Column #83 March 2002 by Jon Williams:

Where in the World is My BASIC Stamp?

A few days before I started writing this article, President Bush submitted his budget proposal to Congress. Part of the new budget calls for dramatic increases in military spending to make up for lack of technological progress in the past several years. I'm not trying to be political here. I simply want to point out that we, as civilians, do and will benefit from technology developed by our military and space programs.

Case in point: GPS, the Global Positioning System. GPS was originally developed by the military to provide precise location and navigation capabilities in a wide range of military applications. We've certainly all heard of GPS-guided "smart" bombs. The American military dropped quite a few of them on those Taliban idiots in Afghanistan. Yes, a few of them missed the mark with tragic consequences. This wasn't the fault of the weapon or the GPS system; it was a problem with the location data loaded into the bombs.

Lucky for us, the technology has been made available to the civilian world and the uses for GPS are far more friendly. GPS navigation systems in cars are very useful, particularly for those driving in an unfamiliar city. Not long ago I went to lunch with a friend who has his car equipped with such a system. From a menu he selected his favorite restaurant and the system, using GPS and voice synthesis, "talked" us all the way there ... with a very friendly female voice, no less. He

even made a deliberate mistake to show me how the system could compensate for the error. The system was very cool, telling us about approaching intersections and to be prepared to turn. It was a very impressive demonstration of GPS technology.

On a smaller scale, GPS receivers have become incredibly popular with hikers and others involved in outdoor recreational activities. One of the most popular consumer GPS units for these applications in the *eTrex* GPS from Garmin. Garmin actually has a family of *eTrex* GPS receivers. At the low end, the unit goes for about $120 retail. That's not too bad for a piece of equipment that can tell you where you are on the planet with an accuracy of down to 150 feet or so – much better depending on conditions. I routinely see estimated accuracies of less than 20 feet on my *eTrex*. While any of the models of the *eTrex* family will work with this month's program, the entry-level model works just fine and is what I used to test the code.

GPS Basics

There are several excellent explanations of the GPS system on the Internet (see sidebar), so I won't go into a lot of detail here. Fundamentally, the system works like this: there are 24 GPS satellites in orbit around the planet. The orbit of these satellites is very carefully controlled by the U.S. Department of Defense. Each satellite broadcasts the precise time (using an atomic clock) and other information required by GPS receivers. The GPS unit on the ground receives signals from three or more satellites and, using some fairly tricky math, is able to triangulate its planetary position.

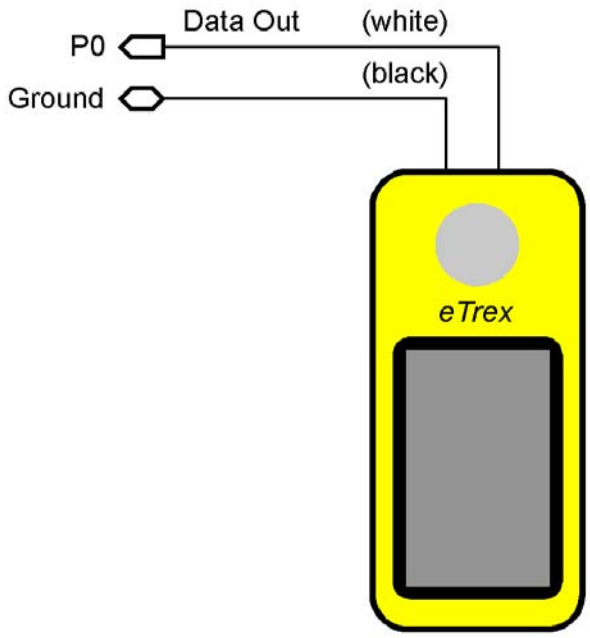
And all of this happens in a unit the size of a cellular telephone. Beyond basic position, the GPS receiver can tell us accurate time, speed, heading and other information relevant to navigation. It's very neat stuff.

Getting Connected To The BASIC Stamp

Most GPS receivers output a stream of data so that it can be used by other devices. The most common format is NMEA 0183 (National Marine Electronics Association). This data is provided as a series of comma-delimited ASCII strings, each preceded with an identifying header. The data output rate is very tame: 4800 baud, N-8-1. This is no problem for the BASIC Stamp.

The trickiest part about connecting the *eTrex* to a BASIC Stamp is the mechanical connection. The *eTrex* uses a proprietary connector and the cables from Garmin a fairly pricey – about a third the cost of the receiver itself. I can vouch for the quality though; they are made very well.

Figure 83.1: BS2p to GPS Connection



It seems that the popularity of the *eTrex* GPS has created a cottage industry of third party connector and cable manufacturers. In the web sites sidebar you'll find a list of sites that will sell you the connectors and a variety of prefabricated cables that will work with your *eTrex*. For the this month's program I used the Garmin bare wire cable (#010-10205-00). You can either purchase that cable from Garmin or go the DIY route. If using the Garmin cable, the wire colors are shown in Figure 83.1.

Web Sites Worth Checking Out

How GPS Works
- celia.mehaffey.com/dale/theory.htm

Decoding NMEA 0183 Strings

- celia.mehaffey.com/dale/nmea.htm
- home.mira.net/~gnb/gps/nmea.html

***eTrex* Related**

- www.garmin.com
- www.firstwaypoint.com
- www.etrex.webz.cz
- www.geocities.com/etrexkb

Alternate Sources for *eTrex* Cables and Connectors

- www.pfranc.com
- www.gpsmemory.com
- www.blue-hills-innovations.com

BASIC Stamps and GPS

- www.parallaxinc.com/downloads/Resources/Aero%20GPS%20Brief.PDF
- www.stoneflyers.com/gps_guided_truck.htm

Decoding The GPS Output

The purpose of the program this month is to "listen" to the GPS output until a certain string is transmitted, grab it when it is and then pull all the data out of it and convert it to numbers that can be used or manipulated. To do this we will be using the BS2p. The BS2p has a (undocumented) serial modifier called **SPSTR**. This modifier collects serial input and saves it in the scratchpad RAM, starting at location zero. Once the string is in the scratchpad, it can be parsed and converted as required.

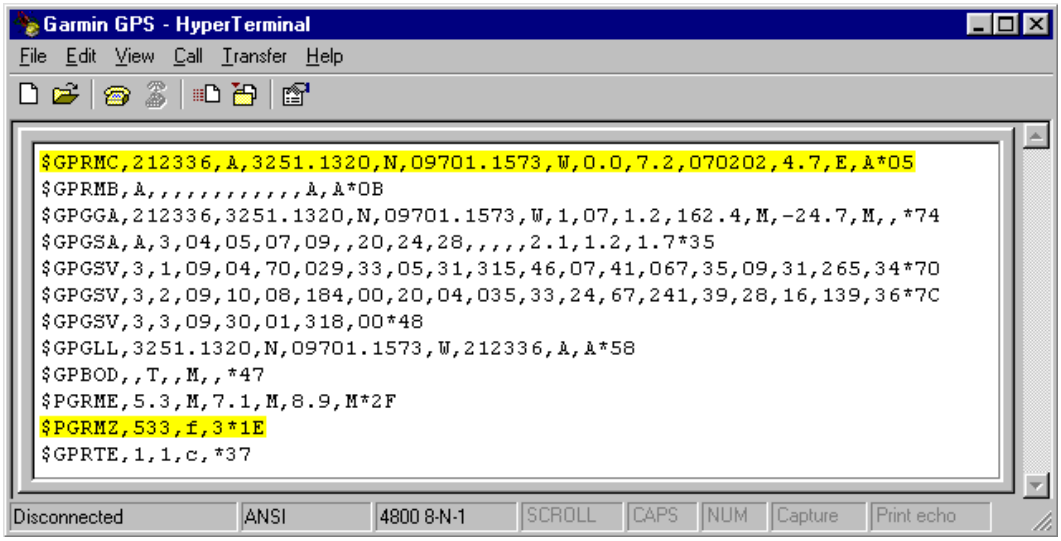
The main string the program waits for is called $GPRMC -- Recommended minimum specific GPS/Transit data. If you connect your *eTrex* to a terminal program you'll see that the $GPMRC sentence is the first line of several that are output every two seconds (Figure 83.2). The baud rate and number of characters transmitted for all of the sentences are what determine the two-second delay between transmissions.

After I had the basic code working, I decided to add another piece: altitude. This is transmitted in the same format using a Garmin-specific string: $PGRMZ. As it turns out, the altitude can be intercepted and decoded within the **SERIN** statement -- without buffering it into the scratchpad RAM. And with the speed of the BS2p (program speed, not ground speed...), there is plenty of time between the arrival of $GPRMC and $PGRMZ to decode the $GPRMC data.

The Code, The Code

Since the connection between the Stamp and the *eTrex* is electrically trivial, I'll jump right into the code. This looks like a long program but it really is fairly straightforward and there are a few neat tricks inside. Again, the purpose is to capture the GPS output and then parse out specific data and convert it to numeric form so that it can be manipulated. The output of the program is a report sent to the **DEBUG** screen. You can use the guts of this program as a jump-off point for your own GPS-based projects.

Figure 83.2: HyperTerminal GPS Output from Garmin eTrex



The **Initialization** section opens the **DEBUG** screen, draws a ruler to help us analyze the $GPMRC sentence and sets up some labels where the GPS data is displayed. This is all pretty easy stuff. I've started using the more of the built-in formatting functions available with **DEBUG**. Notice the constant called **MoveTo**. This code will causes the cursor to move to a specific X/Y location in the **DEBUG** screen. Another that is used a little later is **ClrRt**. The **ClrRt** code will clear everything from the current cursor position to the end of the current line. This code is handy for keeping the screen neat and uncluttered with old data.

The first line in the **Main** section does the work of waiting on the $GPMRC sentence and storing it in the BS2p's scratchpad RAM. As you can see, the **SERIN** command is setup for 4800 baud with a timeout of three seconds (3000 milliseconds). This value make sense since the sentence should show up every two seconds. And notice that the program is actually waiting on "GPRMC," instead of "$GPRMC" as you might expect. The reason for this twofold: the **WAIT** modifier is limited to six characters and I want the first data character (position UTC) to be placed in address zero of the scratchpad.

When using **SPRSTR** we have to tell the Stamp how many characters to buffer. In this case I've set the character count to 65. Usually, the $GPMRC string will be about 60 characters long. I'm going allow 65 so that when I take this project "on the road" and the variable-width speed field gets wider, everything will be captured. Be careful when using the length specification in **SPSTR** – it's okay if the length is specified short, too long and the Stamp will be left waiting for more characters. That's not a problem for this program since several hundred bytes of data are transmitted after the $GPMRC string.

Now comes the real work: extracting numeric data from the string.

Some of you may have seen a little project we did at Parallax with GPS and model airplanes. Just for fun, we created a little GPS data logger from a BS2p and strapped the receiver and Stamp to the wing of a trainer aircraft. My boss, Ken, flew the plane while the program was running and when we got back to the office we were able to graph the flight path of the airplane. It was a lot of fun.

As I look back on that code it was a bit, well ... inelegant ... and I really wanted to clean it up and make it easier to use with different GPS strings. That's what I'm presenting here.

One the key elements of the program is a subroutine called **String_To_Value**. The purpose of the routine is to convert an ASCII string to a 16-bit number. The string is stored in the scratchpad. The subroutine starts by knowing the position of the first digit in the string and how many digits to convert.

Take a look at the code. The routine starts by clearing the return variable *workVal*. Next, the field width is checked to make sure it falls in range. We have to be a little careful when specifying a field width of five since 65,535 is the largest value that a Stamp variable can hold.

The heart of the routine retrieves a character from the scratchpad, converts it from ASCII to a number by subtracting "0" (decimal 48) and then adding the digit value into the return variable. The field width is decremented and checked for zero. If it's zero, the routine is done and the value

the field is returned in *workVal*. If not, the digits in *workVal* are shifted left to make room for the next. Since we're dealing with decimal numbers, this is accomplished by multiplying *workVal* by ten. Quite a lot of the work in the program is done with this routine. After calling **String_To_Value**, the value is moved from *workVal* to the specific variable used for storage.

Just a couple notes on the numeric values. The decimal portion of the latitude and longitude values are fractional minutes. To convert fractional minutes to seconds we multiply by 60. In the case, we actually use 0.06 to get tenths since the routine returns the fractional value multiplied by 1000.

In the past I've used the */ (star-slash) operator for fractional values. In this case, the ** (star-star) operator is used for better precision. Star-star can only be used with fractional values of less than one. To get the value for this operator, multiply the fraction by 65,536. So, 0.06 x 65,536 is 3932 ($0F5C). To use a star-star with fractional values greater than one, you need to multiply the whole portion separately. Take a look the code that converts knots to miles per hour.

Authors Note: For more hints and tricks on using star-star, be sure to visit Dr. Tracy Allen's web site at www.emesys.com/BS2math1.htm. Tracy is the coolest Ph.D. you'll ever meet – a Stamp guru and a real regular guy!

The next step in the decoding process is getting the signal validity, latitude indicator and longitude indicator. These arrive in the $GPRMC string as letters. Since there's really no need to waste eight bits of storage for what are, essentially, binary values, the program uses a variable called *flags* to hold them as bits. Simple comparisons are used to set or clear the associated bits in *flags*.

Next comes speed. The speed value (knots) starts in a fixed position but is a variable-width field. The legal values are from 0.0 to 999.9. Decoding speed is not much different from what has previously been done – the difference, of course, is that the field width is not known. The routine **Mixed_To_Tenths** deals with this problem by looking for the decimal point in the field. Since the decimal point is expected, the return value is in tenths.

The last bit of data that I want to collect from the $GPRMC string is the location date. This is a fixed-with field, but is it located after the variable-width speed field so finding it requires a strategy. The way it's done in this program is by counting the field-delimiting commas. I created a subroutine called **Move_To_Field** to handle this. To use the routine, I put the field number in *field*, then call the routine. When it **RETURN**s, the variable *idx* holds the position of the first digit in that field. Once the position of the date field is known, the rest is easy.

One last thing to do with the $GPMRC data ... I live in Dallas, Texas which is six hours behind UTC time reported by the GPS receiver. Correcting for the offset isn't very hard, it just takes a bit

of code because there may be a date change as well. The time and date correction code looks for a day cross, the adds or subtracts a day based on the UTC offset specified by the program. The only thing this code doesn't do is account for leap years.

Since negative values are stored in two's-compliment form, bit 15 will be set in negative numbers. This fact is used by the **BRANCH** line to direct the code to adding or subtracting a day – with month and year correction if necessary.

I mentioned earlier that we've played with attaching GPS units to model airplanes. Doing that, it would be nice to know the altitude. One thing to understand is that GPS altitude from civilian receivers is terribly inaccurate, but we're willing to pull it out anyway. Later we'll experiment with some of the higher-end Garmin units that have barometric altimeters for better accuracy.

The altitude is broadcast in the Garmin-specific $PGRMZ string. The cool thing about this is that it's a whole value in a single field so we can grab it on-the-fly using the **DEC** modifier. When the comma after the altitude field is hit, **DEC** terminates and the value is stored in *altitude*.

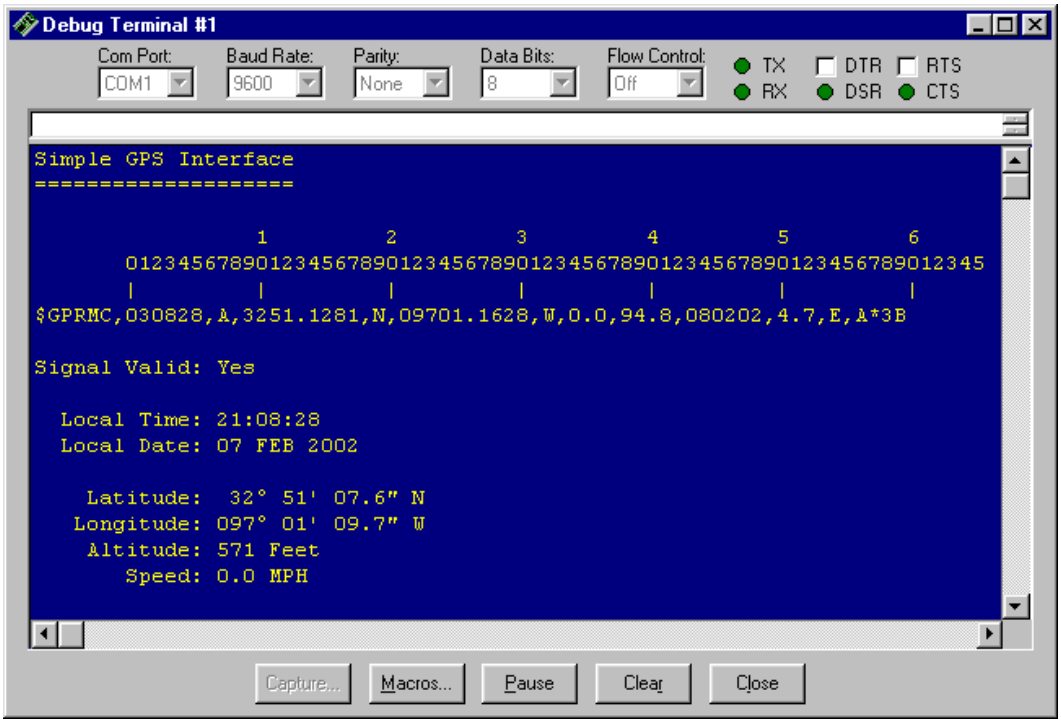
We Got, Let's Show It

Okay, now the latest $GPRMC string is safely stored in RAM, its data is parsed out, so it's time to display the results the program's cool code. First comes the display of the raw $GPMRC string. The display routine is fairly simple. Since the header isn't actually captured with **SERIN**, it gets printed manually. Then, one-by-one, characters are retrieved (with **GET**) from the scratchpad and displayed on screen. Remember that these are ASCII characters so there is no conversion requirement.

Since the string length can vary, what the program does is look for the asterisk which comes right before the two-digit checksum. Once the asterisk is detected and printed, it's a simple matter to **GET** and print the next two characters. Finally, the rest of the line is cleared with **ClrRt** in case the current string was shorter than the previous.

Next, the signal validity is printed. To make the display a little user-friendly, zero-terminated strings are stored in EEPROM and printed using **Print_Z_String**. If the signal was good, the values are printed, otherwise the screen is cleared. Done! The final output of the program is shown in Figure 83.3. Notice that that the time and date have been corrected for the six hour difference between Dallas and UTC time.

Figure 83.3: GPS Debug Window Shows $GPRMC



What Will You Do With It?

Okay, it's your turn. Now that you know how to get GPS data into your Stamp, what will you do with it? Me? Well, I'm thinking about driving to my brother's house in Ohio this summer so I have this idea about creating a "head up" speedometer reading (reflecting large 7-segment displays off the front windshield) and, using a Quadravox QV306, having the system tell me the time, inside and outside temperatures, estimated time to my next waypoint ... and anything else I can figure out. It should be fun. If it works like I hope, I'll write about it when I get back to "Big D."

Until next month, have fun with GPS and Happy Stamping!

```
' -----[ Title ]-----------------------------------------------------------------
'
' Program Listing 83.1
' File...... SIMPLE GPS.BSP
' Purpose... Simple GPS Interface
' Author.... Jon Williams
' E-mail.... jonwms@aol.com
' Started... 16 JUL 2001
' Updated... 07 FEB 2002

' {$STAMP BS2p}


' -----[ Program Description ]---------------------------------------------------
'
' Reads NMEA data string from GPS receiver and parses data.  GPS string is
' buffered into scratchpad RAM with SPSTR modifier.  Once in SPRAM, data is
' parsed out based on its position.
'
' $GPRMC,POS UTC,POS STAT,LAT,LAT D,LON,LON D,SPD,HDG,DATE,MAG VAR,MAG REF,*CC
'
'  POS UTC  - UTC of position. Hours, minutes and seconds. (hhmmss)
'  POS_STAT - Position status. (A = Data valid, V = Data invalid)
'  LAT      - Latitude (ddmm.ffff)
'  LAT D    - Latitude direction. (N = North, S = South)
'  LON      - Longitude (dddmm.ffff)
'  LON D    - Longitude direction (E = East, W = West)
'  SPD      - Speed over ground. (knots) (0.0 - 999.9)
'  HDG      - Heading/track made good (degrees True) (x.x)
'  DATE     - Date (ddmmyy)
'  MAG VAR  - Magnetic variation (degrees) (x.x)
'  MAG REF  - Magnetic variation (E = East, W = West)
'  *CC      - Checksum
'
' Custom Garmin string:
'
' $PGRMZ,ALT,F,X*CC
'
'  ALT,F    - Altitude in feet
'  X        - Position fix dimensions (2 = user, 3 = GPS)
'  *CC      - Checksum


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


' -----[ I/O Definitions ]--------------------------------------------------------
'
GPSpin          CON     0                       ' GPS serial input
```

```
' -----[ Constants ]-------------------------------------------------------
'
N4800           CON     16884                   ' baud rate for GPS

MoveTo          CON     2                       ' DEBUG positioning command
LF              CON     10                      ' linefeed
ClrRt           CON     11                      ' clear line right of cursor

EST             CON     -5                      ' Eastern Standard Time
CST             CON     -6                      ' Central Standard Time
MST             CON     -7                      ' Mountain Standard Time
PST             CON     -8                      ' Pacific Standard Time

EDT             CON     -4                      ' Eastern Daylight Time
CDT             CON     -5                      ' Central Daylight Time
MDT             CON     -6                      ' Mountain Daylight Time
PDT             CON     -7                      ' Pacific Daylight Time

UTCfix          CON     CST                     ' for Dallas, Texas
Comma           CON     ","
DegSym          CON     176                     ' degrees symbol for report
MinSym          CON      39                     ' minutes symbol
SecSym          CON      34                     ' seconds symbol


' -----[ Variables ]-------------------------------------------------------
'
idx             VAR     Byte                    ' index into GPS data in SPRAM
flags           VAR     Byte                    ' holds bit values
valid           VAR     Flags.Bit3              ' is data valid?

tmHrs           VAR     Byte                    ' time fields
tmMins          VAR     Byte
tmSecs          VAR     Byte

latDeg          VAR     Byte                    ' latitude
latMin          VAR     Byte
latSec          VAR     Word
latNS           VAR     flags.Bit0              ' 0 = N

longDeg         VAR     Byte                    ' longitude
longMin         VAR     Byte
longSec         VAR     Word
longEW          VAR     flags.Bit1              ' 0 = E

speed           VAR     Word                    ' in tenths of mph
altitude        VAR     Word                    ' in feet

day             VAR     Byte                    ' day of month, 1 - 31
```

```
month           VAR     flags.Nib1              ' month, 1 - 12
year            VAR     Byte                    ' year, 00 - 99

char            VAR     Byte                    ' byte pulled from SPRAM
workVal         VAR     Word                    ' for numeric conversions
eeAddr          VAR     workVal                 ' pointer to EE data

field           VAR     Nib                     ' field #
fldWidth        VAR     field                   ' width of field


' -----[ EEPROM Data ]------------------------------------------------------
'
NotValid        DATA    "No", 0
IsValid         DATA    "Yes", 0
DaysInMon       DATA    31,28,31,30,31,30,31,31,30,31,30,31
MonNames        DATA    "JAN",0,"FEB",0,"MAR",0,"APR",0,"MAY",0,"JUN",0
                DATA    "JUL",0,"AUG",0,"SEP",0,"OCT",0,"NOV",0,"DEC",0


' -----[ Initialization ]----------------------------------------------------
'
Initialize:
  PAUSE 250                                     ' let DEBUG open
  DEBUG CLS                                     ' clear the screen
  DEBUG "Simple GPS Interface",CR
  DEBUG "===================="

Draw Ruler:
    FOR idx = 0 TO 65
      IF (idx = 0) THEN Print_Ones
      IF (idx // 10) > 0 THEN Print Ones
      DEBUG MoveTo, (7 + idx), 3, DEC1 (idx / 10)
    Print Ones:
      DEBUG MoveTo, (7 + idx), 4, DEC1 (idx // 10)
    Print_Ticks:
      IF (idx // 10) > 0 THEN Next_Digit
      DEBUG MoveTo, (7 + idx), 5, "|"
    Next Digit:
  NEXT

Draw_Data_Labels:
  DEBUG MoveTo, 0,  8, "Signal Valid: "
  DEBUG MoveTo, 0, 10, "  Local Time: "
  DEBUG MoveTo, 0, 11, "  Local Date: "
  DEBUG MoveTo, 0, 13, "    Latitude: "
  DEBUG MoveTo, 0, 14, "   Longitude: "
  DEBUG MoveTo, 0, 15, "    Altitude: "
  DEBUG MoveTo, 0, 16, "       Speed: "
```

```
' -----[ Main Code ]-----------------------------------------------------------
'
Main:

  ' wait for $GPRMC string and store data in SPRAM

  SERIN GPSpin, N4800, 3000, No_GPS_Data, [WAIT("GPRMC,"), SPSTR 65]
  GOSUB Parse GPS Data                          ' extract data from SPRAM

  ' wait for GARMIN custom string
  ' -- use DEC to extract altitude

Get_Altitude:
  SERIN GPSpin, N4800, 2000, Show GPMRC String, [WAIT("PGRMZ,"), DEC altitude]

Show GPMRC String:
  DEBUG MoveTo, 0, 6, "$GPRMC,"                 ' print header
  idx = 0                                       ' start at position UTC

Print GPRMC Char:                               ' print the $GPRMC data string
  GET idx, char                                 ' get char from SPRAM
  DEBUG char                                    ' display it
  IF char = "*" THEN Print Checksum             ' look for checksum indicator
  idx = idx + 1                                 ' point to next char
  GOTO Print_GPRMC_Char

Print Checksum:
  GET (idx + 1), char                           ' get first checksum char
  DEBUG char                                    ' display
  GET (idx + 2), char                           ' get second checksum char
  DEBUG char, ClrRt                             ' display, clear to end of line

Show Report:
  DEBUG MoveTo, 14, 8                           ' was the signal valid?
  LOOKUP valid, [NotValid, IsValid], eeAddr     ' get answer from EE
  GOSUB Print_Z_String                          ' print it
  DEBUG ClrRt                                   ' clear end of line
  IF (valid = 0) THEN Signal Not Valid

Signal Is Valid:
  DEBUG MoveTo, 14, 10, DEC2 tmHrs, ":", DEC2 tmMins, ":", DEC2 tmSecs

  DEBUG MoveTo, 14, 11, DEC2 day, " "
  eeAddr = (month - 1) * 4 + MonNames           ' get address of month name
  GOSUB Print Z String                          ' print it
  DEBUG " 20", DEC2 year

  DEBUG MoveTo, 15, 13, DEC2 latDeg, DegSym, " ", DEC2 latMin, MinSym, " "
  DEBUG DEC2 (latSec / 10), ".", DEC1 (latSec // 10), SecSym, " "
  DEBUG "N" + (latNS * 5)
```

```
  DEBUG MoveTo, 14, 14, DEC3 longDeg, DegSym, " ", DEC2 longMin, MinSym, " "
  DEBUG DEC2 (longSec / 10), ".", DEC1 (longSec // 10), SecSym, " "
  DEBUG "E" + (longEW * 18)

  DEBUG MoveTo, 14, 15, DEC altitude, " Feet"
  DEBUG MoveTo, 14, 16, DEC (speed / 10), ".", DEC1 (speed // 10), " MPH "
  GOTO Main

Signal Not Valid:
  DEBUG MoveTo, 14, 10, "?", ClrRt              ' clear all fields
  DEBUG MoveTo, 14, 11, "?", ClrRt
  DEBUG MoveTo, 14, 13, "?", ClrRt
  DEBUG MoveTo, 14, 14, "?", ClrRt
  DEBUG MoveTo, 14, 15, "?", ClrRt
  DEBUG MoveTo, 14, 16, "?", ClrRt
  GOTO Main


' -----[ Subroutines ]-------------------------------------------------------
'
No GPS Data:
  DEBUG CLS, "Error: No GPS data detected"
  PAUSE 2500
  GOTO Initialize                               ' try again


Parse GPS Data:
  idx = 0 : fldWidth = 2                        ' UTC hours
  GOSUB String To Value
  tmHrs = workVal

  idx = 2 : fldWidth = 2                        ' UTC minutes
  GOSUB String To Value
  tmMins = workVal

  idx = 4 : fldWidth = 2                        ' UTC seconds
  GOSUB String_To_Value
  tmSecs = workVal

  idx = 9 : fldWidth = 2                        ' latitude degrees
  GOSUB String To Value
  latDeg = workVal

  idx = 11 : fldWidth = 2                       ' latitude minutes
  GOSUB String To Value
  latMin = workVal

  idx = 14 : fldWidth = 4                       ' latitude fractional minutes
  GOSUB String_To_Value
  latSec = workVal ** $0F5C                     ' x 0.06 --> tenths of seconds
```

```
  idx = 21 : fldWidth = 3                         ' longitude degrees
  GOSUB String To Value
  longDeg = workVal

  idx = 24 : fldWidth = 2                         ' longitude minutes
  GOSUB String_To_Value
  longMin = workVal

  idx = 27 : fldWidth = 4                         ' longitude fractional minutes
  GOSUB String To Value
  longSec = workVal ** $0F5C                      ' x 0.06 --> tenths of seconds

  ' get non-numeric data

Get Valid:
  GET 7, char
  valid = 1                                       ' assume valid
  IF (char = "A") THEN Get_Lat_Dir                ' it is, so skip
  valid = 0                                       ' set to 0 if not valid

Get Lat Dir:
  latNS = 0                                       ' assume North
  GET 19, char                                    ' check it
  IF (char = "N") THEN Get_Long_Dir               ' confirm
  latNS = 1                                       ' set to 1 if South

Get Long Dir:
  longEW = 0                                      ' assume East
  GET 33, char                                    ' check it
  IF (char = "E") THEN Get_Speed                  ' confirm
  longEW = 1                                      ' set to 1 if West

  ' get variable length data

Get Speed:
  idx = 34
  GOSUB Mixed_To_Tenths                           ' convert "xxx.x" to number
  ' speed = workVal                               ' speed in knots (tenths)
  speed = workVal + (workVal ** $2699)            ' x 1.1507771555 for mph

  ' get date
  ' -- past variable data, so we need to use field search

Get Date:
  field = 8                                       ' set field to find
  GOSUB Move To Field                             ' go get position
  PUT 125, idx                                    ' save date position

  fldWidth = 2
  GOSUB String To Value
  day = workVal                                   ' UTC day, 1 - 31
```

```
  GET 125, idx                                  ' get stored position
  idx = idx + 2 : fldWidth = 2
  GOSUB String To Value
  month = workVal                               ' UTC month, 1 - 12

  GET 125, idx                                  ' get stored position
  idx = idx + 4 : fldWidth = 2
  GOSUB String To Value
  year = workVal                                ' UTC year, 0 - 99

  ' adjust date for local position

Correct Local Time Date:
  workVal = tmHrs + UTCfix                      ' add UTC offset
  IF (workVal < 24) THEN Adjust Time            ' midnight crossed?
  workVal = UTCfix                              ' yes, so adjust date
  BRANCH workVal.Bit15, [Location_Leads, Location_Lags]

Location Leads:                                 ' east of Greenwich
  day = day + 1                                 ' no, move to next day
  eeAddr = DaysInMon * (month - 1)              ' get days in month
  READ eeAddr, char
  IF (day <= char) THEN Adjust_Time             ' in same month?
  month = month + 1                             ' no, move to next month
  day = 1                                       ' first day
  IF (month < 13) THEN Adjust Time              ' in same year?
  month = 1                                     ' no, set to January
  year = year + 1 // 100                        ' add one to year
  GOTO Adjust_Time

Location Lags:                                  ' west of Greenwich
  day = day - 1                                 ' adjust day
  IF (day > 0) THEN Adjust Time                 ' same month?
  month = month - 1
  IF (month > 0) THEN Adjust_Time               ' same year?
  month = 1                                     ' no, set to January
  eeAddr = DaysInMon * (month - 1)
  READ eeAddr, day                              ' get new day
  year = year + 99 // 100                       ' set to previous year

Adjust_Time:
  tmHrs = tmHrs + (24 + UTCfix) // 24           ' localize hours
  RETURN

' ****************************************
' Convert string data (nnnn) to numeric value
' -- idx      - location of first digit in data
' -- fldWidth - width of data field (1 to 5)
' -- workVal  - returns numeric value of field
' ****************************************
```

```
String To Value:
  workVal = 0
  IF (fldWidth = 0) OR (fldWidth > 5) THEN String To Value Done

Get_Field_Digit:
  GET idx, char                                 ' get digit from field
  workVal = workVal + (char - "0")              ' convert, add into value
  fldWidth = fldWidth - 1                       ' decrement field width
  IF (fldWidth = 0) THEN String To Value Done
  workVal = workVal * 10                        ' shift result digits left
  idx = idx + 1                                 ' point to next digit
  GOTO Get_Field_Digit

String To Value Done:
  RETURN

' ***************************************************
' Convert string data (nnn.n) to numeric value (tenths)
' -- idx     - location of first digit in data
' -- workVal - returns numeric value of field
' ***************************************************

Mixed_To_Tenths:
  workVal = 0

Get Mixed Digit:
  GET idx, char                                 ' read digit from speed field
  IF (char = ".") THEN Get Mixed Last           ' skip decimal point
  workVal = (workVal + (char - "0")) * 10       ' add digit, move data left
  idx = idx + 1                                 ' point to next digit
  GOTO Get Mixed Digit

Get Mixed Last:
  GET (idx + 1), char
  workVal = workVal + (char - "0")              ' speed in knots
  RETURN

' ***********************************************************
' Find field location after variable-length data (i.e., speed)
' -- field - field number
' -- idx   - returns position of first digit in field
' ***********************************************************

Move To Field:
  idx = 0
  IF (field = 0) THEN Move To Field Done        ' if zero, we're there

Get_Char:
  GET idx, char                                 ' get char from SPRAM
  IF (char = Comma) THEN Found_Comma            ' is it a comma?
```

```
  idx = idx + 1                                 ' no, move to next char
  GOTO Get Char

Found Comma:
  field = field - 1                             ' was comma, dec field coutner
  idx = idx + 1                                 ' point to next char
  IF (field = 0) THEN Move_To_Field_Done        ' if field = 0, we're there
  GOTO Get Char

Move To Field Done:
  RETURN

' ****************************************
' Print Zero-terminated string stored in EEPROM
' -- eeAddr - starting character of string
' ****************************************

Print_Z_String:
  READ eeAddr, char                             ' get char from EE
  IF (char = 0) THEN Print Z String Done        ' if zero, we're done
  DEBUG char                                    ' print the char
  eeAddr = eeAddr + 1                           ' point to the next one
  GOTO Print Z String

Print_Z_String_Done:
  RETURN
```