PHYS2601 Major Project

Autonomous
Earth
Driver

Balint Seeber
2003

Balint Seeber, 2003
balint at the domain spench point net

Contents

Diagrams

Foreword

During the first stages of brainstorming, I had in fact settled on another idea that was quite different to the one described in this report. My original intention was to build an automatic transmission system for my bicycle. I would briefly like to discuss it here with the focus on why I did not pursue it, as I believe it is a good example of the benefit research and testing can have on the development of a project (or lack thereof).

My original idea was to connect an array of sensors to my bicycle that would measure such physical parameters as speed, slope and rotation of the crank shaft. This data would be read into the BasicMicro Atom, which inturn would change the gears by pushing or pulling on the cables (connected to the derailers) with servo motors.

More specifically:

- **Speed** would be measured by attaching small magnets to the spokes on the front wheel and measuring when they pass a Hall-effect transistor tied to the frame beside the wheel. By measuring the time delay between the passing of individual magnets, a speed can be calculated (more magnets will result in a more accurate measurement).
- **Slope** would be measured by attaching an accelerometer to the frame of the bicycle (correctly oriented to face forward). As the bicycle would climb or descend a hill, the angle could be measured directly from the analog, linear device and the gears would be changed accordingly to make the ride easier.
- The **gears** themselves would be changed by replacing the finger-manipulated gear shifters on the handle-bars with servo motors. The shaft of the motors would somehow be attached to the gear cables, so that winding in a certain direction would increase the gear ratio, and 'unwinding' would decrease it.

I did a number of things to assess the feasibility of this project:

- **Tested a 'heavy duty' servo motor**: although the motor itself certainly looked as if it could turn heavy loads, the shaft's rotation could still be hindered by the grip of human fingers. Two issues arose from this: a very high amount of current would be necessary to drive two servo motors with sufficient torque and, how could the gear cables be connected to their shafts so that they would not release as soon as the motor stopped turning (simply due to the tension of the metal wire). The solutions seemed rather cumbersome and heavy: large, high current batteries carried somewhere on the bicycle and possibly worm gears to prevent the cables from being released (locking them in position).
- **Talked to several professionals**, all who expressed their concern over various components: particularly measurements coming in from the analog devices. As a moving bicycle with a rider is hardly a physically stable system, these inputs quite possibly would show large amount of variation that could upset the decisions made by the Atom. With the accelerometer for example, because of the vibrations running through the frame and 'cadence' of rider, the angle returned would fluctuate quite significantly. Although averaging could be done either inside the Atom or using a low pass filter, this would still remain a cause for concern during testing.
- **Talked to bicycle mechanics** at a reputable bicycle shop: although they thought the idea was novel, they also had doubts. One man, who is renowned for his eccentric modifications to bicycles, was of the same opinion. The other concern was due to fragility of such an experimental set up: the whole system would fall apart after a matter of moments, rendering it useless and causing more headaches while searching for parts littered on the ground in the bicycle's wake.

I was and still am determined to produce a prototype of any idea that does actually work. After considering the feedback I had received above from the experts, and the multitude of technical and engineering hurdles just to get a simple system in place, I decided to 'go back to the drawing board' and brainstorm another idea which would definitely be feasible. So I arrived at my 'GPS-controlled autonomous model racing car'.

Special thanks must go to Dr Joel Barnes, from the school of Spatial Information Systems, for lending me the Rojone GPS Receiver and radio modem for receiving DGPS corrections, and also for the time, effort and expertise he dedicated freely to helping me accomplish this project.

Aim

The scope of this report will cover the aim (see below), show the conceptual development, the actual construction of the project and analyse the results, which will indicate the success of the project. Also, future developments are listed that would have been attempted with more time.

The simple aim of this project was to develop a system that could direct a vehicle through several points on the surface of the Earth using the BasicMicro Atom.

This basic aim then dictates the kind of technology and programming to be employed:

- To track the position, altitude and heading of an object on the Earth's surface, the Global Positioning System must be used. Hence, a receiver is necessary that will output data in a format that can be interpreted by the Atom.
- For the car to actually move around the Earth, the program must contain a list of coordinates that specify waypoints in a chosen coordinate system.
- These waypoints must be dynamic (i.e.: they can be entered and changed during the Atom's program runtime). This can be achieved via numerical entry on a keypad.
- Feedback is logically necessary for effective use and debugging (it would be very difficult to use equipment if a human did not know its state). Consequently an LCD can be used to show character data, providing an interface displaying a series of simple menus for data input and output.
- Due to atmospheric disturbances (but no longer US-enforced 'Selective Availability') coordinates obtained from GPS receivers can be highly inaccurate on the distances scales intended in this experiment. Making use of Differential GPS corrections can greatly increase accuracy to a resolution of a couple of meters. So if navigation that solely relies on GPS is not found to be accurate enough, the receiver's effective resolution can be improved by utilising a DGPS receiver.
- As an extra feature: a distance sensor mounted on the front of the vehicle to detect imminent collisions and drive accordingly to avoid an accident.

Although this kind of autonomous navigation has been accomplished in commercial products, the goal of this project is to make such a system work within a small vehicle, namely a model racing car. Considering the size of such a vehicle, two limitations come into effect:

1. **Total mass of additional components and circuitry**: sensibly, the car will only be able to drive with the mass of the project (sitting on top of the car) if it is under some threshold. Therefore care must be taken in not overloading the motor.
2. **Driving range of car**: As the car is small, the distance over which it can drive is also limited. This means that the resolution of the GPS has to be sufficiently small so that an 'average' change in position (caused by driving the car for a short period of time) will be reflected on the receiver's coordinate fix.

Equipment

Main components:

1.	1 x	BasicMicro Atom PIC Microcontroller (40-pin)
2.	1 x	Tamiya Model Racing Car (with steering and drive servo, driver motor and high capacity drive motor battery)
3.	1 x	Rojone Genius GPS Receiver (with serial and DGPS interfaces)
4.	2 x	Parallax Wireless Transceivers
5.	1 x	Pacific Crest Radio Modem
6.	1 x	16-character, 2-line LCD
7.	1 x	16-button Keypad
8.	1 x	Infra-red Distance Sensor

Additional components:

9.	2 x	4 C-cell battery holders
10.	1 x	4 AA-cell battery holder
12.	2 x	Rojone GPS Receiver-compatible serial cables
13.	1 x	Null modem serial connector set
14.	1 x	Small breadboard
15.	1 x	Cardboard box housing
N/A	N/A	Connector wires

Comments on devices:

1. Forty pins on the Atom makes interfacing less stressful because there are plenty of pins available (especially as certain pins are only available for interrupts and hardware commands), which means there is also room for expansion.
3. The Rojone receiver has a very compact design and is designed for the express purpose of interfacing (there is not display and no buttons), making it highly suited for this project. As the receiver listens for GPS signals from the satellite constellation, it can only get a fix outdoors in an area seeing large amounts of the sky (unobstructed by buildings). Also, it can be fully configured with the SiRFStar GPS software. For this project it was set up to send only the one 'GPRMC' NMEA message at 9600 with no checksum.
4. These wireless transceivers are supposed to be 'plug-n-play' with the Atom. However they were discovered to be temperamental in their serial operating mode (the nature of which will be discussed later).
5. This is a nice device which makes serial communications transparent over the airwaves. All that is needed is a power source and an antenna. Error checking and correction of the serial data is inbuilt. The range with a medium-sized antenna was found to be approximately one kilometre.

The interfacing of the electronics takes place at two locations:

1. Prototype Atom development board (developed for just project)
2. Prototyping breadboard (for wire patching and off-board components)

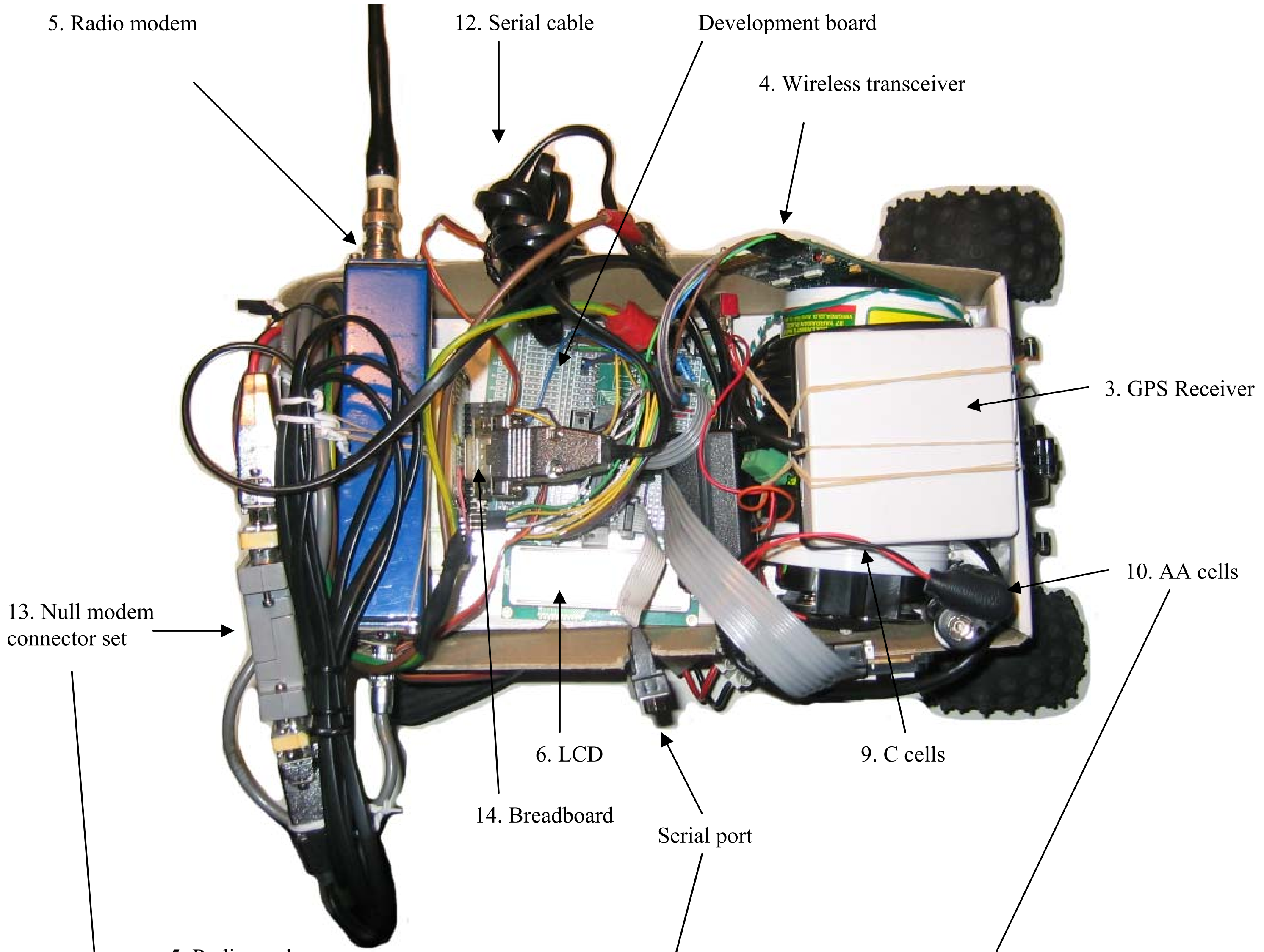

Figure 1: Top view of car

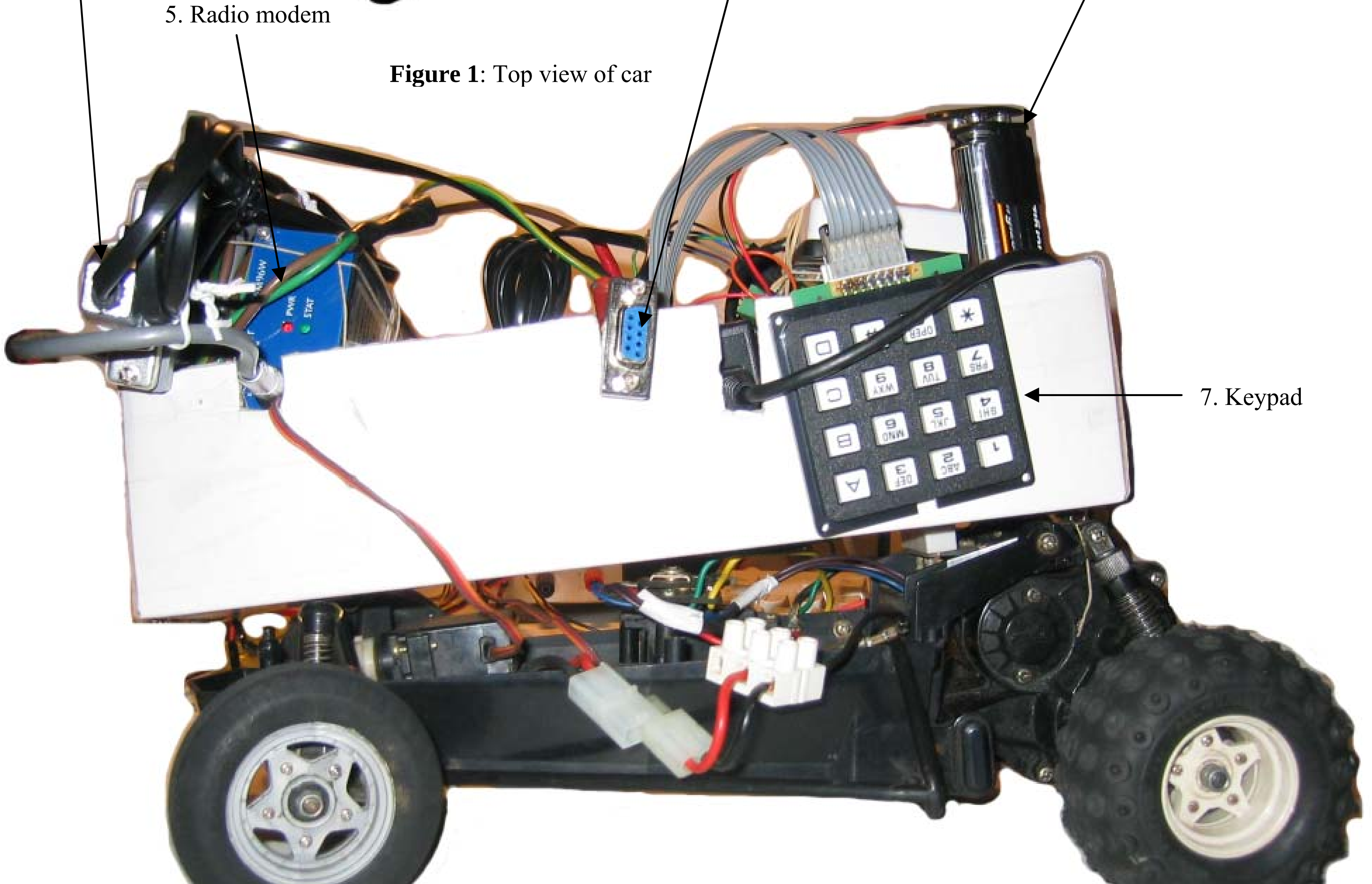

Figure 2: Side view of car

Introduction

Since the vehicle will be directed using data being output by the GPS receiver, two things must be understood to have successful navigation:

1. The format of the raw data being output by the GPS receiver (so that 'intelligent' information can be extracted)
2. The meaning of the outputted GPS information (how it can actually be used to navigate from one point to another)

The majority of modern amateur and professional GPS receivers all communicate with external equipment using the National Maritime Electronics Association (NMEA 0183) protocol. This is an ASCII serial (RS-232) protocol, making the job of decoding data and interfacing a receiver to other equipment a relatively simple task. NMEA information is sent in a 'word' with the following format:

- Begins with a "$"
- Followed by a five letter code defining the meaning of the data in the rest of the word
- Data values separated by comas (data can be integers, floats, letters but is ASCII formatted)
- Terminated with an optional checksum, but mandatory Carriage Return and Line Feed.

Here is an example of a NMEA word:

```
$GPRMC, 115947.999,A,3354.1938,S,15112.0945,E,0.26,323.56,230903,,
```

This is the NMEA word that is parsed by the Atom. The data that is used is:

- Fix status ("A" is valid, "V" is warning)
- Current latitude (degrees and decimal minutes): 33° 54.1938 min S
- Current longitude (degrees and decimal minutes): 151° 12.0945 min E
- Current speed (meters per second): 0.26 m/s
- Track made good in degrees true (this is only meaningful if the unit has been moving – if not, it will fluctuate about zero degrees north): 323.56° True

The question of what can be done with such data still remains. Although the latitude and longitude coordinates obtained directly from the GPS may initially seem as acceptable values for finding distances and angles between waypoints (using early high-school mathematics), for accurate results this is not the case. As the Earth is a spheroid, latitude and longitude cannot be used as a coordinate system on a Cartesian plane. There exist a number of distance approximations (Great Circle, Spheroidal model of the Earth) and coordinate transformations (Earth Centred Earth Fixed to North East Down coordinates) if such accuracy is required. For this project, it is both unnecessary and technically infeasible to use either. (This will be fully explained below.)

Since all of the serial data being output by the GPS receiver must be received by the Atom for successful parsing, a reception method must be used that guarantees that the NMEA words reach the program in their entirety. Normal serial communications (using the `SERIN` command) with the Atom block the program until a certain number of characters are received or a timeout occurs, for example. This is not suitable for this project as the program would not be able to do other important things while waiting to receive a full word of NMEA data. Luckily, the Atom has a built-in hardware serial UART (with internal buffer), which can asynchronously receive the NMEA data and later be read by the program as soon as possible. A quick browse of the 'Hardware Commands' in the Atom command reference manual revealed the commands to manipulate the UART. Another (undocumented) command was found on the BasicMicro forums that can be used to poll the status of the UART (such as to check whether there is any data to be read from the receive buffer). The following commands were used:

1. `HSERIN` – reads in data from the UART's receive buffer
2. `HSERSTAT` – polls the status of the UART (see reference for more information)

As the 40-pin Atom is being used, the general-purpose analog I/O pins are available for use. The infra-red distance sensor is an analog device whose output voltage must be converted into a digital number for use in the program. So the other new command to be used is:

3. `ADIN` – performs and Analog-to-Digital conversion on the specified analog pin (`AX0` – `AX3`) and puts resulting value in a variable

Although it is clear how to use the information obtained from the GPS receiver, numerical calculations must be performed on the parsed numbers. Unfortunately the Atom's library exposes very simplistic mathematical functions (none of which suffice for the project). It lacks all inverse trigonometric functions and has only a basic square root function (which operates in the range of 0-255). Both of these were assumed to be required (especially to work in the maximum number range allowed by the hardware). As a result of these shortcomings of the Atom's library, research was done into simple, fast implementations of the functions. The following subroutines were ported to the Atom's MBasic language (utilising the available floating-point operations) from various other languages (including C, Basic and assembly):

1. `SQRT` – performs 'Fast-Integer Square Root'
2. `FSQRT` – performs 'Fast-Integer Square Root' with float-point conversion
3. `ATAN` – performs arctangent inverse trigonometric function
4. `ASIN` – performs arcsine inverse trigonometric function
5. `ACOS` – performs arccosine inverse trigonometric function

(The inverse trigonometric functions return an angle in radians.)

Apart from using the newly discovered functions and written subroutines above, some work done in previous laboratory exercises was essentially 'copied-and-pasted' into the source for this project. The code snippets were first cleaned up (to match the style of the project's source). The following code was recycled (slightly lessening the development time by making use of existing, working programs):

1. **Reading in the status of a keypad connected to the Atom**: this program is interesting for two reasons. One is that to enable direct connection of buttons to the Atom (to avoid the problem of floating inputs) the invaluable '`SETPULLUPS PU_ON`' command must be executed. Secondly, since the inputs are now 'pulled high', inverted logic must be used to distinguish button states bit-by-bit. Being able to recycle such working code makes programming that much easier.
2. **Writing data out to a LCD connected to the Atom**: although no complete program is required to do this (Atom commands make such writing and reading very simple) it is good to have practise using the more advanced commands (such as cursor positioning and display mode) to enhance a basic user interface.

Bearing in mind the several limitations discussed here and in the Aim related to the distance the car is expected to travel, the coordinate system returned raw from the GPS receiver (latitude and longitude) will be used for navigational purposes – no coordinate transformation will take place. The reason is twofold:

1. On such small distance scales (in the absolute maximum range of 100 meters), latitude and longitude change approximately linearly – that is, enough to calculate angles as if they were horizontal and vertical coordinates on a normal two-dimensional Cartesian place.
2. The physical program space of the Atom introduces a severe restriction on the amount of code that can be written. Considering the number of lines of high-level code to perform the necessary coordinate system transformations, translating them into MBasic would fill up a large portion of the available space and restrict the rest of the (far more 'mission critical') functionality of the project.

Clearly, using the latitude and longitude values alone prevents the vehicle from accurately navigating over large distances comparable to hours of high-speed travel across the Earth's surface.

Thus to be able to navigate from one point (the position of the car) on the Earth to a user-specified waypoint, a simple mathematical subroutine (using the custom written math functions listed above) can solve the problem. The mathematics of it is shown in the next diagram.

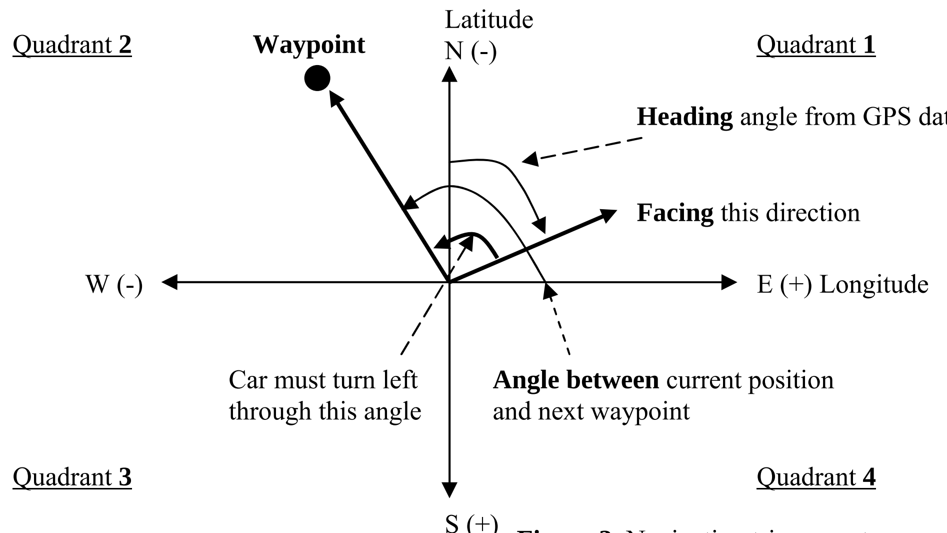
Quadrant 2
Waypoint
Latitude
N (-)
Quadrant 1
Heading angle from GPS data
Facing this direction
W (-)
E (+) Longitude
Car must turn left through this angle
Angle between current position and next waypoint
Quadrant 3
Quadrant 4
S (+)

Figure 3: Navigation trigonometry

To calculate 'x°' (the "**Angle between** current position and next waypoint"), find the gradient between the two and calculate the arctangent of the answer:

$$x° = \arctan \left(\frac{\text{(current latitude – waypoint latitude)}}{\text{(waypoint longitude – current waypoint)}} \right)$$

The current and waypoint latitudes have been swapped on the numerator as the vertical axis is mirrored because latitudes that are increasingly south are of great magnitude.

Before the value of x° can be used, it must represent the full revolution (360°) rather than just the +ve/-ve 90° valued given by arctan (accomplished by know which quadrant the waypoint is in) and then be altered so that 0° is at True north instead of east and the increasing angle proceeds clockwise around the plane, instead of in the normal anti-clockwise direction. To do this (if x° has indeed been converted into degrees):

$$x° = (360° - (x° - 90°)) \bmod 360°$$

Mod'ing ensures that the result remains in the region where it physically makes sense. 'y°' (the heading) is obtained from parsing in the NMEA data. This angle will only be valid if the car has been actually moving.

The angle through which the car must turn 'z°' is: $z° = x° - y°$

(or the angle to the waypoint minus the current heading.)

Now if z° is positive the car should turn right, else if it is negative it should turn left. It may happen that this angle is greater than 180° or less than -180°, in which case the above logic would not work (because the car would end up turning right around first). Therefore if the value is out of these bounds, 360° is subtracted from it or added to it respectively. This will assure z° is in the range [-180°, 180°] and the steering logic will work.

If |z°| is greater than 45°, then the car should steer full lock in the chosen direction. If |z°| is less than 45°, the car should steer fractionally less in that direction by (|z°|/45°).

Procedure

The initial project plan was set to follow these steps:

1. **Obtain parts**: All the items on the equipment list have either been bought or borrowed: the Atom was ordered from BasicMicro, while the GPS receiver was kindly lent to me for the duration of the project.
2. **Experiment with Atom**: Create test code (for coming to grips with hardware commands and serial I/O with the UART) and attempt interfacing (actually connect the GPS receiver and read in its outputted serial data, as well as controlling the servo motors on the model racing car).
3. **Build car**: Put together the physical equipment into one unit (i.e.: have a complete and independent vehicle).
4. **Finalise theory**: Analyse the mathematics and various models that could be used for calculating navigation and select the best one based on constraints. (The model was described in the previous section).
5. **Write main program**: Bring together test code with recycled code and build them into the main project source.
6. **Test and Debug**: Test the program at many levels by allowing car to drive itself and observing behaviour, reading information from the LCD and manually moving the car around in one's hand. Then make appropriate changes to the source (bug tracking and correction) based on these observations.

What was actually done deviated very little from this plan and is described in detail here:

1. Obtain parts

Apart from obtaining the equipment itself, a prototype Atom development board was created for ease-of-use and future experimentation. It is a simple development board and possesses the following features:

1. Terminal block connectors for power
2. Voltage regulator (with power switch and power LED)
3. IC holder for Atom (compatible with any size development chip)
4. SIL pins for connection to other hardware
5. Serial port for PIC programming

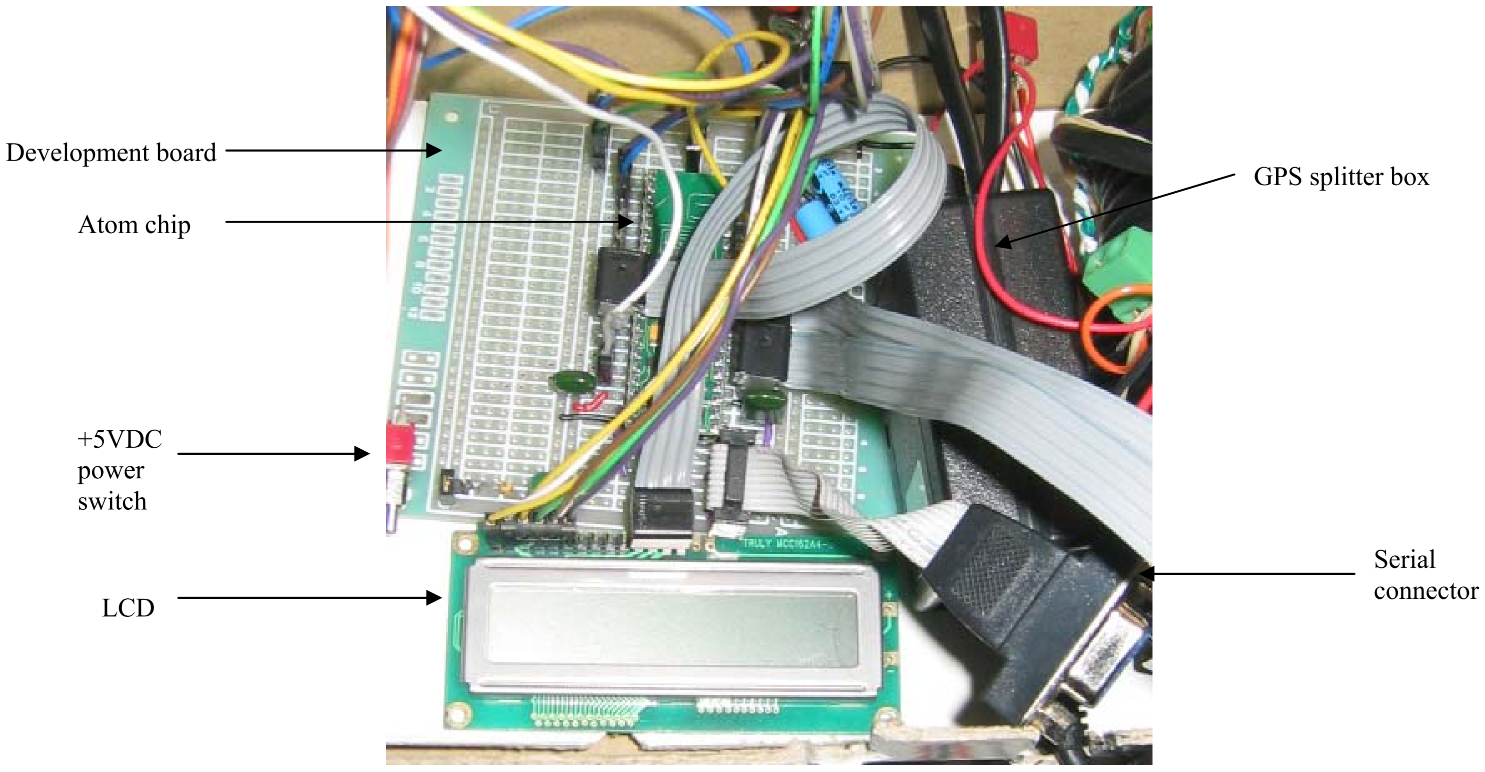

Figure 4: The development board

Later, for interfacing the Atom to other equipment, a small breadboard was used. A DIL pin block was pushed into the board so it could serve as a patch point for the servo motor connectors and I/O lines for the Atom.

The servos can be directly controlled by the Atom using a looped and delayed `PULSOUT` command.

However before the serial data from the GPS receiver can be read by the Atom, line conversion must take place. This is because the receiver is transmitting at correct RS-232 line voltages (measured to peak at -10V for logic high and +10V for logic low), which are incompatible with the Atom and would cause it damage. A line receiver (MC1489-D) mounted on the breadboard is used to convert the RS-232 line into equivalent TTL levels (+5V for logic high, 0V for logic low). The GPS receiver's serial output cable connects into a male DB-9 connector also mounted on the breadboard. The line receiver's output is connected to a pin on the breadboard's DIL pin block, which then connects to the Atom's serial UART input.

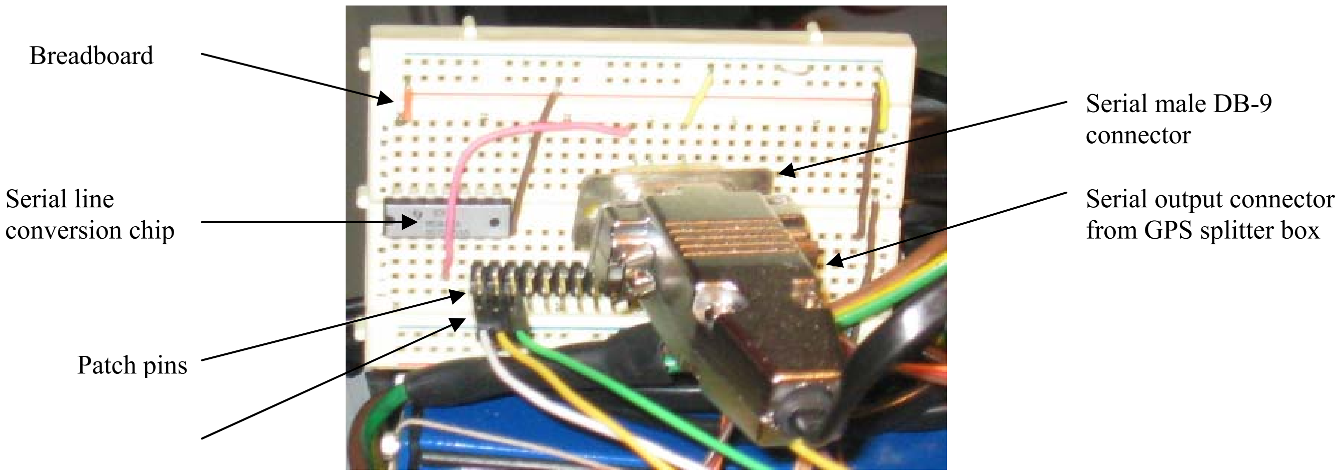
Breadboard
Serial line conversion chip
Patch pins
Serial male DB-9 connector
Serial output connector from GPS splitter box

Figure 5: Breadboard

2. Experiment with Atom

The first stage of experimentation was performed before the car was built up and had the goal of receiving the serial data output by the GPS receiver asynchronously (using the UART) and then parsing it into intelligent information.

Programs were written, starting from the most basic to a complete parser, to test and become familiar with the Atom's buffered serial communication capability. The first program simply output the data that was received by the UART to a connected computer to verify that the hardware really worked (line conversion from the GPS receiver and the UART on the Atom). This data was then read in by a more sophisticated program and parsed byte-by-byte (to prevent any blocking in the software) into respective variables. The exact variables can be found in the source code listing, but examples of them are the integral and decimal components of the current latitude and longitude returned by the GPS receiver. This test program was a success and displayed the correct information on the LCD (so it was recycled into the main source code later on).

Next the serial transceivers were tested. Unfortunately they were found to be incredibly temperamental: sometimes gibberish would often pop up in the received data (or barely any data would get through) even though the transmitter was in close range. Swapping the 'transmitter' and 'receiver' boards around would sometimes alleviate this problem. But even so, great problems were encountered when trying to use them in the project. When transmitting data, care has to be taken to use proper flow control as, although serial data can be sent to the unit at 9600 baud, the pair only communicate at 1200 baud. This introduces delay of about 15 milliseconds per byte. So transmitting a medium sized packet of data (for example current position and heading) would take far too long, blocking the main program (which as previously mentioned is unacceptable). As a result, the transceiver is only used to transmit the fact it has reached a waypoint, as the program pauses (so the car 'parks' temporarily at a waypoint).

The software also had to be calibrated so it could control the servo motors properly (this was done after the complete car was built).

3. Build car

The construction of the complete car itself was an easy task. The model racing car was stripped down to its bare bones. The remote control radio receiver and its power source were removed (as they were no longer necessary). The car's drive motor speed is controlled by a servo that turns a switch. This switch, through a number of contacts, sets the direction (forward or reverse) of the motor and its speed (full speed when current is supplied directly to the motor or medium speed when it is supplied through a 0.3Ω 7W resistor). So the path the current takes across certain contacts is controlled by the servo. The steering is also controlled by a servo motor and is a straightforward translation from servo position to a steering angle.

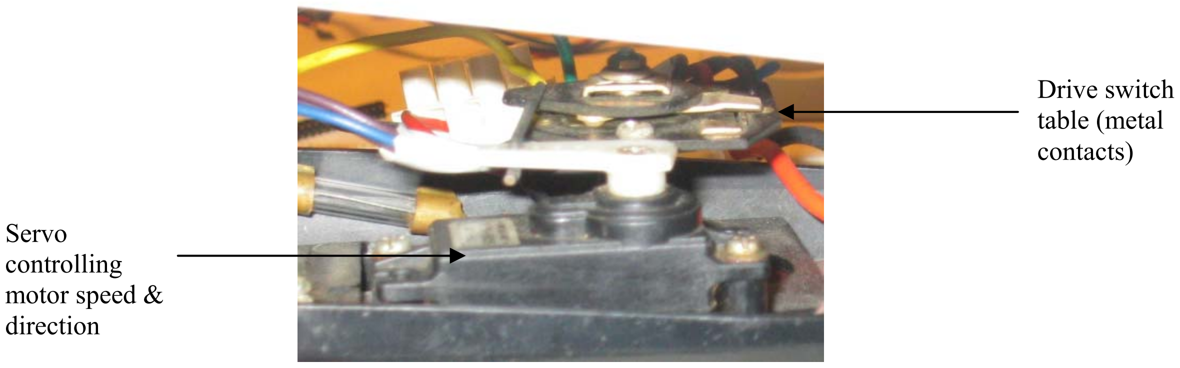
Drive switch table (metal contacts)
Servo controlling motor speed & direction

Figure 6: Drive control servo and switch table

A sturdy box was mounted on top of the car's frame using strong metal wire. In the inside of the box was placed a sloped piece of cardboard so that the surface would lie parallel to the ground (as the box is tilted due to its mounting on the uneven frame). Into this box was placed the electronics and onboard power supply. The servo motor connectors were patched into the breadboard (which inturn connected to the Atom). The infrared distance sensor was mounted on the front of the box and also connected to one of the Atom's analog I/O lines via the breadboard. The keypad and LCD were placed next to the Atom development board and connected to its SIL patch pins. A pair of full battery packs laid side-to-side were put inside a horizontally placed cylindrical plastic container to save space. The cable coming out of the GPS receiver to its own splitter box (into which power and the serial lines are connected) was wrapped around this container. The GPS receiver itself was secured on top of the container for the highest position on the car. The power supply was connected to the power terminals of the Atom development board via a 'main switch' mounted into the box. The GPS splitter box was connected to the power as well, with its serial output connected to the DB-9 plug on the breadboard. Lastly, the serial transceiver was connected to the Atom.

[Intentionally left blank.]

Figure 7: Combined block and circuit diagram of electronics on board

Once the complete car was constructed, development entered the next phase in testing and calibration. This centred on obtaining correct software values to control the servo motors, and hence drive the car at a known speed and direction.

This particular model of racing car uses Japan Radio servo motors (whose colour code is: Brown: 0V, Red: +5V, Orange: servo control). A program was written that would accept a servo position via digits entered on the keypad and pulse the chosen servo to a position. This was essentially a trial-and-error exercise. After experimenting with different values, the precise numbers (to become calibrated constants in the main source code) were found. Although it seems like a trivial advance, it means the Atom actually has the ability to control the motion of the car (precisely steering left and right, and moving forward or backward at either medium or full speed).

Another program was written to test the ability of the Atom to control the driving and see whether the car would actually move under the increased mass. It was hard-coded to drive forward, reverse turn and head back. The car did in fact move but the 'centre' constant of the steering servo of slightly off, making the car veer off to the right instead of driving straight. The value was changed and this problem fixed.

4. Finalise theory

As the basic mathematics behind the navigation procedure had been decided, it had to be turned into a conceptual series of calculations that can be programmed in the project source. The following flowchart summarises the steps involved in calculating navigation while the car is driving (it is only run when a new GPS coordinate is parsed by the program):

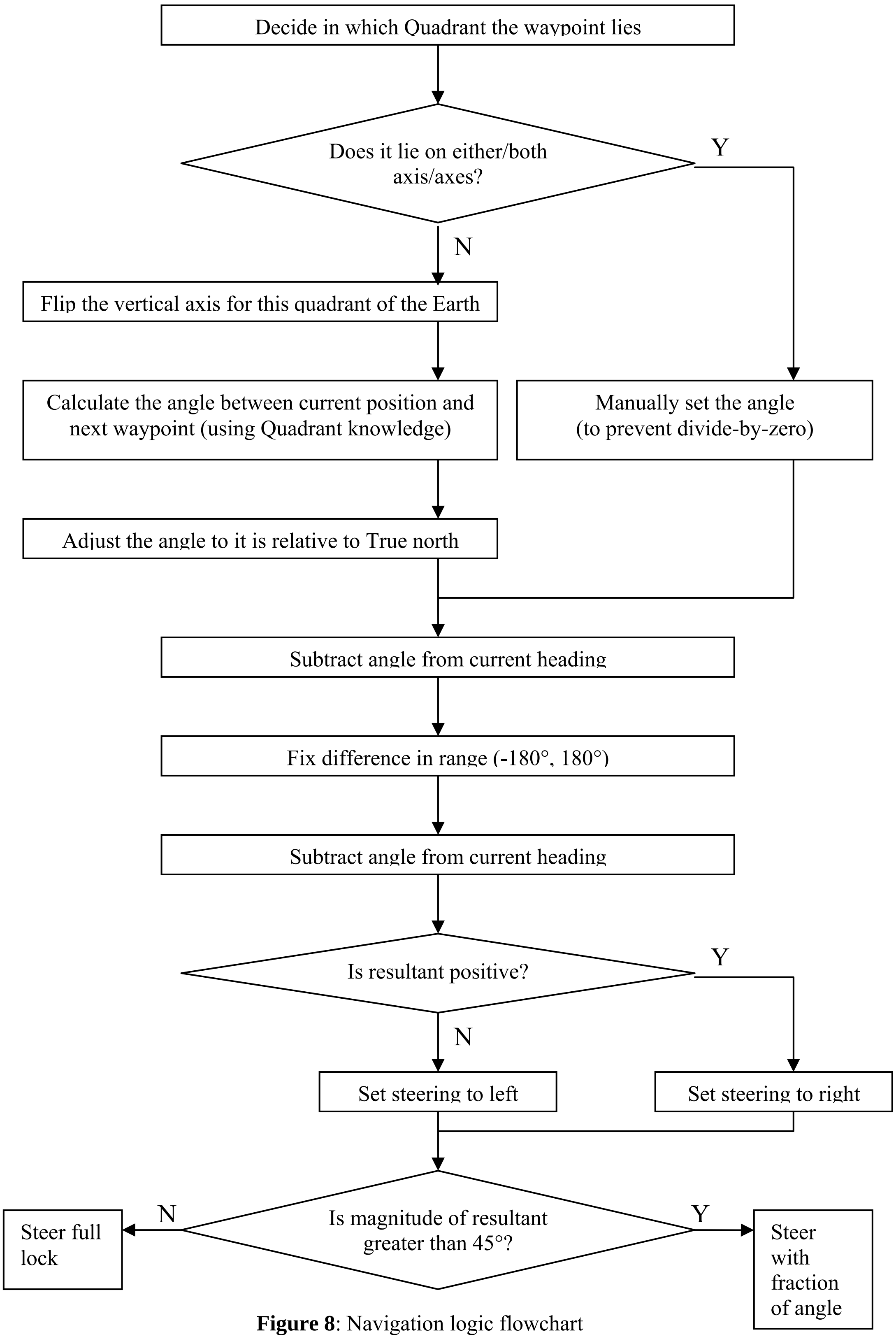
Decide in which Quadrant the waypoint lies
Does it lie on either/both axis/axes?
Y
N
Flip the vertical axis for this quadrant of the Earth
Calculate the angle between current position and next waypoint (using Quadrant knowledge)
Manually set the angle (to prevent divide-by-zero)
Adjust the angle to it is relative to True north
Subtract angle from current heading
Fix difference in range (-180°, 180°)
Subtract angle from current heading
Is resultant positive?
Y
N
Set steering to left
Set steering to right
Is magnitude of resultant greater than 45°?
N
Y
Steer full lock
Steer with fraction of angle

Figure 8: Navigation logic flowchart

To help in the programming, the integers representing a waypoint's latitude and longitude are of a special format specific to this project. The GPS receiver outputs a coordinate fix to an accuracy of four decimal places of minutes. As these coordinates do not change drastically with the movement of car, and considering that the area in which the car is moving is within one degree, the degrees of the coordinate are omitted altogether. The remaining minute values are expanded to a range from 0 to 99 (from 0 to 59) and converted into integers. This is done so that the new coordinate values extend through a full decimal range (making navigational calculations possible) and so they can be stored as `LONG` integers rather than floating point numbers, which, in the context of the Atom, makes mathematical operations much easier and speedy. This new coordinate system will be referred to as 'DMS-100'. (**D**egrees were originally included, but it was realised that to fit the extra degree digits in, the least significant minute digits were being sacrificed. This was unacceptable as this is where the major changes are seen when the car moves around.)

5. **Write main program**

The main program was written, drawing upon the recycled source and converting the navigation procedure into decision making and numerical calculation code blocks. The program was written with flexibility in mind, so nearly all of the 'magic numbers' are defined as constants at the top of the program listing for easy alteration.

The program is divided up into three parts:

1. **Menu interface**: Simple text menus that can be navigated via the keypad, which facilitates user interfacing. The menus expose the following functionality:
    a) **Editing of waypoints**: The program stores up to nine waypoints in RAM. Each can be edited by selecting menu option '1' and then the index of the waypoint to edit ('1' through to '9'). A waypoint comprises of two integers: the latitude and longitude in DMS-100 format.
    b) **Waiting for a GPS fix**: the Atom can be set to wait until the GPS receivers can 'see' enough satellites to start outputting valid NMEA data. When this happens, the data is parsed and displayed on the LCD. This is invaluable for testing as the coordinates for waypoints can be noted down off the LCD for later entry. One additional user interface improvement to be implemented is the direct transfer of on-screen coordinates into a waypoint slot (instead of jotting it down on paper and then typing it in before driving).
    c) **Drive**: Cause the program to enter into 'drive' mode and actually navigate from the car's current position to the next waypoint, and then the next until it reaches the last.
2. **Navigation (or 'driving')**: this is the core of the project encapsulated in code. The main 'drive' subroutine is called whenever a new NMEA word is received and parsed (this happens every second – the GPS receiver has been configured this way). When first starting this section of code, the car is set to drive forward for about two seconds. This is done so that a valid heading value is returned from the GPS receiver and navigational calculations will be meaningful.
3. **Unblocked checks**: there is only one check that is performed during driving, and that is converting the analog input from the infra-red distance sensor and checking whether the resulting number is greater than the threshold constant. If so, the motor is stopped until the number drops below the threshold again.

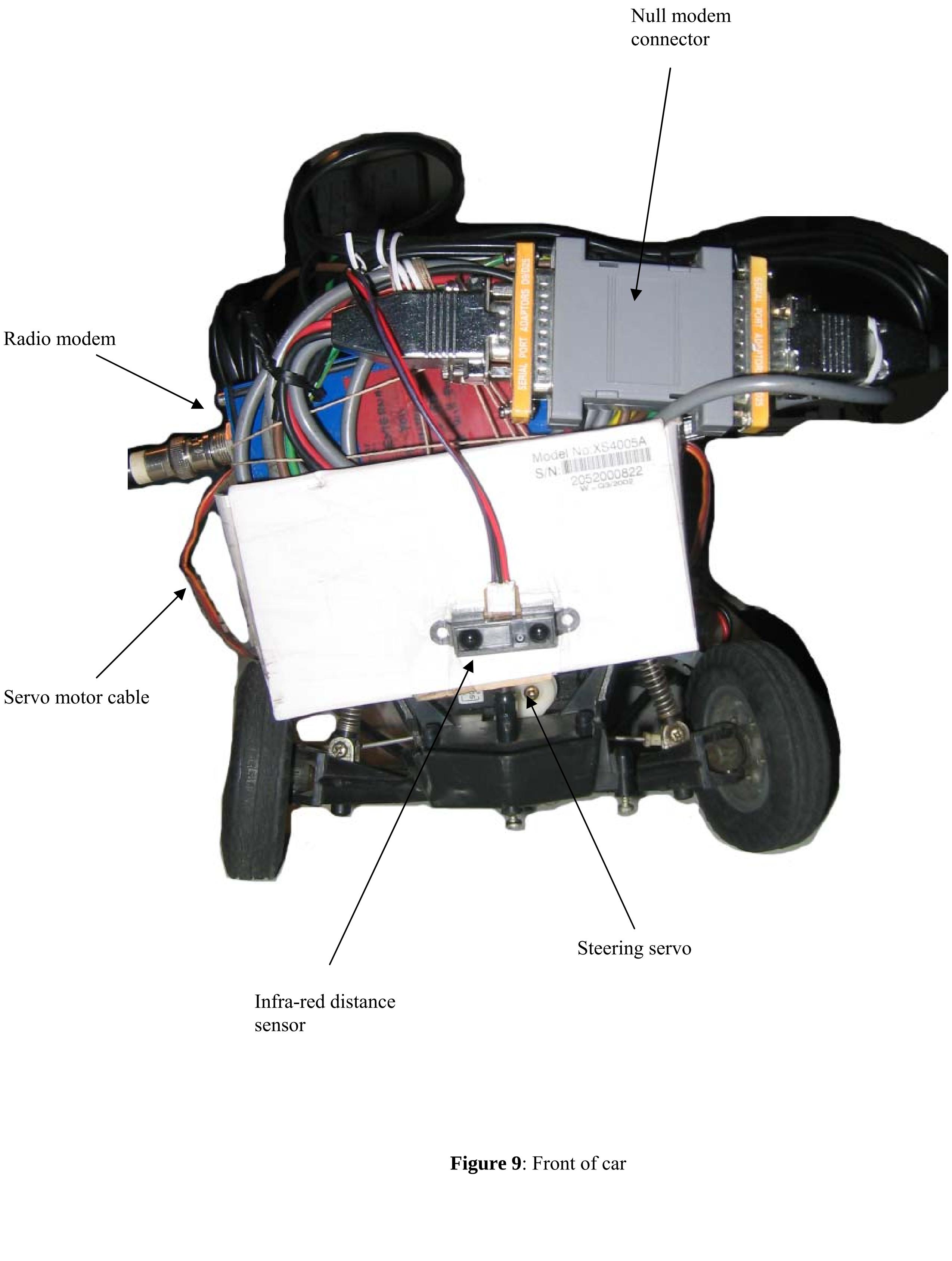

Figure 9: Front of car

6. **Test and Debug**

Once the smaller user interface bugs were eradicated, a very large amount of time was dedicated to manually testing whether the navigation subroutine actually worked. This process is described in the Discussion as it is necessary to first analyse the raw navigational data being show on the LCD.

Despite early hopes of not having to add a DGPS system, testing was severely impeded by the inaccuracy of the GPS fix. Luckily the situation could be improved in a straight forward fashion. The school of Spatial Information Systems has its own DGPS system that can broadcast corrections over a radio modem. All that was required was another radio modem to receive this data. Since the GPS receiver already has a serial port for accepting corrections automatically, the output of the radio modem (after being passed through a ‘null modem’ converter) was plugged straight into this port. No further adjustments were necessary! One additional limitation is put on the project however: it can only be used in the vicinity of the university where the radio modem can receive the corrections signal.

Results

There are two sets of raw data to present:

1. **GPS Coordinates (without DGPS corrections)**: These are the parsed and converted coordinates from the GPS receiver. The main issues are: inaccuracy at low resolutions and the fact that coming back to the same physical position often yields a different coordinate well outside acceptable error margins (throwing the whole system off).
2. **Heading calculations (with DGPS corrections)**: These are used for navigation (the GPS coordinate issues described above are largely solved, or within acceptable limits, using DGPS). There are only a few values as the algorithm is simplistic, but they give important clues as to whether the system is working correctly or otherwise.

The data was collected at different locations, some of which will be shown here. Luckily the ‘issues’ discovered with the GPS coordinate fix and software bugs were not location-specific.

The coordinate data is collected while setting the car into “Wait for GPS fix” mode. When the GPS does have a fix, the DMS-100 coordinates are displayed on the LCD and noted down.

The navigational data is collected while setting the car into “Drive” mode (and having the battery driving the motor disconnected). The program displays the quadrant, angle to waypoint, current heading, angle difference and current speed on the LCD. The quadrant selection is the first thing to check as one walks the car right around the waypoint, checking that it shows all four quadrants. Next the ‘angle to waypoint’ must be checked to see if the inverse trigonometry calculations are correct. Lastly the angle difference must be checked to see that the car will try and steer in the correct direction. This can be confirmed by watching the turning of the tyres as the servo motor moves them.

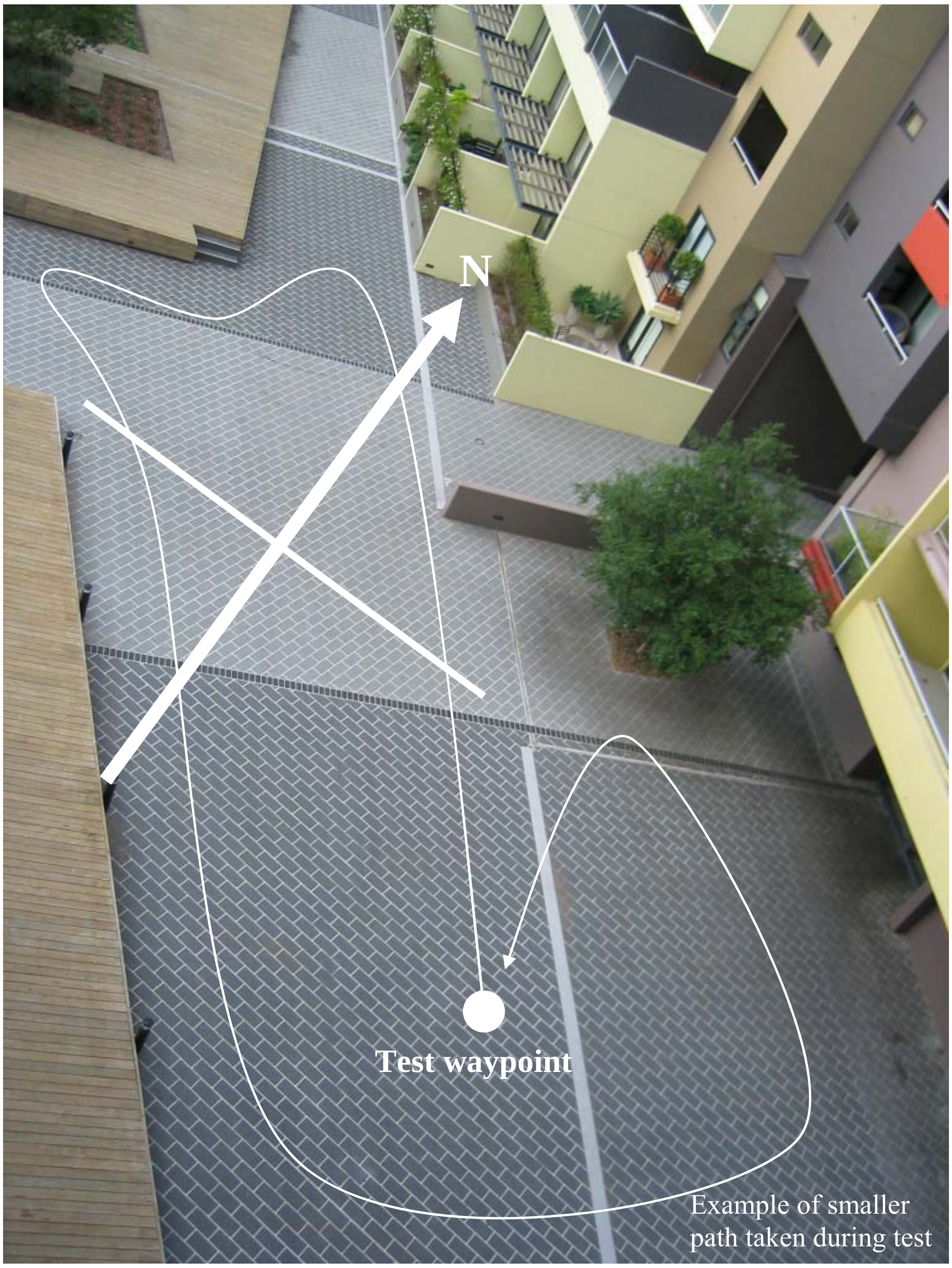

Figure 10: GPS coordinate testing area

After standing still at waypoint, and wandering around and returning, the

latitude varied from:	903468 – 903523	(diff. = 55)
while longitude varied from:	201703 – 201734	(diff. = 31)

Coming back the next day and measuring the same waypoint,

latitude varied from:	903331 – 903456	(diff. = 125)
while longitude varied from:	201344 – 201478	(diff. = 134)

Considering these changes and the small area, the waypoint would have been moving over considerable distance relative to the car (so the quadrant selection would have been completely off).

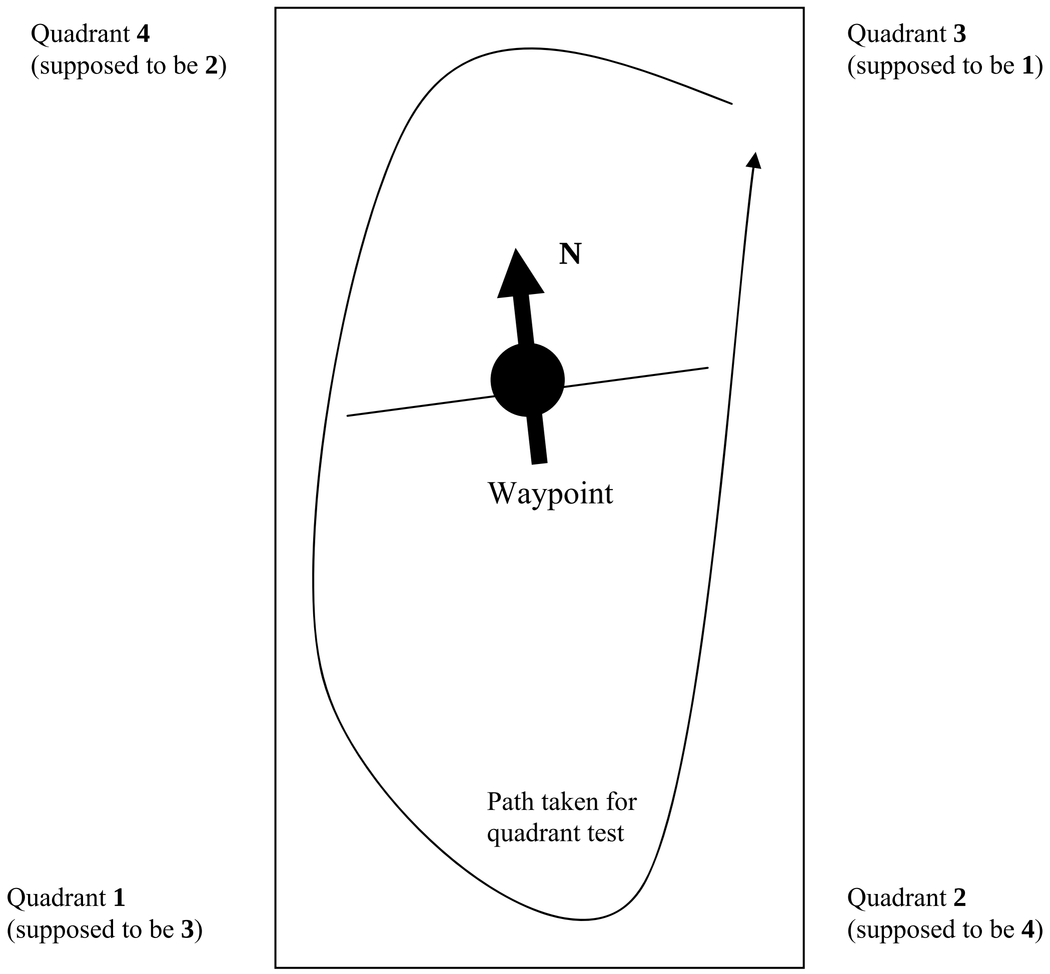

Figure 11: Plan view of roof of CSE car park
(This location was chosen as it has the best view of the sky, is in radio modem range
and is the largest, flat paved area where the car can travel with minimal friction.)

The main purpose of this test was to have a look at the difference DGPS corrections were supposed to make to the coordinate fix. Luckily they did improve both the accuracy and the precision of the physical coordinate (meaning that on returning to the waypoint, the coordinates were not completely off like in the previous test).

Waypoint latitude:	919579 – 919594	(diff. = 15)
Waypoint longitude:	231381 – 231403	(diff. = 22)

Once this waypoint was programmed in, the car was put into manual "drive" mode (no power to drive motor) to test navigation. The quadrants did change around the waypoint, which was a relief. However the calculated quadrants did not match the expected ones (shown above). They have been rotated by 180° clockwise. The reason for this could not be found in the code, nor in the coordinate system model. For the moment it remains the 'mystery quadrant bug' and has been temporarily rectified by manually switching the quadrants in the source code.

Once the Atom was reprogrammed the correct quadrants were being displayed, however the angle calculations were incorrect. It is safe to assume that other corrections need to be made to the source as a result of hardcoding the 180° rotation.

There are three more (though minor) results to discuss:

1. **Collision sensor (IR distance ranger) values**: Software calibration was first done indoors. The threshold value (above which the software would think a collision is about to occur) was consequently set as a constant. During testing outside, the car would often just stop while driving (while manually moving the car around, "`Collision!`" could be seen appear on the LCD). Yet there was no physical obstruction to the car. One possible reason is that infra-red conditions are different outdoors, and very much dependant on the amount of sun reaching the sensor. Even after re-programming the threshold constant, during different weather conditions the car would sporadically come to a halt. In the end the sensor was disconnected and I/O line grounded. The 'nice' way of getting around this problem would be to use dynamic thresholds (this is discussed in the Future Extensions section).
2. **Serial communication via transceivers**: As previously mentioned in the 'Experiment with Atom' subsection, the transceivers are too slow to transmit even medium sizes of data and sometimes cannot even receive the data over short distances. Consequently their role is very limited.
3. **Power consumption**: As there is a considerable number of electronics requiring power, it was found that the primary 12V battery pack would quickly settle between 10-11V. Although the components requiring 5V (via the development board's regulator) would continue to function, the GPS receiver (which has its own 8V regulator) would experience a reoccurring brown-out. This was solved by boosting the voltage to just the receiver by connecting a 6V AA battery pack in series between the primary battery pack and the regulator's positive input.

Discussion

Both sets of raw data above carry their own implications about the project:

1. **Using uncorrected GPS coordinates is too inaccurate**: the most obvious problem lies in the fact that one returning to the same position, the GPS coordinates are well outside the acceptable margin of error. This is as if the waypoint began to wander across the Earth (which of course is ridiculous). The frustration caused by this during testing prompted the installation of the DGPS receiver.
2. **Final software bugs still to be ironed out**: due to the 'mystery quadrant bug' and its illogical patch, the remaining portion of the navigation algorithm still has to be checked and updated to match the different conditions surrounding the heading and direction to drive to the next waypoint.

If the car is unable to steer properly using the current algorithm (constant steering updates with each new NMEA word), then the next option is to introduce a longer forced delay after the car has steered into its new direction and straightened up again. This will ensure that the car does head in a definite direction, as opposed to 'oscillating' about a certain heading if the new GPS coordinates and heading confuse the algorithm too quickly.

The final point to make here is that the project has succumb to physical limitations set by the most critical component: the Atom. Bearing in mind the number of things the program does, and the range of BasicMicro library functions it calls, the number of bytes used by variables in memory and the code in program space has reached both limits. With the included listing, the compiler reports eight bytes of RAM free and 110 bytes of program space free. At one stage both were exceeded, but efforts were made to optimise the source and cut down on the number of debug messages being generated and displayed on the LCD that originally aided in testing.

It must be emphasised that the project is almost at a stage where it should work. All it requires is a software tweak to make the car drive in the right direction.

Future Extensions

It is highly improbable that future extensions would be possible with the current Atom setup; the project has reached a physical limitation enforced by the hardware. Due to the fact that the program and memory space is almost totally full, no additional functionality can be added to the program. This can be solved in two ways:

1. **Replace the Atom with another PIC/microcontroller**: one with much more program and memory space. If the replacement does not support the BasicMicro library then the source code would have to be ported to a supported language.
2. **Manually convert the source code into assembly (thereby keeping the existing Atom)**: after analysing the assembler generated by the Microchip MBasic tokeniser, it is thought that using the BasicMicro library is an extremely inefficient way of create 'tight', small code. Writing the software straight into assembly would produce incredibly compact code, freeing up a large portion of space that could be used to implement the following extensions:

- **Intelligent collision sensing**: the idea here is to augment the one 'dumb' infra-red distance sensor to detect frontal collisions with several distance sensors equally spread around the car. In this way, the microcontroller could read in the converted analog values and 'sense' imminent collisions from multiple angles. The microcontroller could then be intelligent about its future driving by approximating the surrounding surfaces and, for example, reverse in such a way as to avoid the object. This could be further developed by using off-board memory as storage for a collision map the program builds up as the car drives along. This is akin to a mouse trying to find its way through a maze. Although initial implementations would be simple, there's no virtual limit to the complexity of such algorithms. Taking the simple case, the car might encounter a large object that blocks its path over a significant distance (such as a wall). The car could then be smart enough to try and drive, following the wall until it senses that it's in the clear again. If it drives over a threshold distance, it might turn around and head back the way it came (its path being stored in memory) and then attempt to trace around another way on the alternate side of the obstruction. Of course the limitation once again is program space (large amounts would be required for even a simplistic algorithm) and memory to store collision maps (the great resolution in such maps would require increasing amount of RAM – then access times might start to be a problem too).
- **Implementing the 'Options' menu**: One feature originally intended to be in the main program was a fourth menu called 'Options'. This would provide an interactive way to modify those 'magic numbers' defined as constants in the current program (the constants would have to be converted to variables of course, requiring more RAM). The advantage of this is that the microcontroller would not require re-programming if these constants were changed. An example of a constant worth having dynamic access to is the GPS error margin value (specifying the error margin of coordinate fluctuations thought to be the same point to the program) and the distance threshold for the frontal distance sensor. The error margin could be changed as distance scales are changed if a larger or smaller margin is required by a user or the inaccuracy of a GPS fix (without DGPS for example). Extending this idea further, the options (and even previous waypoints) could be stored into EEPROM when the program is shutdown. Then, when the program is booted up or reset, the previous settings would be automatically restored, saving the user the hassle of having to enter numbers in each time power is applied and/or lost.

References

- "Calculating Distances Between Two Points", Geoscience Australia, http://www.ga.gov.au/nmd/geodesy/datums/distance.jsp

- "NMEA FAQ", Peter Bennett (peterbb@interchange.ubc.ca), http://vancouver-webpages.com/peter/nmeafaq.txt

For references on inverse trigonometric functions and fast integer square root, please refer to the references in the source code listing for the custom collection of math subroutines.

Appendix A – Program Listings

The following listings are included:

- **Everything**: this is the project's main program
- **Math**: this contains all the custom written math subroutines
- **Servo test**: this was the software calibration program to obtain the servo pulse times for correct steering and motor control.

Appendix B – Data sheets for all components

The following data sheets are included:

- RS-232E Line conversion chip
- BasicMicro Atom PIC
- Rojone Genius GPS Receiver
- IR Distance Sensor
- Wireless Transceiver

The required wiring information for the LCD and keypad was found in the PHYS2601 laboratory manual.

Appendix C – Reference material

The HSERSTAT command for polling the status of the Atom’s UART:

```
HSERSTAT cmd{,label}

CMD:
0 Clear Input buffer (label not used)
1 Clear Output buffer (label not used)
2 Clear both buffers (label not used)
3 If Data available in input buffer goto label
4 If no Data available in input buffer goto label
5 If Data is waiting to be sent goto label
6 If no Data is waiting to be sent goto label

46 bytes for input and 46 for output along with some system
variables (2 each).
```