of what BASIC should be. Remember that there isn't a lot of resources within the BS1, so the language is deliberately lean, and in many cases the high-level part of the language is very close to the machine code.

The biggest bit of grief that people have is the lack of `IF-THEN-ELSE`. I have news for you: `IF-THEN-ELSE` doesn't exist in assembly language [without macros] either. Most microcontrollers have compare-and-branch type instructions. The PBASIC 1.0 `IF-THEN` uses this structure, so the code after `THEN` is an implied `GOTO` followed by a program label [address]. If you're new to Stamps it will take just a bit of time to get used to the idea, but trust me, the lack of `IF-THEN-ELSE` hasn't stopped tens of thousands of hobbyists and engineers from succeeding with the BS1.

Alright, let's get into it. Since we haven't dealt with the BS1 in a very long time and the code for our slave seven-segment display is very simple, we'll go through it all, even the declarations.

```
SYMBOL Sio      = 7
SYMBOL Segs     = PINS

SYMBOL Baud     = OT2400
SYMBOL Off      = %00000000
SYMBOL MaxDig   = $F

SYMBOL cmd      = B2
SYMBOL value    = B3
```

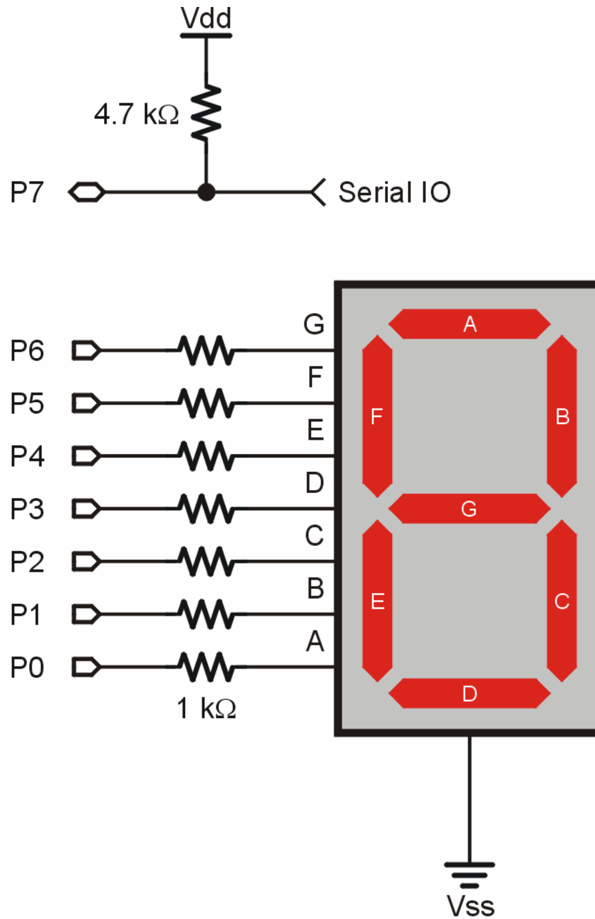


Figure 104.2: BS1 Slave Schematic

Notice that everything in PBASIC 1.0 is a SYMBOL; be it a constant or a variable, PBASIC 1.0 calls it a SYMBOL. The compiler will actually reconcile some of these things for us. PINS, for example, is the internal name for the IO structure. Reading from PINS is the equivalent of reading INL on the BS2. Conversely, writing to PINS is the equivalent of writing to OUTL on the BS2. The BS1 also has internal names for each of the individual pins (PIN0 – PIN7). Some commands in PBASIC 1.0 require constant values for pin numbers and the compiler doesn't reconcile them. We're going to use `SERIN` which requires a constant value, so the SYMBOL for Sio is 7; P7 is where our serial connection will be.

Since the BS1 has a fixed number of possible baud settings, they have internal names. For this program we're going to use OT2400, for Open-True-2400 baud. Strictly speaking, we don't have to define an open mode for a receiver (because `SERIN` makes the pin an input), but we're going to have a command that causes our slave module to send data back to the master. Remember that when using an open baud mode, the serial line needs to be pulled up (for True) or pulled down (for Inverted).

Okay, let's talk about variables. If you've only ever used a BS2, you're accustomed to the compiler automatically assigning variables in the Stamp's RAM by type. The BS1 compiler doesn’t do this. The way that we give variables useful names (aliasing) is to assign a name to a given internal variable. For bytes we can have B0 through B13. Before I go on, look at the listing and note how the first variable assigned is B2. The reason for this is habit, mostly, but a good one. You see, the only variables that allow bit-level access are B0 and B1 (bits can be addressed as Bit0 – Bit15), so reserving these bytes for bits that we might want to add later is a good idea. Finally, don't get lazy and use the internal variable names. Now read that last sentence again. You'll get away with this for a little while, but ultimately the programming demons will catch up to you and you will end up with misbehaving program because of an assignment error. Take a few minutes to give your variables meaningful names. Trust me, it's worth it. Let's move on.

To create our own EEPROM-base tables we'll use the EEPROM statement (similar to the DATA statement in PBASIC 2.x). Here's short table that holds the segment maps for decimal digits:

```
Digits:  ' .gfedcba
  EEPROM (%00111111)
  EEPROM (%00000110)
  EEPROM (%01011011)
  EEPROM (%01001111)
  EEPROM (%01100110)
  EEPROM (%01101101)
  EEPROM (%01111101)
  EEPROM (%00000111)
  EEPROM (%01111111)
  EEPROM (%01100111)
```

As you can see, it's about the same as using a DATA statement on the BS2. Just be aware that we need to track the starting addresses of multiple EEPROM tables manually, we can't use a program label as an address constant. This is usually not a problem since we're dealing with such a small micro and multiple tables are a rarity.

Okay, the definitions are done, it's time to initialize and start the program. We generally start with the IO pins. In this case, we need to make the segment-driving pins outputs. Simple: a "1" in the associated bit of the DIRS register makes the pin an output.

```
Setup:
  Segs = Off
  DIRS = %01111111
```

No sweat there, right? Now we can get into the main program loop. The first thing we're going to do is wait for a serial command. For our slave, we want to wait on a specific header so that we know that the master is talking to us. When that header arrives, we'll save the next two bytes that follow. The first byte will be our command, the second will be a value for that command.

```
Main:
  SERIN Sio, Baud, ("!SS0"), cmd, value
```

In the BS1, the header string to wait on is placed in parenthesis ahead of the input data variables. For this module the header is "`!SS0`". If we want to add more than one of these slaves to the same serial connect we simply change the unit digit in the header.

Another strategy would be to accept the ID byte as a variable after the header. By doing this we could check to see if a command was specified for us, or was a global command that all slave units act on. In that case, the serial input might look something like this:

```
  SERIN Sio, Baud, ("!SS"), id, cmd, value
```

If you decide to adopt the latter method, just be sure that slaves don't send anything out the serial line as a result of a global command. If they do the master won't understand anyone as they'll all try to talk at once. For the time being, let's keep things simple and keep the ID byte in the header.

Once we get a valid header and the command and value bytes we can act on them. Now you'll see the BS1 `IF-THEN` construct in action and the cleanest way to deal with it. The first valid command that we'll check for is "I" (identification).

```
Check_ID:
  IF cmd <> "I" THEN Check_Bits
  PAUSE 1
  SEROUT Sio, Baud, ("1.0")
  GOTO Main
```

The purpose of this routine is to allow the master to see that the slave is connected and working. Notice that if the command byte is not "I" we will skip to the next check, otherwise we will run the code for identification. Yes, this seems inverted, but it works and once you get used to it you won't have to think twice. In this case, we will PAUSE briefly so the master can setup to receive our data, then we'll send the identification string. This can be used by the master to see that we're available and even check what capabilities we have by looking at the ID data.

The next valid command is "B" (for bits). This command lets the master decide which bits (segments) to control on the display. The bit pattern is sent in the value parameter. As you'll see when we hook-up a BS2 master, this can be useful for simple animated indicators.

```
Check_Bits:
  IF cmd <> "B" THEN Check_Numeric
  Segs = value & %01111111
  GOTO Main
```

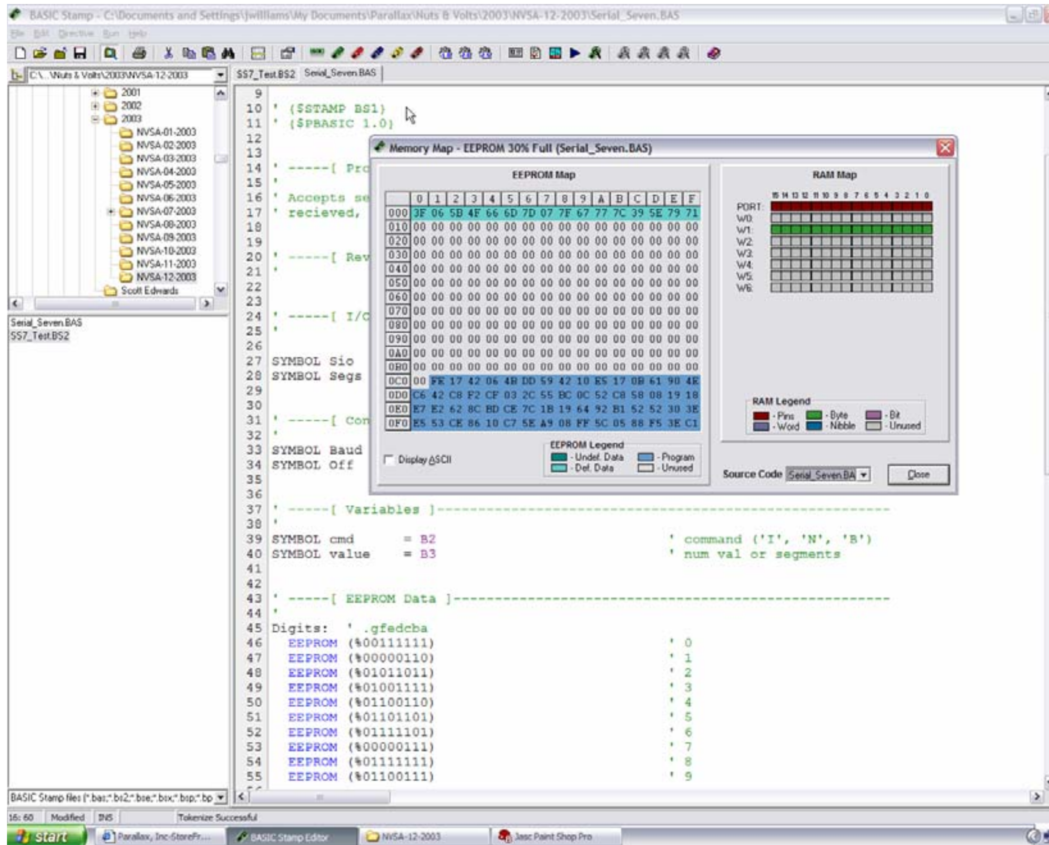
The final valid command is "N" for numeric. This command will cause the slave to place the digit passed in the value byte on the display, and works for all hexadecimal digits (0 – F). This section of code makes use of the EEPROM table that we defined earlier.

```
Check_Numeric:
  IF cmd <> "N" THEN Main
  IF value > MaxDig THEN Main
  READ value, Segs
  GOTO Main
```

As you can see, it also checks the range (defined by the MaxDig constant) so that we don't read invalid EEPROM locations and put garbage on the display.

Before we wrap up our slave and move on to the master demo, I'm sure a few of you advanced Stamp programmers are wondering why we did all the IF-THEN jumping around instead of using a LOOKDOWN table for the command. In fact, both work but it turns out that with small command sets the IF-THEN route actually uses less EEPROM space for the compiled program. If we were building a slave that was processing a lot of commands, I think the opposite would be true, and certainly more convenient. In case you're curious, here's what that would look like:

Figure 104.3: Programming a Stamp 1 in Windows




```
Check_Cmd:
  LOOKDOWN cmd, ("IBN"), cmd
  IF cmd > 2 THEN Main
  BRANCH cmd, (Show_ID, Show_Bits, Show_Dig)
```

If the command is valid the `LOOKDOWN` table will convert `cmd` to a value from zero to two which can be used by `BRANCH`. If not in the table, cmd will fall through unchanged and `IF-THEN` will send the program back to `Main`. Again, if we were processing a lot of possible commands, this would probably be a better way to go.

The reason I know it takes more memory is that with the BS1 integration in the Windows Stamp compiler, we finally have a memory map (yippee!!!). Figure 104.3 shows a screen shot from my system with the memory map of our slave program. Okay, let's test this dude. To test the slave seven-segment display we're going to connect a BS2 (any in the series) and run a simple program.

The first thing to do is check the ID as this lets us know that the slave is connected (remember, this is now BS2 code so the syntax is slightly different).

```
DEBUG CLS, "Serial Seven ID = "
SEROUT Sio, Baud, ["!SS0", "I", value]
SERIN Sio, Baud, 1000, No_Slave, [STR id\3]
DEBUG STR id\3
```

The code sends a message to the Debug Terminal then sends the "`I`" command to the slave. Notice that a value is also being sent. To keep the slave simple it has a fixed-format command structure so even when we don't need it, we will send the value parameter. In this case it will just get ignored so it doesn't matter what we send in value.

As soon as the command is sent we will wait on the input. If it doesn't arrive within a second the program will jump to `No_Slave`. In the demo this just tells us there was no response and loops back to try again. In other systems, we may need to take some corrective action or modify the program behavior if we don't have the slave device.

Let's say it's connected. The slave sends three ASCII characters as its ID so we can use the `STR` modifier to collect them in an array. And since they are ASCII, we can again use the `STR` modifier to spit them right back out onto the Debug Terminal. When you create slave modules that have variable capabilities you'll want to check the ID data and deal with it accordingly.

The next test is the "bits" mode where the master defines which LEDs on the seven-segment display are lit.

```
DEBUG CLS, "Bits Mode"
FOR idx1 = 1 TO 10
  FOR idx2 = 0 TO 5
    READ (Bug + idx2), value
    SEROUT Sio, Baud, ["!SS0", "B", value]
    PAUSE 50
  NEXT
NEXT
SEROUT Sio, Baud, ["!SS0", "B", 0]
```

This code uses two loops. The outer loop runs the inner loop ten times. The purpose of the inner loop is to animate the segments on the display. In this case the segments value is read from a DATA table and will be the outer LEDs on the display. When the demo runs, the LEDs will "chase" in a clockwise manner ten times. This is a great indication for waiting, or to show that a program is busy. Note that at the end of the loops we are sure to clear the display by using the bits command again with a segments value of zero.

The final test is the numeric mode where we pass a single-digit value to display:

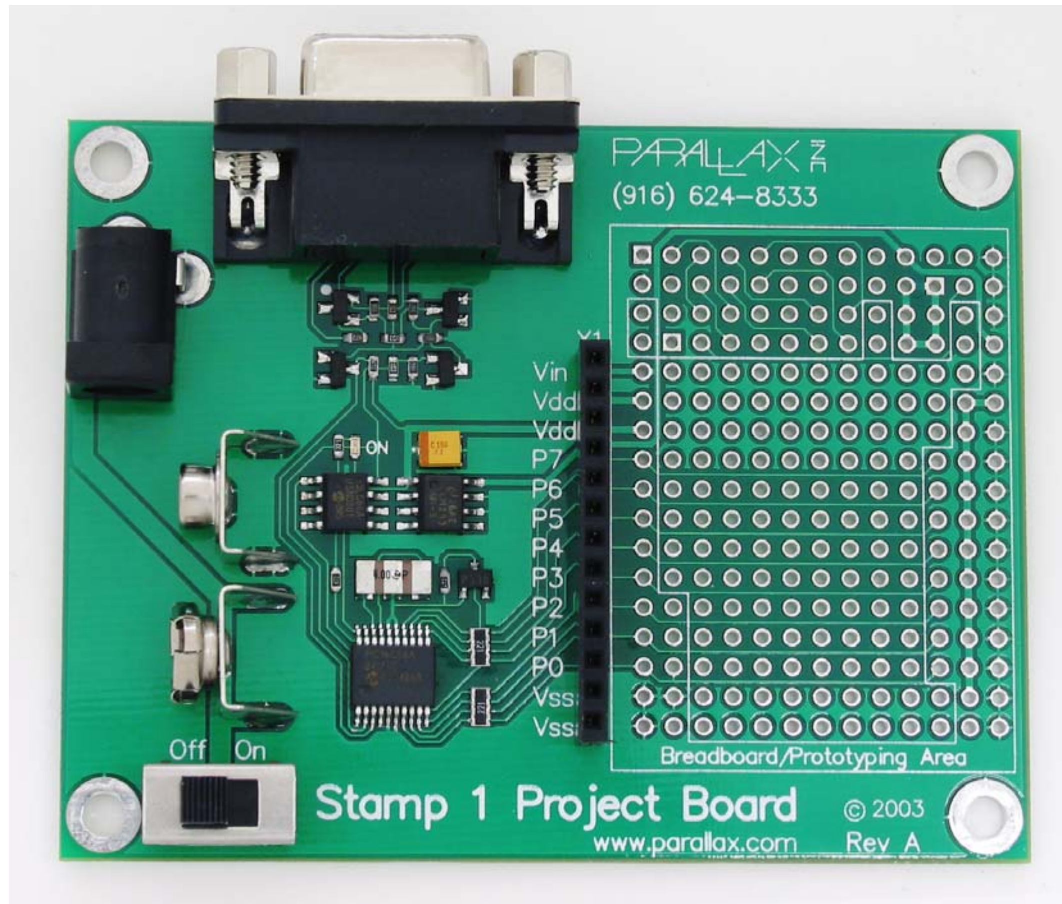
```
DEBUG CLS, "Numeric Mode"
FOR value = $0 TO $F
  SEROUT Sio, Baud, ["!SS0", "N", value]
  PAUSE 500
NEXT
SEROUT Sio, Baud, ["!SS0", "B", 0]
PAUSE 10
GOTO Main
```

There's no mystery here – this is easy stuff. I just want to make one note. Since the BS1 doesn't have a timeout facility on its `SERIN` function you have to make sure it's ready before you start sending information to it. Putting a small `PAUSE` after each command sent to the BS1 slave gives the slave plenty of time to receive, decode, and act on the command before getting ready for the next.

That's A Wrap

Well, I'd say that's about enough for this month and look at that, we've made it through another year! I hope that those of you that have BS1 modules will get them out and rediscover how much fun they are, and those of you that took advantage of the Parallax special pricing on the older model Rev. D modules really have reason to celebrate. The BS1 is a great training tool for kids of all ages. With the popularity of the BS2 HomeWork board, Parallax has created a similar product using the BS1. Figure 104.4 shows the new BS1 Project Board that has the built-in serial adapter and a power switch. You can add a solderless breadboard if you like, or take advantage of the trace layout and add connectors for servos, LCDs, and other accessories. And yes, it will mount on a Boe-Bot chassis so you can get into robotics very inexpensively.

Figure 104.4: BASIC Stamp 1 Project Board



In closing I'd like to say thanks again for all your kind notes and the exchanges that we've had this past year. Please keep them coming – I do my best to write this column for your needs so I love hearing from you. And I do hope that this holiday season brings joy and peace to you and yours. God bless you, peace be with you. Happy Stamping and have a very Happy New Year!

```
' =========================================================================
'
'   File....... SS7_Test.BS2
'   Purpose.... Test Serial Seven-Segment Display
'   Author..... Jon Williams
'   E-mail..... jwilliams@parallax.com
'   Started....
'   Updated.... 17 OCT 2003
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =========================================================================


' -----[ Program Description ]---------------------------------------------


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


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

Sio             PIN     0                       ' serial IO for SS7


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

#SELECT $STAMP                                  ' baud rate parameters
  #CASE BS2, BS2E, BS2PE
    T1200       CON     813
    T2400       CON     396
  #CASE BS2SX, BS2P
    T1200       CON     2063
    T2400       CON     1021
#ENDSELECT

Inverted        CON     $4000
Open            CON     $8000

Baud            CON     Open + T2400


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

id              VAR     Byte(3)                 ' array for ID string
idx1            VAR     Nib                     ' loop counter
idx2            VAR     Nib
value           VAR     Byte                    ' value to send to SS
```

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

Bug             DATA    %00000001, %00000010, %00000100
                DATA    %00001000, %00010000, %00100000


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

Setup:
  PAUSE 500                                     ' let slave power-up


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

Main:
  DEBUG CLS, "Serial Seven ID = "
  SEROUT Sio, Baud, ["!SS0", "I", value]        ' command for ID
  SERIN Sio, Baud, 1000, No_Slave, [STR id\3]   ' get ID string
  DEBUG STR id\3                                ' display it
  PAUSE 1000

  DEBUG CLS, "Bits Mode"
  FOR idx1 = 1 TO 10
    FOR idx2 = 0 TO 5
      READ (Bug + idx2), value                  ' get segs from EE table
      SEROUT Sio, Baud, ["!SS0", "B", value]   ' send them
      PAUSE 50
    NEXT
  NEXT
  SEROUT Sio, Baud, ["!SS0", "B", 0]            ' clear display
  PAUSE 1000

  DEBUG CLS, "Numeric Mode"
  FOR value = $0 TO $F
    SEROUT Sio, Baud, ["!SS0", "N", value]      ' send number 0 - F
    PAUSE 500
  NEXT
  SEROUT Sio, Baud, ["!SS0", "B", 0]            ' clear display
  PAUSE 10
  GOTO Main

  END


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

No_Slave:
  DEBUG "No Response."
  PAUSE 1000
  GOTO                                                                 Main
```

```
' -----[ Title ]-----------------------------------------------------------
'
' File......... Serial_Seven.BAS
' Purpose...... Serial controlled Seven-Segment Display
' Author....... Jon Williams
' E-mail....... jwilliams@parallax.com
' Started......
' Updated...... 16 OCT 2003

' {$STAMP BS1}
' {$PBASIC 1.0}


' -----[ Program Description ]---------------------------------------------
'
' Accepts serial input on PIN7 and, when appropriate header and command is
' received, output pattern to seven-segment display.


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


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

SYMBOL Sio      = 7                             ' serial input
SYMBOL Segs     = PINS                          ' segment outputs


' -----[ Constants ]-------------------------------------------------------
'
SYMBOL Baud     = OT2400                        ' open baud mode
SYMBOL Off      = %00000000                     ' all LEDs off
SYMBOL MaxDig   = $F                            ' maximum digit display


' -----[ Variables ]-------------------------------------------------------
'
SYMBOL cmd      = B2                            ' command ('I', 'N', 'B')
SYMBOL value    = B3                            ' num val or segments


' -----[ EEPROM Data ]-----------------------------------------------------
'
Digits:  ' .gfedcba
  EEPROM (%00111111)                            ' 0
  EEPROM (%00000110)                            ' 1
  EEPROM (%01011011)                            ' 2
  EEPROM (%01001111)                            ' 3
  EEPROM (%01100110)                            ' 4
```

```
  EEPROM (%01101101)                            ' 5
  EEPROM (%01111101)                            ' 6
  EEPROM (%00000111)                            ' 7
  EEPROM (%01111111)                            ' 8
  EEPROM (%01100111)                            ' 9

Hex_Digits:
  EEPROM (%01110111)                            ' A
  EEPROM (%01111100)                            ' b
  EEPROM (%00111001)                            ' C
  EEPROM (%01011110)                            ' d
  EEPROM (%01111001)                            ' E
  EEPROM (%01110001)                            ' F


' -----[ Initialization ]-------------------------------------------------
'
Setup:
  Segs = Off                                    ' leds off
  DIRS = %01111111                              ' segments = outputs


' -----[ Main Code ]------------------------------------------------------
'
Main:
  SERIN Sio, Baud, ("!SS0"), cmd, value         ' wait for input

Check_ID:
  IF cmd <> "I" THEN Check_Bits                 ' if not "I" check "B"
  PAUSE 1                                       ' let master get ready
  SEROUT Sio, Baud, ("1.0")                     ' send ID string
  GOTO Main

Check_Bits:
  IF cmd <> "B" THEN Check_Numeric              ' if not "B" check "N"
  Segs = value & %01111111                      ' display pattern
  GOTO Main

Check_Numeric:
  IF cmd <> "N" THEN Main                       ' if not "N" cmd is bad
  IF value > MaxDig THEN Main                   ' disallow bad value
  READ value, Segs                              ' read pattern from EEPROM
  GOTO Main

  END


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

```
' -----[ Title ]-----------------------------------------------------------
'
' File......... Serial_Seven.BAS
' Purpose...... Serial controlled Seven-Segment Display
' Author....... Jon Williams
' E-mail....... jwilliams@parallax.com
' Started......
' Updated...... 17 OCT 2003

' {$STAMP BS1}
' {$PBASIC 1.0}


' -----[ Program Description ]---------------------------------------------
'
' Accepts serial input on PIN7 and, when appropriate header and command is
' recieved, output pattern to seven-segment display.
'
' This version shows an alternative method for command parsing.


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


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

SYMBOL Sio      = 7                             ' serial input
SYMBOL Segs     = PINS                          ' segment outputs


' -----[ Constants ]-------------------------------------------------------
'
SYMBOL Baud     = OT2400                        ' open baud mode
SYMBOL Off      = %00000000                     ' all LEDs off
SYMBOL MaxDig   = $F                            ' maximum digit display


' -----[ Variables ]-------------------------------------------------------
'
SYMBOL cmd      = B2                            ' command ('I', 'N', 'B')
SYMBOL value    = B3                            ' num val or segments


' -----[ EEPROM Data ]-----------------------------------------------------
'
Digits:  ' .gfedcba
  EEPROM (%00111111)                            ' 0
  EEPROM (%00000110)                            ' 1
  EEPROM (%01011011)                            ' 2
```

```
  EEPROM (%01001111)                            ' 3
  EEPROM (%01100110)                            ' 4
  EEPROM (%01101101)                            ' 5
  EEPROM (%01111101)                            ' 6
  EEPROM (%00000111)                            ' 7
  EEPROM (%01111111)                            ' 8
  EEPROM (%01100111)                            ' 9

Hex_Digits:
  EEPROM (%01110111)                            ' A
  EEPROM (%01111100)                            ' b
  EEPROM (%00111001)                            ' C
  EEPROM (%01011110)                            ' d
  EEPROM (%01111001)                            ' E
  EEPROM (%01110001)                            ' F


' -----[ Initialization ]------------------------------------------------------
'
Setup:
  Segs = Off                                    ' leds off
  DIRS = %01111111                              ' segments = outputs


' -----[ Main Code ]-----------------------------------------------------------
'
Main:
  SERIN Sio, Baud, ("!SS0"), cmd, value         ' wait for input

Check_Cmd:
  LOOKDOWN cmd, ("IBN"), cmd                    ' check cmd against valid
  IF cmd > 2 THEN Main                          ' if not in table, ignore
  BRANCH cmd, (Show_ID, Show_Bits, Show_Dig)

Show_ID:
  PAUSE 1                                       ' let master get ready
  SEROUT Sio, Baud, ("1.0")                     ' send ID string
  GOTO Main

Show_Bits:
  Segs = value & %01111111                      ' display pattern
  GOTO Main

Show_Dig:
  IF value > 15 THEN Main                       ' disallow bad value
  READ value, Segs                              ' read pattern from EEPROM
  GOTO Main

  END
```

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