PARALLAX SEMICONDUCTOR
www.parallaxsemiconductor.com
sales@parallaxsemiconductor.com
support@parallaxsemiconductor.com
phone: 916-632-4664 • fax:916-624-8003

Application Note AN002

Real-time GPS Data Reception and Parsing

Abstract: Use the multicore Propeller P8X32A to receive, parse, and display GPS data in real time. The gps_basic.spin object supports the most common NMEA 0183 sentences ($GPRMC & $GPGGA) at 4800 baud, with a configuration option for higher baud rates.

Introduction

The proliferation of consumer GPS products has provided engineers with a wide variety of low-cost, high-quality GPS modules that are ideally suited for embedded location and navigation applications. Embedded and hand-held GPS devices provide raw output through a serial connection in the form of comma delimited, CrLf (carriage return/line feed) terminated NMEA strings, typically at 4800 baud. Each string begins with a unique identifier and contains one or more fields; for example:

```
$GPRMC,032606,A,3410.2358,N,11819.0865,W,0.0,207.2,180211,13.5,E,A*32
```

An embedded GPS application requires two support processes: 1) reception and buffering of raw strings from the GPS receiver and, 2) parsing select elements from target GPS strings as required by the application.

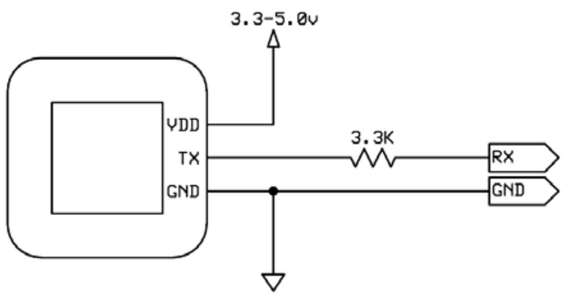
In many processors, a hardware UART and buffer handle the reception of raw GPS data. The P8X32A silicon does not include a hardware UART. This is easily overcome by programming a cog (individual 32-bit processor) to create a virtual UART. The advantage of this approach is that the virtual UART may be configured as desired by the developer and is not restricted in terms of I/O connections, features, buffer size, and performance like its fixed silicon counterpart.

With raw NMEA strings received and buffered, the application must locate the target string(s) for field extraction. The GPS object presented here (gps_basic.spin) devotes one of the P8X32A cogs to NMEA string identification and removal from the serial buffer for field parsing, allowing the main application object to focus on real-time activities.

Hardware Connections

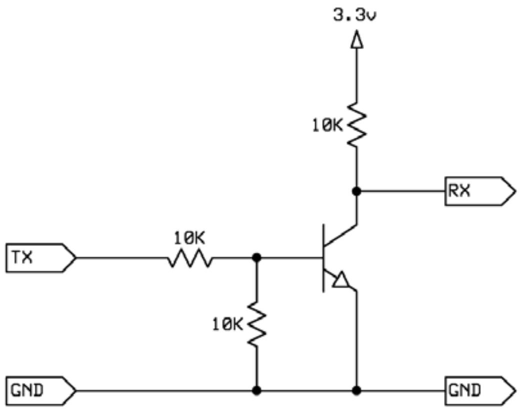
The output of the GPS receiver will be TTL or RS-232 serial data at 4800 baud. With a TTL device, the connections in Figure 1 are appropriate. The 3.3 kΩ series resistor limits the current into the P8X32A I/O pin and allows the application to use a 3.3 V or 5.0 V GPS receiver module.

Figure 1: TTL Connection



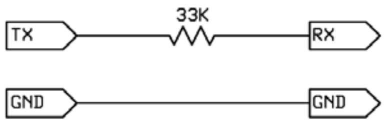
Many hand-held GPS units provide RS-232 output. A MAX3232A or similar device can invert and level-shift the signal to the P8X32A. When the only level translation requirement of the application is the GPS stream, the circuit in Figure 2 works well, with lower component cost and PCB real estate use.

Figure 2: Low Cost RS-232 Interface



As suggested above, the advantage of a software UART is that it can be more flexible than a hardware counterpart. The UART code used in this application supports configuration for RS-232 data inversion. The connection illustrated in Figure 3 is appropriate when using the UART in this mode.

Figure 3: Alternate RS-232 Interface for Software UART



Software Connection and Use

Instantiation of the GPS object is minimally based on the I/O pin used and the location offset from Greenwich Mean Time.

```
gps.start(GPS_RX, PST)
```

This form configures the UART for TTL level, true mode input at 4800 baud, with a 2500 millisecond timeout period for loss-of-GPS data detection. RS-232 input requires an alternate method, with the baud rate specified as a negative value and the timeout period specified in milliseconds.

```
gps.startx(GPS_RX, PST, -4800, 2500)
```

The **startx** method is also suitable for newer GPS modules that provide the NMEA sentences at higher baud rates.

```
gps.startx(GPS_RX, PST, 9600, 1250)
```

GPS Object Methods

The following public methods are available to the application using the gps_basic.spin object.

`start(pin, time_zone_offset)` — The `start` method requires the I/O pin used for the RX UART and the location offset from Greenwich Mean Time. This method configures the UART for 4800 baud, true mode, with a loss-of-signal timeout of 2500 milliseconds.

`startx(pin, time_zone_offset, [-]baud, timeout_ms)` — The `startx` (explicit) method requires the I/O pin used for the RX UART, the location offset from Greenwich Mean Time, the baud rate (negative for inverted serial data), and the loss-of-signal timeout value in milliseconds.

`stop` — The `stop` method terminates the software UART and GPS parsing cogs. This method also clears the GPS buffers, causing any method call after the use of `stop` to return zero or pointer to an empty z-string.

`hasgps` — Returns True when the UART is actively receiving serial data from the GPS module. If the GPS stream is lost for a period that exceeds the timeout value this method returns False.

`s_gpsfix` — Returns a pointer to a z-string that contains "Invalid", "GPS", or "DGPS" indicating GPS fix quality. The pointer is suitable for use with the `str` method provided by many serial objects.

`n_gpsfix` — Returns an integer, 0 to 2, indicating GPS fix quality. 0: No fix; 1: GPS fix; 2: DGPS fix

`s_satellites` — Returns a pointer to a 1- or 2-digit z-string indicating the number of satellites in view of the GPS receiver.

`n_satellites` — returns a decimal value indicating the number of satellites in view of the GPS receiver.

`s_utc_time` — Returns a pointer to a 6-character z-string that contains the UTC time (Greenwich Mean Time) in the form "HHMMSS".

`s_local_time` — Returns a pointer to a 6-character z-string that contains the local time based on the UTC offset specified in the `start` or `startx` method.

`fs_utc_time` — Returns a pointer to a formatted, 8-character z-string that contains the UTC time (Greenwich Mean Time) in the form "HH:MM:SS".

`fs_local_time` — Returns a pointer to a formatted, 8-character z-string that contains the local time based on the UTC offset specified in the `start` or `startx` method.

`s_utc_hrs` — Returns a pointer to a 2-character z-string that contains the UTC time (Greenwich Mean Time) hours in the form "HH".

`n_utc_hrs` — Returns a decimal value, 0 to 23, that contains the UTC time (Greenwich Mean Time) hours.

`s_local_hrs` — Returns a pointer to a 2-character z-string that contains the local time hours in the form “HH”.

`n_local_hrs` — Returns a decimal value, 0 to 23, that contains the local time hours.

`s_mins` — Returns a pointer to a 2-character z-string that contains the time field minutes in the form “MM”.

`n_mins` — Returns a decimal value, 0 to 59, that contains the local time minutes.

`s_secs` — Returns a pointer to a 2-character z-string that contains the time field seconds in the form “SS”.

`n_secs` — Returns a decimal value, 0 to 59, that contains the local time seconds.

`s_date` — Returns a pointer to a 6-character z-string that contains the UTC date in the form “DDMMYY”.

`fs_date` — Returns a pointer to a formatted, 8-character z-string that contains the UTC date in the form “DD/MM/YY”.

`s_day` — Returns a pointer to a 2-character z-string that contains the UTC day in the form “DD” (“01” to “31”).

`n_day` — Returns a decimal value, 1 to 31, that contains the UTC day.

`s_month` — Returns a pointer to a 2-character z-string that contains the UTC month in the form “MM” (“01” to “12”).

`n_month` — Returns a decimal value, 1 to 12, that contains the UTC month.

`s_year` — Returns a pointer to a 2-character z-string that contains the UTC year in the form “YY” (“00” to “99”).

`n_year` — Returns a decimal value, 0 to 99, that contains the UTC year.

`s_latitude` — Returns a pointer to an 11-character z-string that contains the GPS latitude in the form “DDMM.SSSS X” where “X” is “N” for north or “S” for south.

`n_latsign` — Returns a signed value, 1 for North, -1 for South, indicating the latitude hemisphere.

`s_latd` — Returns a pointer to a 2-character z-string that contains the latitude degrees in the form “DD” (“00” to “90”).

`n_latd` — Returns a decimal value, 0 to 90, that contains the latitude degrees.

`s_latm` — Returns a pointer to a 2-character z-string that contains the latitude minutes in the form “MM” (“00” to “59”).

`n_latm` — Returns a decimal value, 0 to 59, that contains the latitude minutes.

`n_lats` — Returns a decimal value, 0 to 59, that contains the latitude seconds, calculated from the fractional seconds provided in the GPS latitude field.

`s_longitude` — Returns a pointer to a 12-character z-string that contains the GPS longitude in the form "DDDMM.SSSS X" where "X" is "E" for east or "W" for west.

`n_lonsign` — Returns a signed value, 1 for East, -1 for West, indicating the longitude direction from the prime meridian.

`s_lond` — Returns a pointer to a 3-character z-string that contains the longitude degrees in the form "DDD" ("000" to "180").

`n_lond` — Returns a decimal value, 0 to 180, that contains the longitude degrees.

`s_lonm` — Returns a pointer to a 2-character z-string that contains the longitude minutes in the form "MM" ("00" to "59").

`n_lonm` — Returns a decimal value, 0 to 59, that contains the longitude minutes.

`n_lons` — Returns a decimal value, 0 to 59, that contains the latitude seconds, calculated from the fractional seconds provided in the GPS longitude field.

`s_spdk` — Returns a pointer to a variable-length z-string that contains the GPS speed in knots.

`n_spdk` — Returns a decimal value indicating GPS speed in 0.1 knots (e.g., 1000 = 100.0 knots).

`n_spdm` — Returns a decimal value indicating GPS speed in 0.1 mph (e.g., 650 = 65.0 mph).

`s_bearing` — Returns a pointer to a variable-length z-string that contains the GPS bearing, "0.0" to "359.9".

`n_bearing` — Returns a decimal value indicating GPS bearing in 0.1 degrees.

`s_altm` — Returns a pointer to a variable-length z-string that contains the GPS altitude in meters, "0.0" to "9999.9".

`n_altm` — Returns a decimal value indicating GPS altitude in 0.1 meters.

`n_altf` — Returns a decimal value indicating GPS altitude in 0.1 feet.

Customizing the GPS Object

As presented, the gps_basic.spin object detects and parses the $GPRMC and $GPGGA strings from the serial stream. To extract a field contained in another string the following steps are required.

- Add a working buffer for the target string (VAR section)
- Add detection/buffering of the target string to the **parse_gps** method
- Add appropriate methods to extract and return desired field data

Example

The GPS altitude is contained in field 9 of the $GPGGA sentence. A working buffer in the VAR section holds the complete string.

```
var
  byte  ggawork[WORK_SIZE]
```

The **parse_gps** method, which runs independently in a separate cog, detects and moves target NMEA strings to the appropriate working buffer. The method begins by clearing a temporary buffer. After detection of the "$" which indicates the start of a string header, subsequent characters are moved to the temporary buffer until the detection of a carriage return (13) which indicates the end of the string.

The first field of the newly captured string is compared with known targets, and if a match is found the temporary buffer is copied to the working buffer for the target string. (See project code for annotated source.)

```
pri parse_gps | ok, c, len

  repeat
    ok := true

    bytefill(@gpswork, 0, WORK_SIZE)
    repeat
      c := rxtime(rxtimeout)
      if (c < 0)
        ok := false
        quit
    until (c == "$")

    if ok
      len := 0
      repeat
        c := rxtime(rxtimeout)
        if (c < 0)
          ok := false
          quit
        if (c <> 13)
          gpswork[len++] := c
        else
          quit

    if ok
      if (strncmp(@RMC_HDR, @gpswork, 6) == 0)
        bytemove(@rmcwork, @gpswork, len)
      elseif (strncmp(@GGA_HDR, @gpswork, 6) == 0)
        bytemove(@ggawork, @gpswork, len)
```

The **s_altm** method returns a pointer to a z-string that holds the GPS altitude in 0.1 meters.

```
pub s_altm

  bytefill(@gpsrslt, 0, RSLT_SIZE)
  gps_fcopy(@gpsrslt, @ggawork, 9)

  return @gpsrslt
```

Most GPS field methods begin by clearing the result string buffer with zeroes. This allows subsequent code to move characters to the field without having to explicitly add termination (0). The **gps_fcopy** method moves the target field (*9*) from the NMEA string working register (**ggawork**) to the result string (**gpsrslt**). Finally, the **s_altm** method returns a pointer to that string for use by the application.

For GPS methods that return a numeric value the **str2dec** method does the conversion. In the example below, the **n_satellites** method calls the **s_satellites** method to extract field 7 from the $GPGGA string, then uses **str2dec** to return the numeric value of this string to the application. If there is no GPS data in the working buffer the **str2dec** method returns zero.

```
pub s_satellites

  bytefill(@gpsrslt, 0, RSLT_SIZE)

  if (strncmp(@GGA_HDR, @ggawork, 6) == 0)
    gps_fcopy(@gpsrslt, @ggawork, 7)

  return @gpsrslt


pub n_satellites

  s_satellites

  return str2dec(@gpsrslt, 2)
```

Optimization

As illustrated above, most numeric methods call an associated string method to extract the required field from a NMEA string. As all methods are intended to be atomic, each will extract the required field, on occasion leading to small inefficiencies.

For example, the following code supplies the application with local time fields in numeric form:

```
  hr := gps.n_local_hrs
  mn := gps.n_mins
  sc := gps.n_secs
```

Each of the method calls extract the time field from the $GPRMC string. The GPS object exposes the address of the field result string through the **rslt_pntr** method. By using this as the source string for the **str2dec** method, numeric values can be derived without redundant retrieval of the GPS field.

```
gps.s_local_time
hr := gps.str2dec(gps.rslt_pntr+0, 2)
mn := gps.str2dec(gps.rslt_pntr+2, 2)
sc := gps.str2dec(gps.rslt_pntr+4, 2)
```

As illustrated above the **s_local_time** method moves the unformatted local time field to the result string. With the time string captured the **str2dec** method is used to extract field values, using the offset within that string (0 for hours, 2 for minutes, 4 for seconds) and the number of characters to convert (2).

Note: The result string is intended for immediate use. Subsequent calls to any field-based methods will modify this string. If a copy is required the **s_copy** method is available to move the contents of the result string to another array.

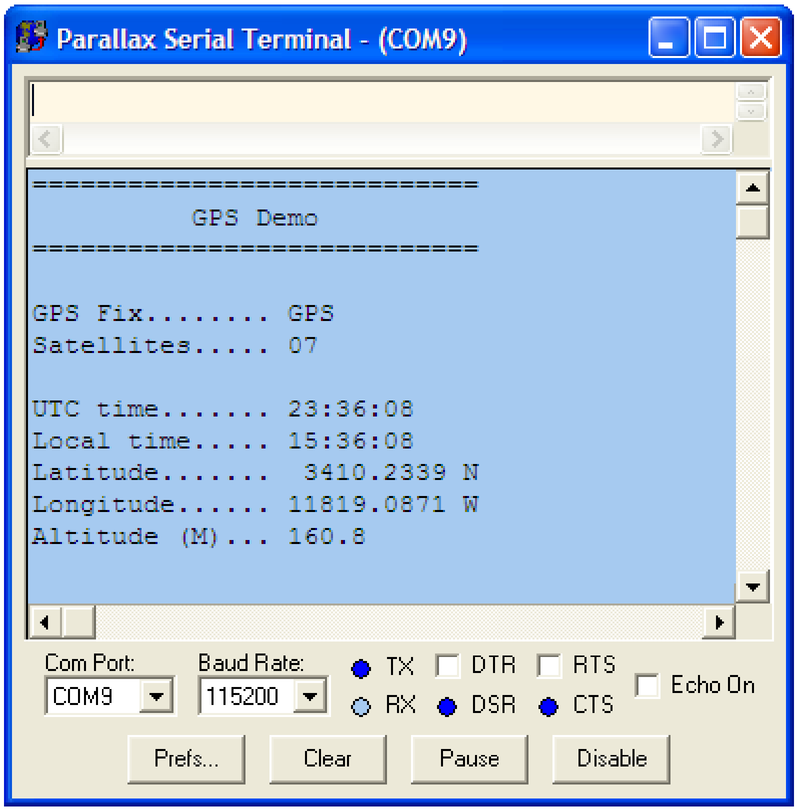
```
gps.s_copy(@mylocaltime)
```

The destination parameter for **s_copy** is a pointer to an array of at least 20 bytes.

Example Application

The attached application (gps_basic_demo.spin) displays GPS fix quality, the number of satellites in view, UTC time, and local time and location information using the Parallax Serial Terminal (Figure 4).

Figure 4: Parallax Serial Terminal displays output of gps_basic_demo.spin



```
============================
         GPS Demo
============================


GPS Fix........ GPS
Satellites..... 07

UTC time....... 23:36:08
Local time..... 15:36:08
Latitude.......  3410.2339 N
Longitude...... 11819.0871 W
Altitude (M)... 160.8
```

Resources

A zip file containing the following items is available from this application note's page at: www.parallaxsemiconductor.com/AN002

gps_basic.spin
gps_basic_demo.spin
FullDuplexSerial.spin

References

1. NMEA sentence definitions: http://aprs.gids.nl/nmea

Revision History

Version 1.0: original document.