# Multiple Autonomous Surface Vehicle Project

## Robotic Kayak

Worcester Polytechnic Institute

Clark Bakstran

Scott Brooks

Angelo Platanias

Greg Sletterink

Nick Solarz

Thomas Womersley

Submitted to Professor Michael Gennert

April 26, 2012

## ACKNOWLEDGMENTS

We would like to thank Professor Christopher Kitts and Thomas Adamek from Santa Clara University for inviting us to participate in this project. We would specifically like to thank Thomas Adamek for his continued support and guidance over the course of this project. We would also like to thank Doug Renfro for his engineering contributions.

We are thankful for the constant support from Tracey Coetzee, who assisted with various project logistics, such as scheduling and ordering parts. We would like to thank Joe St. Germain for offering guidance during the PCB design process. We would also like to thank Adam Sears for his assistance during the manufacturing process.

Most importantly, we would like to express our gratitude and appreciation to Professor Michael Gennert for the guidance, navigation, and control that he provided for this project. Without his support, this project would not have been possible.

## ABSTRACT

The goal of this project was to design and manufacture an Autonomous Surface Vehicle (ASV) in collaboration with Santa Clara University (SCU) that would integrate with their existing ASV fleet. The team was responsible for designing a universal chassis that enabled it to attach to several different kayak hulls, and upgrading the electronics to a modern microcontroller architecture. The vessel is propelled by two motors and controlled by a microcontroller located onboard. The microcontroller receives commands from an off board base station, which facilitates communication between other ASVs in order to perform complex group maneuvers. The chassis and the electronics were successfully designed and implemented on the kayak, demonstrating the overall integration of the teams design to the existing fleet.

# EXECUTIVE SUMMARY

## Introduction

Autonomous surface vehicles that utilize swarm behavior are a promising yet underdeveloped field in robotics. Although the study of ASVs and the implementation of swarm robotics are both independently interesting research topics, it is the combination of the two that makes this project so promising. The idea was brought to WPI by Santa Clara University (SCU), whose Robotics Systems Laboratory was looking for our team to modify and improve on their previous designs.

Equipping unmanned robotic vessels with the capacity for swarm behavior has numerous applications. Such vehicles would be useful in any situation on water that is dangerous or inaccessible to humans. For example, Santa Clara University has already successfully conducted missions using the kayaks to guard a central boat and to compile a gradient map of a lake. In the future these kayaks could be used for many other tasks including search and rescue missions, data collection, or waste cleanup.

Several models of robotic kayak have been built over the years at SCU, with different configurations and propulsion methods. For this project, SCU specified that they would like the design to be built on a sit-on-top kayak. Extensive research was done to locate an ideal kayak and eventually six different models were chosen for comparison. Additional research investigated the advantages and disadvantage of different motor configurations, resulting in a two motor design for added stability and improved control.

## Methodology

The first step in designing the chassis for the kayak was to brainstorm design concepts, which resulted in the selection of a chassis based on feasibility and the specified requirements.

Having selected a design, the team created a detailed CAD model in SolidWorks and created a foam mockup of the chassis. Once the model was modified appropriately, the team began the manufacturing and assembly process, utilizing CNC mills and hand manufacturing.

The electronics team designed a custom PCB shield for the Arduino microcontroller. The purpose of the shield was to provide a "plug and play" method for attaching all the required electronic peripherals. This allows for easy setup and assembly of the electronics package. The electronics team was also responsible for the design and assembly of the wiring harness for the robot. This harness provides all the interconnections between the electronic components, such as power and I/O.

The main task the software team was to adapt the existing code from the outdated electronics. This consisted of two main tasks: writing low-level drivers for the various peripherals such as the GPS and motor controller, as well as establishing a communications protocol to transmit the telemetry data to the base station. Development of this protocol was pivotal to the software design, as it needed to be able to integrate with SCU's existing system. This required a high level of collaboration with SCU in order to establish a protocol that would meet their needs.

## Conclusions

This MQP set out to design and manufacture an Autonomous Surface Vehicle capable of integrating with Santa Clara University's existing and expanding fleet. The biggest original core requirements mandated that the final product be a kayak that is able to communicate with the existing fleet or base station by way of radio, be fully operational un-manned, and be capable of collecting and sending GPS and compass data back to the base station. After conducting research into possible additional features that would improve the overall kayak design, the WPI

team elected to also design a universal chassis capable of mounting a variety of kayaks, to upgrade the on-board microcontroller to an Arduino Mega from the previous two BasicX boards, and equip the robot with both a Ricochet and XBEE radio to allow multiple methods of communication. By the end of the MQP, all of these features had been successfully implemented and included in the final product.

# TABLE OF CONTENTS

## LIST OF TABLES



## LIST OF FIGURES

## AUTHORSHIP PAGE

Clark Bakstran: Executive Summary, Microcontroller Background

Scott Brooks: Mechanical Background, Formatting and Editing, Design

Angelo Platanias: Introduction, Requirements, Corrosion

Greg Sletterink: SCU Electronics Background, Schedule

Nick Solarz: Software Background, SCU Software Background

Thomas Womersley: SCU Mechanical Background, Budget

# CHAPTER 1: INTRODUCTION

Autonomous surface vehicles that utilize swarm behavior are a promising yet underdeveloped field in robotics. Though the construction of autonomous aquatic vehicles and the implementation of swarm behavior in clusters of robots are both independently interesting research topics, it is the combination of the two that makes this project so promising. The idea was brought to WPI by Santa Clara University (SCU), who has been working on this project for the past several years and is looking for Worcester Polytechnic to build an improved model.

Equipping unmanned robotic kayaks with the capacity for swarm behavior has numerous possible applications. Such vehicles would be useful in any situation on water that is dangerous or inaccessible to humans. Furthermore, due to the recent BP oil spill, the ability to execute complex tasks at sea using small boats armed with sensors is a rewarding capability[8]. Santa Clara University successfully conducted missions using the kayaks to guard a central boat and to compile a gradient map of a lake, but in the future these kayaks could be used for fishing or fish farming, military surveillance operations, cleaning chemical spills, collecting data for oceanographic or marine biology research, taking water samples, and countless other possibilities.

The potential technological and societal impacts of swarm robotics are universal and far-reaching. There are various applications in military and private sectors in the areas of research, reconnaissance, and search and rescue. There are great opportunities in developing such systems to help both the robotics industry and society by integrating robots in dangerous or time consuming operations at sea.

Working together with the Santa Clara University Robotic Systems Laboratory (RSL) we are given a chance to look further into to multi-robot ASV systems. By adding one of our own

kayaks into their fleet we are looking at improving several of their processes and tasks by exploring new options and ways to modify their existing system in an effort to make it more effective, efficient, and open for future expansion.

# CHAPTER 2: BACKGROUND

This section reviews the work already done by Santa Clara University related to this project. Topics covered are hull design, motor configuration, electronics, and software. This section also reviews the background research conducted by our team.

## 2.1 SCU Mechanical Design

### 2.1.1 Hull Design

SCU employed a simple yet effective design for the robotic vessel that uses an "off-the-shelf" hull. SCU chose to use kayaks for this project, because they are cheap, readily available, and offer a stable platform for mounting electronics and other equipment. Two different kayak models were used: the Dragonfly Moorea and the Pelican Apex 100. Both kayaks are sit-on-top style hulls and are readily available for purchase at Wal-Mart and other retailers. The Pelican Apex 100 was chosen as an upgrade to the Moorea because it contained drainage holes that were suitable for use as chassis fastening points. The Dragonfly Moorea proved to be the superior kayak, because the Pelican Apex 100 had widespread problems with leaking.

### 2.1.2 Motors

For propulsion, two Minn Kota 30 trolling motors were mounted to the kayak using floating outriggers placed on each side of the kayak. In addition to providing mounting points for the motors, the outriggers also provided the boat with additional floatation and stability. In an earlier design, the batteries were placed inside the outriggers for a lower center of gravity, but this idea was scrapped in successive design iterations. In a later design used to explore upgrading the kayak fleet, the team used Minn Kota 50 motors mounted to plates on the chassis, eliminating the floating outriggers.

### 2.1.3 Electronics Housing

Due to the aqueous operational environment, it is very important that the electronics on the vessel stay dry. SCU chose to use a waterproof Pelican case to house all of the electronics. This is an ideal solution because Pelican cases are waterproof, durable, and readily available for purchase.

## 2.2 SCU Electronics

### 2.2.1 Electronics Stack

The electronics stack on SCU's robotic kayaks contains 2 BasicX microcontroller boards. One board accepts drive commands and controls the motor driver to steer the boat. The other controller transmits position data and interfaces with the communication stack. SCU used a Metricom Ricochet modem (128Kbits/s), which can handle long range communication of up to 2 miles, multiple users, and has frequency hopping for security purposes. A digital Devantech CMPS03 compass was used for heading data, and a Garmin 18 differential GPS unit for position data.

SCU's newest kayak consisted of a Toshiba n450 netbook running Ubuntu 10.10, and using Robot Operating System (ROS). The single netbook performed the tasks previously covered by two BasicX microcontrollers. For wireless communication, XBee Pros were used to replace the old Ricochet modem. The new modems have a much faster transfer rate (250 kbps), but only had a theoretical range of 1 mile, and during testing turned out to be even less than that (1/4 mile). The digital Devantech CMPS03 compass and the Garmin 18 differential GPS both remained in this model.

### 2.2.2 Motors & Batteries

As seen previously, two Minn Kota 30 thrusters were used for propulsion. To control the motors, SCU used a Roboteq AX1500 motor controller. The motor controller interfaced with the BasicX stamps using an RS 232 connection. It was able to drive the motors at 36 volts and 20 amps continuously on both channels. Two deep cycle 12 volt 84 amp-hour batteries provided about eight hours of run time at speeds of up to 2 knots.

On the newest model, Minn Kota 50 thrusters replaced the Minn Kota 30 thrusters. The motor controller was also upgraded to the Roboteq AX3500. Instead of using two marine batteries, SCU decided to go with one 12 volt, 104 amp-hour deep cycle battery to power both motors. At maximum thrust, each motor drew 25 amps of current, giving an operating time of 6 hours at maximum thrust.

### 2.2.3 Base Station

Two Metricom Ricochet modems facilitate radio communications from the base station to the kayaks. SCU used a workstation running Matlab, DataTurbine, Simulink, and a VRML simulator to retrieve, process, display and redistribute sensor data, system information and robot commands. In the upgraded kayak model using ROS, a computer running Windows XP was used to emulate the base station during testing. The software stayed the same, however the base station sent out root commands to the ROS-equipped kayak.

Santa Clara University has been continuing work on cluster space multi-robot systems for a number of years, specifically with their aquatic autonomous surface vessel fleet. The idea behind a cluster space control system is to provide a simple way to abstract complex multi-robot movement using basic three DOF kinematic transforms [7]. This allows for a control loop that resembles that of a single robotic manipulator, with each robot in the cluster acting as a part in a

larger system. This simplification enables the robot cluster to use generalized kinematic transforms to calculate the desired positions and velocities for each robot in the cluster.

## 2.3 SCU Software Applications

Santa Clara University is currently researching a number of applications for their autonomous surface fleet. Currently, they have successfully implemented two different applications; autonomous guarding of marine assets, and gradient mapping and navigation. Autonomous guarding of marine assets describes a control layer that in a base state has the fleet in a patrol pattern around a defensive position. As a potential threat approaches this position, the fleet will form a "shield" between the defensive point and the threat. Gradient mapping describes a method to use multiple robots to map out a surface gradient (i.e. an oil spill). This system uses sensors on the robot to measure samples on both sides of the hull. This data is used to straddle the gradient at a certain level, while paroling that level of the gradient. As an example, this application could visually show the distribution of an oil spill. By having the cluster patrol the gradient at a certain level (say 50% of maximum) one can very simply map out the size of the affected area.

## 2.4 Background Research

### 2.4.1 Hull Design

The entire base of the robotic kayak is the kayak itself, so it was important for the team to research and explore different types of kayak hulls based on cost, transportability, waterproofness, and ease of mounting hardware. Due to price restrictions and the operational environment for these kayaks, the team narrowed down hull selection to recreational kayaks. These kayaks are primarily made for flat-water conditions, and are the cheapest and most readily

available. Within the spectrum of recreational kayaks, the team began narrowing down hull types into two categories: sit-in hull (open top) and sit-on-top hull (closed top).

Sit-in style kayaks are a common choice for recreational kayakers, due to their stability and the fact that the rider's lower body is enclosed inside the hull. For our team's purposes, one of the major advantages of a sit-in, open-top style kayak is the large usable space inside the hull. Components such as batteries and electronics could be placed low in the hull, allowing for a low center of gravity and improved stability. The seat opening in the hull would also allow some limited access to the components inside for service and modifications. However, the large opening could cause more difficulty in waterproofing the entire vessel, and add complication to the overall design. Upon researching kayak hulls, the team also found that, in general, sit-in style kayaks are more expensive than their sit-on-top counterparts.

Sit-on-top kayaks are inherently the most casual form of kayak, and are relatively cheap and easy to find. Sit-on-top kayaks may lack accessible space inside the hull, but components can be mounted directly on top of the hull for easy access. Since the hull is a solid piece, sit-on-top kayaks are theoretically more waterproof than sit-in types, provided that the construction quality is sufficient. In addition, sit-on-top kayaks are generally cheaper than sit-in types, which would reduce the overall cost of each vessel. One of the disadvantages of a sit-on-top kayak is the difficulty in mounting hardware. The top surface is usually contoured to fit a sitting person, with curved surfaces for a person's body. Components mounted on the top of the kayak must either contour to fit those features or bypass them entirely.

The team researched various models of kayaks to serve as the base for the robot. The kayak needs to cost less than $400, be at least 2 meters (6.56 feet) long, and have a payload of at

least 230 lbs. The kayak should preferably have a flat bottom for ease of maneuverability. We researched various kayak models online, and compared their specifications. See Table 1 below:

| Kayak Model | Length | Payload | Cost |
|---|---|---|---|
| Ocean Kayak Yak | 8' | 240lbs | $330 |
| Dragonfly Moorea | 8' | 250lbs | $230 |
| Lifetime Calypso | 8' | 250lbs | $215 |
| Pelican Apex 100 | 8' | 250lbs | $289 |
| Malibu Mini-X | 9' 3" | 300lbs | $499 |
| Coleman Drifter | 8' | 250lbs | $199 |
| Horizon 80xe | 8' | 175lbs | $279 |

**Table 1:** Kayak Model Comparison

### 2.4.2 Motor Configuration

Another important aspect of the team's kayak design is the motor configuration. This includes the number, placement, and configuration of the motors. SCU specified that this project would be using, at a minimum, the Minn Kota 30 trolling motors. Each of these motors has 30lbs of thrust. If deemed necessary, the team is permitted to use the 35 lb models for additional power. The motor configuration directly determines the speed, stability, and steering capabilities of the kayak.

The first aspect the team considered was the number of motors. Within practical limitations, the choice was either one or two motors. A single motor configuration would require mounting the motor along the centerline of the kayak. Because there is a restriction on drilling holes in the hull, this would require mounting the single motor directly off one end of the kayak. A single motor would reduce the weight of the whole kayak, but would require the motor itself to pivot for steering.

The other option is to use two motors. In this configuration, the motors would be placed away from the hull on outriggers, adding to the stability of the craft and allowing the kayak to steer using tank steering. Although cost and weight increase with a two-motor configuration, the

benefits of tank steering allow the kayak to steer using motor speed differential, rather than pivoting the motors themselves or using a rudder. This configuration also allows the vessel to pivot on a point, rather than have a large turning radius associated with a rudder or a single pivoting motor.

### 2.4.3 Microcontroller

During operation, the kayak is expected to carry out a variety of complex tasks. An on-board computer is needed to coordinate the various motor, sensor, and receiver activities necessary to the execution of such tasks. As the more computation-heavy calculations are handled by an off-board computer capable of running Matlab, the local computer need not run a fully-functional OS such as Linux or Windows. It is possible instead to use a simple microcontroller (MCU) to handle all onboard activities. Using a microcontroller brings a number of benefits. Microcontrollers are both cheaper and simpler to program than computers. Additionally, because MCUs generally do not employ full-scale operating systems, they often are able to execute code in real time. As previously mentioned, without an extra software layer to work with, it is common to be able to execute one operation per clock cycle. By equipping the kayak with a microcontroller instead of a CPU, it is possible to have quicker, simpler, and more efficient code for a cheaper price.

In years past, SCU chose the BasicX microcontroller board to handle all computations on their kayak. The BasicX uses a custom language made by its parent company, NetMedia, derived from the programming language BASIC, which is designed to be easy to use and intuitive. For $49.95, the BasicX board can execute 83000 instructions per second and has 400 bytes of RAM. Additionally, SPI and I2C ports are included, but there is no USB port. Also included are 21 digital I/O ports and eight 10-bit resolution analogue ports. SCU used BasicX boards due to the

team's familiarity with the microcontroller and not because it stood at as being particularly suited for its purpose. It therefore makes sense to compare the BasicX to other microcontrollers before one is chosen to be used in the WPI kayak.

The second microcontroller taken into consideration was the Arduino UNO. Made by the Italian company "Arduino Software," the Arduino UNO was designed to easily coordinate the use of multiple electronics and runs the company's own open-source Arduino software. The software is similar to C/C++ and is well known by the members of this project group. The micro controller runs at a 16 MHz clock speed, and contains 1KB of ROM and 2KB of RAM. In order to interact with external peripherals, the Arduino UNO contains 14 digital I/O ports and six 10-bit ADC analog input ports. Additionally, the UNO comes with an I2C, SPI Bus, and USB. For the price of $29.95, the Arduino UNO deserves careful consideration as the microcontroller on our team's robotic kayak.

Another worthwhile controller by the same company is the Arduino Mega. The Arduino Mega runs the same software as the UNO, but the hardware contains a number of significant changes. The ROM and RAM are upgraded to 4KB and 8KB respectively. With 54 I/O ports and 16 analog input ports, the Mega has far more capacity to interact with peripherals then the UNO. The clock speed remains 16MHz and the hardware still includes one SPI, I2C and USB port but the other improvements increase the price to $58.95. The biggest question regarding the Arduino Mega is the true number of I/O ports our project needs.

Similar in price to the Arduino Mega, the Mbed microcontroller by ARM sacrifices the Mega's extensive I/O options for increased computing capabilities. With a 96MHz clock speed, 512KB of ROM and 64KB of RAM, the Mbed microcontroller is by far the most powerful MCU mentioned so far. It also runs C++, a very useful language familiar to most of the kayak's

programing team. For the price of $59.95, the Mbed comes with 25 I/O ports and six analog input ports equipped with 12-bit resolution. The Mbed compiler has the added bonus of running online, and is similar to the popular Eclipse program but slightly less robust. The Mbed is also the only microcontroller we looked at that has an Ethernet port, although the usefulness of that accessory is questionable.

The Netduino microcontroller by Secret Labs has earned popularity among engineers and hobbyists alike for being affordable without sacrificing processing power. The Netduino, though not as powerful as the Mbed, surpasses the Arduino and Basic X boards with 128KB of ROM and 60KB of RAM backed by a 48 MHz processor. The Netduino runs C# which can compile in Visual Studio which is a well-known and easy to use compiler. The Netduino comes with 20 I/O ports, six analog input ports and is available for $34.95.

Since the eventual long-term goal of the kayak project is to have the kayaks themselves run the computations, it makes sense to look at a computer capable of running complex programs such as ROS and Matlab. The Gumstix COM mounted to a RoboVero microcontroller board would be capable of fulfilling such a purpose. For the price of $250.00, the Gumstix is a fully functional computer running at 600MHz with 512MB of both ROM and RAM, 70 I/O ports and six analog input ports.

See the Tables 2 and 3 below for comparison of microcontrollers.

| | Basic X | Mbed | Netduino | Gumstix |
|---|---|---|---|---|
| Cost Per Unit | $49.95 | $59.95 | $34.95 | $250.00 |
| Compiler | Basic X Compiler | Mbed Compiler | Visual studio | Anything |
| Simplicity/Comfort | Simple, but unfamiliar with basic | Coded in C++ | C# | Very complicated |
| Clock Speed | 83000 instructions/s | 96 MHz | 48 MHz | 600 MHz |
| ROM | 32KB | 512KB | 128KB | 512MB |
| RAM | 400B | 64KB | 60KB | 512MB |
| Digital I/O | 21 | 25 | 20 | 70 |
| Analog In | 8 | 6 | 6 | 6 |
| ADC Resolution | 10-bit | 12-bit | 10-bit | ??? |
| SPI | yes | yes | yes | yes |
| I2C | yes | yes | yes | yes |
| USB | no | yes | yes | yes |
| Ethernet | no | yes | no | no |

**Table 2:** Comparison of Microcontroller Models

Final Value analysis

| | Cost | Comfort with Language | IDE Features | MCU Capabilities | Libraries | Total |
|---|---|---|---|---|---|---|
| Weighting | 1 | 5 | 4 | 2 | 3 | Out of 75 |
| BasicX | 4 | 2 | 2 | 1 | 3 | 33 |
| Arduino UNO | 5 | 4 | 2 | 2 | 5 | 52 |
| Arduino Mega | 2 | 4 | 2 | 3 | 5 | 51 |
| mbed | 2 | 4 | 4 | 4 | 3 | 55 |
| netduino | 4 | 3 | 5 | 3 | 4 | 57 |
| gumstix | 0 | 2 | 5 | 5 | 3 | 49 |

**Table 3:** Value Analysis of Microcontrollers

## 2.5 Software Background

### 2.5.1 Robotic Operating System

One of the software options the team researched was the Robotic Operating System (ROS). ROS is a "meta-operating system" that currently runs on top of many Unix-based operating systems such as Apple's OS X and many Linux distributions such as Ubuntu and Fedora. ROS is open-source, but is currently being maintained by Willow Garage, which uses the ROS framework with their robot, the PR2.

#### ROS Overview

The ROS framework provides a neat and simple way to program complex tasks across various components in a robotic system. The simplicity lies in how the program hierarchy works. In a normal program, there is one main function that in turn calls other sub-functions, essentially forming a tree hierarchy that branches as it moves down from the main function. While this solution works, it becomes difficult to debug or add new features. This is where using ROS has real advantages. Instead of the typical top-down hierarchy as previously mentioned, ROS uses a mesh network hierarchy. Instead of one main function calling sub-functions, each major function acts as a Node on the mesh. Each Node can run independently of each other Node, as the communication is independent of function calls.

#### Node Communication

ROS uses a "publisher/subscriber" format to communicate between Nodes. In order to communicate between Nodes, there are Topics. The easiest way to visualize this is to consider a Topic to act like a blog. On a blog, there are people publishing content to it, and people reading the content and acting on it. ROS communication acts in the same way. Each Node can act as a publisher and/or a subscriber to a Topic, or it could not interact with a Topic at all. Each Topic

can have any number of subscribers and publishers. Every time a Node publishes new data to the Topic, all of the subscribers perform their action using the message as an input. This allows simple debugging and new code implementation, as Nodes can be added or removed from the system independently of the rest of the code. For example, in order to integrate a new Node into a system, one needs to only make sure that it uses existing Topics to communicate. In terms of debugging, individual Nodes can be shut down, debugged, and restarted without shutting down other parts of the system.

#### Disadvantages

While ROS provides a lot of advantages for programming complex systems, it does have some disadvantages compared to low-level code. The biggest of these disadvantages is speed, especially compared to a strictly embedded solution. Any program system using ROS needs a software layer to interface with hardware. This added layer increases the time to process commands. An embedded system that runs native code has the potential to run faster due to single clock-cycle execution. However, this disadvantage is negated if ROS is running on a sufficiently powerful system, capable of operating at greater clock speeds to compensate for the additional software layer.

### 2.5.2 Arduino

The Arduino microcontroller uses what is essentially C++, however it does not follow many of the normal conventions that actual C++ uses. The reason behind this change in convention has to do with who the Arduino is marketed to. The Arduino is designed to be easy to use by hobbyists that have minimal programming experience, allowing the user to quickly implement code to fit their project's needs without worrying about some of the more complex programming problems. The biggest example of this is the number of libraries that are included

with the Arduino. Many library APIs like I2C, SPI, and many others are already included, whereas if the user were to develop their project with another MCU, they may have to develop those libraries themselves. This becomes a matter of function over form, as it is simple to write "quick and dirty" code that runs without delving into the inner workings of the Arduino framework.

Arduino's simplicity does not end in its code styling. The Arduino IDE is very minimalist, allowing users to get straight into coding without spending much time worrying about setting up the environment. This simplicity is not without its disadvantages. While the Arduino IDE allows for fast coding and setup, it lacks many of the extensive features that other IDEs have, such as Eclipse and Visual Studio. Things like a robust debugging environment built in revision control are just a few examples of features that the Arduino IDE is missing [2].

# CHAPTER 3: METHODOLOGY

## 3.1 Requirements

During our background research, we came across many restrictions and limitations related to this project. This is a result of the nature of the project; since we are collaborating with SCU (Santa Clara University) on creating robotic kayaks that will communicate with each other, we have to follow several specification set by them, as well as some ground rules, so that all of the robots will be compatible with each other.

### 3.1.1 Mechanical Requirements

1) No holes in the kayak
2) Achieve a cruising speed of 2 knots
3) Use of off-the-shelf parts: SCU's lab policy for such projects states that part that are used to create robotic projects must use off-the-shelve products in order to improve replicability, and ease of construction.
4) Universal chassis that fits multiple kayak hulls.
5) Adaptation: it should be based on a versatile platform, where parts can be added and modified at any time.
6) Should be able to withstand use in both estuaries and ocean. Therefore resist water corrosion etc.
7) Payload of 40 kg.
8) Length: The overall length should run from a minimum of 2 meters to a maximum of 3.5
9) Zero turning radius. Be able to able to achieve an on point turn
10) Reversible thrust. Motors should be able to operate in both directions

### 3.1.2 Electronics Requirements

1) Battery life: The kayak is expected to have a battery life of 6 hours of use. During these 6 hours most of the time will be spent for data transmission rather than moving.
2) Use of an Arduino board.

3) Radio range should be greater than 1 km over water bodies. Due to the fact that water bodies tend to decrease the effective distance of radio moduli.
4) Data transfer rate for the radio should exceed 1Hz. Therefore send information packets back to the station at a rate of at least one packet per second.
5) Minimum number of sensor ports is 3. The least amount of ports available to connect sensors on the Arduino board should exceed 3.
6) GPS unit should update the position at a rate of at least 5Hz.
7) Compass should use an I2C bus protocol to connect to the board.

### 3.1.3 Software Requirements

1) Compatibility with existing code
2) Modular code design
3) Well-documented code structure
4) Thoroughly tested and functional code

### 3.1.4 General Requirements

1) Cost: Each kayak should cost less than $800. That includes all of the materials including kayak, motors and electronics. Construction hours are not calculated for the moment.
2) Fast set up time: The kayak should take less than 10 minutes to be set up and ready for deployment, by an experienced user.
3) Waterproofing for over turning. Even if the kayak overturns electrical components should stay secure and dry.

## 3.2 Mechanical Design

The following sections describe the various aspects of the mechanical design, including the design itself, the materials selection, and the testing processes.

### 3.2.1 Design Concepts

During the course of the project, the mechanical team went through many design iterations before arriving on a final chassis design. The initial designs began before we received

the physical kayak, and one involved a device that would clamp into the seat of the kayak using the footholds as a bracing point. This design was modeled after the idea of the human lower torso. A basic mockup of this design can be seen below in Figure 1.

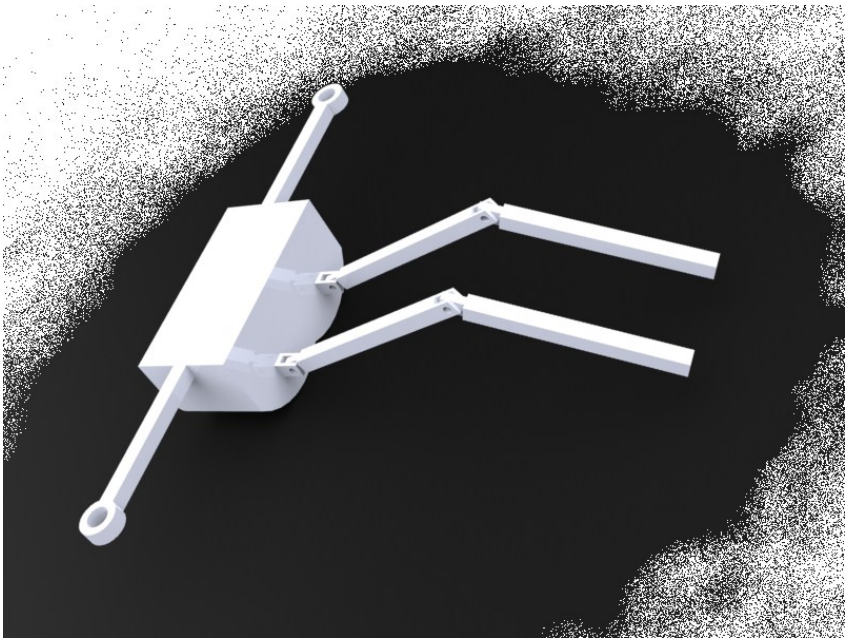

**Figure 1: Lower torso design**

The team also developed an alternate design that involved an adjustable sliding frame that clamped against the insides of the recess in the hull. This design would allow for adjustable width based on different models of kayaks. See Figure 2 below for a picture of this design.

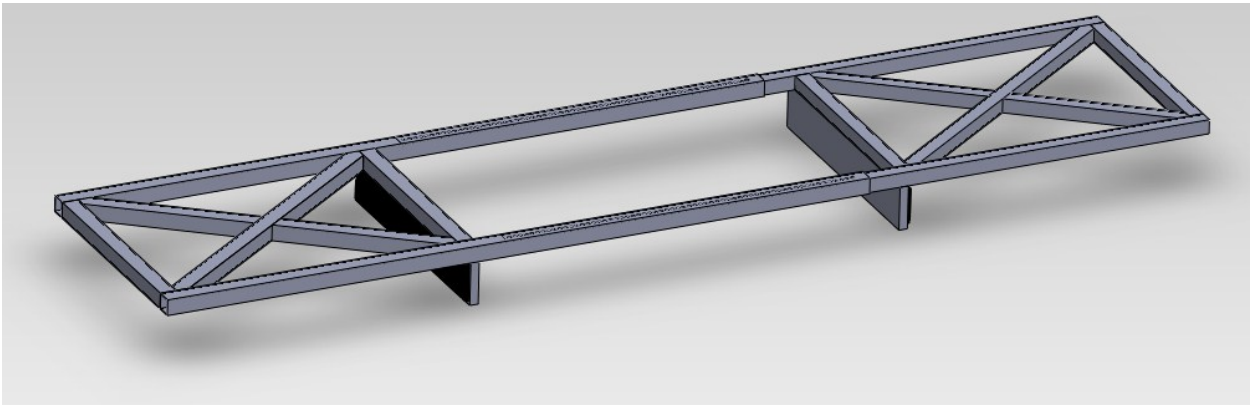

**Figure 2: Adjustable clamp design**

However, once the team received the actual kayak, some problems arose with our initial designs. The recess for the seat is too shallow, the sides are too slanted, and the footholds are too small. All of these aspects made it difficult for our clamping-type designs to attach firmly to the kayak. See Figure 3 below for a picture of the top of the kayak.

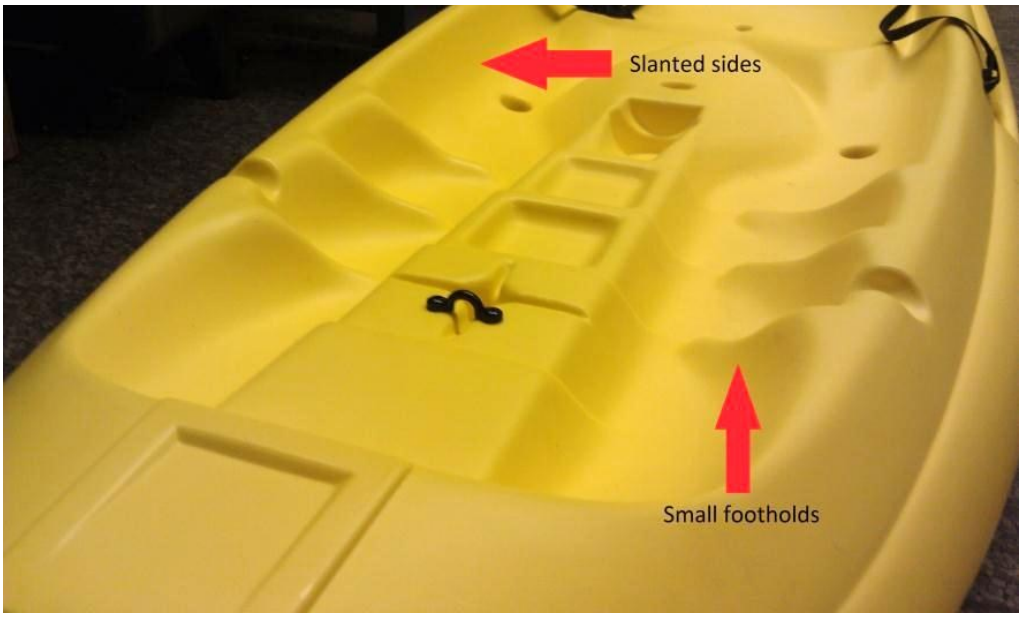


**Figure 3: Top of Kayak Hull**

These problems with the contours of the hull and the shallowness of the recess led us to explore the possibility of using straps to secure the chassis to the hull. We felt like this was a good way to strongly secure the frame to the kayak, but we were concerned about how the straps would wrap around the bottom of the hull. After examining the hull bottom, we discovered that it was not perfectly flat, and the outside edges protruded below the center of the hull. The center of the hull also has a small spine that is raised several millimeters above the bottom. Because of these shapes, any straps wrapped around the hull would not wrap flush to the hull and would flap in the water during forward motion. See the illustration below to see the shape of the kayak bottom.

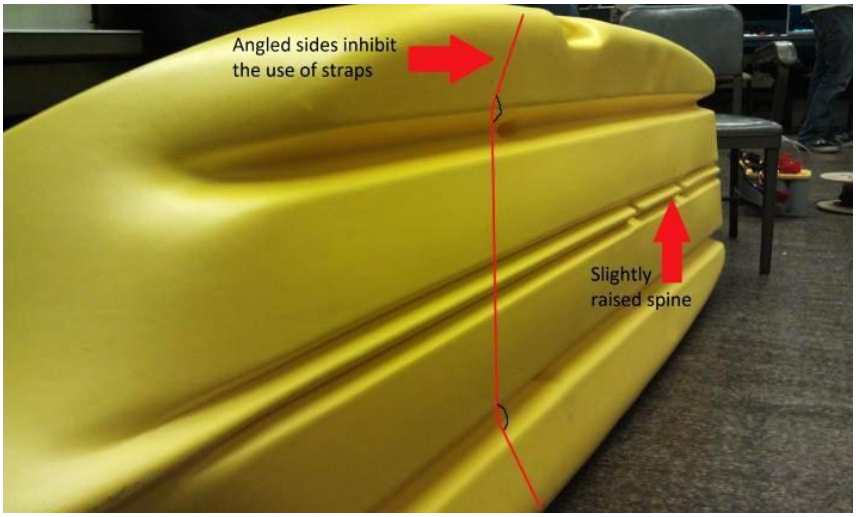


**Figure 4: Bottom of Kayak Hull**

After these unsuccessful design ideas, we decided to go back to the drawing board and design something that could rest on top of any kayak hull and be secured from the top, not by wrapping around the bottom. We began by designing a basic adjustable rectangular frame that would serve as the central part of the chassis. Other components such as the motors, batteries, Pelican case, and any sensors would either be contained in this rectangle or branch off of it somehow. We felt like this offered us the most in terms of flexibility and modularity of the chassis design.

After looking at our kayak hull and various other kayak hulls, we noticed that one common feature was the presence of side rails—that is, raised rails between the recessed passenger area and the side of the kayak itself. Some form of these rails can be found on almost every model of kayak, and would serve as the ideal resting point for the chassis. As an example of the side rails, see the picture below.

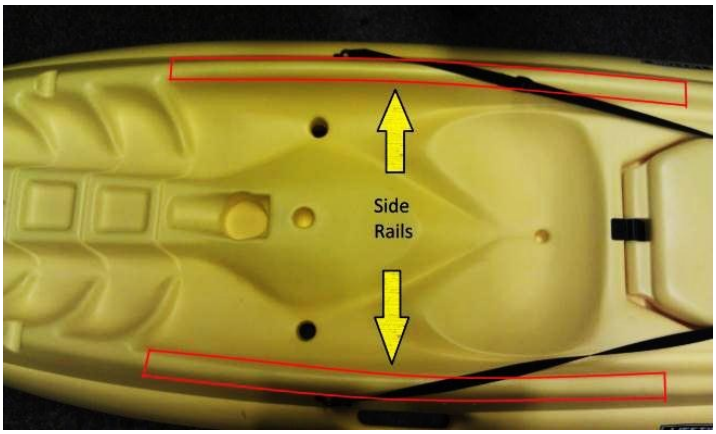


**Figure 5: Side rails on the kayak hull**

Therefore, the team decided to make a design that focused on putting all of its vertical weight on these rails. This way, the shape of the inside of the kayak is not relevant to the chassis itself. The shape of the recessed area varies greatly among different models of sit-on-top kayaks, so we feel that using the side rails for support is a more practical choice. The distance between the rails may vary between kayak models, but the central chassis will be adjustable to account for these differences. See Figure 6 below for a basic concept model of the central rectangular chassis. Note the adjustable triangular-shaped pads for resting on the side rails.

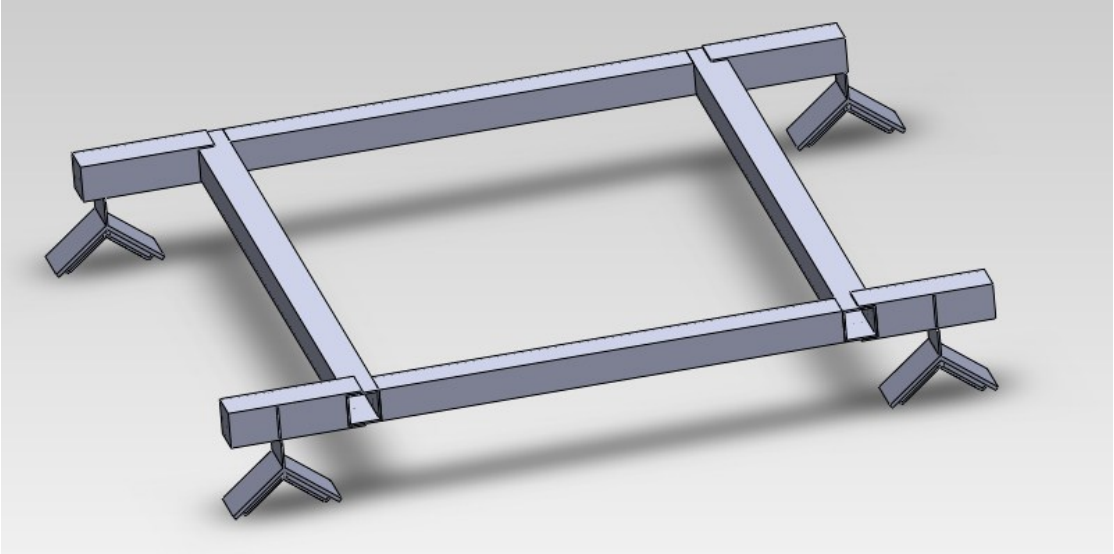

**Figure 6: Rectangular chassis design**

### 3.2.2 Chassis Mounting Methods

Once we were satisfied with the idea of resting the chassis on the side rails, we needed a method for tightly anchoring the chassis itself onto the hull. Our main idea for anchoring centers around the use of adhesives and D-ring hooks used on dinghies and other small boats. See Figure 7 below for an example of this type of anchor.

**Figure 7: D-Ring Hook**

This anchor consists of a 4.5-inch diameter PVC pad with a metal loop attached to the top. Up to four of these hooks will be glued in place inside the recess of the kayak, below the

chassis resting on the side rails. Threaded rods with hooks on the end will attach to these rings and tighten down to firmly secure the chassis onto the hull itself. The pads can be placed in different locations depending on the design of the kayak hull, and multiple mounting points are available on the chassis.

The team researched possible adhesives for sticking the pads to the kayak hull, and will begin testing adhesives for strength and waterproofness in the coming weeks. One of the greatest challenges is finding an adhesive that can bond to High Density Polyethylene (HDPE), the material of the kayak hull. HDPE is a particularly slippery plastic, and many common glues and adhesives do not adhere to it. Discussion of the glue selection can be found the Materials section.

### 3.2.3 Design Mockup

Once a basic conceptual design was established, more advanced design work and modeling began. We measured the exact dimensions of the kayak and created a CAD model in SolidWorks. Using the dimensions of this initial model, we created a foam and cardboard mockup of the design, which can be seen below in Figure 8.


**Figure 8: Foam and cardboard chassis mockup**

After constructing the mockup, we placed it on the kayak and looked for any design issues. We realized that the chassis would need to be extended to fit the Pelican case, and also that parts of the chassis could be made more compact for packaging reasons. Overall, however, the team was satisfied with the design of the mockup and how it would fit on the kayak. Further description of the mockup can be found in the Testing section of the report.

### 3.2.4 Refined CAD Model

After observing the mockup and noting any design changes, work began on creating the final CAD model in SolidWorks. The design went through much iteration but finally settled upon a final version. It follows the same original concept, but has concrete implementations of several features, especially the adjustability. The V-shaped pads can move in and out to adjust to different widths of kayaks, and the hooks can change location to account for different mounting points. The chassis has a location for the batteries and for the Pelican case. This model also contains all of the actual fasteners used in the final design, so it is a complete package. A render of the design is shown in Figure :


**Figure 9: Render of Chassis Model**

### 3.2.5 Materials Selection

Choosing the materials that would eventually form our chassis was a very important decision we had to take early during the design stage. Each material available had its pros and cons and affected the way we would eventually design our versatile platform. The options were many but some seemed more favorable than others, mainly because they would help us cover the initial requirements set forth by SCU. These requirements included mechanical specifications as well as budget restrictions that made the procedure more challenging than initially expected.

#### 3.2.5.1 Framing material

When choosing the material we would use for the chassis low weight, high strength and corrosion resistance were the core attributes we were looking. A material that would ensure such features was aluminum. Aluminum was able to provide us with sufficient amount of strength relative to its weight, this was important to since we wanted our structure to be portable, but at the same time be able to handle the large weights of the batteries and motors that would be mounted to it. As far as the corrosion due to salt water, we had nothing to worry about since the aluminum alloys are very corrosion resistive.

More specifically, we used 6063 aluminum for the most part of our structure, due to its good mechanical properties and it's smooth surface, it allowed use to build a sturdy chassis.

#### 3.2.5.2 Fastener Material

The fastener choice was another part of the chassis that need our attention. Our first reaction was to use aluminum fasteners, but after further research into previous projects we found out that they would not be the right choice. In the past, similar projects had used aluminum fasteners and reported many faults. This led us to exploring other options such as stainless steel

fasteners. Stainless steel fasteners provided us with the strength we were looking for, as well as the corrosion resistive properties demanded for marine applications. This time around weight would not be an issue since the amount of fasteners would not be large compared to the rest of the structure.

#### 3.2.5.3 Corrosion

Corrosion posed an interesting challenge to our project, since our chassis would be used for marine applications. This meant that it had to be designed in such way so as to withstand extensive exposure to salt water that could potentially cause oxidation. That could result in the corrosion of the surface, or even structural failure, it remained untreated.

A simple example where even the slightest amount of surface corrosion would matter is the sliding mechanism that allows the legs to adjust in width. In the case oxidation appeared between the two sliding parts, the motion would be limited and could result in the unification of the two parts.

Galvanic corrosion is the result of an "electrochemical process in which one metal corrodes preferentially to another when both metals are in electrical contact and immersed in an electrolyte"[1]. In our case there was a risk for galvanic corrosion between the aluminum frame and the stainless steel fasteners when exposed to salt water. The reason behind this was their placement of the galvanic series chart (Figure 10). The further apart two metals are the more electric potential is created between them, causing the corrosion.

[1] "Galvanic Corrosion." Wikipedia. Wikimedia Foundation, 23 Feb. 2012. Web. 02 Mar. 2012. <http://en.wikipedia.org/wiki/Galvanic_corrosion>.

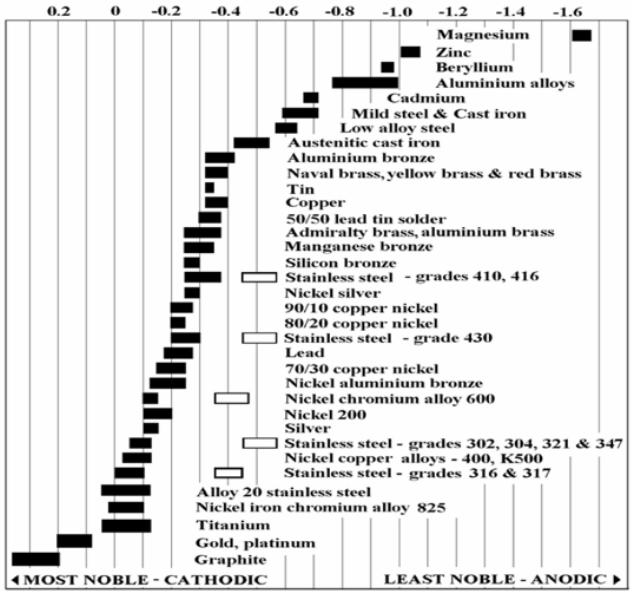


**Figure 10: Galvanic Series**

In order to avoid this phenomenon from occurring we had to replace our fasteners. We looked for a metal on the chart close to the aluminum alloys and preferably even less noble. Zinc seemed to fit our specifications, but we could not find any zinc fasteners that would suit our strength requirements. Therefore, we resulted to stainless steel zinc platted and galvanized fasteners that met both our corrosion and strength requirements.

Another type of corrosion is one caused from Iron Oxidation. This is what is commonly known as rust, which occurs as a reaction of iron and oxygen when exposed to water. To avoid such a reaction we chose to use stainless steel that contains chromium and forms a protective layer of Chromium(III) Oxide when exposed to oxygen and protects from oxidation.

#### 3.2.5.4 Adhesive

In order to attach the D-ring pads to the surface of the kayak we chose to use an adhesive. At first we had trouble finding the right adhesive since we had to bond HDPE (High Density

Polyurethane) with PVC. Eventually we came across an adhesive called TAP Polly-Weld[2], which was not only able to bond the two plastics together, but provided the necessary tensile and shear strength to hold the pad in place.

### 3.2.6 Construction and Fabrication

A few of the components of the chassis contained features such as a series of holes or a slot. These features would be very difficult to manufacture by hand to an acceptable accuracy. Therefore it was decide that a computer numerical control machining (CNC milling) would be the preferred method of manufacturing for these select components. In order to CNC mill these components a few steps had to be taken. The steps include modeling the components accurately, modeling the components and features in a computer aided manufacturing software (CAM), and then physically machining the components.

The entire chassis design was modeled using computer aided design software (CAD) SolidWorks. Each component of the chassis was modeled separately and then constrained together in an assembly file, which allows for geometric constraints to be applied to individual parts so that the structure of the chassis can be created. One of the parts that needed to be manufactured using the CNC machining process is the two lateral beams, which crosses laterally on the chassis and holds together the two longitudinal beams. The features on the lateral beam that requires CNC machining is a series of 22 through holes. This series of holes, illustrated below in Figure 11, allow for the tie down points in the chassis design to be adjustable.

[2] Datasheet can be found at: http://www.tapplastics.com/uploads/pdf/Tech%20Data-2011-%20Poly-Weld.pdf

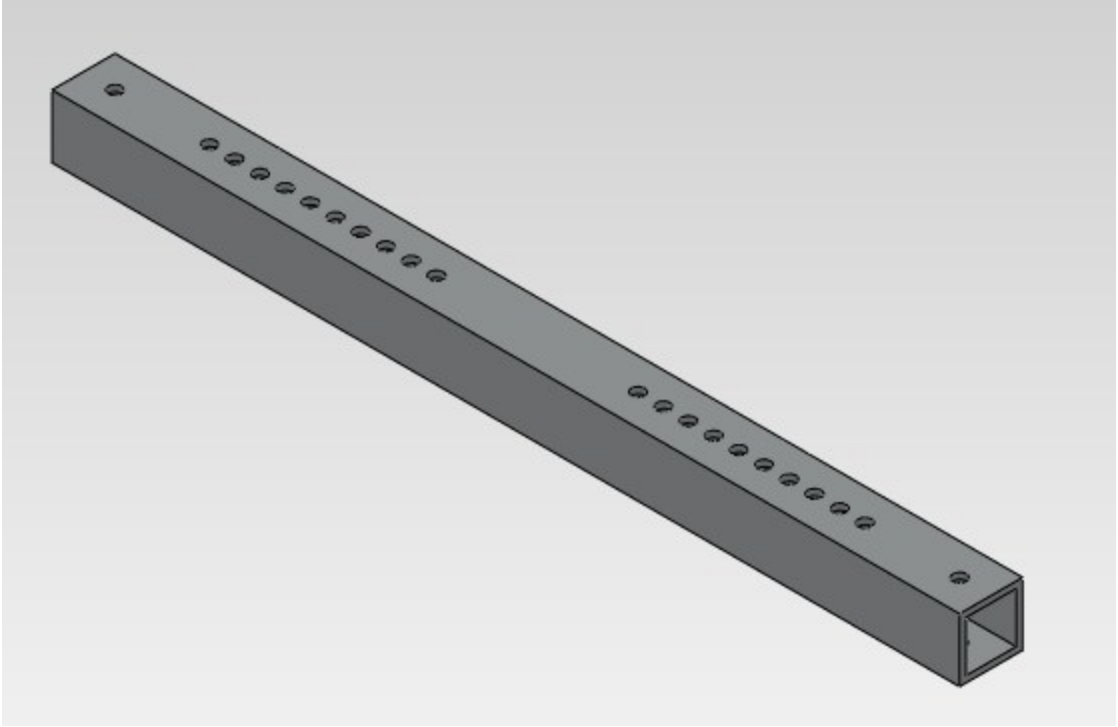

**Figure 11: Lateral Beam Isometric**

It would be very difficult to hand machine these holes and maintain an acceptable level of accuracy. Another reason this feature is better done using a CNC mill is that it significantly reduces the manufacturing time of the component. Another part that requires CNC machining is the adjusting beam. The features, illustrated in Figure 12 and Figure 13, include a lollipop shaped slot, two straight slots on either side of the component, and two through holes. The circular hole at the end of the slot allows for assembly of the sliding mechanism. Figure 14 shows how the sliding mechanism is assembled.

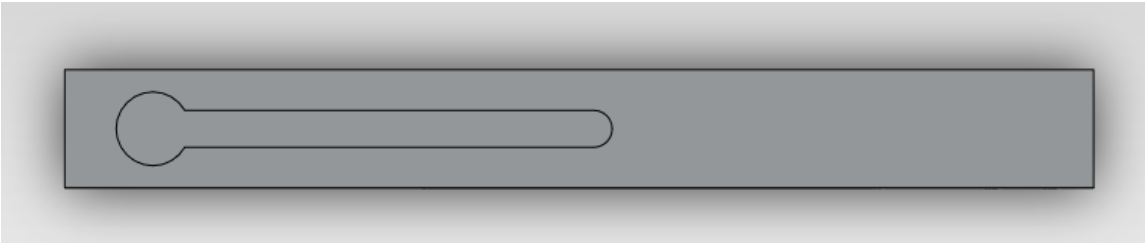

**Figure 12 : Adjusting Beam Top**

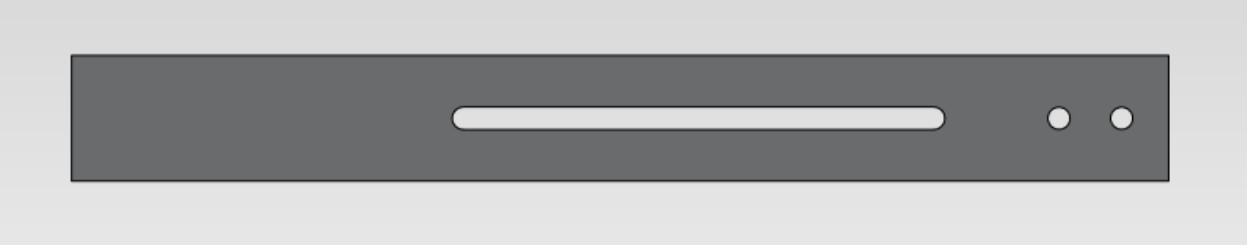

**Figure 13: Adjusting Beam Front**

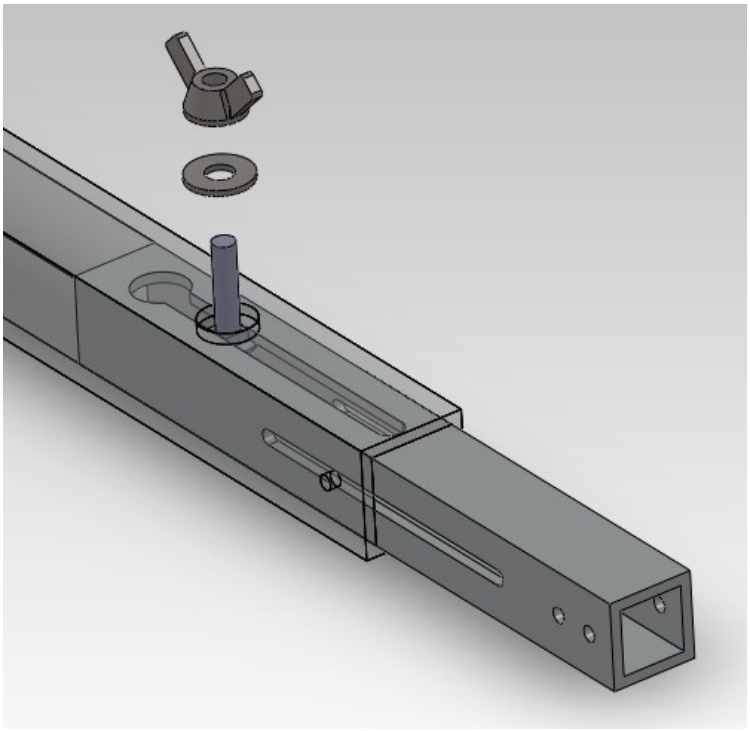

**Figure 14: Sliding mechanism assembly**

The final part that needed to be manufactured using the CNC machining process is the longitudinal beams. These beams have a series of holes similar to the lateral beams. For this reason they are more easily manufactured using the CNC machine. Figure 15 illustrates the placement of these holes on the longitudinal beams.

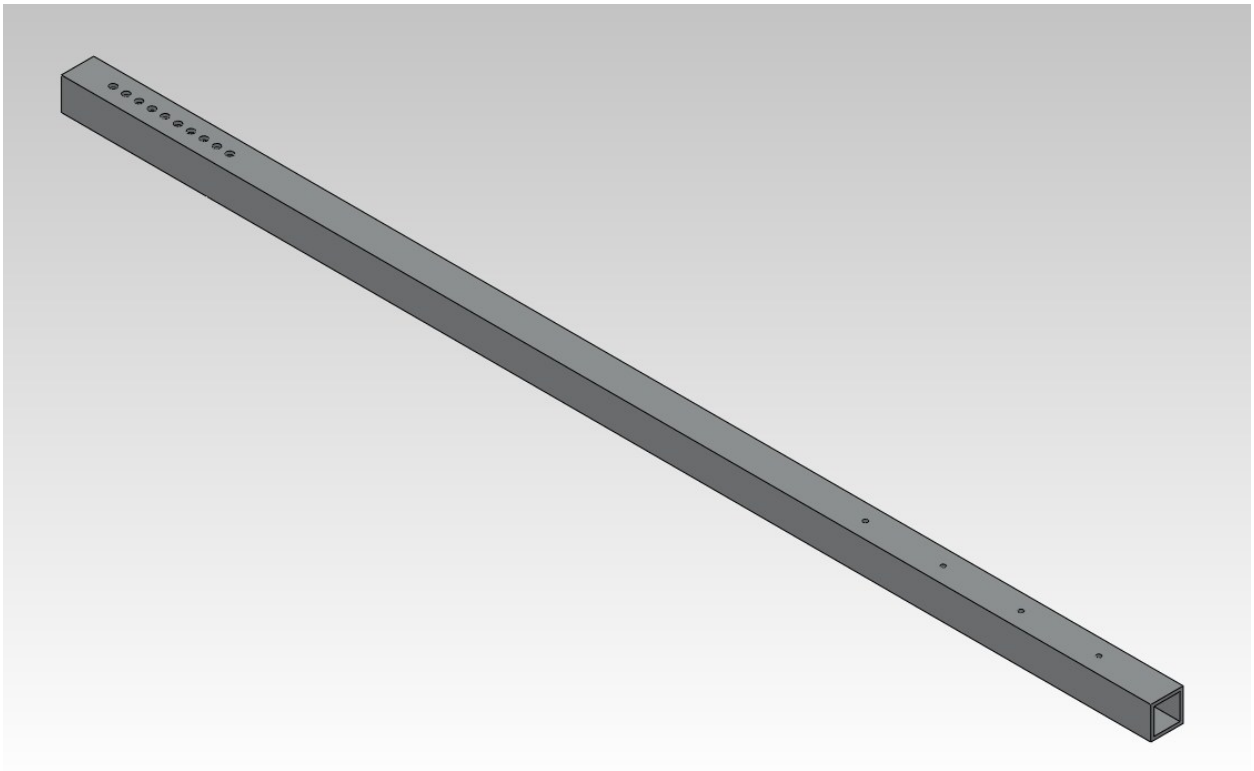

**Figure 15: Longitudinal Beam Top**

These features would be almost impossible to manufacture by hand because of the shape and placement of the slots. The through holes would not be that hard to manufacture manually, but to save time and to improve the accuracy of hole placement the two holes are done using the CNC milling machine.

The software used to CAM the different components is called Esprit. In this program the 3-D model of the component is imported from SolidWorks. Once imported the different features to be manufactured have to be inputted into the program. This is done using a series of selections tools in the software package. Esprit has geometric feature input tools for faces, pockets (slots), holes, and other standard features. The components could be modeled using more complicated methods to include all of the features in one file by employing what is known as “layers”. But it is easier to make one file for each face and corresponding features on that face. Therefore, the

adjusting beam had three separate Esprit files that together machine all the necessary features. There is one file for the lollipop slot, one for the slot on the front face and the through holes machined from the front face, and one file for the slot on the back face.

Once the features are inputted into the software the next step is to create the tooling that will be used to machine the different features. To machine all of these parts the following tools are required. A 1/4 inch end mill, a 1/8 inch end mill, a 0.177 inch drill, and a 0.261 inch drill. An end mill is a flat bottomed drill that is designed to cut away material by translating in the material. The end mills will be used to machine out the slots on the adjusting beam. The 1/4 inch end mill will be used to machine the lollipop slot and the 1/8 inch end mill will be used to machine the slots on the side of the component. The 0.177 inch drill will drill the through holes on the adjusting beam, and the 0.261 inch drill will be used to drill the through holes on the lateral beam.

After selecting the tooling for each of the features several other parameters for the machining and tooling have to be set. These settings include setting up tool offsets, setting various options for machining, and setting the method of machining for each tool. For each of the tools the various cutting strategies can be specified. In some instances when machining slots into parts a cutting strategy of moving the end mill the entire length of the slot in one pass cutting material as the end mill translates. The problem with this method of cutting is that the end mill is completely engaged in the material for the entire translation, which puts a lot of stress on the tooling and could potentially break the tooling. Therefore, a cutting strategy called trochoidal pocketing is used. The difference from this cutting strategy and the full slotting method previously explained is that the tool moves in a looping motion as it cuts. This motion allows a small interval of time when the tool is not engaged in the material, which drastically

reduces the forces that act on the tooling. An illustration of the tooling path is illustrated below in Figure 16.

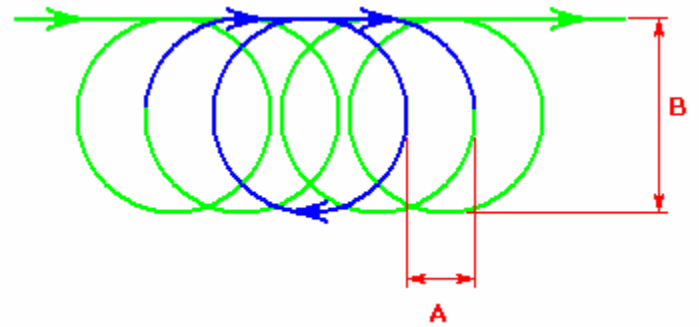


**Figure 16: Trochoidal Tool Path**

Therefore, the trochoidal cutting strategy is the best suited method to ensure that the tooling will not break when manufacturing the part.

Once all of these settings are chosen the manufacturing of the part can be simulated in the software. The next step is to use the software to export NC code, which is the code language that the milling machines can interpret. Once the code is loaded onto the machine the next step is to probe the part so that the mill will have the absolute positional data on where the part is located. The program is simulated once more on the milling machine and then initiated.

## 3.3 Electronics Design

### 3.3.1 Wiring Schematic

The first task for the electronics team was to develop a basic wiring diagram of the components on the kayak. Seen in Figure 17 below, this diagram shows each of the basic electronic components and how they are connected to each other. This served as a conceptual model for the overall layout of components. After creating the wiring diagram we started designing our schematic that we could prototype onto a breadboard. After thorough testing of the breadboard, we can finish designing a printed circuit board (PCB) that could be attached to the

Arduino mega board. As seen in Figure 17 the schematic contains all the components that will fit onto the PCB Arduino shield.

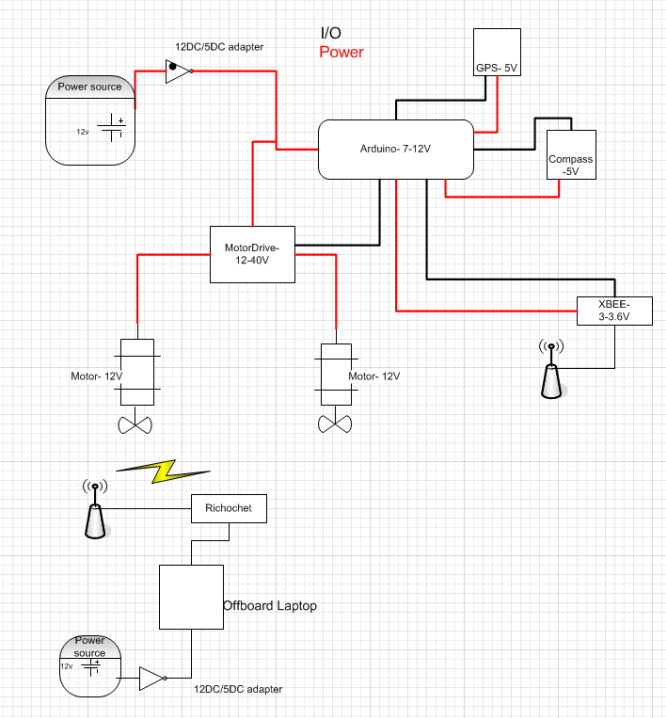


**Figure 17: Basic Wiring diagram**

In order to coordinate complex routs involving multiple kayaks it is essential to track the various kayak locations. The most obvious, comprehensive and simple method of achieving this is to equip each kayak with a GPS. Santa Clara University provided us with a Garmin GPS 18x to serve this very purpose. The Garmin GPS 18x offers users high precision positioning data (within 3 meters) that can be updated as fast as once every 20 milliseconds (5 Hz) in a waterproof package, making it an ideal choice for demanding marine applications. The GPS interfaces to a 6-pin serial port. While reading the Technical Specifications sheet provided Garmin we discovered that the serial port accepts RS-232 level inputs. At first, we took this to mean that an RS-232 serial adapter would be necessary to interface the Garmin with the Arduino Mega board, which contains no serial inputs. However, after testing the logic outputs we

discovered that a simple logic inverter was all that was necessary to serve that purpose. Of the GPS's 6 pins, only two carry data necessary for our project and the other four pins were divided between ground and a 5V rail which was already established to power various other ICs.

The other component used for positioning data of the kayak was the compass. This connection was very easy because all that was necessary was the connection of two data lines SDA, and SCL for I2C to the microcontroller, and its +5V and ground connections. The compass will work in concert with the GPs to give heading information while the GPS gives positional data.

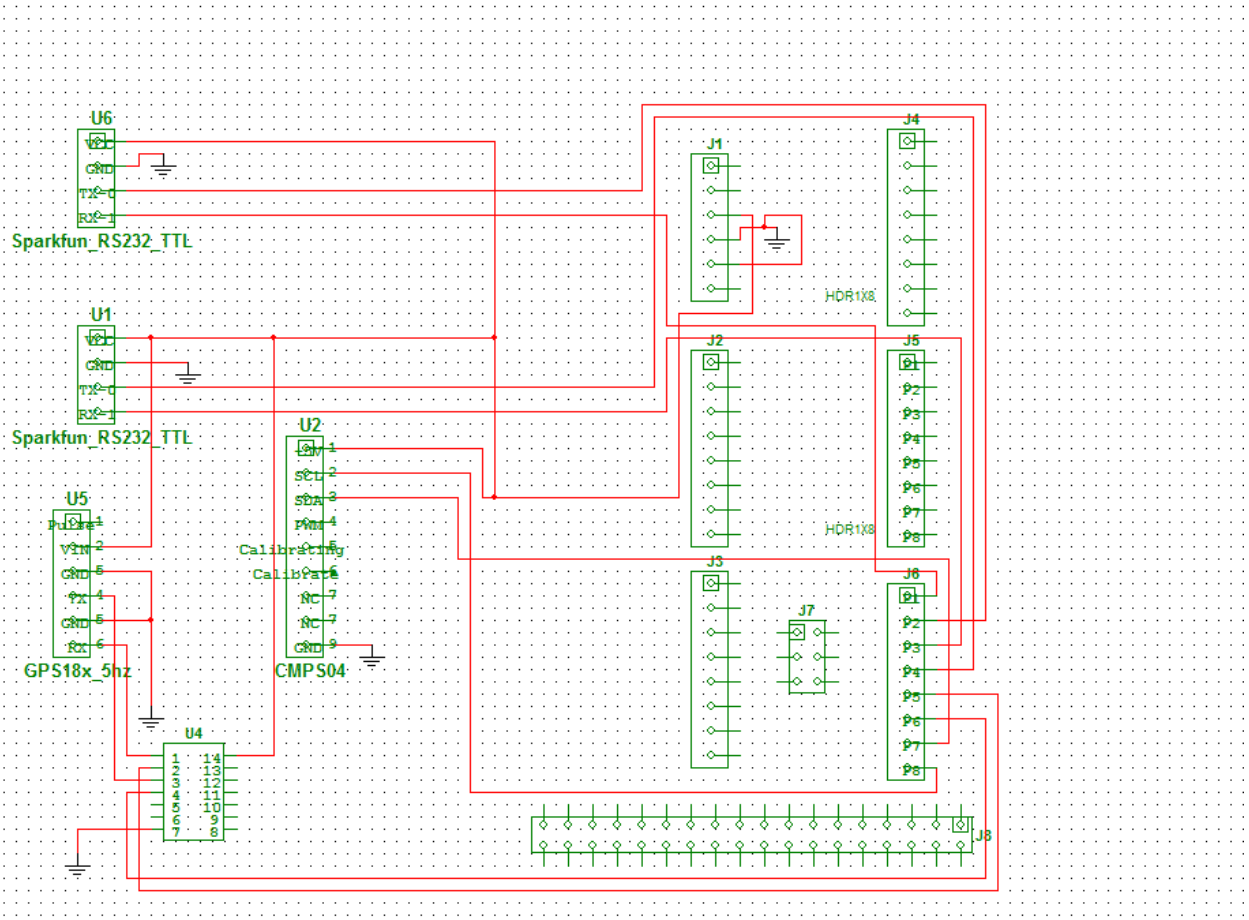


**Figure 18: Final Schematic**

The biggest problem we encountered when setting up the breadboard was the Arduino board being in TTL logic where both the Ricochet modem and AX1500 Motor controller were RS-232 logic. Since TTL logic only outputs from 0-5V and RS-232 logic outputs from -15V to +15V, all of the high signals that were sent out from the Arduino board at +5V were being received as low RS-232 signals because the voltage range of logic low is +3V to +15V and any low signals sent by the Arduino at 0V weren't recognized as RS-232 logic because logic high is defined as -15V to -3V. The RS-232 breakout IC solves this problem by converting TTL to RS-232 and vice versa.

The motor hookup was the most straight forward section for the electrical team. The motor controller has two different sets of connectors. The RS-232 side was connected through the Arduino board and the power side was handled on the other side of the board as seen in Figure 18. There were not any issues with the electrical hookup but the biggest problem we may encounter is waterproofing the controller while also watching to make sure nothing overheats inside of the Pelican case. The next step for the electrical team is to finish designing our PCB for the Arduino and have it sent out to be made for when we return so everything can be tested as a single unit. This will also allow for real world testing and the actual use in the kayak compared to only tests that can be done in lab.

### 3.3.2 PCB Design and Production

During our design phase we decided that a PCB shield for the Arduino would be the best method to include all of the peripherals fairly neatly. Due to the fact that most of the components were RS-232 compared to the TTL of the microcontroller, level shifters were a necessary addition to the shield. Initially there were 3 components that needed to be level shifted; the motor controller, the ricochet modem, and the GPS. However, after testing the GPS we realized that

only a signal inversion was necessary for the GPS signal to be compatible with TTL logic. Since there are two radios onboard, some type of switch between them was needed. At first we were going to create a physical switch to each radio to transmit, but finally decided that using a multiplexer for the data lines coming out of the microcontroller would be a more efficient method.

PCB design began with a program called Fritzing. We decided to go with this program because of its graphical interface and ease of use. Fritzing allowed you to build your PCB, schematic, and breadboard layout concurrently so you could change your PCB layout as you add parts rather than placing all the parts down at once. However, we quickly came to realize that Fritzing had very limited functionality for custom ICs that didn't allow us to create custom parts to our specifications. This prompted the switch to industry-level PCB design software in Ulti-Board

Ulti-Board was very feature rich, but the drawback was the learning curve was very steep. The database of parts that were usable in Multi-Sim was much larger than the one used in Fritzing and that was crucial for some of the parts required for the shield With this software we were able to create a production-level quality PCB Board. When creating the layout of the board, it was spaced such that all of the components would be directly attached to the PCB through header pins. However, during testing with the Pelican case, it came to our attention that the compass doesn't give useful data when it is inside of the case, so changes to the headers were required for the change in design.

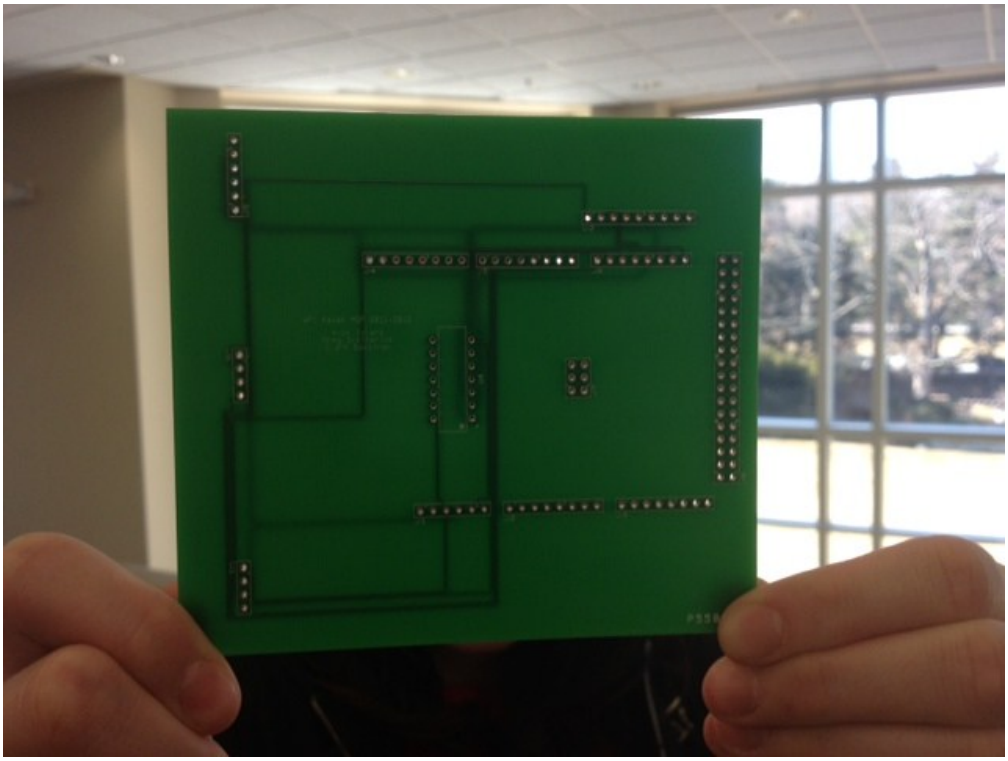

**Figure 19: Final PCB Design**

When sending out the PCB for fabrication, there were multiple online companies available. When searching for the correct company, cost and turnaround time were the most important factors. 4PCB.com was chosen because it offered a 5 day turnaround and it was the cheapest solution at $35 per board. The largest issue with PCB fabrication was that we misconceived 5 day turnaround to mean that the boards would be in our hand after 5 days of submittal, but we did not take shipping into account and that cost us valuable time. Another issue with the board was finding the correct header types for our shield because they had to be stackable header pins with the number of pins ranging between 6 and 38 pins.

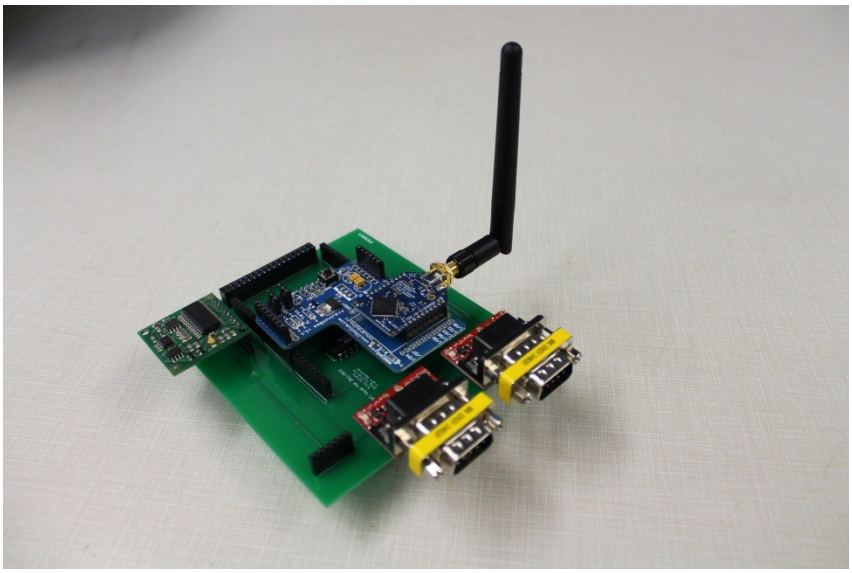

**Figure 20: PCB with components**

### 3.3.3 Power Harness Design

As is true with all electronics, our Kayak required a power harness. In other words, we needed to design a circuit capable of satisfying the diverse power requirements demanded by our various electrical components. Additionally, the power harness required some sort of manual power switch as well as an emergency off switch and something that would protect the electrical components from current surges. Finally, the power harness needed to be designed with the aquatic environment in mind and any water sensitive components would have to fit inside the provided Pelican case or be otherwise protected from the elements.

The AX1500 motor controller and the two motors themselves had the largest voltage requirements. The AX1500 required 12V to 40V for operation while the motors' data sheets simply stated that their voltage demand was highly variable and affected by numerous factors but went no further in providing a voltage range. Fortunately, Santa Clara University used the same motor controller and motors the previous year and was able to confirm that a 12V marine battery would be capable of powering all three devices.

Having a 12V battery meant we could fulfill the Richochet's 12V and Arduino Mega's 7-12V with only a voltage regulator instead of a buck or other step-down converter. Once the Arduino Mega was powered, it was only a matter of connecting the XBEE via its special shield to keep the XBEE powered. The remaining three components, the Garmin 18x, the compass and the Max 232 all were capable of running on a 5V current with fairly low current draws. It was discovered that all three could be powered by the 5V I/O pins on the Arduino, similar to the XBEE.

Santa Clara University also asked us to have the electrical components running off of a different battery then the motors and AX1500 motor controller. Separating the power sources for the motors and the other electrical components offered the practical benefit of allowing us to equip the two batteries with separate circuit breakers to match their unique current requirements. This also allowed us to turn off the two sides independently. Because of the smaller current demands of the non-motor half, this also meant that we could power them with a smaller battery which we could contain entirely inside of the Pelican case.

Originally we were going to run the motor controller off of the bigger battery as SCU had done, but after carefully contemplating the pros and cons of doing so, we realized that if the motors had a sudden surge in current draw, it could lower the battery's voltage output below the 12 volts needed by the motor controller and the motor controller would shut off. It was therefore decided that the motor controller should be moved to running off the smaller 12V battery to avoid this scenario.

A 30A circuit breaker and an emergency off-switch which could be quickly and easily pulled in case of an emergency were inserted between the larger 12V battery and the VMotor inputs in the AX1500. For the other battery, a 2.5A circuit breaker was selected as well as two

manual power switches, one for the AX1500 and one for the other remaining electrical components.

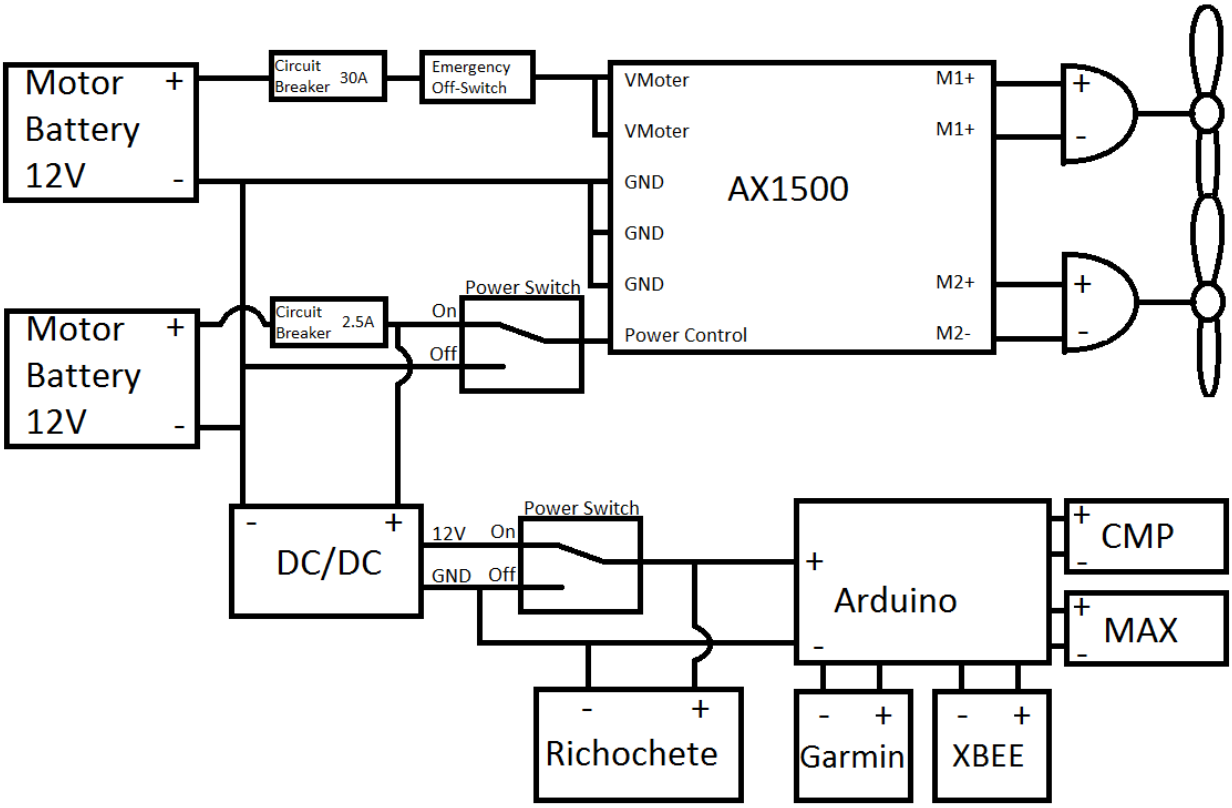


**Figure 21: Wiring Harness Diagram**

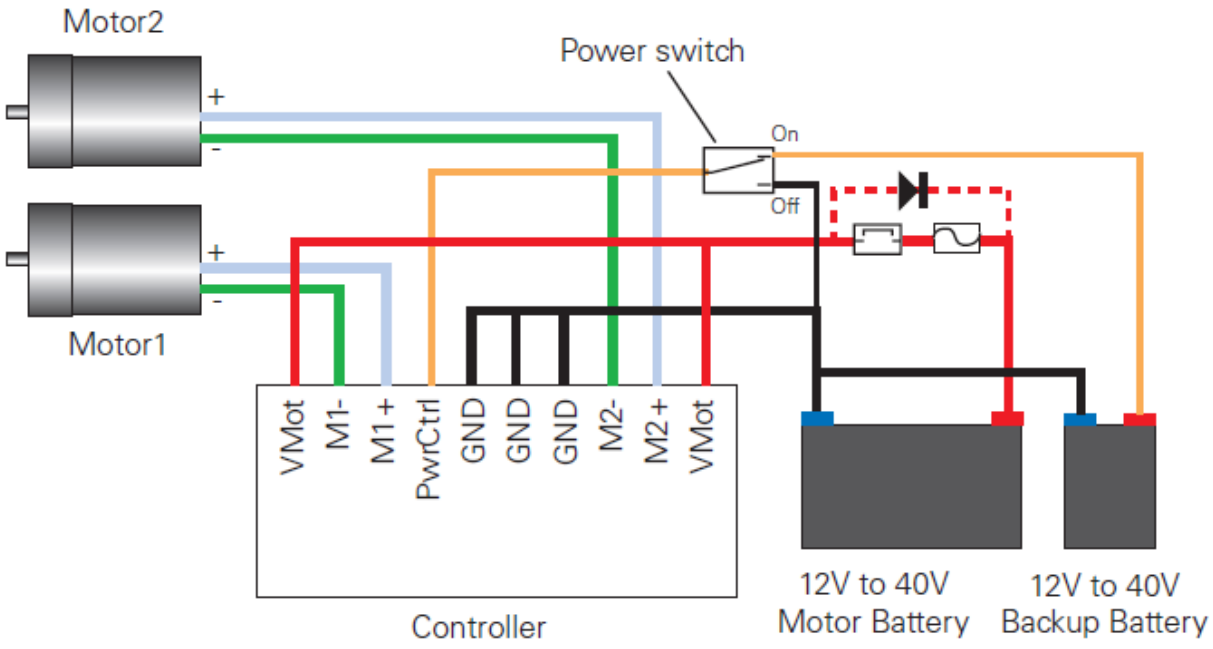


**Figure 22: Motor Controller Wiring**

## 3.4 Software Design

Software design covers two main components. The first of these is relates to the peripheral drivers. Because of the necessity of a number of external peripheral devices, such as a GPS and compass, the software design requires a number of drivers to interface between the micro-controller and these devices. In order to create these drivers, research was done into the specifics of each device, specifically their respective data sheets and documentation. For example, in order to interface with the GPS, the drivers need to be set up with the correct baud rate and be able parse the GPS sentences sent across the serial lines. Parsing these sentences proved to be a non-trivial matter, as each sentence has different sizes and formats. For the

compass, the major component was communicating the correct device address, and then appending the corresponding data bytes together in the correct way.

The other software design consideration was the design of the packet protocol. The packet protocol is responsible for defining how data and commands are communicated back and forth between the base station and the micro-controller on board the kayak. This protocol must be agreed upon by both devices in order to assure correct data transfer between these devices. Failure to assure this may result in misinterpretation of data and/or commands between the devices. This protocol was previously encoded in the programming for the two BASICX stamps. However, the code was undocumented and ambiguous, making it extremely difficult for the protocol to be ported over into the new language/micro-controller. In order to correct this, the team worked with SCU to redefine and document this protocol. This new documentation allows for the protocol to be implemented easily in any language, as it documents byte by byte what is transmitted between the devices. Once this documentation was created, the protocol was programmed for the micro-controller.

| **Kayak Command**: 32 bit (4 byte); -1 to 1 floating point values specifying longitudinal and rotational speeds | | | | |
|---|---|---|---|---|
| Field Name | Length (bytes) | Data Type | | |
| Longitudinal Velocity | 2 | float | A floating point value between -1 and 1 where negative values indicate a backward direction | |
| Rotational Velocity | 2 | float | A floating point value between -1 and 1 where negative values indicate a counter clockwise direction. | |
| | | | | |
| **Kayak Telemetry** (Compass, GPS) | | | | |
| **Field Name** | **Length (bytes)** | **Data Type** | **Data Source** | **Description** |
| Telemetry Payload Length | 1 | integer | - | Indicates the length of the data, including this field |
| Radio Info (arbitrary) | 2 | bytes | - | Information that is included for the use with the Ricochet packet modems |
| Telemetry Payload Length (for use with amigobots) | 1 | integer | - | information that is included for operation with the amigobots; this static field is not taken into account with more modern systems that make use of the pioneers as the cluster platform. |
| Payload Header Delimiter | 1 | integer | - | 0xB0; A delimiter used to indicate that the following bytes are from the sensors assocatied with the robot system. |
| Heading | 2 | integer | Compass | Integer value of the compass heading to a tenth of a degree; |
| Time | 4 | single | GPGGA | Single precision time value |
| Latitude | 4 | integer | GPGGA | Integer value of latitude to 5 decimal places |
| Latitude Hemisphere | 1 | char (ASCII) | GPGGA | ASCII value of 'N' or 'S' |
| Longitude | 4 | integer | GPGGA | Integer value of longitude to 5 decimal places |
| Longitude Hemisphere | 1 | char (ASCII) | GPGGA | ASCII value of 'E' or 'W' |
| Number of Satellites | 1 | integer | GPGGA | |
| Horizontal Dilution | 2 | integer | GPGGA | Integer value of the horizontal dilution to one decimal place |
| Course Over Ground | 2 | integer | GPVTG | Integer value of the heading (course over ground) to a thenth of a degree |
| Magnetic Course Over Groun | 2 | integer | GPVTG | Integer value of the magnetic (heading) course over ground to a tenth of a degree |
| Speed Over Ground | 2 | integer | GPVTG | Integer value of the system speed in km/h |
| Magnetic Variation | - | - | GPRMC | 20120214 (MR): From the controller code it appears that magnetic variation is being derived rather than being extracted from GPS data; This information is included in the GPRMC sentence. |
| Checksum | 1 | integer | - | Longitudinal Parity Check of all data before this field. |
| Delimiter | 2 | integer | - | 0x0A0D; <CR><LF> - Carriage Return and Line Feed. |

**Figure 23: Packet Protocol**

# CHAPTER 4: RESULTS AND ANALYSIS

## 4.1 Testing

The following section describes the various testing techniques employed during the project, including mechanical testing and electronics testing.

### 4.1.1 Mechanical Testing

To ensure that the best possible design and materials were chosen, extensive testing was performed on several aspects of the chassis design. The tests include constructing a scale mockup of the chassis, testing the chosen adhesive that joins the PVC pads to the HDPE kayak, and the corrosion of the aluminum frame and fasteners. All of these tests ensured that the correct design and materials were selected.

The mockup served a few purposes. It created a physical representation of the chassis design, which was helpful in visualizing the size of the chassis and how the chassis will fit on the kayak. These metrics were invaluable in the design process because it aided in the redesign process, which included enlarging the chassis frame to accommodate the Pelican box and the two batteries in battery boxes. The mockup, illustrated above in Figure 12, was constructed using extruded polystyrene insulation board. This material was chosen for its manufacturability, and because it is a relatively light material. Half of one 4 in x 8 in x 2 in sheet was to construct the entire model The half a board was cut to various lengths with dimensions 1.25 in x 1.25 in. The pieces were then assembled together using hot glue. Cardboard was used to emulate the plates for the motor mounts and the v-shaped feet of the chassis. The mockup lead to small design changes, but for the most part it confirmed the validity of the design.

The adhesive underwent a few different tests. The tests were performed using the seat from the kayak because the seat, being made from the same material as the kayak, would be a

good representation of the bond between the pads and the kayak surface. The first test was to determine whether or not the adhesive would bond the PVC pad to the HDPE kayak. The bond was tested by placing a load on the D-ring of the PVC pad. The load bearing tests were done in both dry and wet conditions. To start the dry test 15 pounds was hung from the pad for roughly 21 hours. Then 25 pounds was added bringing the total weight to 40 pounds, which was left for roughly 101 hours. Meaning in total the dry bond was tested for 122 hours.

Since the kayak would be operating largely in saltwater environment the wet test was done using an artificial salt water solution. The salinity of sea water is roughly 3.5 % or 35 g/L. The solution was made using approximately three gallons of water, which is nearly 11 L, and dissolving about 385 grams of salt in the water. The seat with the pad on it was then placed pad down, so that the pad is completely submerged, in the solution. This was left to soak for roughly 17 hours before beginning load testing. Instead of starting at 15 pounds though, 40 pounds was use. The weight was then left alone for roughly 121 hours. The bond between the pad and the kayak seat proved to be exceptional. Testing verified that in both environments the bond was more than adequate.

The final test that was performed on the design was a corrosion test on a sample joint. The test used the same solution of artificial sea water explained above. The joint was submerged in a small container and left for 120 hours. This test was to ensure that the galvanic properties expected for these materials were present.

These tests ensured that a high quality design had been created and chosen. The model of the chassis revealed a few minor issues with the design that would normally have been overlooked when using solely modeling software. The bond between the pad and the kayak needs to be extremely solid, so that there is no question of the chassis falling off the kayak,

which was confirmed in the testing of the adhesive. A structure really isn't a structure if there isn't a way to join together different pieces. If the joints were to fail because the aluminum and the fasteners reacted to each other when wet, then the entire chassis would eventually fail. For this reason the test on a sample joint was necessary, which confirmed that corrosion between the two would be minimal. Each one of these tests was critical in the decision process for the major features of the chassis design.

The stability of the robotic kayak system is an important aspect of the design. To measure the stability a rudimentary test was performed. The test included removing the battery and electronic stack, and replacing them with 40 pounds of weights. Furthermore, the motors were waterproofed to prepare the tipping of the kayak. Once set up the kayak will be pushed until the tipping point. The tipping point of the kayak was measured in degrees from the horizontal. The kayak tipped 50 degrees from the horizontal before flipping over, which demonstrates that the kayak is particularly stable.

### 4.1.2 Electronics Testing

In order to test the range of the radio communications, the kayak was placed under direct operator control. The micro-controller was set up to transmit the GPS coordinates continuously as the kayak was driven. A base location was measured at the shoreline to compare with the readings from the kayak. The kayak was then driven away from the base station until GPS data was no longer being transmitted to the base station. The last transmitted coordinate and the base coordinate were then compared to find the linear distance between the two. In the initial range testing with the XBEE radios, the team found the measured range to be significantly less than what was expected. It was determined that the cause of this range loss was due to the location of the on board antenna. In this initial testing, the XBEE antenna was located within the Pelican

case, which hindered the range due to its close proximity to the ground and the material interference from the case itself. In later testing, the antenna was extended to the outside of the kayak, located on top of an antenna mast, located five feet above water. Once the antennas were relocated to the top of the mast, the maximum transmit distance for the Xbee was determined to be 0.5 kilometers (0.31 miles).

The testing for the GPS and compass were conducted in a similar manner. The GPS and compass were set to transmit data to the base station, with specific messages to denote correct GPS and compass data. The kayak was then driven around for a significant period of time to see how often errors occurred. During this testing, the team found two major contributions to GPS and compass error. Firstly, when the GPS is being read by the micro-controller, if the baud rate for the radio communications transmitting the data was slower than the GPS input, the data would be asynchronous with what was being transmitted. To correct this, the team increased the baud rate of the radio communications, which prevented the GPS from overwriting data before it was transmitted. In addition to this error, the team discovered a similar reception error to that of the XBEE radio. Again, to correct this, both the GPS and compass were repositioned to the antenna mast, with cables running between the devices and the micro-controller.

**Figure 24: Kayak in the water**

### 4.1.3 Software Testing

One of the most important testing aspects to consider was full system integration. Specifically, the team's kayak needed to be able to work with the existing Santa Clara kayak project. In order to test this, the team was provided with a base station from SCU to test with. This base station contains all of the off-board software and controls to run a complete ASV kayak fleet. The first test for full integration was communications testing between the base station and the kayak. This was done to ensure that both data and commands were being transmitted and received correctly by each device. To do this, the kayak was left under operator control while communicating to the base station. The base station was monitored to view the correct interpretation of the telemetry data from the kayak. Similarly, the commands to the kayak were monitored to assure that the kayak was perceiving and executing the commands correctly. After this testing was completed successfully, the motor controller was connected to the micro-controller and the kayak was allowed to run fully autonomously. During this stage, a support

vessel kept pace with the kayak as a precaution should anything had gone wrong (such as communication loss, or incorrect commands to the motors).

## 4.2 Safety and Manufacturability

During the design and manufacturing process, it was important to consider the safety and manufacturability of our design. It is crucial that no one is injured by handling, assembling, or operating our device. As a result, the team added plastic end caps to all of the aluminum tubing and also filed down any sharp edges or protruding surfaces.

When designing any sort of device, it is necessary to consider the manufacturability of that device. Careful consideration went into the design of the chassis in terms of manufacturability, making sure that any part could be easily made with hand machining and basic CNC milling. The chassis design can be easily replicated by anyone with the CAD drawings and assembly instructions.

# CHAPTER 5: RECOMMENDATIONS AND CONCLUSIONS

This MQP set out to design and manufacture an Autonomous Surface Vehicle capable of integrating with Santa Clara University's existing and expanding fleet. The biggest original core requirements mandated that the final product be a kayak that is able to communicate with the existing fleet or base station by way of radio, be fully operational un-manned, and be capable of collecting and sending GPS and compass data back to the base station. After conducting research into possible additional features that would improve the overall kayak design the WPI team elected to also design a universal chassis capable of mounting a variety of kayaks, to upgrade the on-board microcontroller to an Arduino Mega from the previous two BasicX boards, and equip the robot with both a Ricochet and XBEE radio to allow multiple methods of communication. By the end of the MQP, all of these features had been successfully implemented and included in the final product.

The completed WPI kayak will benefit Santa Clara University's future efforts in a number of ways that most previous kayaks are unable to. First of all, the universal chassis designed by WPI can be assembled or disassembled within minutes and is capable of mounting a variety of kayak designs. This allows SCU a level of flexibility and adaptability that they didn't have before; the speed and ease with which they can now switch and transport kayaks will save them a lot of time and effort when running field tests. The Arduino Mega microcontroller has much more raw computing power then their old BasicX boards and therefor offers SCU more options for adding features in the future. Additionally, the Arduino consolidates all of the code previously stored on two separate BasicX boards in one place and snaps onto a shield equipped with the bulk of the electronic circuit, thereby reducing clutter and making the whole apparatus much more manageable. Finally, the inclusion of two radios, the Ricochet and XBEE, means that

WPI's kayak will be able to integrate immediately with the existing fleet and maintain long-term relevance and the project switches to XBEE radios.

Moving forward, some logical next steps for the Santa Clara Kayak Project to take include constructing a chassis with lighter material and continuing to work towards full ROS integration. Our upgrades improved the overall adaptability and manageability of the kayaks and are expected to aid Santa Clara University greatly as they continue to be test the kayaks for various real world applications.

## SOURCES

(2011). WillowGarage Software: ROS [Online]. Available HTTP: http://www.willowgarage.com Directory pages/software/ros-platform 2011

(2011). Robotic Operating System: Documentation [Online]. Available HTTP: http://www.ros.org Directory: wiki/ 2011

(2011). Arduino: Language Reference [Online]. Available HTTP: http://www.arduino.cc Directory: en/Reference/HomePage 2011

A.G. Astray Et. Al, "ANGLER: Autonomous Network for Gradient Location in Environmental Research," B.S. thesis, Dept. Mech. Eng., SCU, Santa Clara, CA, 2011.

P. Manhacek Et. Al "Cluster Space Control of a 2-Robot System as applied to Autonomous Surface Vessels," Robotics Sys. Lab. SCU, Santa Clara, CA.

P. Manhacek Et. Al "Dynamic Guarding of Marine Assets through Cluster Control of Automated Surface Vessel Fleets," *IEEE Transactions on Mechatronics.*

C. Kitts Et. Al "Experiments in the Control and Application of Automated Surface Vessel Fleets" Robotics Sys. Lab. SCU, Santa Clara, CA.

(2011, October 5). "Gulf of Mexico restoration*,"* [Online]. Available: http://www.bp.com/sectionbodycopy.do?categoryId=41&contentId=7067505

# APPENDIX A: PROJECT BUDGET

| Qty | Item | Unit Price | Total |
|---|---|---|---|
| 4 | Architectural Aluminum Tube (Alloy 6063) Square, 1-1/4" X 1-1/4", 1/8" Wall, 6' Length | $28.84 | $115.36 |
| 1 | Architectural Aluminum Tube (Alloy 6063) Square, 1-1/4" X 1-1/4", 1/8" Wall, 3' Length | $17.48 | $17.48 |
| 1 | Architectural Aluminum Tube (Alloy 6063) Square, 1" X 1", 1/8" Wall, 3' Length | $16.27 | $16.27 |
| 1 | Hot-DIP Galv STL Low Strg Hex Head Cap Screw 1/4"-20 Thread, 3" Length, packs of 25 | $8.04 | $8.04 |
| 1 | ASTM A194Hex Nut Hot-Dipped Galvanized Steel, 1/4"-20 Thread Size, packs of 50 | $12.56 | $12.56 |
| 1 | Hot Dipped Galvanized Steel Flat Washer1/4" Screw Size, packs of 100 | $1.85 | $1.85 |
| 1 | SQ Neck Carriage Screw Hot-Dipped Galv, 1/4"-20 Thread, 2" Length, packs of 100 | $11.31 | $11.31 |
| 1 | Zinc-Plated Steel Machine Screw Hex Nut 8-32, packs of 100 | $1.49 | $1.49 |
| 1 | Zinc-Pltd Pan Head Phillips Machine Screw 8-32 Thread, 1-3/4" Length, packs of 100 | $7.14 | $7.14 |
| 1 | Zinc-Plated Steel Flat Washer NO. 8 Screw Size, packs of 100 | $1.42 | $1.42 |
| 1 | Zinc-Plated Steel Wing Nut 1/4"-20 Thread Size, 1-3/32" Wing Spread, packs of 100 | $9.83 | $9.83 |
| 3 | TAP Plastics Polyweld Adhesive 2 oz | $37.95 | $113.85 |
| 4 | D-Ring Pads | $5.99 | $23.96 |
| 1 | Aluminum Tread Plate 3003 H14: .125" x 24" x 24" | $29.48 | $29.48 |
| 12 | Square Finishing plug for Tube | $0.14 | $1.68 |
| 4 | Crown Bolt Zinc Plated 3/8 in. x 10-1/2 in. Turnbuckle Hook/Eye | $3.27 | $13.08 |
| 2 | Crown Bolt Zinc Plated 1/4 in.-20 x 36 in. Threaded Rod | $1.97 | $3.94 |
| 1 | Xbee-Pro XSC | $67.20 | $67.20 |
| 1 | Arduino UNO Microcontroller Board | $29.95 | $29.95 |
| 1 | 12V Marine Battery | $150.00 | $150.00 |
| | | **Subtotal** | **$635.89** |

| | Parts Provided by SCU | Unit Price | Total |
|---|---|---|---|
| 1 | Devantech CMPS03 Magnetic Compass Module | $54.00 | $54.00 |
| 1 | Garmin 18 differential GPS | $80.00 | $80.00 |
| 1 | Roboteq AX1500 Motor Controller | $275.00 | $275.00 |
| 2 | Minn Kota 30 Trolling Motors | $100.00 | $200.00 |
| 1 | Kayak | $250.00 | $250.00 |
| 1 | Pelican Case 1200 | $30.00 | $30.00 |
| | | **Subtotal** | **$889.00** |
| | | **TOTAL** | **$1,524.89** |

## APPENDIX B: PRESENTATION SLIDES


### Project Goals

The goal of this project was to design and manufacture an Autonomous Surface Vehicle (ASV) in collaboration with Santa Clara University (SCU) that would integrate with their existing ASV fleet.

WPI

Santa Clara University

### Project Background

- Multiple Autonomous Surface Vehicle project at Santa Clara University
  - Cluster-based water craft system
- Improve existing kayak design
  - Universal chassis design
  - Electronics and improved radios
  - Use of a more powerful microcontroller
- Santa Clara University is the "customer"
  - Product design process
  - List of requirements resulting in deliverables

SCU Design

### Chassis Design

- Universal chassis fits multiple kayak models
  - Allows for the chassis to be designed and built independently of the hull
- Adjustable components allow for flexibility and easy configuration

### Electronics Design

- Updated Microcontroller Architecture
  - Using Arduino Mega2560
  - Allows for future expandability
  - Extensive community support
- Multiple Radios
  - Rinacher Modem to maintain compatibility with old system
  - Xbee Pro 900Mhz for future projects
- Custom PCB Shield
  - Plug-and-play system allows for quick and easy setup

### Software Design

- Device drivers
  - Wrote drivers for the GPS, compass and motor controllers
  - Modular code design
- Packet Protocol
  - Redefined wireless communication protocol
  - New protocol is simpler to implement and allows for extensibility

## Testing

- Stress testing
  - Corrosion and immersion testing on pads
- Electronics Testing
  - Wireless range testing
- Software integration testing
  - Communications work with Santa Clara's system
  - Effective GPS parsing

## Project Impact

- Gradient Mapping
  - Multiple ASVs map out concentrations of solutes within bodies of water
- Asset Protection
  - Form a perimeter around assets or dive sites
- Environmental Cleanup
- Search and Rescue

## Future Work

- Decentralized robot control (Full Autonomy)
  - Current system requires a central base station
- Robot Operating System (ROS)
  - Using ROS would allow for high-level expansibility and autonomy
- Sensor package
  - The system allows for additional sensors beyond the current suite required for telemetry, such as environmental sensors for data collection
- Lightweight materials
  - Use of composites or reduction of chassis structure to reduce weight

## Conclusions

- Accomplishments
  - Designed and fabricated an upgraded autonomous surface vehicle to integrate with Santa Clara's current fleet
  - Updated design includes a universal chassis and a more versatile electronics architecture
- Big Picture
  - Multiple ASV systems can be adapted to many different purposes

## Video

## QUESTIONS?

# APPENDIX C: GPS SENTENCE FORMAT

**GPRMC** - Recommended Minimu Specific GPS/TRANSIT Data (RMC)

$GPRMC,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,<10>,<11>,<12>,*hh<CR><LF>

| | Field Name | Length (bytes) | Offset (byte) | Data Format | Details |
|---|---|---|---|---|---|
| | Sentence ID | 6 | 0 | $GPRMC | Indicates that the sentence strucure is GPRMC |
| <1> | (UTC) Time | 6 | 7 | hhmmss | UTC |
| <2> | Status | 1 | 14 | A \| V | A = Valid position, V = NAV receiver warning |
| <3> | Latitude | 8 | 16 | ddmm.mmm | Leading zeros are transmitted |
| <4> | Latitude Hemisphere | 1 | 25 | [N,S] | N = North, S = South |
| <5> | Longitude | 8 | 27 | ddmm.mmm | Leading zeros are transmitted |
| <6> | Longitude Hemisphere | 1 | 36 | [E,W] | E = East, W = West |
| <7> | Speed Over Ground | 5 | 38 | xxx.x | 000.0 to 999.9 knots; leading zeros are transmitted |
| <8> | Course Over Ground | 5 | 44 | xxx.x | 000.0 to 359.9 degrees; leading zeros are transmitted |
| <9> | (UTC) Date | 6 | 50 | ddmmyy | UTC |
| <10> | Magnetic Variation | 6 | 57 | xxx.xx | 000.0 - 180.0 degrees; leading zeros are transmitted |
| <11> | Magnetic Variation Direction | 1 | 64 | [E,W] | E = East, W = West |
| <12> | Mode Indicator | 1 | 66 | [A,D,E,N] | A = Autonomous<br>D = Differential<br>E = Estimated<br>N = Data not valid |
| *hh | Checksum | 2 | 68 | | Checksum |
| | Delimiter | 2 | 71 | | Carriage Return, Line Feed |

$GPRMC,235959,A,3851.3561,N,09447.9382,W,00030,221.9,071103,003.3,E*69

**GPGGA** - Global Positioning System Fix Data (GGA)

$GPGGA,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,M,<10>,M,<11>,<12>*hh<CR><LF>

$GPGGA,175250.00,3333.42646711,N,11153.35317335,W,2,07,1.3,406.854,M,-26.294,M,10.4,0100*4F

| | Field Name | Length (bytes) | Offset (byte) | Data Format | Details |
|---|---|---|---|---|---|
| | Sentence ID | 6 | 0 | | Indicates that the sentence structure is GPGGA |
| <1> | UTC Time | 6 | 7 | hhmmss.s | UTC |
| <2> | Latitude | 9 | 14 | ddmm.mmmm | Leading zeros are transmitted |
| <3> | Latitude Hemisphere | 1 | 24 | [N,S] | N = North, S = South |
| <4> | Longitude | 10 | 26 | dddmm.mmmm | Leading zeros are transmitted |
| <5> | Longitude Hemisphere | 1 | 37 | [E,W] | E = East, W = West |
| <6> | GPS Quality Indication | 1 | 39 | [0,1,2,6] | 0 = fix not available<br>1 = Non-differential GPS fix available<br>2 = Differential GPS (WAAS) fix available<br>6 = Estimated |
| <7> | Number of Satellites | 2 | 41 | xx | 00 to 12; leading zeros are transmitted |
| <8> | Horizontal Dilution | 4 | 44 | xx.x | Horizontal dilution of precision, 0.5 to 99.9 |
| <9> | Antenna Height Above Sea Level | | | xxxxx.x | Geoidal height, -999.9 to 9999.9 meters |
| <10> | Geoidal Height | 6 | 49 | xxxxx.x | Geoidal height, -999.9 to 9999.9 meters |
| <11> | Null (Differential GPS) | | | xxx | Age of differential corrections, in seconds |
| <12> | Null (Differential Reference Station ID) | | | xxxx | Reference station identification |
| *hh | Checksum | | | *xx | Packet Checksum |
| <CR><LF> | Delimiter | | | <CR><LF> | Carriage Return, Line Feed |

**GPVTG** - Track Made Good and Ground Speed (VTG)

$GPVTG,<1>,T,<2>,M,<3>,N,<4>,K,<5>*hh<CR><LF>

$GPVTG,103.85,T,92.79,M,0.14,N,0.25,K,D*1E

| | Field Name | Data Format | Details |
|---|---|---|---|
| | Sentence ID | $GPVTG | Indicates that the sentence structure is GPVTG |
| <1> | True Course Over Ground | xxx | 000-350 degrees; leading zeros transmitted |
| <2> | Magnetic Course Over Ground | xxx | 000-350 degrees; leading zeros transmitted |
| <3> | Speed Over Ground | xxx.x | 000.0 to 999.9 knots; leading zeros transmitted |
| <4> | Speed Over Ground | xxxx.x | 0000.0 to 1851.8 km/h; leading zeros transmitted |
| <5> | Mode Indicator | [A,D,E,N] | A = Autonomous<br>D = Differential<br>E = Estimated<br>N = Data not valid |
| *hh | Checksum | *hh | Checksum |
| <CR><LF> | Delimiter | <CR><LF> | Carriage Return, Line Feed |

## APPENDIX D: ARDUINO CODE

```
/*
 * WPI Robotic Kayak MQP
 * With contributions from Doug Renfro
 *
 *
 *  Provides functionality to receive directional commands
 *  and package positional data
 *
 */


// Ricochet -- Serial3
// Motor Controller -- Serial2
// GPS -- Serial1

//Include Extra Libraries

#include <Wire.h>



#define ADDRESS 0x60 // Compass I2C address


//Function Declarations
void reverseBytes(int length);
int convertCompass(float compass);
float convertTime(char time[]);
int convertToIntPlusOne(char kCourse[]);
byte convertHemisphere(char hemisphere);
byte convertSatellites(char pos[]);
long convert_lat_lon(char pos[]);

//Global Declarations
char gpsArray[2][14][20];
String Data[2];
String temp;
byte errorArray[23] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
byte error2Array[23] =
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
byte start[5] = {0,0,0,0,176};
byte identifier[8] = "*Tosca*";
boolean gpsPresent = false;
char channel1[6];
char channel2[6];
byte * data;
int availability;

int compass;
```

```
float time;
long latitude;
byte latHem;
long longitude;
byte lonHem;
byte satellite;
int dilution;
int courseOG;
int magneticCourse;
int speedOG;


void setup() {
//Setup Serial ports and compass port
  Wire.begin();         // Compass
  Serial2.begin(9600);  // Motor Controller

  Serial1.begin(19200); // GPS
  Serial3.begin(38400); // Ricochet Modem
  Serial.begin(38400);     // XBEE
}

void loop() {
  char gpsData;
  int incomingTemp;

  int numStrings = 0;

  int flag2 = 0;
  int i = 0;
  int j = 0;
  byte byteLow;
  byte byteHigh;
  float command;
  gpsPresent = true;

  /*  Receive Commands:
 *
 *  Input bytes and convert to motor controller command
 *
 */


  if(Serial.available())
  {
    byte byteLow1 = Serial.read();
    byte byteHigh1 = Serial.read();
    byte byteLow2 = Serial.read();
    byte byteHigh2 = Serial.read();
    float command1 = ((byteHigh1 << 8) + byteLow1)/100;
    float command2 = ((byteHigh2 << 8) + byteLow2)/100;
    convertChannel1(command1);
```

```
    convertChannel2(command2);
    Serial.println(channel1);
    Serial.println(channel2);


    Serial2.println(channel1);  // Mcontroller
    Serial2.println(channel2);
  }


//Gather compass data as a floating point
  float bearing = compassBearing();

/*
 *  GPS Collection:
 *
 *  Collects GPS data from 2 input strings
 *  GPGGA and GPVTG.
 *  Stores the data into a 3-Dimensional array.
 *  Array Section 1:Whether the information is part
 *  of GPGGA or GPTVG.
 *  Array Section 2:Section of original input
 *    (0 is section after GPGGA/GPTVG)
 *  Array Section 3:The strings to be converted
 */

//Continue reading data until both strings
//are collected
  while(numStrings != 2)
  {

//Check avilabality and delay to allow hardware to catch up
      availability =  Serial1.available();
      delay(1);

//If no byte to read, break set flag
      if(availability == 0)
      {
        gpsPresent = false;

        break;
      }
    else {
//Read byte
      gpsPresent = true;
    String hold;
  char inByte= 'x';
  while (inByte != '\n')
  {
  if (Serial1.available())
  {
  // appends the incomming characters to a string sentence
```

```
    inByte = Serial1.read();
    hold += inByte;

  }
  }
  //Sorts the sentence based on the GPS header. Disregards extraneous
headers
  if (hold.substring(0,6).compareTo("$GPGGA") == 0)
  {
   Data[0] = hold;

}
else if (hold.substring(0,6).compareTo("$GPVTG") == 0)
   {
     Data[1] = hold;

   }

    }
      removes the header for string parsing
 for (int n =0; n < 2; n++)
 {
  String docked;
   docked =  Data[n].substring(7);

 for (int m = 0; m <docked.length(); m++)
 {
   if (i > 13)
   {

     i = 13;
   }
   if (j >19)
   {

     j = 19;
   }

   char gpsData = docked[m];

 //Passes GPS data into the GPS data storage array
   if(gpsData == '\n')
        {
          gpsArray[n][i][j] = '\0';
          //Serial.println(gpsArray[flag2-1][i]);
          i = 0;
          j = 0;
        // Serial.println("new line");


        }
//Parses GPS String into individual components
```

```
//When a comma is read, end of a section in string.
//Add null terminating character, reset variable j,
//and increase place in second dimension of array
        else if(gpsData == ',')
        {
          gpsArray[n][i][j] = '\0';
         // Serial.println("comma");
          //Serial.println(gpsArray[flag2-1][i]);
          j = 0;
          i++;
        }
//Otherwise place collected data into array and
//increase third dimension of array
        else
        {
          gpsArray[n][i][j] = gpsData;
          j++;
        }
 }
 }

  Serial.write(identifier, 7);
  Serial.write(start, 5);


//If no GPS data was read, send compass data and
//send error bytes (all 0's).
  if(!gpsPresent)
  {
    //Turn on No GPS LED
//   Serial.println("nogps");
    //Serial.println(bearing);
    int compass = convertCompass(bearing);
    data = (byte *) &compass;
    Serial.write(data, sizeof (compass));
    Serial.write(errorArray, 23);
    Serial.write('\r');
    Serial.write('\r');
  }


//If any of these 1-Dimensional arrays are
//NULL, then there is no GPS fix.
//Send compass data and error bytes (all 0's);
 else if(gpsArray[0][9][0] == '\0' || gpsArray[1][0][0] == '\0' ||
gpsArray[1][2][0] == '\0' || gpsArray[1][6][0] == '\0')
  {
    //Turn on No Fix LED

    int compass = convertCompass(bearing);
    data = (byte *) &compass;
    Serial.write(data, sizeof (compass));
```

```
    Serial.write(error2Array, 23);
    Serial.write('\r');
    Serial.write('\r');
  }

//Otherwise, perform conversions and send data
  else
  {
   //Serial.println("has gps");
   //Turn off GPS LED
//GPS conversion calls and assignments

    compass = convertCompass(bearing);
    time = convertTime(gpsArray[0][0]);
    latitude = convert_lat_lon(gpsArray[0][1]);
    latHem = convertHemisphere(gpsArray[0][2]);
    longitude = convert_lat_lon(gpsArray[0][3]);
    lonHem = convertHemisphere(gpsArray[0][4]);
    satellite = convertSatellites(gpsArray[0][6]);
    dilution = convertToIntPlusOne(gpsArray[0][7]);
    courseOG = convertToIntPlusOne(gpsArray[1][0]);
    magneticCourse = convertToIntPlusOne(gpsArray[1][2]);
    speedOG = convertToIntPlusOne(gpsArray[1][6]);

//Gather the array of bytes--(data = (byte *) &INFO.
//Send the bytes over serial
    data = (byte *) &compass;
    reverseBytes(2);
    Serial.write(data, sizeof (compass));
    //Serial.write('\r');

    data = (byte *) &time;
    reverseBytes(4);
    Serial.write(data, sizeof (time));
    //Serial.write('\r');

    data = (byte *) &latitude;
    reverseBytes(4);
    Serial.write(data, sizeof (latitude));
    //Serial.write('\r');

    data = (byte *) &latHem;
    Serial.write(data, sizeof (latHem));
    //Serial.write('\r');

    data = (byte *) &longitude;
    reverseBytes(4);
    Serial.write(data, sizeof (longitude));
    //Serial.write('\r');

    data = (byte *) &lonHem;
    Serial.write(data, sizeof (lonHem));
```

```
    //Serial.write('\r');

    data = (byte *) &satellite;
    Serial.write(data, sizeof (satellite));
    //Serial.write('\r');

    data = (byte *) &dilution;
    reverseBytes(2);
    Serial.write(data, sizeof (dilution));
    //Serial.write('\r');

    data = (byte *) &courseOG;
    reverseBytes(2);
    Serial.write(data, sizeof (courseOG));
    //Serial.write('\r');

    data = (byte *) &magneticCourse;
    reverseBytes(2);
    Serial.write(data, sizeof (magneticCourse));
    //Serial.write('\r');

    data = (byte *) &speedOG;
    reverseBytes(2);
    Serial.write(data, sizeof (speedOG));
    //Serial.write('\r');

    Serial.write('\r');
    Serial.write('\r');


//<-----------EXTRA PRINT INFORMATION FOR ERROR CORRECTION------------
->

   /* Serial.println("----------------------HERE------------------");

    Serial.println(gpsArray[0][1]);
    Serial.println(gpsArray[0][1]);
    Serial.println(gpsArray[0][2]);
    Serial.println(gpsArray[0][3]);
    Serial.println(gpsArray[0][4]);
    Serial.println(gpsArray[0][6]);
    Serial.println(gpsArray[0][7]);
    Serial.println(gpsArray[1][0]);
    Serial.println(gpsArray[1][2]);
    Serial.println(gpsArray[1][6]);
/*

/*
    Serial.println(convertTime(gpsArray[0][0]));
    Serial.println(convert_lat_lon(gpsArray[0][1]));
    Serial.println(convertHemisphere(gpsArray[0][2]));
    Serial.println(convert_lat_lon(gpsArray[0][3]));
```

```
    Serial.println(convertHemisphere(gpsArray[0][4]));
    Serial.println(convertSatellites(gpsArray[0][6]));
    Serial.println(convertDilution(gpsArray[0][7]));
    Serial.println(convertCourse(gpsArray[1][0]));
    Serial.println(convertCourse(gpsArray[1][2]));
    Serial.println(convertSpeed(gpsArray[1][6]));

*/
Serial.print("time ");
    Serial.println(time);
    Serial.print("lat ");
    Serial.println(latitude);
    Serial.print("latH ");
    Serial.println(latHem);
    Serial.print("lon ");
    Serial.println(longitude);
    Serial.print("lonH ");
    Serial.println(lonHem);
    Serial.print("sat ");
    Serial.println(satellite);
    Serial.print("dil ");
    Serial.println( dilution);
    Serial.print("course ");
    Serial.println( courseOG);
    Serial.print("mag ");
    Serial.println(magneticCourse);

    //Serial.println(speedOG);

  }
 // Serial.print("looping");
  Serial.flush();
  numStrings = 0;
}

}
//<--------------FUNCTIONS-------------------->

/*
 *  reverseIntBytes
 *
 *  Changes the byte array from little
 *  endian to big endian
 */

void reverseBytes(int length)
{
  byte temp[2];
  int i, j;
  for(i = 0, j = 1; i<length; i++, j--)
    temp[i] = data[j];
  data = temp;
```

```
}

/*
 *  compassBearing()
 *
 *  Gathers compass bytes and converts to a float
 *
 */

float compassBearing()
{
  byte highByte;
  byte lowByte;

  Wire.beginTransmission(ADDRESS);
  Wire.send(2);
  Wire.endTransmission();

  Wire.requestFrom(ADDRESS, 2);
//If no data is being inputed, return error
//float, 360.00 because this is an impossible
//bearing
  while(Wire.available() < 2)
  {
    //Compass Error LED on
    return 436.90;
  }
//Read high byte then low byte
  highByte = Wire.receive();
  lowByte = Wire.receive();
//Put the 2 bytes together
  float value = ((highByte << 8) + lowByte);
//Return that value/10 to get bearing to tenth
//of degree
  return value/10;
}

/*
 *  convertChannel1
 *
 *  Takes floating point input from received
 *  channel1 command. Converts float to channel1 command.
 *
 */

void convertChannel1(float input)
{
  char buffer[2];
//Multiply input by 127 because 127 is max
//number and input is percentage of this
  int temp = abs(input*127);
//First part of command is always !
```

```
  channel1[0] = '!';
//For channel 1, if input is less than 0,
//then reverse power, a, otherwise,
//forward power-A
  if(input > 0)
    channel1[1] = 'A';
  else
    channel1[1] = 'a';
//Get hex of temp integer, add command array,
//and add carriage return and null terminating
//character
  itoa(temp, buffer, 16);
  channel1[2] = buffer[0];
  channel1[3] = buffer[1];
  channel1[4] = '\r';
  channel1[5] = '\0';
}

/*
 *  convertChannel2
 *
 *  Takes floating point input from received
 *  channel1 command. Converts float to channel2 command.
 *
 */

void convertChannel2(float input)
{
  char buffer[2];
//Multiply input by 127 because 127 is max
//number and input is percentage of this
  int temp = abs(input*127);
//First part of command is always !
  channel2[0] = '!';
//For channel 1, if input is less than 0,
//then reverse power, b, otherwise,
//forward power-B
  if(input > 0)
    channel2[1] = 'B';
  else
    channel2[1] = 'b';
//Get hex of temp integer, add command array,
//and add carriage return and null terminating
//character
  itoa(temp, buffer, 16);
  channel2[2] = buffer[0];
  channel2[3] = buffer[1];
  channel2[4] = '\r';
  channel2[5] = '\0';
}

/*
```

```
 *  convertCompass
 *
 *  Receives floating point bearing and multiplies it by
 *  10 to return integer with decimal number included
 *
 */

int convertCompass(float compass)
{
  return(compass*10);
}

/*
 *  convertTime
 *
 *  Takes input array and converts to a floating
 *  point number
 *
 */

float convertTime(char time[])
{
  int flag = 0;
  char * pEnd;
  int j = 0;
  int i = 0;
  int k = 0;
//Temp is for digits before decimal
//Temp2 is for digits after decimal
  char temp[15];
  char temp2[15];
//Gather data until null terminating character is seen
  while(time[i] != '\0')
  {
//If decimal is seen, null terminate temp, increase k(flag),
//increase i to continue going through input,
//and reset j for use
    if(time[i] == '.')
    {
      temp[j] = '\0';
      i++;
      k++;
      j = 0;
    }
    else
    {
//If k>0, then reading after decimal, add time char
//to temp2 and increse i, j, and k(keeps how many
//digits after decimal are read).
      if(k>0)
      {
        temp2[j] = time[i];
```

```
        k++;
        j++;
        i++;
      }
//Otherwise, add time char to temp and increase j, i.
      else
      {
        temp[j] = time[i];
        j++;
        i++;
      }
    }
  }
//Null terminate temp2
  temp2[j] = '\0';
//For some reason, when the last digit is 0, the
//dision rounds up to have a 1 at the end so this
//creates a marker to know when to subtract .01
  if(temp2[j-1] == '0' && atoi(temp2) > 0)
    flag = 1;
//Get the value after the decimal
  float temporary = atoi(temp2)/(pow(10, k-1));
//Subtract .01 if necessary
  if(flag == 1)
    temporary -= pow(10, -(k-1));
//Add part after decimal to converted array of part
//before decimal
  return (strtol(temp, &pEnd, 10))+temporary;

}

/*
 *  convertToIntPlusOne
 *
 *  Takes and input array (dilution, speed, and true or
 *  magnetic course) and returns an int with one spot after
 *  the decimal included
 *
 */

int convertToIntPlusOne(char input[])
{
  char temp[10];
  int i, j;
  for(i = 0, j = 0; i < 15, j <15; i++, j++)
  {
//If end of array, exit for loop
    if(input[i] == '\0')
      break;
//If read a period, add digit after period to temp array
//and exit loop
    else if(input[i] == '.')
```

```
    {
      temp[j] = input[i+1];
      j++;
      break;
    }
//Add input digit to temp array
    else
      temp[j] = input[i];
  }
//Null termitate array and return the int
  temp[j] = '\0';
  return atoi(temp);
}

/*
 *  convertHemisphere
 *
 *  Takes in a char array and returns the ASCII
 *  value of the first item (should be the only item)
 *
 */

byte convertHemisphere(char hemisphere[])
{
  return hemisphere[0];
}

/*
 *  convertSatellites
 *
 *  Takes a char array and returns integer value of that
 *  array
 *
 */

byte convertSatellites(char pos[])
{
  return atoi(pos);
}

/*
 *  convert_lat_lon
 *
 *  Takes char array (latitude or longitude) and returns the
 *  integer value of it with 5 digits after the decimal
 *  included, returns a long (4 bytes)
 *
 */

long convert_lat_lon(char pos[])
{
  char * pEnd;
```

```
  char temp[11];
  int i, k;
//j holds the number of digits after the decimal that have
//ben read
  int j = 0;
  for(i = 0, k = 0; i < 12, k < 12; i++, k++)
  {
//When null terminating character is reached, if 5 digits after
//the decimal have not be read, add necessary number of 0's to
//array to reach the 5 digits
    if(pos[i] == '\0')
    {
      while(j<6){
        temp[k] = '0';
        j++;
        k++;
      }
      break;
    }
//When period is read, increase j to signal that digits are after
period
//and subtract one from k so that when k is increased in the for loop
it
//does not miss a spot
    else if(pos[i] == '.'){
      j++;
      k--;
    }
//If 5 digits after the period have been read, break out of loop
    else if(j == 6)
      break;
//Add position digit to temp array and if reading from after the
//decimal, increase j
    else
    {
      temp[k] = pos[i];
      if(j>0)
        j++;
    }
  }
//Null terminate array and convert array to a long
  temp[k] = '\0';
  return strtol(temp, &pEnd, 10);
}
```