CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

FEATURE ARTICLE

Jeff Stefan

Navigating with GPS

If you're going to be heading out with a minivan full of little ghosts and goblins this Halloween, you may want to brush up on some GPS navigation background info. Luckily, Jeff has all the details that you'll need to find your way.

lobal Positioning System (GPS) receivers are abundant and cheap, paving the way for anyone to write a simple yet powerful navigation program. All you need is a C compiler, a laptop or small computer, and a couple navigation formulae. Seems simple? It is if you know what data to use, what conversions to make, and which formulae to use. This article reveals the twists and turns required to put your GPS receiver to work and get you navigating quickly. The application program and functions I'll present allow you to calculate the distance and heading from your current position to any other position on earth. The distance and bearing functions provide the heart of a dynamic and useful navigation system.

GPS FUNDAMENTALS

GPS became available in 1978 with the successful launch of NAVSTAR 1. NAVSTAR 1 was the first of four NAVSTAR satellites launched that year, creating an operational satellite navigation system for the military. Then in 1982, Russia launched a system called GLONASS.

GPS satellites are incredible instruments. Each satellite contains four atomic clocks that operate on a level of one second of error in three million years. This degree of precision time keeping is required so each satellite can operate autonomously yet remain synchronized. GPS satellites transmit ranging codes based on a signal's time of arrival, not position and motion.

These satellites, which are at known locations at all times, transmit on two L-band carrier signals. The satellite's receiver marks the difference between the time the signal was sent and received, and multiplies the difference by the signal speed (close to the speed of light). Using ranging code from four satellites, a GPS receiver can calculate its own position in three-dimensional space, including the receiver's velocity.

The NAVSTAR system breaks down navigation into two domains, Standard Positioning Service (SPS) and Precise Positioning Service (PPS). PPS accuracy is published at 21-m horizontally and 29-m vertically. The early NAVSTAR SPS was so accurate that it was considered a threat, so the gap between SPS and PPS was intentionally widened. The accuracy level of the SPS was decreased to 100 m in the horizontal plane and 160 m in the vertical plane. The decrease, called

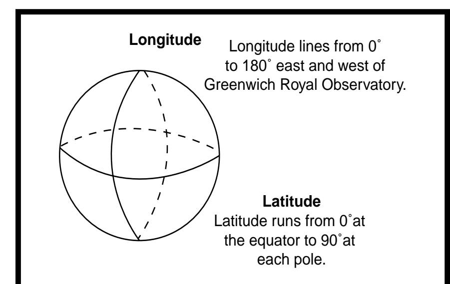

Figure 1—*Longitude lines run east and west from pole to pole. Latitude lines run north and south, parallel to the equator.*

selective availability (SA), introduced error into the satellite orbital data and time transmissions.

SA made life more difficult for commercial GPS-based navigation systems. One hundred meters (roughly 300′) of accuracy isn't bad, but if you're trying to develop a precise hand-held or automotive navigation system, more accuracy is needed. To the delight of the navigation community, the U.S. government turned off SA on May 1, 2000. Instead of 100 m, accuracy now is within 10 to 30 m in the horizontal plane and slightly more in the vertical plane.

Now, the floodgate is open for new and highly accurate GPS applications based on latitude, longitude, and time. GPS receivers turn up in everything from wristwatches to locomotives.

Latitude and longitude are fundamentals of navigation. Sometimes it's difficult to remember which is which. I use the mnemonic "it's a long way from the North Pole to the South Pole." Longitude lines run from the North Pole to the South Pole and are measured in half circles from the Royal Greenwich Observatory in Greenwich, UK. Longitude lines run from 0° to 180° east and 0° to 180° west (see Figure 1).

Latitude lines run in parallel from the equator to the North and South Poles. Latitude lines run from 0° at the equator to 90° at the North and South Poles. As the lines of latitude get closer to the poles, they become smaller. This presents a problem when trying to use a two-dimensional distance formula, as I'll explain later.

GETTING THE DATA

Most GPS receivers output data in NMEA-0183 format. NMEA stands for the National Marine Electronics Association. The data is sent via RS-232 at 4800 bps, with most GPS receivers providing a serial port that outputs NMEA GPS messages at 1-s intervals. NMEA messages are sent by talkers (identified by a two-character ID string with a "GP" prefix) and received by listeners. The messages are one-way: from talker to listener. Thus, an application program's control of the receiver's output is limited. Data enters your application program at 4800 bps at 1-s intervals, period. The data rate and burst time can't be changed. This is OK for most land- and sea-based applications.

An NMEA sentence contains an address field, data field, and checksum. The address field is composed of a sentence formatter and talker identifier. The latter indicates where the data comes from. For GPS, the talker identifier is GP. Two useful GPS talker identifiers are RMC and GGA. The sentence formatter indicates the content of the data field.

NMEA messages are easy for a program to parse because they are consistent and well defined. The general NMEA message format is:

$<Address>,<Data>*<Checksum><CR><LF>

The address field, <Address>, is broken down as <talker><sentence_ formatter>. All fields are comma delimited except <Checksum>, which is delimited by a star (see Figure 2).

Table 1 lists eight examples of NMEA GPS messages. The most useful one is the RMC message, which contains all of the basic information required to build a navigation system. RMC is listed as recommended minimum specific GPS/transit data (see Figure 3). Although I don't know what the "C" stands for in RMC, I know I like the utilitarian nature of this message. It contains time, status, position, speed, course, and date.

Looking at the RMC message, the first chunk of data encountered is $GPRMC. As the NMEA sentence describes, this is the talker and sentence formatter. Universal Time Coordinated UTC data follows the sentence formatter; the time is given in hours, minutes, seconds, and decimal seconds. Next is the GPS status indicator (A), which indicates whether or not the incoming GPS data is valid. The V in this field seems to indicate that the data is valid, however, it means the opposite. An A in this field means that the data is indeed valid.

There are many reasons why a GPS

Table 1—*These are useful GPS NMEA messages used for navigation applications.*

RMC	Contains recommended minimum specific GPS/transit data
ALM	Provides GPS almanac information
GLL	Provides latitude, longitude, and UTC (Universal Time Coodinated) data
ZDA	Contains UTC along with day, month, year, and local time
GGA	Contains UTC, fix, and position data
GSA	Provides GPS DOP and active satellite information
VTG	Provides "track made good" and ground speed
ZDA	Provides the current time and data

Listing 1—*Here's the structure that holds parsed RMC messages.*

```
#define STRING10 10
#define STRING7  7

struct rmc
{
   char UTC[STRING10];
   char Status;
   double Lat;
   char NS;
   double Lon;
   char EW;
   double Speed;
   int Course;
   char Date[STRING7];
   char Var[STRING10];
};
```

Ddmm.mmmm to dd.dddd	Separate and save dd from the incoming latitude and longitude Divide mm.mmmm by 60, yielding 0.dddd Add the saved dd to 0.dddd, yielding dd.dddd
dd.dddd to radians	Radians = dd.dddd/57.2957795
Radians to dd.dddd (degrees)	Degrees = radians × 57.2957795
Radians to nautical miles (NM)	NM = radians × 3437.7387
NM to statute (land) miles (MI)	MI = NM × 1.150779
MI to feet (FT)	FT = MI × 5280

Table 2—*Here are common navigation units and conversion factors.*

receiver would output invalid data. For example, the receiver might not have acquired enough satellites for a position fix yet, foliage or buildings might block the GPS signals, or the GPS almanac or ephemeris data could be out of date. Invalid data output from a receiver is almost always temporary, and a V usually will become an A within seconds or minutes.

The next two fields cover latitude and determine whether the latitude is in the Northern or Southern Hemisphere. Following the latitude fields are the corresponding longitude and east/west indicators. The two fields after that, speed (knots) over ground and course (degrees) over ground, are handy. Next is the date, and then the magnetic variation (east or west).

The RMC message is available on almost all receivers that output NMEA messages. As stated, GPS receivers supporting NMEA messages output data at 1-s intervals at 4800 bps, so processing data at 1200- to 1800-ms intervals ensures enough time to fill up a receiver buffer and transfer the data to a holding buffer. The data in the holding buffer can be parsed and processed while new information enters the receive buffer.

Listing 1 shows a C structure into which you can deposit the parsed RMC message data. The [`#defines`] identify character array lengths and are optional. The structure contains the proper data types to contain the RMC data fields. You can create similar structures for additional NMEA messages that an application needs.

After the GPS receiver deposits its data in a buffer named `InputQueue[]`, the data is transferred to another buffer called `InputMsgBuff[]`. The latter is used to extract the RMC or other NMEA messages of interest. To extract the data, create a pointer and have it point to `InputQueue[]`. For transferring data, you need to de-reference the pointer to `Input- MsgBuff[]`. This is illustrated in the C code fragment in Listing 2.

When a message is in a buffer, the talker and sentence formatter can be identified and processed. This bit of code collects a sequence of messages so multiple messages can be processed. This allows you to create custom structures, spanning the data from multiple messages. For example, a structure can be created that holds speed, course, latitude, longitude, the number of satellites in view, and dilution of precision values.

WAYPOINT NAVIGATION

Waypoint navigation is based on great circle navigation. Great circle navigation is general and good for planes, boats, and cars. Waypoint navigation systems navigate via latitude and longitude pairs. The navigation computer accesses a list of latitude/longitude pairs and calculates the distance and bearing from one point to another. Information presented is usually the current distance and bearing from your present position to the next waypoint. Often, a dynamic directional pointer is displayed, which you follow to the next waypoint.

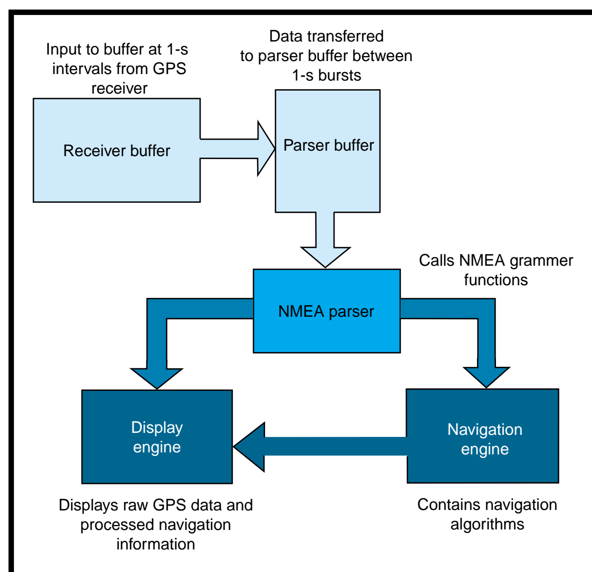

Figure 4—*The simple navigation system architecture is pictured here.*

Before navigating, the data from the GPS receiver must be converted to a form acceptable to the great circle navigation algorithms (i.e., the distance and bearing formulae). First and foremost, all of the data must be in radians. This seems straightforward, but there's a complication. The latitude and longitude data emitted by most receivers is in a form that cannot be directly converted to radians. So, an intermediate latitude and longitude conversion sequence must take place.

All NMEA data is emitted as ASCII data. Latitude and longitude data received from a GPS receiver in NMEA-0183 format is in units ddmm.mmmm, where dd equals degrees, *mm* equals minutes, and *.mmmm* is decimal minutes. These units are not appropriate for the distance and course calculations; they must be converted to degrees and decimal degrees, then to radians.

The first step is converting the latitude and longitude data from the form ddmm.mmmm to dd.dddd. This is a straightforward algorithm, but it still takes a substantial amount of code. To determine the algorithm, first separate and save dd from the incoming latitude and longitude string. Then, divide mm.mmmm by 60, resulting in an exponent of zero and a new mantissa, 0.dddd. Third, add the saved dd to the result, yielding dd.dddd.

After you finish converting both latitude and longitude, radian conversion is possible. The formula to convert from dd.mmmm to radians is:

$$\text{radians} = \frac{\text{dd.dddd}}{57.3}$$

After performing the dis-

tance and bearing calculations, the data needs to be converted back to dd.mm.mmmm. This is done by following the formula degrees = radians × 57.2957795. To convert back to the form ddmm.mmmm, save dd, multiply .dddd by 60, and add the exponent to the result, yielding mm.mmmm. Then concatenate the saved dd, resulting in ddmm.mmmm.

That takes care of the latitude and longitude conversions. Now, you can tackle the knots and nautical miles (NM) conversions. All speed and distance data contained in NMEA messages is in terms of knots and nautical miles. One NM corresponds to the traversal of 1 s of arc. One knot is 1 NMph. So, if you're traveling at five knots (5 NMph), you'll traverse 5 s of arc in 1 h.

Now, convert knots to miles per hour and nautical miles to statute (land) miles (see Table 2). Remember that the output of the navigation calculations is in radians. The conversion to nautical miles is NM = radians × 3437.7387. Next, you can convert nautical miles to land miles (MI) using MI = NM × 1.150779. Converting from land miles to feet (FT), the formula is FT = MI × 5280.

NAVIGATION FORMULAS

Now that the units are all in line, the latitude and longitude data points can be run through the great circle algorithms, yielding correct results. The distance calculation is performed first because the distance is a factor in the bearing calculation. To compute the great circle distance between two pairs of latitudes and longitudes, use:

$$d = \operatorname{acos}(\sin(Lat1) \times \sin(Lat2) + \cos(Lat1) \times \cos(Lat2) \times \cos(Lon1 - Lon2))$$

This formula accurately yields the distance between two points on the globe. Remember that the units are in radians, so to convert from radians to nautical miles, use the formula NM = radians × 3437.7387. Then you can convert to land miles or kilometers. Some languages and programming environments, such as Visual Basic, do not support a direct `acos( )` function. Instead, you can use an `atan( )` function coupled with the relation acos(x) = atan(sqrt(1 – x^2) / x). For calculating distance, I use the sequence of temporary variables as follows:

Figure 2—*This is the NMEA sentence format.*

```
$<Address>,<Data>*<Checksum><CR><LF>
<Address> = <Talker><Sentence_formatter>
<Talker> = GP
<Sentence_formatter> = one of RMC GGA GSV…ZDA
```

$$t1 = \sin(Lat1) \times \sin(Lat2);$$
$$t2 = \cos(Lat1) \times \cos(Lat2);$$
$$t3 = \cos(Lon1 - Lon2);$$
$$t4 = t2 \times t3;$$
$$t5 = t1 + t4;$$
$$rad_dist = \operatorname{atan}\left(\frac{-t5}{\sqrt{(-t5 \times t + 1)}}\right) + 2 \times \operatorname{atan}(1)$$

This sequence works well. While taking a few more steps than one monolithic formula, intermediate variables are exposed, allowing you to debug the distance algorithm as it progresses. And, this sequence works with all programming languages. To prove it, t1 through t5 can be consolidated, but sometimes it's good to see what's going on in a mathematical algorithm at different steps.

Why not use the Pythagorean Theorem (remember, $d\sqrt{x^2 + y^2}$) to compute the distance between two points? For navigation, *x* would be the absolute value of the difference of the latitudes, and *y* would be the absolute value of the difference of the longitudes. This is true for the proximity of 300″ but rapidly deteriorates beyond that.

Hence, the Pythagorean Theorem is useful only in two-dimensional space. You're navigating in three-dimensional space, so for short distances, the theorem appears to work, but it fails for long distances. So, although it's an easier formula touse, you can't use it for any significant distances.

Now that distance is calculated, the next thing to do is calculate the bearing from one point to another. Bearing tells you which way to go. It is defined as the angle measured horizontally from north to the current direction of travel. North can be true north or magnetic north. Again, the great circle distance (d) between two points must be previously calculated. The classic bearing formula is:

$$c = \operatorname{acos}\left(\frac{\sin(Lat2) - \sin(Lat1) \times \cos(d)}{\cos(Lat1) \times \sin(d)}\right)$$

where *d* equals the great circle distance. The result *(c)* must be qualified by testing whether or not sin(Lon2 – Lon1) is negative. If negative, the true course is determined by 360° – c. You

Listing 2—*This code transfers an NMEA message from a raw input queue to a message processing buffer.*

```
unsigned char *locptr;      // local buffer pointer
int i = 0;                  // array index
/***************************************
* Transfer InputQueue data to InputMsgBuff[]
***************************************/
locptr = InputQueue;
/***************************************
* Check for NMEA message start character '$'.
* If '$' is found, transfer message to InputMsgBuf.
****************************************/
if (*locptr == '$')
{
   while (*locptr != '\0')
    {
         InputMsgBuff[i++] = *locptr;
     }
  InputMsgBuff[i] = '\0';
}
```

will end up at the destination, but you'll be taking the long way around the globe. Again, I like to break down the algorithm into discrete steps using temporary variables (see Listing 3).

To create a direction pointer, subtract the current GPS heading of the RMC message from the calculated bearing. Add 360° if the result is negative, creating an angle value that points from one waypoint to another.

Many different sources are available to determine waypoints. Inexpensive PC-based mapping programs provide methods of converting map points to latitude and longitude. Converting from an address to a latitude and longitude value is called geocoding. Converting from latitude and longitude values to an address is called reverse geocoding. Using the algorithms provided here and a GPS receiver, you can create your own waypoint-capturing program. Simply provide some code that will save the incoming RMC message when you pass over a location. The saved message contains the latitude and longitude of the point passed over. You can use a collection of these values to create accurate maneuver lists for roads, trails, rivers, and lakes.

Figure 4 illustrates the main components of a simple navigation system. Data is input from a GPS receiver serially to an input buffer at 4800 bps, 8 data bits, 1 stop bit, and no parity. The data is input periodically at 1-s intervals. During the time between the input data bursts (typically 200 to 800 ms), the input buffer data is transferred to a parser buffer. The data in the parser buffer is used as input to the NMEA parser that separates the data into different components—latitude, speed, and so forth.

RMC message

$GPRMC,nnmmss.ss,A,IIII.II,a,yyyy.yy,a,x.x,x.x,xxxxxx,x.x,a*hh<CR><LF>

Field	Definition
$GPRMC	Address field
hhmmss.ss	UTC of position fix (hours, minutes, seconds, decimal seconds)
A	GPS status: A means data is valid, V means data is invalid
LlII.ll	Latitude
a	North/south
yyyy.yy	Longitude
a	East/west
x.x	Speed over ground in knots
x.x	Course over ground, degrees true
xxxxxx	Date: ddmmyy (day, month, year)
x.x	Magnetic variation
a	East/west
*hh<CR><LF>	Checksum

Figure 3—*Here's the NMEA RMC message format, followed by definitions.*

Data from the NMEA parser is made available to the display and navigation engines. The navigation engine computes the distance, bearing, and direction pointer, then gives the information to the display engine for rendering and display.

The companion program to this article, navcalc.c, takes latitude and longitude pairs from the command line and computes the great circle distance from the first latitude/longitude pair to the last pair. The source and destination latitude and longitude values supplied in the program text are for southeastern Michigan. To create a dynamic navigation program based on this code, use the latitude and longitude received and parsed from a GPS receiver as the source coordinates, and continuously calculate the distance and bearing to the destination coordinates. Try different latitude/longitude source and destination pairs in your city and compare how well the output values match reality.

Listing 3—*There's no doubt this can be easily optimized, but the algorithm is broken up to be more illustrative and instructional than optimal.*

```
if (sin (Lon2 - Lon1) < 0.0)
{
   t1 = sin(Lat2) - sin(Lat1) * cos(rad_dist);
   t2 = cos(Lat1) * sin(rad_dist);
  t3 = t1 / t2;
   t4 = atan(-t3 / sqrt(-t3 * t3 + 1)) + 2 * atan(1);
    rad_bearing = t4;
}
else
{
  t1 = sin(Lat2) - sin(Lat1) * cos(rad_dist);
  t2 = cos(Lat1) * sin(rad_dist);
  t3 = t1 / t2;
  t4 = -t3 * t3 + 1;
  t5 = 2 * 3.14 - (atan(-t3 / sqrt(-t3 * t3 + 1)) + 2 * atan(1);
   rad_bearing = t5;
}
```

PARTING COMMENTS

The C code provided supplies the basic building blocks for a small, low-cost yet significant navigation application program. GPS receivers are available on the 'Net for bargain basement prices. Mapping programs that provide latitude and longitude data are widely available. The C code supplied in the example program is portable, so it runs on most processors that support floating-point operations and trigonometry functions.

Now you're on your way to creating your own navigation program. Being lost will be a thing of the past!

Jeff Stefan is an engineer at OnStar. He holds a B.S. in Computer Science and has worked in embedded systems software design for many years. Jeff is the author of more than a dozen technical articles and currently is working on his first book. You may reach him at jmstefan@mindspring.com.

SOURCE

NMEA Specification
National Marine Electronics Association
(919) 638-2626
Fax: (919) 638-4885
www.nmea.org

© Circuit Cellar, The Magazine for Computer Applications. Reprinted with permission. For subscription information call (860) 875-2199, email subscribe@circuitcellar.com or on our web site at www.circuitcellar.com.