GPS Wildcard User Guide

Version 1.0
February, 2007

GPS Wildcard User Guide

The GPS Wildcard enables your instrument to know where it is, how fast it is moving, the direction it is moving, and the current time and date. Latitude, longitude, altitude, speed, course heading, universal (Greenwich) and local time and date, position error estimates, GPS fix quality, and the number of satellites in view and in use are all available to your application program. The Wildcard incorporates a GPS subsystem made by industry leader Garmin, and includes a connection for a remote-mounted active antenna for good signal reception. This tiny 2" by 2.5" board is a member of the Wildcard™ series that connects to Mosaic controllers.

This document describes the capabilities of the GPS Wildcard, tells how to configure the hardware, and presents an overview of the GPS driver software that runs on the Mosaic controller. A glossary of the Mosaic software device driver functions, demonstration program source code, and hardware schematics are presented.

GPS Overview

The Global Positioning System (GPS) relies on more than two dozen GPS satellites orbiting the Earth. Each satellite transmits signals including a very precise clock plus satellite position information that allows a GPS receiver to determine its location, speed and direction. A GPS receiver calculates its position by measuring the distance between itself and three or more GPS satellites. Because the signal from the GPS satellite travels at a known speed, measuring the time delay between transmission and reception of each GPS radio signal allows the receiver to calculate the distance to each satellite. By determining the position of, and distance to, at least three satellites, the receiver can compute its position. GPS receivers typically do not have perfectly accurate clocks and therefore track one or more additional satellites to correct the receiver's clock error.

The GPS signal is circularly polarized with a typical center frequency of 1.575 gigahertz. For best results, an "active antenna" amplifies the signal before delivering it via a coaxial cable to the GPS receiver. The GPS antenna should be mounted in a position that has a clear view of the sky so that the satellite signals can be received. GPS receivers typically do not work inside buildings, and may have trouble in vehicles or areas where the view of the sky is obstructed by tall buildings or dense foliage. Operating in urban areas can confuse the receiver due to signal reflections off buildings that result in "multipath" effects that lengthen the time it takes for a signal to reach the receiver, causing errors in the reported position.

Some GPS receivers have a built-in "Wide Area Augmentation System", or "WAAS", that improves accuracy using a network of ground stations to provide additional information to the GPS receiver. The Garmin GPS-15 receiver on the GPS Wildcard is not WAAS enabled.

When a GPS receiver is first turned on, it must go through an "acquisition" process to locate and lock onto the satellite signals. The more initial information the GPS has about its location and time, the faster the acquisition process. A "warm" acquisition occurs when the initial receiver location, time, and satellite location (ephemeris) data is known. A "cold" acquisition occurs when the initial receiver location, and time is known, but the satellite location data is unknown. If even less information is known, a “sky search” is required to complete the acquisition process and obtain a position “fix”.

The Garmin GPS Subsystem

The Garmin GPS subsystem specifications are summarized in Table 1-1.

Table 1-1 Garmin GPS-15 subsystem specifications.

Property	Value
Receiver	12 channel receiver computes and updates position information
Current Draw	75 mA + antenna draw; 200 mA for entire Wildcard, drawn from +5V supply
Warm Acquisition	About 15 seconds
Cold Acquisition	About 45 seconds
Sky Search Acquisition	5 minutes
Update Rate	One NMEA-0183 frame (6 or 7 sentences) per second at 4800 baud
Position Accuracy	< 15 meters, 95% typical
Velocity Accuracy	0.1 knot RMS steady state
Dynamics	999 knots (only limited at altitude > 60000 feet), 6g acceleration, <6g jerk
Required Active Antenna	10-40 dB gain, MCX male connector, total noise < 7 dB, powered from +3.3V GPS supply

The key component of this Wildcard is the GPS-15 subsystem manufactured by Garmin, the industry leader in GPS instrumentation. It implements a 12-channel GPS receiver, and includes a rechargeable backup battery to maintain the contents of the subsystem’s clock and memory for up to 21 days in the absence of external power. The battery recharges whenever power is applied to the unit. The GPS subsystem outputs standard NMEA (National Marine Electronics Association) ASCII strings, called “sentences”, using the NMEA 0183 Version 2.20 protocol. Each sentence reports information about one or more GPS parameters such as location, speed, heading (direction), time, or position error estimates. A “frame” of 6 to 7 sentence types is output each second at 4800 baud by the subsystem. The NMEA sentence format is explained in more detail below.

This Garmin GPS subsystem is encased in a shielded metal enclosure, and includes a female MCX connector for an active GPS antenna. Mosaic sells a mating antenna with a male MCX connector that can be mounted in a location that provides a good view of the sky for optimal operation.

The Garmin GPS subsystem performs a “warm” satellite acquisition in about 15 seconds, and a “cold” acquisition in about 45 seconds. These relatively fast acquisitions rely on initial position and time data that is stored in the battery-backed RAM in the subsystem. If the stored information has been lost or is not correct (for example, if the internal backup battery is discharged or the receiver has been transported to a new location since it was last used), then a “sky search” acquisition is performed, requiring up to 5 minutes to acquire a fix.

The position accuracy of the GPS-15 subsystem is better than 15 meters 95% of the time. The velocity accuracy is 0.1 knot if the receiver is moving at a steady speed. A “knot” is a nautical mile per hour, equal to 1.152 miles per hour (MPH). The maximum reportable speed is 999.9 knots.

GPS Wildcard Hardware

The picture in Figure 1-1 illustrates the Wildcard’s hardware.

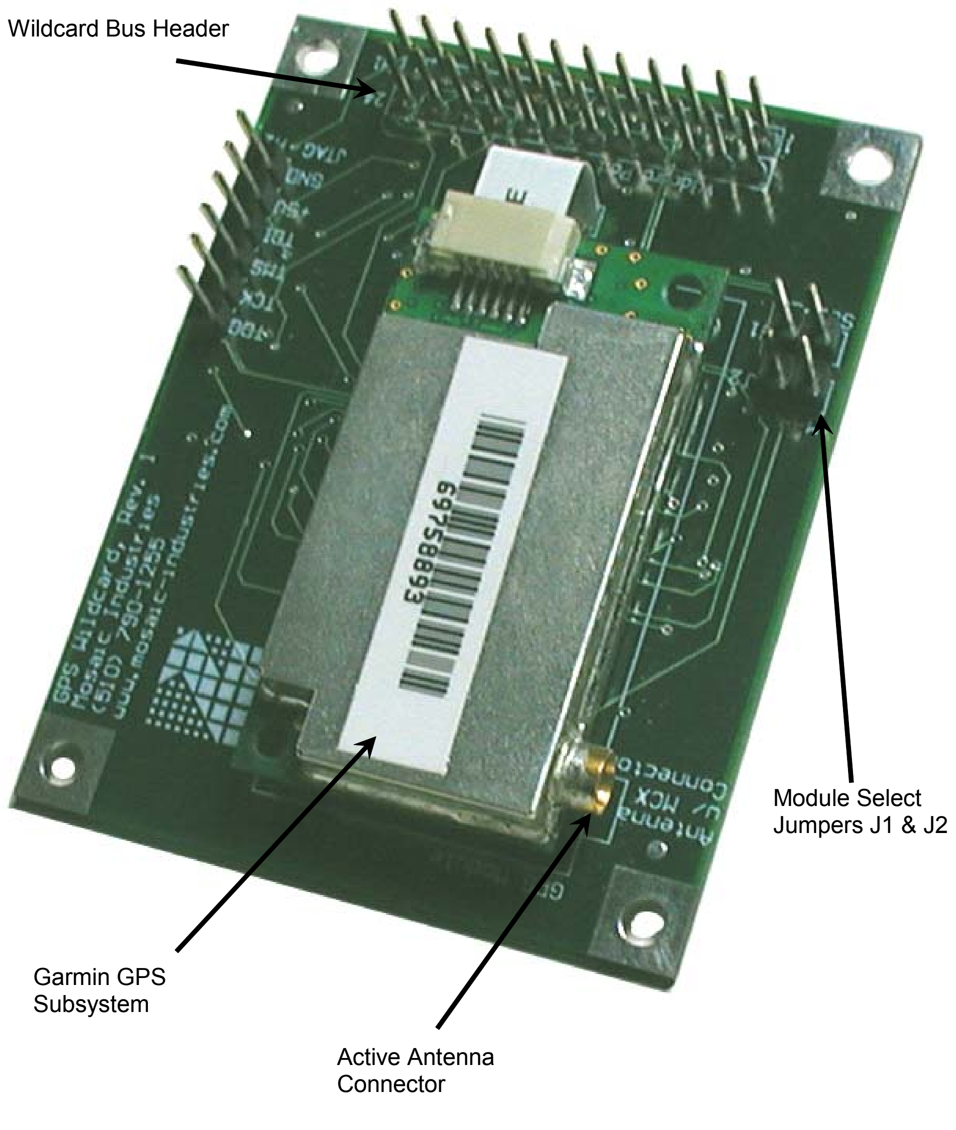


Figure 1-1 The GPS Wildcard.

As shown in the Figure, the GPS Wildcard comprises a Wildcard bus header, a Garmin GPS-15 subsystem with active antenna connector, a UART chip, and digital logic circuitry. Jumpers enable address selection for the card. The Wildcard bus header interfaces to the host processor (QCard, QScreen, Handheld, or PDQ series controller). A flex cable passes through a slot in the board to interface the Garmin GPS subsystem to the UART serial-to-parallel converter chip and power conditioning circuitry on the bottom of the Wildcard.

As shown in Figure 1-1, the Garmin GPS subsystem is encased in a shielded metal enclosure, and includes a female MCX connector for an active GPS antenna. The GPS will not work unless an active antenna is attached. Mosaic sells a mating antenna with a male MCX connector that can be mounted in a location that provides a good view of the sky for optimal operation.

Connecting To the Wildcard Bus

To connect the GPS Wildcard to the Wildcard bus on the controller board:

With the power off, connect the female 24-pin side of the stacking go-through Wildcard bus header on the bottom of the GPS Wildcard to Wildcard Port 0 or Wildcard Port 1 on the controller or its mating Docking Panel (formerly the “PowerDock”). The corner mounting holes on the Wildcard should line up with the standoffs on the controller board. The Wildcard ports are labeled on the silkscreen of the controller board. Note that the GPS Wildcard headers are configured to allow direct stacking onto the controller board, even if other Wildcards are also installed. Do not use ribbon cables to connect the GPS Wildcard to the Wildcard bus.

CAUTION: The Wildcard bus does not have keyed connectors. Be sure to insert the Wildcard so that all pins are connected. The Wildcard bus and the GPS Wildcard can be permanently damaged if the connection is done incorrectly.

Selecting the Wildcard Address

Once you have connected the GPS Wildcard to the Wildcard bus, you must set the address of the Wildcard (also called the *module number*) using jumper shunts across J1 and J2.

The Wildcard Select Jumpers, labeled J1 and J2, select a 2-bit code that sets a unique address on the Wildcard port of the controller board. Each Wildcard port on the controller accommodates up to 4 Wildcards. Wildcard Port 0 provides access to Wildcards 0-3 while Wildcard Port 1 provides access to Wildcards 4-7. Two Wildcards on the same port cannot have the same address (jumper settings). Table 1-2 shows the possible jumper settings and the corresponding addresses.

Table 1-2 Wildcard address jumper settings.

Wildcard Port	Wildcard Address	Installed Jumper Shunts
0	0	None
	1	J1
	2	J2
	3	J1 and J2
1	4	None
	5	J1
	6	J2
	7	J1 and J2

Mosaic GPS Driver Software

A package of pre-coded device driver functions is provided to make it easy to control the GPS Wildcard. This code is available as a pre-compiled "kernel extension" library to C and Forth programmers. Both C and Forth source code versions of a demonstration program are provided. The demo program illustrates how to initialize and use the GPS Wildcard in a multitasking application.

In the following sections we examine the formatted data that is transmitted by the Garmin GPS subsystem, and summarize how the Mosaic GPS driver software extracts the data and makes the GPS information available to your application program. The **gps_info** data structure that holds the collected GPS information is described, and a comprehensive glossary of functions is presented.

Driver Functions Extract GPS Data from NMEA Sentences

The NMEA (National Marine Electronics Association) 0183 standard specifies the format of ASCII serial data output by the GPS receiver. On the GPS Wildcard, this serial data is transmitted by the Garmin GPS subsystem and received by a UART (Universal Asynchronous Receiver Transmitter) chip. The processor can read the data out of the UART receive buffer via the Wildcard bus, and the driver functions facilitate extraction of the GPS parameters from the serial data stream.

The GPS serial data is organized into "sentences". Each sentence starts with the ASCII '$' character, followed by the sentence name field, followed by the data fields, followed by an ASCII '*' character and the hexadecimal ASCII checksum. The checksum is calculated as the 8-bit exclusive-OR sum of all ASCII characters between the '$' and '*' characters (not including the '$' and '*'). The sentence is terminated by the carriage return (0x0D) and linefeed (0x0A) characters. There is a comma after the name field, and after each data field except the last one before the checksum. Any alphabetic letters in the sentence must be upper case. In the Garmin implementation of the NMEA protocol, a sentence may contain up to 82 characters.

Table 1-3 summarizes the four NMEA sentences from which data is extracted by the GPS Wildcard driver software. Each of these sentences is discussed in turn.

Table 1-3 GPS sentences parsed by the GPS Wildcard driver software.

Sentence	Description	Fields Extracted by Mosaic Driver
GPRMC	Recommended minimum specific GPS data	UTC and local time/date, position valid flag, latitude N/S, longitude E/W, ground speed, ground course, magnetic variation E/W
GPGGA	Global positioning fix data	Fix available flag, number of satellites in use, altitude
GPGSV	GPS satellites in view	Number of satellites in view
PGRME	Garmin estimated error info	Estimated horizontal, vertical and overall position errors

The GPRMC sentence delivers the fundamental position, time, course, and direction information. The Mosaic software driver extracts all of the fields from this sentence and stores them in the `gps_info` data structure described below. For those who are interested in the details of the sentence structure, the format is as follows:

```
$GPRMC,<hhmmss_utc>,<A|V>,<ddmm.mmmm_lat>,<N|S>,<dddmm.mmmm_long>,<E|W>,
<kkk.k_knots>,<ddd.d_degrees_course>,<ddmmyy>,<ddd.d_mag_var>,<E|W_mag_var_dir>*hhcrlf
```

After the sentence name, the universal time is presented in hours, minutes and seconds. The "Coordinated Universal Time" is somewhat surprisingly abbreviated as "UTC"; it is the time in Greenwich England that serves as the global time reference. The next field is a single-character 'A' or 'V', where A indicates a valid position, and 'V' indicates an invalid position warning. The next field is the latitude expressed as a 2-digit number of degrees in the range 00 to 89 degrees, followed by the number of minutes expressed with 2 digits to the left of the decimal point and 4 digits to the right in the range 00.0000 to 59.9999 minutes. The next field is 'N' or 'S' to indicate North or South latitude (relative to the equator). The longitude is expressed in similar fashion, except that there are 3 longitude degree digits in the range 000 to 179 degrees, followed by the longitude minutes and the 'W' or 'E' hemisphere. The longitude hemisphere is relative to the prime meridian which passes through Greenwich England. The next field is the ground speed in the range 000.0 to 999.9 knots (a knot is 1.152 miles per hour). The next field is the course (also known as the heading or direction) expressed in degrees with respect to true north in the range 000.0 to 359.9 degrees. The next field is the UTC date expressed as a 2-digit date (01-31), 2-digit month (01-12), and 2-digit year (00-99). The next field is the magnetic variation in degrees in the range 000.0 to 180.0 degrees; this variation quantifies the difference between "true north" and the "magnetic north" indicated by a compass. The next field is the direction of the magnetic variation; westerly magnetic variation adds to the true course. The '*' checksum indicator, hexadecimal ASCII checksum, and carriage return/linefeed complete the sentence.

As explained in detail below, the function `GPS_Frame_Extract` and its calling function `GPS_Update` automatically parse this sentence, extract the fields, and store them in the `gps_info` data struct so your application program can use the data. The fields are stored as binary quantities. As summarized in the glossary, a family of additional driver functions is available to fetch the fields out of the `gps_info` data struct and return them for use in your program.

The GPGGA sentence describes whether a GPS fix is available, the number of satellites in use, and the current altitude of the GPS antenna in meters. For those interested in the sentence structure, here it is:

```
$GPGGA,<hhmmss_utc>,<ddmm.mmmm_lat>,<N|S>,<dddmm.mmmm_long>,<E|W>,<0|1>,
<00-12_numsats_in_use>,<0.5-99.9_horiz_DOP>,<antenna_height>,M,
<geoidal_hght>,M,,*hhcrlf
```

After the GPGGA sentence name is the UTC time field, the latitude field, the latitude hemisphere, the longitude, and the longitude hemisphere. All of these are redundant with information from the GPRMC sentence and are ignored. The next field is an ASCII '0' or '1', where 0 means that the GPS fix is not available, and 1 means that a standard non-differential GPS fix is available. The next field is the 2-digit number of satellites in use. We ignore the following "dilution of precision" field. The next field is the antenna height (altitude) in meters expressed as a number in the range -9999.9 to 99999.9. The following field is an ASCII 'M' to indicate that the altitude is expressed in meters. The final fields present the geoidal height in meters, followed by the 'M' for meters, followed by two empty fields, hex checksum, and the carriage return linefeed terminators. Only the fix availability indicator, number of satellites in use, and altitude fields are extracted by `GPS_Frame_Extract` and its calling function `GPS_Update`. These fields are stored in the `gps_info` data struct. Your application program can access them directly, or invoke a family of additional driver functions to fetch the fields.

The GPGSV sentence is used to extract a single field: the number of satellites in view. The sentence structure is as follows:

```
$GPGSV,<num_gpgsv_sentences>,<index_of_this_sentence>,<00-12_numsatinview>,
<01-32_sat_prn>,<00-90deg_sat_elev>,<000-359_sat_azim>,<00-99db_snr>,
<repeat the above line for each satellite up to max of 4 per sentence>*hhcrlf
```

After the sentence name is a field that tells the total number of GPGSV sentences, followed by a sentence index as described below. The next field is a 2-digit number of satellites visible to the GPS receiver in the range 00 to 12; this field is extracted by the driver functions. If the GPS receiver is inside a building, this field will typically report zero satellites in view, while in unobstructed outside situations all 12 GPS receiver channels may have a satellite in view. The remainder of the sentence presents satellite-specific elevation, azimuth and signal-to-noise ratio information that is ignored by the Mosaic GPS driver.

In a single 1-second frame of sentences from the GPS subsystem, there is one GPGSV sentence describing up to 4 of the satellites in view; let's say this is the GPGSV statement with sentence index 1. If there are more than 4 satellites in view, the next frame includes a GPGSV sentence describing up to 4 of the additional satellites in view, say with index 2. If there are more than 8 satellites in view, the next frame will include GPGSV sentence with index = 3 to describe the remaining satellites, and the process starts again with GPGSV index 1 present in the subsequent frame. This is explained further in the commentary describing Listing 1-1 on page 8.

The PGRME sentence is a "proprietary" statement created by Garmin to describe the estimated position errors. The sentence structure is as follows:

```
$PGRME,<0.0-999.9meter_hor_pos_err>,M,<0.0-999.9meter_vert_pos_err>,M,
<0.0-999.9meter_overall_position_err>,M*hhcrlf
```

After the sentence name, the horizontal position error field presents a number in the range 0.0 to 999.9, followed by an ASCII 'M' field to indicate that the units are meters. In parallel fashion, the vertical position error and overall position error fields and their 'M' unit fields are presented. These estimated errors are extracted by `GPS_Frame_Extract` and its calling function `GPS_Update`, and are stored in the `gps_info` data struct so your application program can access them

Three additional sentences are transmitted by the GPS subsystem: GPGSA (dilution of precision report), PGRMT (sensor status information, output only once per minute), and PGRMM (name of map currently in use; the default is WGS84). The Mosaic GPS Wildcard driver software does not extract data from these sentences.

Table 1-4 summarizes the sentences transmitted by the GPS subsystem each second, and shows which provide data extracted by the GPS driver functions. The NMEA sentences are listed in the order in which they are transmitted, but note that the PGRMT sentence is transmitted only once per minute, while the other 6 sentences are transmitted every second.

Listing 1-1 presents a sample of four 1-second frames of data as transmitted by the Garmin GPS subsystem. Carriage returns have been inserted after each frame for clarity; the actual GPS output does not include any blank lines. You can generate a listing similar to this by passing the Wildcard address and local hour offset to `GPS_Init` and then invoking the `GPS_Dump` function. Your controller and GPS Wildcard will then stream the raw NMEA sentence frames to your Mosaic terminal window.

There are a few key items to notice while inspecting Listing 1-1. Notice that the first 3 frames shown contain 6 sentences, while the last one contains 7, including the PGRMT sentence which is transmitted only once per minute (that is, once every 60 frames). Also note while there is only one GPGSV sentence present in each frame of data, there are 3 distinct GPGSV sentences that present themselves in successive frames. The first numeric field of each GPGSV sentence is the number of distinct GPGSV sentences, and the second numeric field is the GPSSV sentence index. Each GPGSV sentence summarizes the information about a maximum of 4 satellites, and there can be as many as 12 satellites in use. The Mosaic driver software extracts only the number of satellites in view from this sentence, and this information is present in each instance of the sentence. The satellite specific information is not extracted or stored.

Table 1-4 Summary of NMEA sentences transmitted by GPS sensor.

Sentence	Description	Data Extracted?
GPRMC	Recommended minimum specific GPS data	Yes
GPGGA	Global positioning fix data	Yes
GPGSA	GPS DOP (dilution of precision) and active satellites	No
GPGSV	GPS satellites in view	Yes
PGRME	Garmin estimated error info	Yes
PGRMT	Garmin sensor status info (output only once per minute)	No
PGRMM	Garmin name of map datum currently in use (default = WGS 84)	No

Listing 1-1 Four 1-second frames of NMEA sentences from the GPS.

```
$GPRMC,223321,A,3732.3791,N,12201.1625,W,0.0,194.8,220207,15.1,E*5A
$GPGGA,223321,3732.3791,N,12201.1625,W,1,08,1.1,11.0,M,-28.0,M,,*45
$GPGSA,A,3,03,,07,09,,16,18,19,21,,,26,1.7,1.1,1.3*3A
$GPGSV,3,2,12,14,24,188,00,16,13,244,39,18,71,018,46,19,15,319,48*7B
$PGRME,5.0,M,5.8,M,7.7,M*26
$PGRMM,WGS 84*06

$GPRMC,223322,A,3732.3792,N,12201.1625,W,0.0,194.8,220207,15.1,E*5A
$GPGGA,223322,3732.3792,N,12201.1625,W,1,08,0.9,11.1,M,-28.0,M,,*4D
$GPGSA,A,3,03,,07,09,,16,18,19,21,,,26,1.6,0.9,1.3*32
$GPGSV,3,3,12,21,56,082,47,22,63,268,00,24,11,102,00,26,13,037,42*7B
$PGRME,5.0,M,5.8,M,7.7,M*26
$PGRMM,WGS 84*06

$GPRMC,223323,A,3732.3792,N,12201.1625,W,0.0,194.8,220207,15.1,E*5B
$GPGGA,223323,3732.3792,N,12201.1625,W,1,08,0.9,11.2,M,-28.0,M,,*4F
$GPGSA,A,3,03,,07,09,,16,18,19,21,,,26,1.6,0.9,1.3*32
$GPGSV,3,1,12,03,39,301,34,06,08,157,00,07,18,163,33,09,09,096,46*7E
$PGRME,5.0,M,5.8,M,7.7,M*26
$PGRMM,WGS 84*06
```

```
$GPRMC,223324,A,3732.3793,N,12201.1625,W,0.0,194.8,220207,15.1,E*5D
$GPGGA,223324,3732.3793,N,12201.1625,W,1,08,0.9,11.3,M,-28.0,M,,*48
$GPGSA,A,3,03,,07,09,,16,18,19,21,,,26,1.6,0.9,1.3*32
$GPGSV,3,2,12,14,24,188,00,16,13,244,39,18,71,018,46,19,15,319,48*7B
$PGRME,5.0,M,5.8,M,7.7,M*26
$PGRMT,Bravo PDA Ver. 2.01,P,P,R,R,P,,21,*6C
$PGRMM,WGS 84*06
```

The Mosaic GPS Wildcard driver software does not send any sentences to the GPS. Thus the sentences named PGRMI, PGRMC, and PGRMO that configure various options and parameters in the GPS subsystem are not supported or used. The Mosaic driver accept the default performance of the GPS, which does its best to remember the proper time and location information to minimize acquisition time. Experts who are interested in the details of these configuration sentences can consult the Garmin GPS-15 datasheet.

Overview of the Mosaic GPS Software Device Driver Functions

On the GPS Wildcard, the NMEA sentence serial data is transmitted by the Garmin GPS subsystem and received by a UART (Universal Asynchronous Receiver Transmitter) chip. The serial data is transmitted at 4800 baud (bits per second). The UART contains a 64-byte receiver FIFO (First In/First Out) buffer that buffers the data until it is read by the processor via the Wildcard bus.

Data is continuously streaming out of the GPS subsystem, one multi-sentence frame per second. While some applications may need to continuously extract data to perform navigation functions, other applications may need only infrequent updates of GPS information. The Mosaic GPS driver is written to accommodate this broad range of application needs.

For applications with infrequent needs for GPS updates, the data streaming out of the GPS subsystem can be safely ignored until an update is needed. The unused data will overflow the FIFO buffer in the UART, and will effectively be discarded. At any point in time, the **`GPS_Update`** function can be called to flush any old characters from the UART FIFO, accept a valid frame of data sentences from the GPS subsystem, and extract the GPS data to a structure where it is accessible to the application program. Applications that need frequent updates from the GPS can run a tight loop in a task that continually calls **`GPS_Update`** and uses the resulting data as it becomes available.

`GPS_Update` is the highest level function in the GPS driver. It calls **`GPS_Next_Frame`** which waits for 7 sentences from the GPS subsystem and puts them into the **`GPS_Inbuf`**, a 600 byte character buffer allocated in common RAM. (To be precise, **`GPS_Inbuf`** is a function that returns the 32-bit base xaddress of the **`gps_default_inbuf`** character array.) **`GPS_Update`** then invokes **`GPS_Frame_Extract`**. This workhorse function parses the sentences in the **`GPS_Inbuf`**, extracts the relevant data parameters, and stores them into the **`gps_info`** structure for use by the application program. Your application program can access the **`gps_info`** struct fields in common RAM directly, or invoke a family of additional driver functions to fetch the fields out of the struct and return them.

The defining source code for the **`gps_info`** struct is presented in Listing 1-2 on page 19 (for the C language) and Listing 1-3 on page 33 (for the Forth language). Please take a moment now to examine the listing in your chosen programming language. The field names should be familiar, as they correspond to the data fields from the GPS sentences as described in the prior section. The comments accompanying each field provide additional descriptive information. While the NMEA sentences transmit the numeric data fields as ASCII values, the **gps_info** data structure stores the numeric data as binary integers (but the directions 'N' 'S' 'E' and 'W' are stored as ASCII values in the struct). This maximizes the programmer's flexibility in performing computations with the numeric data.

As an example, let's take the latitude of Mosaic Industries headquarters, which is approximately 37 degrees, 32.3793 minutes North of the equator. Recall that a minute is one sixtieth (1/60) of a degree. This latitude is reported in the GPRMC sentence as:

3732.3793,N

You can see this latitude report in Listing 1-1 on page 8 in the GPRMC sentence; the latitude is reported in the field after the 'A'. The first 2 digits are the latitude degrees (37), the next 2 digits are the integer part of the latitude minutes (32), then there is a decimal point, followed by the decimal fractional part of the latitude (.3793), followed by the 'N' field for North. After executing **GPS_Update** (or its component functions **GPS_Next_Frame** and **GPS_Frame_Extract**). the latitude is available in the following fields of the **gps_info** struct:

```
int sgps_lat_degrees;          // 00-89 degrees latitude
int sgps_lat_minutes_intpart;  // 00-59 integer latitude minutes
int sgps_lat_minutes_fraction; // .xxxx lat minutes fractional part (tenthousandths)
int sgps_lat_hemisphere;       // ascii 'N' or 'S'
```

Each field name begins with 's' to indicate that it is a struct field, and the field name is descriptive. In this example, the **sgps_lat_degrees** field contains a binary representation of the latitude degrees (37), and the **sgps_lat_minutes_intpart** field contains a binary representation of the integer part of the latitude minutes (32). The **sgps_lat_minutes_fraction** field contains a binary representation of the fractional part of the latitude minutes (3793); its dimensions (units) are ten-thousandths of a minute. The driver provides these fields to preserve all of the latitude information reported by the GPS subsystem in a useable form.

To directly fetch the latitude degrees into a variable named **latitude_degrees**, a C application program could use the statements:

```
int latitude_degrees;
latitude_degrees = gps_info.sgps_lat_degrees;
```

An application program can also use the function **GPS_Lat_Degrees** that fetches the parameter from the **gps_info** struct and returns it for use in the program. The following two C statements have the same effect as the statements above:

```
int latitude_degrees;
latitude_degrees = GPS_Lat_Degrees;
```

The function named **GPS_Lat_Minutes_Times_10000** scales up the latitude minutes by a factor of 10000 and returns it as a 32-bit integer in the range 0 to 599,999. This function returns the sum of contents of the **sgps_lat_minutes_fraction** field, plus 10000 times the contents of the **sgps_lat_minutes_intpart** field. The numeric value of the number returned by this routine is the current latitude minutes times 10000. The dimensions (units) of the returned values are ten-thousandths of a minute.

If desired, your application program can convert the latitude minutes into a single floating point number. For example,

```
float latitude_minutes;
latitude_minutes = GPS_Lat_Minutes_Times_10000 / 10000.0 ;
```

Some programmers may want to use fast integer math and avoid converting to the slower floating point math. (Forth programmers using the V4.xx kernels should also be aware of the 5 decimal digit resolution of the legacy floating point package; C programmers and V6.xx Forth programmers have a floating point package with about 7 decimal digits of resolution).

The **GPS_Lat_Hemisphere** function returns the ASCII 'N' or 'S' latitude character, or your program can fetch it out of the **gps_info** field using an assignment statement.

The longitude is reported in a parallel set of fields in the **gps_info** struct:

```
int sgps_long_degrees;          // 000-179 degrees longitude
int sgps_long_minutes_intpart;  // 00-59 integer longitude minutes
int sgps_long_minutes_fraction; // .xxxx long minutes fraction part (ten-thousandths)
int sgps_long_hemisphere;       // ascii 'E' or 'W'
```

The driver functions **GPS_Long_Degrees**, **GPS_Long_Minutes_Times_10000**, and **GPS_Long_Hemisphere** can be used to return the longitude information to the application program.

The GPS reports altitude as a number in the range -9999.9 to 99999.9 meters relative to mean sea level. The GPS driver software multiplies this by ten to convert it to a 32-bit long value in the range -99999 to 999999 in the **sgps_altitude_tenth_meters** field. The **GPS_Altitude_Meters_Times_10** function returns this value for use in the application program.

Ground speed is reported by the GPS subsystems in knots, or nautical miles per hour. A nautical mile is 1.152 times longer than a standard (statute) mile, and corresponds to 1 second of latitude arc. A knot is equivalent to 1.152 miles per hour. The GPS reports speed as a number in the range 0.0 to 999.9 knots, and the driver software multiplies this by ten to convert it to an integer in the range 0 to 9999 stored in the **sgps_tenth_knots** field. The **GPS_Knots_Times_10** function returns this value for use in the application program.

Course heading (direction) is reported by the GPS as the number of degrees in the range 0.0 to 359.9 relative to true north. The Mosaic GPS driver software multiplies this by ten to convert it to an integer in the range 0 to 3599 stored in the **sgps_course_tenth_degrees** field. The **GPS_Course_Degrees_Times_10** function returns this value for use in the application program.

The GPS reports the universal "UTC" time and date at the prime meridian that runs through Greenwich England. These time and date fields are saved as binary quantities in the **gps_info** struct where they can be accessed directly by the application program. There are no pre-defined driver functions that fetch the time and date quantities. The universal time/date differs from the local time/date by an integer number of hours. For convenience, the **GPS_Init** function accepts a "**local_hour_offset**" parameter which is the number of hours that must be added to the universal "UTC" (Greenwich) time to obtain the correct local time. In general, the **`local_hour_offset`** for a location with "West" longitude such as the United States has a negative **`local_hour_offset`**, while a location with "East" longitude has a positive **`local_hour_offset`**. The pre-defined constants **`GPS_PACIFIC_TIME`**, **`GPS_MOUNTAIN_TIME`**, **`GPS_CENTRAL_TIME`**, and **`GPS_EASTERN_TIME`** can be passed to the **`GPS_Init`** routine to specify the named time zones. To specify daylight savings time, add 1 to the specified time zone constant before passing it to this function (see the glossary entry for **`GPS_DAYLIGHT_TIME`**). The **`GPS_Update`** function (via its callee, **`GPS_Frame_Extract`**) extracts the universal UTC time and date from the GPRMC sentence, and uses the **`local_hour_offset`** to calculate the local time and date. These are stored in the following **`gps_info`** fields:

```
int sgps_utc_hour;        // 00-23 hour
int sgps_utc_date;        // 01-31 date of the month
int sgps_utc_month;       // 01-12 month of the year
int sgps_utc_year;        // 00-99 2-digit year
int sgps_local_second;    // 00-59 seconds after minute; utc_second = local_second
int sgps_local_minute;    // 00-59 minutes after hour; utc_minute = local_minute
int sgps_local_hour;      // 00-23 hour = (utc_hour + local_time_hour_offset) mod 24
int sgps_local_date;      // 01-31 date of the month, result of utclocal conversion
int sgps_local_month;     // 01-12 month of the year, result of utclocal conversion
int sgps_local_year;      // 00-99 2-digit year, result of utclocal conversion
```

The **`GPS_Good_Fix`** function returns a true (-1) value if the contents of the struct field **`sgps_fix_quality`** = 1 and if the contents of the struct field **`sgps_position_valid`** is true. In other words, if **`GPS_Good_Fix`** returns a nonzero value, the GPS information can be relied upon (according to the GPS receiver itself). The **`GPS_Numsats_In_Use`** and **`GPS_Numsats_In_View`** functions also report information that is useful for determining the status of the GPS receiver.

The **`GPS_Init`** function initializes the data structures and the GPS Wildcard hardware. To force a reset and satellite reacquisition, pass a true flag to the **`GPS_Shutdown`** function to shut down the GPS, and then pass a false flag to the **`GPS_Shutdown`** function to restart the GPS.

If the GPS will not be used for significant periods of time, you can save power by shutting it down using **`GPS_Shutdown`**. Of course, when the GPS is turned back on there will be a delay while it reacquires the satellite fix.

GPS Service Loop Is Best Placed In Its Own Task

A typical GPS application program initializes the GPS by calling **`GPS_Init`**, and then repeatedly invokes the high level **`GPS_Update`** function to accept a frame of data from the GPS subsystem and populate the fields in the **`gps_info`** data structure. **`GPS_Update`** calls **`GPS_Next_Frame`** which can take over 1 second to obtain a frame of data from the GPS subsystem. If the **`GPS_Update`** or **`GPS_Next_Frame`** function is invoked in a single-task system, all other instrument tasks could be halted for over a second.

To avoid this problem, create a separate task and call **`GPS_Update`** from within that task's infinite loop. Invoke **`StartTimeslicer`** (`START.TIMESLICER` in Forth) to run the multitasker. In this way the GPS will be serviced without blocking time-critical code running in other tasks.

As described above, the GPS driver is designed to allow GPS data to be ignored and discarded when it is not needed by the application. Once the application requires data, **`GPS_Update`** (or its callee function `GPS_Next_Frame`) must be able to obtain an unadulterated frame of data from the GPS and store it in the 600 byte `GPS_Inbuf` buffer. At the serial data rate of 4800 baud, and given that there are 10 bits per character (1 start bit, 8 data bits, and 1 stop bit), each character is transmitted in approximately 2 milliseconds. Thus the 64 byte UART FIFO will not overflow if the processor services the GPS Wildcard at least 8 times per second. This constraint is easily met in a multitasking system with the timeslicer running.

Demo Program Prints Formatted GPS Data

The demonstration program presents a simple example of how to use the GPS Wildcard.

The `GPS_Demo` function (or the `main` function in C) starts and runs the demonstration using the GPS Wildcard. The demonstration program source code is presented at the end of this document in both C and Forth.

The demo initializes the GPS by passing the `local_hour_offset` and `module_num` (the Wildcard address) to the `GPS_Init` function. If there is no initialization error, the following descriptive text header is printed to the terminal:

```
Starting GPS info dump.  Type any key to exit this function.
This function gets a data frame from the GPS once per second,
and prints a statement telling of the fix validity.
If the fix is not valid, make sure your antenna is outside,
and give the GPS some time (5 minutes worst case) to acquire its satellites.
This function periodically calls GPS_Info_Dump to summarize the GPS data.
```

Then the demo enters a loop calling `GPS_Update`. If `GPS_Update` returns a true error flag, the demo prints a statement telling us that no GPS fix is available. If the error flag is zero, the demo prints a statement that a fix is available. Every 5 seconds, the demo program calls `GPS_Info_Dump` to print a formatted summary of the GPS parameters.

This simple demo program runs from the default task and uses the default serial port to print data to the Mosaic terminal program running on your PC. In a real application, the `GPS_Update` function would be called from within a task dedicated to servicing the GPS.

Installing the Mosaic GPS Wildcard Driver Software

The GPS Wildcard device driver software is provided as a pre-coded modular runtime library, known as a “kernel extension” because it enhances the on-board kernel's capabilities. The library functions are accessible from C and Forth.

Mosaic Industries can provide you with a web site link that will enable you to create a packaged kernel extension that has drivers for all of the hardware that you have on your system. In this way the software drivers are customized to your needs, and you can generate whatever combination of drivers you need. Make sure to specify the GPS Wildcard Drivers in the list of kernel extensions you want to generate, and download the resulting “packages.zip” file to your hard drive.

For convenience, a separate pre-generated kernel extension for the GPS Wildcard is available from Mosaic Industries on the Demo and Drivers media (diskette or CD). Look in the Drivers directory, in the subdirectory corresponding to your hardware (the Mosaic Controller of your choice) in the GPS_Wildcard folder.

The kernel extension is shipped as a “zipped” file named “packages.zip”. Unzipping it (using, for example, winzip or pkzip) extracts the following files:

- readme.txt - Provides summary documentation about the library.
- install.txt - The installation file, to be loaded to COLD-started Mosaic Controller.
- library.4th - Forth name headers and utilities; prepend to Forth programs.
- library.c - C callers for all functions in library; #include in C code.
- library.h - C prototypes for all functions; #include in extra C files.

Library.c and library.h are only needed if you are programming in C. Library.4th is only needed if you are programming in Forth. The uses of all of these files are explained below.

We recommend that you move the relevant files to the same directory that contains your application source code.

To use the kernel extension, the runtime kernel extension code contained in <u>the install.txt file must first be loaded into the flash memory of the Mosaic Controller</u>. Start the Terminal software with the Mosaic Controller connected to the serial port and turned on. If you have not yet tested your Mosaic Controller and terminal software, please refer to the documentation provided with the Terminal software. Once you can hit enter and see the 'ok' prompt returned in the terminal window, type

```
COLD
```

to ensure that the board is ready to accept the kernel extension install file. Use the “Send File” menu item of the terminal to download the install.txt to the Mosaic Controller.

Now, type

```
COLD
```

again and the kernel has been extended! <u>Once install.txt has been loaded, it need not be reloaded each time that you revise your source code.</u>

Using the Mosaic GPS Driver Code with C

Move the library.c and library.h files into the same directory as your other C source code files. After loading the install.txt file as described above, use the following directive in your source code file:

```
#include “library.c”
```

This file contains calling primitives that implement the functions in the kernel extension package. The library.c file automatically includes the library.h header file. If you have a project with multiple source code files, you should only include library.c once, but use the directive

```
#include “library.h”
```

in every additional source file that references the GPS functions.

To load the optional demonstration program described above, use the "make" icon of the C compiler to compile the file named

`GPSdemo.c`

that is provided on the distribution media. Use the terminal to send the resulting `GPSdemo.txt` file to the Mosaic Controller, and type `main` to run the program. See the demo source code listing below for more details.

Note that all of the functions in the kernel extension are of the `_forth` type. While they are fully callable from C, there are important restrictions. First, the `_forth` functions may not be called as part of a parameter list of another `_forth` function. Second, `_forth` functions may not be called from within an interrupt service routine unless the instructions found in the file named

`\fabius\qedcode\forthirq.c`

are followed (this second restriction is lifted for V6.xx kernels as shipped with the PDQ line). Also, in most cases Key and Emit functions should not be called from within interrupt service routines, because these routines call `PAUSE`, and use of `PAUSE` within an interrupt routine can halt the multitasker.

Using the Mosaic GPS Driver Code with Forth

After loading the install.txt file and typing `COLD`, use the terminal to send the "library.4th" file to the Mosaic Controller. Library.4th sets up a reasonable memory map and then defines the constants, structures, and name headers used by the GPS Wildcard kernel extension. Library.4th leaves the memory map in the download map.

After library.4th has been loaded, the board is ready to receive your high level source code files. Be sure that your software doesn't initialize the memory management variables DP, VP, or NP, as this could cause memory conflicts. If you wish to change the memory map, edit the memory map commands at the top of the library.4th file itself. The definitions in library.4th share memory with your Forth code, and are therefore vulnerable to corruption due to a crash while testing. If you have problems after reloading your code, try typing `COLD`, and reload everything starting with library.4th. It is very unlikely that the kernel extension runtime code itself (install.txt) can become corrupted since it is stored in flash on a page that is not typically accessed by code downloads.

We recommend that your source code file begin with the sequence:

```
WHICH.MAP 0=
IFTRUE 4 PAGE.TO.RAM  \ if in standard.map...
       5 PAGE.TO.RAM
       6 PAGE.TO.RAM
      DOWNLOAD.MAP
ENDIFTRUE
```

This moves all pre-loaded flash contents to RAM if the Mosaic Controller is in the standard (flash-based) memory map, and then establishes the download (RAM-based) memory map. At the end of this sequence the Mosaic Controller is in the download map, ready to receive additional code.

We recommend that your source code file end with the sequence:

```
4 PAGE.TO.FLASH
5 PAGE.TO.FLASH
```

```
6 PAGE.TO.FLASH
STANDARD.MAP
SAVE
```

This copies all loaded code from RAM to flash, and sets up the standard (flash-based) memory map with code located in pages 4, 5 and 6. The `SAVE` command means that you can often recover from a crash and continue working by typing `RESTORE` as long as flash pages 4, 5 and 6 haven't been rewritten with any bad data.

Glossary of GPS Driver Functions

This glossary defines important constants and functions from the GPS driver code and demo program.

Overview of Glossary Notation

The main glossary entries presented in this document are listed in case-insensitive alphabetical order (the underscore character comes at the end of the alphabet). The keyword name of each entry is in **bold** typeface. Each function is listed with both a C-style declaration and a Forth-style stack comment declaration as described below. The "C:" and "4th:" tags at the start of the glossary entry distinguish the two declaration styles.

The Forth language is case-insensitive, so Forth programmers are free to use capital or lower case letters when typing keyword names in their program. Because C is case sensitive, C programmers must type the keywords exactly as shown in the glossary. The case conventions are as follows:

- Function names begin with a capital letter, and every letter after an underscore is capitalized. Other letters are lower case, except for capitalized acronyms such as "GPS".
- Constant names and C macros use capital letters.
- Variable names use lower case letters.

Each glossary entry starts with C-style and Forth-style declarations, and presents a description of the function. Here is a sample glossary entry:

C: int **GPS_Update** (int module_num)
4th: **GPS_Update** (module -- error)
This high level function invokes GPS_Next_Frame to get a frame of GPS ASCII data into the gps_inbuf, then calls GPS_Frame_Extract to parse the GPS sentences in the order that they appear, extracting data from these sentences and storing the data into the gps_info struct. The module_num passed to this function is the address of the Wildcard and it must match the hardware jumper settings of J1 and J2 on the GPS Wildcard. GPS_Update can take over 1 second to obtain a frame of data from the GPS subsystem, so it is best placed in its own task where it will not block time-critical code. Make sure to call GPS_Init before invoking GPS_Update. For an example of use, see the GPS_Run and GPS_Demo functions which are provided in source code form in the demonstration program listing at the end of this document.

The C declaration specifies that return data type before the function name, and lists the comma-delimited input parameters between parentheses, showing the type and a descriptive name for each.

The Forth declaration contains a "stack picture" between parentheses; this is recognized as a comment in a Forth program. The items to the left of the double-dash (--) are input parameters; multiple parameters are separated by a \ character which is read as "under". The item to the right of the double-dash is the output parameter. Forth is stack-based, and the first item shown is lowest on the stack. In the Forth declaration the parameter names and their data types are combined. All unspecified parameters are 16-bit integers. Forth promotes all characters to integer type.

The presence of both C and Forth declarations is helpful: the C syntax shows the types of the parameters, and the Forth declaration provides a descriptive name of the output parameter.

Glossary Quick Reference

This overview categorizes the driver function into four groups:

- Initialization functions initialize the GPS and describe the driver's data structures; the high level GPS_Init function must be called after each powerup or restart.
- Data Extraction functions convert ASCII data coming from the Garmin GPS subsystem into useable parameters that describe position, time, speed, and heading. GPS_Update is the highest level data extraction function; it calls GPS_Next_Frame to get ASCII data from the GPS subsystem, and then calls GPS_Frame_Extract to parse the data and store it into the relevant fields in the gps_info structure.
- Most of the Data Reporting functions fetch the elements from the gps_info structure and return them. They are made available as a convenience, ant their functionality can be mimicked by directly reading the corresponding elements out of the gps_info struct. The GPS_Dump and GPS_Info_Dump routines are serial output functions that are useful during debugging.
- The Demonstration functions are presented in source code form in the listing at the end of this document.

Data Extraction

int **GPS_Frame_Extract** (xaddr xbuf, int numchars)
int **GPS_Get_Altitude_Fix_Numsats** (xaddr xbuf, int maxchars)
C: int **GPS_Get_Frame** (xaddr bufbase,int starting_os,int max_os,int numsentences,int module)
int **GPS_Get_Pos_Errors** (xaddr xbuf, int maxchars)
int **GPS_Get_Pos_Speed_Course_Time** (xaddr xbuf, int maxchars)
int **GPS_Get_Sats_In_View** (xaddr xbuf, int maxchars)
int **GPS_Next_Frame** (int module_num)
int **GPS_Parse_Altitude_Fix_Numsats** (xaddr xbuf, int dollar_offset, int maxchars)
int **GPS_Parse_Pos_Errors** (xaddr xbuf, int dollar_offset, int maxchars)
int **GPS_Parse_Pos_Speed_Course_Time** (xaddr xbuf, int dollar_offset, int maxchars)
int **GPS_Parse_Sats_In_View** (xaddr xbuf, int dollar_offset, int maxchars)
int **GPS_Update** (int module_num)

Data Reporting

long **GPS_Altitude_Meters_Times_10** (void)
int **GPS_Course_Degrees_Times_10** (void)
void **GPS_Dump** (int module_num)
int **GPS_Good_Fix** (void)
void **GPS_Info_Dump** (void)
int **GPS_Knots_Times_10** (void)
int **GPS_Lat_Degrees** (void)
char **GPS_Lat_Hemisphere** (void)
long **GPS_Lat_Minutes_Times_10000** (void)
int **GPS_Long_Degrees** (void)
char **GPS_Long_Hemisphere** (void)
long **GPS_Long_Minutes_Times_10000** (void)
int **GPS_Numsats_In_Use** (void)
int **GPS_Numsats_In_View** (void)

Initialization

GPS_BUFSIZE
GPS_CENTRAL_TIME
GPS_DAYLIGHT_TIME
gps_default_inbuf [GPS_BUFSIZE]
void **GPS_Default_UART_Init** (int module_num)
GPS_EASTERN_TIME
xaddr **GPS_Inbuf** (void)
gps_info
int **GPS_Init** (int local_hour_offset, int module_num)
GPS_MOUNTAIN_TIME
GPS_PACIFIC_TIME
void **GPS_Shutdown** (int shutdown_flag, int module_num)
GPS_STRUCT
void **GPS_Struct_Init** (xaddr gps_info_xbase, xaddr xinbuf, int local_hour_offset)

Demonstration Program

void **GPS_Demo** (void)
int **GPS_Run** (int local_hour_offset, int module_num)
GPS_MODULE_NUM

GPS_STRUCT Definition

The position, time, date, speed, heading, and error estimation parameters extracted from the Garmin GPS subsystem are stored in the gps_info struct in common RAM. The elements of this struct can be accessed from your application program to obtain the latest GPS data after each invocation of GPS_Update. Listing 1-2 on page 19 presents the commented C source code definition of the structure, and Listing 1-3 on page 33 presents the commented Forth source code definition of the structure. In C, this is a typedef, and in Forth, it returns the struct size of the gps_info struct. The

elements of the gps_info struct are described at the start of this glossary; they hold data obtained from the Garmin GPS subsystem. The struct elements are updated by the functions GPS_Run, GPS_Update, GPS_Frame_Extract, and by functions having names that start with GPS_Parse.
C structure use example: To read the contents of the structure element named sgps_lat_degrees in C, use the standard C construct:

gps_info.sgps_lat_degrees

as the right-hand side of an assignment statement.
Forth structure use example: To read the contents of the structure element named sgps_lat_degrees in Forth, use the standard Forth construct:

gps_info sgps_lat_degrees @

to fetch the contents to the data stack.

Listing 1-2 C GPS_STRUCT commented source code definition.

```
    typedef struct gps_struct     // holds information for the gps wildcard
    {
    xaddr sgps_inbuf;    // holds base xaddr of buffer for sentences coming from gps
    xaddr sgps_outbuf;   // holds base xaddr of buffer for outgoing sentences: unused!
    int sgps_local_hour_offset; // eg, pst (pac std time) = -8, pdt = -7
     // the following are from the $gprmc data from gps sensor:
    int sgps_utc_hour;       // 00-23 hour
    int sgps_utc_date;       // 01-31 date of the month
    int sgps_utc_month;      // 01-12 month of the year
    int sgps_utc_year;       // 00-99 2-digit year
    int sgps_local_second;   // 00-59 seconds after minute; utc_second = local_second
    int sgps_local_minute;   // 00-59 minutes after hour; utc_minute = local_minute
    int sgps_local_hour;     // 00-23 hour = (utc_hour + local_time_hour_offset) mod 24
    int sgps_local_date;     // 01-31 date of the month, result of utclocal conversion
    int sgps_local_month;    // 01-12 month of the year, result of utclocal conversion
    int sgps_local_year;     // 00-99 2-digit year, result of utclocal conversion
    int sgps_position_valid; // holds boolean flag; -1=valid pos; 0=rcvr warning
    int sgps_lat_degrees;    // 00-89 degrees latitude
    int sgps_lat_minutes_intpart;  // 00-59 integer latitude minutes
    int sgps_lat_minutes_fraction; // .xxxx lat minutes fractional part (tenthousandths)
    int sgps_lat_hemisphere;       // ascii 'N' or 'S'
    int sgps_long_degrees;         // 000-179 degrees longitude
    int sgps_long_minutes_intpart; // 00-59 integer longitude minutes
    int sgps_long_minutes_fraction; // .xxxx long minutes fraction part (ten-thousandths)
    int sgps_long_hemisphere;      // ascii 'E' or 'W'
    int sgps_tenth_knots;    // = 0-9999 speed over ground; field = 000.0 to 999.9 knots
    int sgps_course_tenth_degrees; // = 0-3599 course; field= 000.0 to 359.9 degrees true
    int sgps_mag_var_tenth_degrees; // 0-1800 magnetic variation,field= 000.0 to 180.0 deg
    int sgps_mag_var_direction;     // 'E' or 'W' mag var dir;west var adds to true course
      // the following is from the $gpgsv data from gps sensor:
    int  sgps_numsats_in_view;      // 00-12
       // the following are from the $gpgga data from gps sensor:
    int sgps_fix_quality; // contains binary: 0= fix not available; 1 = non-diff fix avail
    int sgps_numsats_in_use;    // 00-12 = number of satellites in use
    double sgps_altitude_tenth_meters;  // field = -9999.9 to 99999.9 meters rel to SeaLev
    // the following are from the $pgrme data from gps sensor:
    int sgps_hor_error_tenth_meters; // 0-9999 est horiz pos err;field=0.0 to 999.9 meters
    int sgps_vert_error_tenth_meters; // 0-9999 est vert pos err;field=0.0 to 999.9 meters
    int sgps_position_error_tenth_meters; // 0-9999 overall pos err;fld=0 to 999.9 meters
} GPS_STRUCT
```

Glossary Entries

C: long **GPS_Altitude_Meters_Times_10** (void)
4th: **GPS_Altitude_Meters_Times_10** (-- d)
The Garmin GPS subsystem reports altitude as the height of the GPS antenna relative to mean sea level in meters, with a resolution of 0.1 meter. The altitude can range from -9999.9 to +99999.9 meters, where positive values are above sea level. To maximize flexibility in the selection of fast integer math or slower floating point math when performing longitude computations, the current altitude value returned by this function is scaled up by a factor of 10 and returned as a 32-bit signed integer in the range -99999 to 999,999. The numeric value of the number returned by this routine is the current altitude meters times 10. The dimensions (units) of the returned values are tenths of a meter.
Implementation detail: This function returns the contents of the sgps_altitude_tenth_meters field. The struct fields are updated by GPS_Update; see its glossary entry.

C: **GPS_BUFSIZE**
4th: **GPS_BUFSIZE** (-- n)
A constant whose value equals decimal 600, used to dimension the gps_default_inbuf as initialized by GPS_Init. This size is enough to hold 7 maximum size GPS sentences from the Garmin GPS subsystem as required by GPS_Next_Frame. See GPS_Inbuf and gps_default_inbuf.

C: **GPS_CENTRAL_TIME**
4th: **GPS_CENTRAL_TIME** (-- n)
A constant whose value equals -6, used as the local_hour_offset passed to the GPS_Struct_Init, GPS_Init and GPS_Run functions to specify central standard time. The local_hour_offset is the number of hours that must be added to the universal "UTC" (Greenwich) time to obtain the correct local time. In general, the local_hour_offset for a location with "West" longitude such as the United States has a negative local_hour_offset, while a location with "East" longitude has a positive local_hour_offset. To specify daylight savings time, add 1 to this time zone constant before passing it to this function. See also GPS_PACIFIC_TIME, GPS_MOUNTAIN_TIME, GPS_EASTERN_TIME, and GPS_DAYLIGHT_TIME.

C: int **GPS_Course_Degrees_Times_10** (void)
4th: **GPS_Course_Degrees_Times_10** (-- n)
The Garmin GPS subsystem reports course over the ground with respect to true north in degrees, with 0.1 degree resolution. Values range from 0.0 to 359.9 degrees with respect to true (not magnetic) north. To maximize flexibility in the selection of fast integer math or slower floating point math when performing speed computations, the current course value returned by this function is scaled up by a factor of 10 and returned as a 16-bit integer in the range 0 to 3599. The numeric value of the number returned by this routine is the current course degrees times 10. The dimensions (units) of the returned values are tenths of a degree.
Implementation detail: This function returns the contents of the sgps_course_tenth_degrees struct field.

The struct fields are updated by GPS_Update; see its glossary entry.

C: **GPS_DAYLIGHT_TIME**
4th: **GPS_DAYLIGHT_TIME** (-- n)
A constant whose value equals +1, which is added to the local_hour_offset passed to the GPS_Struct_Init, GPS_Init and GPS_Run functions to specify daylight savings time for the specified time zone. The local_hour_offset is the number of hours that must be added to the universal "UTC" (Greenwich) time to obtain the correct local time. In general, the local_hour_offset for a location with "West" longitude such as the United States has a negative local_hour_offset, while a location with "East" longitude has a positive local_hour_offset. To specify daylight savings time, add GPS_DAYLIGHT_TIME to the time zone constant before passing it to this function. For example, to specify pacific daylight time, pass to GPS_Init the sum of GPS_PACIFIC_TIME and GPS_DAYLIGHT_TIME. See also GPS_PACIFIC_TIME, GPS_MOUNTAIN_TIME, GPS_CENTRAL_TIME, and GPS_EASTERN_TIME.

C: **gps_default_inbuf [GPS_BUFSIZE]**
4th: **gps_default_inbuf** (-- xaddr)
A character array buffer whose size equals GPS_BUFSIZE (decimal 600). This size is enough to hold 7 maximum size GPS sentences from the Garmin GPS subsystem as required by GPS_Next_Frame. The base xaddress of this buffer is passed to GPS_Struct_Init by GPS_Init, and is stored into the sgps_inbuf field in the gps_info struct. After GPS_Init has run, the GPS_Inbuf function returns the 32-bit base address of this buffer. See GPS_Inbuf.

C: void **GPS_Default_UART_Init** (int module_num)
4th: **GPS_Default_UART_Init** (module_num --)
This low-level function initializes the UART chip on the GPS Wildcard that communicates with the Garmin GPS subsystem. This function is automatically called by the standard high level initialization function GPS_Init.
Implementation detail: Configures the UART for 8 bits per character, 1 stop bit, no parity, 4800 baud, with 64byte input and output first in/first out (FIFO) buffers. Sets the /3.3v_shutdown (/out2) hardware pin inactive high, thus powering up the Garmin GPS subsystem.

C: void **GPS_Demo** (void)
4th: **GPS_Demo** (--)
A high level function in the demonstration program; see its source code listing at the end of this document. This function passes a default local_hour_offset (GPS_PACIFIC_TIME in the demo code listing) and the GPS_MODULE_NUM constant to the GPS_Run function. <u>The value of the GPS_MODULE_NUM constant must match the hardware jumper settings of J1 and J2 on the GPS Wildcard</u>. The local_hour_offset is the number of hours that must be added to the universal "UTC" (Greenwich) time to obtain the correct local time. In general, the local_hour_offset for a location with "West" longitude such as the United States has a negative local_hour_offset, while a location with "East" longitude has a positive local_hour_offset. The pre-defined constants GPS_PACIFIC_TIME, GPS_MOUNTAIN_TIME, GPS_CENTRAL_TIME, and GPS_EASTERN_TIME can be used in this routine to specify the named time zones. To specify daylight savings time, add 1 to the specified time zone constant. After initializing the GPS, the function prints a descriptive text introduction, and enters a loop that repeatedly calls GPS_Update to

get a frame of data from the GPS subsystem, and periodically calls GPS_Info_Dump to print to the serial port a formatted version of the extracted data from the gps_info struct. Depending on the prior state of the GPS, a valid fix can be acquired in a few seconds, or it can take as long as 5 minutes if the GPS subsystem does not have enough stored time and location information to speed its acquisition process. Typing any key terminates the function. This demonstration function is meant to be invoked interactively using the Mosaic terminal. See also GPS_Run.

C: void **GPS_Dump** (int module_num)
4th: **GPS_Dump** (module_num --)
Dumps the raw ASCII data stream from the Garmin GPS subsystem to the serial port. The module_num passed to this function must match the hardware jumper settings of J1 and J2 on the GPS Wildcard. This routine is provided for debugging purposes, as it allows you to examine the raw GPS sentences using a terminal. This routine invokes PauseOnKey, so you can abort the listing by typing a carriage return at the terminal. This function requires that the GPS Wildcard has been initialized using GPS_Init. Once the GPS has been initialized, it outputs 6 or 7 sentence types each second. The $PGRMT sentence is output only once per minute. An example 1-second frame of data resulting from execution of the GPS_Dump function is as follows:

```
$GPRMC,223324,A,3732.3793,N,12201.1625,W,0.0,194.8,220207,15.1,E*5D
$GPGGA,223324,3732.3793,N,12201.1625,W,1,08,0.9,11.3,M,-28.0,M,,*48
$GPGSA,A,3,03,,07,09,,16,18,19,21,,,26,1.6,0.9,1.3*32
$GPGSV,3,2,12,14,24,188,00,16,13,244,39,18,71,018,46,19,15,319,48*7B
$PGRME,5.0,M,5.8,M,7.7,M*26
$PGRMT,Bravo PDA Ver. 2.01,P,P,R,R,P,,21,*6C
$PGRMM,WGS 84*06
```

C: **GPS_EASTERN_TIME**
4th: **GPS_EASTERN_TIME** (-- n)
A constant whose value equals -5, used as the local_hour_offset passed to the GPS_Struct_Init, GPS_Init and GPS_Run functions to specify eastern standard time. The local_hour_offset is the number of hours that must be added to the universal “UTC” (Greenwich) time to obtain the correct local time. In general, the local_hour_offset for a location with “West” longitude such as the United States has a negative local_hour_offset, while a location with “East” longitude has a positive local_hour_offset. To specify daylight savings time, add 1 to this time zone constant before passing it to this function. See also GPS_PACIFIC_TIME, GPS_MOUNTAIN_TIME, GPS_CENTRAL_TIME, and GPS_DAYLIGHT_TIME.

C: int **GPS_Frame_Extract** (xaddr xbuf, int numchars)
4th: **GPS_Frame_Extract** (xbuf\numchars -- error)
Examines up to numchars characters in the specified buffer xbuf containing ASCII sentences from the GPS subsystem, and parses the GPS sentences in the order that they appear, extracting data from these sentences and storing the data into the gps_info struct. This function expects that GPS_Next_Frame (which returns the numchars parameter passed to this function) has been called to receive 7 GPS sentences; this ensures that a full “frame” of sentence data from the Garmin GPS subsystem is stored in the xbuf buffer. The high level GPS_Update function calls both GPS_Next_Frame and GPS_Frame_Extract; see their glossary entries. GPS_Frame_Extract searches the specified buffer for the next ASCII ‘$’ sentence start character, and then checks the

sentence identifier string to see if it is one of the GPRMC, GPGGA, GPGSV, or PGRME sentences that we extract data from. If so, it calls one of the appropriate lower level functions GPS_Parse_Pos_Speed_Course_Time, GPS_Parse_Altitude_Fix_Numsats, GPS_Parse_Sats_In_View, or GPS_Parse_Pos_Errors (see their glossary entries) to extract and store the data into the gps_info struct. If all of the relevant sentences have a valid checksum and all required fields are present, a false error flag is returned; otherwise a true flag is returned.

C: int **GPS_Get_Altitude_Fix_Numsats** (xaddr xbuf, int maxchars)
4th: **GPS_Get_Altitude_Fix_Numsats** (xbuf\maxchars -- error)
This function expects that GPS_Get_Frame or GPS_Next_Frame has run so that a valid frame of data is in the buffer starting at xbuf; the maxchars parameter returned by these functions is passed to GPS_Get_Altitude_Fix_Numsats to specify the number of characters present in the buffer. This function examines the contents of the buffer beginning at xbuf looking for the sentence identifying sequence $GPGGA. Once the $GPGGA sentence has been found in the buffer, GPS_Parse_Altitude_Fix_Numsats is called to extract the relevant fields of the sentence. It extracts the GPS fix quality (0= fix not available, 1=non-differential gps fix available), number of GPS satellites in use, and altitude (antenna height in meters with respect to sea level). If parsing of the sentence fails due to a bad checksum or other error, no data is stored and a -1 error flag is returned. If parsing of the sentence is successful, this routine returns 0 and updates the following fields in the gps_info struct: sgps_fix_quality, sgps_numsats_in_use, and sgps_altitude_tenth_meters

C: int **GPS_Get_Frame** (xaddr bufbase,int starting_os,int max_os,int numsentences,int module)
4th: **GPS_Get_Frame** (xbufbase\starting_os\max_os\numsentences\module -- numchars)
A low level function called by GPS_Next_Frame (see its glossary entry). This function receives numsentences GPS sentences which start with ASCII '$' and end with a carriage return/linefeed. The ASCII sentences are emplaced starting at the specified starting_os offset from the bufbase buffer base xaddress. <u>The module_num passed to this function must match the hardware jumper settings of J1 and J2 on the GPS Wildcard</u>. The number of characters stored is limited to the difference between the max_os and starting_os offsets. This function returns the number of characters emplaced in the buffer. GPS_Get_Frame can take over 1 second to obtain 7 sentences, so it is best placed in its own task where it will not block time-critical code.
Implementation detail: Flushes all existing characters out of the UART input FIFO (first in/first out buffer) on the GPS Wildcard to get rid of any partial sentences that might be in the FIFO, and then puts characters into the specified xbufbase buffer as described above. Each sentence is recognized by its starting '$' and its ending linefeed character, but the terminating ASCII linefeed (0x0A) is not stored in the buffer. Thus each sentence in the buffer ends with the ASCII carriage return (0x0D) character. After each sentence (except the last one received) this routine calls PAUSE while waiting for the starting ASCII '$' of the next sentence. This ensures that the function is calling PAUSE to share processor time while waiting for the next sentence.

C: int **GPS_Get_Pos_Errors** (xaddr xbuf, int maxchars)
4th: **GPS_Get_Pos_Errors** (xbuf\maxchars -- error)
This function expects that GPS_Get_Frame or GPS_Next_Frame has run so that a valid frame of data is in the buffer starting at xbuf; the maxchars parameter returned by these functions is passed to GPS_Get_Pos_Errors to specify the number of characters present in the buffer. This function examines the contents of the buffer beginning at xbuf looking for the sentence identifying sequence

$PGRM. Once the $PGRM sentence has been found in the buffer, GPS_Parse_Pos_Errors is called to extract the relevant fields of the sentence. It extracts the estimated horizontal, vertical, and overall position errors reported by the Garmin GPS subsystem. Reported errors are in the range 0 to 999.9 meters. If parsing of the sentence fails due to a bad checksum or other error, no data is stored and a -1 error flag is returned. If parsing of the sentence is successful, this routine returns 0 and updates the following fields in the gps_info struct: sgps_hor_error_tenth_meters, sgps_vert_error_tenth_meters, and sgps_position_error_tenth_meters

C: int **GPS_Get_Pos_Speed_Course_Time** (xaddr xbuf, int maxchars)
4th: **GPS_Get_Pos_Speed_Course_Time** (xbuf\maxchars -- error)
This function expects that GPS_Get_Frame or GPS_Next_Frame has run so that a valid frame of data is in the buffer starting at xbuf; the maxchars parameter returned by these functions is passed to GPS_Get_Pos_Speed_Course_Time to specify the number of characters present in the buffer. This function examines the contents of the buffer beginning at xbuf looking for the sentence identifying sequence $GPRMC. Once the $GPRMC sentence has been found in the buffer, GPS_Parse_Pos_Speed_Course_Time is called to extract the relevant fields of the sentence. It extracts the universal (UTC) time and date, position validity flag, latitude and longitude (degrees, minutes, hemisphere), speed in knots, course over ground in degrees true, and magnetic variation. Based on the local_hour_offset parameter passed to GPS_Run, GPS_Init or GPS_Struct_Init, this routine calculates the local time and date. If parsing of the sentence fails due to a bad checksum or other error, no data is stored and a -1 error flag is returned. If parsing of the sentence is successful, this routine returns 0 and updates the following fields in the gps_info struct: sgps_utc_hour, sgps_utc_date, sgps_utc_month, sgps_utc_year, sgps_local_second, sgps_local_minute, sgps_local_hour, sgps_local_date, sgps_local_month, sgps_local_year, sgps_position_valid, sgps_lat_degrees, sgps_lat_minutes_intpart, sgps_lat_minutes_fraction, sgps_lat_hemisphere, sgps_long_degrees, sgps_long_minutes_intpart, sgps_long_minutes_fraction, sgps_long_hemisphere, sgps_tenth_knots, sgps_course_tenth_degrees, sgps_mag_var_tenth_degrees, and sgps_mag_var_direction.

C: int **GPS_Get_Sats_In_View** (xaddr xbuf, int maxchars)
4th: **GPS_Get_Sats_In_View** (xbuf\maxchars -- error)
This function expects that GPS_Get_Frame or GPS_Next_Frame has run so that a valid frame of data is in the buffer starting at xbuf; the maxchars parameter returned by these functions is passed to GPS_Get_Sats_In_View to specify the number of characters present in the buffer. This function examines the contents of the specified buffer beginning at xbuf looking for the sentence identifying sequence $GPGSV. This sentence contains a 2-byte field in the range 00 to 12 representing the number of GPS satellites that are currently visible (but not necessarily in use) by the Garmin GPS subsystem. Once the $GPGSV sentence has been found in the buffer, GPS_Parse_Sats_In_View is called to extract the third numeric field of the sentence. If a valid 2-byte field is found in the sentence, this function returns 0 and stores the binary representation of the field in the sgps_numsats_in_view element of the gps_info struct. If valid data is not found, this routine stores nothing and returns a -1 error flag. This function is automatically called by the higher level functions GPS_Frame_Extract and GPS_Update; see their glossary entries.

C: int **GPS_Good_Fix** (void)
4th: **GPS_Good_Fix** (-- n)

This routine returns a nonzero value if the current GPS data can be relied upon. Returns true (-1) if the contents of the struct field sgps_fix_quality = 1 and if the contents of the struct field sgps_position_valid is true. Returns false if either sgps_fix_quality or sgps_position_valid contains 0. The struct fields are updated by GPS_Update; see its glossary entry.

C: xaddr **GPS_Inbuf** (void)
4th: **GPS_Inbuf** (-- xaddr)
Returns the xaddress of the buffer used to hold the raw serial sentences from the GPS subsystem by fetching the 32-bit contents of the sgps_inbuf field in the gps_info struct. This routine requires that GPS_Struct_Init or GPS_Init has already been invoked. If GPS_Init has been invoked, then GPS_Inbuf will return gps_default_inbuf as the input buffer specifier; see its glossary entry.

C: **gps_info**
4th: **gps_info** (-- xaddr)
The struct instance in common RAM that contains data obtained from the Garmin GPS subsystem. The struct elements are updated by the functions GPS_Run, GPS_Update, GPS_Frame_Extract, and by functions having names that start with GPS_Parse. The elements of the gps_info struct are described at the start of this glossary.
C structure use example: To read the contents of the structure element named sgps_lat_degrees in C, use the standard C construct:

```
gps_info.sgps_lat_degrees
```

as the right-hand side of an assignment statement.
Forth structure use example: To read the contents of the structure element named sgps_lat_degrees in Forth, use the standard Forth construct:

```
gps_info sgps_lat_degrees @
```

to fetch the contents to the data stack.

C: void **GPS_Info_Dump** (void)
4th: **GPS_Info_Dump** (--)
A useful debugging function that prints a formatted summary of the data stored in the gps_info struct to the serial port. This function is typically called after GPS_Update has executed to populate the gps_info struct with the latest data. Note that the units are converted and reported as miles per hour and feet (instead of knots and meters). Latitude and longitude are expressed in degrees (deg) and minutes (‘). The reported local time is based on the local_hour_offset parameter passed to GPS_Run, GPS_Init or GPS_Struct_Init. See also GPS_Dump, GPS_Run and GPS_Demo. Here is a sample of the formatted printout generated by the GPS_Info_Dump function:

```
UTC Universal time (hh:mm:ss): 20:46:07 and date (mm/dd/yy): 02/22/07
Local time (hh:mm:ss): 12:46:07 and date (mm/dd/yy): 02/22/07
Latitude: 37 deg 32.3814’ N
Longitude: 122 deg 01.1624’ W
Altitude (feet):     55.1
Ground speed (Miles Per Hour):      0.0
Ground course (degrees true): 194.8
Magnetic variation (degrees): 015.1 E
Satellites: 6  in use.   12 in view.
Fix available: Yes.   Valid position: Yes.
```

Position error (feet): Horizontal: 23.9 Vertical: 36.1 Overall: 43.3

C: int **GPS_Init** (int local_hour_offset, int module_num)
4th: **GPS_Init** (local_hour_offset\module_num -- error)
This is the high level initialization function for the GPS Wildcard; it must be called after each powerup or software reset before using any of the GPS driver functions. This routine globally enables interrupts, starts the timeslicer, and calls GPS_Default_UART_Init to initialize the UART chip that communicates with the Garmin GPS subsystem. It then calls GPS_Struct_Init, passing it the base xaddress of the gps_info struct which holds all of the GPS result fields, and the base xaddress of the gps_default_inbuf buffer which holds the raw serial sentences from the GPS subsystem. The module_num passed to this function is the Wildcard's bus address and it must match the hardware jumper settings of J1 and J2 on the GPS Wildcard. The local_hour_offset is the number of hours that must be added to the universal "UTC" (Greenwich) time to obtain the correct local time. In general, the local_hour_offset for a location with "West" longitude such as the United States has a negative local_hour_offset, while a location with "East" longitude has a positive local_hour_offset. The pre-defined constants GPS_PACIFIC_TIME, GPS_MOUNTAIN_TIME, GPS_CENTRAL_TIME, and GPS_EASTERN_TIME can be passed to this routine to specify the named time zones. To specify daylight savings time, add 1 to the specified time zone constant before passing it to this function (see GPS_DAYLIGHT_TIME). GPS_Init returns a true error flag if an ASCII '$' character is not received from the Garmin GPS subsystem within 1 second. A nonzero value returned by this function indicates that the GPS is not present or is malfunctioning. See also GPS_Shutdown.

C: int **GPS_Knots_Times_10** (void)
4th: **GPS_Knots_Times_10** (-- n)
The Garmin GPS subsystem reports speed over the ground in "knots", or nautical miles per hour. A nautical mile is 1.152 times longer than a standard (statute) mile. The speed can range from 000.0 to 999.9 knots. To maximize flexibility in the selection of fast integer math or slower floating point math when performing speed computations, the current speed value returned by this function is scaled up by a factor of 10 and returned as a 16-bit integer in the range 0 to 9,999. The numeric value of the number returned by this routine is the current ground speed knots times 10. The dimensions (units) of the returned values are tenths of a knot. To calculate a floating point speed in miles per hour, convert the value returned by this field to a floating point number and multiply by 0.1152.
Implementation detail: This function returns the contents of the sgps_tenth_knots field.
The struct fields are updated by GPS_Update; see its glossary entry.

C: int **GPS_Lat_Degrees** (void)
4th: **GPS_Lat_Degrees** (-- n)
The Garmin GPS subsystem reports latitude in degrees and minutes, where a minute is 1/60 of a degree. This function returns the most recent latitude expressed as an integer number of degrees north or south of the equator. Values range from 0 to 89 degrees. The latitude hemisphere (N or S) is returned by GPS_Lat_Hemisphere. For a latitude resolution better than 1 degree, use the GPS_Lat_Minutes_Times_10000 function to access the latitude minutes.
Implementation detail: This function returns the contents of the sgps_lat_degrees struct field.
The struct fields are updated by GPS_Update; see its glossary entry.

C: char **GPS_Lat_Hemisphere** (void)
4th: **GPS_Lat_Hemisphere** (-- char) ascii 'N' or 'S'
Returns the current latitude hemisphere. Returns ASCII 'N' (=0x4E) for latitudes north of the equator, or ASCII 'S' (0x53) for latitudes south of the equator.
Implementation detail: Returns the contents of the sgps_lat_hemisphere struct field.
The struct fields are updated by GPS_Update; see its glossary entry.

C: long **GPS_Lat_Minutes_Times_10000** (void)
4th: **GPS_Lat_Minutes_Times_10000** (-- d)
The Garmin GPS subsystem reports latitude in degrees and minutes, where a minute is 1/60 of a degree. Degrees are expressed as an integer between 0 and 89, while the minutes represented as numbers in the range 0.0000 to 59.9999. To maximize flexibility in the selection of fast integer math or slower floating point math when performing latitude computations, the current latitude minute value returned by this function is scaled up by a factor of 10000 and returned as a 32-bit integer in the range 0 to 599,999. The numeric value of the number returned by this routine is the current latitude minutes times 10000. The dimensions (units) of the returned values are ten-thousandths of a minute.
Implementation detail: This function returns the sum of contents of the sgps_lat_minutes_fraction field, plus 10000 times the contents of the sgps_lat_minutes_intpart field.
The struct fields are updated by GPS_Update; see its glossary entry. See also GPS_Lat_Hemisphere and GPS_Lat_Degrees.

C: int **GPS_Long_Degrees** (void)
4th: **GPS_Long_Degrees** (-- n)
The Garmin GPS subsystem reports longitude in degrees and minutes, where a minute is 1/60 of a degree. This function returns the most recent longitude expressed as an integer number of degrees east or west of the "prime meridian" which runs through Greenwich England. Values range from 0 to 179 degrees. The longitude hemisphere (E or W) is returned by GPS_Long_Hemisphere. For a longitude resolution better than 1 degree, use the GPS_Long_Minutes_Times_10000 function to access the longitude minutes.
Implementation detail: This function returns the contents of the sgps_long_degrees struct field.
The struct fields are updated by GPS_Update; see its glossary entry.

C: char **GPS_Long_Hemisphere** (void)
4th: **GPS_Long_Hemisphere** (-- char)
Returns the current longitude hemisphere with respect to the "prime meridian" which runs through Greenwich England. Returns ASCII 'W' (=0x57) for longitudes west of the prime meridian, or ASCII 'E' (0x45) for longitudes east of the prime meridian.
Implementation detail: Returns the contents of the sgps_long_hemisphere struct field.
The struct fields are updated by GPS_Update; see its glossary entry.

C: long **GPS_Long_Minutes_Times_10000** (void)
4th: **GPS_Long_Minutes_Times_10000** (-- d)
The Garmin GPS subsystem reports longitude in degrees and minutes, where a minute is 1/60 of a degree. Degrees are expressed as an integer between 0 and 179, while the minutes represented as

numbers in the range 0.0000 to 59.9999. To maximize flexibility in the selection of fast integer math or slower floating point math when performing longitude computations, the current longitude minute value returned by this function is scaled up by a factor of 10000 and returned as a 32-bit integer in the range 0 to 599,999. The numeric value of the number returned by this routine is the current longitude minutes times 10000. The dimensions (units) of the returned values are ten-thousandths of a minute.
Implementation detail: This function returns the sum of contents of the sgps_long_minutes_fraction field, plus 10000 times the contents of the sgps_long_minutes_intpart field.
The struct fields are updated by GPS_Update; see its glossary entry. See also GPS_Long_Hemisphere and GPS_Long_Degrees.

C: **GPS_MODULE_NUM**
4th: **GPS_MODULE_NUM** (-- n)
A constant in the demonstration program whose value equals the Wildcard module number. This number is the Wildcard's bus address and it must correspond to the jumper settings as shown in Table 1-2. Edit the source code of the demonstration program so that the value of this constant matches your hardware jumper settings. This constant is used in the GPS_Demo function.

C: **GPS_MOUNTAIN_TIME**
4th: **GPS_ MOUNTAIN_TIME** (-- n)
A constant whose value equals -7, used as the local_hour_offset passed to the GPS_Struct_Init, GPS_Init and GPS_Run functions to specify mountain standard time. The local_hour_offset is the number of hours that must be added to the universal "UTC" (Greenwich) time to obtain the correct local time. In general, the local_hour_offset for a location with "West" longitude such as the United States has a negative local_hour_offset, while a location with "East" longitude has a positive local_hour_offset. To specify daylight savings time, add 1 to this time zone constant before passing it to this function. See also GPS_PACIFIC_TIME, GPS_CENTRAL_TIME, GPS_EASTERN_TIME, and GPS_DAYLIGHT_TIME.

C: int **GPS_Next_Frame** (int module_num)
4th: **GPS_Next_Frame** (module -- numchars)
This function receives 7 GPS sentences which start with ASCII '$' and end with a carriage return/linefeed; this ensures that a full "frame" of sentence data from the Garmin GPS subsystem is stored. Once the GPS has been initialized, it outputs 6 or 7 sentence types each second, with sentence identifiers $GPRMC, $GPGGA, $GPGSA, $GPGSV, $PGRME, $PGRMT (only once per minute), and PGRMM. The $PGRMT sentence is output only once per minute. This function emplaces the ASCII sentences at the gps_inbuf buffer, and the number of characters stored is clamped to GPS_BUFSIZE which is adequate to accommodate the full 1-second frame of data. The module_num passed to this function is the Wildcard's bus address and it must match the hardware jumper settings of J1 and J2 on the GPS Wildcard. This function returns the number of characters emplaced in the buffer. GPS_Get_Frame can take over 1 second to obtain 7 sentences, so it is best placed in its own task where it will not block time-critical code. The ASCII character sentences stored by this function are used by GPS_Frame_Extract to parse the data into the gps_info struct fields where they become easily accessible by the application program. The high level function GPS_Update invokes GPS_Next_Frame and GPS_Frame_Extract; see their glossary entries.

Implementation detail: Flushes all existing characters out of the UART input FIFO (first in/first out buffer) on the GPS Wildcard to get rid of any partial sentences that might be in the FIFO, and then puts characters into the specified xbufbase buffer as described above. Each sentence is recognized by its starting '$' and its ending linefeed character, but the terminating ASCII linefeed (0x0A) is not stored in the buffer. Thus each sentence in the buffer ends with the ASCII carriage return (0x0D) character. After each sentence (except the last one received) this routine calls PAUSE while waiting for the starting ASCII '$' of the next sentence. This ensures that the function is calling PAUSE to share processor time while waiting for the next sentence. Once the GPS has been initialized, it outputs 6 or 7 sentence types each second. The $PGRMT sentence is output only once per minute. An example 1-second frame of data stored by this function is as follows:

```
$GPRMC,223324,A,3732.3793,N,12201.1625,W,0.0,194.8,220207,15.1,E*5D
$GPGGA,223324,3732.3793,N,12201.1625,W,1,08,0.9,11.3,M,-28.0,M,,*48
$GPGSA,A,3,03,,07,09,,16,18,19,21,,,26,1.6,0.9,1.3*32
$GPGSV,3,2,12,14,24,188,00,16,13,244,39,18,71,018,46,19,15,319,48*7B
$PGRME,5.0,M,5.8,M,7.7,M*26
$PGRMT,Bravo PDA Ver. 2.01,P,P,R,R,P,,21,*6C
$PGRMM,WGS 84*06
```

C: int **GPS_Numsats_In_Use** (void)
4th: **GPS_Numsats_In_Use** (-- n)
Returns the number of GPS satellites currently in use by the 12-channel Garmin GPS receiver. This value can range from 0 to 12. Note that, in most cases, the GPS antenna must be outside with a clear view of the sky for the Wildcard to acquire and use GPS satellites.
Implementation detail: This function returns the contents of the sgps_numsats_in_use struct field. The struct fields are updated by GPS_Update; see its glossary entry. See also GPS_Numsats_In_View and GSP_Good_Fix.

C: int **GPS_Numsats_In_View** (void)
4th: **GPS_Numsats_In_View** (-- n)
Returns the number of GPS satellites currently in view by the 12-channel Garmin GPS receiver. This value can range from 0 to 12. A satellite may be "in view" but not "in use". Note that, in most cases, the GPS antenna must be outside with a clear view of the sky for the Wildcard to acquire and use GPS satellites.
Implementation detail: This function returns the contents of the sgps_numsats_in_view struct field. The struct fields are updated by GPS_Update; see its glossary entry. See also GPS_Numsats_In_Use and GSP_Good_Fix.

C: **GPS_PACIFIC_TIME**
4th: **GPS_PACIFIC_TIME** (-- n)
A constant whose value equals -8, used as the local_hour_offset passed to the GPS_Struct_Init, GPS_Init and GPS_Run functions to specify pacific standard time. The local_hour_offset is the number of hours that must be added to the universal "UTC" (Greenwich) time to obtain the correct local time. In general, the local_hour_offset for a location with "West" longitude such as the United States has a negative local_hour_offset, while a location with "East" longitude has a positive local_hour_offset. To specify daylight savings time, add 1 to this time zone constant before passing

it to this function. See also GPS_MOUNTAIN_TIME, GPS_CENTRAL_TIME, GPS_EASTERN_TIME, and GPS_DAYLIGHT_TIME.

C: int **GPS_Parse_Altitude_Fix_Numsats** (xaddr xbuf, int dollar_offset, int maxchars)
4th: **GPS_Parse_Altitude_Fix_Numsats** (xbuf\ dollar_offset \maxchars -- error)
This low level utility function expects that the $GPGGA sentence is in the specified xbuf buffer with the specified dollar_offset pointing to the ASCII '$' at the start of the $GPGGA sentence, and maxchars specifying the maximum size of the buffer. In other words, the sum of xbuf and dollar_offset must yield the xaddress of the ASCII '$' at the start of the $GPGGA sentence in the buffer. This function is called by GPS_Get_Altitude_Fix_Numsats, and is automatically called by the higher level functions GPS_Frame_Extract and GPS_Update; see their glossary entries. GPS_Parse_Altitude_Fix_Numsats extracts the GPS fix quality (0= fix not available, 1=non-differential gps fix available), number of GPS satellites in use, and altitude (antenna height in meters with respect to sea level). If parsing of the sentence fails due to a bad checksum or other error, no data is stored and a -1 error flag is returned. If parsing of the sentence is successful, this routine returns 0 and updates the following fields in the gps_info struct: sgps_fix_quality, sgps_numsats_in_use, and sgps_altitude_tenth_meters

C: int **GPS_Parse_Pos_Errors** (xaddr xbuf, int dollar_offset, int maxchars)
4th: **GPS_Parse_Pos_Errors** (xbuf\dollar_offset\maxchars -- error)
This low level utility function expects that the $PGRME sentence is in the specified xbuf buffer with the specified dollar_offset pointing to the ASCII '$' at the start of the $PGRME sentence, and maxchars specifying the maximum size of the buffer. In other words, the sum of xbuf and dollar_offset must yield the xaddress of the ASCII '$' at the start of the $PGRME sentence in the buffer. This function is called by GPS_Get_Pos_Errors, and is automatically called by the higher level functions GPS_Frame_Extract and GPS_Update; see their glossary entries. GPS_Parse_Pos_Errors extracts the estimated horizontal, vertical, and overall position errors reported by the Garmin GPS subsystem. Reported errors are in the range 0 to 999.9 meters. If parsing of the sentence fails due to a bad checksum or other error, no data is stored and a -1 error flag is returned. If parsing of the sentence is successful, this routine returns 0 and updates the following fields in the gps_info struct: sgps_hor_error_tenth_meters, sgps_vert_error_tenth_meters, and sgps_position_error_tenth_meters

C: int **GPS_Parse_Pos_Speed_Course_Time** (xaddr xbuf, int dollar_offset, int maxchars)
4th: **GPS_Parse_Pos_Speed_Course_Time** (xbuf\ dollar_offset\ maxchars -- error)
This low level utility function expects that the $GPRMC sentence is in the specified xbuf buffer with the specified dollar_offset pointing to the ASCII '$' at the start of the $GPRMC sentence, and maxchars specifying the maximum size of the buffer. In other words, the sum of xbuf and dollar_offset must yield the xaddress of the ASCII '$' at the start of the $GPRMC sentence in the buffer. This function is called by GPS_Get_Pos_Speed_Course_Time, and is automatically called by the higher level functions GPS_Frame_Extract and GPS_Update; see their glossary entries. GPS_Parse_Pos_Speed_Course_Time extracts the universal (UTC) time and date, position validity flag, latitude and longitude (degrees, minutes, hemisphere), speed in knots, course over ground in degrees true, and magnetic variation. Based on the local_hour_offset parameter passed to GPS_Run, GPS_Init or GPS_Struct_Init, this routine calculates the local time and date. If parsing of the sentence fails due to a bad checksum, missing field, or other error, no data is stored and a -1 error

flag is returned. If parsing of the sentence is successful, this routine returns 0 and updates the following fields in the gps_info struct: sgps_utc_hour, sgps_utc_date, sgps_utc_month, sgps_utc_year, sgps_local_second, sgps_local_minute, sgps_local_hour, sgps_local_date, sgps_local_month, sgps_local_year, sgps_position_valid, sgps_lat_degrees, sgps_lat_minutes_intpart, sgps_lat_minutes_fraction, sgps_lat_hemisphere, sgps_long_degrees, sgps_long_minutes_intpart, sgps_long_minutes_fraction, sgps_long_hemisphere, sgps_tenth_knots, sgps_course_tenth_degrees, sgps_mag_var_tenth_degrees, and sgps_mag_var_direction.

C: int **GPS_Parse_Sats_In_View** (xaddr xbuf, int dollar_offset, int maxchars)
4th: **GPS_Parse_Sats_In_View** (xbuf \ dollar_offset\ maxchars -- error)
This low level utility function expects that the $GPGSV sentence is in the specified xbuf buffer with the specified dollar_offset pointing to the ASCII '$' at the start of the $GPGSV sentence, and maxchars specifying the maximum size of the buffer. In other words, the sum of xbuf and dollar_offset must yield the xaddress of the ASCII '$' at the start of the $GPGSV sentence in the buffer. This function extracts the third numeric field of the $GPGSV sentence, which is the satellites in view field. If a valid 2-byte field is found, this function returns 0 and stores the binary representation of the field in the sgps_numsats_in_view element of the gps_info struct. If valid data is not found, this routine stores nothing and returns a -1 error flag. This function is called by GPS_Get_Sats_In_View, and is automatically called by the higher level functions GPS_Frame_Extract and GPS_Update; see their glossary entries.

C: int **GPS_Run** (int local_hour_offset, int module_num)
4th: **GPS_Run** (local_hour_offset\module_num --)
A high level function in the demonstration program; see its source code listing at the end of this document. This function passes the specified local_hour_offset and module_num to the GPS_Init function. <u>The module_num passed to this function is the Wildcard's bus address and it must match the hardware jumper settings of J1 and J2 on the GPS Wildcard</u>. The local_hour_offset is the number of hours that must be added to the universal "UTC" (Greenwich) time to obtain the correct local time. In general, the local_hour_offset for a location with "West" longitude such as the United States has a negative local_hour_offset, while a location with "East" longitude has a positive local_hour_offset. The pre-defined constants GPS_PACIFIC_TIME, GPS_MOUNTAIN_TIME, GPS_CENTRAL_TIME, and GPS_EASTERN_TIME can be passed to this routine to specify the named time zones. To specify daylight savings time, add 1 to the specified time zone constant before passing it to this function (see GPS_DAYLIGHT_TIME). After initialization, the function prints a descriptive text introduction, and enters a loop that repeatedly calls GPS_Update to get a frame of data from the GPS subsystem, and periodically calls GPS_Info_Dump to print to the serial port a formatted version of the extracted data from the gps_info struct. Depending on the prior state of the GPS, a valid fix can be acquired in a few seconds, or it can take as long as 5 minutes if the GPS subsystem does not have enough stored time and location information to speed its acquisition process. Typing any key terminates the function. This demonstration function is meant to be invoked interactively using the Mosaic terminal. See also GPS_Demo.

C: void **GPS_Shutdown** (int shutdown_flag, int module_num)
4th: **GPS_Shutdown** (shutdown_flag\module_num --)
This routine shuts down the Garmin GPS subsystem if the shutdown_flag is nonzero, or powers up the GPS subsystem if the shutdown_flag is zero. Shutting down the GPS typically saves 1.25 watt,

equivalent to 0.25 amp at 5 volts. The module_num passed to this function is the Wildcard's bus address and it must match the hardware jumper settings of J1 and J2 on the GPS Wildcard. To force a Garmin GPS subsystem reset and satellite reaquisition, call this routine with a true shutdown_flag, then call it again with a false shutdown_flag. The GPS is powered up by GPS_Default_UART_Init and its calling function GPS_Init.

C: **GPS_STRUCT**
4th: **GPS_STRUCT** (-- n)
This struct specifier is used to declare the gps_info struct in common RAM. In C, this is a typedef, and in Forth, it returns the struct size of the gps_info struct. The elements of the gps_info struct are described at the start of this glossary; they hold data obtained from the Garmin GPS subsystem. The struct elements are updated by the functions GPS_Run, GPS_Update, GPS_Frame_Extract, and by functions having names that start with GPS_Parse.
C structure use example: To read the contents of the structure element named sgps_lat_degrees in C, use the standard C construct:

```
gps_info.sgps_lat_degrees
```

as the right-hand side of an assignment statement.
Forth structure use example: To read the contents of the structure element named sgps_lat_degrees in Forth, use the standard Forth construct:

```
gps_info sgps_lat_degrees @
```

to fetch the contents to the data stack.

C: void **GPS_Struct_Init** (xaddr gps_info_xbase, xaddr xinbuf, int local_hour_offset)
4th: **GPS_Struct_Init** (gps_info_xbase \xinbuf\local_hour_offset --)
This function zeros the GPS_STRUCT starting at the specified gps_info_xbase, and blanks the GPS_BUFSIZE bytes starting at the specified xinbuf which holds the raw serial sentences from the GPS subsystem. This routine then stores the specified xinbuf into the spgs_inbuf field of the GPS_STRUCT, and stores the specified local_hour_offset into the sgps_local_hour_offset field of the GPS_STRUCT. This low-level function is called by GPS_Init; see its glossary entry for additional details.

C: int **GPS_Update** (int module_num)
4th: **GPS_Update** (module -- error)
This high level function invokes GPS_Next_Frame to get a frame of GPS ASCII data into the gps_inbuf, then calls GPS_Frame_Extract to parse the GPS sentences in the order that they appear, extracting data from these sentences and storing the data into the gps_info struct. The module_num passed to this function is the Wildcard's bus address and it must match the hardware jumper settings of J1 and J2 on the GPS Wildcard. GPS_Update can take over 1 second to obtain a frame of data from the GPS subsystem, so it is best placed in its own task where it will not block time-critical code. Make sure to call GPS_Init before invoking GPS_Update. For an example of use, see the GPS_Run and GPS_Demo functions which are provided in source code form in the demonstration program listing at the end of this document.

Listing 1-3 Forth GPS_STRUCT commented source code definition.

```
STRUCTURE.BEGIN:  GPS_STRUCT     \ holds information for this wildcard
XADDR-> sGPS_INBUF  \ holds base xaddr of buffer for sentences coming from gps
XADDR-> sGPS_OUTBUF \ holds base xaddr of buffer for outgoing sentences: unused!
INT-> sGPS_LOCAL_HOUR_OFFSET \ eg, PST (pac std time) = -8, PDT = -7; EST = -5
\ the following are from the $GPRMC data from gps sensor:
INT-> sGPS_UTC_HOUR       \ 00-23 hour
INT-> sGPS_UTC_DATE       \ 01-31 date of the month
INT-> sGPS_UTC_MONTH      \ 01-12 month of the year
INT-> sGPS_UTC_YEAR       \ 00-99 2-digit year
INT-> sGPS_LOCAL_SECOND   \ 00-59 seconds after minute; utc_second = local_second
INT-> sGPS_LOCAL_MINUTE   \ 00-59 minutes after hour; utc_minute = local_minute
INT-> sGPS_LOCAL_HOUR     \ 00-23 hour = (utc_hour + local_time_hour_offset) mod 24
INT-> sGPS_LOCAL_DATE     \ 01-31 date of the month, result of utc->local conversion
INT-> sGPS_LOCAL_MONTH    \ 01-12 month of the year, result of utc->local conversion
INT-> sGPS_LOCAL_YEAR     \ 00-99 2-digit year, result of utc->local conversion
INT-> sGPS_POSITION_VALID \ holds BOOLEAN flag; -1 = valid pos; ‘0 = rcvr warning
INT-> sGPS_LAT_DEGREES    \ 00-89 degrees latitude
INT-> sGPS_LAT_MINUTES_INTPART  \ 00-59 integer latitude minutes
INT-> sGPS_LAT_MINUTES_FRACTION \ .xxxx latitude minutes fractional part (ten-
thousandths)
INT-> sGPS_LAT_HEMISPHERE        \ ascii ‘N’ or ‘S’
INT-> sGPS_LONG_DEGREES          \ 000-179 degrees longitude
INT-> sGPS_LONG_MINUTES_INTPART  \ 00-59 integer longitude minutes
INT-> sGPS_LONG_MINUTES_FRACTION \ .xxxx longitude minutes fraction part (ten-
thousandths)
INT-> sGPS_LONG_HEMISPHERE \ ascii ‘E’ or ‘W’
INT-> sGPS_TENTH_KNOTS     \ 0-9999 speed over ground; field = 000.0 to 999.9 knots
INT-> sGPS_COURSE_TENTH_DEGREES  \ 0-3599 course; field= 000.0 to 359.9 degrees true
INT-> sGPS_MAG_VAR_TENTH_DEGREES \ 0-1800 magnetic variation, 000.0 to 180.0 degrees
INT-> sGPS_MAG_VAR_DIRECTION \ ‘E’ or ‘W’ mag var dir;west var adds to true course
\ the following is from the $GPGSV data from gps sensor:
INT-> sGPS_NUMSATS_IN_VIEW    \ 00-12
\ the following are from the $GPGGA data from gps sensor:
INT-> sGPS_FIX_QUALITY    \ binary: 0= fix not available; 1 = non-diff fix avail
INT-> sGPS_NUMSATS_IN_USE \ 00-12 = number of satellites in use
DOUBLE-> sGPS_ALTITUDE_TENTH_METERS  \ -9999.9 to 99999.9 meters,antenna hght wrt SL
\ the following are from the $PGRME data from gps sensor:
INT-> sGPS_HOR_ERROR_TENTH_METERS  \ 0-9999 meters est horiz pos err, fld=0 to 999.9
INT-> sGPS_VERT_ERROR_TENTH_METERS \ 0-9999 meters est vert pos err, fld=0 to 999.9
INT-> sGPS_POSITION_ERROR_TENTH_METERS \ 0-9999 meter overall pos err,fld=0 to 999.9
STRUCTURE.END
```

C Demonstration Program

This section presents the ANSI C version of the demonstration program source code.

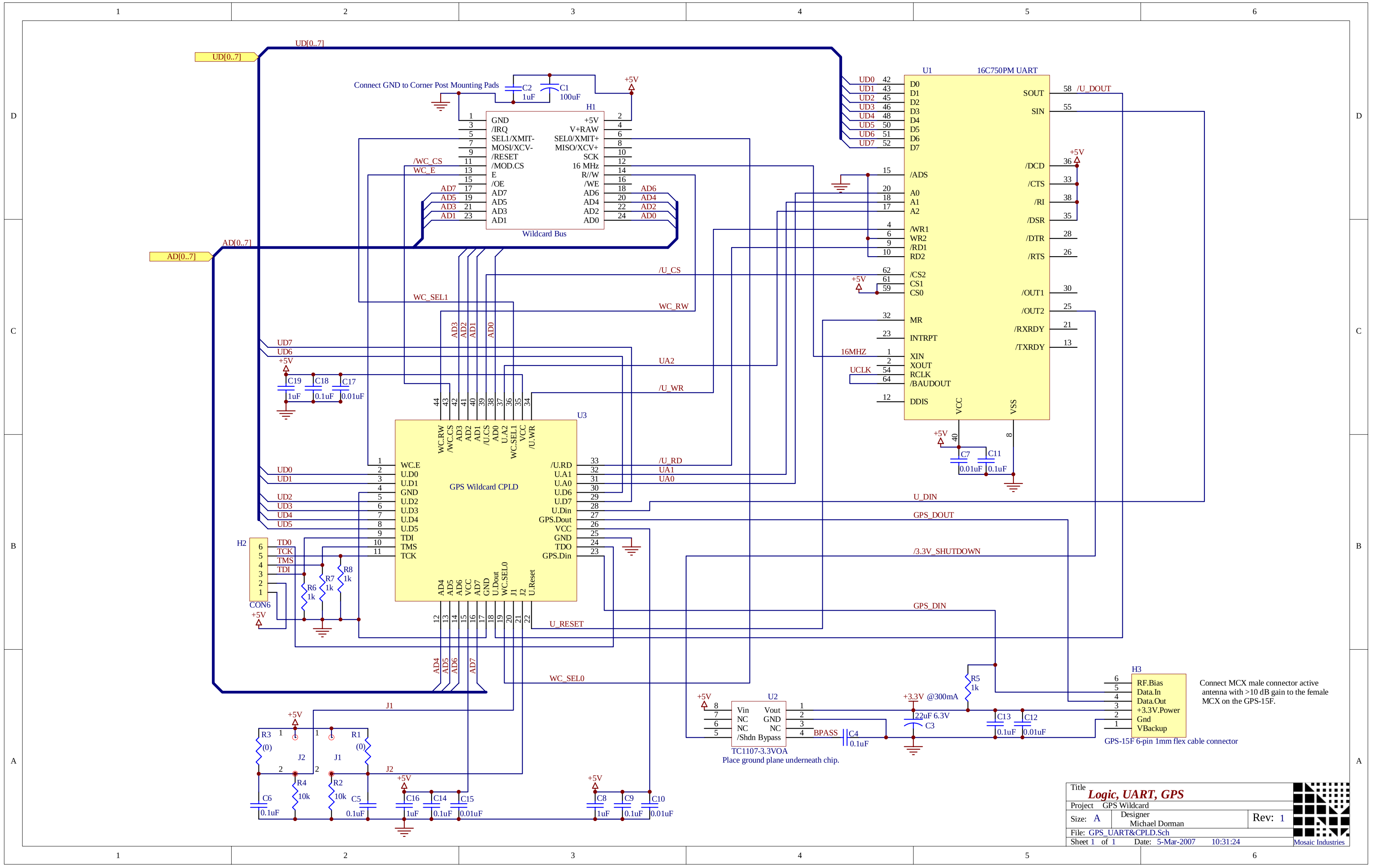
```
// Filename: GPS_WC_Demo.c
// Date: Feb 27, 2007
// copyright 2007 Mosaic Industries, Inc.  All rights reserved.


// This is a demonstration program for the GPS Wildcard.
// Make sure that GPS_MODULE_NUM matches your hardware jumper settings.
// Note: in real applications, the GPS service should be invoked from a separate task.


#define GPS_MODULE_NUM 0   // MUST match hardware jumper settings!

void GPS_Run ( int local_hour_offset, int module_num )
// this function initializes the GPS Wildcard and prints a formatted summary
// of GPS info to the terminal every 5 seconds.  Type any key to terminate the function.
// Note: in real applications, the GPS service should be invoked from a separate task.
{  int print_counter;
   if(GPS_Init(local_hour_offset, module_num))
      printf("GPS_Init FAILED!\r\n");     // we're done if GPS_Init error flag is true
   else                            // if initialization ok, proceed...
   {  printf("\r\nStarting GPS info dump.  Type any key to exit this function.\r\n");
      printf("This function gets a data frame from the GPS once per second,");
      printf("and prints a statement telling of the fix validity.");
      printf("If the fix is not valid, make sure your antenna is outside,");
      printf("and give the GPS some time (5 minutes worst case) to acquire its satellites.");
      printf("This function periodically calls GPS_Info_Dump to summarize the GPS data.\r\n");
      while(?Key==0)                 // type any key to end this routine
      {  printf("\r\n\r\nValid fix?: ");
         if(GPS_Update(module_num)      // runs every second
            printf("NO");            // if GPS_Update returned true error flag: no fix
         else
            printf("YES");           // if no GPS_Update error: yes, we have a fix
         if(print_counter++ == 5)       // detailed print every 5 seconds
         {  print_counter = 0;          // restart counter
            GPS_Info_Dump();         // print formatted extracted gps info
         }
      }
   }
}


void GPS_Demo ( void )
// this function initializes the GPS Wildcard with a specified time zone and module number,
// and prints a formatted summary of GPS info to the terminal every 5 seconds.
// Type any key to terminate the function.
// Note: in real applications, the GPS service should be invoked from a separate task.
{  GPS_Run(GPS_PACIFIC_TIME, GPS_MODULE_NUM);
}

void main(void)
{  GPS_Demo();
}
```

Forth Demonstration Program

This section presents the Forth version of the demonstration program source code.

```
\ Filename: GPS_WC_Demo.c
\ Date: Feb 27, 2007
\ copyright 2007 Mosaic Industries, Inc.  All rights reserved.

\ This is a demonstration program for the GPS Wildcard.
\ Make sure that GPS_MODULE_NUM matches your hardware jumper settings.
\ Note: in real applications, the GPS service should be invoked from a separate task.

HEX
FIND WHICH.MAP         \ do this only for page-swapping platforms!
IFTRUE                 \ nesting is allowed if ends are sequential
  EXECUTE  0=          ( -- standard.map? ) \ run which.map
  IFTRUE 4 PAGE.TO.RAM  \ if in standard.map, transfer to download map
         5 PAGE.TO.RAM
         6 PAGE.TO.RAM
        DOWNLOAD.MAP
  ENDIFTRUE
ENDIFTRUE

\ if your memory map is not already set, set it here after load of GPS driver:
\ 800 4 DP X!  5800 4 NP X!  \ for kernel V4.xx
\ 0x8000 1 DP X!  0x8000 0x11 NP X!  \ for kernel V6.xx

F WIDTH !      \ set width of names stored in dictionary

ANEW GPSDemo_Code   \ define forget marker for easy re-loading


0 CONSTANT GPS_MODULE_NUM  \ MUST match hardware jumper settings!

: GPS_RUN   ( local_hour_offset\module_num -- )
\ this function initializes the GPS Wildcard and prints a formatted summary
\ of GPS info to the terminal every 5 seconds.  Type any key to terminate the function.
\ Note: in real applications, the GPS service should be invoked from a separate task.
   2 NEEDED DUP 0
   LOCALS{ &print_counter &module } ( local_hour_offset\module_num -- )
   GPS_INIT 0= ( ok? -- )
   IF CR ." Starting GPS info dump.  Type any key to exit this function."
      CR ." This function gets a data frame from the GPS once per second,"
      CR ." and prints a statement telling of the fix validity."
      CR ." If the fix is not valid, make sure your antenna is outside,"
      CR ." and give the GPS some time (5 minutes worst case) to acquire its satellites."
      CR ." This function periodically calls GPS_Info_Dump to summarize the GPS data."
      CR CR
      BEGIN
         &module GPS_UPDATE  CR CR ." Valid fix?: " ( error -- ) \ runs every second
         IF ." NO"
         ELSE ." YES"
         ENDIF
         &print_counter 1+ DUP TO &print_counter
         5 =                           \ detailed print every 5 seconds
         IF 0 TO &print_counter
            GPS_INFO_DUMP                 \ print extracted info
            CR CR
         ENDIF
         ?KEY UNTIL  \ type any key to end this routine
   ELSE CR ." GPS–INIT FAILED!"
   ENDIF
;

: GPS–DEMO  ( -- )
\ this function initializes the GPS Wildcard with a specified time zone and module number,
\ and prints a formatted summary of GPS info to the terminal every 5 seconds.
\ Type any key to terminate the function.
\ Note: in real applications, the GPS service should be invoked from a separate task.
   GPS–PACIFIC–TIME GPS–MODULE–NUM ( local–hour–offset\module–num -- )
   GPS–RUN
;


FIND WHICH.MAP
IFTRUE                \ for kernel V4.xx platforms...
    XDROP             ( -- ) \ drop xcfa
   4 PAGE.TO.FLASH
   5 PAGE.TO.FLASH
   6 PAGE.TO.FLASH
   STANDARD.MAP
   SAVE
OTHERWISE            \ for V6.xx kernels, store to shadow flash and save pointers
    SAVE.ALL .     \ this takes some time, should print FFFF for success
ENDIFTRUE
```

Hardware Schematics


U1 16C750PM UART
H1 Wildcard Bus
U3 GPS Wildcard CPLD
U2 TC1107-3.3VOA
Place ground plane underneath chip.
H3
GPS-15F 6-pin 1mm flex cable connector
Connect MCX male connector active antenna with >10 dB gain to the female MCX on the GPS-15F.
Connect GND to Corner Post Mounting Pads
H2 CON6
UD[0..7]
AD[0..7]
/U_DOUT
U_DIN
GPS_DOUT
/3.3V_SHUTDOWN
GPS_DIN
U_RESET
WC_SEL0
WC_SEL1
/WC_CS
WC_E
/U_CS
WC_RW
UA2
/U_WR
/U_RD
UA1
UA0
16MHZ
UCLK
BPASS
+3.3V @300mA
+5V
J1
J2
Title Logic, UART, GPS
Project GPS Wildcard
Size: A
Designer Michael Dorman
Rev: 1
File: GPS_UART&CPLD.Sch
Sheet 1 of 1
Date: 5-Mar-2007 10:31:24
Mosaic Industries