Column #139, November 2006 by Jon Williams:

Hacking the Parallax GPS Module

In September we talked about exciting new updates in the SX/B compiler and now were going to put a few of them to use with a cool new GPS product from Parallax. This isn't just another GPS module, it was specifically designed to be hacker friendly. How? Well, it uses an SX20 and the firmware was written in SX/B – and you can download this code from Parallax. Better, still, is the addition of several nondescript pads on the PDB; these pads give us access to I/O pins on the SX20, and with a little effort, the ability to reprogram the module with custom firmware.

I've never considered myself much of a hacker, but with the Parallax GPS module I couldn't pass up the opportunity to give it a go – especially since the module is just begging to be hacked! Again, that's by design, and it should come as no surprise that Parallax teamed up with one of the best known hackers in the western world, Joe Grand of Grand Idea Studio.

Many BASIC Stamp users know of Joe through his collaboration with Parallax on the RFID Reader. When Joe happened across a neat little GPS receiver module from Polstar, he showed it to Parallax and a new project was born. GPS is becoming increasingly more available and popular with experimenters, and this is especially true in hobbyist robotics as evidenced by the recent "mini DARPA challenge" competitions sponsored by robotics clubs. The Parallax GPS module makes things very easy – and the size and form-factor work well, too. Figure 139.1 shows the Parallax GPS next to a Garmin eTrex unit that we've used in previous experiments.

Figure 139.1: The Parallax GPS Module (left) and the Garmin eTrex GPS Unit (right)

If you look at the schematic in Figure 139.2 you'll see that the design is quite neat and tidy: the major components are a Polstar PMB-248 GPS receiver (connected to J1), the SX20, and an analog switch. The insertion of the analog switch is particularly clever on Joe's part; this allows the module to reroute the serial output from the GPS receiver directly to the SIO pin when the RAW pin is pulled low. In normal ("smart") operation, this pin is left to be pulled high and the SX20 processes the GPS data for us.

What I like about the module is that the GPS receiver spits out NMEA string every second. To test the output I configured the module for raw mode and fed the SIO pin to HyperTerminal through a MAX232 level shifter. Figure 139.3 shows the NMEA sentences provided by the Polstar GPS module. Like common GPS modules, the baud rate is 4800.

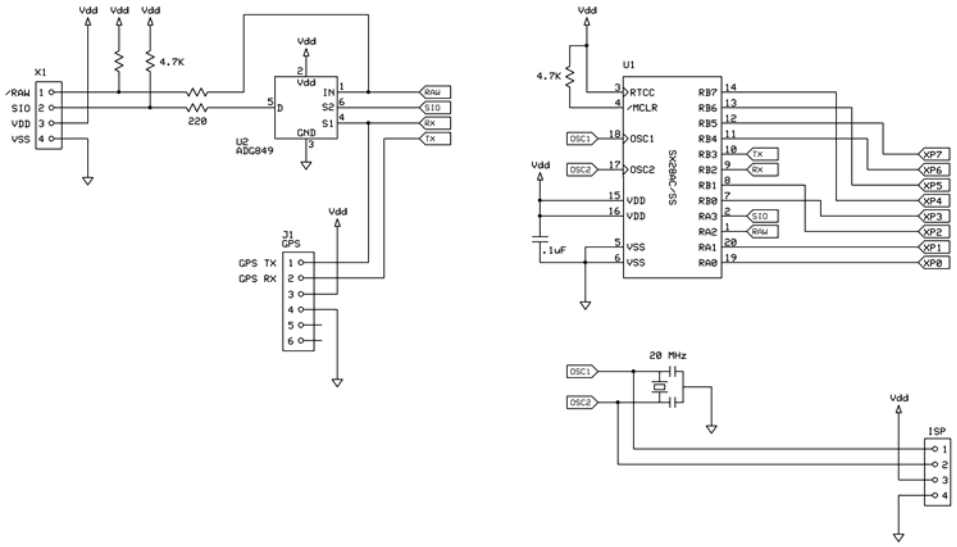

Figure 139.2: Parallax GPS Module Schematic

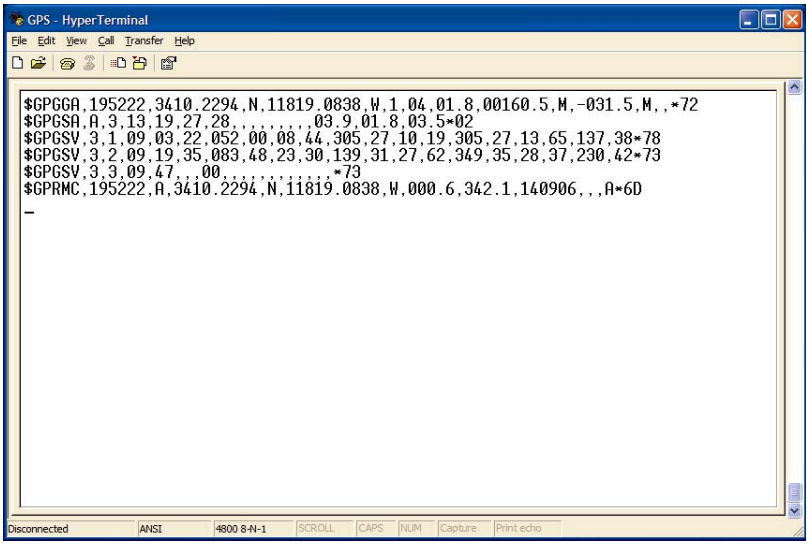

Figure 139.3: GPS Output in NMEA Sentence Format

Custom GPS

Okay… why customize the module when it works so well? For me, there were a few reasons: First, it would be fun and educational. Second, I could take advantage of the extra SX20 I/O pins and, finally, I could bump up the communication speed to make it compatible with the Parallax Servo Controller. It seems to me that the GPS module and PSC make a good robotics pair, so making the GPS module more compatible is a good idea. And in the end, even if you don't decide to customize your GPS module, the code we'll develop here can be used for other AppMod-compatible devices.

Some time back Parallax developed a simple serial communications protocol for its AppMods; devices that connected to the BOE's AppMod header. The protocol uses Open-True serial mode – this is critical. True mode means that the idle state of the serial line is high, and Open True means that the idle (0) state is accomplished with a pull-up. When a transmitting device wants to exert a "1" bit it pulls the line low. This configuration allows several devices to be bussed together without fear of electrical conflicts as the serial line is never driven high.

By tradition, the AppMod protocol starts with a "!" character, followed by a two- or three-character device ID string. The original intent of the "!" character was to allow devices to determine the baud rate of the incoming stream. We're not going to do that here as auto-baud programming can be quite messy, and our goal is to have fun. Since the likely host of the GPS module will be a BS2-type processor, we're going to fix the baud rate to 38.4K; this matches the high-speed mode of the PSC, and allows the host program to have a single baud constant for both devices.

Let's jump in, shall we? After the obligatory definitions, our program starts as follows:

```
Main:
  IF Raw = 0 THEN Main

Wait_For_Header:
  char = RXBYTE
  IF char <> "!" THEN Wait_For_Header
  char = RXBYTE ToUpper
  IF char <> "G" THEN Wait_For_Header
  char = RXBYTE ToUpper
  IF char <> "P" THEN Wait_For_Header
  char = RXBYTE ToUpper
  IF char <> "S" THEN Wait_For_Header

Get_Cmd:
  char = RXBYTE ToUpper
  IF char = "I" THEN Get_ID
  IF char = "V" THEN Check_Valid
```

```
  IF char = "U" THEN Set_Time_Offset
  IF char = "T" THEN Get_Time
  IF char = "L" THEN Get_Lat
  IF char = "O" THEN Get_Long
  IF char = "K" THEN Get_Knots
  IF char = "H" THEN Get_Heading
  IF char = "X" THEN X_Port
  GOTO Main
```

The first thing we need to do is check the start of the Raw input pin (RA.2). This pin is normally pulled high to enable "smart" mode, i.e., intervention by the SX20. When connected to ground, this signal reroutes the GPS output directly to the SIO pin using the analog switch. Since the SX20 is no longer connected to SIO we simply loop at Main until Raw returns high. Normally, one will select "raw" or "smart" mode via hardware, yet this is a standard input and could be controlled dynamically by the host processor. Just keep in mind that raw mode output is at 4800 baud, and smart mode serial I/O is at 38.4 kBaud.

Assuming we're in smart mode the program jumps done to Wait_For_Header where we do just that: we wait on the "!GPS" header that precedes a command or request from the host. From a PBASIC standpoint this may look a little clumsy but this is, essentially, how the PBASIC WAIT modifier operates. And if you look at the compiled output you'll see that it uses very little code. Note that we've wrapped the SERIN function in the RXBYTE subroutine to conserve program space – this is always a good idea for commands that generate a lot of assembly code, and we'll see more examples of that in our program.

After the proper header has arrived we pull the next byte from the stream and compare it to a list of valid commands. If there is a match we will jump to the code that handles that command, otherwise we'll branch back to the top and wait for another header/command sequence. I chose the IF-THEN style for this section because I find it easy to update and modify. SX/B now includes an ON-GOTO structure that some programmers will find a bit more elegant.

Let's work through a few of the commands – we don't have to go into detail about all because you'll see that they take advantage of some core subroutines, and those routines I think you'll find useful in other SX/B programs.

My version of the "smart" GPS module behaves a little differently than the Parallax program in that it buffers the GPS stream (like we did with the scratchpad RAM in the old BS2p GPS project), and then pulls data from it instead of waiting on a new stream for each command. What this means is that we need to get a valid stream before we ask for any other data, and this is the purpose of the Check_Valid (command = "V") code:

```
Check_Valid:
  GETRMC
  FINDFIELD 1
  char = GETBUF bufPntr
  DELAYMS 15
  TXBYTE char
  GOTO Main
```

What we're going to see is that this little section of code has a whole lotta stuff going on behind it. First things first: we need to receive and buffer the $GPRMC sentence from the receiver, and that is the purpose of the GETRMC subroutine.

```
GETRMC:
  char = GPSRX
  IF char <> "$" THEN GETRMC
  char = GPSRX
  IF char <> "G" THEN GETRMC
  char = GPSRX
  IF char <> "P" THEN GETRMC
  char = GPSRX
  IF char <> "R" THEN GETRMC
  char = GPSRX
  IF char <> "M" THEN GETRMC
  char = GPSRX
  IF char <> "C" THEN GETRMC
  char = GPSRX
  IF char <> "," THEN GETRMC

  FOR bufPntr = 0 TO 63
    char = GPSRX
    PUTBUF bufPntr, char
    IF char = "*" THEN EXIT
  NEXT
  RETURN
```

Note that the top part of this routine works just like the code that waits on the command header, the difference being that we're now getting characters from the GPS receiver at 4800 baud, hence the GPSRX subroutine. Once the front part of the $GPRMC sentence has been received we will put the rest of the characters (up to the checksum) into a 64-byte buffer.

Wait a minute… we can only have 16-byte arrays with the SX20, so how do we get a 64-byte buffer? Well, we build it out of four 16-byte arrays and a little code. I originally wrote this buffer code for the Parallax Inkjet module project and have found it quite useful. First, let's look at the variable assignments:

```
bufPntr     VAR        Byte
bufA           VAR     Byte (16)
```

```
bufB            VAR     Byte (16)
bufC            VAR     Byte (16)
bufD            VAR     Byte (16)
```

As you can see, we've created four, 16-byte arrays that we'll concatenate with a little code. Let's start with putting a byte into the buffer. This works very much like the BS2p's PUT instruction that requires an address and the byte value to write.

```
PUTBUF:
  tmpB1 = __PARAM1
  tmpB2 = __PARAM2
  tmpB3 = tmpB1 & %00111111
  tmpB3 = tmpB3 >> 4
  tmpB4 = tmpB1 & %00001111

  IF tmpB3 = %00 THEN
    bufA(tmpB4) = tmpB2
  ENDIF
  IF tmpB3 = %01 THEN
    bufB(tmpB4) = tmpB2
  ENDIF
  IF tmpB3 = %10 THEN
    bufC(tmpB4) = tmpB2
  ENDIF
  IF tmpB3 = %11 THEN
    bufD(tmpB4) = tmpB2
  ENDIF
  RETURN
```

This routine starts by collecting the buffer address (passed in __PARAM1) and the data byte (passed in __PARAM2) and then creating the array and index pointers for the buffer. The array pointer is derived by shifting the address right four bits (we're extracting the high nibble), and the index within the array comes from the low nibble of the address.

This is a useful technique so let's talk about it. What if we needed a 24-byte buffer? The trick is to keep the array sizes equal, and use an even power of two. For a 24-byte buffer I would use three, 8-byte arrays. This fits neatly within the SX's banked RAM space and allows the techniques used above for addressing, the difference being we'll shift the address right three bits, and use the lower three bits of the address as the index within each array.

Okay, we have the $GPRMC string buffered, but we don't know if the string was valid. There is a character in the string that gives us this information, and it is in the field that follows the time. To locate a field within the string we can use the FINDFIELD subroutine. Its purpose is to count commas until the desired position is located (and then stored in bufPntr).

```
FINDFIELD:
  tmpB5 = __PARAM1

  bufPntr = 0
  IF tmpB5 > 0 THEN
    DO
      char = GETBUF bufPntr
      INC bufPntr
      IF char = "," THEN
        DEC tmpB5
        IF tmpB5 = 0 THEN EXIT
      ENDIF
    LOOP
  ENDIF
  IF bufPntr >= 64 THEN
    bufPntr = 0
  ENDIF
  RETURN
```

For this routine to work we must pass a field number that is greater than zero. Then we start at the beginning of the buffer, pulling characters from it with GETBUF. For each comma found the field number is decremented and when we reach zero we're sitting at the start of the desired field. I put a little error trapping in the code to prevent the pointer from being incremented outside the bounds of the buffer.

Back to Check_Valid (yes, we're still working on that). Now that we're pointing at the validity character in the GPS string we can pull it and send it back to the host. It's a good idea to insert a short delay (I used 15 milliseconds) before sending anything back to the host. This allows the host to be ready with its SERIN instruction. BASIC Stamps don't take that long to setup, but if the host is a Javelin Stamp using one pin for serial input and output, it takes a little extra time.

As you can see, we've done quite a bit of background work to support the commands required for the "smart" mode of the module – and there's more! One of the useful features of GPS is getting accurate time, so let's explore that as there are support routines used here that will nicely migrate to other applications.

```
Get_Time:
  DELAYMS 15
  result = BUF2DEC 0, 2
  result_LSB = result_LSB + utcOffset
  IF result_LSB > 23 THEN
    result_LSB = result_LSB - 24
  ENDIF
  TXBYTE result_LSB
```

```
  result = BUF2DEC 2, 2
  TXBYTE result_LSB
  result = BUF2DEC 4, 2
  TXBYTE result_LSB
```

When we have a valid $GPRMC string the first six characters indicate the current time as hhmmss. Remember that these are text characters, so we're forced to do a numeric conversion. I created a little subroutine called BUF2DEC to do just that. What this does is lets us point to a number within the buffer, specify how wide it is, and get back the decimal value – somewhat analogous to the PBASIC DEC modifier when used with SERIN.

```
BUF2DEC:
  bufPntr = __PARAM1
  tmpB5 = __PARAM2
  tmpW3 = 0
  IF tmpB5 >= 1 THEN
    IF tmpB5 <= 5 THEN
      DO
        tmpW3 = MULT tmpW3, 10
        tmpB6 = GETBUF bufPntr
        tmpB6 = tmpB6 - "0"
        tmpW3 = tmpW3 + tmpB6
        INC bufPntr
        DEC tmpB5
      LOOP UNTIL tmpB5 = 0
    ENDIF
  ENDIF
  RETURN tmpW3
```

As I stated above we need to pass the location within the buffer (__PARAM1) and the width of the number (__PARAM2). If the width is legal then a loop construct iterates through the field width, doing a decimal shift left (multiply by 10) and adding in the next digit.

Multiplication is another instruction that generates a lot of assembly code and it's best to encapsulate within a subroutine to conserve code space. We do that in the MULT code as shown below.

```
MULT:
  IF __PARAMCNT = 2 THEN
    tmpW1 = __PARAM1
    tmpW2 = __PARAM2
  ENDIF
  IF __PARAMCNT = 3 THEN
    tmpW1 = __WPARAM12
    tmpW2 = __PARAM3
  ENDIF
  IF __PARAMCNT = 4 THEN
```

```
    tmpW1 = __WPARAM12
    tmpW2 = __WPARAM34
  ENDIF

  tmpW1 = tmpW1 * tmpW2
  RETURN tmpW1
```

Now, there's nothing magic here but I'm including it to illustrate a clean method for dealing with mixed values. We can determine what got passed to the subroutine via __PARAMCNT. When this has a value of 3 we know that we're mixed. I tend to like to pass Word, then Byte when I used mixed values. Note that the first parameter (the word) is passed by the compiler via __WPARAM12, and the second parameter (the byte) is passed through __PARAM3.

Can you do it the other way, i.e., byte value first? Yes, just change the middle section like this:

```
  IF __PARAMCNT = 3 THEN
    tmpW1 = __PARAM1
    tmpW2 = __WPARAM23
  ENDIF
```

It doesn't matter which you use, just pick a strategy and stick with it so you can port your subroutines from program to program without trouble.

To finish with the time the local UTC offset is added to the hours and corrected to keep the value 0 to 23. We can send the UTC offset to the module with the "U" command; this lets us get localized time back and not have to deal with that on the host side. The UTC offset is always positive, so to handle a negative offset (as in -8 hours for Hollywood, CA) we will add that value to 24.

As you look through the full listing you'll see that the majority of the commands work like Get_Time, plucking requested values from the GPS string and sending them back to the host.

Remapping Bits

At the top I started by saying the GPS module was hacker friendly, and it is, but it's not entirely hacker convenient. The realities of PCB design sometimes get us – and they do with the extra "hacker pins" on the module. Figure 139.4 shows the bottom of the Parallax GPS module. On the left is the host connection port, the white connector at the lower left goes to the GPS receiver, and you can clearly see the SX20, the "hacker pads," and the pads for reprogramming using an SX-Key or SX-Blitz.

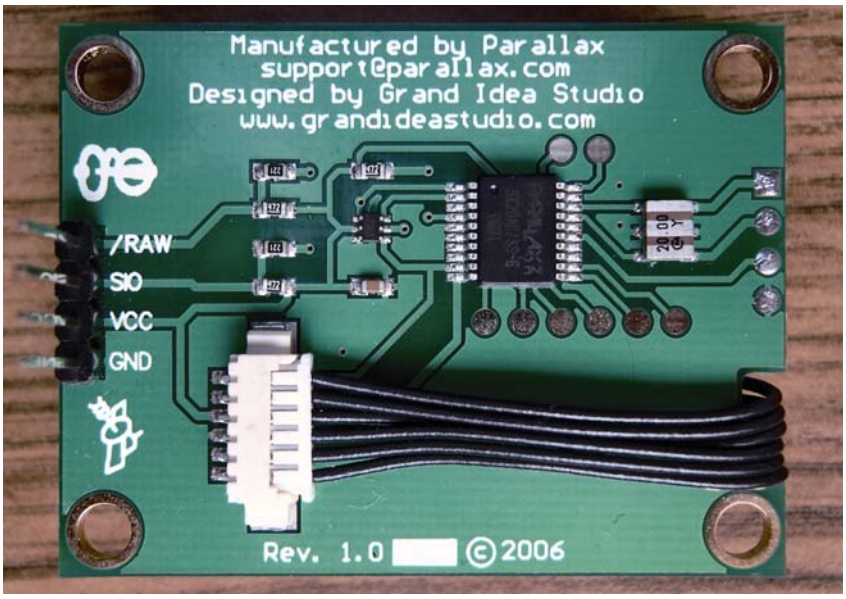

Figure 139.4: The Parallax GPS Module, Bottom View

There are eight "hacker pads" and in an ideal world these would have been mapped to port RB on the SX20. In the real world, however, they're not. My guess is that his would have complicated the PCB. No problem, we'll "fix it in software" (as a product manager I always hated hearing that phrase, and I use it only teasingly – nothing is broken).

This "problem" actually gives us an opportunity to explore a bit more of SX/B. In the SX20 and SX28, the data direction (TRIS) registers are not readable, they can only be written. In the BASIC Stamp and in SX/B, this limitation is dealt with by creating shadow registers of the current TRIS states. When we use a command that affects a pin's I/O direction this register is read, modified, and then written back to the associated TRIS register for that pin. We're going to take advantage of this "shadow register" to make those eight "hacker pins" look like one contiguous port.

On receipt of an "X" command to the GPS module the program jumps to a section that processes one of three secondary commands for the port: "S" for setup (set TRIS), "W" for write bits, and "R" for read bits. Before we get to that, let's look at the individual pin assignments for what I'm calling the xport:

```
XP0             PIN     RA.0
XP1             PIN     RA.1
XP2             PIN     RB.1
XP3             PIN     RB.0
XP4             PIN     RB.7
XP5             PIN     RB.6
XP6             PIN     RB.4
XP7             PIN     RB.5
```

I know from the assignments above that it seems the bits are all over the place. Figure 139.5 shows the remapping of the xport bits – you can see the organization is set up to make things easy to remember.

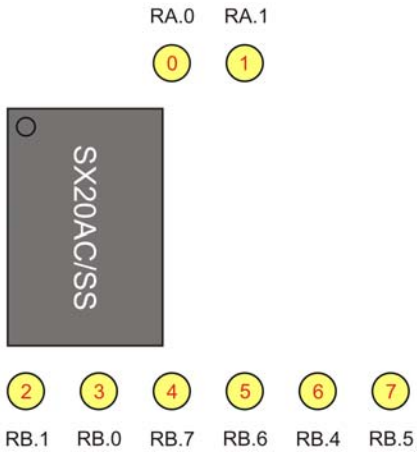

Figure 139.5: Remapping of the XPORT Bits

The "XS" command sequence lets us set up the xport pins like any other SX port, and we will use SX conventions, that is, a "0" bit indicates an output, a "1" bit indicates an input. Here's the code:

```
SETUPXP:
  tmpB1 = __PARAM1
  tmpB2 = TRIS_A
  tmpB2.0 = tmpB1.0
  tmpB2.1 = tmpB1.1
  TRIS_A = tmpB2
  tmpB2 = TRIS_B
```

```
  tmpB2.1 = tmpB1.2
  tmpB2.0 = tmpB1.3
  tmpB2.7 = tmpB1.4
  tmpB2.6 = tmpB1.5
  tmpB2.4 = tmpB1.6
  tmpB2.5 = tmpB1.7
  TRIS_B = tmpB2
  RETURN
```

As you can see there are really two sections to this code, one for each of the SX ports (RA and RB). What we have to do is get a copy of the TRIS shadow register, modify the appropriate bits without touching the others, and then write it back. This is in fact what a lot of SX/B functions do when there is a necessary I/O state for an instruction. This method ensures that the port pins not associated with our xport are not adversely affected.

Okay, now that the port bits are setup the write and read methods are downright trivial.

```
WRXPORT:
  tmpB1 = __PARAM1
  XP0 = tmpB1.0
  XP1 = tmpB1.1
  XP2 = tmpB1.2
  XP3 = tmpB1.3
  XP4 = tmpB1.4
  XP5 = tmpB1.5
  XP6 = tmpB1.6
  XP7 = tmpB1.7
  RETURN

RDXPORT:
  tmpB1.0 = XP0
  tmpB1.1 = XP1
  tmpB1.2 = XP2
  tmpB1.3 = XP3
  tmpB1.4 = XP4
  tmpB1.5 = XP5
  tmpB1.6 = XP6
  tmpB1.7 = XP7
  RETURN tmpB1
```

Yes, the funky bit mapping forces us to do things a bit at a time, but keep in mind that the SX20 on the module is running at 20 MHz – this is pretty zippy and it the transfer of bits happens in about two microseconds.

I think that's about enough fun for this month, don't you? Even if you don't hack the Parallax GPS module, do give it a try as it's small, neat, and works very nicely. Remember, its raw mode output lets us use it like any other GPS, so we can quickly port old programs to it.

Until next time, Happy Thanksgiving and Happy Stamping!

Additional Resources

Grand Idea Studio
www.grandideastudio.com

Project Code

```
' =============================================================================
'
'   File..... GPS_JW.SXB
'   Purpose... Custom Parallax GPS Receiver Control Code
'   Author.... Jon Williams
'   E-mail.... jwilliams@efx-tek.com
'   Started...
'   Updated... 05 NOV 2006
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Note: While Parallax included "hacker friendly" features into their GPS
' receiver module, hacks are not officially supported and you do so at
' your own peril.


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

'{$DEFINE USE_SX28_OFF}                              ' use SX28 on PDB


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

'{$IFDEF USE_SX28}
DEVICE          SX28, OSCXT2, TURBO, STACKX, OPTIONX, BOR42
'{$ELSE}
```

```
DEVICE          SX20, OSCXT2, TURBO, STACKX, OPTIONX, BOR42
'{$ENDIF}

FREQ            20_000_000
ID              "GPS JW02"


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

Raw             PIN     RA.2                    ' data I/O mode
Sio             PIN     RA.3                    ' to/from host
RX              PIN     RB.2                    ' from GPS
TX              PIN     RB.3                    ' to GPS

XP0             PIN     RA.0                    ' hacker port pins
XP1             PIN     RA.1
XP2             PIN     RB.1
XP3             PIN     RB.0
XP4             PIN     RB.7
XP5             PIN     RB.6
XP6             PIN     RB.4
XP7             PIN     RB.5

'{$IFDEF USE_SX28}
UnusedC         PIN     RC      INPUT PULLUP
'{$ENDIF}


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

GpsBaud         CON     "T4800"                 ' to/from GPS module
HostBaud        CON     "OT38400"               ' match Parallax PSC

ToUpper         CON     1


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

idx             VAR     Byte
char            VAR     Byte
utcOffset       VAR     Byte
result          VAR     Word

bufPntr         VAR     Byte                    ' pointer for GPS buffer
bufA            VAR     Byte (16)               ' GPS buffer
```

```
bufB            VAR     Byte (16)
bufC            VAR     Byte (16)
bufD            VAR     Byte (16)

tmpB1           VAR     Byte                    ' subroutine vars
tmpB2           VAR     Byte
tmpB3           VAR     Byte
tmpB4           VAR     Byte
tmpB5           VAR     Byte
tmpB6           VAR     Byte
tmpW1           VAR     Word
tmpW2           VAR     Word
tmpW3           VAR     Word


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


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

RXBYTE          FUNC    1, 0, 1                 ' get byte from host
GPSRX           FUNC    1, 0                    ' get byte from GPS
GETBUF          FUNC    1, 1                    ' get byte from buffer
RDXPORT         FUNC    1, 0                    ' read bits from X port
MULT            FUNC    2, 2, 4                 ' 16-bit multiplication
DIV             FUNC    2, 2, 4                 ' 16-bit division
BUF2DEC         FUNC    2, 2                    ' pull dec nums from buffer

DELAYUS         SUB     1, 2                    ' pause in microseconds
DELAYMS         SUB     1, 2                    ' pause in milliseconds
TXBYTE          SUB     1                       ' send byte to host
GETRMC          SUB     0                       ' get fresh GPRMC scan
FINDFIELD       SUB     1                       ' move bufPntr to field
GPSTX           SUB     1                       ' send byte to GPS
PUTBUF          SUB     2                       ' put byte into buffer
SETUPXP         SUB     1                       ' setup TRIS bits for X port
WRXPORT         SUB     1                       ' write bits to X port


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

Start:
  ' additional start-up items here

Main:
```

```
  IF Raw = 0 THEN Main                          ' hold if raw mode

Wait_For_Header:                                ' wait for !GPS
  char = RXBYTE
  IF char <> "!" THEN Wait_For_Header
  char = RXBYTE ToUpper
  IF char <> "G" THEN Wait_For_Header
  char = RXBYTE ToUpper
  IF char <> "P" THEN Wait_For_Header
  char = RXBYTE ToUpper
  IF char <> "S" THEN Wait_For_Header

Get_Cmd:
  char = RXBYTE ToUpper                         ' get command byte
  IF char = "I" THEN Get_ID
  IF char = "V" THEN Check_Valid
  IF char = "U" THEN Set_Time_Offset
  IF char = "T" THEN Get_Time
  IF char = "L" THEN Get_Lat
  IF char = "O" THEN Get_Long
  IF char = "K" THEN Get_Knots
  IF char = "H" THEN Get_Heading
  IF char = "X" THEN X_Port
  GOTO Main


Get_ID:
  DELAYMS 15
  FOR idx = 0 TO 2
    READ Pgm_ID + idx, char                     ' read a character
    TXBYTE char                                 ' no, send the char
  NEXT
  GOTO Main


Check_Valid:
  GETRMC                                        ' get fresh data from GPS
  FINDFIELD 1                                   ' move pointer
  char = GETBUF bufPntr                         ' read status character
  DELAYMS 15                                    ' let host get ready
  TXBYTE char                                   ' send status char to host
  GOTO Main


Set_Time_Offset:
  char = RXBYTE                                 ' get offset value
  IF char < 24 THEN                             ' valid?
    utcOffset = char                            ' yes, save it
  ENDIF
  GOTO Main
```

```
Get_Time:
  DELAYMS 15                                ' let host get ready
  result = BUF2DEC 0, 2                     ' hours, two digits
  result_LSB = result_LSB + utcOffset       ' localize time
  IF result_LSB > 23 THEN
    result_LSB = result_LSB - 24
  ENDIF
  TXBYTE result_LSB                         ' send localized hours
  result = BUF2DEC 2, 2                     ' mins, two digits
  TXBYTE result_LSB                         ' send minutes
  result = BUF2DEC 4, 2                     ' secs, two digits
  TXBYTE result_LSB                         ' send seconds


Get_Lat:
  DELAYMS 15
  result = BUF2DEC 9, 2                     ' degrees
  TXBYTE result_LSB
  result = BUF2DEC 11, 2                    ' mins
  TXBYTE result_LSB
  result = BUF2DEC 14, 3                    ' fractional minutes
  result = MULT result, 60                  ' convert to seconds
  result = DIV result, 1000
  TXBYTE result_LSB                         ' send to host
  TXBYTE result_MSB
  char = GETBUF 19                          ' get direction character
  TXBYTE char
  GOTO Main


Get_Long:
  DELAYMS 15
  result = BUF2DEC 21, 3                    ' degrees
  TXBYTE result_LSB
  result = BUF2DEC 24, 2                    ' mins
  TXBYTE result_LSB
  result = BUF2DEC 27, 3                    ' fractional minutes
  result = MULT result, 60                  ' convert to seconds
  result = DIV result, 1000
  TXBYTE result_LSB                         ' send to host
  TXBYTE result_MSB
  char = GETBUF 32                          ' get direction character
  TXBYTE char
  GOTO Main


Get_Knots:
  DELAYMS 15
  result = BUF2DEC 34, 3                    ' get whole knots
  result = MULT result, 10                  ' convert to tenths
```

```
  char = BUF2DEC 38, 1                          ' get 0.1 knots
  result = result + char                        ' add to result
  TXBYTE result_LSB                             ' send to host
  TXBYTE result_MSB
  GOTO Main


Get_Heading:
  DELAYMS 15
  result = BUF2DEC 40, 3                        ' get whole degrees
  result = MULT result, 10                      ' convert to tenths
  char = BUF2DEC 44, 1                          ' get 0.1 degrees
  result = result + char                        ' add to result
  TXBYTE result_LSB                             ' send to host
  TXBYTE result_MSB
  GOTO Main


X_Port:
  char = RXBYTE ToUpper
  IF char = "S" THEN X_Port_Setup
  IF char = "W" THEN X_Port_Write
  IF char = "R" THEN X_Port_Read
  GOTO Main

X_Port_Setup:
  char = RXBYTE                                 ' get setup bits
  SETUPXP char                                  ' write to TRIS regs
  GOTO Main

X_Port_Write:
  char = RXBYTE                                 ' get new outputs
  WRXPORT char                                  ' write them
  GOTO Main

X_Port_Read:
  char = RDXPORT                                ' read xport pins
  DELAYMS 15                                    ' let host get ready
  TXBYTE char
  GOTO Main


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

' Use: theByte = RXBYTE { ConvertToUppercase }
' -- optional parameter (1) converts byte to upper case if "a".."z"

RXBYTE:
  IF __PARAMCNT = 1 THEN                        ' option specified
```

```
    tmpB2 = __PARAM1                          ' yes, save it
  ELSE
    tmpB2 = 0                                 ' no, set to default
  ENDIF
  SERIN Sio, HostBaud, tmpB1
  IF tmpB2.0 = ToUpper THEN
    IF tmpB1 >= "a" THEN                      ' lowercase?
      IF tmpB1 <= "z" THEN
        tmpB1.5 = 0                           ' ...yes, make uppercase
      ENDIF
    ENDIF
  ENDIF
  RETURN tmpB1

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

' Use: TXBYTE theByte

TXBYTE:
  tmpB1 = __PARAM1                            ' get byte to send
  SEROUT Sio, HostBaud, tmpB1
  DELAYUS 52                                  ' 2 extra stop bits
  RETURN

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

' Use: DELAYUS usecs

DELAYUS:
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                            ' save byte value
  ELSE
    tmpW1 = __WPARAM12                          ' save word value
  ENDIF
  PAUSEUS tmpW1
  RETURN

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

' Use: DELAYMS msecs

DELAYMS:
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                            ' save byte value
  ELSE
    tmpW1 = __WPARAM12                          ' save word value
  ENDIF
  PAUSE tmpW1
  RETURN

' -----------------------------------------------------------------------------
```

```
' Waits for and buffers GPRMC string

GETRMC:
  char = GPSRX                                  ' wait for $GPRMC
  IF char <> "$" THEN GETRMC
  char = GPSRX
  IF char <> "G" THEN GETRMC
  char = GPSRX
  IF char <> "P" THEN GETRMC
  char = GPSRX
  IF char <> "R" THEN GETRMC
  char = GPSRX
  IF char <> "M" THEN GETRMC
  char = GPSRX
  IF char <> "C" THEN GETRMC
  char = GPSRX                                  ' remove leading comma
  IF char <> "," THEN GETRMC

  FOR bufPntr = 0 TO 63
    char = GPSRX
    PUTBUF bufPntr, char
    IF char = "*" THEN EXIT                     ' end of data
  NEXT
  RETURN

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

' Use: PUTBUF pntr, byteVal
' -- stores "byteVal" at "pntr"

PUTBUF:
  tmpB1 = __PARAM1                              ' get pointer
  tmpB2 = __PARAM2                              ' byte to store
  tmpB3 = tmpB1 & %00111111                     ' keep legal
  tmpB3 = tmpB3 >> 4                            ' point to array
  tmpB4 = tmpB1 & %00001111                     ' index in the array

  IF tmpB3 = %00 THEN
    bufA(tmpB4) = tmpB2
  ENDIF
  IF tmpB3 = %01 THEN
    bufB(tmpB4) = tmpB2
  ENDIF
  IF tmpB3 = %10 THEN
    bufC(tmpB4) = tmpB2
  ENDIF
  IF tmpB3 = %11 THEN
    bufD(tmpB4) = tmpB2
  ENDIF
  RETURN
```

```
' -----------------------------------------------------------------------------

' Use: byteVal = GETBUF pntr

GETBUF:
  tmpB1 = __PARAM1                          ' get pointer
  tmpB3 = tmpB1 & %00111111                 ' keep legal
  tmpB3 = tmpB3 >> 4                        ' point to array
  tmpB4 = tmpB1 & %00001111                 ' index in the array

  IF tmpB3 = %00 THEN
    tmpB2 = bufA(tmpB4)
  ENDIF
  IF tmpB3 = %01 THEN
    tmpB2 = bufB(tmpB4)
  ENDIF
  IF tmpB3 = %10 THEN
    tmpB2 = bufC(tmpB4)
  ENDIF
  IF tmpB3 = %11 THEN
    tmpB2 = bufD(tmpB4)
  ENDIF
  RETURN tmpB2

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

' Use: FINDFIELD

FINDFIELD:
  tmpB5 = __PARAM1                          ' field number

  bufPntr = 0                               ' start at beginning
  IF tmpB5 > 0 THEN                         ' do we need to look
    DO
      char = GETBUF bufPntr                 ' read char from buf
      INC bufPntr                           ' point to next
      IF char = "," THEN                    ' this char a comma?
        DEC tmpB5                           ' yes, count down
        IF tmpB5 = 0 THEN EXIT              ' are we there?
      ENDIF
    LOOP
  ENDIF
  IF bufPntr >= 64 THEN                     ' trap error
    bufPntr = 0
  ENDIF
  RETURN

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

' Use: theByte = GPSRX
```

```
GPSRX:
  SERIN RX, GpsBaud, tmpB1                    ' get byte from PolStar
  RETURN tmpB1

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

' Use: GPSTX theByte

GPSTX:
  tmpB1 = __PARAM1                            ' save byte
  SEROUT TX, GpsBaud, tmpB1                   ' send byte to PolStar
  RETURN

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

' Use: SETUP_XP trisBits
' -- setup TRIS bits for hacker expansion port

SETUPXP:
  tmpB1 = __PARAM1                            ' save new setup
  tmpB2 = TRIS_A                              ' get current TRISA copy
  tmpB2.0 = tmpB1.0                           ' RA.0 (XP0)
  tmpB2.1 = tmpB1.1                           ' RA.1 (XP1)
  TRIS_A = tmpB2
  tmpB2 = TRIS_B                              ' get current TRISB copy
  tmpB2.1 = tmpB1.2                           ' RB.1 (XP2)
  tmpB2.0 = tmpB1.3                           ' RB.2 (XP3)
  tmpB2.7 = tmpB1.4                           ' RB.7 (XP4)
  tmpB2.6 = tmpB1.5                           ' RB.6 (XP5)
  tmpB2.4 = tmpB1.6                           ' RB.4 (XP6)
  tmpB2.5 = tmpB1.7                           ' RB.5 (XP7)
  TRIS_B = tmpB2
  RETURN

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

' Use: WR_XPORT byteVal
' -- writes "byteVal" to hacker expansion port

WRXPORT:
  tmpB1 = __PARAM1
  XP0 = tmpB1.0                               ' move bits to x-port
  XP1 = tmpB1.1
  XP2 = tmpB1.2
  XP3 = tmpB1.3
  XP4 = tmpB1.4
  XP5 = tmpB1.5
  XP6 = tmpB1.6
  XP7 = tmpB1.7
  RETURN
```

```
' -----------------------------------------------------------------------------

' Use: byteVal = RDXPORT

RDXPORT:
  tmpB1.0 = XP0                                ' get bits from x-port
  tmpB1.1 = XP1
  tmpB1.2 = XP2
  tmpB1.3 = XP3
  tmpB1.4 = XP4
  tmpB1.5 = XP5
  tmpB1.6 = XP6
  tmpB1.7 = XP7
  RETURN tmpB1                                 ' return to caller

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

' Use: MULT val1, val2
' -- if mixed (byte and word), word must be specified first

MULT:
  IF __PARAMCNT = 2 THEN
    tmpW1 = __PARAM1
    tmpW2 = __PARAM2
  ENDIF
  IF __PARAMCNT = 3 THEN
    tmpW1 = __WPARAM12
    tmpW2 = __PARAM3
  ENDIF
  IF __PARAMCNT = 4 THEN
    tmpW1 = __WPARAM12
    tmpW2 = __WPARAM34
  ENDIF

  tmpW1 = tmpW1 * tmpW2
  RETURN tmpW1

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

' Use: DIV val1, val2
' -- if mixed (byte and word), word must be specified first

DIV:
  IF __PARAMCNT = 2 THEN
    tmpW1 = __PARAM1
    tmpW2 = __PARAM2
  ENDIF
  IF __PARAMCNT = 3 THEN
    tmpW1 = __WPARAM12
    tmpW2 = __PARAM3
```

```
  ENDIF
  IF __PARAMCNT = 4 THEN
    tmpW1 = __WPARAM12
    tmpW2 = __WPARAM34
  ENDIF

  tmpW1 = tmpW1 / tmpW2
  RETURN tmpW1

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

' Use: wordVal = BUF2DEC start, width
' -- converts "width" chars at "start" to word

BUF2DEC:
  bufPntr = __PARAM1                          ' start of field
  tmpB5 = __PARAM2                            ' number of digits
  tmpW3 = 0                                   ' result output
  IF tmpB5 >= 1 THEN
    IF tmpB5 <= 5 THEN
      DO
        tmpW3 = MULT tmpW3, 10
        tmpB6 = GETBUF bufPntr
        tmpB6 = tmpB6 - "0"
        tmpW3 = tmpW3 + tmpB6
        INC bufPntr
        DEC tmpB5
      LOOP UNTIL tmpB5 = 0
    ENDIF
  ENDIF
  RETURN tmpW3


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

Pgm_ID:
  DATA  "0.2", 0
```