Global Positioning System and Global Navigation Satellite System Arduino Interface Software, Version 1.1

By R. G. Sparber

Scope

Most Global Positioning System (GPS) and Global Navigation Satellite System (GNSS) receivers can output the same format of data: NEMA 0183 "sentences." You can get an overview at https://en.wikipedia.org/wiki/NMEA_0183. These sentences are human-readable, but I found them devilishly hard to read with software.

I chose to write interface software that provided a machine-readable output that contains time and position information.

Software Approach

I did not try to be creative in my coding but did try to cover all possible fault cases. The code contains expansive names for both variables and subroutines. Most lines in the source file have comments which try to explain what is going on.

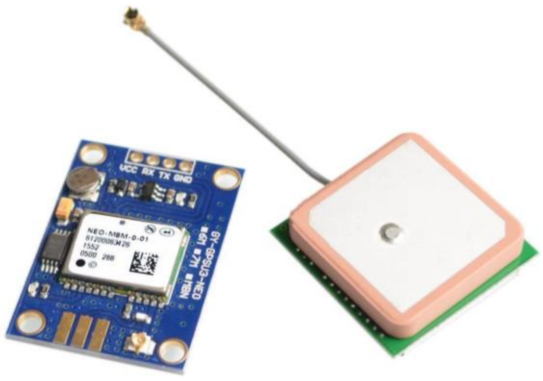
I designed the software to run on an Arduino Compatible from Sparkfun called the Pro Micro. Given that the software only depends on having access to a UART, I expect that it to run on many members of the Arduino family. The code is self-contained and does not need any header files. It occupies about 6K of program store plus 350 bytes of dynamic memory.

You can find the code at https://rick.sparber.org/GNSS.txt. Download it and change the extension from `.txt` to `.ino`.

[1] This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Contents

Hardware Overview



Devices are available for surprisingly little money. A simple Global Positioning System (GPS)[2] receiver, including antenna, can be bought for around $4, including shipping from China. They use the USA's constellation of navigation satellites and is typically accurate to ±10 meters.

For a lot more money, you can buy a GNSS device that uses all of the USA's navigation satellites plus ones from other countries. The more navigation satellites involved, the better the accuracy. For example[3], the SparkFun GPS-RTK2 Board - ZED-F9P costs $220 and is accurate to within a few millimeters.

Since GPS and GNSS devices are available with the same hardware interface and the same output data format, I will refer to them both as GNSS.

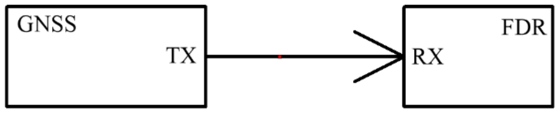
Software Overview

Each time the user calls the `GlobalNavigationSatelliteSystem()` subroutine, it updates the 16-byte array `NavigationDataByte[]` with time and location data.

Hardware Details

The GNSS has three connections of interest to us:

- VCC – connect to +3.3V or +5V depending on the device
- TX – data comes **out** of the GNSS board
- GND – connects to ground



The GNSS's TX node connects to the Arduino's RX node.

[2] GNSS refers to a category of receivers. Global Positioning System is one of them.
[3] As of 3/12/2020.

Software Details

General Structure of Arduino Code

I assume you have an understanding of how to program an Arduino. The major sections of the code are:

- Definition of all global variables.
- The `setup()` subroutine.
- The `loop()` subroutine.
- All subroutine definitions.

Global Variables

I take full advantage of the generous limit on variable name lengths. For example

```
GNSS_WaitingForDataTimeLimitMsULong
```

contains information on what the variable represents, any units, and the data format:

`GNSS Waiting For Data Time Limit` – gives you a hint to its function.

`Ms` – the time limit is in milliseconds

`ULong` – this is an unsigned long which is good to know to avoid bugs related to mixing data types

Setup()

Here is where I set up the serial link from the GNSS module. For the Pro Micro, it is `Serial1` while the USB ties to `Serial`.

Page 7 gives a few details on the parameters.

I also initialize the output array so all elements are equal to 205. This value tells the user that the array has not been updated with GNSS data yet.

Loop()

Here is where you call the GNSS subroutine. It also contains `GNSS_Timing()`, which prevents my software from hanging up if the hardware fails to respond in a timely fashion.

The Output Array

The `NavigationDataByte[]` array contains all GNSS data in a software readable[4] format:

Byte	**description**
0	hours Coordinated Universal Time (UTC)
1	minutes (UTC)
2	seconds (UTC)
3	degrees (latitude)
4	minutes (latitude)
5	seconds (latitude)
6	0 for north, 1 for south
7	degrees (longitude)
8	minutes (longitude)
9	seconds (longitude)
10	0 for east, 1 for west
11	altitude MSB (byte 3)
12	altitude (byte 2)
13	altitude (byte 1)
14	altitude LSB (byte 0)
15	units: 0 for meters, 1 for feet

Note that all parameters are single bytes except altitude, which is four bytes.

If you view the raw bytes, they are in hexadecimal. If sent to a terminal emulator via a print statement, they display as their decimal equivalent.

The user can look at the UTC to determine if they have the newest data.

The altitude is represented by four bytes that must be combined to form an unsigned long. Your software can directly obtain this number by accessing the variable AltitudeULong. It is updated every time the array is updated. The satellite system sets the units associated with altitude. If using GPS, it is in feet.

[4] Would you rather deal with the mess shown on page 5?

Error Detection

I populate elements of the output array with values that cannot have come from the GNSS.

`InitializeNavigationDataByteArray()` initializes all elements of the array to 205 as part of `setup()`. If you see this value in the array, it means you have not called `GlobalNavigationSatelliteSystem()` yet.

Any GNSS data that is null has the corresponding array element set to **200**. For example, at start-up, the GNSS hardware might not have yet determined latitude, so would leave these parameters blank. The `GlobalNavigationSatelliteSystem()` subroutine would then set bytes 3, 4, and 5 to 200.

If hours = **201**, it means we timed out waiting for data from the GNSS and would likely be a hardware problem and not a satellite availability issue.

If Hours = **203** it means we have failed to get a response from the GNSS too many times[5] since power-up, so gives up trying. The root cause is likely a hardware failure.

After you have processed `NavigationDataByte[]`, it would be a good idea to initialize all elements to a unique value higher than 205 but less than 256. Such a value alerts you to the fact that the subroutine has not updated the array.

[5] See page 7 for details.

Software Interface to Hardware

I have configured the UART for a data rate of 19,200 baud. It may be necessary to change settings on the receiver to match this rate or to change the rate in the software.

Looking at the software, you see

```
Serial1.begin(19200); //set up a communications path
between Arduino and GPS; Default of 8 data bits, 1 stop
bit, no parity or flow control is what the GPS needs.
```

Change the number, recompile the code, and download it to your Arduino.

Software Design Details (One step above the code)

You do not need to know any of the following to obtain GNSS data. I have included it in case you wish to understand how the GNSS subroutine works.

NMEA Sentences

After satellite lock, the data flowing out of the GNSS device looks like this:

```
$GPVTG,,T,,M,0.049,N,0.090,K,A*27
$GPGGA,191415.00,3317.56663,N,11205.05235,W,1,07,1.39,367.0,M,-28.1,M,,*6F
$GPGSA,A,3,29,20,05,21,12,13,15,,,,,,3.28,1.39,2.97*06
$GPGSV,3,1,10,05,44,046,30,12,19,176,22,13,21,110,41,15,23,150,44*70
$GPGSV,3,2,10,20,14,222,31,21,29,282,31,26,13,316,18,29,69,350,27*70
$GPGSV,3,3,10,46,47,209,37,51,51,171,40*75
$GPGLL,3317.56663,N,11205.05235,W,191415.00,A,A*72
$GPRMC,191416.00,A,3317.56662,N,11205.05242,W,0.010,,220819,,,A*68
$GPVTG,,T,,M,0.010,N,0.019,K,A*2A
$GPGGA,191416.00,3317.56662,N,11205.05242,W,1,07,1.39,367.3,M,-28.1,M,,*6E
$GPGSA,A,3,29,20,05,21,12,13,15,,,,,,3.28,1.39,2.97*06
$GPGSV,3,1,11,05,44,046,30,12,19,176,22,13,21,110,41,15,23,150,44*71
$GPGSV,3,2,11,20,14,222,31,21,29,282,30,26,13,316,16,29,69,350,26*7F
$GPGSV,3,3,11,30,,,26,46,47,209,37,51,51,171,40*73
$GPGLL,3317.56662,N,11205.05242,W,191416.00,A,A*70
$GPRMC,191417.00,A,3317.56662,N,11205.05249,W,0.031,,220819,,,A*61
```

These are called NMEA sentences and contain much information[6]. The sentence we need that contains time, latitude, longitude, and altitude is the one starting with $GPGGA:

$GPGGA,191415.00,3317.56663,N,11205.05235,W,1,07,1.39,367.0,M,-28.1,M,,*6F

When you call the `GlobalNavigationSatelliteSystem()` subroutine, it starts to look for "$". Once found, it reads the next five characters and determines if they form "GPGGA". Only then does it start to read and store the rest of the sentence.

With the desired sentence in memory, the parsing can begin without having to worry about keeping up with the data stream.

Here is the header plus the first data field

$GPGGA,191415.00,

Note that the number has a comma at the start and end of the field. These are delimiters that used to identify the data. That is easy enough. The tricky part involves the optional number to the right of the decimal. It is my understanding that we could get

$GPGGA,191415,

or

$GPGGA,191415.xxx,

Where "x" can be any number, and there is an undetermined number of them.

This variation complicates the code because I must move across the field until I reach a decimal or a comma. Then go back and figure out the value.

It is also possible to have a partially filled out sentence. Any null field has two consecutive commas.

[6] Do a web search on NMEA for more information. One hit is: https://www.gpsinformation.org/dale/nmea.htm

$GPGGA,165833.00,,,,,0,00,99.99,,,,,,*6C

This sentence has the time (165833.00, which is 16:58:33.00 UTC) but no latitude, longitude, or altitude.

When the subroutine finds a null field, it puts 200 in the corresponding array byte. This value can never be a valid field value and is unlikely to be due to a hardware problem.

Preventing the Code From Hanging Up Due to a Hardware Failure

The GNSS code must periodically read a buffer that is filled by the GNSS. The filling and the reading are not synchronized, so, likely, the code must first wait an undefined number of milliseconds before the data arrives. The danger is that without oversight, the code could become stuck waiting if the hardware suddenly fails., Timers and flags prevent this condition.

The highest level of protection involves counting how many times we have tried and failed to read GNSS data from the buffer. `GNSS_RetryCounterByte` starts at 1 and is advanced each time we are about to retry. When this count exceeds `GNSS_RetryCountLimitByte`, calls to `GlobalNavigationSatelliteSystem()` are returned with Hours set to 203 to indicate that we have given up trying to read the GNSS. The count is only reset to 1 if the Arduino's power is removed and restored.

We wait up to 5 seconds for data to appear in the buffer. During this time, no other code executes. Under non-fault conditions and at power-up, the delay should be no more than 1 second.

If the code does not see data within `GNSS_WaitingForDataTimeLimitMsULong`, it will not try again for `GNSS_DelayBeforeRetryOfReadingDataMsUInt`.

At the time of this writing:

- `RetryCountLimitByte` is set to 2
- `GNSS_DelayBeforeRetryOfReadingDataMsUInt` is set to 60,000 milliseconds
- `GNSS_WaitingForDataTimeLimitMsULong` is set to 5,000 milliseconds

The above means we wait for 5 seconds and then stop looking for data. After 1 minute, we try again. After retrying twice, we give up on the GNSS.

I welcome your comments and questions.

If you want me to contact you each time I publish an article, email me with "Subscribe" in the subject line. In the body of the email, please tell me if you are interested in metalworking, software plus electronics, or both so I can put you on the best distribution list.

If you are on a list and have had enough, email me "Unsubscribe" in the subject line.

Rick Sparber
Rgsparber.ha@gmail.com
Rick.Sparber.org