



Column #131, March 2006 by Jon Williams:

## Wading the BS1 DEBUG Stream

*Until recently, I had never really considered using a BS1 as a front-end to a PC program – unlike the BS2, the BS1 doesn't have the ability to do SEROUT on its programming port. But then my colleague, Chuck, had this idea about building a BS1 right into a USB connector. Well, that changes things; now the BS1 is not just connected to the PC, it's nearly a part of it. With no SEROUT on the programming port, it's time to roll up the pant legs and wade through the BS1's DEBUG stream.*

For those of you that have been around for a long time, or have taken the time to go back through past issues of this column, you may remember that Scott Edwards tackled this subject back in October of 1996. Scott was able to [correctly] deduce most of the aspects of the BS1 DEBUG stream through empirical observation. I have the advantage of working "on the inside," and after spending an hour chatting with our compiler engineer it's my intent to show you how to use the BS1's DEBUG output in your PC projects.

### **The Dirt on BS1 DEBUG**

As Scott pointed out, any time you have a DEBUG instruction in a BS1 program, everything (all variables) get sent to the PC. This may seem odd at first, and yet there is a perfectly logical explanation: the BS1 only has 256 bytes of program memory. As we all know, that's not a lot of space and anything that can be done to conserve it is important. Chip, the BASIC Stamp's inventor, came up with an interesting solution that can be demonstrated with a simple program.

## Column #131: Wading the BS1 DEBUG Stream

Enter this program in your BASIC Stamp IDE and check the Memory Map:

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

SYMBOL count = B2

Main:
  FOR count = 1 TO 10
    DEBUG #count, CR
    PAUSE 300
  NEXT
  GOTO Main
```

If you typed it in just like the listing, you should see that the last location used is \$EF, so the program consumes 17 bytes of EEPROM. Okay, now modify the DEBUG line like this and open the Memory Map again:

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

SYMBOL count = B2

Main:
  FOR count = 1 TO 10
    DEBUG "The value of 'count' is: ", #count, CR
    PAUSE 300
  NEXT
  GOTO Main
```

Again, the last location used \$EF for a program total of 17 bytes. What the?... Interesting, isn't it? Here's the secret: that big string of characters we added to the program does not get stored inside the BS1, it's actually stored in a table inside the BASIC Stamp IDE. The reason for this is that we normally use DEBUG when we're connected to the IDE, so it made better sense to store the strings there than in the BS1 where they would very quickly eat through our precious program memory.

When we compile and download a program to our BS1 the editor creates a special table for all the occurrences of DEBUG; the table is indexed by the program counter (location) where the DEBUG instruction is placed. Part of the DEBUG packet is the program counter. When the editor receives a DEBUG packet, it grabs the program counter first and then checks the table to see how that specific DEBUG instruction is to be formatted for display.

Okay, let's talk about the DEBUG packet. Most of the time, it will be 97 bytes long. The first part of the packet is [usually] 64 bytes – this is for synchronization with the PC, and all the

sync bytes are \$F0. Remember that the BS1 was designed long before Windows and the original programming was via bit-banged serial on the PC's printer port (the printer port was used because of its TTL-level I/O which meant the connection could be simple and direct – no level shifting required). After the sync bytes we'll find either a \$5A (this is a DEBUG packet) or \$A5 (this is a Connect packet). What follows is a 32-byte data dump of the PIC16C56's (host micro) RAM space.

You may be wondering about the Connect packet. If you look at the programming circuit, there are only two lines (serial in and serial out) for programming. Unlike its big brother, the BS2, the BS1 cannot be reset by the BASIC Stamp IDE. So what the BS1 does is interleave Connect packets with the DEBUG output; when the IDE sees a Connect packet it can download a new program. This explains why when programming the BS1 we're occasionally forced to wait a bit – the editor is holding for a Connect packet.

In Scott's article he said that he had occasional connection errors and I think the reason why is that the sync section can actually be up to 128 bytes long – though I've not seen this in any of my own experiments. That said, we still have to deal with that if it occurs. Our basic strategy for using the BS1 DEBUG output in a PC program is simple: we'll program the BS1 with a DEBUG instruction, and then open the serial port used to program the BS1 and collect incoming bytes. Once we have enough bytes to constitute a DEBUG packet we'll look for a bunch of sync bytes followed by \$5A. If we find that, we'll grab the next 32 bytes and parse the variable values out of it. Ready? Let's give it a shot.

### BS1 DEBUG to PC

I'm most comfortable using Visual Basic for my PC programming chores, but if you're using another product (especially REALbasic) you should be able to port this code pretty easy. As with most of my projects, I definitely employ the KISS principle here.

The first thing to do, of course, is select an open a serial port. One thing to remember is that we can't open a port in use, so after we've downloaded a program to the BS1 we need to close the IDE's Debug Terminal window to free the port. Here's the bit of code that sets up and opens the serial port with VB:

```
With MSComm1
    .CommPort = bs1PortNum
    .DTREnable = True
    .RTSEnable = False
    .Settings = "4800,n,8,1"
    .RThreshold = 1
    .PortOpen = True
End With
```

## Column #131: Wading the BS1 DEBUG Stream

For those of you using a different language, here's what all that means: we set the port to 4800 baud, N81-style serial, we turn on the DTR line, make sure the RTS line is off, and set the serial object to fire an event any time a character shows up. When you open the full listing you'll see that the variable called `bs1PortNum` comes from a simple port selection dialog.

With the port open and a BS1 attached, bytes should start streaming in. The serial port `OnComm()` event is really simple.

```
Private Sub MSCComm1_OnComm()  
  
    If (MSCComm1.InBufferCount >= 97) Then  
        bs1Buffer = bs1Buffer & MSCComm1.Input  
    End If  
  
End Sub
```

As you can see, any time a character shows up we check to see if we have enough bytes in the buffer to constitute a valid DEBUG packet. If we do, we append the contents of the serial buffer to an internal variable, `bs1Buffer` (a string), for our processing. Note that this action empties the contents of the PC serial buffer.

Since Visual Basic is event based, we need some event to fire in order check the size of `bs1Buffer` to see if it needs processing. Again, let's keep things very simple: we'll use a Timer set to trip every 50 milliseconds when the serial port is open.

```
Private Sub Timer1_Timer()  
  
    If Len(bs1Buffer) >= 97 Then  
        Parse_Debug_Packet  
        Update_Display  
    End If  
  
End Sub
```

We could of course use any event, a button press for example, but if the BS1 is constantly streaming data the serial buffer could be overrun. Since a BS1 DEBUG takes about 200 milliseconds (97 bytes at 4800 baud), checking our buffer any time under this window will make sure the program responds quickly to the output from the BS1. Okay, if we have enough bytes to consider, it's time to get down to the nitty-gritty and fish the BS1 variables out of the DEBUG stream (Oi vai, that was corny).

```
Private Sub Parse_Debug_Packet()  
  
    Dim syncStr As String
```

```

Dim foundSync As Long
Dim checkLen As Long
Dim idx As Byte

syncStr = String$(17, &HF0) & String(1, &H5A)
foundSync = InStr(1, bs1Buffer, syncStr, vbTextCompare)

If (foundSync > 0) Then
    checkLen = Len(bs1Buffer) - (foundSync + 18)
    ' do we have the whole packet?
    If (checkLen >= 32) Then
        bs1Buffer = Mid$(bs1Buffer, (foundSync + 18))
        bs1IORegs(0) = Asc(Mid$(bs1Buffer, 18, 1))
        bs1IORegs(1) = Asc(Mid$(bs1Buffer, 17, 1))
        bs1IORegs(2) = Asc(Mid$(bs1Buffer, 7, 1))
        For idx = 0 To 13
            bs1ByteRegs(idx) = Asc(Mid$(bs1Buffer, (19 + idx), 1))
        Next
        For idx = 0 To 6
            bs1WordRegs(idx) = (CLng(bs1ByteRegs(idx * 2 + 1)) * 256) _
                + bs1ByteRegs(idx * 2)
        Next
        bs1Buffer = Mid$(bs1Buffer, 32)
    End If
End If

End Sub

```

This may look a little tricky at first, but really, it's not that bad. One of the nice things about using the serial port object in VB is that the buffer is treated like a string; this fact lets us use some of the neat string functions of Visual Basic. We see that first with the creation of syncStr. On the advice of Parallax's compiler engineer, Jeff, we want to look for at least 17 sync bytes (\$F0) followed by the \$5A packet descriptor. This is pretty simple using VB's String() function. With that string created, we can use the Instr() function to determine if the DEBUG sync header exists in buffer. If not (could have been a Connect packet), we simply exit the subroutine; if the sync header is present then we can parse out the variables.

Before we attempt to do the parsing, however, we need to ensure that we've got the entire packet. It is possible that the serial port got closed in the middle of the BS1 DEBUG output and we don't have the whole thing. It's a simple matter to check: we look that the starting position of the sync string in the buffer, move forward 18 bytes (to account for the sync string), and then check to see if there are at least 32 bytes left in the buffer. If the answer is yes, we move on with the parsing.

We'll start by using the Mid\$() function to trim away the leading sync and packet descriptor bytes. After that it's a simple matter of pulling the variables from their respective positions in

the stream. In this program we have three arrays: bs1IORegs() which holds the DIRS and PINS (one for Outs, one for Ins) values, bs1ByteRegs() which holds the values of BS1 variables B0-B13, and bs1WordRegs() which is actually assembled from the values of bs1ByteRegs().

Of particular note is the bs1WordRegs() is an array of Longs. In VB, an integer is a 16-bit signed value, so assembling the unsigned word variables from the byte variables requires more bits. Longs use four bytes so that gives us the space to handle any value in a BS1 word variable. As you can see, we have to use the CLng() (convert to Long) function to ensure the value is correctly calculated. Note that when accessing strings in VB, the first byte is in position 1. Knowing this we can see that the PINS inputs are in position 7, the PINS outputs in position 17, the DIRS register in position 18, and, finally, B0-B13 are located in positions 19-32. These positions correspond with their locations in the PIC RAM space.

Okay, once the current values are parsed out we trim the packet from the front end of the buffer and move on to other things. See, it's not that bad.

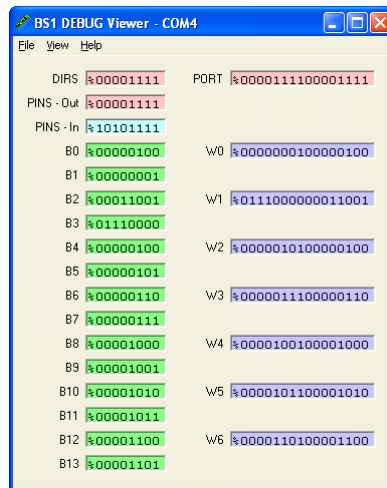


Figure 131.1: BS1 DEBUG Viewer

So what do we do now? Well, while developing the program I thought it might be useful to create a little utility that displays the BS1 DEBUG data. A screen shot of that program is shown in Figure 131.1. For those of you who have used VB, you know that VB doesn't know how to display binary variables – the solution is a simple custom function that takes a value and converts it to a binary string:

```

Private Function BinStr(value As Long, width As Byte) As String

    Dim idx As Byte
    Dim testBit As Long
    Dim tmpBin As String

    testBit = 2 ^ width
    value = value Mod testBit
    tmpBin = ""
    For idx = 1 To width
        testBit = testBit \ 2
        If (value >= testBit) Then
            tmpBin = tmpBin & "1"
            value = value - testBit
        Else
            tmpBin = tmpBin & "0"
        End If
    Next
    BinStr = tmpBin
End Function

```

This works similar to the BIN modifiers in the BS2. If, for example, we wanted to display the lowest four bits of a value, we would use the function like this:

```
lblLedStatus.Caption = BinStr(ledStatus, 4)
```

The routine is pretty simple: it calculates the next-highest bit value and truncates any unneeded bits from the input with Mod (modulus). Then it's a simple matter of looping through each bit position to see if it's set. This is done mathematically instead of logically; it's just simpler that way in VB. The result is a binary string representation of our value. In the full program listing you'll find a similar function for converting values to fixed-width hexadecimal strings.

You can use the BS1 DEBUG Viewer application with any BS1 program that has at least one DEBUG instruction. One of the interesting things you'll see is the manipulation of W6 by a program that uses GOSUB – W6 is used as the RETURN stack. You'll also see that the PINS outputs will affect the corresponding PINS inputs.

### Show Me the Temperature

As most of you know, I am freakishly sensitive to the temperature of my environment, hence I'm constantly looking at thermometers and adjusting the thermostat. Perhaps I need professional help.... Well, until that day, I decided to make it easier to check the temperature where I spend most of my day: in front of my computer.

What I did is take an 8-pin socket and extend the legs with wire and a 1K resistor to match our standard DS1620 circuit as shown in Figure 131.2. I popped a DS1620 into the socket and plugged my temperature "spider" into the USB-BS1.

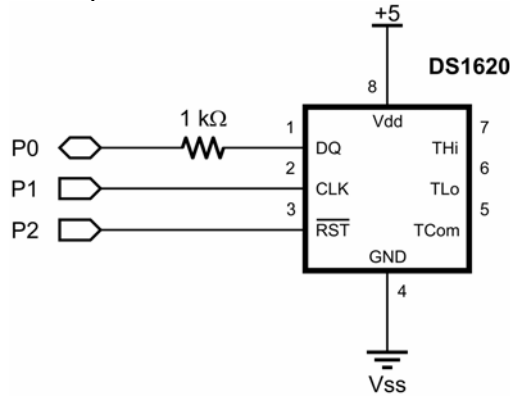


Figure 131.2: A Standard DS1620 Circuit

From there it was a very simple matter to gut and convert the BS1 DEBUG Viewer application to display temperature. Since we've used the DS1620 so many times in the past I'm not going to go through all the details, I just want to cover shifting data out and in with the BS1.

The BS1 does not have the BS2's SHIFTOUT and SHIFTIN instructions, so we're forced to synthesize these in code. It's really pretty simple. One of the things you'll notice about my BS1 programs is that I always start assigning variables at B2 (or W1). The reason for this is that I want to leave B0 and B1 (W0) free in case I need bit-level access later (W0 is the only variable that allows bit-level access). That is the case with shifting subroutines; let's have a look.

Here's a subroutine that will shift an eight-bit value to the DS1620, LSB first:

```
Shift_Out:
  DIRS = %00000111
  Clock = IsHigh
  FOR shift = 1 TO 8
    DQ = BIT0
    PULSOUT Clock, 10
    dByte = dByte / 2
  NEXT
  RETURN
```

The key section here is the middle of the FOR-NEXT loop. Notice that DQ (the output pin to the DS1620) is set to the value of BIT0. This is the LSB of B0, which in this program is aliased as dByte. After placing the bit on the DQ line, the clock line is blipped with PULSOUT, and then the value of dByte is divided by two. Dividing by two is the same as shifting right by one bit, so this process puts the next highest bit into the BIT0 position.

Coding to shift bits in is similar. In the DS1620 the temperature returned as a nine-bit, two's-compliment (if negative) value, with the MSB is used as the sign (0 for positive, 1 for negative).

```
Temp_In:
  DIRS = %00000110\
  tempC = 0
  Clock = IsHigh
  FOR shift = 1 TO 9
    tempC = tempC / 2
    Clock = IsLow
    BIT8 = DQ
    Clock = IsHigh
  NEXT
  tempHi = -sign
  RETURN
```

As you can see, things are reversed: the first thing we do is shift the result variable (tempC) to the right, the clock gets blipped, and then we collect a bit from DQ and put it into the MSB position (BIT8). This seems odd, at first, but after you work your way through it makes good sense.

Since we only get nine bits back from the DS1620 but need all 16 bits of a word to be properly configured to designate a negative value, the final line of [working] code handles this neatly for us. And, big bonus here, Visual Basic understands PBASIC negative values. This makes the temperature display program a breeze to code. Here's the working part:

```
Private Sub Timer1_Timer()

  If Len(bs1Buffer) >= 97 Then
    Parse_Debug_Packet
    tempC = CSng(bs1WordRegs(0)) / 2
    tempF = tempC * 9 / 5 + 32
    Update_Display
  End If

End Sub
```

How simple is that? After we get and parse a DEBUG packet the value of W0 (this is what we use to return the raw value from the DS1620) is converted to floating point with the CSng() function and the rest is automatic. Remember that the DS1620 returns the temperature in units of 0.5 degrees Celsius, so we have to divide by two to get the correct value. Converting to Fahrenheit requires just a bit of high school math. I kept the program very simple – you can see the output in Figure 131.3.

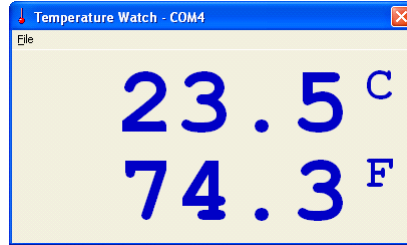


Figure 131.3: Temperature Watch Display

One last note on using the BS1: there is no way to get information from our PC application to the BS1 through the programming port. If that limitation is not a barrier, you now have the expertise and the means to use the BS1 for getting information into your PC application.

And, finally, my friends at Nuts & Volts have asked me to put additional focus on completed projects. That's okay by me, and I'd really like to do projects that many people will find interesting and useful. I'm currently working on a six-digit, 13-segment display using an exciting new product that Parallax will be announcing shortly. What kind of projects would you like to see? Please e-mail your ideas, and I'll see what I can do about turning them into full-blown projects.

Until next time, Happy St. Patrick's Day and Happy Stamping!

**Project Code**

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

```
SYMBOL count = B2
```

```
Main:
```

```
FOR count = 1 TO 10
  DEBUG #count, CR
  PAUSE 300
NEXT
GOTO Main
```

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

```
Setup:
```

```
DIRS = $0F          ' make P0 - P3 outputs
PINS = $0A          ' set P1 and P3 high
```

```
B0 = 0              ' initialized variables
B1 = 1
B2 = 2
B3 = 3
B4 = 4
B5 = 5
B6 = 6
B7 = 7
B8 = 8
B9 = 9
B10 = 10
B11 = 11
B12 = 12
B13 = 13
```

```
Main:
```

```
FOR B0 = 1 TO 3
  GOSUB Shake_It_Up      ' affect W6 (RETURN stack)
  GOSUB Shake_It_Up
  GOSUB Shake_It_Up
NEXT
DEBUG W1                ' send values to PC
PAUSE 800                ' 800 + 200 (for DEBUG) = 1
sec
```

```
GOTO Main
```

```
Shake_It_Up:
```

```
RANDOM W1              ' "stir" random value
RETURN
```

## Column #131: Wading the BS1 DEBUG Stream

```
' =====  
'  
' File..... DS1620-USB.BS1  
' Purpose.... Using a DS1620 connected to the BS1-USB  
' Author..... Jon Williams -- Parallax, Inc.  
' Started....  
' Updated.... 11 JAN 2006  
'  
'   {$STAMP BS1}  
'   {$PBASIC 1.0}  
' =====  
'  
' -----[ Program Description ]-----  
'  
' Reads the raw temperature from a DS1620 and sends to the PC using DEBUG  
' every second. The delay between readings is set to 800 milliseconds  
' because the BS1 DEBUG output takes 200 milliseconds by itself.  
'  
' Output can be captured by "Temperature Watch.EXE" for display on the  
' PC screen.  
'  
' -----[ I/O Definitions ]-----  
'  
SYMBOL DQ           = PIN0           ' DS1620.1 (data I/O)  
SYMBOL Clock       = PIN1           ' DS1620.2  
SYMBOL Rst         = PIN2           ' DS1620.3  
'  
' -----[ Constants ]-----  
'  
SYMBOL IsHigh      = 1  
SYMBOL IsLow       = 0  
'  
SYMBOL RdTmp       = $AA           ' read temperature  
SYMBOL WrHi        = $01           ' write TH (high temp)  
SYMBOL WrLo        = $02           ' write TL (low temp)  
SYMBOL RdHi        = $A1           ' read TH  
SYMBOL RdLo        = $A2           ' read TL  
SYMBOL StartC      = $EE           ' start conversion  
SYMBOL StopC       = $22           ' stop conversion  
SYMBOL WrCfg       = $0C           ' write config register  
SYMBOL RdCfg       = $AC           ' read config register  
'  
' -----[ Variables ]-----  
'  
SYMBOL dByte       = B0           ' value sent to DS1620  
SYMBOL shift       = B2           ' shift counter  
SYMBOL tempC       = W0           ' temp in C * 0.5  
SYMBOL sign        = BIT8         ' 1 = negative  
SYMBOL tempHi      = B1           ' high byte of tempC
```

```

' -----[ Initialization ]-----
Reset:
  DIRS = %00000111          ' set pins to outputs
  Rst = IsHigh              ' alert the DS1620
  dByte = WrCfg             ' prepare to write config
  GOSUB Shift_Out
  dByte = %00000010        ' with CPU; free run mode
  GOSUB Shift_Out
  Rst = IsLow
  PAUSE 10                  ' 10 ms for EEPROM write
  Rst = IsHigh              ' alert the DS1620
  dByte = StartC
  GOSUB Shift_Out          ' continuous conversion
  Rst = IsLow               ' end DS1620 comms

' -----[ Program Code ]-----
Main:
  PAUSE 800                 ' delay between readings
  Rst = IsHigh              ' alert the DS1620
  dByte = RdTmp             ' prep to read temperature
  GOSUB Shift_Out
  GOSUB Temp_In
  Rst = IsLow               ' end DS1620 comms
  DEBUG tempC               ' send to PC
  GOTO Main

' -----[ Subroutines ]-----
' Send byte to DS1620
' - 8-bit value is sent LSB -> MSB
'
Shift_Out:
  DIRS = %00000111          ' set pins to outputs
  Clock = IsHigh
  FOR shift = 1 TO 8        ' shift eight bits
    DQ = BIT0                ' get LSB of cmdByte
    PULSOUT Clock, 10       ' clock out the bit
    dByte = dByte / 2       ' shift byte for next bit
  NEXT
  RETURN

' -----
' Retrieve temperature from DS1620
' - value comes in LSB -> MSB in 9 bits
' - 1 bit for sign
' - 8 bits for temp (in 2's compliment format if negative)
'

```

## Column #131: Wading the BS1 DEBUG Stream

```
Temp_In:
  DIRS = %00000110           ' make DQ an input
  tempC = 0                  ' clear result value
  Clock = IsHigh
  FOR shift = 1 TO 9         ' get nine bits
    tempC = tempC / 2        ' shift temp bits
    Clock = IsLow
    BIT8 = DQ                ' get data bit
    Clock = IsHigh
  NEXT
  tempHi = -sign             ' extend sign bits
  RETURN
```