

Project Final Report

Black Box

Apr. 26, 2008

Michail Turovskiy

mturovskiy@gmail.com

Project Abstract

The idea of this project is for a data logger that processes GPS information and stores them in a useable format for off-board processing later. The concept is a small board that attaches to a GPS device, which can be easily transportable. When the board is powered, it parses the output of the GPS and stores it and at a later point, the data can be analyzed and plotted on a map to display where the user has been. Other metadata can be extracted as well, such as average velocity, distance covered, etc between any 2 traversed points.

The current project is able to interface with a GPS unit, read the output as NMEA sentences, parse them until it finds the localization data, package it up in an easily-read report and sends it out via serial port. On the other side is a PC. I have written code to read that data off the serial cable and store it in a log file and then process the log file to generate a map of the route traveled using Google Maps API.

Project Status

The status of the project is as follows: the GPS interface is fully functional, as is the parsing of the GPS output and report generation. The original project called for storing the report data onboard the board by integrating some kind of memory storage and writing it out. That was not done because I did not have enough time. Instead, it outputs the reports via serial. The processing of the reports and generation of maps is done on a PC and is working as well. I did not have time to extract any more metadata except for the lat/long/alt coordinates at a given time. Originally, I have wanted to interface with an accelerometer to measure the acceleration/velocity of the device, but I realized that I can get that data from the lat/long/alt information, thought he Haversine formulae¹. I did not have time to implement that.

Project Specifications

The overall development board used in my design is the Z8 Zilog provided in class. It is running at stock-speed of 18.432MHZ. I used both UARTs for reading and writing serial data. Uart0 is used for outputting well-formed reports and for outputting all GPS outputs in debug mode. UART1 is used to read from the GPS device. Only the RX pin of UART1 needs to be connected; I have not had to send any commands to the GPS device. The board outputs reports at 4800, 8N1; meaning at 4800baud, 8data bits, 1stop bit, no parity

¹ http://en.wikipedia.org/wiki/Haversine_formula,
implementation details at <http://www.movable-type.co.uk/scripts/latlong.html>

and no hardware flow control. It could actually handle much faster communications but I made it match the connection for the GPS device to make it easier to debug and write code.

The GPS device used is EM-408, as specified in my original project proposal. It was bought from sparkfun. It has 5 pins that need to be connected: Enable, VCC, Tx, Rx, Gnd. VCC was connected to 3.3V on the development board, Gnd to the board's ground pin, Tx went to PD5 – the receive pin for uart1. Enable and Rx were tied together and pulled high at 3.3V on the board as well. The default settings that it came with proved sufficient for my use so I never had to send it any commands; it outputs valid NMEA sentences² over 4800, 8N1, so once I figured that out I was able to just plug it in and go.

The code consistent of several modules; written in C, Perl, and some JavaScript/HTML thrown in for good measure. All the code running on the development board was written in C, using the ZDS II editor, version 4.10.0. The first module written sets up the board to some useable state, configures the pins manually for serial communication with UARTs. The end-result is that the system is ready to start talking with the GPS. The majority of that code is in “gpio.c”.

The next step is to start reading the GPS device and parsing the data. Cold-start times for the EM408 is around 42secs, although in my experience it is almost always faster than that. In any case, there will be some time at the beginning when the output of the GPS is malformed³. The first thing that happens upon receipt of any character is that it is stored in a ring buffer for parsing. I've implemented a simple ring buffer for this use; it is found in “ring_buffer.c”, and is a slightly modified version of what I've used for the keyboard lab to store keystrokes. The difference is that we always add exactly 1 character at the end. It is externally synchronized; which means I assume that we will not have any interleaved calls to the ring buffer. This is a safe assumption because I do not have any interrupt routines enabled. The buffer size was set to 200 characters through trial and error. The output of the GPS includes several types of NMEA strings; we're only interested in 1 type – GGA – which gives us the lat/long/alt of the receiver. The distance between consecutive GGA strings in a well-formed NMEA stream for the EM408 is slightly more than 400 characters. And the length of a GGA string is about 75 characters. (Also, consecutive GGA strings are almost exactly 1sec apart.) I've found that with a buffer of 200, I almost never miss a GGA string due to lack of space.

“gps.c” deals with parsing the data stream and extracting the needed information from it. Once the buffer is full, an attempt to parse the valid data is triggered. I have written a small parser to read and validate NMEA GGA strings⁴. It will attempt to read from the beginning of a buffer until it can find a valid GGA string; any characters that are not a match, are removed from the buffer. In theory, that means that if we get the first ‘N’ characters in a buffer, and then the buffer is filled and processed, we will not be able to

² See “capture.txt” for a raw log of the GPS output

³ For the NMEA specifications, see <http://www.geoaps.com/NMEA.htm>, or wikipedia.

⁴ For examples of GGA strings and how to decode them, see <http://www.sparkfun.com/datasheets/GPS/NMEA%20Reference%20Manual1.pdf>

find a valid GGA string and will discard the whole buffer, and then when we receive the rest of the GGA string, we will also not be able to match it – the GGA string **must** lie on the buffer boundaries. However, I have run the program a good deal and have not observed this yet. In a worst case, if a GGA string is missed, it will always hit the next GGA string and because they are 1sec apart, we won't miss a whole lot of data. There is also validation code to make sure the GGA string is well-formed according to the NMEA specifications, and that it contains valid data.

When a valid GGA string is detected, a GpsLocation report is generated. It is represented by the following structure:

```
typedef struct {
    char UTC_time[UTC_TIME_SIZE]; //format of utc time: hhmmss.sss, all digits char
    char Lat[LAT_SIZE]; //format of latitude: ddm.mmmmm, all digits, char
    char NS_indicator; //format of north/south indicator: N or S
    char Lon[LON_SIZE]; //format of longitude: dddmm.mmmmm, all digits, char
    char EW_indicator; //format of east/west indicator: E or W
    Position_Fix_Indicator fix_indicator; //is fix valid?
    int num_sats; //number of satellites used to get fix
}GpsLocation;
```

This is what gets output via serial over UART0. This completes the main body loop of the board; it is reading the GPS output, forming the reports and sending them out.

The next part of the design is written in Perl. I've created a script to read the data off the serial cable and save it to "logfile.txt". This is instead of the onboard memory storage that I was not able to implement. Another Perl script parses the file created previously and outputs valid HTML/JavaScript page that will generate a Google Map such that every report in the file is represented on the map by a pushpin, which if clicked, will display the UTC time when this location was recorded. There really is not a whole lot to this code; it's mainly a template for JavaScript code generation that exercises the Google Maps API. I've needed to apply for and use the Google Maps license key.

Project Implementation

The pin connections diagram is enclosed below.

The software process workflow for the Z8 development board is described below:

- >init gpio pins and prep system
- >main processing loop:
 - >read a character from GPS
 - >store character in ring buffer
 - > if no more space in buffer:
 - >parse buffer to read GGA strings
 - >if found GGA string:
 - >fill out GpsReport structure
 - >if not found, remove characters from buffer
 - >send report via serial over UART0

The software process workflow for the PC code is described below:

- >open COM port 1
- >main processing loop:
 - >read data from COM1
 - >output data to "logfile.txt"

The software process workflow for creating the Google Maps display is described below:

- >open "logfile.txt"
- >create header html elements
- >generate header javascript code to init Google Maps api
- >for every GpsReport in "logfile.txt" do:
 - >create a point of interest on the map
 - >create onclick() event handler that will display UTC time of data capture
- >generate javascript code to center map on the first GpsReport in "logfile.txt"
- >generate javascript/html footers
- >save resulting document as "route_map_XXX.html" where XXX is timestamp of first GpsReport in "logfile.txt"

I have started out trying to get the GPS working at least a month ahead of time. Since they were pretty expensive I only bought 1. Thinking its only got 5 pins to connect, how could I **possible** screw that up? The first problem I encountered is that there are 5 wires, 1 grey the rest white. However, they're not labeled. Reading the pinout diagram did not say anything about its orientation. I assumed the grey one was for power. I was wrong and accidentally connected VCC to GND and vice versa. Next 2 weeks were spent trying to debug the device. It was outputting a signal at almost the right voltage and what looked like semi-valid data on the o-scope. However, it was not ascii characters. I've tried to vary the baud rates and start/stop bits. The manual did not specify what the connection settings were. It seemed to suggest 57600bps was the default. In online forums, I've

found that the default seems to be 4800, 8N1. The manual could not have been less clear about that particular factotum.

I could not get valid data out of the device so I ended up ordering 2 more of the same model. Another problem I immediately encountered is that the connectors used are extremely small and I could not solder them. I had to strip the wire and hand-solder them to a couple of jumper cables. The problem there was that I was not so good at soldering then. I got plenty of practice though! The wires were very thin and if heated for a long time, they tended to break. That was probably the biggest issue for me. The 2 new devices worked great but the connection would cut out a lot and I would start seeing non-ascii characters on the input. That could be resolved with either re-soldering or jiggling the wires. The wires were extremely noisy. I tried wrapping them in some electrical tape and that improved it somewhat but they could not be reliably be used. In fact, moving the device anywhere would almost always trigger a period of noise on the wires. Some of the connectors I've re-soldered 3 times. I suspect that the connectors they use are just not very good.

That is what happened in class during the presentation; the connection was noisy and was not 100% working.

By the time I finally was able to talk to the GPS, albeit unreliably, I did not have time to do anything about on-board storage so I decided to just dump the reports via the serial. That worked out great for me; reading data off the serial cable with Perl turned out to be easier than I hoped.

Finally, for offline processing I've always wanted to play with Google Maps api and this project gave me the excuse to do that. It was very easy to use and worked correctly pretty much the first time through. For my needs, I got everything I needed with 3 api commands: initialize the api interface with my api key, initialize the map and place a marker at a given location.

One problem that took some time is that the output of the GPS for GGA data is in DDMM.MMMM and DDDMM.MMMM whereas the coordinates that the Google Maps api wanted was in DD.MMSS format. However, that format difference was not mentioned anywhere and I was only able to figure out after searching online. In particular, if the coordinates are entered manually in Google Maps using the DDMM.MMMM format, they're accepted and converted to the DD.MMSS format internally. I had to do that conversion prior to calling the api myself. (It's actually trivial once I realize what was needed: divide by 100, multiply by 60).

Verification was done at each step by observing the outputs and making sure they matched. In debug mode, Z8 outputs the raw GPS stream without actually parsing it, to make sure that the connections are working and we're getting ascii character data. Those raw GPS coordinates can be entered in Google Maps and manually verified to match the actual location. Photos of the final assembly are included with my code.

Project Retrospective

The most important design decision I had to make was change the scope of the project to exclude the on-board data storage. That simplified things a lot and made it possible for me to finish. Another thing I found out was that I did not need an accelerometer or any other sensor besides the GPS; I could extract everything I wanted from the data stream. However, I did not have time to do any of that. That is something I can expand on in the future.

I have learned a great deal about hardware throughout the course of this project. I needed to get pretty good at soldering to get the wires hooked up. Also, hardware debugging techniques such as using the o-scope in class; and checking the voltages coming into the device and going out, to make sure they match the data sheet. I have learned somewhat about serial communications and parsing data streams in real-time. I expected to have timing conflicts with the GPS outputting data faster than I could read/process it but that never happened as far as I could tell. I was particularly worried about using the ring buffer and how much more time it takes using that abstraction than a simple array with a head/tail indexes. In fact, the Z8 kept up great!

In retrospect I would probably pick a different project. The GPS I used is not in any way suitable for moving applications; the wires are way too noisy and finicky. Most of the other GPS modules I've looked at used similar connectors so maybe its just that I'm not good at connecting them? In any case, even if I had a AC->DC adapter and could take the board out for a walk, it would most likely not work!

Project Enclosures

The following are enclosed:

- *my source files for the Z8 platform
- *my Perl script and example files it produces
- *sample outputs from the Z8 for use in debugging
- *sample html pages generated by my Perl scripts from the sample data
- *a picture of my setup
- *NMEA reference manual and the data sheet for the GPS⁵

⁵ http://www.usglobalsat.com/download/47/em408_ug.pdf for data sheet,
<http://www.sparkfun.com/datasheets/GPS/NMEA%20Reference%20Manual1.pdf>, for NMEA data

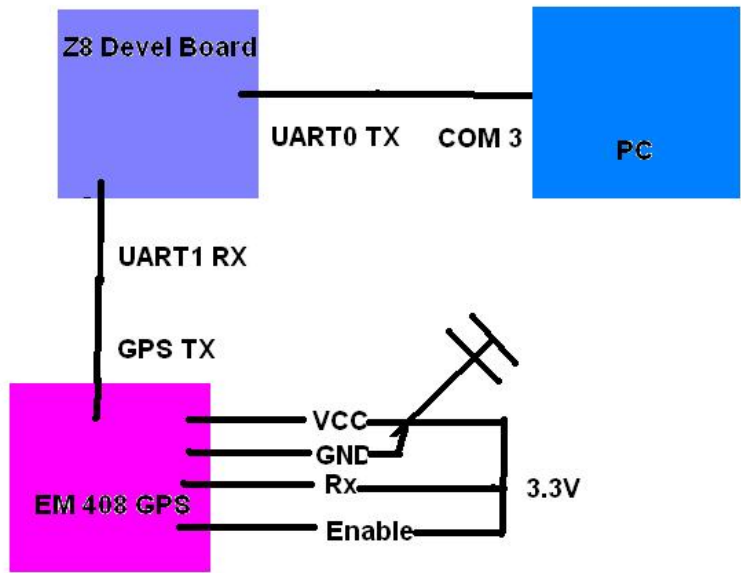


Diagram 1: pin connections diagram