Column #129, January 2006 by Jon Williams:

PlayStation Robot Controller

The other day my boss, Ken, pointed out that I have written over six year's worth of columns for Nuts & Volts. Wow. Aren't you guys tired of me yet? (Okay, don't answer that question) For all the columns I've written, clearly one of the top three in reader interest was called "PlayStation Control Redux" (September 2003) where we delved more deeply into the PlayStation controller protocol work started by Aaron Dahlen. Well, between then and now Parallax released the SX/B compiler for the SX micro and the speed issues we dealt with when using a BASIC Stamp are no longer issues. That, and Ken is building a cool new treaded robot that might need a full featured control device – let's hack a PlayStation controller for him and let him drive that dude around, shall we?

During a recent conversation with a Parallax EFX customer I was asked how difficult it is to learn SX assembly language – my friend is interested in building custom accessory devices for his props and holiday displays using the SX28. He was actually quite surprised to learn that, to date, all of the EFX accessory products (RC-4, DC-16, AP-8) that use the SX are actually programmed in SX/B – I know because I'm part of the team that designed those products and wrote a few of the programs myself.

Why did I use SX/B? Well, I'm part of the SX/B development team so I'm really comfortable with it and – here's the kicker – I still haven't taken the time to commit to learning enough assembly programming to write full-blown applications. What it actually comes down to is a lack of patience on my part, and with SX/B I really don't need to be; I can write very PBASIC-like code that gets compiled. I get the benefits of high-level programming with the execution speed of assembly language.

That said, SX/B is a not a compiler in the terms that we typically think about, that is, SX/B doesn't optimize and automatically remove redundant code. Why not? The reason is that Parallax created SX/B so that those interested in assembly could learn from it – that's very tough to do when one looks at the assembly output of an optimized compiler. With SX/B you can see the assembly output from your high-level code (which gets included in the comments) and see how the various instructions work "under the hood."

So does that mean SX/B is inefficient? No, I don't think so; it is what it is: an inline (some call "macro") compiler. The code we write gets compiled inline as it appears in the source file. If, for example, we have two consecutive PAUSE instructions, the code to execute PAUSE will be expanded twice – and this does use more code space. This is not a problem if we understand and design around it, and that's really what I'm going to focus on in this month's column.

If you look at enough of my SX/B programs you'll notice that they are all similarly structured and, in fact, I reuse a lot of the same subroutines. The reason is this: By keeping my code consistent I can follow my own programs and get back into them more quickly after a break and, here's the real import part for SX/B, by putting "big" (lots of assembly code required) instructions into a subroutine those instructions only get expanded once and I'm able to conserve code space. The additional benefit to putting these commands into subroutines is that we can add our own (even optional) features to the routines. We'll see how just a bit later.

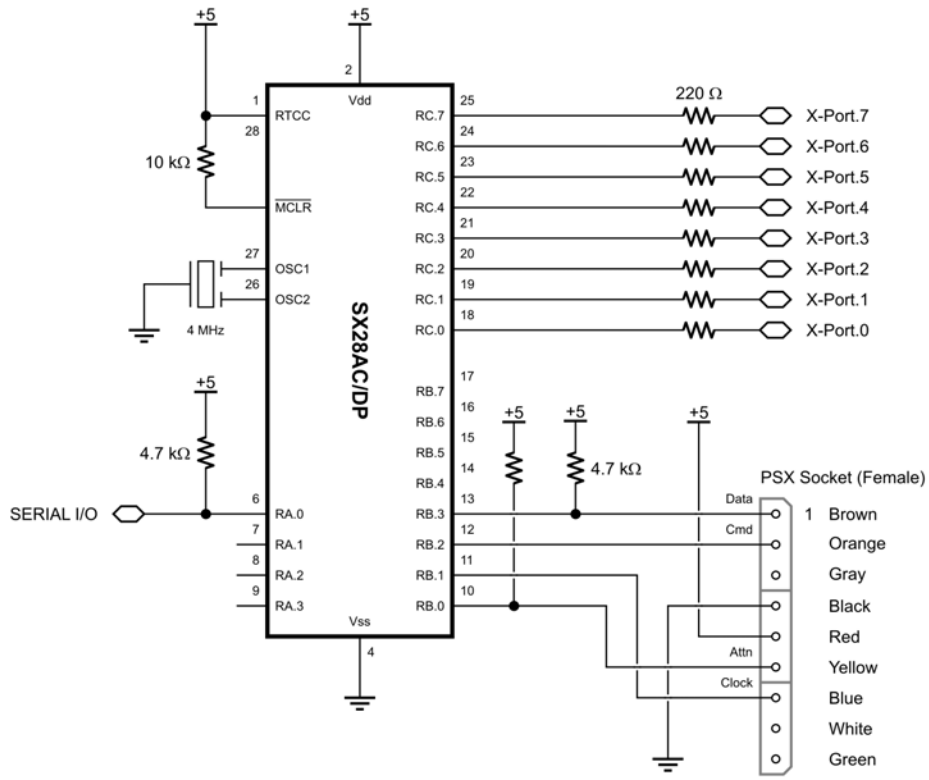

Figure 129.1: SX28 PlayStation Controller Schematic

PlayStation Controller Protocol

It turns out that the PlayStation controller is actually very easy to connect to a microcontroller – in fact, it behaves just like a big shift register. The difference is that it has separate data in (called Command) and data out (called Data) lines. When we used the BASIC Stamp SHIFTOUT and SHIFTIN were used, but this created a problem with the last bit of data when using an analog controller. What we ended up doing was synthesizing a routine that could send and receive bytes at the same time, but in PBASIC that's a little on the slow side. Not so with the SX, in fact we now have to consider speed for the other side so that we don't do things too quickly.

Figure 129.2 shows the signal timing and relationships between the host and the PlayStation controller. Communication is initiated by bringing the PsxAttn (attention) pin low. After a 20 microsecond delay the bits are clocked in and out, with everything happening based on the falling edge of the clock signal.

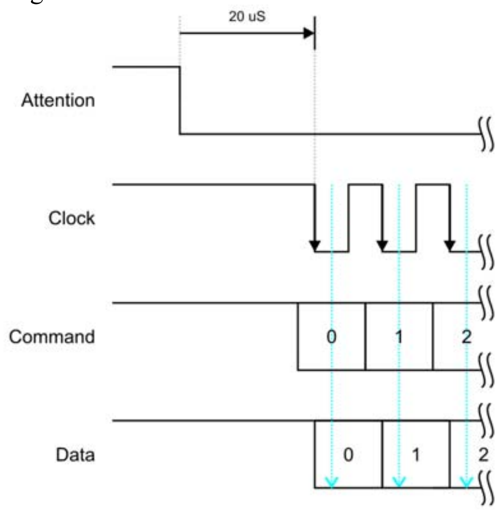

Figure 129.2 Signal Timing between Host and PlayStation Controller

From a programming standpoint we need to put a bit (starting with the LSB) on the PsxCmd pin before pulling the clock line low. After the clock has been pulled low and we allow a bit of setup time we can read a bit from the PsxData pin. We'll get into the specific code mechanics a little later.

Figure 129.3 shows the relationship of input and output bytes. The host transmits $01 (start) and $42 (get data), the PlayStation controller sends back its type, $5A (ready), then two (digital controller) or six data bytes (analog controller). Note that the controller transmits its type while the host is sending the $42 byte. What we're going to do as we develop this program is create a routine that does the equivalent of SHIFTOUT and SHIFTIN – but at the same time.

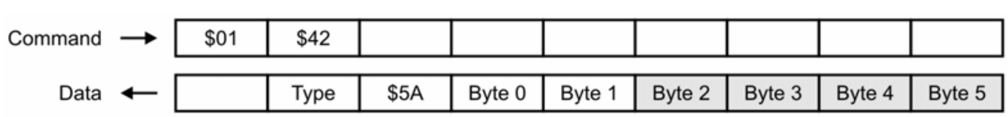

Figure 129.3: Relationship of Input and Output Bytes

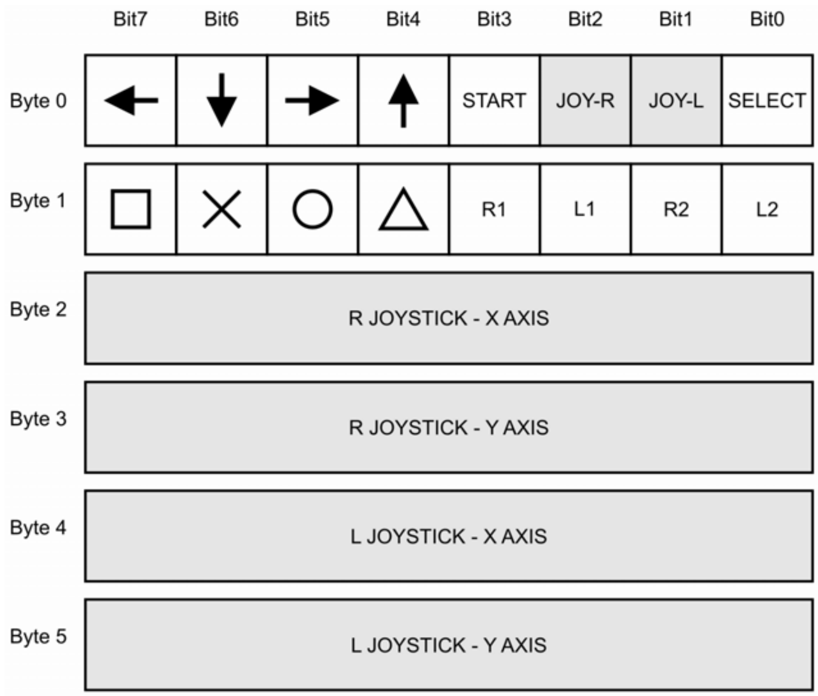

Figure 129.4: Composition of Data Byte Packets

The Tao of SX/B

Okay, I know that's a bit of a cheeky section title, since almost every programming language can be manipulated in any way by an experienced programmer. So this is my Tao of SX/B, at least for serial accessory devices. Let's start at the top.

One of the features I like best about SX/B is the ability to define subroutines with the SUB keyword. This serves two important functions: 1) It causes the compiler to create a jump table that lets us put the subroutine code anywhere in memory (remember, in the SX, subroutines usually have to be in the top half of a code page unless a jump table is used) and 2) It lets us tell the compiler how many parameters are used by the subroutine. This allows the compiler do syntax checks on our custom routines – very handy! Here are the subroutines used in the PlayStation Helper module:

```
WAIT_US          SUB      1, 2
WAIT_MS          SUB      1, 2
RX_BYTE          SUB
TX_BYTE          SUB      1
TX_OUT           SUB      1, 2
READ_PSX         SUB
PSX_SHIFTIO      SUB      0, 1
```

Here's a secret: Only the last two subroutines are specific to this project; all the others form the core of most of the serial accessory projects I developed using the SX. Looking at the code you'll see that each subroutine has a name, followed by the keyword SUB, and then information on parameters used by each subroutine. Notice that not every subroutine requires parameters sent to it (like RX_BYTE) and most actually have a variable number of parameters. WAIT_US (a shell for PAUSEUS), for example, requires one parameter and can take two.

With the subroutines defined we can jump into the main body of the program. As with similar devices, the PlayStation Helper chip is going to wait on a specific command header from the host and respond as instructed. We're going to use open-baudmode style serial communications with this product so that it's compatible with other serial accessories. By doing that we could connect this device to a BASIC Stamp using the same serial line that commands a Parallax Servo Controller (PSC). With a BASIC Stamp, a PSC, and the PlayStation Helper you could put together a very cool robot.

The Parallax AppMod protocol is really more of a configuration than a defined protocol – as I just stated it uses open-baudmode communications and a text header that starts with the "!" character. For example, when we want to send a command to the PSC we use the header "!SC" at the beginning of each command message. Let's be logical, shall we, and use "!PSX" as the header for our PlayStation Helper. Okay, then, let's wait for the header:

```
Main:
  char = RX_BYTE
  IF char <> "!" THEN Main
  char = RX_BYTE
  IF char <> "P" THEN Main
  char = RX_BYTE
  IF char <> "S" THEN Main
  char = RX_BYTE
  IF char <> "X" THEN Main
```

You see, I told you it was simple. We grab one character at a time, compare it to the header sequence and jump back to Main if anything is out of whack. Now, if you're new to SX/B you're probably wondering how this can work, that is, having a comparison between incoming serial bytes.

This works fine because the SX is running assembly language and even at the 4 MHz clock we're using each instruction only takes 0.25 microseconds! At 38.4k baud, each bit is 26 microseconds long so there is plenty of time during the stop bit to get the comparison done. Remember, this code gets compiled to assembly language. Here's a small section of the compiled code:

```
Main:
  CALL @__RX_BYTE
  MOV  char, W
  CJNE char, #"!", @Main
```

The first line calls the RX_BYTE subroutine – note that @ is used so the subroutine call can cross code pages. On return, the value that was received is retrieved from the W (working) register; this takes one cycle. The comparison is just one line of assembly code, but is a compound statement that takes either four or six cycles, depending on the comparison result. Still, in the worst case we've only consumed seven cycles – 1.75 microseconds – during the 26 microsecond window between bytes. I'm not suggesting we go crazy and try to squeeze a whole lot more (in actual fact a few more cycles are consumed with the call to and return from the RX_BYTE subroutine), but I want you to rest easy that when compiled we can do the comparison as shown without any fear of missing the next serial byte.

Okay, speaking of serial bytes, let's look at the code that handles that:

```
RX_BYTE:
  SERIN Sio, Baud, temp1
  IF temp1 >= "a" THEN
    IF temp1 <= "z" THEN
      temp1 = temp1 - $20
    ENDIF
  ENDIF
  RETURN temp1
```

This subroutine actually serves two purposes: it receives the serial byte and if the byte is a lowercase letter it gets converted to uppercase. This subroutine points out one of the changes in SX/B as it has matured and developed an expanding customer base, specifically the ability to return a value to the subroutine caller. As we saw in the compiled code above, the W register is used as the mechanism for handling the return value.

Let me emphasize on final time the reason for this subroutine: SERIN is a complex statement that requires several line of assembly code. If we were to use SERIN every place in the program that required serial input we would use a lot of code space with redundant code. And, by encapsulating SERIN in a subroutine, we're able to add the lowercase-to-uppercase conversion feature.

Now that we have the header, the next step is to process receive and process the command byte sent by the host controller:

```
Get_Command:
  char = RX_BYTE
  IF char = "V" THEN Show_Version
  IF char = "T" THEN Get_Type
  IF char = "S" THEN Get_Status
  IF char = "B" THEN Get_Buttons
  IF char = "J" THEN Get_Joysticks
  IF char = "C" THEN Config_IoPort
  IF char = "W" THEN Write_IoPort
  IF char = "R" THEN Read_IoPort
  GOTO Main
```

After receiving the command byte the program simply compares it to the list of commands available to the program. You may think that LOOKDOWN and BRANCH would be more efficient, but in practice it doesn't use any less code (after being compiled) and it's not quite as easy to follow in my book.

The first command is "V" for version; this is a good idea to include in your designs, especially if you're selling them as products and make incremental improvements. Providing a version number allows the end user to design around the features available in the product he has. On receiving the "V" command the PlayStation Helper will send back a three-byte version string. Here's the top level code:

```
Show_Version:
  WAIT_MS 1
  TX_OUT Version
  GOTO Main
```

There's no big mystery here; the only thing you may wonder about is the WAIT_MS 1 line. This inserts a one millisecond delay before returning the version string so that the BASIC Stamp can load up its SERIN instruction to receive the data from the SX. Here's the code for WAIT_MS:

```
WAIT_MS:
  temp1 = __PARAM1
  IF __PARAMCNT = 1 THEN
    temp2 = 1
```

```
  ELSE
    temp2 = __PARAM2
  ENDIF
  IF temp1 > 0 THEN
    IF temp2 > 0 THEN
      PAUSE temp1 * temp2
    ENDIF
  ENDIF
  RETURN
```

This is a subroutine that can handle a variable number of parameters (one or two). The first parameter is required and is the base delay time in milliseconds. If a second parameter is provided this is used as a multiplier, otherwise the multiplication factor is set to one. The internal variable, __PARAMCNT, is used to check the number of parameters sent to the subroutine, and as you can see it gives us a lot of flexibility. Finally, we check to see that neither parameter was set to zero and do the delay using the version of PAUSE that uses the multiplication of two bytes.

After the delay we send the version string back to the host with TX_OUT. Let's look at that code:

```
TX_OUT:
  temp3 = __PARAM1
  IF __PARAMCNT = 2 THEN
    temp4 = __PARAM2
    DO
      READ temp4 + temp3, temp5
      IF temp5 = 0 THEN EXIT
      TX_BYTE temp5
      INC temp3
      temp4 = temp4 + Z
    LOOP
  ELSE
    TX_BYTE temp3
  ENDIF
  RETURN
```

TX_OUT is quite flexible in that it can be used to transmit a single byte or multi-byte strings (stored as z-strings). Again we use __PARAMCNT to determine the behavior of the subroutine. When a single byte is passed there will only be one parameter. When a string is passed to the subroutine two parameters are required due to the 12-bit size of the string address. In the case of returning the version to the host two parameters will be passed to the subroutine: the base and offset address values of that string.

It's important to note that strings can be handled in two ways. For the version string we're going to store it in a DATA statement like this:

```
Version:
  DATA    "0.1", 0
```

When we use a stored string like this we must append the zero terminator ourselves and we'll pass the string label to the subroutine – this gets resolved by the compiler to the base and offset memory locations. The nice thing about this subroutine is that it also lets us send inline strings like this:

```
  TX_OUT "Nuts & Volts rocks!"
```

When we pass an inline string the compiler adds the zero-terminator for us. Note that if we're going to send the same string more than once then the most efficient method is to store the string in a DATA statement.

Getting back to TX_OUT we see that it uses a DO-LOOP construct to transmit the string. READ is used to retrieve each character from memory and if it's zero we're done (hence the use of EXIT). Remember that SX/B variables are bytes only but we're using a 12-bit address for the string characters. What this means is that when we increment the offset value we need to update the base value on a roll-over. This is actually quite easy to do as the Z flag will be set (to 1) when we increment the offset from 255 to 0 – all we have to do is add the Z bit to the base after incrementing the offset. In most cases the Z bit will be zero but when we have a roll-over it will be set to 1 and the base will be updated properly.

Note that TX_OUT calls the TX_BYTE subroutine. This one is really easy; it simply makes a copy of the byte passed to it and then transmits it with SEROUT on the specified port at the program baud rate:

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

In actual fact, TX_OUT started as TX_STR (transmit string) and always required two bytes. It was a simple matter to update the subroutine to handle one byte or two so the main code only ever needs to call TX_OUT. Yes, we could use TX_BYTE, but if we made a change from sending a byte to sending a string we'd also have to change which subroutine gets used. By only using TX_OUT in the main body of our program we never have to worry about that.

So far the program has been pretty generic – and that's the point. What I'm suggesting is that we can use this framework for a whole host of serial accessories that are useful for BASIC Stamp (and other microcontroller) projects. As I indicated earlier, this framework runs in the RC-4, DC-16, and AP-8 products that are part of the Parallax EFX line; you can do it too.

Let's get into the PlayStation-specific code. Remember that the PlayStation controller acts like a big, smart shift register, and it can receive and transmit data at the same time. Since SHIFTOUT and SHIFTIN do only one thing each, let's create a subroutine that handles the full-duplex nature of the controller.

```
PSX_SHIFTIO:
  IF __PARAMCNT = 1 THEN
    temp3 = __PARAM1
  ELSE
    temp3 = 0
  ENDIF
  temp4 = 0
  FOR temp5 = 1 TO 8
    PsxCmd = temp3.0
    temp3 = temp3 >> 1
    PsxClock = 0
    WAIT_US 5
    temp4 = temp4 >> 1
    temp4.7 = PsxData
    PsxClock = 1
    WAIT_US 5
  NEXT
  RETURN temp4
```

This is definitely the trickiest subroutine in the program in that it can send a byte to the controller, it can get a byte from the controller, and it can do both at the same time. We'll see all three uses of the subroutine's capabilities in just a bit.

When the subroutine is called with an output parameter that value is copied into temp3 – if not provided, temp3 is set to zero as this is the output byte to the controller. Before entering the transmission loop, temp4 gets cleared; this is the input byte from the controller and will be passed back to the caller. A FOR-NEXT loop is used to send and receive eight bits, and the transmission – in PBASIC terms – is LSBFIRST. The first step is to put the LSB (temp3.0) onto the PsxCmd pin and then pull the clock line low to output that bit. Note that we shift the next bit right before the clock to add a bit of timing delay before the clock change and to have the next bit in place for the next iteration of the loop.

After the clock line goes low the controller will output a data bit (LSBFIRST) onto the PsxData pin. Here's where things can look a little confusing at first. We start by shifting temp4 to the right by one bit and then placing the data line bit into temp4.7. We have to do this because we ultimately want the first bit read to end up in temp4.0 – this will in fact happen after eight iterations of the loop.

One thing of note is the clock timing. I don't actually have a PlayStation console but I met a guy named Jim in the Parallax user forums who happened to borrow one from his nephew. He connected a 'scope and told me that the high and low timing of the clock line was about five microseconds. That's what I've been using and have never had a problem – I suspect it's probably a bit on the generous side but I see no need to push it. At this speed it takes just about 5 milliseconds to get the entire packet from the controller.

And here's the code that does just that:

```
READ_PSX:
  PsxAttn = 0
  WAIT_US 20
              PSX_SHIFTIO $01
  psxId     = PSX_SHIFTIO $42
  psxStatus = PSX_SHIFTIO
  psxThumb1 = PSX_SHIFTIO
  psxThumb2 = PSX_SHIFTIO
  IF psxId = $73 THEN
    psxJoyRX  = PSX_SHIFTIO
    psxJoyRY  = PSX_SHIFTIO
    psxJoyLX  = PSX_SHIFTIO
    psxJoyLY  = PSX_SHIFTIO
  ELSE
    psxJoyRX  = $80
    psxJoyRY  = $80
    psxJoyLX  = $80
    psxJoyLY  = $80
  ENDIF
  PsxAttn = 1

  psxThumb1 = ~psxThumb1
  psxThumb2 = ~psxThumb2
  RETURN
```

This routine starts by pulling the PsxAttn line low to activate the controller. According to Jim, the PlayStation console waits 20 microseconds before transmitting the start byte ($01) so I've put that into my code. For those of you that have used the BASIC Stamp to connect to the PlayStation controller we need to keep in mind that it takes at least 100 microseconds to load each instruction so there's a lot of built-in delays. Since we're dealing with compiled code we have to manually put those delays in. The WAIT_US subroutine is identical to the

WAIT_MS routine that we looked at earlier, the difference being that it uses PAUSEUS instead of PAUSE.

The READ_PSX subroutine shows the flexibility that we built into the PSX_SHIFTIO routine. We start by sending $01 – notice that we don't care about anything that gets returned so there is no assignment. The next line, however, sends $42 (get data) with PSX_SHIFTIO and assigns the return value to psxId. This tells us what kind of controller is connected; it will usually be $41 for digital controllers or $73 for analog controllers. After the ID byte the controller transmits a packet header of $5A. After this header controller sends two bytes of button data and, if in analog mode, four bytes of joystick data.

I happen to have Sony analog controller that can be set to digital or analog mode. I made a decision for this subroutine to stuff the joystick bytes with $80 if the controller is digital or set to digital mode. The value $80 represents the center position of each joystick axis and allows me to simplify my BASIC Stamp programs. If we don't include this conditional code then each joystick value will be set to $FF (extreme right or down position) when in digital mode, and in my mind this is not the best value to return to the host.

Finally, the subroutine inverts the button bits so that a pressed button bit has a value of 1 when sent back to the BASIC Stamp.

Okay, now that we can read the controller, the command sections that handle the various requests for data are a breeze.

```
Get_Status:
  WAIT_MS 1
  READ_PSX
  TX_OUT psxThumb1
  TX_OUT psxThumb2
  TX_OUT psxJoyRX
  TX_OUT psxJoyRY
  TX_OUT psxJoyLX
  TX_OUT psxJoyLY
  GOTO Main

Get_Buttons:
  WAIT_MS 1
  READ_PSX
  TX_OUT psxThumb1
  TX_OUT psxThumb2
  GOTO Main

Get_Joysticks:
  WAIT_MS 1
  READ_PSX
```

```
  TX_OUT psxJoyRX
  TX_OUT psxJoyRY
  TX_OUT psxJoyLX
  TX_OUT psxJoyLY
  GOTO Main
```

As you can see, all of this code is very straightforward and gives us the ability to request from the PlayStation Helper module just what we need. Figure 129.5 shows the output from a simple BASIC Stamp controller that retrieves and displays the controller values (it's included in the download files).

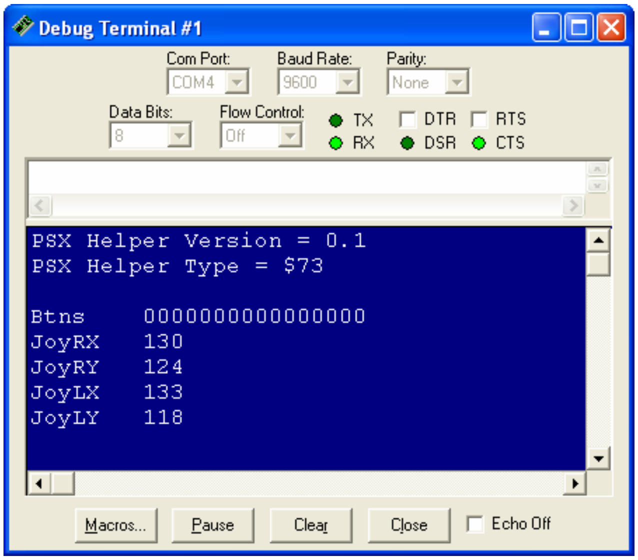

Figure 129.5: PSX Helper Test Output

Since this is designed to be a robot controller let's take advantage of those spare pins on the SX28. By using the "C," "W," and "R" commands we can configure, write, and read the RC port. Just one caveat: the SX uses 0 to indicate an output bit, and 1 to indicate an input bit – this is exactly opposite of what we do in the BASIC Stamp (DIRS register). Knowing this we will send BASIC Stamp style data to the PlayStation Helper module and invert the bits before assigning the configuration value to the TRIS register. Here's the code for handling the extra I/O port:

```
Config_IoPort:
  char = RX_BYTE
```

```
  PlpIO = char
  char = ~char
  TrisIO = char
  GOTO Main

Write_IoPort:
  IoPort = RX_BYTE
  GOTO Main

Read_IoPort:
  WAIT_MS 1
  TX_OUT ~IoPort
  GOTO Main
```

One of the things that you'll notice about the Config_IoPort section is that the SX pull-ups are activated on any pin that is made an input. Now this means that inputs will be active-low, so we'll invert the bits sent back to the BASIC Stamp to make them look active-high – just as we did with the controller button bits.

What about Force Feedback?

To be honest, I was really hoping to conquer the force feedback motor control before using the SX with the PlayStation controller; sadly, every one of my attempts has failed. I have scoured the Internet for information and while there is some information out there, it is usually incomplete and not documented. What I'm going to be forced to do, I think, is rent or borrow a console and connect a logic analyzer to the PsxAttn, PsxClock, PsxCmd, and PsxData lines to see exactly what happens when the motors are activated. Unfortunately, my friend Jim doesn't have a multi-channel logic analyzer and couldn't do that for me – and it's not something that can be done with a two-channel scope; one needs to know what the console and controller are doing and in relation to each other.

I tell you what… if you have a console and are able to do that analysis for me I will send you a shiny new Parallax Professional Development Board. Here's the offer: the first person that sends me working code, or enough information that I can add working code (that is, independent motor control through the seria link) to this project wins the PDB.

Until next time – Happy Stamping!

Project Code

```
' =========================================================================
'
'   File....... PSX_Test.BS2
'   Purpose.... Test program for SX-based PSX Helper module
'   Author..... Jon Williams -- Parallax, Inc.
'   E-mail.... jwilliams@parallax.com
```

```
'   Started....
'   Updated.... 19 NOV 2005
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =========================================================================

' -----[ Program Description ]---------------------------------------------
'
' Simple test program for PlayStion Helper chip.

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

Sio             PIN     15

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

#SELECT $STAMP
  #CASE BS2, BS2E, BS2PE
    T1200       CON     813
    T2400       CON     396
    T4800       CON     188
    T9600       CON     84
    T19K2       CON     32
    TMidi       CON     12
    T38K4       CON     6
  #CASE BS2SX, BS2P
    T1200       CON     2063
    T2400       CON     1021
    T4800       CON     500
    T9600       CON     240
    T19K2       CON     110
    TMidi       CON     60
    T38K4       CON     45
  #CASE BS2PX
    T1200       CON     3313
    T2400       CON     1646
    T4800       CON     813
    T9600       CON     396
    T19K2       CON     188
    TMidi       CON     108
    T38K4       CON     84
#ENDSELECT

SevenBit        CON     $2000
Inverted        CON     $4000
Open            CON     $8000

Baud            CON     Open + T38K4
```

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

id              VAR     Byte(3)
type            VAR     Byte

psx             VAR     Byte                    ' psx data
psxThumb1       VAR     psx                     ' thumb buttons
psxThumb2       VAR     Byte                    ' thumb buttons
psxJoyRX        VAR     Byte                    ' r joystick - X axis
psxJoyRY        VAR     Byte                    ' r joystick - Y axis
psxJoyLX        VAR     Byte                    ' l joystick - X axis
psxJoyLY        VAR     Byte                    ' l joystick - Y axis

idx             VAR     Byte
xport           VAR     Byte

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

Reset:
  DEBUG CLS
  PAUSE 100

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

Main:
  DEBUG HOME

  SEROUT Sio, Baud, ["!PSX", "V"]               ' get helper version
  SERIN  Sio, Baud, [STR id\3]
  DEBUG  "PSX Helper Version = ", STR id\3, CR

  SEROUT Sio, Baud, ["!PSX", "T"]               ' get helper type (mode)
  SERIN  Sio, Baud, [type]
  DEBUG  "PSX Helper Type = ", IHEX2 type, CR, CR

  SEROUT Sio, Baud, ["!PSX", "S"]               ' get PSX status
  SERIN  Sio, Baud, [STR psx\6]

  DEBUG "Btns", TAB, BIN8 psxThumb2, BIN8 psxThumb1, CR,
        "JoyRX", TAB, DEC psxJoyRX, CLREOL, CR,
        "JoyRY", TAB, DEC psxJoyRY, CLREOL, CR,
        "JoyLX", TAB, DEC psxJoyLX, CLREOL, CR,
        "JoyLY", TAB, DEC psxJoyLY, CLREOL, CR

  GOTO Main

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

```
' =========================================================================
'
'   File...... PSX_EZ.SXB
'   Purpose... Playstation Controller Interface for the BASIC Stamp
'   Author.... Jon Williams -- Parallax, Inc.
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 19 NOV 2005
'
' =========================================================================

' -------------------------------------------------------------------------
' Program Description
' -------------------------------------------------------------------------
'
' Connects a Sony Playstation game controller to the BASIC Stamp using a
' single serial wire and the Parallax AppMod protocol.
'
' Even though the program runs at 4 MHz an external resonator must be used
' as the internal 4 MHz source is not accurate enough for serial
' communications.

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

DEVICE          SX28, OSCXT2, TURBO, STACKX, OPTIONX, BOR42
FREQ            4_000_000
ID              "PSX v0.1"

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

Sio             VAR     RA.0                    ' bi-directional serial

PsxAttn         VAR     RB.0                    ' attention
PsxClock        VAR     RB.1                    ' clock to PSX
PsxCmd          VAR     RB.2                    ' command bits to PSX
PsxData         VAR     RB.3                    ' data bits from PSX

IoPort          VAR     RC
TrisIO          VAR     TRIS_C
PlpIO           VAR     PLP_C

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

Baud            CON     "OT38400"               ' bi-directional serial
```

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

char            VAR     Byte                    ' serial I/O byte
idx             VAR     Byte                    ' loop control

psxID            VAR     Byte                    ' controller ID
psxStatus        VAR     Byte                    ' status ($5A)
psx             VAR     Byte(6)                 ' psx data
psxThumb1        VAR     psx(0)                  ' thumb buttons
psxThumb2        VAR     psx(1)                 ' thumb buttons
psxJoyRX         VAR     psx(2)                      ' r joystick - X axis
psxJoyRY         VAR     psx(3)                      ' r joystick - Y axis
psxJoyLX         VAR     psx(4)                      ' l joystick - X axis
psxJoyLY         VAR     psx(5)                      ' l joystick - Y axis

temp1           VAR     Byte                    ' subroutine work vars
temp2           VAR     Byte
temp3           VAR     Byte
temp4           VAR     Byte
temp5           VAR     Byte

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

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

WAIT_US         SUB     1, 2                    ' delay in microseconds
WAIT_MS         SUB     1, 2                    ' delay in milliseconds
RX_BYTE         SUB                             ' receive a serial byte
TX_BYTE         SUB     1                       ' transmit a serial byte
TX_OUT          SUB     1, 2                    ' transmit byte or string
READ_PSX        SUB                             ' read PSX joystick
PSX_SHIFTIO     SUB     0, 1                    ' send/get PSX byte

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

Start:
  PLP_A = %0001                                 ' configure pull-ups
  PLP_B = %00001111
  PLP_C = %00000000

  RB = %00000111                                ' initialize pins high
  TRIS_B = %11111000                            ' make outputs
```

```
Main:                                             ' wait for header
  char = RX_BYTE
  IF char <> "!" THEN Main
  char = RX_BYTE
  IF char <> "P" THEN Main
  char = RX_BYTE
  IF char <> "S" THEN Main
  char = RX_BYTE
  IF char <> "X" THEN Main

Get_Command:
  char = RX_BYTE                                  ' get command byte
  IF char = "V" THEN Show_Version
  IF char = "T" THEN Get_Type
  IF char = "S" THEN Get_Status
  IF char = "B" THEN Get_Buttons
  IF char = "J" THEN Get_Joysticks
  IF char = "C" THEN Config_IoPort
  IF char = "W" THEN Write_IoPort
  IF char = "R" THEN Read_IoPort
  GOTO Main

Show_Version:
  WAIT_MS 1                                       ' give host time to setup
  TX_OUT Version                                  ' send version string
  GOTO Main

Get_Type:
  WAIT_MS 1                                       ' give host time to setup
  READ_PSX                                        ' read PSX inputs
  TX_OUT psxID                                    ' send id byte to host
  GOTO Main

Get_Status:                                       ' returns buttons and joysticks
  WAIT_MS 1                                       ' give host time to setup
  READ_PSX                                        ' read PSX inputs
  TX_OUT psxThumb1                                ' send buttons data
  TX_OUT psxThumb2
  TX_OUT psxJoyRX                                 ' send joystick data
  TX_OUT psxJoyRY
  TX_OUT psxJoyLX
  TX_OUT psxJoyLY
  GOTO Main

Get_Buttons:
  WAIT_MS 1                                       ' give host time to setup
  READ_PSX                                        ' read PSX inputs
  TX_OUT psxThumb1                                ' send buttons data
  TX_OUT psxThumb2
  GOTO Main
```

```
Get_Joysticks:
  WAIT_MS 1                                     ' give host time to setup
  READ_PSX                                      ' read PSX inputs
  TX_OUT psxJoyRX                               ' send joystick data
  TX_OUT psxJoyRY
  TX_OUT psxJoyLX
  TX_OUT psxJoyLY
  GOTO Main

Config_IoPort:
  char = RX_BYTE                                ' get config bits
  PlpIO = char                                  ' pull-up inputs only
  char = ~char                                          ' invert bits
  TrisIO = char
  GOTO Main

Write_IoPort:
  IoPort = RX_BYTE                              ' get new port bits
  GOTO Main

Read_IoPort:
  WAIT_MS 1                                     ' give host time to setup
  TX_OUT ~IoPort                                ' send current port bits
  GOTO Main

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

' Use: WAIT_US microseconds {, multiplier}
' -- multiplier is optional

WAIT_US:
  temp1 = __PARAM1                                 ' get microseconds
  IF __PARAMCNT = 1 THEN                           ' if no multiplier
    temp2 = 1                                      '   set to 1
  ELSE                                             ' else
    temp2 = __PARAM2                               '   get multiplier
  ENDIF
  IF temp1 > 0 THEN                                ' no delay if either 0
    IF temp2 > 0 THEN
      PAUSEUS temp1 * temp2                        ' do the delay
    ENDIF
  ENDIF
  RETURN

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

' Use: WAIT_MS milliseconds {, multiplier}
' -- multiplier is optional
```

```
WAIT_MS:
  temp1 = __PARAM1                              ' get milliseconds
  IF __PARAMCNT = 1 THEN                        ' if no multiplier
    temp2 = 1                                   '   set to 1
  ELSE                                          ' else
    temp2 = __PARAM2                            '   get multiplier
  ENDIF
  IF temp1 > 0 THEN                             ' no delay if either 0
    IF temp2 > 0 THEN
      PAUSE temp1 * temp2                       ' do the delay
    ENDIF
  ENDIF
  RETURN

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

' Use: aByte = RX_BYTE
' -- receives one byte from serial I/O pin
' -- converts lowercase letters to uppercase

RX_BYTE:
  SERIN Sio, Baud, temp1
  IF temp1 >= "a" THEN                       ' lowercase?
    IF temp1 <= "z" THEN
      temp1 = temp1 - $20                    ' yes, convert to uppercase
    ENDIF
  ENDIF
  RETURN temp1                               ' return byte to caller

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

' Use: TX_BYTE aByte
' -- transmits one byte to serial I/O pin

TX_BYTE:
  temp1 = __PARAM1                              ' copy outgoing byte
  SEROUT Sio, Baud, temp1                    ' send it
  RETURN

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

' Use: TX_OUT [ byte | string | label ]
' -- "aByte" is variable or constant byte value
' -- "string" is an embedded literal string
' -- "label" is DATA statement label for stored z-String

TX_OUT:
  temp3 = __PARAM1                              ' get byte or string offset
  IF __PARAMCNT = 2 THEN
    temp4 = __PARAM2                                ' get string base
    DO
```

```
      READ temp4 + temp3, temp5               ' read a character
      IF temp5 = 0 THEN EXIT                         ' if 0, string complete
      TX_BYTE temp5                                  ' send the byte
      INC temp3                                      ' point to next character
      temp4 = temp4 + Z                       ' update base on overflow
    LOOP
  ELSE
    TX_BYTE temp3                             ' transmit the byte value
  ENDIF
  RETURN

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

' Use: READ_PSX
' -- returns PSX buttons and joystick info

READ_PSX:
  PsxAttn = 0                                 ' get controller attention
  WAIT_US 20
              PSX_SHIFTIO $01                 ' send "start"
  psxId     = PSX_SHIFTIO $42                 ' send "get data", get ID
  psxStatus = PSX_SHIFTIO                     ' get status ($5A)
  psxThumb1 = PSX_SHIFTIO                     ' read buttons
  psxThumb2 = PSX_SHIFTIO
  IF psxId = $73 THEN
    psxJoyRX  = PSX_SHIFTIO                   ' read joysticks
    psxJoyRY  = PSX_SHIFTIO
    psxJoyLX  = PSX_SHIFTIO
    psxJoyLY  = PSX_SHIFTIO
  ELSE
    psxJoyRX  = $80                           ' force to center value
    psxJoyRY  = $80
    psxJoyLX  = $80
    psxJoyLY  = $80
  ENDIF
  PsxAttn = 1                                 ' deactivate controller

  ' update buttons to make Stamp-like (active-high)

  psxThumb1 = ~psxThumb1
  psxThumb2 = ~psxThumb2

  RETURN

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

' Use: {inByte} = PSX_SHIFTIO {outByte}
' -- sends [optional] "outByte" to PSX
' -- receives [optional] "inByte" from PSX

PSX_SHIFTIO:
```

```
  IF __PARAMCNT = 1 THEN
    temp3 = __PARAM1                            ' copy output byte
  ELSE
    temp3 = 0
  ENDIF
  temp4 = 0                                     ' clear input byte
  FOR temp5 = 1 TO 8                            ' shift eight bits
    PsxCmd = temp3.0                            ' move lsb to Cmd pin
    temp3 = temp3 >> 1                          ' shift for next bit
    PsxClock = 0                                ' clock the bit
    WAIT_US 5
    temp4 = temp4 >> 1                          ' prep for next bit
    temp4.7 = PsxData                           ' get lsb from Data pin
    PsxClock = 1                                ' release clock
    WAIT_US 5
  NEXT
  RETURN temp4

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

Version:
  DATA  "0.1", 0                                ' firmware version
```