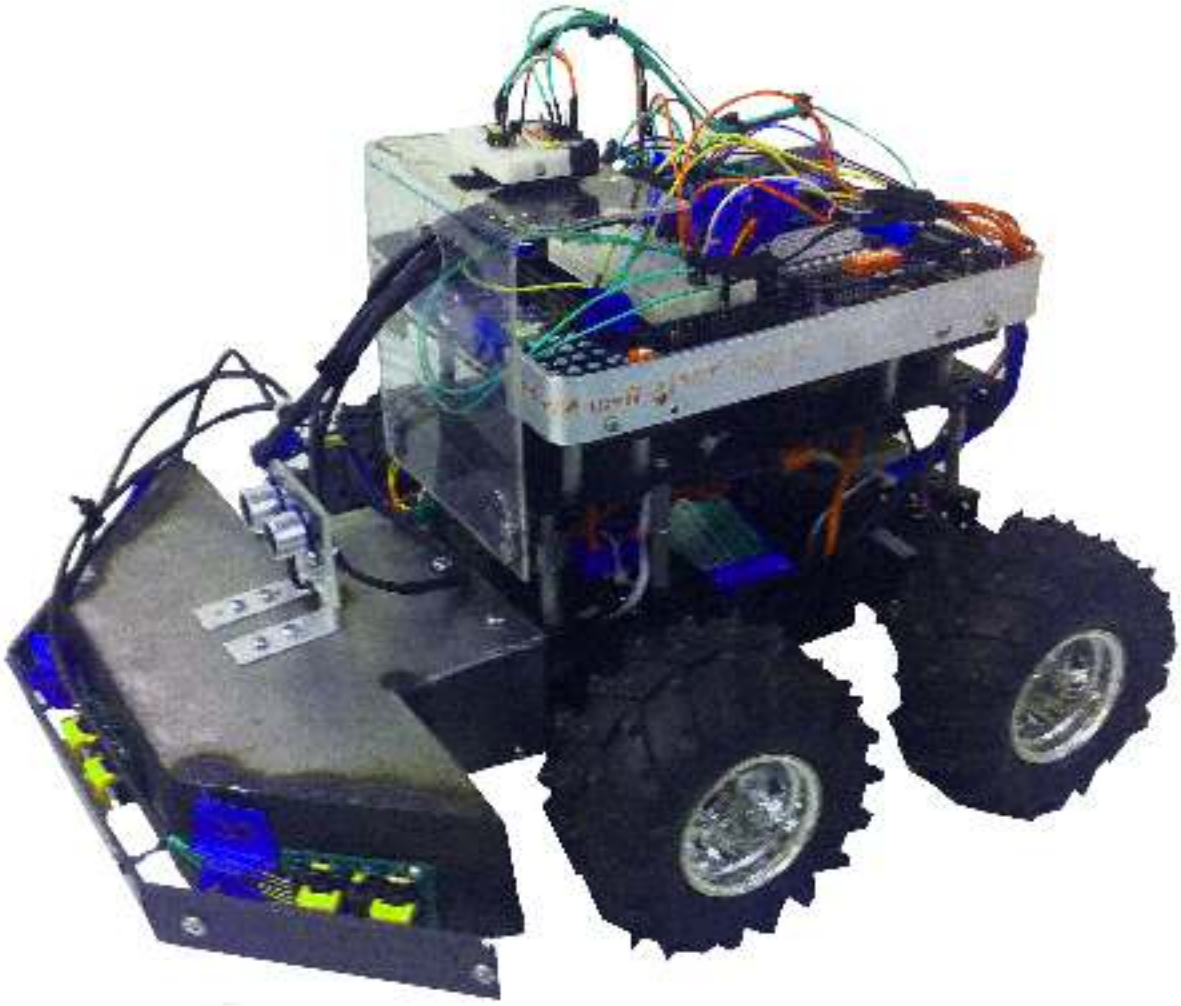
INTRODUCTION TO DESIGN FINAL PAPER



By TEAM A:
Alan Benalil, AaronBentley, Charles Bernson,
ChrisYelton & Deepak Rai

EE382

SPRING 2011

ELECTRICAL ENGINEERING

New Mexico Institue of Mining and Technology

NEW MEXICO TECH

SCIENCE • ENGINEERING • RESEARCH • UNIVERSITY

Alan Benalil, Aaron Bentley, Charles Bernson, Deepak Rai, Chris Yelton

Dr. El-Osery, Mr. Tubesing

EE 382

6 May, 2011

Team A: Junior Design Final Report

I. Introduction

A. Project Description

For this year's Junior Design objective, we were given the daunting task of designing an autonomous robot that would be able to navigate through a designated course to locate five unique waypoints, each surrounded by a red, 3 meter radius circle. It was our job to have the robot first independently navigate to a circle. Once the robot got to a circle, we needed to devise a strategy for the robot to be able to locate the object in that particular waypoint. The robot needed to both locate the object and come within one meter of that object. Once that robot reached its destination, it would have to somehow let the observer know that it had found its goal. For example, we programmed our robot so that when it found what it was looking for, it would back up and spin in a 360 degree turn. After the robot let the observer know it had accomplished that particular leg of the course, it would move around the object and go on its way to the next waypoint until it reached the end of the course. A further requirement was also incorporated into our design: each team was given a maximum spending limit of $750 to keep the robots on approximately even ground as far as parts went. Of that $750, $125 was allotted to every team to spend as they needed, such as on assorted parts like sensors and batteries. These funds came from the EE department, and could be spent on whatever we needed. At the beginning of the project, we were given a chassis, a pair of H-bridge motor controllers, and a

6.6V RC battery; these formed the base of our robot. In order to both communicate with the robot and incorporate sensors and navigation tactics, we used a HC9S12 microcontroller. Using a microcontroller gave us the freedom to control the robot as we wished. Once we incorporated the microcontroller, we devised a way to navigate the course with methods such as dead reckoning and the use of a global positioning system (G.P.S.) unit, along with a way to locate an object in a waypoint such as a sensor network.

B. Course Description

The course itself was located outside of the Workman building as seen in Figure 1 below. From the starting line, the robot has to run along the side of Workman. For this leg from the start to waypoint one, dead reckoning seemed to be the best navigation approach. Once the robot reached the red circle, a color sensor would be triggered to let the robot know it is close to its destination. At this point the robot would run a search algorithm to find the object. In the beginning, we only knew that waypoint one was going to be some object; at the final evaluations we learned the object was an orange traffic cone. The journey to waypoint two was also accomplished by dead reckoning. At waypoint two the object was a bucket of ice. From waypoint two to waypoint three, we planned to implement a combination of dead reckoning and G.P.S. We decided to use the G.P.S. unit for waypoint three, four, and five because that area was open and we would not get much interference from the building. At waypoint three we were given the information that the object was a 40kHz ultrasonic beacon. We could use different sensors to find the beacon. For waypoint four, during the final evaluation, we were given a relative location. We could program that location into our microcontroller to have the robot go to that exact spot. Finally, waypoint five was another unknown object, later revealed to be a bucket of paint.

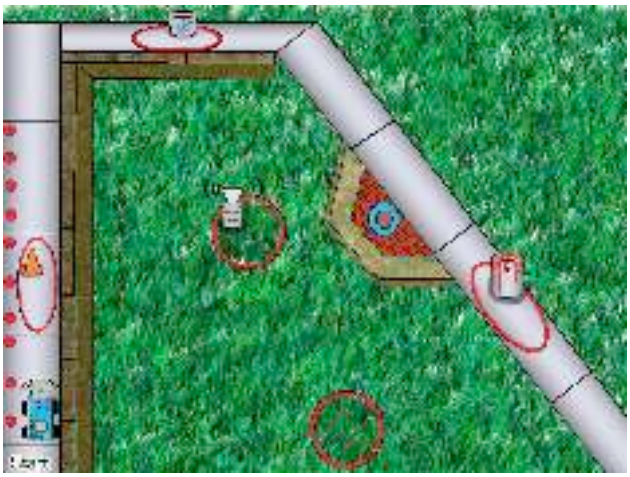


Fig. 1: Course Layout

One of the challenges we ran across was the difference between indoor testing verses outdoor testing. For indoor testing, we had control of what the robot would encounter. Because of this control, we found a lot of new challenges awaiting us outside. One of the major obstacles was how the robot would preform on grass verses pavement. The pavement was a lot like the floors inside, but when the robot ran on the grass, the robot experienced slippage and drag on the wheels. Because of this, we had to raise the duty cycle in the grass. Another common occurrence was the problem with ambient light. The color sensor we used turned out to be very sensitive and ambient light gave us a lot of problems. We had to design a way to fight that interference. In the end, we were able to design around the hurdles outside testing gave us.

C. Initial Research

Our first assignment of the semester was to research autonomous navigation in order to gain a better understanding of what was being asked of us in this project. We were asked to perform a case study of autonomous navigation and to research what had been done in the field previously, including different methods as well as the hardware and software used to implement such methods. Throughout our research, a few main methods stood out to us as possibilities for implementation in our project, the first being the use of vector potential fields for navigation.

This method employs computing steering angles and velocity at each instant on the relative location of the desired waypoint. Selecting the path that needs the least adjustment to steering angle and velocity then optimizes this, leading to a very precise navigation scheme. The same idea can be used for obstacle avoidance, but unfortunately this requires us to generate a map with precise curvatures of our course. This course mapping was highly unfeasible since it would take a large amount of manpower to create such a map. Therefore, we promptly scrapped the idea.

The next method we researched was full dead reckoning. Dead reckoning is a positioning scheme that bases current position on past known positions, for example, counting wheel revolutions while knowing how far each revolution moves the robot as it proceeds forward. This method seemed feasible, but further online research explained it to be quite inaccurate over longer distances. Although it had the potential to be inaccurate, we still kept the idea of using dead reckoning for our navigation due to its simplicity of implementation compared to other researched methods.

The next method we looked into was G.P.S. Just as the name suggests, this method moves the robot based on precise global positions measured in precise earth coordinates using satellites. This method seemed very applicable and possible for our project, but since G.P.S. signal was not guaranteed, we were not completely sure we would be able to implement such a system.

The last method we looked into was Image Processing and Path Planning. This method involved taking a digital image of the area and allowing the robot to transform the image into something it could understand. Using edge detection, the robot could then avoid any obstacles seen on the course in the processed image. This method, however, requires immense processing power, which made it a less attractive approach to our project.

Using this research to expand our knowledge of the field of autonomous navigation, we were able to look at what had been done previously to accomplish tasks, including what systems were used, any detections sensors used, as well as navigation sensors. Using the research, we were able to narrow our navigation scheme down to a combination of dead reckoning and G.P.S. navigation. This would give us the simplicity of dead reckoning, which was accurate over short distances, as well as the precision of G.P.S. when signal was available. We also looked into detection sensors, both for obstacles and for desired targets. These ranged anywhere from digital cameras to ultrasonic rangefinders and proximity sensors, all of which were considered for our original design. Along with detection methods, our navigation methods would require a few key sensors that could supply navigation data. Research allowed us to explore the possibilities of implementing sensors such as digital compasses, G.P.S., digital cameras, optical encoders, and many others.

Armed with a set of ideas based on our research, we began a battle plan of sorts. This plan changed much over the course of the semester as we slowly learned what would work and what would not. Initially, however, we planned to use a combination of G.P.S.-based navigation and dead reckoning navigation. Our G.P.S. unit would tell us where our robot was and where it had to go, and dead reckoning would allow us to choose a heading and go a certain distance toward our target. Once the robot reached the target area, a sensor suite with sensors such as ultrasonic receivers, temperature sensors, and color sensors would allow us to find objects within that area. Because we were given the layout of the course in the beginning, we planned to measure compass angles, G.P.S. latitudes and longitudes, and distances within the course. The robot could then be programed with these parameters and could proceed from circle to circle in a prearranged sequence. We also wanted to integrate all our systems in such a way that the entire

robot could run off of a single power supply, namely, the 6.6V RC battery that was provided at the beginning of the semester. This would eliminate the need for other batteries in the project, as the RC battery is rechargeable.

II. Subsystem Descriptions

A. Power Systems

To run a robot and all its subsystems, we needed to properly power everything. From the start we were given a 6.6V battery. This battery would be the key to running the motors and H-bridges. With power for the motors and H-bridges, we had a robot that could go in a straight line. This would be sufficient if we had been given the task of making the robot go in a straight line, but our objective was a bit more in-depth. We needed a way to not only provide power to the motors and H-bridges, but also to power the microcontroller and all our sensors. To do this, we had a couple different options. The microcontroller needed 9 Volts to run. This would provide us with enough power to run our microcontroller and any sensors we wanted to use. One option would be the quick and easy way of running multiple standard 9V batteries in parallel. This method is its very easy to implement and is reliable, however, we would either have to keep buying new batteries or buy a set of rechargeable batteries and then have to wait while they recharged. The second option would be to design and fabricate a power board, which would give us power to all our systems from one battery. We had to come up with a way to step up the 6.6V battery to 9V to supply power to the motors, H-bridges, microcontroller and all our sensors. To do this, we designed a switching power supply. We used the LM3578 integrated circuit from the Advanced Electronics lab. This LM3578 is just the switch; the rest of the circuit is built around it. In the LM37578's data sheet, there was an assorted array of different circuits for its function.

For our purpose we decided to go with the standard boost regulator configuration as shown below in Figure 2.

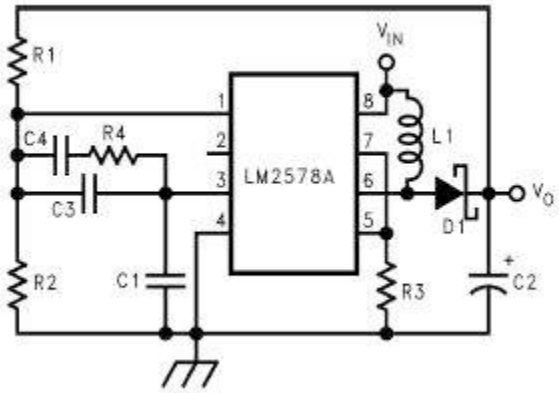


Fig. 2: Boost Regulator Configuration

We had to tailor all the discrete components to our specifications. We built the circuit on a bread board and tested the output. After a few modifications we where able to get the desired voltage on the oscilloscope. When the bread board testing was done, we decided to fabricate a circuit board. We spent some time implementing the circuit with Multisim and Ultiboard. The final layout is shown below in Figure 3.

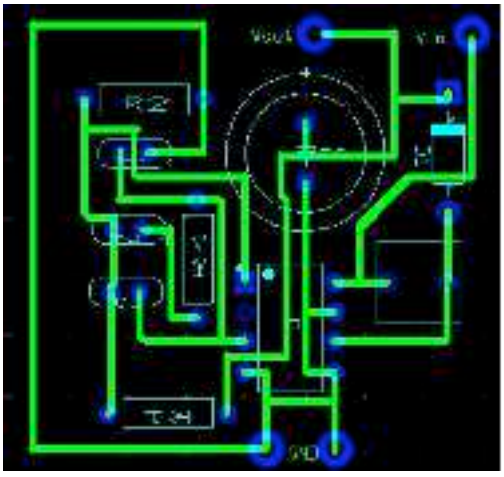


Fig. 3: Final Power Board Layout

When we finished the board layout, we went into tech room to etch the board. After our first attempt, we found out we etched the board backwards. After our second attempt, we found out the our traces were too small for drill holes and modifications. Our third revision turned out to be a success. We ran a lot of test running the microcontroller, H-bridges and motors with success. The next step was to combine the compass and wheel encoders. When we started running all other subsystems, the voltage on the power board was not able to sustain the constant voltage we needed. This resulted in the microcontroller resetting over and over again every time the motors pulled a large amount of current (especially when the motors first started up). We were given the suggestion to add a large capacitor to the circuit to filter the power. When we tested the power board our LM3578 chip burnt up, which we believed because the capacitor pulled too much current from the regulator. At that point we decided to run an array of 9V batteries in parallel. We wasted a lot of time and resources with the power board that could have been used to integrate other systems. Accordingly, our final implementation was to run only the motors and H-bridges from the 6.6V battery. All the sensors, microcontroller and navigation subsystems were run from the 9V batteries. For the compass and the G.P.S, we had to design a current limiting circuit, so that the unit would not fry. In the end all our primary and subsystems worked well on both batteries. The only drawback was the time we spent charging the batteries.

B. Microcontroller Selection

For our project, it was clear that we would need some form of controlling device that would act as the decision-making center of our robot. For this component, we decided to use the Dragon 12-PLUS Development Board that was provided to us last spring in our Microcontrollers course. We chose this particular board for a number of reasons, the first being that it was readily available; everyone in our group has one of the boards. This allowed us to work individually on

our own boards and having back up boards just in case of hardware problems. The second reason is that this board runs on the C programming language. Also, the D-Bug 12 system used to communicate and load programs onto the board made interfacing simple. Again, both of these technologies were used in our course last spring, allowing us to dive right into work instead of spending time learning a new system that might not even have satisfied our computing needs. Along with fulfilling our software needs, the board also provided us with sufficient hardware interfaces for our project with its bank of standard input/output ports, Pulse Width Modulation system, Analog-To-Digital Converter, Input Capture Output Compare Ports, and other timing and interrupt subsystems. Another factor constantly under our consideration was power and power consumption. We wanted to be sure that our design was easily powered and did not require much extra, if any, aside from the power the battery provided. Our controller fits our requirements very well and did not require large amounts of current. Although we had to deal with the issue of increasing our 6.6V source to 9V for the controller, but this task was minor compared to searching for another controller with such ease of use to us. In the end, we chose this controller due to the fact that we have used it in great details before. The fact that we have intricate knowledge of each of its individual subsystems saved us valuable time in that we did not have to educate ourselves on the systems of another device. Instead, this time could be put towards sensor selection and integration, helping keep us on track throughout the project. Along with understanding the systems, the fact that this controller can communicate through the C language allows us to avoid having to learn a new programming language. The prior experience with the C language allowed us to dive right into integrating our sensors once they arrived, instead of wasting precious time trying to learn how to communicate with a new language.

Shown in Figure 4 is a basic system structure diagram showing how the various subsystems tied into the microcontroller.

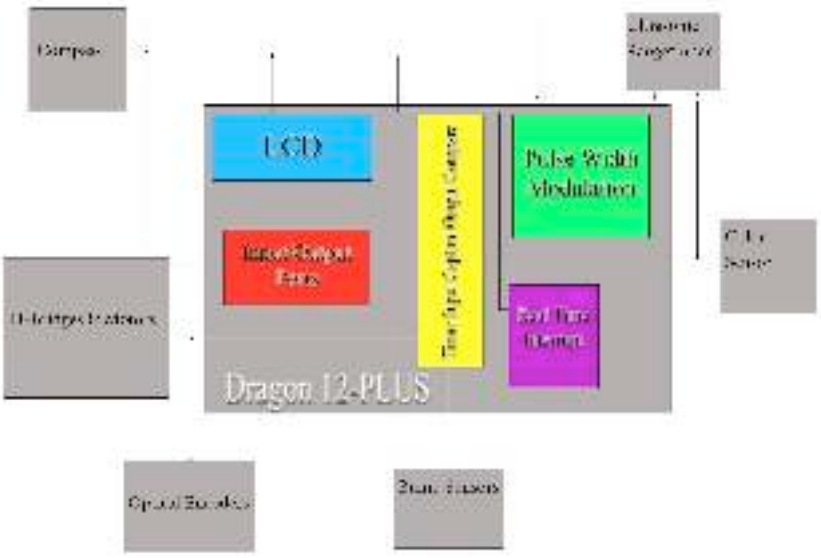


Fig. 4: System Structure

C. Navigation Scheme

For our final design, we chose to use Honeywell's HMC6352 Compass Module. We chose this device for a few reasons, the first being that it had relatively low power consumption. It was also fairly accurate, as close as 0.1°, giving us greater accuracy in our navigations scheme. The most important reason, however, was the compass's interface to the controller, the inter-integrated circuit (I^2C) interface. This is an interface that we have had much previous experience with before in our Microcontrollers course, allowing us ease of access to the compass's data. Since we have seen this type of serial communication before, it made it really easy for use to set up and run the compass on our microcontroller.

The HMC6352 requires typically 3 V to run, drawing typically 1 mA, however, in order to keep our logic levels consistent, we ran our compass at 5 V, inducing a draw of 2 mA. By

sending the compass specific ASCII commands, we can calibrate the device, put it into sleep mode, or request a continuous data transfer. The two main commands we used were the “C” command, which puts the compass in calibration mode, and the “A” command, which requests that the compass send heading data to the master device via the I^2C bus. In order to check the compass's operation after proper implementation, we had the compass display its data on our controller’s LCD screen and measured it against known readings to check its accuracy. Within the code, we implemented a Real Time Interrupt that would then update the reading on the LCD constantly, as well provide the controller with the data itself, stored in a global variable. This data could then be accessed by any subroutine or function inside the main allowing the robot to put the compass heading to use. In order to apply this data to navigation, we first had to decide how we wanted to implement the compass as a navigation sensor. We decided that it would be best if we could give the robot a desired heading, and have it follow that as accurately as possible, such that it could maintain a straight line path. In order to do so, first we needed to robot to be able to understand where it was facing. Saving the compass data as a global variable accomplishes this task. Next, we had to give the robot a desired heading and the direction we wanted it to go. Inside our main function, we would then have the robot check its current heading against a desired heading. We began simply by giving the robot a desired heading, and writing a simple movement routine that would rotate the robot via skid-steering until it faced the desired heading. Next, we implemented a small routine that would have the robot follow a small, square path by having the robot drive forward, rotate 90^o, drive forward again, rotate again, and repeat infinitely. This rough program thoroughly illustrated the functionality and accuracy of the HMC6352 and proved its worth to our project. Although our robot was able to drive in a straight path over a short distance, this did not incorporate a process that constantly tracked the heading

and adjusted the robot's position such that the desired heading was maintained, once reached. This task was then implemented, first using a simple proportional control system. The robot would check its heading, subtract that value from the desired heading, and then adjust the Pulse Width Modulated signal that controlled the motor speed such that the robot would rotate in the necessary direction to correct its offset. This worked over short distances spanning one to five meters, but soon the robot would fall off course and drift to one side. In order to overcome this, we looked into adding an integral control system that would increase our current system's accuracy, however, due to hardware issues, we never achieved. One of the first problems we had with our compass was the fact that our microcontroller's 5 V references did not supply the necessary current to run the compass. In order to overcome this, we designed a current/voltage limiting circuit shown below in Figure 5 that gave the compass the necessary voltage and current by drawing it from the actual source for the controller.

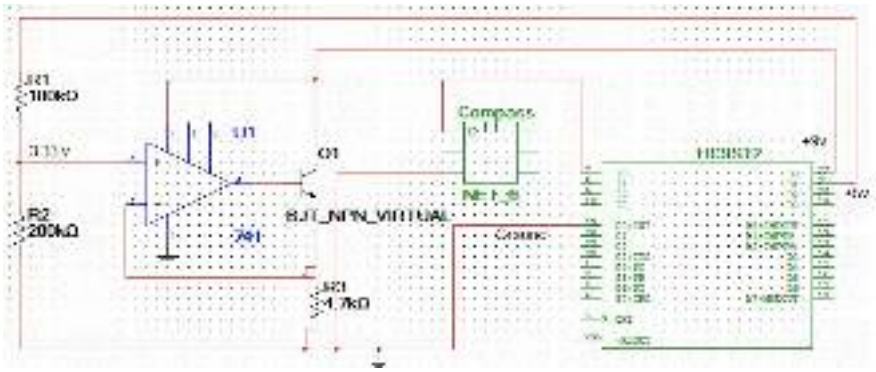


Fig. 5: Compass power supply

Once the power issue was resolved, we could then properly run our compass on the microcontroller's evaluation board without a problem. Our next issue proved disastrous to our compass. While attempting to implement our new Proportional Integral (P.I.) controller, a power surge in our lab caused a large spike in current through the microcontroller source. Normally, a large capacitor and a voltage regulator shield the controller's internal components, and anything

connected to it. With the compass being connected at the input terminal however, it did not receive the safety of the shielding capacitor, and the surge destroyed the internal workings of the HMC6352. This occurred the week before our final design evaluation, putting our team under a lot of strain. Luckily, we were able to get a compass shipped overnight to us so that we could continue our work, but unfortunately we lost a good few days worth of time before we realized that the compass had died. We obtained our next compass a few days before the evaluation of our final product, but unfortunately, after calibrating the new component, the data received from the device lost all usability and essentially was worthless. The final status of our compass was operational in the respect that we could communicate with it, but failed in the aspect of providing accurate, much less usable data in any way.

Our initial plan for navigation was to use both dead reckoning and G.P.S. data from a receiver, and for the robot's heading to be corrected with the use of PWM signal from our microcontroller that controls motor voltage based on the digital compass readings. The robot continuously tracks its position and orientation with respect to its desired position and orientation, and the control system would guide the robot towards it destination.

The main reason we chose to use G.P.S. for localization is because the data obtained from it does not depend on previous readings and therefore errors in localization do not grow over the course of time (as it would with wheel encoders in dead reckoning). The main disadvantage of using G.P.S. is its dependency on its surrounding. We had problems with the accuracy of G.P.S. data when there was obstruction in between the receiver and the satellites it receives the data from.

The G.P.S. module we used was a Fastrax UP501R. We chose this module because it was within our budget range, it updates at 10Hz, it has a relatively low power consumption, and it

gave us the resolution of 1.8 meters that was sufficient for our project requirement. The precision given by the G.P.S. manufacturer was not as exact as our G.P.S. usage, because the conditions that they were tested in were not ideal. Moreover, G.P.S. is a passive receiver that updates its positioning every second, and in our case if the robot speed was too fast, it would have created a problem of localization accuracy.

Figure 6 below shows the interface connections of G.P.S. with the microcontroller. The first task we had was to power our G.P.S, which needed voltage between 3-4 V. Since there was no reference voltage in microcontroller that was within the range to power our G.P.S., we designed a voltage follower and current limiter circuit with an appropriate voltage divider that gave us the voltage output of 3.33V to power our G.P.S.. We then setup the serial communications interface 1 (SCI1) pin in our microcontroller to establish communication between G.P.S. and microcontroller. We then connected the Pin2, TXD of G.P.S. to port PS2 of SCI1. Initially we had problem getting data from G.P.S. to microcontroller. The complier, Code Warrior that we were using was not compiling the interface program correctly in EE digital lab, so we downloaded Code Warrior in our personal computer and verified that it worked by writing and compiling a simple program that turned LED in DRAGON12.

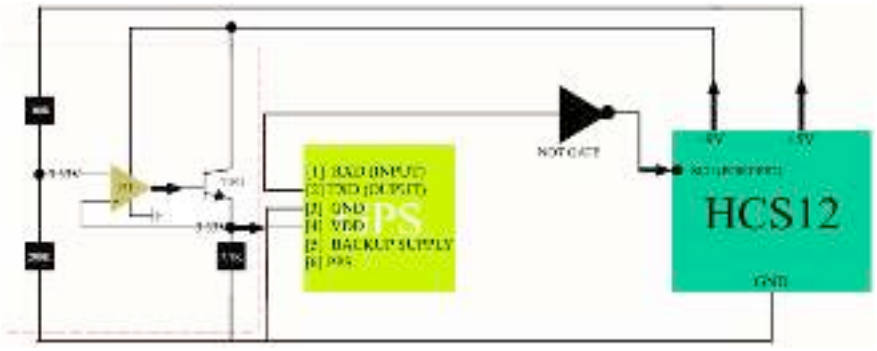


Fig. 6: G.P.S. Power Layout

We then used a logic probe to verify that we were getting data from G.P.S. After knowing that we were getting data from G.P.S., we tried to interface the G.P.S. with microcontroller, but we were not able to get any data displayed in HyperTerminal that we were reading from G.P.S.. After revising the interface code for few more times, we then inspected the output signal that G.P.S. was outputting with the oscilloscope. The data output that we recieved had the voltage reading of 10Vp-p, but our microcontroller can only read the logic between 0-5V. In order to get rid of the negative voltage, we used a NOT-gate which fixed our problem.

The G.P.S. was outputting four different kinds of NMEA data strings listed below:

$GPGGA,035958.000,3403.7851,N,10654.3480,W,1,4,2.06,1427.3,M,-24.1,M,,*53

$GPGSA,A,3,28,24,08,17,,,,,,,,,2.29,2.06,0.99*0D

$GPGSV,3,1,10,17,74,251,18,28,60,026,25,08,54,138,24,26,35,257,*78

$GPRMC,035958.000,A,3403.7851,N,10654.3480,W,0.46,199.17,280411,,,A*73

We decided to use GPGGA, NMEA data string which had all the information that we need to obtained longitude and latitude for navigation. To obtain the NMEA data string from G.P.S., we started by writing an interrupt with case statements that would detect the NMEA data string that we decided to use. When the designated NMEA data is detected then the program stores the string in a buffer. In order to parse the data out of the string that is highlighted above, we defined a variable for each character with its location in the string and once the program detects an end character '*', it extracts each data character from the NMEA data string that is stored in the buffer and passes it to another global variable which makes the data from the G.P.S. accessible to be used for navigation. After we verified that we were getting the right data out of the G.P.S., we then displayed the G.P.S. data in the DRAGON12 LCD. The final status is that the G.P.S.

subsystem works by itself but unfortunately we were not able to integrate the G.P.S. in our final design due to the lack of time.

The final components in our navigation scheme were the wheel encoders. In our Microcontrollers class, we had a series of labs that used input capture and pulse width modulation to control and record speed and distance of a motor with an encoder wheel. By the end of the lab set, all of us had developed expertise in using these wheels. For this reason, when we saw that, for this project, we would have to keep track of how far our robot had gone, the wheel encoders from the Microcontrollers lab were the first things to come to mind. After checking prices, availability, and potential usability, we decided to use the H21LTB-ND, the same optical encoder we used before in the lab. This subsystem was made operational early in the semester and proved to be rather robust. It was used in the final design and was working up to and past the final evaluation.

The H21LTB-ND, manufactured by Fairchild, is an encoder that uses a GaAs light emitting diode (LED) and light sensor. When the sensor "sees" the signal generated by the LED, the package generates an output signal, either a logical high or a logical low, depending on how the user has set up the device. For our project, we set the device to give a 1 for when the sensor "sees" the LED. Following the model of the lab motors, we fashioned 2 encoder wheels out of aluminum, each with 15 holes equally spaced around its perimeter and with one hole in the exact center. The wheels were then attached with epoxy directly to the robot's wheels' axels, one on each of the back wheels. This ensured that with every wheel rotation, there would also be one complete rotation of the encoder wheel. We then constructed brackets to mount the optical encoders themselves to the motors; the suspension made it such that if we had mounted the encoders to the chassis, there would most definitely have been problems with the encoders

picking up the wheels. With the encoders and the encoder wheels in place, the encoders could shine through each of the holes in the wheels as it passed by as the robot was moving. This generated a high on the encoder's output line when the holes on the wheel passed through the encoder, and a low when the solid part of the wheel passed through.

The encoders drew little enough current that we were able to power them directly from the microcontroller (as with many of our other systems); the supply lines went to +5V and ground on the board, and the signal line went to one of the input capture ports. Code was set up that enabled the port and set the system to trigger an interrupt on a rising edge, whenever the encoder saw a hole on the wheel. The interrupt (refer to the Wheel Encoders Interrupts and Subroutines section in the final code in the appendices) first checked to see how many holes had been counted in the past. If 15 holes had been counted, a global variable assigned to the number of wheel rotations was incremented and the hole count was reset to zero. If the hole count had not yet reached 15, the hole count, which was also a global variable, was incremented. In this way, every time the wheel turned 24 degrees and reached another hole, the controller could keep count, and when one full rotation was reached, the number of rotations could be recorded and updated. This simple interrupt routine allowed us to keep track of how far our robot had traveled. We could also give very precise commands about when to start, stop, or turn based on how far the robot had gone by polling the global wheel count variable. Finally, we could use the encoders to do a fair amount of adjusting our course to stay in a straight line by comparing the number of hole counts on one side with the number of hole counts on the other. An error was calculated from this and the speed of both motors was adjusted accordingly to keep the robot on course.

D. Waypoint Detection

Once we had navigated to the general area of each waypoint, our plan was to use a small group of sensors to navigate our robot inside the actual circle and to find the actual waypoints inside the circle. To accomplish this, we used a color sensor, an ultrasonic range finder, and three bump sensors, which were all powered by the microcontroller. Once the actual waypoints had been found navigation takes over again to navigate to the next waypoint area.

The first thing to be used once in the general area of the waypoint is to sense the red circle around each of the waypoints. To do this we used a color sensor, the OPB780Z reflective color sensor manufactured by TT Electronics, which was made with four different filters for red, green, blue, and clear. The sensor has three power pins, two for the sensor itself which require 5V and one for the attached LED which required 3.3V, two ground pins, two signal pins used to switch between the filters, and one output pin.

The sensor outputs a certain frequency for each color and the frequency of each color is usually the highest when sensing the color that the filter is selected to detect. The output pin of the sensor was connected to an input capture pin of the microcontroller so that the frequency of the light waves could be detected. Instead of reading frequency, the sensor was set up to read time, which meant that the color to be detected will have the lowest time value. To get an output that makes sense, the time had to be output in microseconds.

To use the color sensor, we made a box at the front of the chassis of the robot to cover the sensor from ambient light interference. To also cut down on interference, we had make a shroud out of pieces of rubber that hung all the way to the ground and acted like mud flaps on a vehicle. This was done so that the color sensor would be able to detect the red line around all of the waypoint areas while the robot was on the move. Once the robot had navigated into the waypoint

circle, the color sensor was to be used to keep the robot inside the circle until the actual waypoint had been detected. This was done the same way as it was to find the circle; the only difference is that the robot will be navigating in straight lines and turns only.

Once inside the circle, we used the ultrasonic range finder and the bump sensors to detect the waypoint object. The ultrasonic range finder, the Parallax #28015, was set up on an input capture pin to detect the output that was in microseconds. The sensor has a 40^{o} beam width, 20 degrees on either side of the receiver, and has a distance range of two cm to three m. The range finder required three pins from the microcontroller: 5V, ground, and input/output.

One of the pins on Port A of the microcontroller had to be switched back and forth from an input pin to an output pin so that a high signal could be sent to the sensor for about five microseconds to activate the range finder. When the pin went low there had to be a delay of about 750 microseconds for the hold time. After the delay, the sensor sent out a 40 kHz burst that would go out and bounce back from any object in its path. The time for the burst to leave and return is proportional to the distance the object was from the sensor. We were able to determine the amount of time that it took for the burst to travel 1 meter and return so that we would be able to tell when our robot was within 1 meter of the waypoints. Figure 7 below shows how the input and output signal of the range finder works.

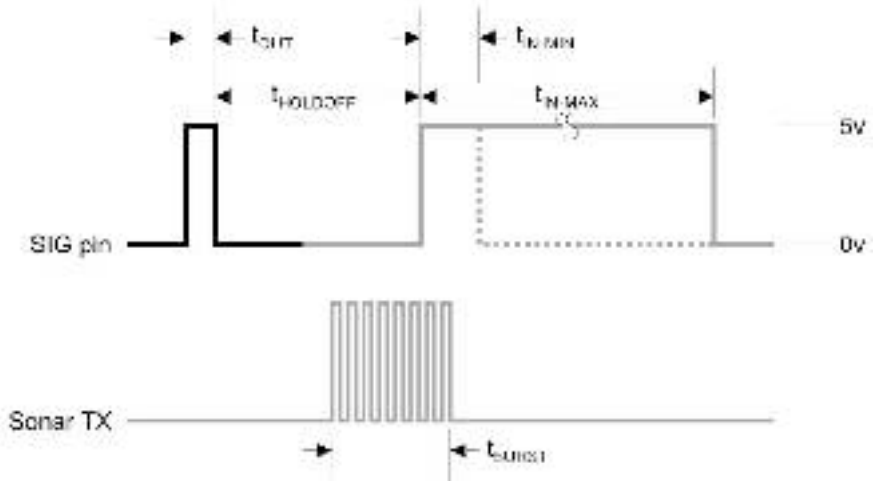


Fig. 7: Ultrasonic range finder input/output pulse

For the waypoints where we were not able to use the range finder (i.e. the waypoint containing the ultrasonic beacon), or if the object was not completely in the path of the range finder, as well as for object detection for the rest of the course, we attached bump sensors to the front of the robot to detect any waypoint the robot may run into. We made these bump sensors from the debounced switches we made in the Digital Electronics class two years ago. These sensors require three pins from the microcontroller: 5-volts, ground, and input/output. To make sure that the bump sensors would be able to detect anything that the front end of the robot hit, we fixed some rigid plastic to each switch that stuck out far enough to completely encase the entire front end.

Just like the last two sensors, the bump sensors were set up on an input capture pins. But for these sensors, the amount of time between pulses did not matter; all we needed to know was that the signal had gone high. When the signal went high, the robot would perform the move programmed to show that it had found the object. These were also used for object avoidance in

the rest of the course. So the input capture interrupt was just set to let the robot know when the signal from one of the three bump sensors goes high to do a certain action depending on which sensor went high. (See the example further on in the paper.)

By the end of the semester, we had no problems with the range finder or the bump sensors, they worked just as we had planned. The only problem we had with any of the sensors was trying to block out all of the ambient light from the color sensor. We tried shrouding it with many different materials and in many different ways, but could not ever get all of the ambient light blocked. We were going to try a couple of other ways to block out the light but just did not have enough time to implement the changes. The bump sensors worked as they were supposed to- every time the robot ran into something it would respond appropriately. The range finder worked as expected as well, with the only malfunction being that we did not get the exact values of time versus distance that was specified in the data sheet. The only sensor that did not work was the color sensor. We had no problems getting it to work indoors, but outside the UV radiation from the Sun had a huge effect on it and it could not tell the difference between any colors.

E. Other Sensor Options

Because we had sufficient funds and because we were originally feeling so ambitious, we tried to incorporate other sensors into our design. These, for one reason or another, were not included in the final product. The first of these was the Parallax MLX90614 Infrared Thermometer. Obviously, this was our original plan on how we would find the cold object in waypoint 2, as already described. The MLX90614's data sheet boasted a simple interface: single wire serial. The breakout board was also set up with a detection system that automatically matched its baud rate to the microcontroller's. Once the part arrived, we began work to get it operational.

Switching the microcontroller's serial system to single-wire operation was a simple matter of writing values to the right registers. Writing a code to communicate to the IR sensor was also easy, as Dr. Rison had already made portions of serial communications code available to us when we were studying the subject in our Microcontrollers course. However, despite the seemingly simplistic interfacing and having a large portion of the code written for us, we soon ran into problems. Probing with a logic analyzer showed us that though we were "saying" the right things to the IR sensor, the IR sensor was returning either meaningless data or nothing at all.

We spent nearly three and a half weeks trying to get the sensor to work before our instructor (who was trying just as hard as we were to get the sensor to work) suggested we put the subsystem on the back burner, as it were, and to return our attention and efforts to our other systems. Because we were short on time, we never had a chance to return to the possibility of using the IR sensor, but we were able to compensate for its absence in our design by deciding to simply collide with the cold object to detect it.

During our first sensor selection process, one particular compass module that stuck out to us was Hitachi's HM55B Compass Module. This device seemed to provide everything we were looking for in our compass module, low power consumption, and fairly accurate resolution readout. Unfortunately however, we did not look into the interfacing in great detail, and once we received the device, we realized that we lacked prior knowledge of how to interface with the device. We referred to the data sheet to try to gain information as to how to pull data from the device, but the company we ordered from ran almost all of their software in the BASIC language, rather than the C that we were using. After searching extensively for calibration routines written in C and trying to translate BASIC to C, we decided that it was not worth our

time to try to get the device to work with our controller. Not long after, we found the slightly more expensive, but more accurate and easy-to-use HMC6352 previously described. This allowed us to continue on our course of designing an autonomously navigating robot.

III. System Integration

All of the sensors were integrated into the main program via subroutines. Even though we had no navigation working during the final evaluation, we were able to integrate all of the sensors into the motion code and use the motion code and were able to get readings from the range finder and the color sensor as well as run into something so the robot could perform its preprogrammed maneuvers. For the most part everything for the sensor code worked, we just had trouble with ambient light.

After all the data was acquired from the sensors through interrupts and saved to global variables, we could process the data and program different responses for the robot to perform based on that data. This was accomplished in our "main," the routine in the program that is run. Our aim in this project was to keep the main as uncluttered as possible by writing various subroutines and interrupts, including those mentioned earlier. With the large amount of the data processing going on in the subroutines, it was a significantly simpler matter to line up the right routines in the main. Most of the decision-making process was done by polling global variables that were output and updated by the low-level subroutines and interrupts, either in "if-then-else" statements or in "while" loops. In this way, raw sensor data was turned into useful information and this information was then used to decide which action the robot should take. The desired action was executed in the form of motion which was controlled by the H-Bridge controllers and carried out by the robot's four motors.

The Pololu "Dagu Wild Thumper" Chassis that the instructors provided at the start of the project was outfitted with four high-torque motors. We were also given two discrete MOSFET H-Bridges that were used to control the motors. These act almost as electrical current floodgates that give more or less current to drive the motor based on the duty cycle of an input pulse width modulated signal. Besides this signal and input power, there were also pins on the controller that controlled the direction of the motors. We made use of Port A, a general I/O 8-bit port, on the Dragon 12 Board to control the two direction pins. This allowed us to either drive both motors forward, both backward, as well as one forward and the other backward or vice versa. The PWM signal was also provided by our microcontroller, this time from the PWM subsystem built into the board. With these controls in place, motion control was simply a matter of writing code.

There are three different levels of motion subroutines in our final code: a low-level, a medium-level, and an upper-level. The low-levels give extremely simple commands to the H-Bridges, such as our "Both_Motors_Forward" routine. This takes a PWM duty cycle (in percentages) as an argument and performs simple math to calculate what number is that duty cycle percentage of the total PWM variable (we used 240 because it gave us ample resolution). The appropriate duty cycle is then assigned to the "PWMDTY" variables in the microcontroller's subsystem which in turn directs the H-Bridges to supply the appropriate amount of current to the motors. In addition, the subroutine sets both of the pins in Port A so that both motors will drive forward. We wrote similar subroutines that perform other simple actions, such as instructing the motors to both drive backward. From these basic building blocks, we wrote medium-level subroutines which ran the low-level blocks, either for a certain time duration based on delay routines or for certain compass headings. Finally, as can be inferred by the name, the high-level routines ran strings of medium-level and low-level subroutines. These performed very specific

tasks, such as making the robot perform its victory dance (see subroutine "Dance_Dance" in the appendices).

To illustrate the complete process of how sensor data is used to make the robot perform an action, let us look briefly at an example utilizing the bump sensors. When the left bump sensor is pressed (therefore indicating a collision on the robot's left side), the sensor changes its signal output from a logical 0 to a logical 1. The input capture system on the Dragon 12 Board detects the rising edge and generates an interrupt. The program stops running and is sent to the interrupt vector for the left-side bumper input capture port and then to the interrupt subroutine we defined in the setup. In this routine, the interrupt flag is reset and a global variable is set to 1. The program then resumes cycling through the infinite loop in our main until it hits an if statement that checks if the variable for the left bump sensor has been set. Because it has, the program enters the if statement. Contained in this statement is a command to reset the global variable to 0, as well as a command to execute a high-level motion subroutine. Within this routine are function calls for the low-level backup routine and the mid-level turning routine. The robot is instructed to back up (away from the object it collided with) and then to turn a certain angle from on its current compass heading before proceeding forward again. With strings of different leveled subroutines and based on constant input from both the input capture interrupts and the real time interrupts, our robot was ready to run through the course.

IV. Results

For all our subsystems, we where able to design working program. For our navigation systems, all our code worked great. In the end our biggest drawback was the lack of time for

integration and implementation. We had the G.P.S. running great but ran our of time to include it into our final design. We also had problems keeping the robot on a compass heading.

We had a lot of problems with our compass. Our first one we could not communicate with. Our second one was destroyed due to a power outage and subsequent spike. Our final compass after a calibration routine we ran gave us garbage data. We decided not to use a power board in the end because of all the time we spent trying to get it to work with everything. Our color sensor worked great in a very dark environment or indoors- whenever there was little ambiance. The ultrasonic rangefinder worked as well, however we ran out of time time to implement it for the course. The wheel encoders and bump sensors did exactly what we designed them to do.

On Tuesday, May 3rd, all of the teams performed in the final functionality and design review. The course was arranged, the mystery items were revealed, and the exact location for waypoint 4 was set. All of the waypoint detection sensors that made it onto the final design were all up and running. The wheel encoders, too, were functioning as expected. On the other hand, as previously mentioned, our compass began to give false data a mere day before the evaluation, throwing a proverbial wrench into the proverbial gears of our navigation scheme. Also, our G.P.S., though giving reliable data as precisely as had been promised in the product specs, had not been integrated due to time constraints.

Without two-thirds of our navigation system integrated and/or operating properly, we were only able to make it to the circle of waypoint one, thanks to the wheel encoders allowing for corrections to keep a fairly straight heading. The bump sensors also were working properly, as repeated collisions with the poles and wall near waypoint one proved. The color sensor and the ultrasonic rangefinder provided data only as outputs on the microcontroller's L.C.D., not as

part of the detection system. Our run was short and we did not find any of the actual waypoint objects, but we did not see this failure to complete the course as equivalent to failing to complete the project.

V. Conclusion

The ability to navigate the course at the end of a semester of hard work is not necessarily an accurate indicator of success. Rather, whether or not we learned anything from the class should be the test, the final exam. Junior Design was a very interesting mix of school and "real life." On one hand, it is a class with instructors and assignments and grades, but on the other hand, there are elements of real-world engineering included, such as ambiguities in expectations, system research for practical use, and working with a team of people. That fact alone, working with a group of people, proved to be a new experience for us all. Each of us has a different personality, different areas of expertise and understanding, and different areas that he likes to work in, whether it be with software or hardware. The challenge often was learning how to distribute work so that multiple things could be done at once to maximize efficiency. This led to another important lesson we all took away from this semester: the importance of time management. The project could easily have been made into a two-semester, six-hour class due to the amount of work and dedication it took. Because of this, every day of research, lab work, coding, wiring, or welding counted; a day wasted was a day lost and made for a panicked day sometime further down the line. (The main reason we were unable to present a complete, integrated product at the end of the project was because we ran out of time.) The last thing we all took away from the class was the concept of money management. With an outrageous amount of money, outrageous things can be done. With a limited budget, however, real engineering comes

into play- how can we do something outrageous but for a cheaper price? We were forced to be frugal but also to be creative and to improvise. Perhaps with more time (and maybe a little more money), we could finish what we started and make it all the way to waypoint five, but we believe that we have already successfully completed the objectives of the EE 382 course.

Table of Contents

Table of Figures

Appendix A: Main Code

```
/************************************
EE 382: Junior Design - Group A
Dr. Aly El-Osery and Andrew Tubesing
Jan 22 - May 5 2011

Group Members:
Aaron Bentley
Deepak Rai
Charles Bernson
Alan Benalil
Chris Yelton

Updated: Apr 14 2011
************************************/

/* Main file combining compass and wheel encoders with PWM control.
   Revised 4/14/2011, adb
*/

/* INCLUDE FILES */

#include <stdio.h>
#include <hidef.h>
#include "derivative.h"
#include "vectors12.h"
#include "lcd.h"
#include "math.h"
#define enable() __asm(cli)
#define disable() __asm(sei)

/* GLOBAL VARIABLES DEFINITIONS */

short PWM_Total = 240;
int global_input_capture_left_var;
int global_input_capture_right_var;
int global_left_wheel_rev;
int global_right_wheel_rev;
unsigned int heading;


/* FUNCTION DEFINITIONS */

void iic_init(void);
void iic_start(char address);
void iic_response(void);
void iic_stop(void);
void iic_transmit(char data);
void iic_swrcv(void);
unsigned char iic_receive(void);
unsigned char iic_receive_m1(void);
```

```
unsigned char iic_receive_last(void);
void delay(unsigned short num);
char* hex2bcd(unsigned int x);
unsigned int get_heading(void);
void delay_1ms(int n);
void delay_50us(int n);
interrupt void rti_isr(void);
void Forward_Motors(float DUTYCYCLE);
void Backward_Motors(float DUTYCYCLE);
void Stop_Motors(void);
void CCW_turn(float DUTYCYCLE);
void CW_turn(float DUTYCYCLE);
void interrupt input_capture_left(void);
void interrupt input_capture_right(void);
void calib(void);

/* MAIN */

void main(void)
{
  iic_init();         // initialize I2C and set slave address

  disable();    // disable all interrupts

  /* Real Time Interrupt Setup */
  UserRTI = (unsigned short) & rti_isr; // define real time interrupt
vector
  RTICTL = RTICTL | 0x6F;
  CRGINT = CRGINT | 0x80;
  CRGFLG = 0x80;

  /* Pulse Width Modulation Set Up */
  PWMCTL = 0x00;            //8-bit mode
  PWMCAE = 0x00;            //left-aligned
  PWMPOL = 0xFF;            //high polarity
  PWMCLK = PWMCLK & ~0x30;  //select clock mode 0 for Ch 4 & 5
  PWMPRCLK = 0x01;          //Divides bus clock by 64
  PWMPER4 = PWM_Total;
  PWMPER5 = PWM_Total;
  PWME = 0x30;              //enable PWM on Ch 4 & 5

  /* Ports Setup */
  DDRA = 0xFF;          //Sets up Port A as an output
  DDRB = 0xFF;          //Sets up Port B as an output
  DDRP = DDRP | 0x0F;
  PTP = PTP | 0x0F;
  DDRJ = DDRJ | 0x02;
  DDRH = 0x00;          //Sets up Port H as an input

  /* Input Capture Setup */
  TSCR1 = 0x80; // turn on subsystem
  TSCR2 = 0x05; // set prescaler for 87.38ms
```

```
  // IC1 (Left Wheel) setup
  TIOS = TIOS & ~0x02;            // IOC1 set for input capture
  TCTL4 = (TCTL4 | 0x04) & ~0x08; // capture rising edge
  TFLG1 = 0x02;                   // clear IC1 flag
  UserTimerCh1 = (short) &input_capture_left; // set interrupt vector
for timer channel 1
  TIE = TIE | 0x02;               // enable IC1 interrupt

  // IC2 (Right Wheel) setup
  TIOS = TIOS & ~0x04;            // IOC2 set for input capture
  TCTL4 = (TCTL4 | 0x10) & ~0x20; // capture rising edge
  TFLG1 = 0x04;                   // clear IC2 flag
  UserTimerCh2 = (short) &input_capture_right; // set interrupt vector
for timer channel 2
  TIE = TIE | 0x04;               // enable IC2 interrupt

  enable();       // enable all interrupts

  //Stop_Motors();

  /*
  while(1){

  while(PTH == 0x00)
  {
    calib();
    __asm(swi);
  }
  Stop_Motors();

  }
  */
  /*
  while(1)
  {



    while((global_left_wheel_rev <= 1) || (global_right_wheel_rev <=
1))  //Sect 1: Move forward, rotate to 90 degrees
    {
      Forward_Motors(50);
    }
    Stop_Motors();
    delay_1ms(10);
    while(heading <= 900)
    {
      CW_turn(35);
    }
    Stop_Motors();
    delay_1ms(10);
```

```
    global_left_wheel_rev = 0;
    global_right_wheel_rev = 0;

    while((global_left_wheel_rev <= 1) || (global_right_wheel_rev <=
1))  //Sect 2: Move forward, rotate to 180 degrees
    {
      Forward_Motors(50);
    }
    Stop_Motors();
    delay_1ms(10);
    while(heading <= 1800)
    {
      CW_turn(35);
    }
    Stop_Motors();
    delay_1ms(10);
    global_left_wheel_rev = 0;
    global_right_wheel_rev = 0;


   while((global_left_wheel_rev <= 1) || (global_right_wheel_rev <=
1))   //Sect 3: Move forward, rotate to 270 degrees
    {
      Forward_Motors(50);
    }
    Stop_Motors();
    delay_1ms(10);
    while(heading <= 2700)
    {
      CW_turn(35);
    }
    Stop_Motors();
    delay_1ms(10);
    global_left_wheel_rev = 0;
    global_right_wheel_rev = 0;

   while((global_left_wheel_rev <= 1) || (global_right_wheel_rev <=
1))   //Sect 4: Move forward, rotate to 0 degrees
    {
      Forward_Motors(50);
    }
    Stop_Motors();
    delay_1ms(10);
    while((heading <= 3570) || (heading <= 20))
    {
      CW_turn(35);
    }
    Stop_Motors();
    delay_1ms(10);
    global_left_wheel_rev = 0;
    global_right_wheel_rev = 0;
    */
```

```
    /*Forward_Motors(80);       //This code works.//
    Stop_Motors();
    delay_1ms(50);
    CW_turn(50);
    Stop_Motors();
    delay_1ms(50);
    */
    /*Forward_Motors(80);       //This code works.//
    Stop_Motors();
    delay_1ms(50);
    CW_turn(100, 60);
    Stop_Motors();
    delay_1ms(50);
    */

    /*  Forward_Motors(50);    //This code works, too.//
    delay_1ms(100);
    Stop_Motors();
    delay_1ms(100);
    Backward_Motors(87);
    delay_1ms(100);
    Stop_Motors();
    delay_1ms(100);
    CCW_turn(200, 30);
    Stop_Motors();
    delay_1ms(100);
    CW_turn(150, 70);
    Stop_Motors();
    delay_1ms(100);


  }
 */

}

/* COMPASS INTERRUPTS AND SUBROUTINES */

interrupt void rti_isr(void)
{
    heading = get_heading();

   /************************/
    /* Displays value of   */
    /* compass readout to  */
    /* LCD display         */
    /************************/

      put2lcd(0x01,CMD);   //Sends lcd to first line
    delay_1ms(2);
    puts2lcd(hex2bcd(heading));    //Pass data to LCD display
```

```
    delay_1ms(5);

    CRGFLG = 0x80;

}

void iic_init(void)
{
  IBFD = 0x23;    //Set 100kHz Operation
  IBAD = 0x42;    //Slave address 1
  IBCR = IBCR | 0x80;    //Enables I2C bus
}

void iic_start(char address)
{
  while((IBSR & 0x20) == 0x20);    //Waits for IBB flag to clear
  IBCR = IBCR | 0x10 | 0x20;    //Sets XMIT and MASTER mode, sends
start
  IBDR = address;
  while((IBSR & 0x20) == 0);
  iic_response();
}

void iic_response(void)
{
  while((IBSR & 0x02) == 0);    //Wait for IBIF to set
  IBSR = 0x02;    //Clear IBIF
  if((IBSR & 0x01) == 0x01) iic_stop();   //sends NACK
}

void iic_stop(void)
{
  IBCR = IBCR & ~0x20;    //Releases the bus
}

void iic_transmit(char data)
{
  IBDR = data;    //Writes data to data register
  iic_response();   //Waits for response
}

void iic_swrcv(void)    //Sets I2C to receive data from compass
{
  volatile char dummy;    //Dummy variable

  IBCR = IBCR & ~0x10;    //Set to recieve mode
  dummy = IBDR;   //Dummy read to start serial clock so slave will
send data
}

unsigned char iic_receive(void)
{
```

```
  while((IBSR & 0x02) == 0);   //Waits for IBIF to be set
  IBSR = 0x02;    //Clear IBIF bit
  return IBDR;    //Read data and start next read
}

unsigned char iic_receive_m1(void)
{
  while((IBSR & 0x02) == 0);   //Waits for IBIF to be set

  IBSR = 0x02;    //Clear IBIF bit
  IBCR = IBCR | 0x08;

  return IBDR;    //Read data and start next read

}

unsigned char iic_receive_last(void)
{
  while((IBSR & 0x02) == 0);   //Waits for IBIF to be set

  IBSR = 0x02;    //Clear IBIF bit
  IBCR = IBCR & ~0x08;   //Re-Enable ACK for next read
  IBCR = IBCR & ~0x20;    //Generate STOP by switching to slave mode

  IBCR = IBCR | 0x10;   //Set transmit to reading IBDR without clock
start

  return IBDR;    //Read data and start next read

}

void delay(unsigned short num)
{
    volatile unsigned short i; /* volatile so compiler does not
optimize */

    while (num > 0)
    {
     i = 2000;
                                          /* ----------------------
-   */
     while (i > 0)              /*
*/
     {                                    /* Inner loop takes 12
cycles */
          i = i - 1;            /* Execute 2000 times to       */
     }                                     /* delay for 1 ms
      */
                                          /* ------------------------
*/
     num = num - 1;
  }
```

```
}

char* hex2bcd(unsigned int x)
{
  unsigned int d3,d2,d1,d0;

  char* STRval = "0000";

  d3 = x/1000;    //Creates 1000s place integer
  x = x-d3*1000;

  d2 = x/100;     //Creates 100s place integer
  x = x-d2*100;

  d1 = x/10;      //Creates 10s place integer
  x = x-d1*10;

  d0 = x;         //Creates 1s place integer

  STRval[0] = d3+0x30;    //Puts integer values into proper order
  STRval[1] = d2+0x30;
  STRval[2] = d1+0x30;
  STRval[3] = d0+0x30;

  return STRval;    //Returns the string to be displayed on the LCD
}

unsigned int get_heading(void)
{
   unsigned int data;

   iic_start(0x42);        //Start write operation
     while((IBCR & 0x20) == 0);       //Waits for control of bus
     iic_transmit(0x41);        //Sends "A" telling compass to begin
sending data
     iic_stop();      //Stops write operation
   iic_start(0x43);        //Starts read operation
     iic_swrcv();          //Switches I2C to receive mode

     data = (((unsigned int) iic_receive_m1()) << 8) & 0xFF00;
     data = data | (((unsigned int) iic_receive_last()) & 0x00FF);

     iic_stop();
}

void calib(void)
{
   /*Start Calibration*/
   iic_start(0x42);        //Start write operation
     while((IBCR & 0x20) == 0);       //Waits for control of bus
     iic_transmit(0x43);        //Sends "C" telling compass to begin
calibration routine
```

```
     iic_stop();      //Stops write operation

     /*Movement Routines*/
     CW_turn(50);
     delay_1ms(500);
     Stop_Motors();
     delay_1ms(10);

     /*End Calibration*/
   iic_start(0x42);         //Starts write operation
     while((IBCR & 0x20) == 0);       //Waits for control of bus
     iic_transmit(0x45);         //Sends "E" telling compass to end
calibration routine
     iic_stop();      //Stops write operation
}


/* DELAY SUBROUTINES (A LA RISON) */

#define D50US 133 /* Inner loop takes 9 cycles; need 50x24 = 1200
cycles */
void delay_50us(int n)
{
  volatile int c;
  for (;n>0;n--)
  for (c=D50US;c>0;c--) ;
}

void delay_1ms(int n)
{
  for (;n>0;n--) delay_50us(200);
}

/* INPUT CAPTURE SUBROUTINES */

void interrupt input_capture_left(void)
{
  if(global_input_capture_left_var == 14)                 // If we've
counted 15 holes,
  {
    global_left_wheel_rev = global_left_wheel_rev + 1;    // increment
the wheel count
    global_input_capture_left_var = 0;                    // & reset
the hole count.
  }
  else                                                    //
Otherwise,
  {
    global_input_capture_left_var = global_input_capture_left_var +
1;// increment the hole count.
  }
    TFLG1 = 0x02; // clear flag
```

```
}

void interrupt input_capture_right(void)
{
  if(global_input_capture_right_var == 14)                // If we've
counted 15 holes,
  {
    global_right_wheel_rev = global_right_wheel_rev + 1;  // increment
the wheel count
    global_input_capture_right_var = 0;                   // & reset
the hole count.
  }
  else                                                    //
Otherwise,
  {
    global_input_capture_right_var = global_input_capture_right_var +
1;// increment the hole count.
  }
  TFLG1 = 0x04; // clear flag
}

/* BASIC MOTION SUBROUTINES */

void Backward_Motors(float DUTYCYCLE)
{
  int stickyb4;
  int stickyb5;
    PORTA_BIT0 = 0;
    PORTA_BIT1 = 0;//SET DIRECTION OF PIN 00000011-PORTA 0,1
    stickyb4 =  (DUTYCYCLE/100)*PWM_Total;//BOTH MOTOR WITH SAME SPEED
FORWARD
    PWMDTY4 = stickyb4;
    stickyb5 =(DUTYCYCLE/100)*PWM_Total;//BOTH MOTOR WITH SAME SPEED
FORWARD
    PWMDTY5 = stickyb5;
}

void Forward_Motors(float DUTYCYCLE)
{
  int stickyf4;
  int stickyf5;
    PORTA_BIT0 = 1;
    PORTA_BIT1 = 1;
    stickyf4 =(DUTYCYCLE/100)*PWM_Total;//Right Motor: BOTH MOTOR WITH
SAME SPEED FORWARD
    PWMDTY4 = stickyf4;
    stickyf5 =(DUTYCYCLE/100)*PWM_Total;//BOTH MOTOR WITH SAME SPEED
FORWARD
    PWMDTY5 = stickyf5;
}

void Stop_Motors()
```

```
{
    PWMDTY4 = 0; // TURNS OFF BOTH MOTORS
    PWMDTY5 = 0;
}

void CCW_turn(float DUTYCYCLE)
{
  int pwmLR;
  //Stop_Motors();                          // turn off motors
  pwmLR = (DUTYCYCLE/100)*PWM_Total;     // set left motor to same
duty cycle as right
  PWMDTY4 = pwmLR;
  PWMDTY5 = pwmLR;
  PORTA_BIT0 = 1;                         // sets the directions
  PORTA_BIT1 = 0;
  //delay_1ms(5);
  //Stop_Motors();
}

void CW_turn(float DUTYCYCLE)
{
  int pwmLR;
  //Stop_Motors();                          // turn off motors
  pwmLR = (DUTYCYCLE/100)*PWM_Total;     // set left motor to same
duty cycle as right
  PWMDTY4 = pwmLR;
  PWMDTY5 = pwmLR;
  PORTA_BIT0 = 0;                         // sets the directions
  PORTA_BIT1 = 1;
  //delay_1ms(5);
  //Stop_Motors();
}
```

Appendix B: GPS Code

```
//INTERFACING GPS (UP501R) WITH DRAGON12 HCS12 MICROCONTROLLER
#include "vectors12.h"
#include <hidef.h>
#include "hcs12.h"
#include "DBug12.h"
#include "lcd.h"


#define S1      1   /* Wait for first start char '$' */
#define S2      2   /* Wait for second start char 'G' */
#define S3      3   /* Wait for third start char 'P' */
#define S4      4   /* Wait for message id 'G' */
#define S5      5   /* Wait for message id 'G' */
#define S6      6   /* Read position-time message */
#define S7      7
#define S8      8   /* Get check-sum for position-time message */
#define S9      9
#define S10    10

#define ID1       'G'  /* UP501 packet id 'G' */
#define ID2       'G'  /* UP501 packet id 'G' */
#define ID3       'A'  /* UP501 packet id 'A' */
#define CR       0x0D  /*ascii carrage return - framing char   */
#define LF       0x0A  /*ascii line feed - framing char         */
#define COMMA    0x2C
#define ASTRIX   0x2A

/*
$GPGGA,025113.000,3403.7829,N,10654.3442,W,1,6,1.77,1432.4,M,-
24.1,M,,*50
*/
#define HOUR_POS_1  7  /*position of the hour info in packet   */
#define HOUR_POS_2  8
#define MIN_POS_1   9  /*position of the minute info in packet */
#define MIN_POS_2   10
#define SEC_POS_1   11  /*position of the second info in packet */
#define SEC_POS_2   12
/*-----------------------------------------------------------------
--------*/
#define LATITUDE_DEGREE_POS_1    18  /*position of the degree info in
packet   */
#define LATITUDE_DEGREE_POS_2    19
#define LATITUDE_MIN_POS_1       20   /*position of the minute info in
packet */
#define LATITUDE_MIN_POS_2       21
#define LATITUDE_SEC_POS_1       23   /*position of the second info in
packet */
#define LATITUDE_SEC_POS_2       24
#define LATITUDE_SEC_POS_3       25
#define LATITUDE_SEC_POS_4       26
```

```
#define LATITUDE_DIRECTION_POS   28   /*position of the direction
info in packet */
/*-----------------------------------------------------------------
--------*/
#define LONGITUDE_DEGREE_POS_1   30  /*position of the degree info in
packet   */
#define LONGITUDE_DEGREE_POS_2   31
#define LONGITUDE_DEGREE_POS_3   32
#define LONGITUDE_MIN_POS_1      33   /*position of the minute info in
packet */
#define LONGITUDE_MIN_POS_2      34
#define LONGITUDE_SEC_POS_1      36   /*position of the second info in
packet */
#define LONGITUDE_SEC_POS_2      37
#define LONGITUDE_SEC_POS_3      38
#define LONGITUDE_SEC_POS_4      39
#define LONGITUDE_DIRECTION_POS  41  /*position of the direction info
in packet */
/*-----------------------------------------------------------------
--------*/
//$GPGGA,035958.000,3403.7851,N,10654.3480,W,1,4,2.06,1427.3,M,-
24.1,M,,*53

#define PS12_LEN   74  /*length of the Up501 data packet to save   */

//Global variables for the GPS time data
unsigned char hour_1s=0x30, hour_10s=0x30, minute_1s=0x30,
minute_10s=0x30, second_1s=0x30, second_10s=0x30,

latitude_seconds_1s=0x30,latitude_seconds_10s=0x30,latitude_seconds_10
0s=0x30,latitude_seconds_1000s=0x30,
              latitude_minute_1s=0x30,latitude_minute_10s=0x30,
latitude_degree_1s=0x30, latitude_degree_10s=0x30,
              latitude_direction=0x30,

longitude_seconds_1s=0x30,longitude_seconds_10s=0x30,longitude_seconds
_100s=0x30,longitude_seconds_1000s=0x30,

longitude_minute_1s=0x30,longitude_minute_10s=0x30,longitude_degree_1s
=0x30,longitude_degree_10s=0x30,
              longitude_degree_100s=0x30,longitude_direction=0x30;

unsigned char ps12_buf[PS12_LEN]; /* Up501r PS12 data packet array */
unsigned char temp;

char ISR_count=0;

interrupt void SCI_isr(void);

char msg1_array[20];
char* msg1=msg1_array;  /* Declaring pointer to the first line of text
on the display*/
```

```
char msg2_array[20];
char* msg2=msg2_array;  /* Declaring pointer to the second line of
text on the display*/

void InitGPS(void); //SCI intialization
unsigned char temp;
void main(void)
{
  __asm(sei);

  UserSCI1 = (unsigned short)& SCI_isr;
  InitGPS();

  __asm(cli);
l  while(1)
  {
    __asm(wai);
    openlcd();
    // If GPS is not receiving any messages, it displays no signal
message
     if(latitude_degree_10s== COMMA) {

     msg1_array[0]   = 'N';
    msg1_array[1]   = 'O';
    msg1_array[2]   = ' ';
    msg1_array[3]   = 'S';
    msg1_array[4]   = 'I';
    msg1_array[5]   = 'G';
    msg1_array[6]   = 'N';
    msg1_array[7]   = 'A';
    msg1_array[8]   = 'L';
    msg1_array[9]   ='\0';
    puts2lcd(msg1);
    put2lcd(0xC0,CMD);   // move cursor to 2nd row, 1st column

    msg2_array[0]     = 'N';
    msg2_array[1]     = 'O';
    msg2_array[2]     = ' ';
    msg2_array[3]     = 'S';
    msg2_array[4]     = 'E';
    msg2_array[5]     = 'N';
    msg2_array[6]     = 'A';
    msg2_array[7]     = 'L';
    msg2_array[8]     = '.';
    msg2_array[9]     = '.';
    msg2_array[10]    = '.';
    msg2_array[11]    = '.';
    msg2_array[12]    = '.';
    msg1_array[13]    ='\0';
    puts2lcd(msg2);

     }
```

```
      else {
    //Displaying Latitude
    msg1_array[0]   = 'L';
    msg1_array[1]   = 'a';
    msg1_array[2]   = ':';
    msg1_array[3]   = latitude_degree_10s;
    msg1_array[4]   = latitude_degree_1s;
    msg1_array[5]   = '|';
    msg1_array[6]   = latitude_minute_10s;
    msg1_array[7]   = latitude_minute_1s;
    msg1_array[8]   = '|';
    msg1_array[9]  = latitude_seconds_1000s;
    msg1_array[10]  = latitude_seconds_100s;
    msg1_array[11]  = latitude_seconds_10s;
    msg1_array[12]  = latitude_seconds_1s;
    msg1_array[13]  ='|';
    msg1_array[14]  = latitude_direction;
    msg1_array[15]  ='\0';

    puts2lcd(msg1);
    put2lcd(0xC0,CMD);   // move cursor to 2nd row, 1st column
    //Displaying Longitude
    msg2_array[0]   = 'L';
    msg2_array[1]   = 'o';
    msg2_array[2]   = ':';
    msg2_array[3]   = longitude_degree_100s;
    msg2_array[4]   = longitude_degree_10s;
    msg2_array[5]   = longitude_degree_1s;
    msg2_array[6]   = '|';
    msg2_array[7]   = longitude_minute_10s;
    msg2_array[8]   = longitude_minute_10s;
    msg2_array[9]  = '|';
    msg2_array[10]  = longitude_seconds_1000s ;
    msg2_array[11]  = longitude_seconds_100s ;
    msg2_array[12]  = longitude_seconds_10s ;
    msg2_array[13]  = longitude_seconds_1s ;
    msg2_array[14]  = '|';
    msg2_array[15]  = longitude_direction ;
    msg2_array[16]  ='\0';
    puts2lcd(msg2);

      }

  }
}

/*************************************************************************

GPS string Output:
$GPGGA,035958.000,3403.7851,N,10654.3480,W,1,4,2.06,1427.3,M,-
24.1,M,,*53
$GPGSA,A,3,28,24,08,17,,,,,,,,,2.29,2.06,0.99*0D
```

```
$GPGSV,3,1,10,17,74,251,18,28,60,026,25,08,54,138,24,26,35,257,*78
$GPGSV,3,2,10,24,28,062,20,07,24,142,,11,22,051,,15,18,298,*7F
$GPGSV,3,3,10,27,14,320,,04,13,182,*74
$GPRMC,035958.000,A,3403.7851,N,10654.3480,W,0.46,199.17,280411,,,A*73


**********************************************************************/

interrupt void SCI_isr(void)
{

    unsigned char this_byte;
    static unsigned char count;
    static unsigned char sci_state = S1;

  if ((SCI1SR1 & 0x20)==0) return;
    this_byte = SCI1SR1;
    this_byte = SCI1DRL;   /*clear the SCI flag, and grab the data*/
    temp = SCI1DRL;

    switch (sci_state)
    {
     /*state S1 waits for an '$' in the data then continues*/
     case S1:
          if (this_byte == '$') sci_state = S2;
          break;

          /*state S2 represents having the first 'G'.  it waits for
            another then continues*/
     case S2:
          if (this_byte == 'G') sci_state = S3;
          else sci_state = S1;
          break;

          /*state S3 represents having the first 'P'.  it waits for
            another then continues*/
      case S3:
        if (this_byte == 'P') sci_state = S4;
          else sci_state = S1;
          break;

          /*state S4 represents having the start pattern '$GP'.  at
this point, both leading
            framing characters have been seen.  S4 now waits for the
first id*/
     case S4:
          if(this_byte == ID1) sci_state = S5;
          else sci_state = S1;
          break;

          /*state S5 represents having the start pattern '$GP'.  at
this point, all three leading
```

```
            framing characters have been seen.  S5 now waits for
the second id*/
      case S5:
        if(this_byte == ID2) sci_state = S6;
           else sci_state = S1;
           break;

           /*state S6 represents having the three framing chars and
the first and second id.
             it now waits for the third id then continues*/
     case S6:
           if(this_byte == ID3)
           {
                count = 6;
                sci_state = S7;
           }
           else sci_state = S1;
           break;


           /*state S7 rpresents having the packet identified.  the
packet data is then
             scanned in until all needed data is saved in the receive
buffer */
     case S7:
           ps12_buf[count++] = this_byte;
           if(count > PS12_LEN-6)    /* All data except check sum, CR,
LF */
                sci_state = S8;
           break;

           /* state S8 represents receiving the check sum */
     case S8:
           ps12_buf[count++] = this_byte;
           sci_state = S9;


           /*Grabbing Latitude Data*/
           latitude_degree_10s       =
ps12_buf[LATITUDE_DEGREE_POS_1];
           latitude_degree_1s        =
ps12_buf[LATITUDE_DEGREE_POS_2];
           latitude_minute_10s       = ps12_buf[LATITUDE_MIN_POS_1];
           latitude_minute_1s        = ps12_buf[LATITUDE_MIN_POS_2];
           latitude_seconds_1000s    = ps12_buf[LATITUDE_SEC_POS_1];
           latitude_seconds_100s     = ps12_buf[LATITUDE_SEC_POS_2];
           latitude_seconds_10s      = ps12_buf[LATITUDE_SEC_POS_3];
           latitude_seconds_1s       = ps12_buf[LATITUDE_SEC_POS_4];
           latitude_direction        =
ps12_buf[LATITUDE_DIRECTION_POS];
           /*Grabbing Longitude Data*/
           longitude_degree_100s      =
```

```
ps12_buf[LONGITUDE_DEGREE_POS_1];
            longitude_degree_10s       =
ps12_buf[LONGITUDE_DEGREE_POS_2];
            longitude_degree_1s        =
ps12_buf[LONGITUDE_DEGREE_POS_3];
            longitude_minute_10s       = ps12_buf[LONGITUDE_MIN_POS_1];
            longitude_minute_1s        = ps12_buf[LONGITUDE_MIN_POS_2];
            longitude_seconds_1000s    = ps12_buf[LONGITUDE_SEC_POS_1];
            longitude_seconds_100s     = ps12_buf[LONGITUDE_SEC_POS_2];
            longitude_seconds_10s      = ps12_buf[LONGITUDE_SEC_POS_3];
            longitude_seconds_1s       = ps12_buf[LONGITUDE_SEC_POS_4];
            longitude_direction        =
ps12_buf[LONGITUDE_DIRECTION_POS];

            /* state S9 represents receiving the trailing CR */
      case S9:
            ps12_buf[count] = this_byte;
            if (this_byte ==  ASTRIX)
            {
                  sci_state = S10;
                  count++;
            }
            else
            {
                  sci_state = S1;
            }
            break;

            /* state S10 - represents receiving the trailing LF */
            /* Position-time message complete; decode to send to PLD */
      case S10:
            ps12_buf[count] = this_byte;
            sci_state = S1;
            break;
     }
     return;
}
```

Appendix C: Total Financial Budget

Item	Quantity	Cost Per Item	Total Cost	Estimated Remaining Budget ($750.00)
Robot Chasis	1	$250.00	$250.00	$500.00
Dragon-12 EVB	1	$140.00	$140.00	$360.00
H-Bridge (755)	2	$39.95	$79.90	$280.10
9V Battery	1	$1.86	$1.86	$278.24
9V Battery Connection	1	$0.30	$0.30	$277.94
RC Battery	1	$27.00	$27.00	$250.94
Wiring Harnesses	1	$5.00	$5.00	$245.94
Color/Reflector Sensor	1	$16.31	$16.31	$229.63
Temperature Sensor	1	$19.95	$19.95	$209.68
Ultrasonic Range Finder	1	$29.99	$29.99	$179.69
GPS	1	$37.50	$37.50	$142.19
Optical Sensors	2	$1.00	$2.00	$140.19
Compass	1	$34.95	$34.95	$105.24
Boost Converter	1	$6.00	$6.00	$99.24
Debounce Switches	3	$10.00	$30.00	$69.24
HMC6352	1	$34.95	$34.95	$34.29
HMC6352 Second	1	$34.95	$34.95	-$0.66
Interdepartment Costs	1	10.25	$10.25	-$10.91

Appendix D: Power Budget

9 V Source	
Microcontroller	124mA
UP501R GPS	25mA
HMC6352 Compass	10mA
6.6 V Source	
H-Bridges/Motors	15A