Synapse
Unmanned Autonomous Vehicle

Dariel Marlow darielmarlow@hotmail.com
Michael DeLisi delisi@eng.utah.edu
Toren Monson nicmonson@gmail.com
Matt Stoker matt.stoker@gmail.com

http://www.cs.utah.edu/~delisi/cs3992/

Table of Contents

Abstract 3
Introduction 3
Pocket PC Application 4
GPS 4
Serial 5
GPS Course Selection 6
Turning Calculations 6
Sonar Devices 7
Microcontroller Interfaces 7
UAV Analog Interface 7
Sonar Interface 8
Pocket PC Interface 8
Base Station 8
Risk Assessment Reviewed 10
Conclusion 10
Appendix 12
Figure 1 12
Figure 2 12
Figure 3 13
Figure 4 13
Figure 5 14
Bibliography 17

Abstract

The Synapse UAV is a GPS guided remote controlled car that houses a Pocket PC, GPS unit, and micro-controller. The vehicle follows a route, programmed by a base station laptop, and guided by a GPS. The GPS unit, connected to the Pocket PC, constantly receives data from satellites and uses this information to guide the vehicle through all of the predefined points. The Pocket PC sends commands to the micro-controller via serial, which changes the vehicle's direction to reflect the GPS data. The microcontroller interfaces directly with the vehicle, controlling its velocity and direction based on values received from the Pocket PC. The base station assigns the route to the Pocket PC and awaits detailed route information received upon vehicle's return.

Introduction

The purpose of this project was to create an autonomous remote controlled car through the utilization of several components. The main components needed to control the vehicle include the HP H4155 Pocket PC [1], the Motorola MC9S12C32 microcontroller [2], and the GlobalSat SD-502 SDIO GPS [3]. The Pocket PC connects to the GPS through a SD card port to receive position information. The Pocket PC processes the position information and sends direction and velocity changes to the microcontroller through an RS232 serial connection. The microcontroller parses the input and sends it out on 6 digital I/O pins. This goes through circuitry which delivers the appropriate voltages to control the vehicle. A general overview can be found in Figure 1 on page 12. The microcontroller circuit can be found in Figure 5 on page 14.

Beyond the basic GPS control, we wanted to add functionality that allowed for greater visualization and obstacle avoidance. Because this is an unmanned vehicle, the camera provides the operators information on how the vehicle did on its mission aside from simple points on a map. The video information can be recorded on the base station laptop for future review. GPS provides a location and how to get there, but it does not tell you what obstacles are in the path from the initial to the final position. With the addition of sonar object detection (R145-SRF08 Devantech SRF10 Range Finder [4]), the Pocket PC can control the vehicle to avoid obstacles detected by the sonar system.

Pocket PC Application

The purpose of the Pocket PC application is to interface the GPS unit with the vehicle. The GPS connects to the Pocket PC through its SD card port. On the Pocket PC, the GPS shows up as a device on COM4 from which the GPS sentences can be read. There is a second port on the bottom of the Pocket PC which allows the device to sync to a computer using either USB or RS232 serial. The USB interface is used to establish a connection to the base station to load software and maps onto the device. When the serial cable is plugged in, it shows up on the Pocket PC as COM1 which can be read and written to in our Pocket PC's C# code. A picture of the Pocket PC in action can be found in Figure 2 on page 12.

GPS

The GPS data that is being constantly received is read from a COM port on the Pocket PC. The data comes in the format of NMEA sentences which contain the current longitude and latitude. We use GeoFrameworks [5] GPS libraries to parse this data and generate events when there is new data available. GeoFrameworks is a development framework for .NET languages which abstracts the complexity of reading, parsing, and interpreting the GPS data; it also presents the new data by firing events (i.e. executing methods).

When a new position is received from the GPS, an event is fired which calls the PositionChanged method. This method stores the new position's longitude and latitude in a list and calculates a new current position through an averaging algorithm; this is done to compensate for the inaccurate data received from the GPS unit. This is taken to be the current position of the vehicle. For each new position received, a line is written into a file which contains the route that the vehicle took. The file is stored in Google Earth's KML format. Once the trip is complete, the programmed and saved routes can be compared by opening the two files in Google Earth on the base station computer.

There were several problems that we ran into when working with the GPS. First, the computed bearing is not accurate unless you've been going straight for 5 or 6 seconds. Our original algorithm was to read the GPS bearings received, and then turn until the vehicle is headed straight toward the point. The issue was that once the vehicle turned, it used the GPS bearing a second later. So, the bearing is no longer accurate because the vehicle didn't go straight for long enough for the bearing to get updated. Second, at this time of the year there are lots of days with cloudy skies, and rain/snow storms. We found that these bad weather conditions result in GPS reads that are twenty to thirty meters away from the actual position.

Serial

The serial connection to the microcontroller, which we were concerned about last semester, actually turned out to be very simple. We acquired a cable that plugged into the bottom of the Pocket PC and had a serial connection at the other end. This showed up on the Pocket PC simply as COM1. We then used the CFSerialClass [6], a .NET library for communicating via the serial port, to send commands from the Pocket PC to the microcontroller. It then was simply a matter of setting the baud rate to 9600 bps along with other settings to match what the microcontroller was set to.

The serial class calls a specified method whenever new data is available by firing an event. This method parses the input from the microcontroller and sets the appropriate state. There are three commands that can be sent by the microcontroller to the Pocket PC. The first command, E, is an echo reply. When the "Start Serial" button is pressed, it sends an E to the microcontroller which, in return, sends an E back to acknowledge. So, when the Pocket PC receives an E, it knows the serial connection is established and sets the serial portion to the ready state. The other two commands are from the sonar devices: F<number> and R<number>. These are the distances the front and right sonar devices are reading in inches, respectively. The <number> portion of the command is a one byte value that represents the distance in inches and ranges from 0 to 255. These distances are used in order to provide collision avoidance to the vehicle.

Furthermore, there are three commands that the Pocket PC can send to the microcontroller. The first command, E, is an echo request discussed above. The next command, V<number>, is used to request that the microcontroller change the velocity of the vehicle. The <number> portion of the command can be an ASCII number from 0 to 7; ASCII proved convenient when testing serial commands with a keyboard. Sending a 0 will stop the vehicle while sending a 7 will make the vehicle go at its maximum velocity. The last command, D<number>, specifies how the wheels should turn. The number can once again be between 0 and 7. Originally, the vehicle was thought to be able to turn in small increments; unfortunately, it was not capable of doing this. Due to the vehicle's circuit, the commands are less intuitive that one might hope for; sending a 0 will make the vehicle turn right, sending a 3 will make it go straight, and sending a 7 will make it go left. The direction of the vehicle's wheels will stay in the position of the last command received. So, if the vehicle receives a turn command it must receive a straight command after some time. Otherwise, it will go in circles. In the vehicle's circuit there is not enough power to go from left to center, so the program first tells the wheels to turn right, waits 100ms, then tells the wheels to turn to the center.

GPS Course Selection

The aforementioned course is selected using Google Earth on the base station computer. The path is saved to a KML file, which uses XML to store the points in longitude and latitude format. This is the same file format that the Pocket PC application writes for the actual route taken. This file is uploaded to the Pocket PC through a USB connection using the ActiveSync software. The Pocket PC parses this file and stores these points as the points the vehicle needs to reach, also known as the points of interest (POI). Each time a new position is received from the GPS, the distance from the current point to the new point of interest is calculated. If it is within a certain range (five meters used for our demonstration, but the value is a modifiable constant), it advances to the next point of the course. After it arrives at the last point, it sends a signal to stop the vehicle.

Turning Calculations

Calculating how much to turn, and when, has proved to be the most difficult challenge of the entire program. In our original approach, we assumed that the vehicle would be able to turn in seven discrete increments (0 to 6 for the D command). We used GeoFrameworks to calculate the bearing from the last position received from the GPS to the POI that the vehicle is headed to. It then used the current bearing reading from the GPS to determine whether it should turn left or right, and how much it should turn (the selection from 0 to 6). Every time the GPS received a new position, it would do this same calculation. So, when the vehicle turned enough, the calculation would wind up with a 3 and make the vehicle go straight. This worked when testing it with the emulator we built because the emulated GPS coordinates and bearing information were always exactly correct.

When we hooked the Pocket PC up to the vehicle, this turned out not to work at all, making the vehicle go in circles. We realized that the bearing that the GPS gives us can't be trusted for the purpose we are using it for. So, we switched to a method of making the vehicle go straight for two seconds while storing the points, and then using linear regression to calculate our current bearing. The vehicle used this bearing to determine the direction of the turn and the length of the turn. It used the following formula: constant*degrees-to-turn / 180. Once the vehicle turned, it again waited for two seconds to calculate the new bearing, and the new amount to turn.

Although the linear regression seemed like it would work, it wound up going in wrong directions when testing. This could have been from allocating too much time turning, the bad weather conditions at the time, or not collecting data for a sufficient interval. From walking around with the GPS, we noticed that

if you walk in a straight line for around five seconds, the bearing that the GPS module gives is very accurate. We changed the program once again to go straight for five seconds after turning, then collect five bearing readings and average them. We then used this bearing to compute how much time to turn and in which direction. In our tests, this method works very well, and the vehicle finished a course successfully. We did have to manually keep the vehicle from driving on low lying snow, as the sonar devices wouldn't pick it up.

Sonar Devices

When the Pocket PC receives sonar readings from the microcontroller via serial, it checks to see if the front sonar reading is less than an "avoid distance," a constant that can be changed. When it receives a front sonar reading that is less than this distance, it slows the vehicle and tells it to turn left immediately. It will continue to turn left until it detects forward to be clear. It waits one second after the front sonar is clear, because the obstacle may be at an angle between the front and right sonar devices. In this mode, the vehicle will continue to go straight (turning is disabled) until it detects the right side of the vehicle to be clear. Once it detects the right side of the vehicle to be clear, turning is re-enabled and the original course is resumed.

Microcontroller Interfaces

The microcontroller interfaces the Pocket PC, the RC vehicle, and two sonar devices into the UAV. The microcontroller receives directions from the Pocket PC, which it feeds directly to the UAV's analog motor control system. The microcontroller gets data from the sonar devices and sends them to the Pocket PC, which uses this information to better plot the direction the GPS vehicle should go. In the subsections following, the communication protocols will be explained, as well as the varying success of their individual implementations.

UAV Analog Interface

The interface between the microcontroller and the vehicle turned out to be much harder than expected and the most troublesome part of all the interfaces. Originally, it was decided to interface the vehicle with the microcontroller via the remote control. Through two events, our design changed to interface the microcontroller directly to the vehicle. First, it was found that the motor system was electrically very simple. Second, the circuitry in the remote control that makes the vehicle go in reverse was fried. With

these issues in mind, we decided to implement the vehicle control by building our own interface between the vehicle and the microcontroller.

The interface can be subdivided into two parts: a digital to analog converter with voltage gain and a current amplifier. The voltage gain 3-bit DAC was simple to build. It is a simple discrete component DAC whose output stage includes a 2x gain using a few resistors and an Op-Amp. The Op-Amp does not provide enough current so another current amplifier stage is required. The current amplifier was implemented as a Class B push-pull BJT amplifier. It became a source of constant headache. It produced problems each time the circuit was rebuilt and modified. The bipolar transistors that fed it had to be replaced by better versions. Subsequently heat-sinks were added to them. In retrospect, this was not a complicated design, but the designer of this interface had to learn many new techniques involving circuit tolerances and configurations. In conclusion, the final design provided both the ability to turn the vehicle in varying degrees and control the vehicle at varying velocities.

Sonar Interface

The other interfaces were trivial in comparison given that they were all digital. The sonar interface took a bit to understand but documentation was plentiful and the interfaces were not hard to implement. The sonar used the I^2C interface to receive commands and return 16-bit distance data. The microcontroller did not inherently contain this interface, but two serial I/O pins were programmed to implement it. Each of the sonar devices needs a unique address, so we changed one device’s address using the protocol provided in the documentation.

Pocket PC Interface

The simplest interface was between the microcontroller and Pocket PC from the point of view of the microcontroller. The microcontroller has a built-in RS232 UART. All that was needed to use it was to set the appropriate registers and set up a buffer to store input data.

Base Station

The UAV's Pocket PC communicates several forms of information with a laptop. To facilitate this, the group produced a conglomeration of tools to provide and receive this information. Thus, the laptop and associated software were designated as the base station of the system. Software was acquired or written for communication of GPS routes, logs, and live video data.

The GPS routes that the Pocket PC uses are encoded in XML files produced by Google Earth, called KML files. These files are produced by Google Earth to store the "My Places" section of its interface, and can include location markers, routes, notes, picture references, and many other types of location-specific data. When we produced a coordinate list for the UAV to follow, we created a route in Google Earth by selecting the route tool and clicking on the points of interest the UAV should follow in order. Then, we highlighted the route in the "My Places" section, clicked File->Save Place As, and saved the file in KML format. This file was then transferred to the Pocket PC. A sample KML file can be found in Figure 4 on page 13.

Originally, the group wrote software to continuously communicate the file with the Pocket PC over Wi-Fi, including the route the UAV actually was following according to the GPS. While this software was being written, the group used Microsoft Active Sync to exchange the data at the start and end of a test. Since this system worked sufficiently well, resources were moved from the continuous communication software to other areas of the software development. If development was continued, the software could be completed rather quickly.

The UAV has a front-mounted wireless camera that transmits a NTSC video signal to a receiver. This camera-receiver package, purchased from Spy Camera Specialists, Inc. [7], was originally produced as a security camera solution. We modified the system by adding an inexpensive USB A/V video tuner from ADS Video [8]. This allowed us to watch a live 640x480 video feed from the UAV's point of view on the base station laptop.

Again, originally the group wrote software to interface with the USB video tuner that would be in one program with the file communication, but used the software that came with the tuner until the software was completed. The software that came with the tuner allowed the window to always sit on top of other windows, so it could just hover over the top of the GUI designated for the video camera, and be used effectively. Necessity asked for resources, so they were diverted from this software development. This subsystem could also be completed with a few hours work.

The base station software does provide one important tool to the team. It has a custom vector graphing package written by the team which was used to test our linear regression algorithm. When this algorithm proved functional, the code was transported to the Pocket PC code base. Both were written in C# with the .NET framework, so the code could run with no modifications on the Pocket PC. However, the GPS data was used directly on the Pocket PC instead of being converted to the PointD data type, a

double precision floating point tuple. So the linear regression algorithm was modified to use the GPS coordinates from the GPS directly.

Risk Assessment Reviewed

In our proposal, the assessed risks included: interfacing the Pocket PC and microcontroller with serial, interfacing the vehicle and microcontroller with circuitry, insufficient time to complete the project due to courses, and damage to the vehicle due to crashes. Of the risks we originally assessed, the greatest turned out to be the interface between the vehicle and microcontroller. Among the issues in interfacing these parts, one of them was amplifying the current necessary to power the vehicle's motors. This became a big issue because we didn't have the knowledge to design this type of circuitry. Also, with high currents some circuitry became very hot and stopped working. Finally, we needed complex voltage configurations to power the interfaces between the microcontroller and the vehicle.

In addition to the things we assessed in the proposal, there were two great risks that we didn't realize. First, while building the circuitry to interface the vehicle and the microcontroller, we did not assess the chance that we could damage the microcontroller by applying too much voltage accidently. We had to borrow a new microcontroller from a friend and order a second one from Freescale, since two microcontrollers stopped working. This issue cost us time and the ability to better test the Pocket PC and microcontroller integration. Second, we didn't realize the inaccuracies of the GPS; the civilian GPS system is only accurate within about five meters. So, we had to give a wide range of precision for reaching the points of interest, and run the vehicle in a wide open area so it could follow an imprecise path. This imprecision can be seen in Figure 3 on page 13. Also, we didn't originally realize that we couldn't trust the bearing measurement that comes from the GPS module. In order for this to be accurate, the vehicle has to be going straight for around five seconds. Had this been known before, a digital compass would have been part of our design.

Conclusion

The Synapse UAV project has been more difficult than we originally anticipated. It has stretched the abilities of all of the team members, and caused us to investigate many technologies that we would never have investigated otherwise. Our original risk assessment was incomplete in retrospect. Too many assumptions were made on the accuracy of GPS data.

If we had this project to do over again, we would have used a simple hardware and software system involving an electronic compass for bearing, GPS module for position, microcontroller for the hardware-software interface, and a simple C# controller program on the Pocket PC. Thus, much of what the group has produced would be included in the final project even if we had the opportunity to do it all over again. We make this statement with at least that feeling of success.

Appendix

Figure 1
A general overview of the project.

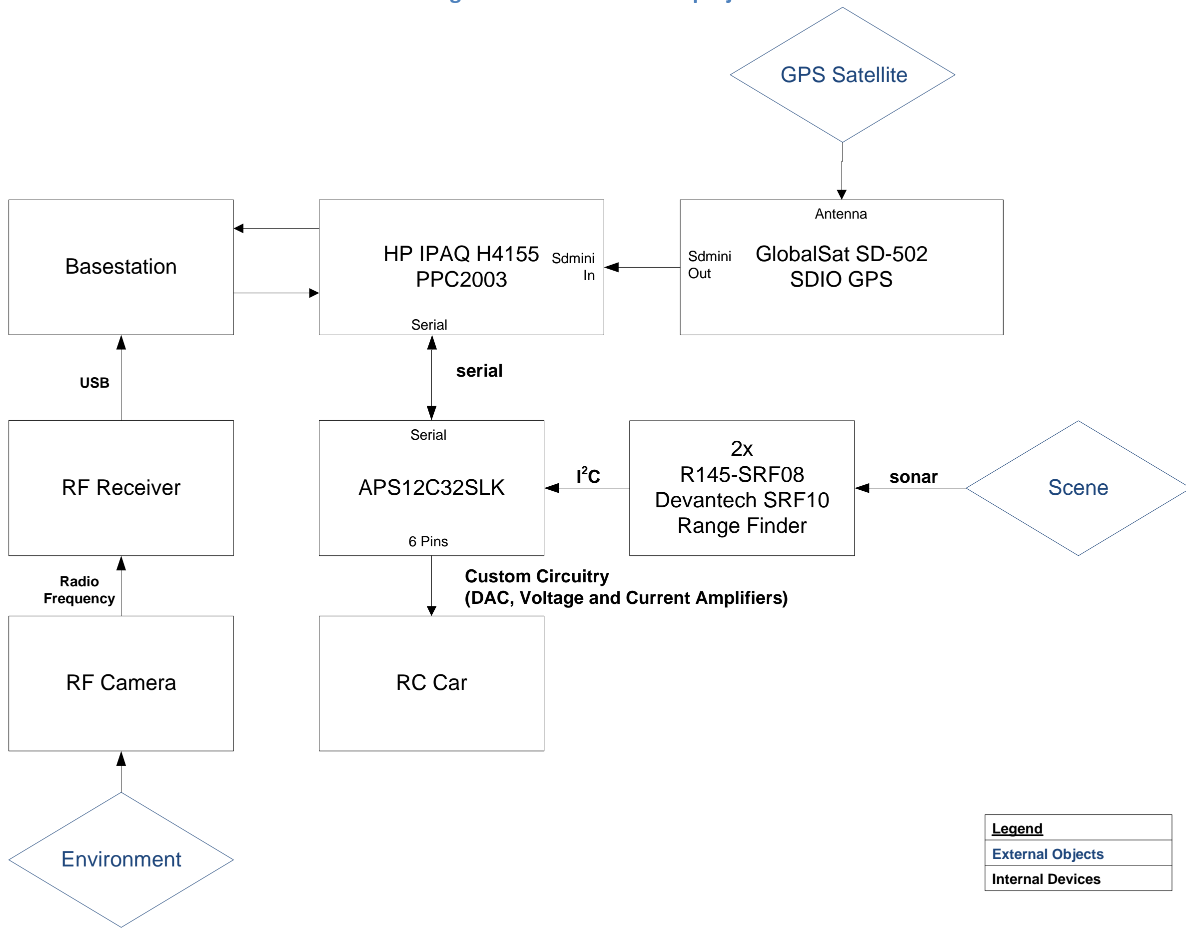


Figure 2
Pocket PC software for GPS and vehicle communication.

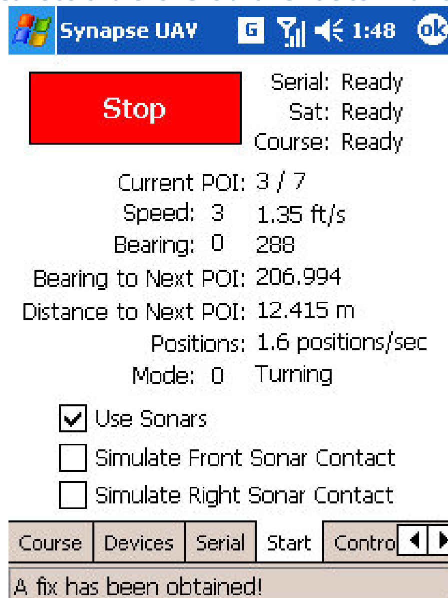


Figure 3

This is the GPS output of a successful course trial. The GPS inaccuracies are shown, since the GPS remained on the sideway at all times. Orange is the route stored by the Pocket PC unit and white is the path that was to be followed.

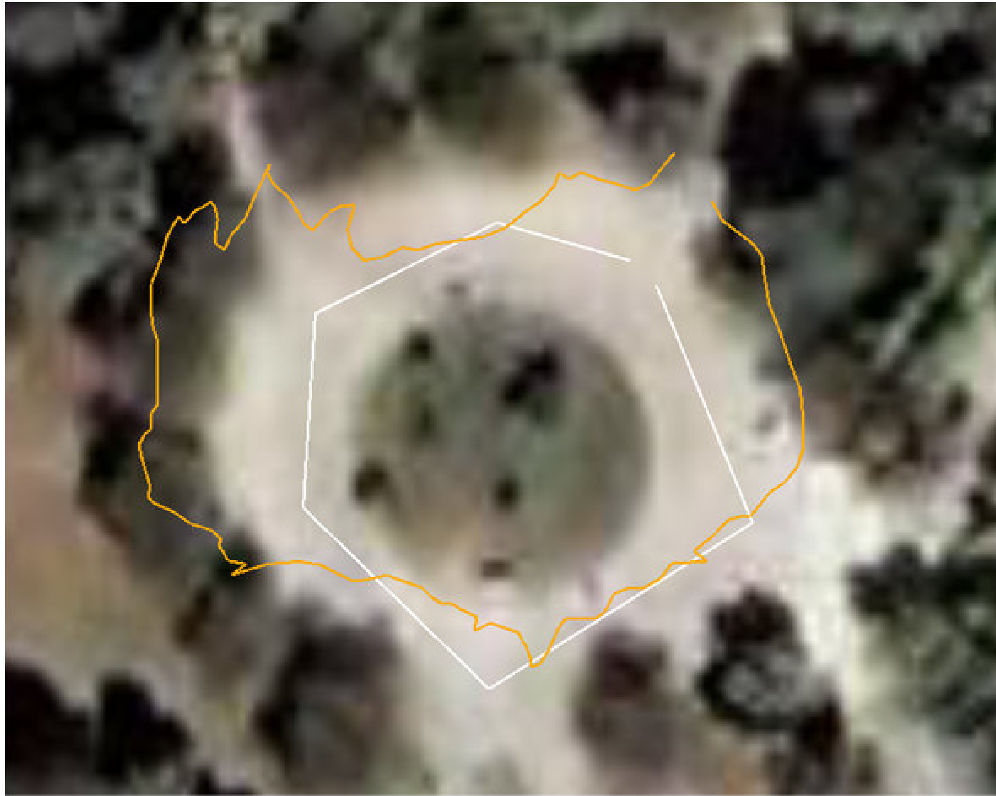


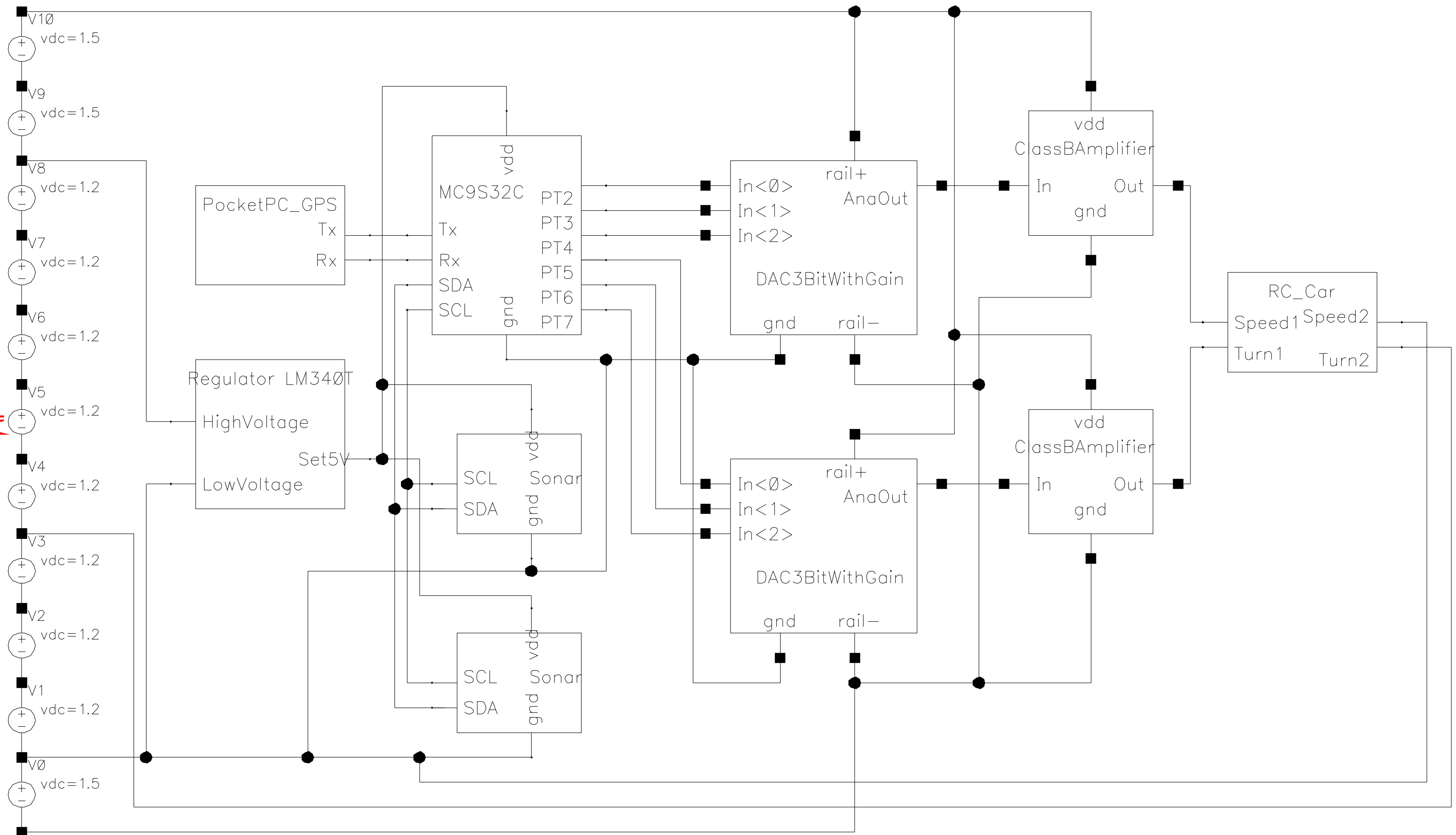
Figure 4

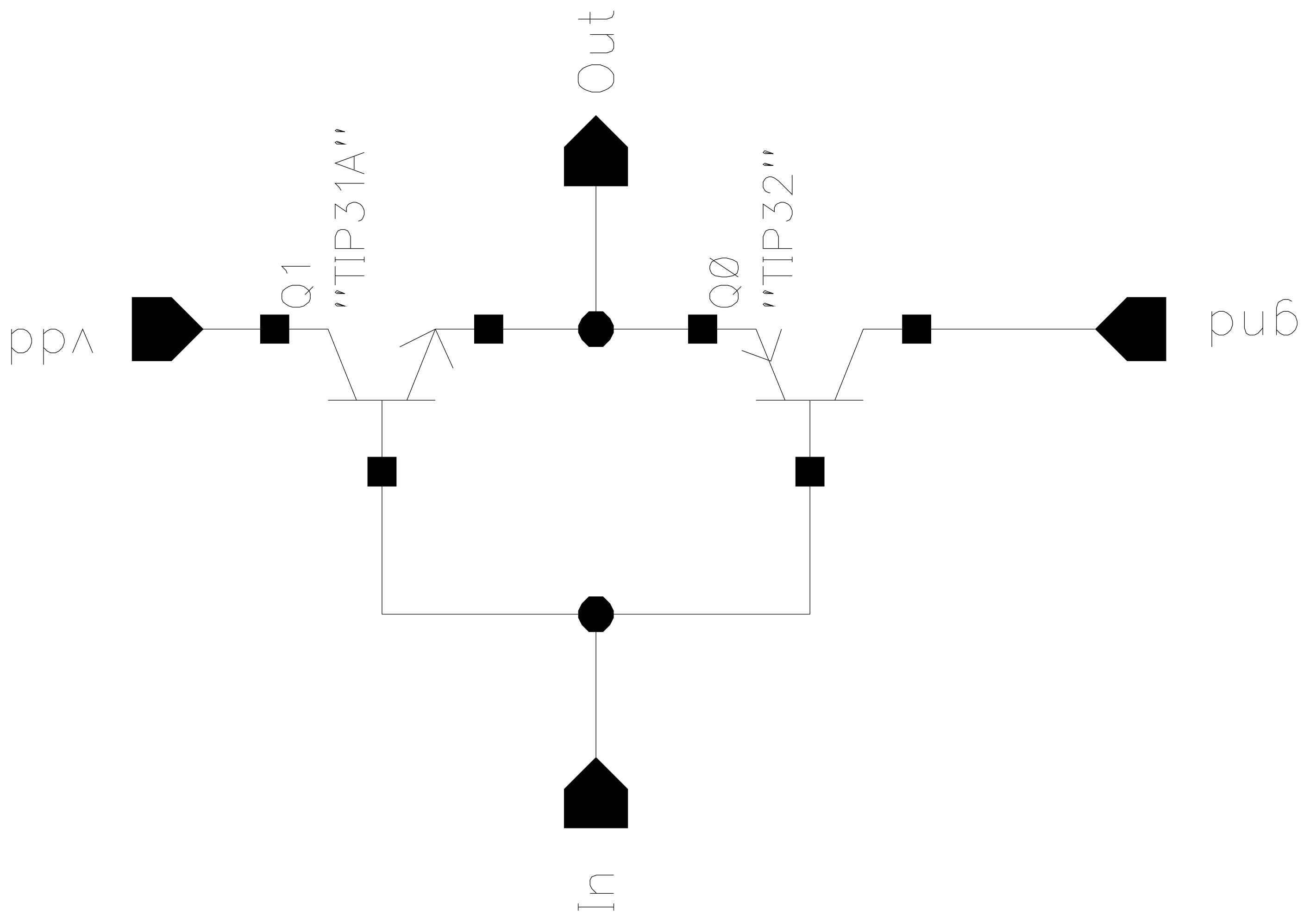
A sample KML file to illustrate the syntax.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://earth.google.com/kml/2.2">
        <Document>
                <Placemark>
                        <LineString>
                                <coordinates>
                                        -111.847904337799,40.76834424984754,0
                                        -111.8479331279533,40.7687906043826,0
                                        -111.8481655360586,40.76878142150248,0
                                        -111.8481547374151,40.76828836798575,0
                                        -111.8479497214831,40.76827106765924,0
                                </coordinates>
                        </LineString>
                </Placemark>
        </Document>
</kml>
```

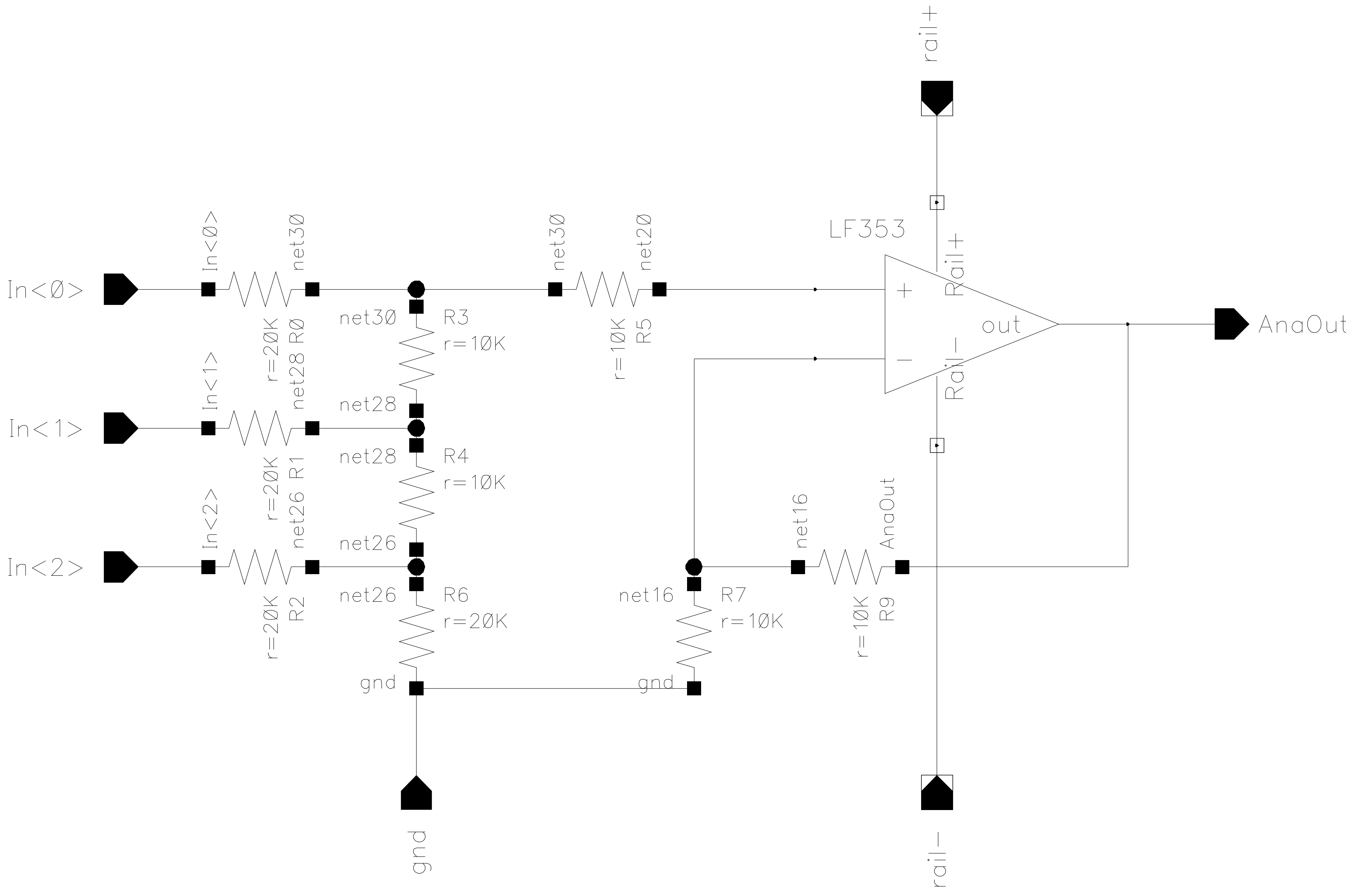
Longitude,Latitude,Height format. Height is not used.

Goes in order from first point in route to last point in route.


V10
vdc=1.5
V9
vdc=1.5
V8
vdc=1.2
V7
vdc=1.2
V6
vdc=1.2
V5
vdc=1.2
V4
vdc=1.2
V3
vdc=1.2
V2
vdc=1.2
V1
vdc=1.2
V0
vdc=1.5
PocketPC_GPS
Tx
Rx
MC9S32C
vdd
Tx
Rx
SDA
SCL
gnd
PT2
PT3
PT4
PT5
PT6
PT7
Regulator LM340T
HighVoltage
Set5V
LowVoltage
SCL
SDA
Sonar
vdd
gnd
SCL
SDA
Sonar
vdd
gnd
In<0>
In<1>
In<2>
rail+
AnaOut
DAC3BitWithGain
gnd
rail−
In<0>
In<1>
In<2>
rail+
AnaOut
DAC3BitWithGain
gnd
rail−
vdd
ClassBAmplifier
In
Out
gnd
vdd
ClassBAmplifier
In
Out
gnd
RC_Car
Speed1
Speed2
Turn1
Turn2



vdd
Q1
"TIP31A"
Out
Q0
"TIP32"
gnd
In


In<0>
In<1>
In<2>
In<0>
In<1>
In<2>
r=20K R0
r=20K R1
r=20K R2
net30
net28
net26
net30
net28
net28
net26
net26
R3
r=10K
R4
r=10K
R6
r=20K
gnd
net30
net20
r=10K R5
LF353
+
−
Rail+
Rail−
out
AnaOut
rail+
rail−
net16
R7
r=10K
gnd
net16
AnaOut
r=10K R9
gnd

Bibliography

[1]. HP H4155. [Online] HP. [Cited: April 14, 2007.] http://h10025.www1.hp.com/ewfrf/wc/product?lang=en&lc=en&cc=us&dlc=en&dest_page=product&product=425743.

[2]. Freescale MC9S12C32. [Online] Freescale. [Cited: April 14, 2007.] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MC9S12C32.

[3]. Globalsat Technology Corporation. [Online] [Cited: December 17, 2007.] http://www.globalsat.com.tw/eng/product_detail_00000053.htm.

[4]. SRF07 Ultra sonic range finder. [Online] [Cited: December 17, 2007.] http://www.robot-electronics.co.uk/htm/srf08history.shtml.

[5]. GeoFrameworks. *GIS and GPS Components for Visual Studio.NET.* [Online] [Cited: December 17, 2007.] http://www.geoframeworks.com/.

[6]. OpenNETCF Consulting, LLC > Home. [Online] 2007. [Cited: December 17, 2007.] http://www.opennetcf.com.

[7]. Spy Camera Specialists, Inc. *COINSIZE WIRELESS PINHOLE CAMERA SYSTEM.* [Online] [Cited: December 17, 2007.] http://www.spycameras.com/cm-1201.htm.

[8]. Video Xpress. [Online] [Cited: December 17, 2007.] http://www.adstech.com/products/USBAV-191-EF/intro/USBAV_191_intro.asp?pid=USBAV-191-EF.