AN ABSTRACT OF THE THESIS OF

Thomas Shepherd for the degree of Honors Baccalaureate of Science in Electrical and Computer Engineering, from Oregon State University, presented on June 4th, 2012. Title: Designing and Building a Bluetooth Controlled Blimp with Autonomous Capabilities

Abstract approved: _____
Roger L. Traylor

This thesis describes the design and implementation of a Bluetooth controlled blimp with autonomous capabilities through Dead Reckoning. Reasonable research was conducted on current autonomous systems, wireless communication techniques, and suitable microprocessors that could handle the required processing. Building a remote-controlled blimp requires a light understanding of mechanics to build the blimp itself and a strong understanding of electronics in order to design the remote-control and built-in computer systems. This thesis will focus on the various blocks necessary to create a remote-controlled, autonomous blimp that does not rely on GPS, all while explaining the design process we faced as well as the final results. Dead Reckoning is used in place of GPS to control the blimp's navigation. All design decisions are backed up with concrete data and logical conclusions. The end system met the specified requirements and could successfully navigate around large, stationary objects, but further improvements to the Dead Reckoning system are still required.

Key Words: engineering, electrical, blimp, Dead Reckoning, robotics, autonomous, wireless

E-mail address: shephert@onid.orst.edu

Designing and Building a Bluetooth Controlled Blimp with Autonomous Capabilities

by

Thomas Shepherd

A PROJECT

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Electrical and Computer Engineering
(Honors Associate)

Presented June 4th, 2012
Commencement June 2012

CONTRIBUTION OF CO-AUTHORS

Three engineering students, Chris Stoddard, Cyrus Heick, and myself, comprised the design team that completed this project. Each member was responsible for the completion of different aspects of the project, however we all contributed equally and collectively to accomplish each task. The general areas of responsibility for each member are listed below, although we worked as a group to best fulfill each goal.

- Chris Stoddard
    - Bluetooth Transceiver
    - Microcontroller Hardware Design
    - Blimp to PC Communication
- Cyrus Heick
    - Blimp & Gondola Design
    - Motor Controller & Propulsion System
    - Navigation
- Thomas Shepherd
    - PC to Blimp User Interface
    - Power Supply & Battery
    - Collision Detection

Honors Baccalaureate of Science in Electrical and Computer Engineering project of Thomas Shepherd presented on June 4th, 2012.

APPROVED:

_____

Mentor, representing Electrical and Computer Engineering

_____

Committee Member, representing Electrical and Computer Engineering

_____

Committee Member, representing Electrical and Computer Engineering

_____

Dean, University Honors College

I understand that my project will become part of the permanent collection of Oregon State University, University Honors College. My signature below authorizes release of my project to any reader upon request.

_____

Thomas Shepherd, Author

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDIX FIGURES

DESIGNING AND BUILDING A BLUETOOTH CONTROLLED BLIMP WITH AUTONOMOUS CAPABILITIES

INTRODUCTION

Problem Background

Hewlett-Packard came up with the initial idea to design an inertial navigation blimp that would be semi-autonomous, capable of flying a pre-determined 3D route within a large structure or building. The blimp would consist of three, one-axis accelerometers capable of effective navigation. Our team of engineers, consisting of myself, Chris Stoddard, and Cyrus Heick, took this initial concept, expanding on it as well as focusing it to address a particular problem.

We wanted to develop a blimp that could be flown manually or autonomously, capable of maneuvering intelligently through the use of various sensors. While small helicopters can be used to successfully navigate the interior of a structure and achieve the same purpose, they consume a large amount of power and cannot be flown for sustained periods of time. However, a blimp outfitted with a proper propulsion system and sensors can stay in the air longer and cover a greater amount of territory, all while gathering useful data specific to a particular, separate application.

After the basic functionality was in place, we felt we could improve on the design by changing the manual control mechanism over to a typical video game controller. The controller would have to be connected to the same computer system monitoring the blimp's activities, but would allow for natural, intuitive steering and control of the

blimp's propulsion system. As autonomous navigation was one of the driving forces behind the project inception, we also documented our challenges with incorporating the Dead Reckoning system that governed the navigation system and proposed possible ideas for improvement.

Method

To solve the problem of designing a fully controllable and autonomous blimp, we broke the hardware and software into multiple sub-systems. The largest blocks consisted of the following: Power Supply, User Interface, Bluetooth Transceiver, Microcontroller hardware, Microcontroller code, Motor Controller, Motors, Navigation Sensor, Collision Sensor, Blimp, and Battery. For each block, we wrote up a plan of attack with an appropriate design and verified that the proposed solution would actually solve the problem at hand and meet the specifications we set up for ourselves. The exact system requirements that we wanted to meet are detailed in the next section. As a group, we approved the plans for each block before moving into the building stage.

While addressing the problem, we prioritized low cost and simplicity, as we wanted our results to be easily reproducible, and a high degree of flexibility and control in our user-friendly user interface, giving the user the utmost ease possible when programming and flying the blimp. We planned to build a single prototype, keeping a detailed BOM and documenting the build process as much as possible. The prototype was necessary to show that we successfully created a flying, autonomous device capable of gathering data and staying airborne for long periods of time. Once the prototype was

complete, we ran a series of tests, the results of which are included towards the end of this document.

SYSTEM REQUIREMENTS

Specifications

In order to accurately and completely solve the problem, our group agreed upon a certain number of design specifications that we would force ourselves to meet. The specifications are summarized in the Table 1.

	Specification	**Description**
1.	Autonomous Mode	The blimp should also have the ability to independently fly a particular flight pattern that is programmed by the user. These pre-determined paths should include, but are not limited to, basic patterns or curves that can occupy 3D space while steering clear of walls and other large obstacles in the flight path. The blimp must be able to store up to 10 patterns, each allowing for a maximum of 100 meters of distance.
2.	Blimp and Computer Data Transfer	The blimp must be capable of transmitting data over a wireless signal to a computer 100ft. away. The error rate in the data must be less than 0.2%.
3.	Blimp Lift Capacity	The blimp must have adequate lift so that it can stay buoyant and float without using any battery power. The blimp's drift on a vertical axis must be less than 3 inches per second while carrying a load of approximately 10oz. that includes all electronics and motors required.
4.	Collision Detection	An automatic collision detection system should be built into the blimp's hardware and software system so that it can override any programmed commands in order to avoid collisions with a wall or other sizeable objects. At maximum speed, the blimp should be able to stop or maneuver around an object in its path.
5.	Exterior Scanning	The blimp should contain basic built-in scanning features that enable it to detect physical objects up to 15ft away in all directions except directly behind the blimp. This leaves the directions of in front, to the left, to the right, above, and below. Distance between the blimp and the exterior object must be accurate to within 2ft. All scanned data should be transmitted to a computer in a form that allows for mapping of the blimp's location as well as the location of objects around the blimp.
6.	Flight time	The blimp should have at least one hour of flight time when being run under worst case conditions (using a constant, maximal power draw).

7.	Remote Control	At any time, up to 50ft away, the blimp can be controlled in manual mode by a remote control. The user should have full, intuitive control over the blimp's movements, able to move the blimp up/down and forwards/backgrounds, as well as turn the blimp from left to right.
8.	Usability	At least 80% of users must find the controls and operational features "intuitive" and "easy to use". The capabilities of the blimp should be transparent, and the user interface graphical in nature and simple to navigate.

Table 1: Project Specifications

REVIEW OF CURRENT TECHNOLOGY

Similar complete solutions

The YARB 1.0, developed by Surveyor corporation, was a development project in 2008 that focused on autonomous flight through advanced computer vision algorithms by placing a camera on a blimp [1]. The design was based in part off ongoing SLAM (simultaneous localization and mapping) research. The camera contained a 500MHz Blackfin processor capable of transmitting video feed at 30 frames/second over an 802.11g WiFi connection. All navigation and 3D mapping capabilities are based solely off the information gathered by the sophisticated camera. Manual control was available through a G1 Android phone, communicating on the same WiFi connection. The blimp bag is 66 inches in length, but cannot carry any additional hardware and only contains two small motors. The entire system sells for $875.

In a very similar system to the YARB, a design by DIY Drones consisted of a 52 inch blimp that used IR sensors and a homing beacon which allowed the blimp to travel out and find its way home [2]. Instead of using a camera, the blimp used ultrasonic rangefinders to “see” the environment. The electronics necessary to build this blimp are packaged in a “Blimpduino Kit” for $89.99 as in add-on to the YARB system.

On a more general scale, a commonly known autonomous, flying system is the Predator Drone which navigates using GPS, cameras, and IR. A typical drone has a wingspan of 48.7 feet and costs approximately $4.5 million [3].

There are also a number of grounded robots capable of navigation and surveillance such as the Mars rovers which are tasked with collecting data and taking

samples as they automatically traverse across potentially harsh terrain. A similar robot that is available on the market by Surveyor corporation is the SRV-1 Blackfin Mobile Robot. It is extremely small, no more than 5 inches long, and uses 2 laser pointers for object detection. It can be controlled manually over an 802.11b WiFi connection, comes fully programmable, and is sold for $525 [4].

Although there are more unmanned vehicles on the market, there are two that really seem to match our vision for our autonomous blimp- the YARB Robotic Blimp from the Surveyor Corp, and the Blimpduino blimp kit from DIY Drones. A common thread between these two blimps is the use of Maxbotics Ultrasonic Rangefinders. The Blimpduino takes a simplified approach, using only one unit to monitor the distance from the ground. The YARB implements a much more in depth system, using two rangefinders and a compass in order to perform object detection. One of the main reasons for the increased cost of the YARB is its data capabilities. Simply put, Surveyor's YARB is much more sophisticated in terms of its ability to capture visual and spatial data, and transmit it in real-time to a Java based computer console. It is also able to receive data from the Java console, yet perhaps its most exciting feature is its Android app. Surveyor Corp has provided an app that allows a user to control the blimp using the tilt-sensing features of the HTC Android G1. The Tier II MAE “Predator” drone is a military application that is an example of some of the reconnaissance capabilities we would like to implement. Although physically it differs greatly from our blimp, it serves as an example regarding data collection in real world/military situations. The drone is piloted remotely by a fully trained pilot and is capable of transmitting real-time video. This makes it ideal for surveillance in long distance situations. The main drawbacks to the Predator include

maneuverability, flight reliability and camera accuracy, but these are mainly obstacles to military operation and are beyond the standards of what we wish to accomplish. The SRV-1 Blackfin mobile robot is the much smaller (5" vs. 66") ground-based counterpart to the YARB blimp from the same vendor (Surveyor). As such, it has many of the same capabilities but trades in the ability to fly in three dimensions for increased control and maneuverability on the ground. This makes it more ideal for surveillance in small spaces where the aerial blimp would be too large to navigate.

System Name	**YARB 1.0**	**Blimpduino Kit**	**Predator Drone**	**SRV-1 Blackfin Mobile Robot**
Cost	$875.00	$89.99	$4.5 million	$525.00
Vendor	Surveyor Corp	DIY Drones	General Atomics	Surveyor Corp
Size	66 inches	52 inches	48.7 foot wingspan	5"x4"x3"
Object Detection	Ultrasonic range finder	Ultrasonic range finder	Camera/Infrared	2 laser pointers
User Interface	Computer / Android phone	RC remote control	Fully trained pilot via UHF/SATCOM	RC remote control or remote console
Autonomous Capabilities	3D navigation in development	Homing with IR beacon	Unknown	Fully programmable for 2D navigation
Data Transmission	802.11g	None	UHF/Satellite	802.11b
Vehicle Type	Blimp	Blimp	Fixed-wing UAV	Tracked ground vehicle

Table 2: Examples of Similar Solutions

Autonomous methods

There are several methods of navigation that are typically used in today's market. Navigation is critical when a robotic vehicle must travel from one point to another. Even

being given a simple command such as traveling 100 ft in a single direction is difficult without absolute navigation since the actual distance traveled would be highly dependent on the specific system. However, some systems still use these simple techniques. Cars, for example, typically track the number of miles traveled by only taking into account the absolute distance traveled. But in order for a car to independently travel from one city to another, it must keep track of its exact location at any time, even if the location is only relative to the vehicle's starting point. To solve this problem, systems such as GPS or Dead Reckoning are used to continuously track position.

GPS (Global Positioning System) relies on satellites to continuously update a unit's location in terms of coordinates on the earth's surface. Each update is independent of the unit's previous position. However, the electronics necessary for GPS are relatively complex, so adding a GPS system to a blimp would not be very cost effective, especially considering that the blimp would not be covering large distances during a single flight. Additionally, since GPS depends on a signal to orbiting satellites, the ideal operating environment would be outside. Since the blimp is intended to be used indoors, it could potentially lose its GPS connection if the blimp travels too far underground or into a shielded area.

Dead Reckoning varies significantly from a GPS solution. Dead Reckoning relies on accelerometers and a gyroscope or compass to continuously update a unit's location relative to its pre-existing location. By integrating the accelerometers and using trigonometry to determine the direction of travel, a fast enough processor can successfully track the position of the device. Unlike GPS, the amount of error in the system is accumulative, since even a small error bias in the accelerometer measurements

would be incorrectly added over time to the unit's location. For a blimp which is traveling in 3D space, a total of 3 one-axis accelerometers would be required, along with a gyroscope to know which way the blimp is facing. The gyroscope would allow the blimp to travel "forward" as witnessed by the user, and would compensate for the ever changing X and Y axes as the blimp makes lateral turns. The cost to implement Dead Reckoning is considerably less than GPS, which is why further research into Dead Reckoning systems is needed as a possible new means of navigation.

Wireless communication

In order to properly communicate with the blimp, the user would need some kind of wireless connection to both transmit commands to the blimp and receive any gathered sensor data back. There are a variety of methods of wireless communication ranging from IR to RF transmission. Here we explore a variety of RF protocols and compare them to a basic IR solution.

WLAN (wireless local area network) is commonly known as the method used by laptops to connect to a wireless access point, or router, in order to get access to the internet. There are a number of standards within WLAN that fall under the 802.11 protocol. Typically these systems offer relatively high power solutions capable of transmitting up to 100ft, however the network cards necessary to implement WLAN are quite complex and can be difficult to setup. They are usually reserved for more sophisticated systems such as computers, and we had difficulty finding a cheap, elegant solution to building a basic transceiver capable of WLAN communication into an onboard circuit.

Bluetooth and ZigBee are also wireless protocols that use a 2.4GHz frequency to communicate at short distances between devices. Most Bluetooth and ZigBee transceivers only operate around 30 – 50 feet, but they are relatively simple and can be easily incorporated into a microcontroller design to communicate wirelessly. Some transceivers are even built into microcontrollers, such as the AT86RF230 by Atmel [5]. Bluetooth and ZigBee can also be used to communicate with smart phones and other mobile applications without many modifications. The largest downside to Bluetooth and Zigbee is the reduced effective range. To be used in the blimp design, these protocols would require an atypical power boost to get up to the desired range assigned in our specifications. With Bluetooth for example, there are different classes that represent the range. The vast majority of Bluetooth devices fall into Class 2 which uses ~2.5mW (4dBm) of power for ~10 meters in any direction. By increasing the power output to 100mW (20dBm), a Bluetooth transmission can go as far as 100 meters, as defined in Class 1. ZigBee units have similar statistics, but are intended to be slightly cheaper and simpler than Bluetooth.

Basic FM (Frequency Modulation) transmissions can be found in a typical radio, whether listening to music or using a walky-talky. While FM can be considered a lower level of data transmission, it is viewed here as a separate solution as one can build their own FM transmitter and receiver out of simple passive components and filters. FM would not have the embedded security and modulation/demodulation techniques that the Wi-Fi protocols contain, but it could be used to minimize power by sending the absolute minimum amount of data to a designated transceiver on the user side. A basic FM

solution particular to our problem may be optimal for a large scale project, but would require an impractical amount of work to obtain.

Finally, one of the other remaining techniques of wireless communication on the market is IR (Infrared) which is commonly found in TV remotes and other point and click devices. IR uses on/off infrared light to send a digital signal. The obvious downside to IR is it requires a visual line of sight between the transmitter and the receiver in order for the signal to be relayed. Even though IR is extremely easy to implement and quite cost effective, the blimp will need the ability to travel around objects and to the other side of walls without losing its connection to the user.

Some of the possible RF solutions are described in the table below.

System Name	**AT86RF230 [6]**	**AT86RF212 [7]**	**Custom Design [8]**	**CB-OEM [9]**
Cost	$3.70	$4.00	$10.00	$68.55
Vendor	Atmel	Atmel	Custom	Connect Blue
Function	Transceiver (ZigBee)	Transceiver (ZigBee)	Separate Transmitter and Receiver (FM)	Transceiver (Bluetooth)
Type of Design	IC (excluding antenna, and crystal)	IC (excluding antenna, and crystal)	PCB	Complete transceiver with pinout
Operation Frequency	2.4GHz	700-900MHz	Customizable	2.4GHz
Voltage	1.8-3.6V	1.8-3.6V	9V	3.0-6.0V
Weight	Light	Light	Heavy	Heavy

Table 3: Examples of RF transceivers

Microcontroller

For the blimp design, we required a microcontroller capable of handling communication with the user, controlling the motors and servo in the propulsion system, interpreting the navigation data to determine location, and any additional environment

sensing. While there are many varieties of microcontrollers, some of the most common microcontrollers on the market are Atmel's Atmega series and Texas Instrument's MSP series. For example, the Atmega128 and MSP430 series both contain a sufficient amount of programming memory space, over a dozen IO ports for controlling the circuit, PWM outputs for the motor controller, UART communication capabilities to interface with the wireless transceiver, as well as ADC's to read in analog measurements for the navigation system. These general purpose microcontrollers are perfect for our application in that they can read in the various sensors and sufficiently control the blimp. Processing speeds vary from 1MHz up to 500MHz, but for the sake of simplicity and cost we are targeting a microcontroller with an 8MHz processor.

The MSP430 micro processor from TI fits well into the desired requirements we specified. Running at 8MHz with 22 GPIO (general purpose input output) pins, 6 ADC (analog to digital converter) pins, and multiple serial communication connections all on an active current of less than 1mA and an extremely low standby current requirement of 0.3uA, we felt it was well suited for our design. Using TI's extensive user guides for their MSP430 series, we will quickly be able to configure the microcontroller settings to meet our needs. The MSP430 is also compatible with a simple JTAG programming device that can interface with any personal computer.

DESIGN SOLUTION

We addressed the problem of designing an autonomous, Dead Reckoning blimp using a divide and conquer strategy, breaking up the required hardware and software components into 11 engineering design blocks. Each block focused on structural integrity or addressed a key function of the blimp. Here we describe each of the 11 blocks and demonstrate how they combine to form a self-navigating, or optionally wirelessly-controlled blimp.

1. Power Supply

The device's power requirements are split into two categories. A high power, 5V, rail is designed specifically for a pair of motors in the propulsion system. A second, low current, 3.3V rail is designed for the MSP430 microcontroller as well as some of the sensors. A powerful yet rechargeable battery was chosen to supply power to these DC converters, and is described in further detail in a later section. For simplicities sake, the same buck converter was chosen for both the high and low power rails, dropping the voltage from the battery's 6.4V input to the specified output.

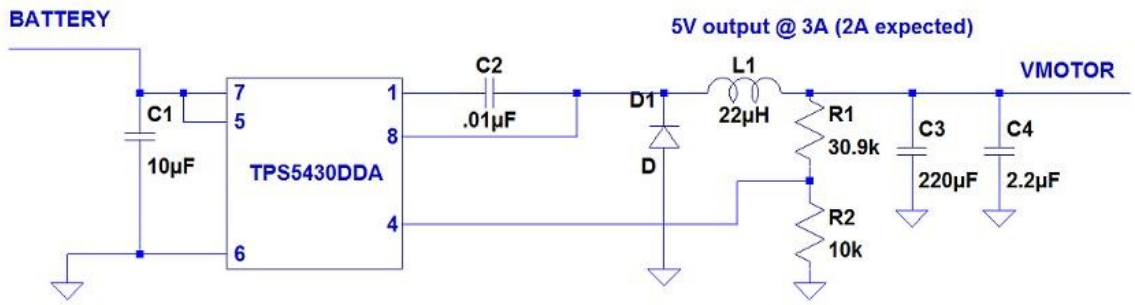

Figure 1: 5V Power Supply schematic diagram

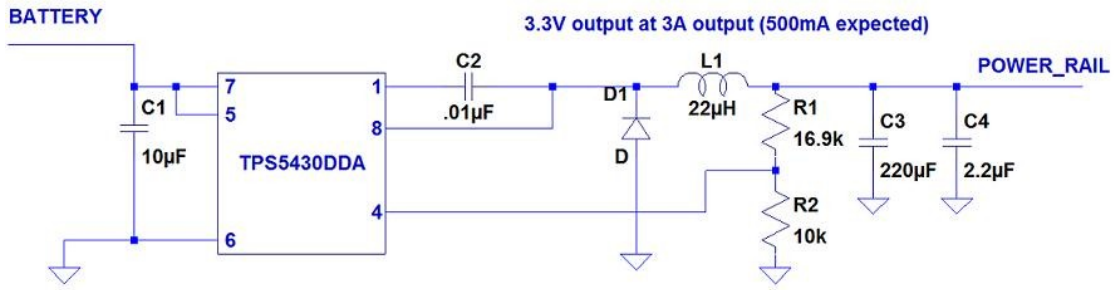

Figure 2: 3.3V Power Supply schematic diagram

The TPS5430 Buck Converter was chosen for its high efficiency and simplicity. The chip supports a maximum load of 3A with a large range of input and output voltages. The internal reference voltage is set to 1.221V, so the feedback resistors were chosen to drop the 5V or 3.3V respective outputs down to 1.221V. The internal PWM frequency is set internally to 500kHz. The minimum inductor size works out to 19.8uH, so a 22uH inductor was chosen. The 220uF capacitor fits with the 22uH inductor to create a stable, low noise output. The circuit was based off an application circuit in the TPS5430 datasheet.

The battery's power is passed through a fuse and connected to the inputs of both TPS5430 packages. The step down converters are designed to take a voltage input of between 5.5V and 36V, so the power supplies should work properly as long as the battery maintains adequate voltage. A single trace connects the fused battery output to an ADC on the MSP430 after being scaled down by a large resistor voltage divider. This allows the software to record the current voltage and report to the user. As calculated in the battery design block, the battery can output enough current to keep the blimp running for

almost two hours. Design requirements only call for an hour minimum of flight time before the battery requires recharging.

2. User Interface

A Windows computer is used as the primary control system for programming, flying, and communicating with the blimp. A Graphical User Interface is provided that works on all Windows machines, Windows XP and later. The .NET framework 3.0+ will be required to run the program. The program itself will be developed in C#, using a variety of built in control techniques and higher level access to a Bluetooth transceiver. In order to properly access the blimp, the computer must either have a built-in, internal, Bluetooth transceiver or an external, plug-in Bluetooth module. It is recommended that the computer be able to support a Bluetooth transmission of 50 – 100m to match the signal strength on of the transmitter on the blimp.

Since the blimp operates in a variety of modes, the user will have the ability to assign the blimp to a particular mode of operation.

In Manual Mode, the user has complete control of the movements of the blimp, much like using a remote control. The user will have the option to use keyboard hotkeys or a PS3 controller to fly the blimp. They will have full control over X, Y, and Z axes in a custom control scheme. If the user prefers, an option will also be available to set the speed in each direction. Using the keyboard interface, the speed can be selected from 0 to 100%. Using the PS3 controller, the left and right triggers will act as analog speed control, where the more the triggers are depressed, the faster the blimp will fly. The speed and direction of the blimp will show up on the user's screen to indicated current

movement. Also in Manual Mode, the user may initiate a scan of the blimp's surroundings and start graphing the scan's data. Graphs of the blimp's distance to objects in each directions will be graphed individually and show up in their corresponding locations around the image of a blimp on the GUI. If the user selects to view the navigation data instead of the scan data, an XYZ flight path will be mapped out that shows where the blimp started as well as its current location.

In Automatic Mode, the user will be able to create, save, load, and transmit flight paths to the blimp. The user can select from a list of shapes or paths. An entire flight path will consist of an array of destinations. The blimp will run through the flight path by reaching the next destination and continuing on until the final destination is reached. A flight path building program will be included in the GUI. Convenience features, such as a return to home command, will also be added to save time in building new flight patterns. The blimp will be able to run through the flight path as many times as the user prefers. Automatic mode will also provide the blimp with intelligent scanning capabilities, assuming it is within a building. For example, the collision detection system will automatically steer the blimp away from walls and large objects that get in the blimp’s way. By default, this option is selected to protect the blimp, and any flight paths will be overridden to prevent a collision. However, the blimp will try to successfully maneuver around the potential collision and resume the flight path if possible.

The overriding priority behind the user interface design is allowing the user to test the capabilities of the Dead Reckoning navigation system and push the blimp to its limits in terms of data collection as well as mobility. While much of the AI behind the blimp’s

autonomous abilities are embedded within the microcontroller's C program, the user should feel like they are seeing the results of the blimp's drone-like behavior first hand.

3. Bluetooth Transceiver

The blimp needs a Bluetooth transceiver that has a strong enough signal to reliably stay in contact with the connected computer system. For this reason we chose to go with a 'package' Bluetooth device made by ConnectBlue, OEMSPA331. It has all the necessary communication software already installed, but is also light weight compared to a standard USB Bluetooth module. The transceiver is also designed to be installed directly onto a PCB by soldering onto open solder pads. It is powered by the 3.3V rail power line and communicates with the microcontroller through the UART. Using only 5mW, the Bluetooth transceiver is still classified as Class 1, meaning it has an operating range of 100 meters in open air space. We felt this was a good balance between excessive battery use, and maximum operating distance from the computer. A normal Bluetooth transceiver falls into class 2 or 3 with an operating range between only 1 and 10 meters.

The connection scheme between the transceiver and the MSP430 microcontroller is outlined below. The 100 ohm resistors on the Data Set Ready and UART Receive lines, pins 16 and 11, are necessary to drop the 3.3V output of the microcontroller down to the 3.0V maximum input voltage on the transceiver. These values were recommended in the Connect Blue datasheet. The incoming signal voltage does not need to be adjusted because the microcontroller can properly read the 3.0V output of the Bluetooth transceiver.

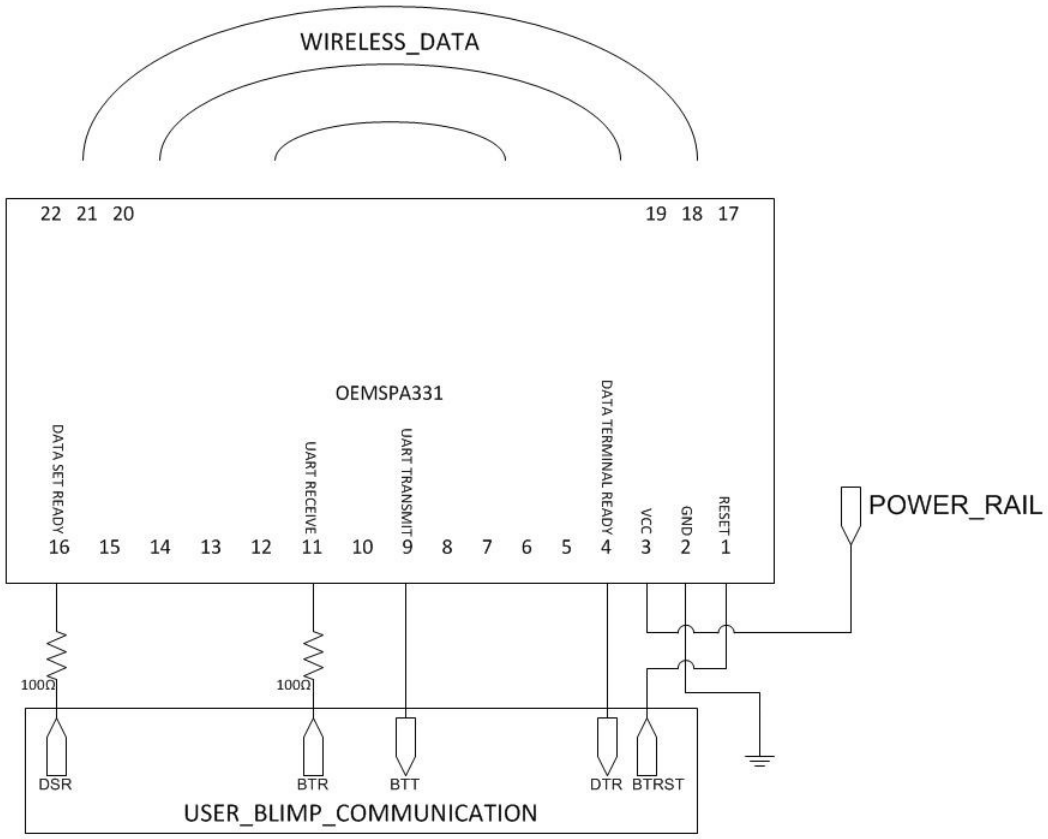

Figure 3: Bluetooth transceiver schematic diagram

In order to keep up constant communication with the connected computer system, any incoming Bluetooth transmissions will be picked up by an interrupt signal on the microcontroller for an immediate response. In this manner, the user can exit the automatic mode of the blimp at any moment and take over with manual control or manual commands. Outgoing transmissions are sent out periodically as sensor data is collected.

4. Microcontroller

As stated during our microcontroller research, the Dead Reckoning blimp needs a microcontroller that has multiple UART/SPI buses to be able to communicate with our

sensors and Bluetooth transceiver. It also needs multiple ADC inputs and multiple PWM channels to control the motors and servo. After TI offered to help fund our project, we decided on the MSP430F2410, which was one of the controllers that had more than enough buses, I/O's, ADC's, and PWM's that runs with sufficient performance at minimal power. The two diagrams below show the high level microcontroller block with its various connections, as well as a low level connection diagram of the microcontroller and surrounding components in the same subsystem.

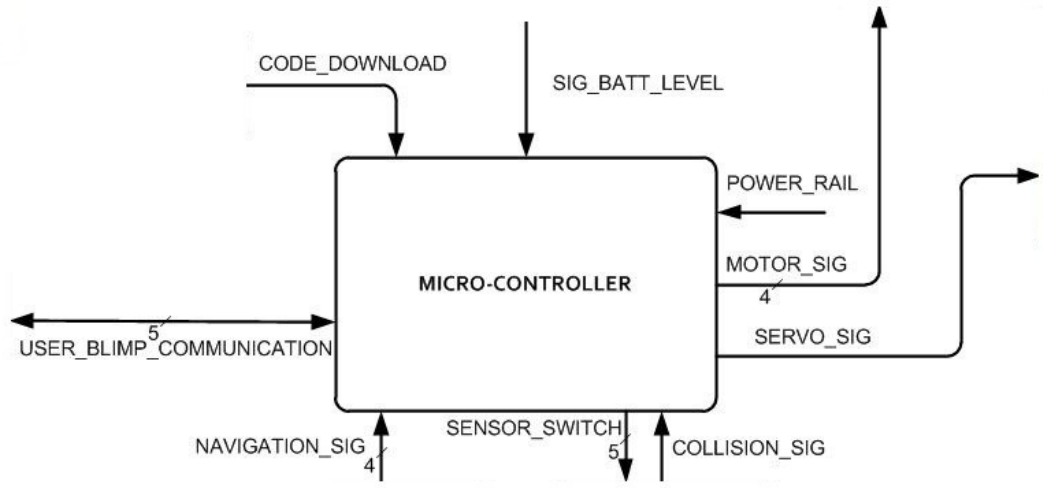

Figure 4: Microcontroller block diagram

The Code Download input is necessary for programming the microcontroller. The Signal Battery Level measures the battery voltage through an ADC. The four outgoing Motor Signals are used to control both propellers in the propulsion system. These are actually PWM lines that control motor speed by adjusting the duty cycle. The single

Servo Signal also uses a PWM output to turn a servo and adjust the direction of the propellers. The Collision Signal input and Sensor Switch lines represent the connection to the Ultrasonic sensors, which can map out an area and detect imminent collisions. The four Navigation Signals are connected to three one-axis accelerometers and one electronic gyroscope, making up the backbone of the Dead Reckoning navigation system. Finally, the User Blimp Communication two-way connection represents all communication with the Bluetooth transceiver, and therefore with the user. The more sophisticated diagram below includes four LED's that are tied directly to the microcontroller. These were very useful in debugging the blimp's internal systems and couple as status lights during normal operation.

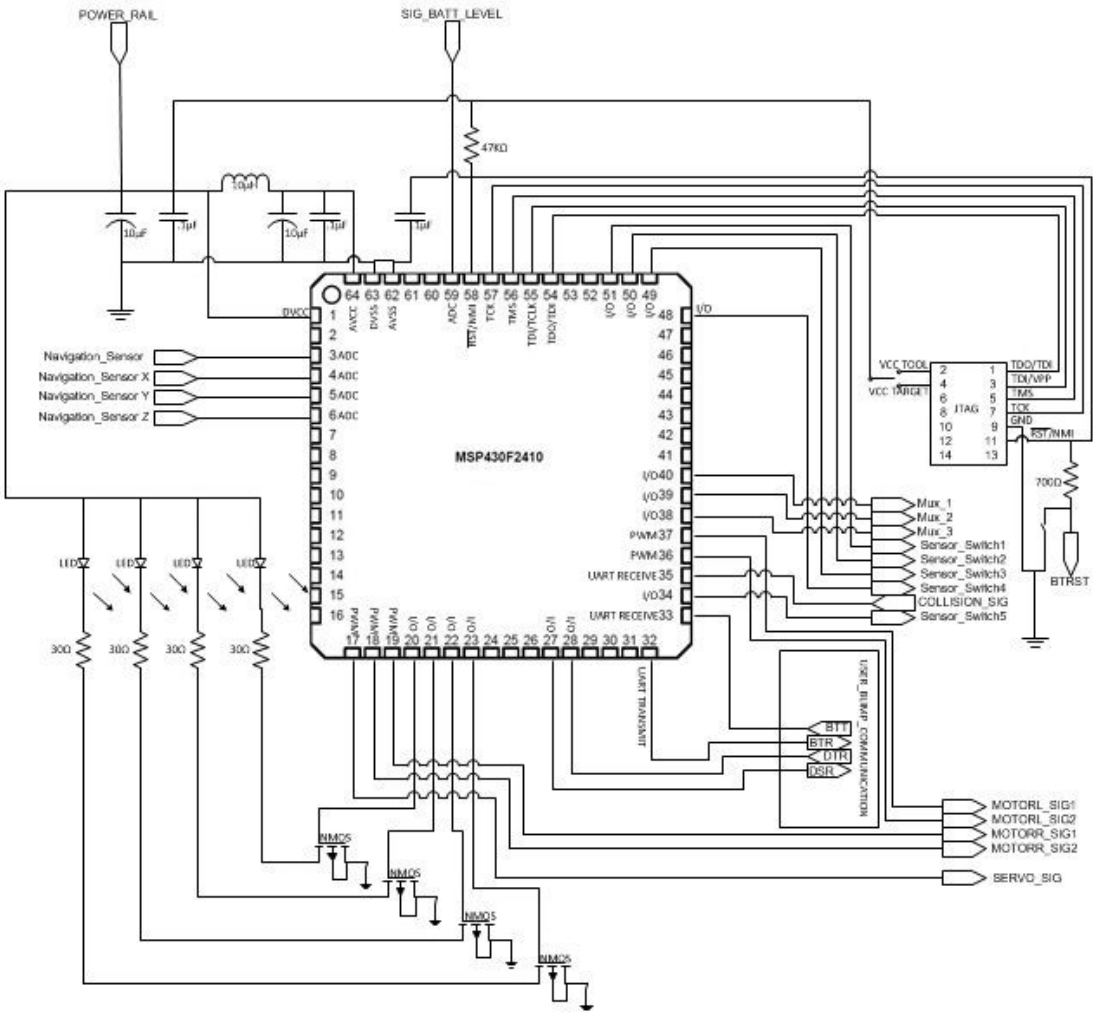

Figure 5: Microcontroller schematic diagram

5. Microcontroller Code

The microcontroller is responsible for controlling blimp movement and passing sensor data along through a Bluetooth output, all while managing the Dead Reckoning algorithm. An interrupt service will collect commands received via Bluetooth. The microcontroller will automatically respond by triggering the sensors or changing the motor speed conditions, all depending on the particular command received. Most sensors

will be continually polled for up-to-date data, including data from the accelerometers and the gyroscope. The current distances recorded by all the ultrasonic sensors will be on an interrupt service in order to detect collisions as quickly as possible. This data will be used to watch out for imminent collisions as well as scanning when required.

To control the movement of the blimp, five PWM signals are utilized. Two PWM signals per motor give the forward and backward speeds of that motor respectively. The fifth PWM signal controls a servo to tilt the motors up and down for movement in the Z axis.

Dead Reckoning is implemented by polling the most recent acceleration values in each of the X, Y, and Z axes, as well as the current direction the blimp is pointing from the gyroscope. The velocity of the blimp is constantly being updated by integrating the acceleration values. Likewise, the position of the blimp is calculated by integrating the velocity values. A custom algorithm that utilizes filtering and an exponentially weighted moving average helps reduce error over time. We found in our initial trials that an erroneous acceleration reading could significantly change the assumed velocity of the blimp. Over time, the incorrect velocity value would cause a dramatic drift in the blimp's assumed position.

To implement the blimp's automated functionality, we simply programmed an array of coordinate points along the appropriated path. When the blimp comes within a few feet of its current waypoint, the waypoint is replaced by the next coordinate point until the array is exhausted, at which point the final destination is reached. While the user interface on the personal computer is responsible for generating the coordinate point

array, the microcontroller stores and traverses the path by relying on the Dead Reckoning navigation system.

The diagram below represents the responsibilities of the microcontroller as well as the priority levels of each.

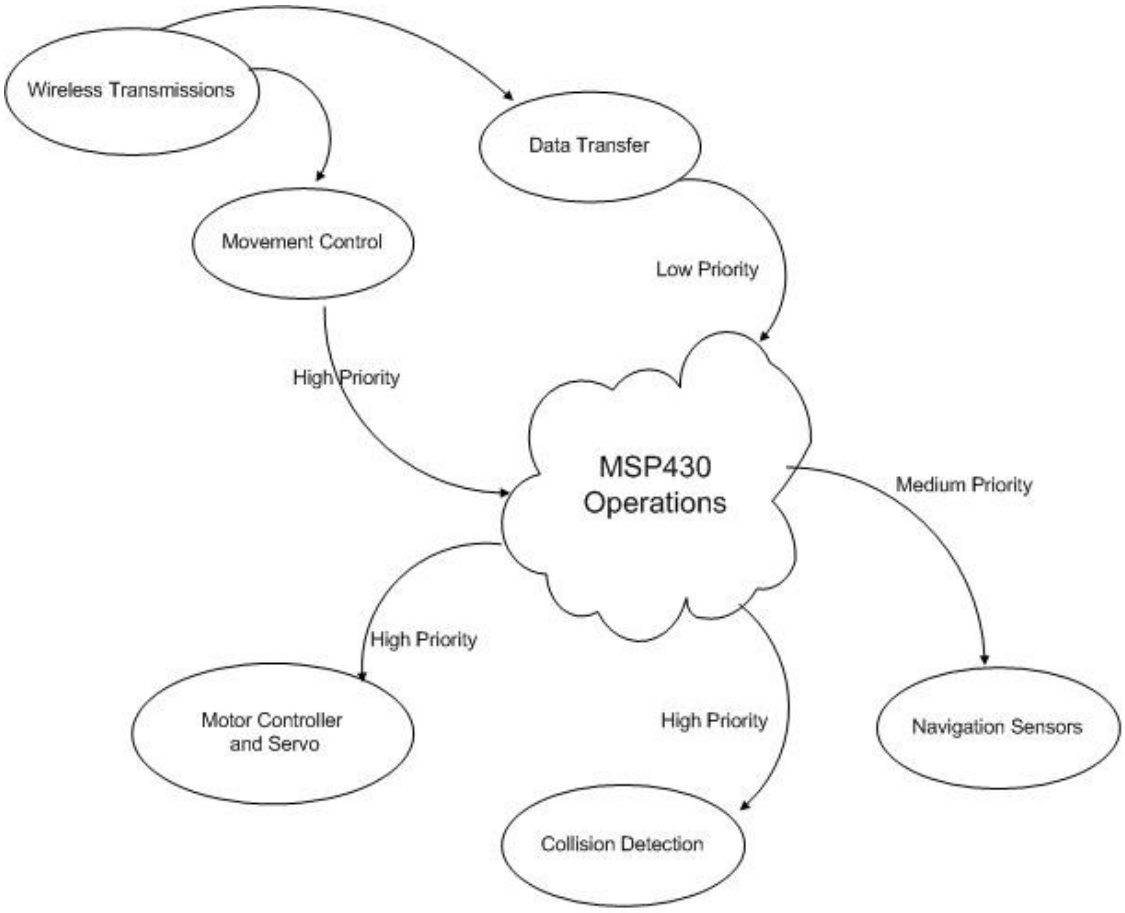

Figure 6: Microcontroller code operation priorities

6. Motor Controller

The motor controller block takes four PWM signals from the microcontroller and uses an H-bridge configuration to provide a bi-directional DC voltage to two separate motors. Two BD6211 motor controller chips from Rohm Semiconductor are used, where

each chip controls a separate motor. The supply voltage for both chips is 5V, requiring the second step-down converter in the power supply system. Each H-bridge uses two switching poles to vary the voltage between 0 and 5V in either polarity. Signal frequencies between 20 and 100kHz are supported. The full 5V of power is achieved by passing a PWM signal with a duty cycle of 98% or above, while a duty cycle of approximately 20% will result in no power being supplied to the motors at all. Two inputs per controller allow us to control the power to each motor in either direction, therefore controlling the speed of the propellers.

7. Propellers

Two propellers are mounted beneath the blimp, one on each side to provide stability, and the ability to turn laterally. The propeller blades are connected to small motors that run on 5V from the motor controller block. The blade and motor system weighs about one-third of an ounce per assembly.

Figure 7: Blimp propeller

In order to connect the propellers to the blimp frame, we ran a small, light-weight carbon fiber beam through the control box holding the electronics. The motor assemblies

were super-glued to the beam and small electrical wires were run along the length of the beam connecting the motors to the circuit board housing the motor controllers. A gear was placed on the metal beam and connected directly to the servo. In this way, the servo could adjust the direction of the propellers by pivoting the entire beam up or down from horizontal. The servo handles up to 180 degrees of turning radius, allowing the propellers to point straight up or straight down. Along with the backwards and forwards motion of the motors, this allows for a full 360 degrees of movement.

The plastic propeller is 6 inches long in diameter and provides up to 1oz of thrust at maximum power. The microcontroller is setup to initiate a turn by dropping the speed of one propeller below the other. For example, a gentle left turn could be achieved by leaving the right propeller at 100% power, and decreasing the left propeller down to 70%. The motors were also placed beneath the center of mass of the blimp, allowing the blimp to roughly pivot on a dime by turning one motor to 100% power forward, and the other motor to 100% power backwards.

8. Navigation Sensor

The purpose of the navigation sensor is to provide data to the microcontroller regarding the acceleration and bearing of the blimp. The block will use three one-axis accelerometers to send an analog voltage signal corresponding to the acceleration experienced by the sensor. The sensor will output a voltage of 1.6V on each output signal if there is no acceleration, with approximately 172mV/g centered around the 1.6V. Capacitors were selected to provide a 0 to 500Hz bandwidth as recommended by the accelerometer data sheet. A sufficiently high frequency is required to pick up any small,

sudden acceleration. Ideally the frequency should match the processing capabilities of the microcontroller so that the integration can keep up with the incoming acceleration data. When the microcontroller reads in the accelerometer output voltage, the voltage level is immediately converted to meters per second squared and fed into the Dead Reckoning system.

The gyroscope will output a voltage corresponding to the rotational velocity of the blimp about the Z-axis. This signal will be used by the microcontroller to determine the bearing of the blimp. The sensor will output an analog voltage of 0.83mV/degree/second. When the blimp is initially powered up, its bearing is set to a relative 0 degrees. As the gyroscope output voltage is measured, it gets integrated to calculate the current bearing of the blimp. The direction the blimp is facing is important when programming it with a particular flight pattern. The pattern is based off the blimp's current position, so if the blimp was told to fly in a circle, the waypoints can be placed in the blimp's current direction of travel.

9. Collision Sensor

The collision sensor block consists of 5 Maxbotics LV-Maxsonar-EZ1 Ultrasonic sensors powered using the 3.3V power rail from the power supply. Each ultrasonic sensor communicates the distance between the sensor and the surrounding environment by outputting a digital signal representing the distance in inches. Because the ultrasonic sound wave produced by one sensor can easily interfere with a neighboring sensor, we needed a way to ensure that only one sensor would be active at a time. We also did not want to assign five serial ports on a microcontroller just to communicate with the five

ultrasonic sensors. Therefore, we pass all five sensors through a MUX, where we toggle one sensor on at a time, wait for a response, and move on to the next. In this manner, no interference takes place and only one serial port is required on the MSP430 microcontroller.

The purpose of the collision sensors is to detect objects in proximity to the blimp. The sensors are rated for object detection up to 254 inches, which will let us reliably "see" objects within twenty feet of the blimp. There will be five sensors placed on the blimp. They will be placed to monitor the front, left, right, top and bottom of the blimp. Although the sensors offer analog and PWM output as well, our system will utilize the asynchronous serial output from the sensors. The output signal is an ASCII capital "R" followed by three ASCII character digits representing the range up to 254 inches, followed by a carriage return.

The values returned by the ultrasonic sensors do not only have to be used for collision detection, but can be used for roughly mapping out the surrounding environment. We measure the distance from the sensors to the exact center of the blimp, and calibrate the sensors so that we are being fed the distances between the environment and the blimp in five different directions. The percentage error of the ultrasonic sensors turned out to be less than 5%, and they worked on almost all materials except for very soft, sound absorbent objects.

10. Blimp Bag

The blimp is what actually provides vertical lift to the system as a whole. The blimp should be light enough that the buoyant force renders it nearly stationary in relation

to the ground. If anything, the blimp must be slightly heavier than the air it displaces, so that as a fail safe, it does not float away without motor power. The 7 foot long blimp bag is able to support up to 10 oz. of weight in addition to the weight of the bag itself. This sufficiently handles the approximate 9 oz. of electronic weight that makes up the control system. The table below shows the weight requirements of the design.

Component	Weight (oz)
Propeller Assembly	0.6
Servo	0.2
4 cell battery	5.2
FR4 PCB	1.22
5 Ultrasonic Sensors	1.29
Bluetooth Transceiver	0.11
Gondola	0.2
Extra wiring	0.31
Total	9.13

Table 4: Weight distribution of blimp components

11. Rechargeable Battery

For mobility and reliability, a powerful yet rechargeable battery was chosen to supply power to the system. The Lithium-Ion battery supplies 6.4V with a maximum current of 18A. It is designed to run for 2400mAH even at a quick pace of a flat 8.4A. The battery can be recharged with a smart charger at a rate of 1.2A. A 4A fuse is connected between the battery and all device circuitry in the unlikely event of a short. The battery's supply is then input to two switching DC converters.

The battery for this device needed to be fairly powerful, rechargeable, and light weight all at the same time. We decided on a particular brand of Lithium-Ion batteries, a LiFePO4 18650 Battery, which uses 4 individual cells (2 parallel groups in series to make

4) to get 6.4V. Lithium-Ion batteries tend to be much lighter weight for the amount of energy density they offer. The battery includes a Polyswitch to prevent damage caused by shorts. Assuming a DC converter efficiency of 90% with the motors and sensors are running at full power, approximately 2A are running at 5V and another 0.2A are running at 3.3V, working out to 1.86A being drawn continuously from the battery. This works out to almost 1.5 hours of minimum battery life since the batteries current capacity increases when being used at a lower rate of discharge.

FINDINGS

This thesis set out to design an autonomous blimp that could navigate using Dead Reckoning. To test the design, we ran numerous flight trials inside a large, enclosed atrium. Seemingly the most successful aspect of the design was the wireless, manual control system. The blimp correctly responded to all wireless commands up to 50 ft away from the windows laptop we were using as the control hub (most laptops contain only a class 2 Bluetooth transceiver, allowing an operating range of 10m). Flying the blimp around with a Six-Axis joystick system such as the PS3 controller was intuitive and easy to control. The collision detection system worked perfectly, reliably detecting objects 20 feet away and veering away from them once it came within 4 feet of an imminent collision. 3D mapping was also a possibility with the distance data collected by the ultrasonic sensors, although we stuck to graphing one dimensional distances for the extent of this project.

The main focus of the thesis was solving the problem of whether or not Dead Reckoning is a viable solution for self-navigation. From a simple budget analysis and power performance review, it is easy to tell that the resources required for Dead Reckoning are much more manageable than those required for GPS; however the question at hand is how well the accuracy compares to GPS.

We quickly determined that the largest difficulty in Dead Reckoning is retaining accuracy over a long period of time. Unlike GPS, Dead Reckoning's error quickly accumulates because even a single erroneous data value will offset the position until the system is reset. In GPS, an uplink with a satellite results in an entirely new position estimation each time. GPS might have a lower accuracy in the short run, but Dead

Reckoning can result in significant error build-up over time. The other major downside to Dead Reckoning is that the calculated position is only relative to its starting location, while GPS can give an absolute geographic coordinate. In this thesis, we focused on addressing the first problem of reducing error as much as possible.

Due to the sensitivity to error in Dead Reckoning, the electronic filtering applied to the accelerometers line as well as the processing speed of the microcontroller are extremely important. Dead Reckoning requires a high degree of processing power because on top of the double integration required to calculate position, a significant amount of trigonometry is needed. When the blimp turns laterally, the accelerometer's X and Y axes are changed so the acceleration data being read in does not always align with the absolute X and Y scale. Therefore, the gyroscope is needed to readjust the acceleration data back to the original coordinate system. This means that on each navigational update, double integration as well as a trigonometry calculation is required, which can be very taxing on a microcontroller.

In the end, the highest degree of accuracy we could reliably achieve still contained approximately 8% error, where error is calculated in terms of how much the calculated distance covered deviated from the actual distance covered in any direction. For example, we ran multi-dimensional tests where we flew the blimp back and forth, measuring the actual amount of distance traveled in each axis of direction and compared it to the blimp's calculated position. Running a total length of 2000cm over the course of 15 seconds, including speed up time and slow down time, the blimp's calculated position was usually off by no more than plus or minus 160cm. On average, the error rate was closer to 4%, only deviating by approximately 80cm from the target distance. After

running the test over 30 times, we found the sample deviation to be almost 100cm wide, explaining the full range from -160cm to +160cm.

In terms of navigating around, the blimp's AI was able to figure out where it needed to go next based off the waypoint system we implemented for Dead Reckoning, however we underestimated the amount of momentum the blimp carries. When the blimp's propellers are completely switched off, the blimp will still coast at almost the same speed for a long period of time. This prevented any tight flight paths with sharp turns and closely knit waypoints. Instead, we found the best results with long flight paths and wide turns. If the blimp failed to turn in time and missed a waypoint, it could take a while to get back on track.

PROBLEMS ENCOUNTERED

Our design was not able to calculate the blimp's location quickly enough to keep up with the incoming acceleration data, forcing a loss of acceleration information. We tried to mask the issue through increased averaging and error reduction algorithms, but our navigation accuracy was never strong enough to compete with that of GPS. Our 8MHz microcontroller did not have the performance necessary for the required number of mathematical operations. When testing the blimp, we would start it moving from a stationary position, and stop it again after a few seconds. Even using slow, non-jerky movements, we found that the microcontroller would sometimes only pick up the positive acceleration or the negative acceleration, but not both reliably. If, for example, the blimp did not register its deceleration to the same degree that it measured acceleration, the blimp would believe it was still traveling forward when in reality the blimp had stopped moving. This results in dramatic error in position after only a short period of time.

One logical error we made in designing the blimp was not taking into account the slight rocking motion of the blimp from side to side as it flies in a straight line. While one gyroscope is used to make sure the X and Y axes are correctly updated when the blimp turns laterally, a second gyroscope is needed to correctly update the X and Z axes if the blimp rocks sideways (where the Y axis is assumed to be forward and backward motion). This is because as the blimp tilts to the side, the X axis is pointed slightly towards the ground while the Z axis is pointed slightly towards the side, causing the real acceleration affects to the X and Z directions to have a lesser affect on the blimp's measurement

system. Fortunately, the blimp does not often rock from side to side unless it is subjected to a strong wind current.

FURTHER IMPROVEMENTS

As noted previously, the most significant possible improvement would be a more powerful microcontroller that could process the Dead Reckoning data quickly enough to more accurately determine the blimp's position. Plenty of microcontrollers can operate in the 100MHz range, and would be much more suitable for our application. Also, noise cancellation on the analog accelerometer lines should be an utmost priority in circuit design as even a single noise spike can completely throw off the system's position for the entirety of its flight.

Furthermore, to help ensure that the blimp does not fly outside the operating Bluetooth transmission range, the signal strength should be taken into account. The operating system could then automatically steer the blimp back to home or at least send the user a warning when the blimp has almost flown outside the operating range. Currently, the blimp is programmed to fly straight down and hover four feet above the ground when it stops communicating with the user. In this manner, the blimp will not accidentally fly away if an error were to occur.

CONCLUSION

With the correct amount of fine-tuning, Dead Reckoning is a viable alternative to GPS in autonomous devices, especially in applications that cover only a short distance or operate for short periods of time. However, for long term use, or environments that require a high degree of accuracy, Dead Reckoning requires careful design to be effective especially in scenarios that are subject to vibration or random, jerky movement. Compared to a GPS solution, Dead Reckoning is much more cost effective and requires only a small degree of power.

As a wireless solution, Bluetooth Class 1 is an excellent tool for medium range transmissions and can be easily integrated into many computer and mobile applications. For an autonomous blimp, Bluetooth was extremely effective in giving the user the ability to manually control the blimp or communicate with it mid-flight. Our custom AI software that utilized a waypoint system to autonomously navigate around was simple and easy to program, although it required an upper-end micro-processor to operate correctly. Altogether, the blimp design was a success and an interesting look into the feasibility of Dead Reckoning for autonomous navigation.

Bibliography

[1] "YARB 1.0 (Yet Another Robotic Blimp)." Surveyor, 02 2010. Web. http://www.surveyor.com/YARB.html

[2] “Blimpduino Kit” DIY Drones, 2012. Web. http://store.diydrones.com/ProductDetails.asp?ProductCode=KT-0000-01

[3] “UAVs” Air and Space Journal. Web. http://www.airpower.maxwell.af.mil/airchronicles/cc/uav.html

[4] “SRV-1 Blackfin Mobile Robot” Surveyor. Web. http://surveyor-corporation.stores.yahoo.net/srrowestkit.html

[5] “AT86RF230” Atmel. Web. http://www.atmel.com/devices/AT86RF230.aspx

[6] “AT86RF230” Atmel. Web. http://www.atmel.com/dyn/products/product_card.asp?part_id=3941

[7] “AT86RF212” Atmel. Web. http://www.atmel.com/dyn/products/product_card.asp?part_id=4349

[8] “Low Power FM Transmitter” Elliott Sound Products. 03 2000. Web. http://sound.westhost.com/project54.htm

[9] “SERIAL PORT ADAPTER OEM 331I” Digikey. Web. http://search.digikey.com/scripts/DkSearch/dksus.dll?Detail&name=809-1011-ND

APPENDICES

APPENDIX A: SCHEMATICS

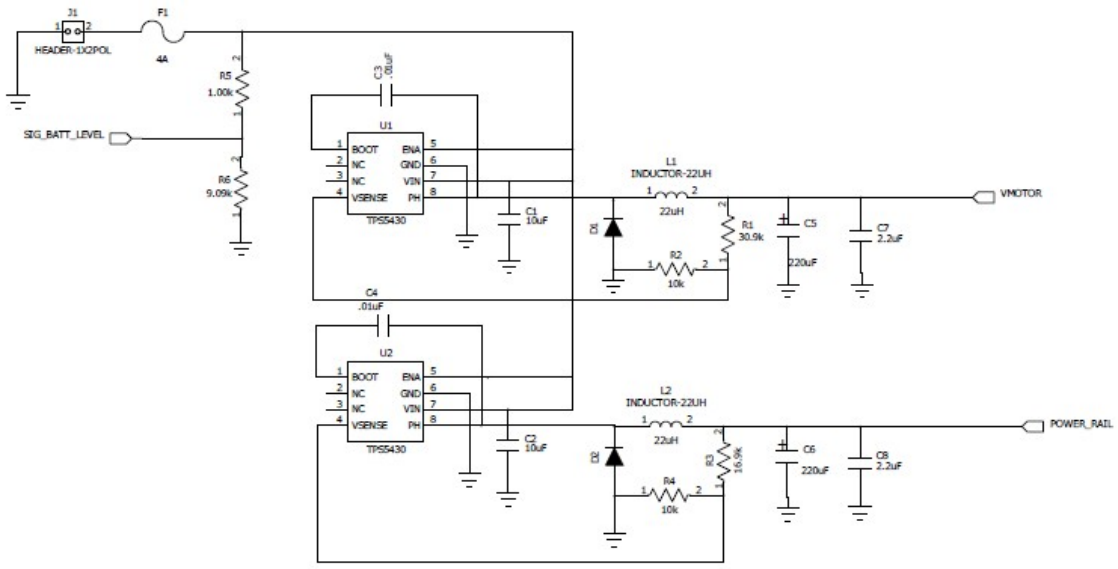

Figure A-1: Schematic of the Power System

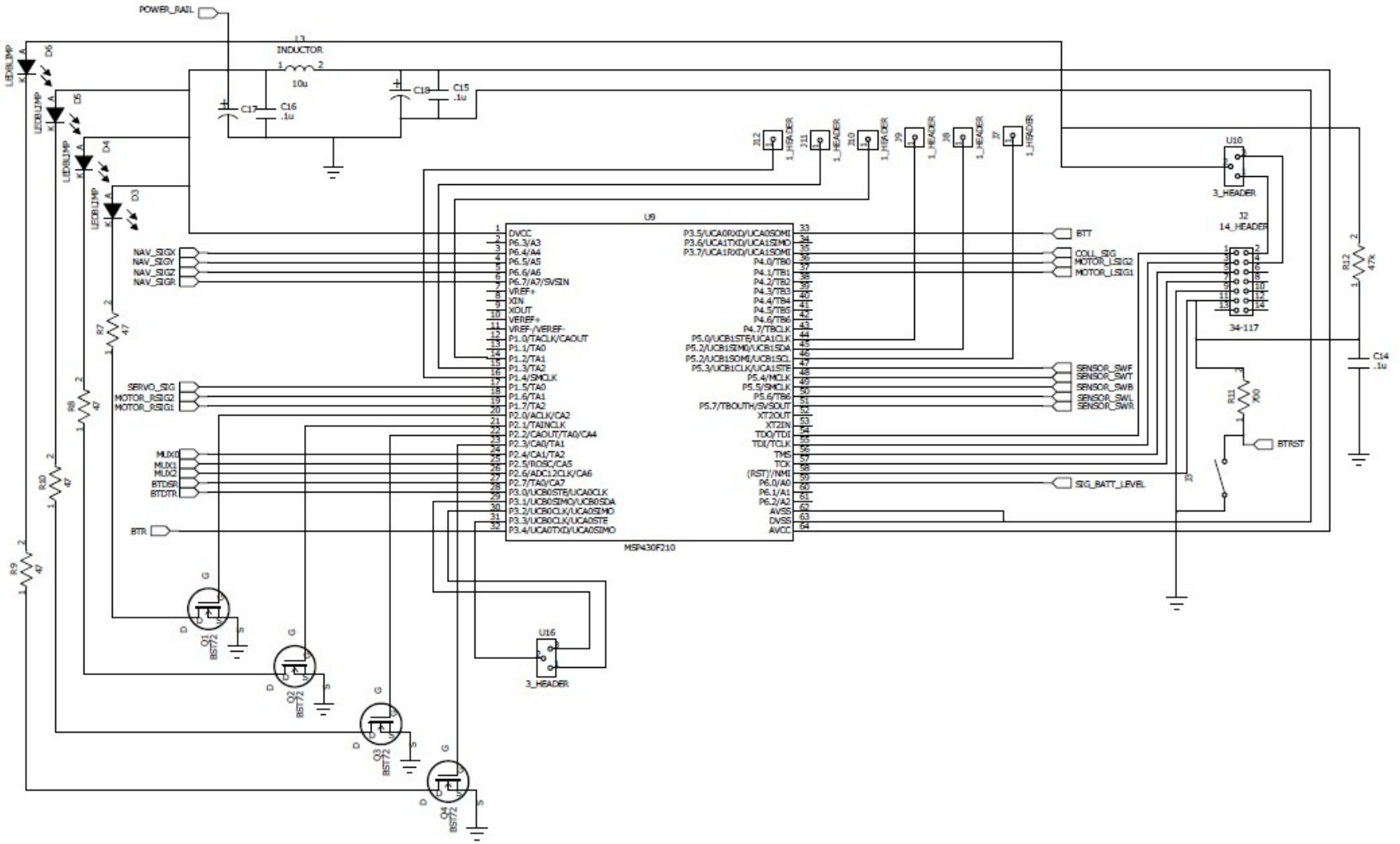

Figure A-2: Schematic of the Microcontroller System

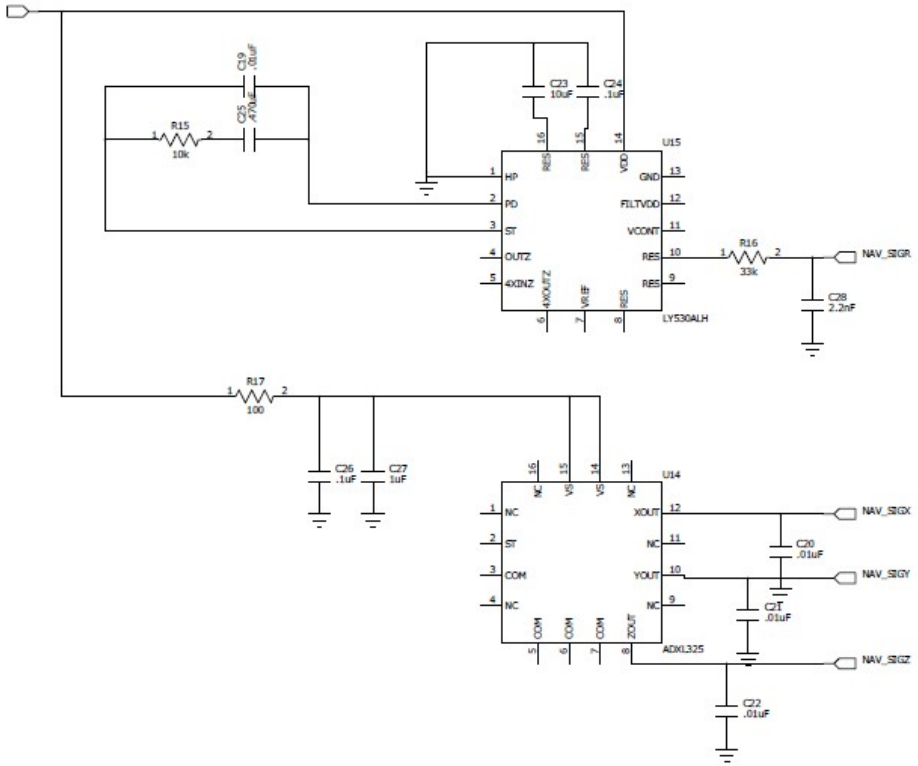

Figure A-3: Schematic of the X and Y Navigation System
(Z axis connected in the same manner)

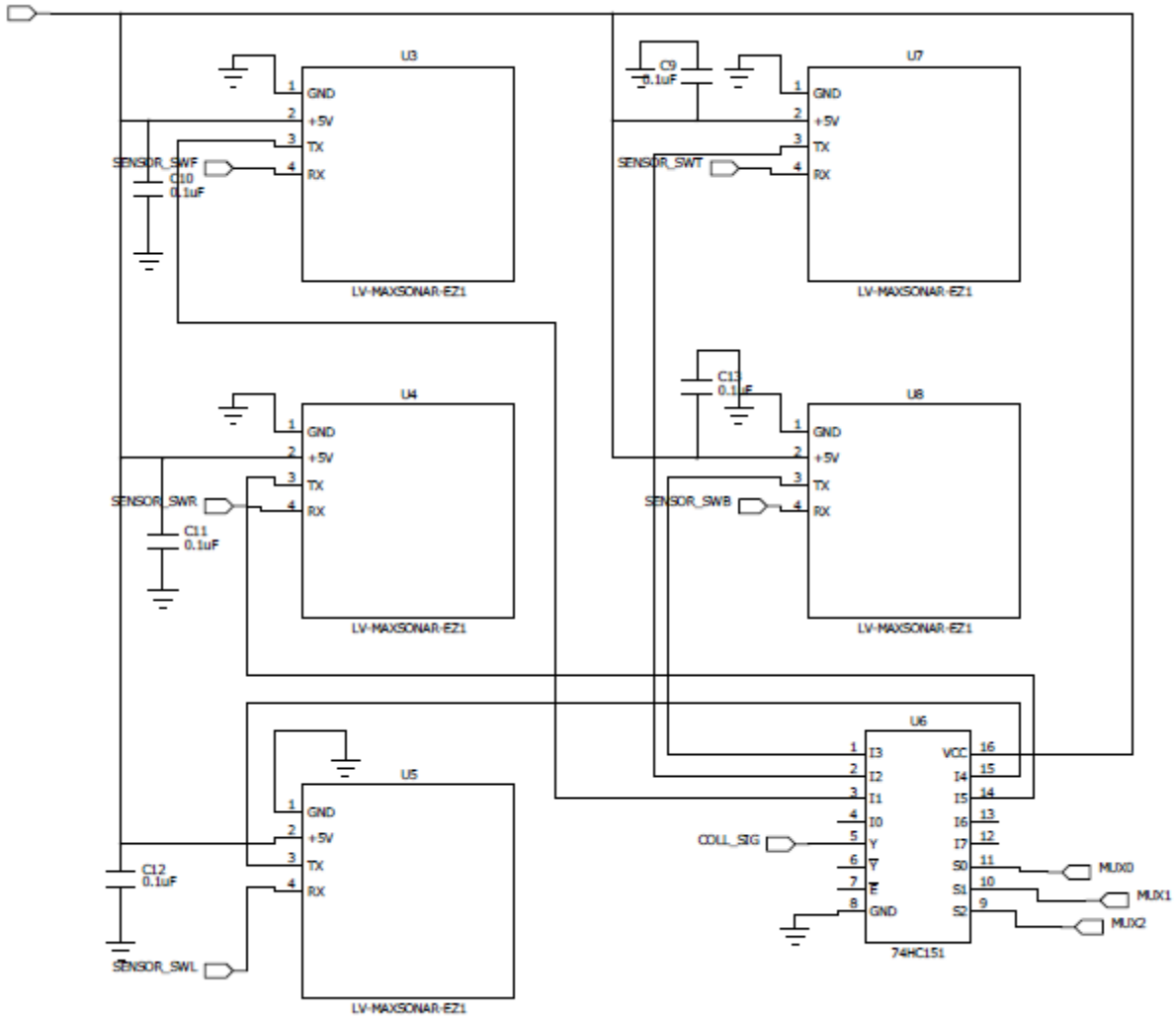

Figure A-4: Schematic of the Collision System

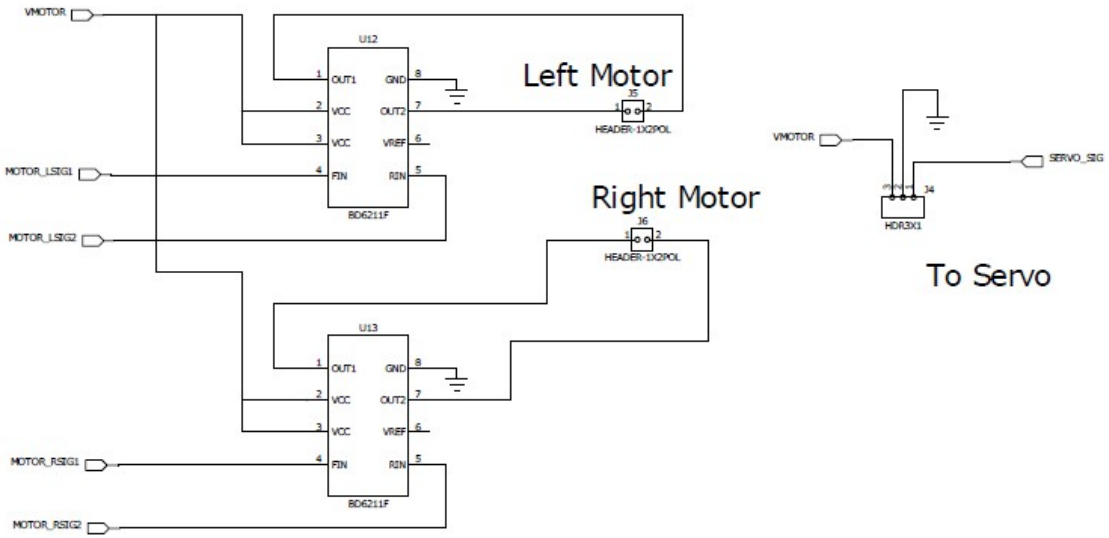

Figure A-5: Schematic of the Motor Control System

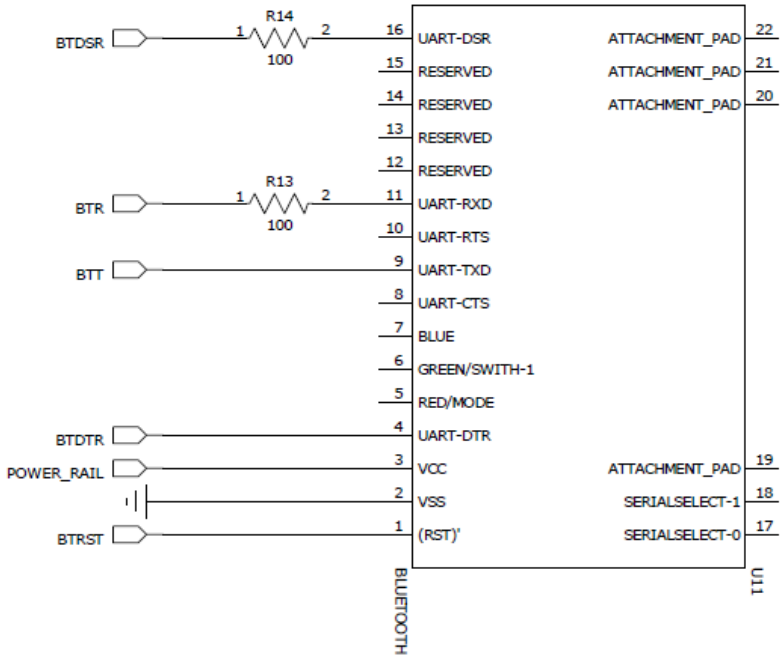

Figure A-6: Schematic of the Bluetooth System

APPENDIX B: MICROCONTROLLER CODE

Main.c

```
/**
 * Thomas Shepherd
 * Cyrus Heick
 * Chris Stoddard
 * ECE 44X, Created: 1/22/11
 *
 * MSP430F2410 microController code
 */

#include <io.h>                         // msp430 defs
#include <msp430/iostructures.h>        // must be inlcuded after io.h!
#include <mspgcc/util.h>                // delay()
#include <signal.h>
#include "bluetooth.h"
#include "collision.h"
#include "leds.h"
#include "logic.h"
#include "navigation.h"
#include "propulsion.h"

void gatherUltraSonicData(void); //read the UltraSonicRange in interrupt only

static int navInt=0;
static int servoTimer = 0; //keeps track of timer counts for setting servo PWM

//Set up Navigation Variables, Used in navigation.c
extern volatile double zaccel;
extern volatile double zspeed;
extern volatile double zpos;
extern volatile double yaccel;
extern volatile double yspeed;
extern volatile double ypos;
extern volatile double xaccel;
extern volatile double xspeed;
extern volatile double xpos;
extern volatile double gspeed;
extern volatile double gpos;

extern volatile int servoPulse; //keeps track of duty cycle of servo PWM
extern volatile int ultraSonicReceived; //holds data returned by UltraSonic
extern volatile int ultraSonicNoData; //if ultraSonicNoData == 0, there is data

int main(void) {



    //-----SETUP INTERNAL CLOCK FREQUENCY-----//
    //Internal 16MHz
    BCSCTL1 = CALBC1_16MHZ; //Turns the internal oscillator to 16MHz
    DCOCTL = CALDCO_16MHZ; //Turns the internal oscillator to 16MHz

    //-----SETUP TIMERA PWM-----//
    //Up/Down Mode, Output Mode 6
    P1SEL |= 0xC0; //P1.6 - TimerA Out1 PWM selected
    P1DIR |= 0xC0;
    TACCTL0 = 0x0C00;
    TACCTL1 = 0x0CC0;
    TACCTL2 = 0x0CC0;
    TACCR0 = 0x0190;
    TACCR1 = 0x0000;
    TACCR2 = 0x0000;
    TACTL = 0x0230;

    //-----SETUP TIMERB PWM-----//
    //Up/Down Mode, Output Mode 6: Toggle/Set
```

```
P1DIR |= 0x20; //For the Servo output
TBCCTL0= 0x38D0;
TBCCTL1 = 0x38C0;
TBCCTL2 = 0x38C0;
P4SEL |= 0x02;
P4DIR |= 0x03;
TBCCR0= 0x0190;
TBCCR1 = 0x0000;
TBCCR2 = 0x0000;
TBCTL = 0x0230;



//-----SETUP UARTA0/Bluetooth IO Pins--------//
P5REN |= 0x06;     //0b00000110
P5DIR |= 0x04;     //0b00000100
P5OUT |= 0x06;     //0b00000110

P3OUT  |= 0x08;     //Init output data of port3
P3SEL  |= 0x30;  //Select port or module -function on port3
P3DIR  |= 0x28;  //Init port direction register of port3

UCA0CTL1 = 0xFF;   // RST
UCA0CTL0 = 0x00;   // 0b00000000
UCA0CTL1 = 0xF0;          // 0b11110000 0xF4 if doesn't work...
UCA0BR0 = 0x11;           // 0b00010001
UCA0BR1 = 0x00;    // 0b00000000
UCA0MCTL = 0x61;          // 0b01100001
IE2 |= 0x01;


    //-----SETUP UARTA1-----//
//Also setup I/O pins for ultrasonic sensors and mux

P5DIR |= 0xF8;
P5OUT &= ~0xF8;
P2DIR |= 0x70;
P2OUT &= ~0x70;
P3SEL |= 0x80;
UCA1CTL1 = 0xFF;
UCA1CTL0 = 0x00;
UCA1CTL1 = 0xF0;
UCA1BR0 = 0x68;
UCA1BR1 = 0x00;
UCA1MCTL = 0x31;
UC1IE = 0x01;

//-----SETUP ADC12------//
//ADC12CTL0 = 0x0C91;
// ADC12CTL1 = 0x0202;


//ADC12CTL0 = SHT0_2 + ADC12ON; // Set sampling time, turn on ADC12
//ADC12CTL0 |= 0xFF80;
//ADC12CTL1 = SHP; // Use sampling timer
//ADC12CTL1 |= 0x2000;
//ADC12CTL1 |= 0x02; //Set single sequence
ADC12MCTL0 = 0x06; //Sample A6 (on P6.6) Zinput
ADC12MCTL1 = 0x05; //Sample A5 (on P6.5) Yinput
ADC12MCTL2 = 0x04; //Sample A4 (on P6.4) Xinput
ADC12MCTL3 = 0x07; //Sample A7 (on P6.7) Gyro Input
ADC12MCTL4 = 0x80; //Sample A0 (on P6.0) Battery Level
ADC12CTL0 = 0x0C91;
ADC12CTL1 = 0x0202;

ADC12IE = 0x00; //Disable Interupts
ADC12CTL0 |= ENC; // Conversion enabled
P6SEL |= 0xF1; // Enable ports 0,4,5,6,7 for ADC

//-----SETUP Watchdog Timer----//
WDTCTL = WDTPW|WDTHOLD; // Init watchdog timer
```

```
    //-----SETUP LED lights-----//
    P2REN |= 0x0F; //0b00001111
    P2DIR |= 0x0F;
    P2OUT &= ~0x0F;

    eint(); //turn on interrupts



    //turnOnUltraSonicSensor(0x1); //turn on one of the UltraSonic Sensors
    setServo(20);


       turnOnLEDs(0x8);
       //long cntLED = 0;
    while (1) {                          // main loop, never ends...

       //blinkLEDs(0x8,cntLED);
       //cntLED++;

       if(getBTFlag())
       {
              BTFlagExecute();
       }

       if(getAModeFlag())
       {
              flyBlimp();
       }

       if(navInt==1)
       {
              updatePos();
       }

    //checkADC();      needed?

    }
}

//pulses every 50us
//Used to change servo duty ratio
interrupt(TIMERB0_VECTOR) wakeup TIMERB0_ISR (void)
{

if(servoTimer%20==0){
    navInt=1;
}
else{
     navInt=0;
     }

//P1OUT = 0x20;
if(servoTimer%400==0)
    {
    P1OUT |= 0x20;
    servoTimer = 0;
    }
    if(servoTimer>=servoPulse)
    {
    P1OUT &= ~0x20;
    }
    servoTimer++;

}

interrupt(USCIAB1RX_VECTOR) USCIAB1RX_ISR (void)
{
    gatherUltraSonicData();
}
```

```
interrupt(USCIAB0RX_VECTOR) USCIAB0RX_ISR (void)
{
    addBuf(UCA0RXBUF);
}

void gatherUltraSonicData()
{
        int ultraSonicTemp = UCA1RXBUF;
         if((ultraSonicTemp==0x52) & (ultraSonicNoData==0))
         {
        ultraSonicNoData = 1;
        ultraSonicReceived = 0x00;
         }
        else
        {
                int multiplier = 1;
                switch(ultraSonicNoData)
                {
                        case 1:
                                multiplier = 100;
                                break;
                        case 2:
                                multiplier = 10;
                                break;
                        case 3:
                                multiplier = 1;
                                break;
                        case 4: multiplier = 0;
                                break;
                }
                ultraSonicReceived += ((ultraSonicTemp - 0x30)*multiplier);
                if(ultraSonicNoData>=4)
                {
                        ultraSonicNoData = 0;
                }
                else ultraSonicNoData++;
        }
}
```

Logic.c

```
#include <io.h>                         // msp430 defs
#include <msp430/iostructures.h>        // must be inlcuded after io.h!
#include <mspgcc/util.h>                // delay()
#include <signal.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "leds.h"
#include "logic.h"
#include "bluetooth.h"
#include "navigation.h"
#include "propulsion.h"

int xcoor[500];
int ycoor[500];
int zcoor = 0;
int coorCnt =0;
int arrayLength = 0;
float pi = M_PI; //pi = 3.141592654;
int direction = 3;
int aMode = 0;

/*
        Takes array passed by BlueTooth and calls correct functions.
*/
void readBufArray(char array[], int length)
{
```

```
char send[50];
int sndCntr = 0;
int rad = 0;
int len = 0;
int wid = 0;
int success = 3;

switch(array[0])
{
case CIRCLE:
	rad = char2int(array[1],array[2],array[3],array[4]);
	success = makeCircle(rad, array[5]-'0');
	if(success == 1)
	{
		strcpy(send, "Success!");
		sndCntr = 8;
	}
	else
	{
		if(success == 0)
		{
			strcpy(send, "Failure!");
			sndCntr = 8;
		}
		else
		{
			strcpy(send, "ERROR!");
			sndCntr = 6;
		}
	}
	setAModeFlag();
	break;
case SQUARE:
	len = char2int(array[1],array[2],array[3],array[4]);
	wid = char2int(array[5],array[6],array[7],array[8]);
	success = makeSquare(len, wid, array[9]-'0');
	if(success == 1)
	{
		strcpy(send, "Success!");
		sndCntr = 8;
	}
	else
	{
		if(success == 0)
		{
			strcpy(send, "Failure!");
			sndCntr = 8;
		}
		else
		{
			strcpy(send, "ERROR!");
			sndCntr = 6;
		}
	}
	setAModeFlag();
	break;
case HOME:
	strcpy(send, "ET GO HOME!!!");
	sndCntr = 13;
	setAModeFlag();
	break;

case COLLDETECTON:
	break;

case COLLDETECTOFF:
	break;

case MANAULMODE:

	setTask(array);
```

```
                manualMode();
                clearAModeFlag();
                break;

        case CALIBRATE:
                break;

        case SCANNINGON:
                break;

        case SCANNINGOFF:
                break;

        default:
                strcpy(send, "No option selected.");
                sndCntr = 19;
                break;
        }
        sendDataBT(send, sndCntr);
}

/*
        Makes a circle. radius is in cm, direction = 1 for Clockwise, 0 for counter-
clockwise.
*/
int makeCircle(int radius, int dir)
{
        double angle = 0;
        direction = dir;
        for(int a=0; a<360; a++)
        {
                angle = a*(pi/180);
                if(direction == 1)
                {
                        xcoor[a] = (int)(-1*((cos(angle)*radius)-radius+getPos('x')+.5));
                }
                else
                {
                        if(direction == 0)
                        {
                                xcoor[a] = (int)((cos(angle)*radius)-radius+getPos('x')+.5);
                        }
                        else
                        {
                                return 0;
                        }
                }
        ycoor[a] = (int)((sin(angle)*radius)+getPos('y')+.5);
        }
        zcoor = (int)(getPos('z')+.5);
        arrayLength = 360;

        return 1;
}

/*
        Makes a square. length and width are in cm, direction = 1 for Clockwise, 0 for
counter-clockwise.
*/
int makeSquare(int length, int width, int direction)
{
        xcoor[0] = (int)(getPos('x')+.5);
        xcoor[1] = (int)(getPos('x')+.5);
        if(direction == 1)
        {
                xcoor[2] = width+(int)(getPos('x')+.5);
                xcoor[3] = width+(int)(getPos('x')+.5);
        }
        else
        {
                if(direction == 0)
```

```
                {
                        xcoor[2] = -1*width+(int)(getPos('x')+.5);
                        xcoor[3] = -1*width+(int)(getPos('x')+.5);
                }
                else
                {
                        return 0;
                }
        }
        ycoor[0] = (int)(getPos('y')+.5);
        ycoor[1] = length+(int)(getPos('y')+.5);
        ycoor[2] = length+(int)(getPos('y')+.5);
        ycoor[3] = (int)(getPos('y')+.5);

        zcoor = (int)(getPos('z')+.5);

        arrayLength = 4;

        return 1;
}

/*
        Returns x coordinate for given counter value.
*/
int getXCoordinate(int deg)
{
        return xcoor[deg];
}

/*
        Returns y coordinate for given counter value.
*/
int getYCoordinate(int deg)
{
        return ycoor[deg];
}

/*
        Returns z coordinate for given counter value.
*/
int getZCoordinate(void)
{
        return zcoor;
}

/*
        Returns array length of coordinates.
*/
int getArrayLength(void)
{
        return arrayLength;
}

/*
        converts ascii char values to an int. If wanting to pass 0 value pass char '0'.
*/
int char2int(char num1000, char num100, char num10, char num1)
{
        return ((num1000 - '0')*1000+(num100 - '0')*100+(num10 - '0')*10+(num1 - '0'));
}

void flyBlimp(void)
{

        int absValue = 0;
        int destx = 0;
        int desty = 0;
        int destz = 0;
        int destg = 0;

        int currentx = (int)(getPos('x')+.5);
```

```
	int currenty = (int)(getPos('y')+.5);
	int currentz = (int)(getPos('z')+.5);
	double currentg = (getPos('g')+.5);

	int nextx = getXCoordinate(coorCnt);
	int nexty = getYCoordinate(coorCnt);
	int nextz = getZCoordinate();
	double nextg = 0;

	absValue  =  (int)sqrt((float)(((currentx-nextx)^(2))  +  ((currenty-nexty)^(2))  +
((currentz-nextz)^(2))));
	if(absValue < 50)
		coorCnt++;
	if(coorCnt == getArrayLength())
		coorCnt = 0;


	nextx = getXCoordinate(coorCnt);
	nexty = getYCoordinate(coorCnt);
	nextz = getZCoordinate();

	destx = nextx-currentx;
	desty = nexty-currenty;
	destz = nextz-currentz;

	if(destx == 0)
		return;
	nextg = atan(desty/destx);

	if(destx > 0 && desty > 0 && direction == 0)
	{
		destg = (nextg - currentg);
	}
	else
	{
		if(destx > 0 && desty > 0 && direction == 1)
		{
			destg = (pi + nextg - currentg);
		}
		else
		{
		if(destx > 0 && desty < 0 && direction == 0)
		{
			destg = ((2*pi) + nextg - currentg);
		}
		else
		{
		if(destx > 0 && desty < 0 && direction == 1)
		{
			destg = (pi + nextg - currentg);
		}
		else
		{
		if(destx < 0 && desty > 0 && direction == 0)
		{
			destg = (pi + nextg - currentg);
		}
		else
		{
			if(destx < 0 && desty > 0 && direction == 1)
			{
				destg = ((2*pi) + nextg - currentg);
			}
			else
			{
				if(destx < 0 && desty < 0 && direction == 0)
				{
					destg = (pi + nextg - currentg);
				}
				else
				{
```

```
                                        if(destx < 0 && desty < 0 && direction == 1)
                                        {
                                                destg = (nextg - currentg);
                                        }
                                        else
                                        {
                                                sendDataBT("Error Flying!", 13);
                                        }
                                }
                        }
                }
        }
}
}
}
        if(abs(destg) >= (pi/2))
        {
                if(destg > 0)
                {
                        hardTurn(0, 1);
                }
                else
                {
                        if(destg < 0)
                        {
                                hardTurn(1, 0);
                        }
                        else
                        {
                                sendDataBT("Error Destination1!", 19);
                        }
                }
        }
        else
        {
                if(abs(destg) < (pi/2))
                {
                        calculateMotorSpeed(destg);
                }
                else
                {
                        sendDataBT("Error Destination2!", 19);
                }
        }
}

void calculateMotorSpeed(double rad)
{

        double multiplier = 3.2;
        double percent = 63.662;
        double engineSpeed = 0;

        //Gives Int 0-320
        engineSpeed = multiplier*(100-(percent*abs(rad)));

        if(rad > 0)
        {
                leftForward(engineSpeed);
        }
        else
        {
                if(rad<0)
                {
                        rightForward(engineSpeed);
                }
                else
                {
                        sendDataBT("Error Motor Speed!", 18);
                }
        }
```

```
        int q;
        for(q=0; q<1000; q++)
        {
                delay(10000);
        }

        //leftForward(getDefaultSpeed());
        //rightForward(getDefaultSpeed());

}

void hardTurn(int left, int right)
{
        if(left == 0 && right == 1)
        {
                //leftReverse(getDefaultSpeed());
                //rightForward(getDefaultSpeed());
        }
        else
        {
                if(left == 1 && right == 0)
                {
                        //leftForward(getDefaultSpeed());
                        //rightReverse(getDefaultSpeed());
                }
                else
                {
                        sendDataBT("Error Hard Turn!", 16);
                }
        }

        int q;
        for(q=0; q<1000; q++)
        {
                delay(10000);
        }

        //leftForward(getDefaultSpeed());
        //rightForward(getDefaultSpeed());
}

void setAModeFlag(void)
{
        aMode = 1;
}

int getAModeFlag(void)
{
        return aMode;
}

void clearAModeFlag(void)
{
        aMode = 0;
}
```

Collision.c

```
#include <io.h>                         // msp430 defs
#include <msp430/iostructures.h>        // must be inlcuded after io.h!
#include <mspgcc/util.h>                // delay()
#include <signal.h>
#include "collision.h"
#include "propulsion.h"



volatile int ultraSonicReceived = 0;
volatile int ultraSonicNoData = 0;
```

```
volatile int range[5]; //Array where 0-255 ranges are stored

int getUltraSonicRange(void)
{
    while(ultraSonicNoData!=0);
    return ultraSonicReceived;
}

void turnOnUltraSonicSensor(int sensorNum)
{
    switch(sensorNum)
    {
        case 0x1:
               P5OUT &= ~0xF8; //turn off all sensors
               P5OUT |= 0x08; //turn on sensor1
               P2OUT &= ~0x70;
               P2OUT |= 0x10; //pass sensor1 through MUX
               break;
        case 0x2:
               P5OUT &= ~0xF8; //turn off all sensors
               P5OUT |= 0x10; //turn on sensor1
               P2OUT &= ~0x70;
               P2OUT |= 0x20; //pass sensor1 through MUX
               break;
        case 0x3:
               P5OUT &= ~0xF8; //turn off all sensors
               P5OUT |= 0x20; //turn on sensor1
               P2OUT &= ~0x70;
               P2OUT |= 0x30; //pass sensor1 through MUX
               break;
        case 0x4:
               P5OUT &= ~0xF8; //turn off all sensors
               P5OUT |= 0x40; //turn on sensor1
               P2OUT &= ~0x70;
               P2OUT |= 0x40; //pass sensor1 through MUX
               break;
        case 0x5:
               P5OUT &= ~0xF8; //turn off all sensors
               P5OUT |= 0x80; //turn on sensor1
               P2OUT &= ~0x70;
               P2OUT |= 0x50; //pass sensor1 through MUX
               break;
    }
}

/**
When an object is detected in front of the blimp, determine which side has more space and
turn
until there is room to move forward again.
**/
void collisionMode(void){
     //Copy distances to Array for each sensor
     for(int i=0;i<5;i++){
     turnOnUltraSonicSensor(i);
     range[i]=getUltraSonicRange();
     if(getRange(FRONTSENSOR)<75){
          stop();
               if(getRange(LEFTSENSOR)>getRange(RIGHTSENSOR)) //Determine which direction
to turn
                    {
                      while(getRange(FRONTSENSOR)<150){ //Turn until there is space ahead
                           turnLeft();
                      }
                    }
               else{
                         while(getRange(FRONTSENSOR)<150){
                            turnRight();
                         }
                    }
               stop();
          }
```

```
    }
}


//Our sensor cannot measure fractions of cm's so i'm returning an int for processing ease
int getRange(int num){
    int distance=range[num]*2.54; //Return distance in cm
    return distance;
    }
```

Bluetooth.c

```
#include <io.h>                         // msp430 defs
#include <msp430/iostructures.h>        // must be inlcuded after io.h!
#include <mspgcc/util.h>                // delay()
#include <signal.h>
#include "bluetooth.h"
#include "leds.h"
#include "logic.h"


char ans[50];
int bufCnt = 0;
int ansLength = 0;
int EOT = 0x04;
int BTFlag = 0;

/*
       Used to program the bluetooth in the case of a restore default settings.
*/
void programBT(void)
{
       enterATMode();
       char name[25] = "AT*AGLN=\"BLIMP\",1"; //17 Bytes - Names Blimp
       char flow[25] = "AT*AMRS=8,1,1,1,2,0,1"; //21 Bytes - Sets Flow Control
       char secKey[25] = "AT*AGFP=\"12w35tg7\",1"; //20 Bytes - Sets Fixed Security Key

       sendATBT(name, 17);
       sendATBT(flow, 21);
       sendATBT(secKey, 20);

}

/*
       Takes an array of ints and pass its to the buffer one at a time to be sent via
bluetooth, while in data mode.
*/
void sendDataBT(char array[], int length)
{
       int arrayCnt = 0;
       while(arrayCnt < length)
       {
              sendBT(array[arrayCnt]);
              arrayCnt++;
       }
       sendBT(0x04);
}

/*
       Takes a char array and sends it to the bluetooth to edit settings. Size is of the
array being passed in.
*/
void sendATBT(char array[], int size)
{
       int arrayCnt = 0;
       while(array[arrayCnt] < size)
       {
              while(!(IFG2 & BIT1));
              UCA0TXBUF = array[arrayCnt];
              arrayCnt++;
       }
```

```
        while(!(IFG2 & BIT1));
        sendBT(0x0D);
}

/*
        Enters Data Mode in the bluetooth if in AT mode currently.
*/
void enterDataMode(void)
{
        char data[7] = "AT*ADDM";
        int p;
        for(p=0; p<7; p++)
        {
                sendBT(data[p]);
        }
        sendBT(0x0D);
}

/*
        Takes Bluetooth from Datamode to ATmode.
*/
void enterATMode(void)
{
        char at = 0x2F;

        int q;
        for(q=0; q<1000; q++)
        {
                delay(10000);
        }

        int j;
        for (j = 0; j<3;j++)
        {
                while(!(IFG2 & BIT1));
                UCA0TXBUF = at;
        }

        int p;
        for(p=0; p<1000; p++)
        {
                delay(10000);
        }
}

/*
        Sends one byte to Bluetooth.
*/
void sendBT(char data)
{
        while(!(IFG2 & BIT1));
        UCA0TXBUF = data;
}

/*
        Resets buffer Counter
*/
void bufCntReset(void)
{

        ansLength = bufCnt;
        bufCnt = 0;
}

/*
        Called by the bluetooth interrupt, takes receive buffer and moves it into array.
*/
void addBuf(char var)
{
        if (var == EOT)
        {
```

```
                bufCntReset();
                setBTFlag();
                return;
        }
        ans[bufCnt] = var;
        bufCnt++;
}

void BTFlagExecute(void)
{
        readBufArray(ans, bufCnt);
        BTFlag = 0;
}

void setBTFlag(void)
{
        BTFlag = 1;
}

int getBTFlag(void)
{
        return BTFlag;
}
```

Navigation.c

```
#include <io.h>                         // msp430 defs
#include <msp430/iostructures.h>        // must be inlcuded after io.h!
#include <mspgcc/util.h>                // delay()
#include <signal.h>
#include <stdlib.h>
#include "leds.h"
#include <math.h>
#include "navigation.h"

volatile double zaccel=0;
volatile double zspeed=0;
volatile double zpos=0;
volatile int zin=2075;
volatile double yaccel=0;
volatile double yspeed=0;
volatile double ypos=0;
volatile int yin=2028;
volatile double xaccel=0;
volatile double xspeed=0;
volatile double xpos=0;
volatile int xin=2028;
volatile int gin;
volatile double gpos=0;
volatile double gspeed=0;
volatile double battery;

double xstat=2028;
double ystat=2048;
double zstat=2260;
double gstat=2048;

static int batteryFlag; //Set to 1 if battery goes low.

int count=0; //Set speed and pos to 0 for a certain amount of iterations

void checkADC(void)
{
   ADC12CTL0 |= ADC12SC; //start new ADC conversions


   if(ADC12CTL1 & 0x01){
                turnOnLEDs(0xFF);
                }
                else{
```

```
                    turnOffLEDs(0xFF);
                    }

  // while((ADC12IFG & 0x1F)==0); //I think this is where my code is having issues. Only
set to MEM0.

   zin=ADC12MEM0;
   yin=ADC12MEM1;
   xin=ADC12MEM2;
   gin=ADC12MEM3;
   battery=ADC12MEM4;

 //Delay the start of measurements
   if (count<20){
      xspeed=0;
      yspeed=0;
      zspeed=0;

      xpos=0;
      ypos=0;
      zpos=0;

      count++;
      }

  //If battery divider falls below 540mV
   if(battery<670){
       setBatteryFlag(1);
       }

}



void updatePos(void) //Called every 1ms by main
{
     //turnOnLEDs(0xF);
     //Constants for comparison


     gspeed=(gin-gstat);
    // gpos=gpos+(gspeed*0.00005);
    // heading=gpos/180*PI;


    // xvoltage=(xin/4096*3.3);
    // yvoltage=(yin/4096*3.3);
    // double zvoltage=(zin/4096*3.3);
    // gvoltage=(gin/4096*3.3);

     //22 units per (m/s^2)
     xaccel=(xin-xstat)*cos((double) gpos);
     yaccel=(yin-ystat)*sin((double) gpos);
     zaccel=(zin-zstat);//22.02;

     //Velocity Measured in units/millisecond
     xspeed=xspeed+(xaccel);
     yspeed=yspeed+(yaccel);
     zspeed=zspeed+(zaccel);
     gspeed=gin;

     //Positioin Measured in units/millisecond
     xpos=xpos+(xspeed);
     ypos=ypos+(yspeed);
     zpos=zpos+(zspeed);
     gpos=gpos+(gspeed);
  /**
    //Method to send up to 4 digits output to bluetooth
    char buf[4];
    int i;
    itoa((int) zpos, buf,10);
```

```
    int length=arLen((int) zpos);
    char sendData[length];
    for(i=0; i<length; i++){
                sendData[i]=buf[i];
                }
                sendDataBT(sendData, length);

    **/
}


//Get array length for bluetooth send.
int arLen(int a){
    if(a>0){
        int b=a/10;
        return arLen(b)+1;
        }
    else if(a<0){
         a=abs(a);
         int b=a/10;
         return arLen(b)+2;
         }
    else{
        return 0;
        }

}

double getSpeed(char a){
       if(a=='x'){
                 return xspeed*2.2015;
                 }
       if(a=='y'){
                 return yspeed*2.2015;
                 }
       if(a=='z'){
                 return zspeed*2.2015;
                 }
       if(a=='g'){
                  return gspeed*59.026;
                  }
            return 0;
}

double getPos(char a){
       if(a=='x'){
                 return xpos*0.0022015;
                 }
       if(a=='y'){
                 return ypos*0.0022015;
                 }
       if(a=='z'){
                 return zpos*0.0022015;
                 }
       if(a=='g'){
                  return gpos*0.059026;
                  }
            return 0;
}

void calibrate(void){
     double i=0;
     double x=0;
     double y=0;
     double z=0;
     double g=0;
     for(i=0;i<10;i++){
          x=x+getPos('x');
          y=y+getPos('y');
          z=z+getPos('z');
          g=g+getPos('g');
```

```
          }
     xstat=x/10;
     ystat=y/10;
     zstat=z/10;
     gstat=g/10;
}

void setBatteryFlag(int flag)
{
     batteryFlag=flag;
}

int getBatteryFlag(void)
{
     return batteryFlag;
}
```

Propulsion.c

```
#include <io.h>                          // msp430 defs
#include <msp430/iostructures.h>         // must be inlcuded after io.h!
#include <mspgcc/util.h>                 // delay()
#include <signal.h>
#include <stdlib.h>
#include "logic.h"
#include "propulsion.h"


int motorSpeed=0x100;
int flat=90; // Constant for flat servo angle


volatile int servoPulse = 0; //Set the servo pulse according to angle

char task[8];
char pos[4];


//Basic Motor Controls
void leftForward(int speed)
{
     TACCR1=speed;
     TACCR2=0x0000;
}

void rightForward(int speed)
{
     int spd=0x0190-speed;
     TBCCR1=spd;
     TBCCR2=0x0000;
}

void leftReverse(int speed)
{
     //int spd=0x0190-speed;
     TACCR1= 0x0000;
     TACCR2= speed;
}

void rightReverse(int speed)
{
     //int spd=0x0190-speed;
     TBCCR1=speed;
     P4OUT |= 0x01;
}



//Advanced Motor Functions

void forward(void)
```

```
{
     rightForward(motorSpeed);
     leftForward(motorSpeed);
}

void reverse(void)
{
     rightReverse(motorSpeed);
     leftReverse(motorSpeed);
}

void veerLeft(void)
{
     rightForward(motorSpeed);
     leftForward(motorSpeed/2);
}

void veerRight(void)
{
     leftForward(motorSpeed);
     rightForward(motorSpeed/2);
}

void turnLeft(void)
{
     rightForward(motorSpeed);
     leftReverse(motorSpeed);
}

void turnRight(void)
{
     leftForward(motorSpeed);
     rightReverse(motorSpeed);
}

void stop(void)
{
     rightForward(0);
     leftForward(0);
}

void dive(int speed, int angle)
{
     rightForward(speed);
     leftForward(speed);
     setServo(angle);
}



//Set the servo angle
void setServo(int angle)
{
     servoPulse=((angle-1)/5)+12;
}


//Sets the bytes after 'M' into the task array
void setTask(char array[])
{
     int length;
     if(array[1]== SETSERVO){
          length=5;
          }
     else{
          length=1;
          }
     for(int i=0;i<length;i++){
             task[i]=array[i+1];
             }
}
```

```
void manualMode(void)
{

     switch(task[0]){
          case FORWARD:
               forward();
               break;
          case REVERSE:
               reverse();
               break;
          case TURNLEFT:
               turnLeft();
               break;
          case TURNRIGHT:
               turnRight();
               break;
          case VEERLEFT:
               veerLeft();
               break;
          case VEERRIGHT:
               veerRight();
               break;
          case SETSERVO:
               for(int i=0;i<4;i++){
                       pos[i]=task[i+1];
                       }
               int ang=atoi(pos);
               setServo(ang);
               break;
          }

}
```

APPENDIX C: PRODUCT PHOTOS

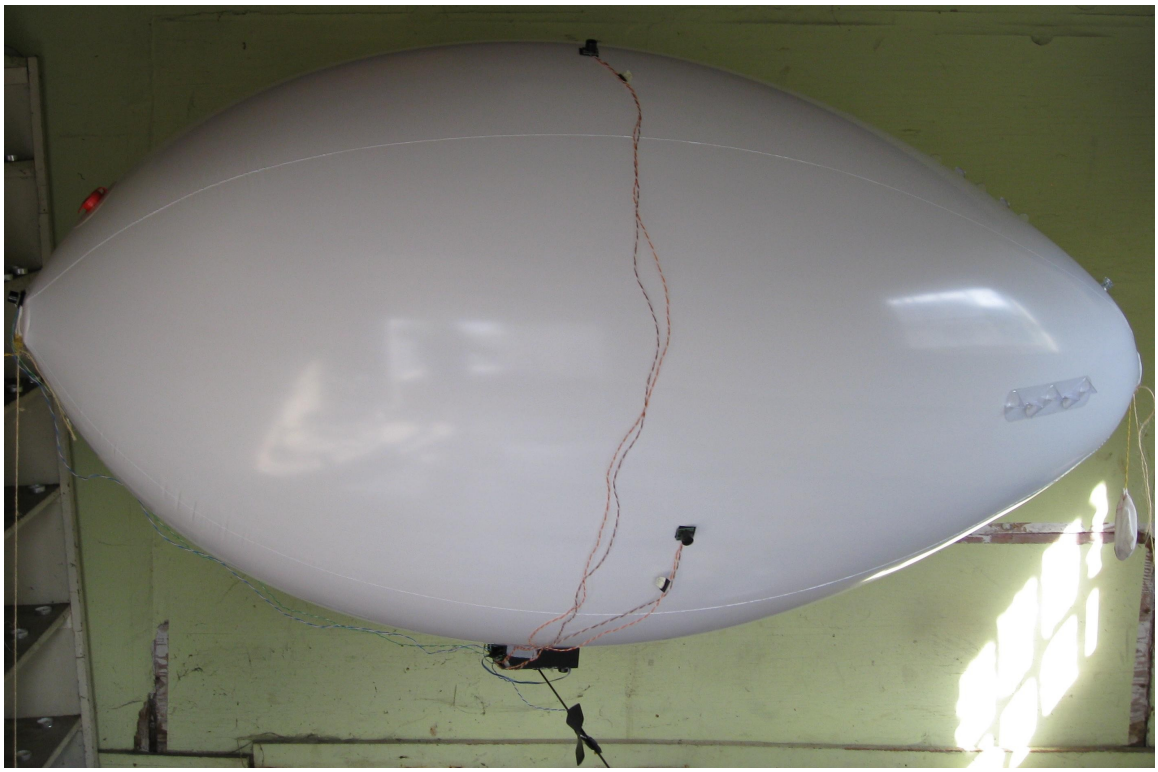

Figure C-1: Blimp in storage

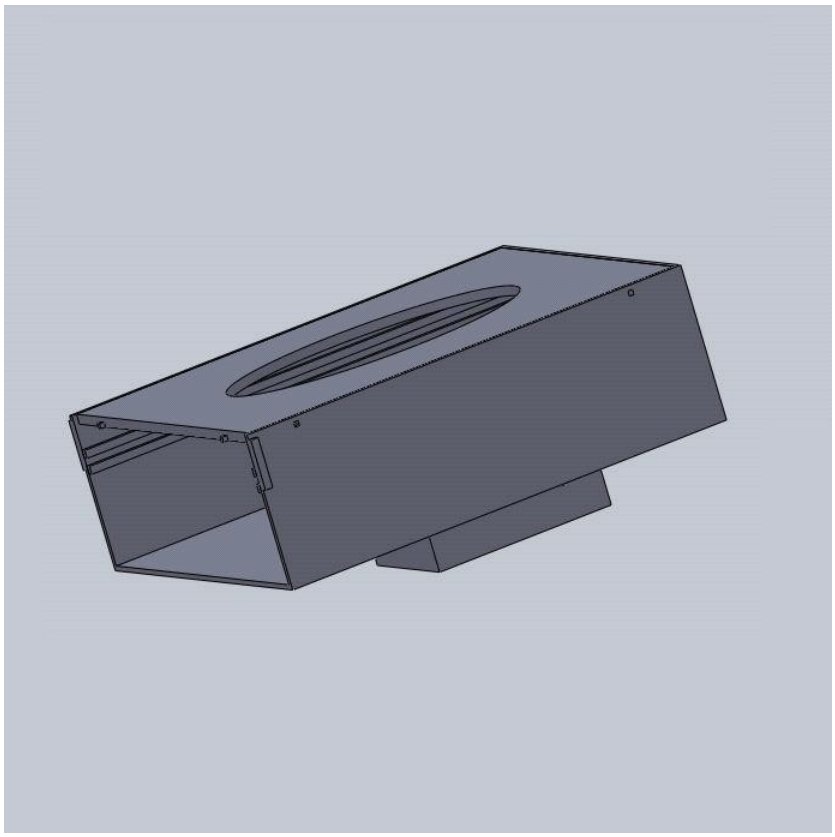

Figure C-2: Gondola model

Figure C-3: Ultrasonic Sensor

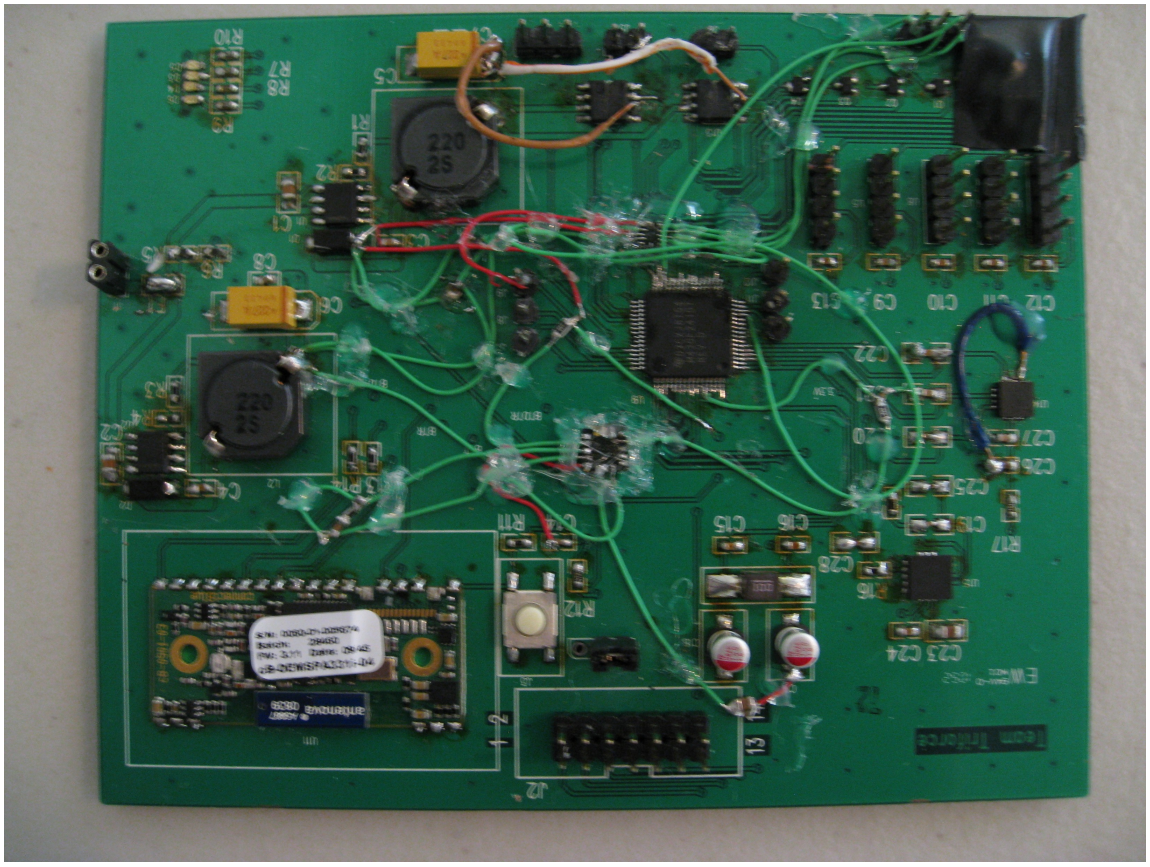

Figure C-4: Control System PCB

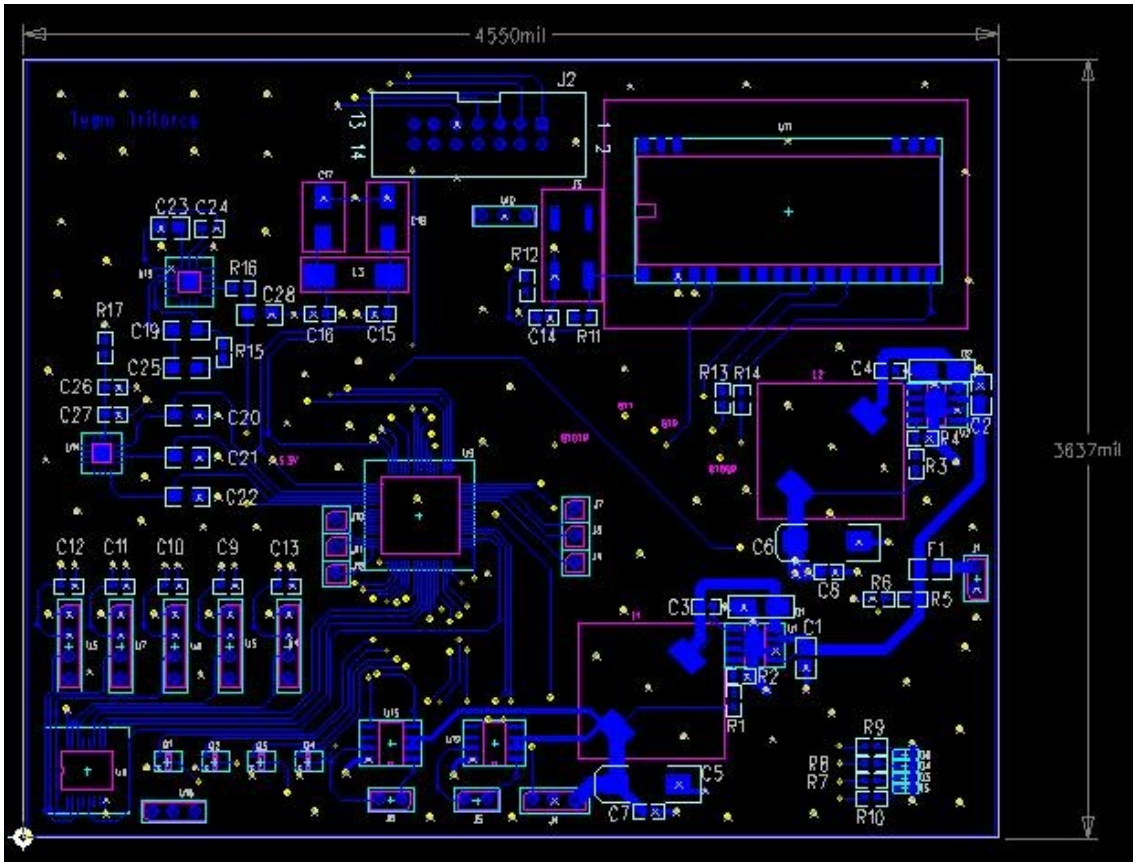

Figure C-5: Control circuit layout

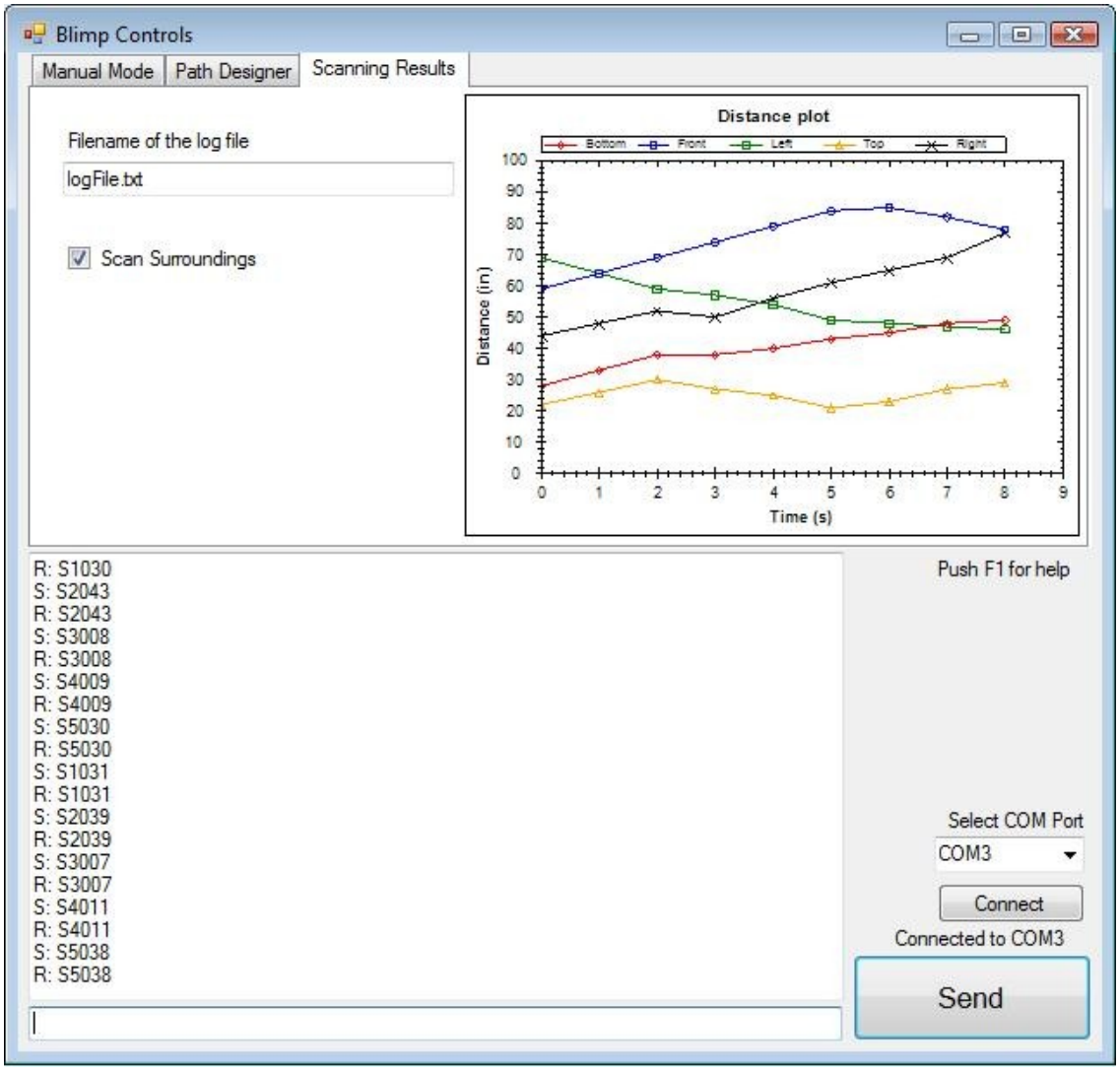

Figure C-6: Host computer control GUI – Scanning Results tab

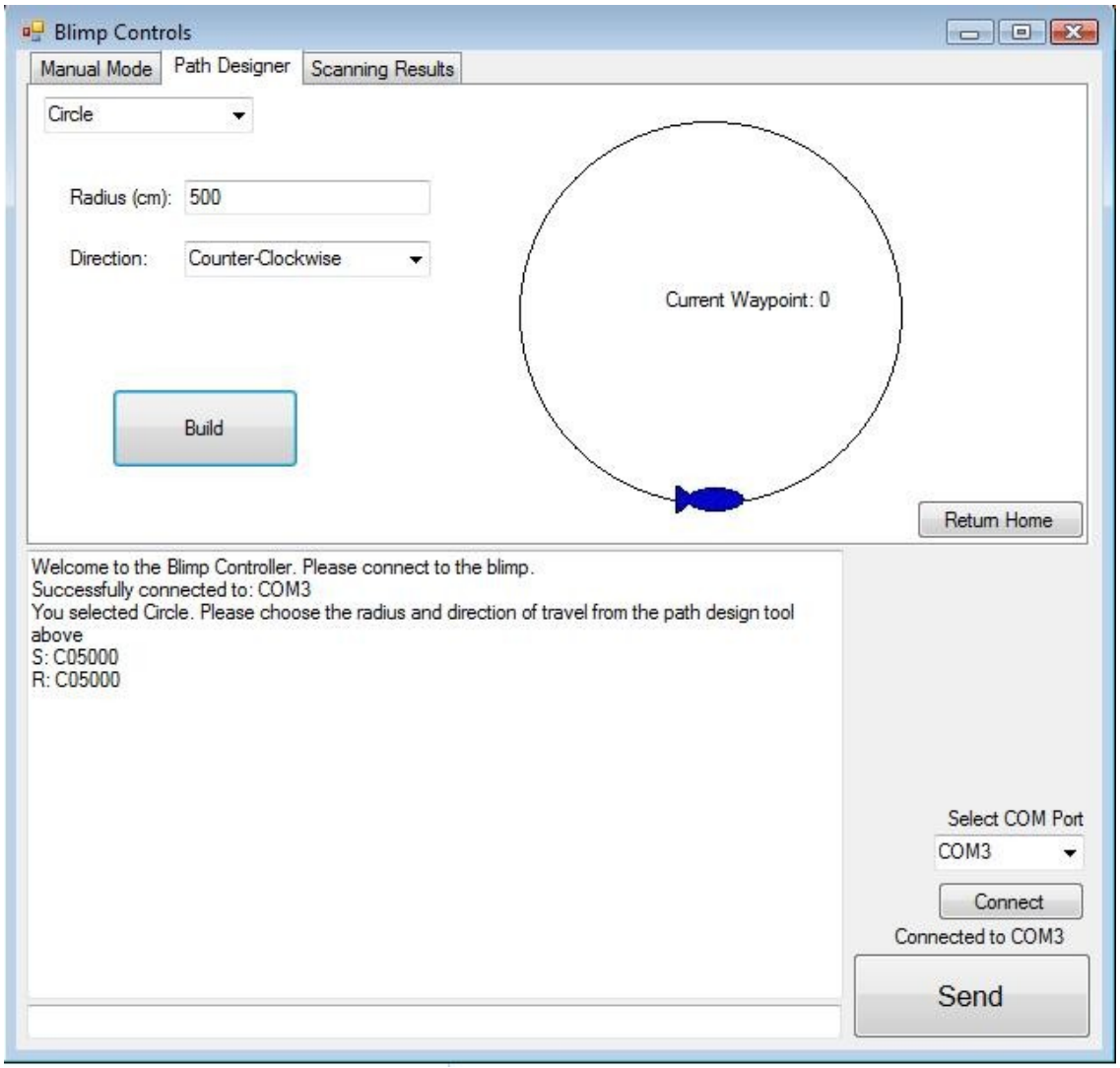

Figure C-7: Host computer control GUI – Path Designer tab