Column #103 November 2003 by Jon Williams:

Stamping on Down the Road

Summertime is vacation time, and I – like so many others around the world – hit the road for a few days out of the office for some fun and relaxation. The thing is, my job isn't usually very stressful (sometimes magazine deadlines can be tough...) so I rarely feel like I need a vacation. That said, it's still nice to take a vacation every once in a while, especially when I can see friends and family that don't live near me in Dallas.

This year I decided to pack up the SUV and drive to my brother's house in Columbus, Ohio. And, as I often do, I made it a working vacation by taking a bunch of Stamp goodies to work with while I was away from my office. One of my projects actually helped me drive door-to-door to my brother's house without hiccups. If you're guessing we're going to work with GPS again, you guessed correctly.

The reason for my renewed interest in GPS has more to do with my boss than me. You see, my boss Ken is the commander of the Parallax model air force. If you ever visit our facility in California you'll find several gas-powered model airplanes in our warehouse. When the weather is good (a frequent occurrence in northern California) a small group of Parallax flyers will head to the local model airport and fly planes.

Of course, many of the Parallax planes are equipped with BASIC Stamp projects. Ken created a little flight recovery system using the Stamp and an accelerometer. One of our senior engineers, John, created a dual-engine synchronizer for one of his big airplanes. So where do I fit in? Well, Ken wanted to track his plane's flight path and speed, and asked me to come up with a method of doing it. The solution was to strap a small GPS receiver (Garmin eTrex) onto the plane with a BS2p acting as a data logger. I wrote about the methods used in our airplane data logger back in March of 2002.

The program has served us well, but Ken has been asking for better resolution in the data. You see, the old program uses standard NMEA 0183 strings from the GPS receiver that are spit out at 4800 baud. With the bulk of information dumped by the receiver at this baud rate we only get updates every two seconds. For a model airplane traveling around 80 mph, this isn't great. What to do?

While reviewing the eTrex manual I found that it has a simple text output method that can be set to 9600 baud. This is a good start as it doubles the communication speed. The other nice thing about this method is that it uses fixed-position fields. This will help make parsing data easier as we know exactly where everything is within the string. When using the $GPMRC string, some fields are variable-width which complicates the location of data.

Building a Digital Dash

Before tackling Ken's airplane code I decided to experiment with the Garmin text output by creating a supplementary digital dash for my SUV. This would give me the opportunity to work with the text output in a useful manner – and help me get to my brother's house without having to call for directions.

The specs, then, for my digital dash are to display speed (in mph), current time, direction of travel (in degrees), direction as a compass point (i.e. "NW"), and current segment distance (a secondary trip odometer). This all sounds very simple – until you look at the output from the Garmin GPS receiver when set to simple text output.

Simple Text – Not So Simple

For the benefits of speed when using the simple text output, we're strapped with a couple of tricky conversions to display the data as required for my digital dash project. The simple text output looks like this:

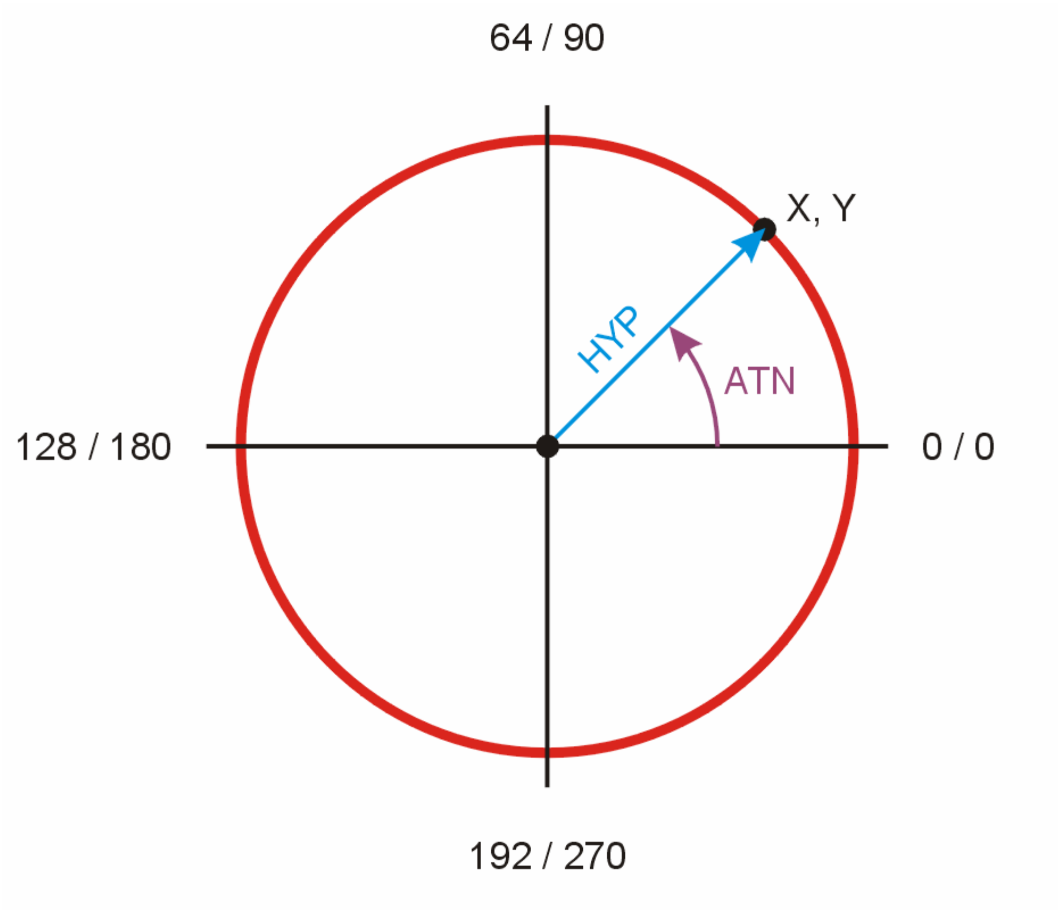
```
@020202183142N3251129W09701159G008+00165E0000N0000D0001
```

For details on all the field positions, please have a look at this URL:

www.garmin.com/support/text_out.html

Here's the tricky bit: speed and direction – both very important to my project – are not directly available. This information is actually derived from two vectors. The first vector represents North/South speed in meters-per-second; the second represents East-West speed in meters-per-second. Sounds like a bit of trigonometry is going to be required.... You got it.

Figure 103.1: PBASIC Unit Circle (Brads / Degrees)



Hey, Look What I Found!

An interesting thing happened when our compiler engineer, Jeff, ported the BASIC Stamp tokenizer from assembly language (not written by him) to C so that it could be compiled for other operating systems... he found two previously-undocumented functions: `ATN` and `HYP`.

The first function, `ATN` (Arctangent), returns the angle, in brads, to the vector represented by X and Y. Wow, that was a mouthful, so let's go through it. Take a look at Figure 103.1 – a PBASIC unit circle. The difference between a PBASIC unit circle and that we'd use in our trig class is that it is divided into 256 units instead of the 360 units we're accustomed to. These units are called binary radians, or brads. Each brad is about 1.4 degrees. When expressing the vector, X and Y are limited to values between -127 and 127 (signed bytes).

The other newly-found function is `HYP`. As you'd expect, this function returns the hypotenuse of a right-triangle with the sides represented by X and Y. And, like `ATN`, the X and Y values for HYP must be limited to -127 to 127.

Okay, then, let's get to it.

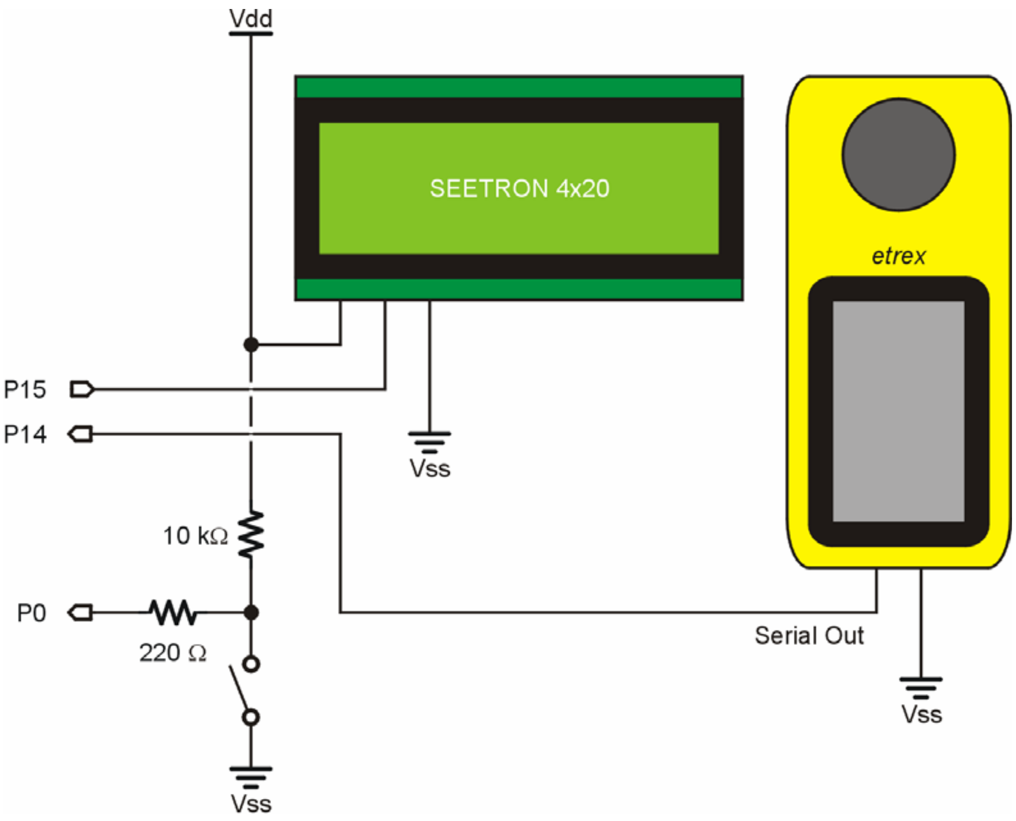
Code for the Road

Something that new programmers frequently run up against is the frightening feeling that their goals are bigger greater than their programming skills. We've all been there – don't sweat it if you're in that state now. An easy way to overcome this fear is to write an outline program, then flesh it out as you go. The reason for this is that it gets you going, and frees you from the details that you will ultimately work out later.

Let me show you a real example. After thinking about my digital dash program, I wrote this chuck of code and it stuck:

```
Main:
  DO
    SERIN GPS, N9600, 3750, No_GPS, [WAIT("@"), SPSTR 50]
    GOSUB Parse_GPS
    GOSUB Calc_Speed
    GOSUB Show_Speed
    GOSUB Show_Time
    GOSUB Show_Vector
    GOSUB Show_Compass
    GOSUB Show_Miles_Acc
  LOOP
```

Figure 103.2: Stamping on Down the Road Schematic



There's only one *real* line of code (`SERIN`), and that wasn't a problem since I lifted it from a previous GPS program. The rest are calls to subroutines. The next step, of course, is to create empty subroutines (just a label and `RETURN`) so that the program will compile without problems. Nothing will happen, except that we'll know we have a structure that works. After that we flesh out each subroutine – testing as we go – until the program is complete.

Let's have a look at the `SERIN` line, since it does quite a bit of work. The first thing to note is that it will timeout if no data is received within 1.5 seconds (the timeout units on the BS2p are 400 microseconds). Assuming a signal does show up, the function will wait for the "@"

character which is the string header. Once it arrives, the next fifty characters will be buffered into the BS2p's scratchpad RAM so we can deal with them later.

The next step is to pull the fields we're interested in; this is done with the `Parse_GPS` subroutine. Here's a section of that code:

```
Parse_GPS:
  idx = 6 : fldWidth = 2
  GOSUB Parse_Field
  hr = workVal + UtcAdj // 24
```

You'll see when you download the full listing that `Parse_GPS` routine is filled with calls like this. The purpose is to send the proper control values to `Parse_Field` and then store what gets lands in `workVal` in the target variable. The variable `idx` is used to point to the start of a field, and `fldWidth` defines how wide that field is. After returning from `Parse_Field`, `workVal` holds the field value.

The hours field in the GPS string starts at position six and is two characters wide. The value returned will actually be UTC time (aka Greenwich Mean Time) – not Central Daylight Time as I needed. The constant `UtcAdj` allows us to adjust for the difference between our local position and GMT.

The real work, of course, takes place in the `Parse_Field` subroutine. Let's have a look:

```
Parse_Field:
  workVal = 0
  IF (fldWidth < 6) THEN
    DO WHILE (fldWidth > 0)
      workVal = workVal * 10
      GET idx, char
      workVal = workVal + (char - "0")
      fldWidth = fldWidth - 1
      idx = idx + 1
    LOOP
  ENDIF
  RETURN
```

On entering `Parse_Field` the variable `workVal` gets cleared. The reason for this is that zero will be returned if a bad field width is passed to it. This seems like a better choice than returning the value from the last legal field access. Assuming the field width is between one and five, it grabs a character from the string, converts it from ASCII to decimal, then shifts it left (remember, we're dealing with decimal numbers so a left-shift is a multiply by 10) if there are more digits in the string. Each pass through the loop decrements the width value and updates the character pointer.

Okay, I know what you're thinking: "Why did you shift first?" The reason is that it simplifies the code. One the first pass, no harm is done because `workVal` is zero. If we waited, we'd have to insert a line of code that tests `fldWidth` for zero before shifting. I know this seems a bit odd, but once you run it through your head a couple times it will make sense. To help out, let's look at a three character field that holds "123" and run through the value of `workVal`:

0 - entry
1 - first pass; fldWidth = 2
12 - second pass; fldWidth = 1
123 - third pass; fldWidth = 0 (loop terminates)

Now that we have our numbers, it's time to crunch them and put them onto an LCD. The first calculated value is the trickiest: speed in miles per hour.

As I told you earlier, speed is derived from two vectors: North/South speed in meters-per-second, and East-West speed in meters-per-second. The `HYP` function is perfect for this – with one caveat. Since the limit for `HYP` is 127 we may have to scale the vectors before using the function. The reason for scaling is that 12.7 meters per second is about 28 miles per hour – and I was certainly planning to drive faster than that on my way to Ohio (otherwise it would have been a very long trip....).

```
Calc_Speed:
  IF (velEW > velNS) THEN
    workVal = velEW
  ELSE
    workVal = velNS
  ENDIF

  LOOKDOWN workVal, <[128,255,382,509], workVal
  workVal = workVal + 1
  velEW = velEW / workVal
  velNS = velNS / workVal
  speed = velEW HYP velNS
  speed = speed * (workVal * 10) ** 14660 + 5 / 10

  mtr10 = mtr10 + ((velEW HYP velNS) * workVal)
  RETURN
```

The `Calc_Speed` subroutine starts by finding the larger of the two speed vectors. This value is used in a `LOOKDOWN` table to create a scaling factor (1 to 4). Each vector is divided by the scaling factor and the speed is calculated with `HYP`. The next step involves a bit more math than we're used to. We have to scale the speed back up and convert to miles per hour. Notice that all the elements are scaled by 10 so that we can add five and divide by 10 to round up. Remember that we can multiply by fractional values with the `**` operator. The term for `**` in this case is 14660 – the same as multiplying by 0.2236 (the conversion factor for going from 0.1 mps to 1 mph).

Before we finish this routine there is one last bit. Part of the project is to accumulate segment distance and that is done in mtr10 (0.1 meters). This is actually quite simple since the speed vectors are returned in tenths of a meter per second. Since we update once per second, we simply need add our speed value to the accumulator. We'll convert it to tenths of miles later.

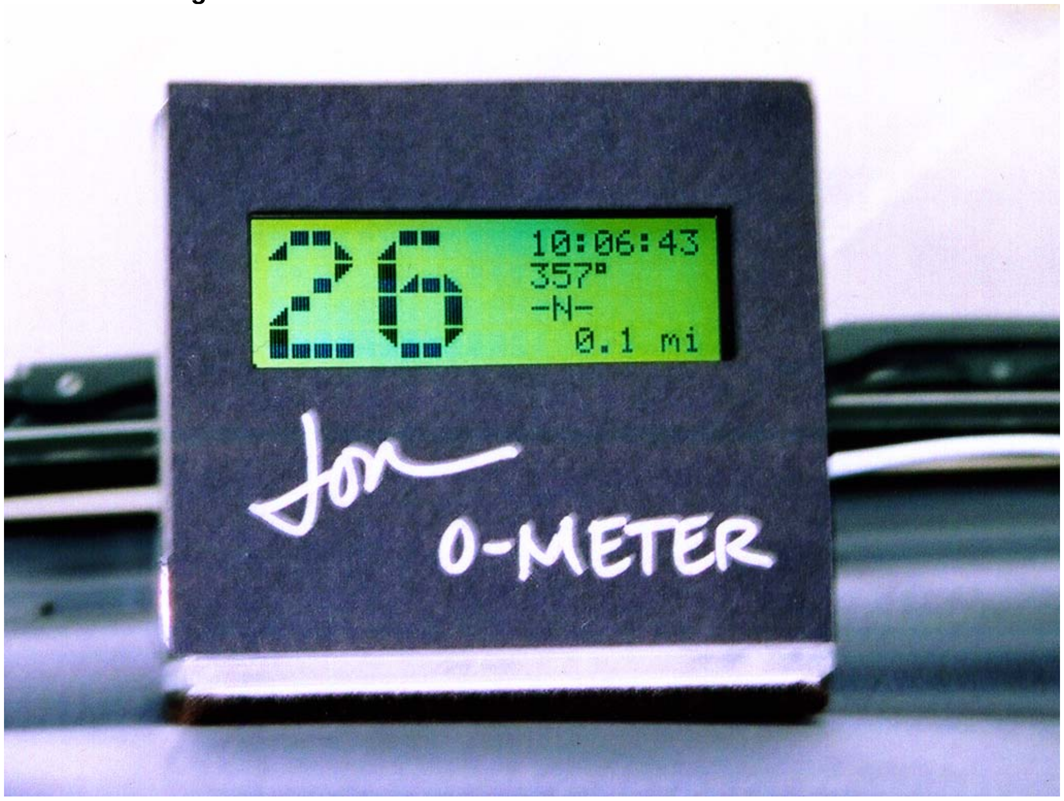
Displaying speed is so trivial that I'm not going to go through it. Why so easy? Because my old pal, Scott Edwards, thoughtfully added a "big characters" function to his 4-line LCD controllers. You can see how this looks in Figure 103.3 which shows the project in use.

Now that we know how fast we're going it would be nice to know the direction we're headed. This information, like speed, is derived from the two direction vectors. This time we'll use the ATN function. Keep in mind that ATN returns brads, not degrees, so we'll have to convert. And there's another thing to deal with. Take a look again at Figure 103.1. Notice how brads and degrees in the polar coordinate system start at the three o'clock position and increase as we rotate counter-clockwise. Now look at a standard magnetic compass. Notice how zero degrees is at the twelve o'clock position and increase while rotating in the clockwise direction. We'll have to deal with this. Don't worry though, it's not difficult. The code that handles this is the `Show_Vector` subroutine:

```
Show_Vector:
  IF (speed > 0) THEN
    GET 39, char
    IF (char = "W") THEN velEW = -velEW
    GET 44, char
    IF (char = "S") THEN velNS = -velNS
    vector = velEW ATN velNS
    vector = vector */ 360
    vector = 360 - vector + 90 // 360

    SEROUT LCD, N9600, [PosCmd, 35 + 64, RtAlign,
                        "3", DEC vector,
                        PosCmd, 35 + 64, DegSym]
  ELSE
    SEROUT LCD, N9600, [PosCmd, 32 + 64, "---"]
  ENDIF
  RETURN
```

Figure 103.3: GPS Tracker In Use on Jon's Dashboard



All you math wizards recognize that the `ATN` function actually converts Cartesian coordinates to a polar (rotational) value. One adjustment that may be required to the vectors is to place them in the proper quadrants so that we get the correct angle from `ATN`. If we overlay a compass onto a Cartesian graph we'll see that the south and west sides of the compass bearings fall into negative graph values. The Garmin GPS doesn't tell us this – it simply tells us North or South, East or West. So we do a quick check. If the North/South vector is south, we take the negative value of the speed vector; we do the same for the East/West vector.

Again, the ATN function returns brads. To convert to degrees we will multiply by 1.4. (*/ 360). Finally, we have to reverse the direction by subtracting our direction from 360, then add 90 degrees to reorient zero to the correct (12 o'clock) position.

Notice that we don't actually go through this trouble if we're not moving. You see, we can't tell which direction we're pointed unless the GPS receiver is moving. If we pass zero speed vectors to this routine it will tell us we're pointed East (90 degrees). This will usually not be the case so we'll simply display dashes when the vehicle is stopped.

You may be wondering why we didn't scale the vector values as when we calculated the speed. The reason is that the vectors were scaled by the Calc_Speed subroutine and not modified after – so they are already scaled appropriately for the ATN function.

The next step in the program is to convert the direction of travel to a more useful string; "-N-" when we're traveling on a heading of zero degrees, for example. The subroutine that handles this is called Show_Compass:

```
Show_Compass:
  SEROUT LCD, N9600, [PosCmd, 52 + 64]
  IF (speed > 0) THEN
    eePntr = vector * 100 + 1125 / 2250 // 16
    eePntr = eePntr * 3
    FOR idx = 0 TO 2
      READ eePntr + idx, char
      SEROUT LCD, N9600, [char]
    NEXT
  ELSE
    SEROUT LCD, N9600, ["---"]
  ENDIF
  RETURN
```

As with direction in degrees, we'll only show this value if moving. What this routine does is create a pointer to one of sixteen strings. If we divide the compass face of 360 degrees by 16 we get 22.5. Since we want our pointer to be right in the middle of a 22.5 degree segment, we divide the segment width by two to get 11.25. The BASIC Stamp doesn't do floating point, so we multiply everything by 100 before doing the pointer math.

A simple loop handles pulling the string from a DATA table and printing it on the LCD. This is actually quite useful. While driving to my brother's house I was watching the display to make sure that I was generally traveling in a north-easterly direction will driving. That was actually quite comforting across some very long, flat stretches of road.

We're almost there. The last thing to discuss is the trip meter built into the code. What I did before leaving home was go to one of the many Internet mapping sites and downloaded door-to-door directions to my brother's house. The directions basically told me to get on a specific road and travel a given distance. As you'll see in the schematic, I've got a button on P0 so that I can reset the meter. What I did while driving was get on a road, press the segment reset button, then watch for it to increment to the distance given in my directions. Since the display was sitting on the dashboard right above the steering wheel, it was very easy to see and monitor. Without fail, every new junction was within a tenth of a mile of where it should have been.

Let's have a look at the segment code.

```
Show_Miles_Acc:
  IF (FuncBtn = Pressed) THEN
    mtr10 = 0
    mi10 = 0
  ENDIF

  IF (mtr10 >= 1609) THEN
    mi10 = mi10 + 1 // 10000
    mtr10 = mtr10 - 1609
  ENDIF

  SEROUT LCD, N9600, [PosCmd, 75 + 64,
                     RtAlign, "3",
                     DEC (mi10 / 10),
                     PosCmd, 75 + 64, ".",
                     DEC1 mi10, " mi"]
  RETURN
```

On entering this subroutine we check to see if the button is being pressed. If it is, we clear the segment accumulators. Keep in mind that this routine – like all routines – only gets called once per second so you may need to hold the button for a moment. Most of the time the button will not be pressed. In this case we will look at the tenths-of-meters accumulator and compare it to 1609 (the conversion factor to go from tenths-of-meters to tenths-of-miles).

Let me go back and review some logic. The speed output values in the GPS string are in tenths-of-meters per second. Since we're checking every second, the speed value becomes our distance traveled between GPS scans. The `Calc_Speed` subroutine took care of accumulating distance in tenths-of-meters. We can convert this value to tenths-of-miles by dividing by 1609. Finally, we put it up on the display taking advantage of another nice feature or SEETRON displays: the ability to right-justify numbers.

The Need For Speed Monitoring

This code was a bit more complicated than the projects I usually present here but I think it was fun and was certainly a good learning experience for me. Be sure to download the full listing and go through it slowly – it should all make sense once you've studied it a while.

The cool thing about this project is that it provides non-contact speed and distance monitoring. Since it uses GPS it can be put on anything that moves: a car, a boat, a scooter, a go-kart. Do you want to know how fast your soapbox derby car goes? Now you can.

Before I sign-off this month let me give you one more web link. While working on this program, I found the following conversion site to be useful:

www.sciencemadesimple.com/conversions.html

Stamps in the Shack!

Finally, for those of you who read this column but haven't actually started with Stamps, or those of you who might want to get a friend or youngster started ... good news. Parallax has teamed with Radio Shack to put a great starter kit in select Radio Shack stores (those in major population centers). The kit includes the Parallax HomeWork board and the "What's a Microcontroller?" text and components. It's a great way to get started and at a fantastic price – and you can pick it up at your local Radio Shack. Look for the starter kits to hitting shelves this holiday season.

That's all for now. Happy Thanksgiving to you and yours ... and as always, Happy Stamping.

```
' =========================================================================
'
'   File...... Stamp_Dash.BSP
'   Purpose... Digital Instrumentation for Moving Vehicle
'   Author.... Jon Williams
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 21 SEP 2003
'
'   {$STAMP BS2p}
'   {$PBASIC 2.5}
'
' =========================================================================


' -----[ Program Description ]---------------------------------------------
'
' This program accepts text data from a Garmin GPS receiver using the
' Garmin text output at 9600 baud.  The advantage over NMEA 183 for this
' application is speed (2x) and fixed field positions and widths.
'
' String:
'
'   @020202183142N3251129W09701159G008+00165E0000N0000D0001 <-- GPS data
'    |
'    |         1         2         3         4         5
'    012345678901234567890123456789012345678901234567890123 <-- SPRAM addr
'
'
' @             header
' 020202        UTC date (yr [2], mo, dy)
' 183142        UTC time (hr, mn, sc)
' N             lat hemisphere
' 32            lat degrees
' 51            lat minutes
' 129           lat frac minutes (1/1000)
' W             long hemisphere
' 097           long degrees
' 01            long minutes
' 159           long frac minutes (1/1000)
' G             position status
' 008           horizontal position error (meters)
' +             altitude sign
' 00165         altitude above/below sea level (meters)
' E             E/W velocity direction
' 0000          E/W velocity (meters per sec - tenths xxx.x)
' N             N/S velocity direction
' 0000          N/S E/W velocity (meters per sec - tenths xxx.x)
' D             vertical velocity direction (Up / Down)
' 0001          vertical velocity (meters per sec - hundredths xx.xx)
```

```
'
'
' For additional details see: http://www.garmin.com/support/text_out.html


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


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

LCD             PIN     15                        ' serial to 4x20 LCD
GPS             PIN     14                        ' serial in from GPS
FuncBtn         PIN     0                         ' function btn (active low)


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

N9600           CON     $40F0

Null            CON     0                         ' null = 1 ms delay
CrsrHm          CON     1                         ' cursor home
BigOn           CON     2                         ' big characters
BigOff          CON     3                         ' -- off
NoCrsr          CON     4                         ' no cursor
ULCrsr          CON     5                         ' underline cursor
BlnkCrsr        CON     6                         ' blinking cursor
ClrLcd          CON     12                        ' clear LCD
BLOn            CON     14                        ' backlight on
BLOff           CON     15                        ' -- off
PosCmd          CON     16                        ' positioning command
ClrCol          CON     17                        ' clear column
RtAlign         CON     18                        ' right align
Esc             CON     27

DegSym          CON     223
Pressed         CON     0                         ' button is active-low

UtcAdj          CON     24 - 5                    ' adjust UTC for CDT


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

idx             VAR     Byte                      ' loop counter
fldWidth        VAR     Nib                       ' width of data field
workVal         VAR     Word                      ' temp value from parsing

hr              VAR     Byte                      ' adjusted hours
mn              VAR     Byte                      ' minutes
sc              VAR     Byte                      ' seconds

velEW           VAR     Word                      ' velocity E/W
```

```
velNS           VAR     Word                    ' velocity E/W
speed           VAR     Word
vector          VAR     Word                    ' direction vector

eePntr          VAR     Word                    ' eeprom pointer
char            VAR     Byte                    ' display character

mtr10           VAR     Word                    ' meters x 0.1
mi10            VAR     Word                    ' miles x 0.1


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

CPoints         DATA    "-N-", "NNE", "N-E", "ENE"
                DATA    "-E-", "ESE", "S-E", "SSE"
                DATA    "-S-", "SSW", "S-W", "WSW"
                DATA    "-W-", "WNW", "N-W", "NNW"


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

Setup:
  PAUSE 500                                     ' let LCD initialize
  SEROUT LCD, N9600, [ClrLcd]                   ' clear screen
  IF (FuncBtn = Pressed) THEN
    SEROUT LCD, N9600, [BLOn]                   ' backlight on
  ENDIF


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

Main:
  DO
    SERIN GPS, N9600, 3750, No_GPS, [WAIT("@"), SPSTR 50]
    GOSUB Parse_GPS
    GOSUB Calc_Speed
    GOSUB Show_Speed
    GOSUB Show_Time
    GOSUB Show_Vector
    GOSUB Show_Compass
    GOSUB Show_Miles_Acc
  LOOP

  END


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

' Signal lost -- or receiver set to wrong output

No_GPS:
  SEROUT LCD, N9600, [ClrLCD, CR,
                      "- GPS SIGNAL ERROR -"]
  PAUSE 1000
  SEROUT LCD, N9600, [ClrLcd]
  GOTO Main


' Grab fields from GPS string

Parse_GPS:
  idx = 6 : fldWidth = 2                        ' hours
  GOSUB Parse_Field
  hr = workVal + UtcAdj // 24                   ' update for local position

  ' 12 hour mode
  hr = hr // 12
  IF (hr = 0) THEN hr = 12

  idx = 8 : fldWidth = 2                        ' minutes
  GOSUB Parse_Field
  mn = workVal

  idx = 10 : fldWidth = 2                       ' seconds
  GOSUB Parse_Field
  sc = workVal

  idx = 40 : fldWidth = 4                       ' east/west velocity
  GOSUB Parse_Field
  velEW = workVal

  idx = 45 : fldWidth = 4                       ' north/south velocity
  GOSUB Parse_Field
  velNS = workVal

  RETURN


' Parses numeric data field from GPS string in ScratchPad
' -- pass digits in field in "fldWidth"
' -- value returned in "workVal"

Parse_Field:
  workVal = 0                                   ' clear return value
  IF (fldWidth < 6) THEN                        ' valid fldWidth?
    DO WHILE (fldWidth > 0)
      workVal = workVal * 10                    ' shift result digits left
      GET idx, char                             ' get digit from field
      workVal = workVal + (char - "0")          ' convert, add into value
      fldWidth = fldWidth - 1                   ' decrement field width
      idx = idx + 1                             ' point to next digit
    LOOP
  ENDIF
  RETURN


' Show time on LCD
' -- HH:MM:SS

Show_Time:
  SEROUT LCD, N9600, [PosCmd, 12 + 64,
                     DEC2 hr, ":", DEC2 mn, ":", DEC2 sc]
  RETURN


' Calculate speed from E/W and N/W velocities
' -- values are scaled to fit 0 - 127 range
' -- derived value scaled back
' -- saves direction vector
' -- accumulates meters traveled
'
' Note: This routine divides down the velocity values so that they
' will fit within the constraints of HYP and ATN functions.

Calc_Speed:
  ' find biggest vector
  IF (velEW > velNS) THEN
    workVal = velEW
  ELSE
    workVal = velNS
  ENDIF

  ' create scaling factor (1 to 4)
  LOOKDOWN workVal, <[128,255,382,509], workVal
  workVal = workVal + 1
  velEW = velEW / workVal                       ' scale to < 127
  velNS = velNS / workVal
  speed = velEW HYP velNS                       ' calculate speed (mps)

  ' speed (mph) = meters/sec (tenths) * 0.223694
  speed = speed * (workVal * 10) ** 14660 + 5 / 10

  mtr10 = mtr10 + ((velEW HYP velNS) * workVal) ' accumulate meters moved
  RETURN


' Show current speed in "big digits" format
' -- code right justifies

Show_Speed:
  SEROUT LCD, N9600, [CrsrHm]                   ' move to line 1, col 1 (Home)
  IF (speed < 10) THEN                          ' if 1 digit
    FOR idx = 0 TO 4                            ' -- erase previous 10's
```

```
      SEROUT LCD, N9600, [ClrCol]
    NEXT
    ' print 1's
    SEROUT LCD, N9600, [BigOn, DEC1 speed, BigOff]
  ELSE
    ' print 2-digit speed
    SEROUT LCD, N9600, [BigOn, DEC2 speed, BigOff]
  ENDIF
  PAUSE 20
  RETURN


' Show current direction in degrees
' -- 0º to 359º
' -- vector is converted to Brads, then Degrees
' -- value is rotated to match compass orientation
'
' Note that this routine counts on velEW and velNS being
' scaled to < 127 (done in Calc_Speed)

Show_Vector:
  IF (speed > 0) THEN
    ' get EW direction
    GET 39, char
    IF (char = "W") THEN velEW = -velEW         ' if W, change quadrant
    ' get NW direction
    GET 44, char
    IF (char = "S") THEN velNS = -velNS         ' if S, change quadrant
    vector = velEW ATN velNS                    ' in Brads
    vector = vector */ 360                      ' convert to degrees
    vector = 360 - vector + 90 // 360           ' orient to compass

    SEROUT LCD, N9600, [PosCmd, 35 + 64, RtAlign,
                        "3", DEC vector,
                        PosCmd, 35 + 64, DegSym]
  ELSE
    SEROUT LCD, N9600, [PosCmd, 32 + 64, "---"]
  ENDIF
  RETURN


' Show current direction as compass point
' -- 0º = "N"

Show_Compass:
  SEROUT LCD, N9600, [PosCmd, 52 + 64]          ' move to line 3, column 13
  IF (speed > 0) THEN
    eePntr = vector * 100 + 1125 / 2250 // 16   ' calc hextant (22.5°)
    eePntr = eePntr * 3                         ' point to start of string
    FOR idx = 0 TO 2                            ' print compass string
      READ eePntr + idx, char
      SEROUT LCD, N9600, [char]
    NEXT
  ELSE
    ' not moving
    SEROUT LCD, N9600, ["---"]
  ENDIF
  RETURN


' Show leg mileage accumulator
' -- 0.0 to 999.9
' -- input on P0 will reset acc if low

Show_Miles_Acc:
  IF (FuncBtn = Pressed) THEN                   ' reset on button press
    mtr10 = 0
    mi10 = 0
  ENDIF

  IF (mtr10 >= 1609) THEN                       ' gone 1/10 mile?
    mi10 = mi10 + 1 // 10000
    mtr10 = mtr10 - 1609
  ENDIF

  SEROUT LCD, N9600, [PosCmd, 75 + 64,
                     RtAlign, "3",
                     DEC (mi10 / 10),
                     PosCmd, 75 + 64, ".",
                     DEC1 mi10, " mi"]
  RETURN
```