A.R.M.O.R.D

Armored Remote Monitoring and Operated Recon Device

Senior Design Group 9

May 5, 2010

Andrew Lichenstein
Kevin Jadunandan
Thomas Kehr

Table of Contents

List of Figures

List of Tables

1.0.0 Executive Summary

This summary will touch on the main points covered in this design document. We will first discuss the purpose behind this project, followed by a brief explanation of the goals and specifications we hope to achieve. We will then move on to provide a high-level overview of our research and design process. Finally, we will conclude by explaining our testing procedure and provide some final thoughts on our system.

Our purpose in undertaking this robot was to see how far we could push our knowledge and abilities in the robotic domain. While our robot will have many practical applications in real world situations, our main goal in building it was to prove to ourselves that we could undertake the task at a fraction of the market value of similar robotic systems. Much of the design inspiration for this robot was based on a similar robot designed by Carnegie Mellon University, the Dragon Runner.

We will demonstrate several advanced capabilities with our robotic system. The first is the ability to remotely control the system by means of an Apple iPhone portable device over a specified range. While controlling the robot, it will provide us with real-time video and audio feedback from the use of an on-board camera and microphone. The most impressive aspect of our robotic system is its ability to sustain severe trauma and stress without damaging vital electronics and systems. Our system is able to survive being thrown through windows, being dropped from three story buildings, and even being thrown from a moving vehicle. We have also added several secondary features, such as GPS tracking and limited ballistics resistance.

All aspects of our project were researched heavily before any design or construction began in an effort to be as efficient as possible with funding. We looked at several robot case studies and paralleled their research and design process. From these studies, we found that it was best to assign each member in the group to a certain developmental task: programming, microelectronics, and drive train/chassis. Over the course of several months, each member researched all aspects associated with their task. Key topics for research included Apple software development, wireless communication standards, and in-depth study of raw materials and their durability.

The final design of our robotic system satisfies all of our initial requirements. The durability of our robot system is attributed to both a highly durable shell, constructed from high impact polymers, and an extremely well structured chassis and suspension system that allows for proper dampening of inertial forces. The system utilizes a custom hybrid iPhone interface to both control the system’s movement as well as display video from the onboard camera. Power is supplied by two, fully rechargeable, batteries that allow the robot to operate continuously for up to half an hour.

We plan to evaluate our robot's capabilities through a number of system tests. The robot will be subjected to a communication test that involves driving our robot to a distance until it becomes un-responsive. We will also conduct several impact tests, in which we will utilize G-force sensors to evaluate against known failure values from dropping the robot from a given elevation. There are also plans to test speed, maneuverability, cooling, and user friendliness of the robot

Our final system proves to be an exemplary example of the current state of human-robot interaction that is currently taking place in the military domain. In the future we hope to regroup and develop our system further, by adding several aspects that were both out of our budget and time constraints. Overall, the project proved to be successful in that we achieved a fully functional robotic system while also gaining valuable experience and knowledge.

2.0.0 Introduction

"The robot revolution is upon us."

-PW Singer, author of Wired for War: The Robotics Revolution and Conflict in the 21st Century, in discussing the current state of robotics in the armed forces.

Robots are currently being fielded by all branches of the armed forces, not just in the US, but all over the world. The United States initially went into Iraq having only a handful of aerial drones and zero unmanned ground units; they have since fielded approximately 5,300 drones and over 12,000 unmanned ground based robots. Professionals in the industry estimate that in 25 years we will have tens of thousands unmanned systems being fielded with over one-billion times the capabilities as today. This surge in unmanned system has raised many paradigms in the current state of warfare. An example of such a paradigm is that unlike other military hardware, robotics doesn't require a massive manufacturing system to accomplish, as most of robotics in the military is off-the-shelf technology; in fact a simple UAV drone can be built for fewer than one-thousand dollars, given the right equipment.

It is with this "Do-It-Yourself" attitude of robotics that we were inspired to create a robot for our senior design project. From our groups' experience in the defense industry we decided to create a robot for military applications. We knew that our group possessed the skills needed to build one, as we were skilled in both electrical and computer engineering, as well as the basics of mechanical engineering. We also had a good deal of knowledge in electrical components and programming. Several members of our group were also enrolled in a robotics course, which worked to fuel our interest in the topic even more.

It was in this robotics class that we finally found our inspiration for a military robotic platform. During class one day we watched a compilation of videos regarding current robotic systems being used by the military. The unit that caught our eye was known as the Dragon Runner, a system developed for the Marines by Carnegie Mellon University, as seen in Figure . The system serves as a sentry and monitoring device and has even been used in IED (Improvised Explosive Device) disposal. What makes this system so unique is that it is almost impervious to physical abuse. A video that we watched in class illustrated this by showing several clips of the unit being thrown through glass windows, being dropped from second story balconies, and even thrown from a moving vehicle. In each case, the robot kept moving immediately after, as if nothing had happened. We found this to be both hilarious and utterly amazing. We quickly set forth to learn how the developers achieved such an engineering feat. While researching certain dynamics of the unit, we also happened to come across the Dragon Runner's price tag: a mere $32,000. This was an immediate shock to us, as the unit was no more than two feet long by one foot wide. Thus, a second dynamic was added to our project: to be able to create the same system as Carnegie Mellon, for a fraction of the price.

Figure - Dragon Runner
(Figure printed with permission by Wikipedia GNU)

2.1.0 Current Research

The area of law enforcement and military robotics is starting to become a large industry with no current plans of slowing down. The reason for this is that military units are trying to save as many lives as possible by sending robots in a dangerous area instead of sending a human. This is best illustrated by a quote from a U.S. Navy petty officer, “When a robot dies, you don’t have to write a letter to his mother.” I’m hoping one day that there will be more robotic soldiers than human equivalents, so that lives may be saved. Currently, there are many robotics projects being developed that are trying to achieve this task.

The Dragon Runner, our inspiration for this project, is currently one of the top sentry robots in production. This urban combat robot only weights around 9 lbs and can be carried on soldiers back. It was originally developed at Carnegie Melon University and funded by the Marine Corps Warfighting Lab. An additional image of this vehicle can be seen in Figure . This robot is now over in Iraq helping soldiers in everyday situations. Our robot parallels many features of the Dragon Runner; however, because of funding they are able to have more upgraded features then ours, such as a top speed of 45 miles per hour and thermal imaging capabilities. Carnegie Mellon has spent over $32,000 dollars in the building of a single Dragon Runner system, whereas our robot will come out to be less than $1,000 dollars.

Figure - Top View of Dragon Runner Unmanned Vehicle
(Image reproduced from the Public Domain through the Marine Corp.)

Another robot that compares to our robot is the Battlefield Extraction-Assist Robot, or B.E.A.R as it is commonly referred to. This robot holds true to its name, as it is able to carry up to 500 lbs, which is more than enough for a human and his gear. This robot compares to our robot in that it is highly dynamic on how it moves. This is a great robotic invention and is currently being designed to help downed soldiers in unsafe areas by picking them up and bringing them to a safe zone. They engineered this robot to be able to traverse battlefields, mine shafts, areas near toxic chemicals, and even inside destroyed buildings. I believe this is the next evolution of the Dragon Runner, due to its ability to save human lives through its autonomy.

Finally, we look at a robot that we have been researching in our robotics course, “Big Dog”, seen in Figure . This amazing quadruped robot is able to traverse on various surfaces, such as snow, broken debris, and sand. Big Dog has a balancing mechanism so that if it is kicked or pushed it can balance itself within seconds. This robot illustrates the ability to reproduce biological behaviors in a mechanical form; it’s really very creepy.

Figure - Big Dog Robotic Quadruped Built by Boston Dynamics.
(Figure printed with permission by Wikipedia GNU)

2.2.0 Goals and Objectives

The goal for this project was to create an extremely durable, user controlled, and low cost robot for military and law enforcement applications. The price on the real robot was in the range of $30,000 dollars, so our goal was to produce one for under $1,000 dollars. We have broken up our main goals into hardware and software categories.

We plan on controlling our robot long distances through the use of wireless radio communication. We will use socket connections to send packets back and forth between the robot and the central control unit. We also want to be able to see and hear what is going on with our robot from a distance, so we will use a camera with an audio/video transmitter. We will install a global positioning unit module onto our robot to track its' location at all times in case of communication failure. We will have two main power supplies, one will control the motors and the other will control the remaining electrical components, including microcontroller, camera, wireless communication, GPS, and fans. We do this to insure the high current of the motors does not interfere with any of our sensitive and expensive electrical components. We will use rechargeable batteries and install plugs on the side of the robot to facilitate easy charging. The locomotion of our robot will be controlled using two high-powered DC geared motors connected to wheels connected directly to motor shafts. We will use four all terrain wheels on our robot, to get the best possible traction on any surface we drive our robot. We will have a motor controller connected to our microcontroller that controls the motors via pulse width modulation. The enclosure of the robot will be fabricated by us using materials such as fiberglass, carbon fiber, and truck bed liner. The robots enclosure will be able to withstand large amounts of force from falls or collisions. The robots frame will be constructed of aluminum, which will hold all of our electronic components and peripherals. The frame and electronics will be supported by a suspension system in order absorb forces of a collision or fall.

The robot will constantly stream audio and video wirelessly to the user controlling the robot from a custom software application. We will show current GPS location in our software using Google Maps or another variant. The software portion of the robot will be controlled primarily by a user interface through a computer application. In addition to this, an Apple iPhone will connect to the application and become the primary controller for the robot system. We will create a wireless gateway that will allow data to be sent from the robot, to the gateway, and finally to the user. Our gateway will consist of either a laptop or desktop computer that will be running Windows XP or Linux. It will be able to control our robot and be able to allow the iPhone to connect and take over control. The iPhone application will be able to move the robot and view the video feed.

Since we have a lot of work ahead of us we decided to create a Gantt chart that will help us in setting our goals within our time limitations. We will start this project by first researching every section of our robot thoroughly. We will then move on to purchasing our parts, based off what we researched. When our parts come in we will go straight into design mode, starting with the electronics components, followed by the drive train, and then finish the outside body of our robot. The next step in our process would be to program our software interface for the gateway and iPhone application. Then we will test our robot vigorously, making sure we follow our requirements section to insure we deliver a fully, completed robot. Finally, we will do hardware and software cleanup, which will be done to make sure our code is at the highest efficiency possible and that we have all the wires and components connected securely.

2.3.0 Specifications

After researching ideas for our design, we decided on some specifications for the capabilities and physical properties of our robot.

Table below lists the major specifications for our robot and shows the acceptable (low) and improved (high) possibilities for each specification. There's actually more specifications than these presented here but we chose to list only the most necessary for our project.

Specifications Table

		Low	**High**	**Units**
Body				
	Length	1.5	2.5	Feet
	Width	0.5	1.5	Feet
	Height	0.5	1.0	Feet
	Weight	5	10	Lbs.
Power				
	Battery Voltage	9	15	Volts
	Battery Life-High Usage	0.5	1	Hours
	Battery Life-Low Usage	1	4	Hours
Locomotion				
	Peak Speed	5	25	MPH
	Wheel Size	5	10	Inches
	Motor RPM	900	9000	RPM
	Turn Radius	0	5	Degrees
Communication				
	Radio Frequency	2.4	5.8	GHz
	Video Frame Rate	10	40	FPS
	Camera	160 X 320	480 X 640	Pixels

	Microphone Distance	5	25	Feet

Table - Specifications
(Table created by our group)

2.4.0 Requirements

In this section we will go over the requirements for our robot. We made sure to test all necessary features of our robot, by selecting good requirements. We have more tests we wish to complete; however, these are the most important requirements. We have created some detailed test procedures for some of the more important requirements, which is are section 4 of our documentation. In Table , we show each of our requirements and how we are going to test them. In the last stages of our project we will make sure to use these requirements to test our robot to insure that we are completing all important requirements to deliver a complete working system.

Requirement Table		
No.	**Requirement**	**How To Test Requirement**
1	ARMORD must have a minimum communication range of approximately 100 ft.	We will drive the robot until communication failure and then measure distance.
2	ARMORD must have a minimum peak speed of 15 MPH.	We will mark an initial area and then time the robot until it reaches the final mark.
3	ARMORD must be able to withstand a 2-story drop and equivalent impact and still be able to function normally.	After the robot is dropped we will continue to use and watch performance of the robot.
4	ARMORD must be able to provide video/audio	We will check the live feed of our

	feedback via the onboard camera.	robot as it drives.
5	ARMORD must have a battery life of 1/2 hour while mobile, 3 hours while operating in standby.	We will time the robot when the robot is running motors until it dies and then we will time it when it is stationary.
6	ARMORD must be able to navigate multiple types of terrain specifically pavement, grass and sand.	We will drive the robot over different terrains to ensure that the robot can move efficiently.
7	ARMORD must be able to be carried easily in a book bag by an average size human.	We will try and store in normal book bags that we wear normally on campus.
8	ARMORD must weigh less than 15 lbs.	We will put the robot on a scale and measure it's weight.
9	ARMORD must be able to reverse motor directions within 5 second.	We will start the robot in one direction and immediately switch the direction of the motors and watch what happens.
10	ARMORD must have zero degree turn radius.	We will make one motor drive one way and make one motor go the opposite way and watch what happens.
11	ARMORD's video stream must have less than 5 seconds delay between robot and user.	We will put a clock in front of the robot and one in front of us both with the same time and check the difference.
12	ARMORD's must have user interface that is	We will get someone completely who has not touched our project

	simple to use.	to interact with the user interface and then fill out a form explaining their experience.
13	ARMORD's must have an accurate way of being tracked via the GPS.	We will set the location of our robot in many different locations throughout Orlando and checking the corresponding GPS location.
14	ARMORD must have minimum packet loss.	We will set 10 different sets of data multiple times and check the output to make sure we are receiving all packets.
15	ARMORD must maintain a temperature low enough that no electronic equipment will fail.	We will set up thermal readers on the inside of the device and monitor while the robot is running.
16	ARMORD must be able to continue movement if it is flipped.	We will test this by driving it over and object and letting it drive on the opposite side.
17	ARMORD must be able to work in areas that are surrounded by 2.4GHz.	We will bring our robot to UCF and drive it around to insure no interference.
18	ARMORD must have a cold start time of under a minute.	We will let the system sit for a few hours and then turn it on and time it to make sure it is within time limits.

Table - Requirments Table
(Table created by our group)

2.5.0 Risks

Every project comes with risks; due to us all being full time student and hold demanding internships, we probably have more risks than most groups. One of the risks that separated us as a team from most groups is that we have only three people in our group and probably have a lot more design work than most other projects, even though we knew what we were getting into when we picked this project. Another risk is that we have never actually ever built an entire robot before; this will be a new experience to us all. There are also the risks of getting sick, which we can't control. Illness would set us back in our project easily a week or two. In, this event other team members will have to step up and pick up the slack.

2.6.0 Team Management

We decided from the beginning of this project we would not have a set leader, but would all take equal responsibility in the organization and completion of the project. We all knew what had to be done in order to complete our project and accompanying documentation. We scheduled weekly meetings in order to discuss updates with each section being worked on; this insured we were all on the same page with the project. For our documentation we had a configuration management approach set up that allowed one team member to keep the main document at all times and have the other team members send updates to that team member. This insured we didn't accidently delete anyone's writing. This also made sure we frequently communicated with each other.

Another way that we managed the project was by dividing it up into smaller pieces that each person could individually manage. Figure illustrates how the software and computer hardware was divided, and Figure has the electronic and chassis construction breakdown of the project. When splitting up the project, we looked at who had experience in certain systems and what people's interests were. We knew that by allowing people to work on segments that they enjoyed more than they would be more enthusiastic with the project, which would result in a better system design and project outcome.

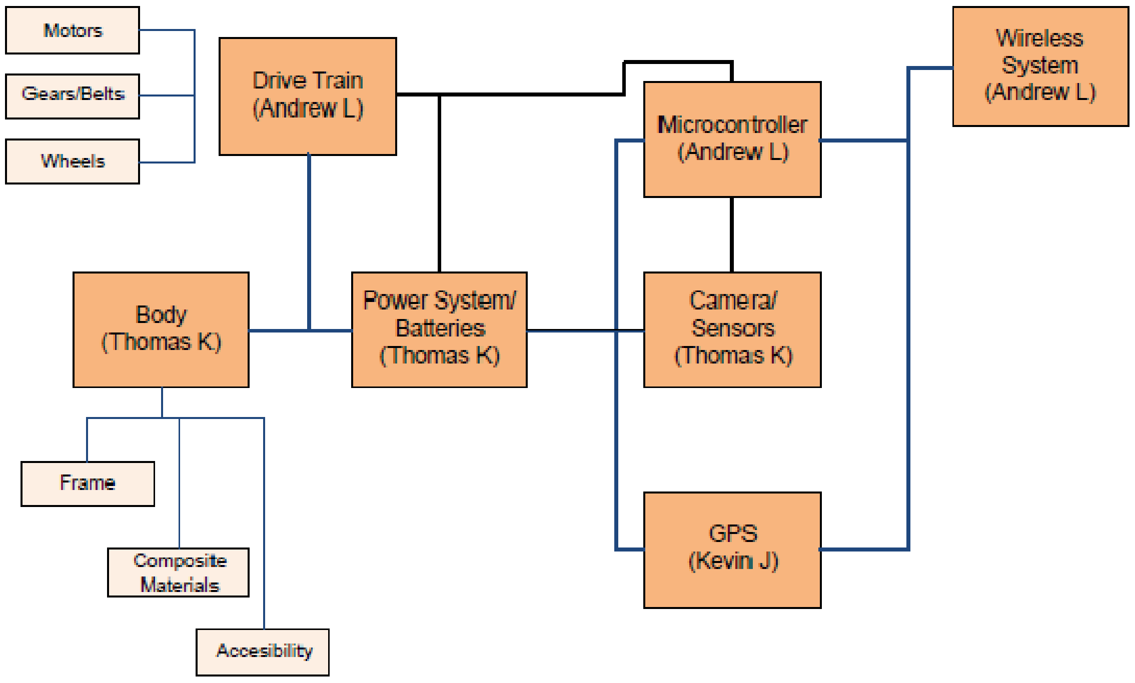

Motors
Gears/Belts
Wheels
Drive Train (Andrew L)
Microcontroller (Andrew L)
Wireless System (Andrew L)
Body (Thomas K)
Power System/ Batteries (Thomas K)
Camera/ Sensors (Thomas K)
Frame
Composite Materials
Accesibility
GPS (Kevin J)

Figure - Hardware Breakdown of Project
(Figure created by our group)

As you can see from the diagrams, we essentially broke down the project into software, electronics and the body design. We had three people in our group, and each had experience in one of those fields so the assignment of each portion was trivial.

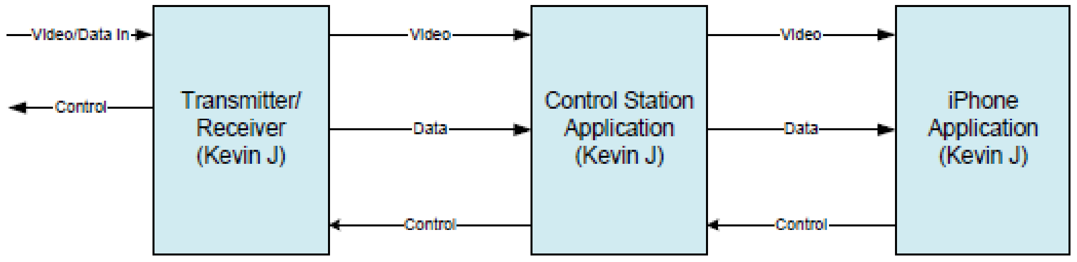

Video/Data In
Control
Transmitter/ Receiver (Kevin J)
Video
Data
Control
Control Station Application (Kevin J)
Video
Data
Control
iPhone Application (Kevin J)

Figure - Software and computer hardware breakdown of our project
(Figure created by our group)

3.0.0 Research

This section of our paper goes into detail of each of the subsections of our robot. We used all resources available to us, including the Internet, books, research papers, and technical journals. We made sure to get permission before we used their information or data in any way. We have our works cited section in Appendix A and our permissions in Appendix B.

3.1.0 Microcontroller

The microchip we used for our project will be the brain of our robot. It will control all the features of our robot and will be constantly sending and receiving data. When it receives data from our wireless communication it will need to be able to take that data and parse it, to be able to send it to the different electronic components. The microcontroller will also need to take data in from the electronic components, such as the GPS unit. Figure below shows a brief overview of how the robot will use the microchip.

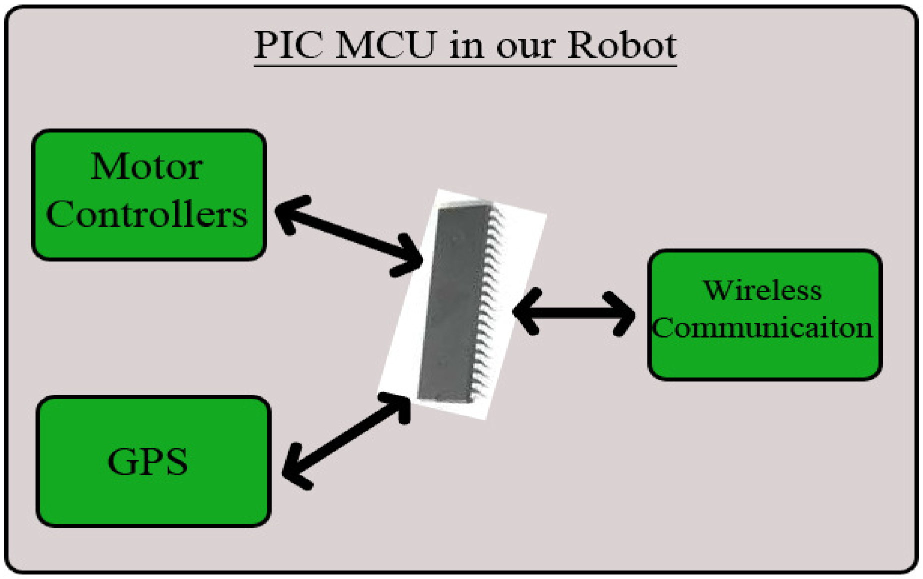


Figure - Microcontroller in our Robot
(Figure created by our group)

We will be heavily researching the high-end PIC devices, such as the PIC18 and PIC24 chips, to see which best suits our project. The PIC18 is an 8bit microcontroller and the PIC24 is a 16bit microcontroller, where functionality and performance increase from the PIC18 to the PIC24. We will come to a final decision based off instructions sets, data space, and code space. After reviewing the different types of packages that the chips comes in, we have decided to go with Plastic Dual Inline Packaging (PDIP), because it is easy to get to the pins when using a breadboard and will be easy to solder when we build our final PCB. Table below shows a few of the PIC chips that we have researched so far.

We will be able to program these chips very easily with the MPLAB IDE that Microchip gives you to use. It allows you to program in C++ and it will assemble it to the machine language of the microchip. This is extremely helpful, because assembly code is hard to code and takes a very long time if you are not experienced like us.

PIC Comparison Table

PIC Name	Program Memory	CPU Speed	RAM Bytes	Data EEPROM Bytes	Operating Voltage Range
PIC18F2553	32KB	12MIPS	2048	256	2 to 5.5 Volts
PIC18F2680	64KB	10MIPS	3328	1024	2 to 5.5 Volts
PIC18F4520	32KB	10MIPS	1536	256	2 to 5.5 Volts
PIC18F4550	32KB	12MIPS	2048	256	2 to 5.5 Volts
PIC18F24J11	16 KB	12MIPS	3800	N/A	2V - 3.6V
PIC18F24J50	16 KB	12MIPS	3800	N/A	2V - 3.6V
PIC18F25J11	32 KB	12MIPS	3800	N/A	2V - 3.6V
PIC18F25J50	32 KB	12MIPS	3800	N/A	2V - 3.6V
PIC18F26J11	64 KB	12MIPS	3800	N/A	2V - 3.6V
PIC18F26J50	64 KB	12MIPS	3800	N/A	2V - 3.6V
PIC24F08KA101	8 KB	16MIPS	1536	512	1.8V - 3.6V
PIC24F16KA101	16 KB	16MIPS	1536	512	1.8V - 3.6V

Table - Possible PIC Microchips
(Table created by our group)

The chips listed above have many of the same features that we need, such as low operating voltage and CPU speed. The top five microchips only have 1 USART (Universal Synchronous Asynchronous Receiver/Transmitter) module and the bottom have 2 USART modules. Depending on how we use our wireless communication and our GPS unit, this will depend on what chip we use. So far from what I have read from other people on various forums is that it is better to have two USART modules in the microchip to allow two lines of input and output from continuously running. The USART is what is going to allow the structures we send in our code to be transmitted as individual bits in a sequential order, so that it can be easily decoded. When it reaches the destination, the other side reassembles the bits into complete bytes allowing us to be able to get our structure of data.

Another set of microchips that we may be interested in purchasing is the Atmel AVR microchips. The AVR is a modified Harvard architecture, 8-bit single chip microcontroller. We will also be looking at the DIP modules of the AVR chips seeing which are the easiest to work with on a breadboard. Both the prices and packaging types are similar in the AVR and PIC microchips. I have a table that shows the AVR chips I have been looking at; all of these chips have either two or three USART communications terminals.

AVR Comparison Table

AVR Name	Program Memory	CPU Speed	RAM Bytes	Data EEPROM Bytes	Operating Voltage Range
ATxmega16D4	16KB	32MIPS	2048	1024	1.6V – 3.6V
ATxmega192D3	192KB	32MIPS	16K	4096	1.6V – 3.6V
ATxmega256D3	256KB	32MIPS	16K	4096	1.6V – 3.6V
ATxmega32D4	32KB	32MIPS	4096	1024	1.6V – 3.6V
ATxmega64D3	64KB	32MIPS	4096	2048	1.6V – 3.6V
ATxmega64D4	64KB	32MIPS	4096	2048	1.6V – 3.6V

Table - Possible AVR Microchips
(Table created by our group)

As you can see from Table , the AVR microcontrollers have a lot more program memory than the PIC microchips. One of the key features these chips have that we really like is the fact that they have more than two USART ports.

I have been constantly talking about the USART ports on the microchips, but haven't told you why they are so crucial. Universal Synchronous –Asynchronous Receiver Transmitter takes bytes of data and transmits the individual bits in a sequential fashion. When the data reaches the other side, a USART reassembles the bits back into complete bytes. The way this works is that after it sends all the bits needed, it will then send an additional parity bit, which is used to check for errors, and a stop bit which tells it when to stop communication.

3.2.0 Motor Controllers

In order to correctly drive our vehicle with motors, we need to use a motor controller. There are two types of motor controllers ASD (Adjustable Speed Drive) and VSD (Variable Speed Drive), for our project we will be using ASD because it has acceleration control, smoother operation, accurate positioning and control for the torque. As we all know, the way a motor control works is with an H-Bridge, which is made up of four PNP BJTs or P-Channel MOSFETs. The H-Bridge allows the motor to brake because it shorts the motors terminals; here are the basic states to and H-Bridge.

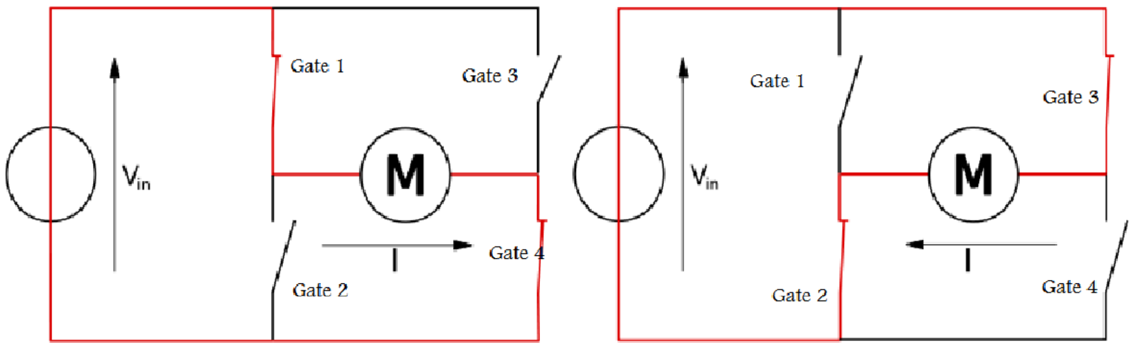


Figure - H-Bridge Current Flow
(Figure printed with permission by Wikipedia GNU)

Figure above show's how the motor controller works, a gate is a MOSFET/BJT device and the M is a motor. When gates 1 and 4 are closed and gates 2 and 3 are open, the motor (M) gets a positive current through it, making the motor turns clockwise, which is shown by the red path. When gates 2 and 3 are closed and gates 1 and 4 are open the motor (M) gets a negative current which turns the motor counter clockwise. The MOSFET/BJT is what flips the current through the motors, due to the open and closed states of the others.

We had the option of either buying our own motor controller or building one from scratch, we decided to buy one which would save us time. We looked for a motor controller that could take enough amperage for two motors, making it easier for our overall system.

3.3.0 Motors

From the group's previous experience with robots, we knew we needed a DC geared motor to move our robot effectively. By having a geared motor we would be able to achieve sufficient speed without sacrificing the torque needed to negotiate rough and uneven terrain. The gear ratio refers to the number of teeth on the input gear versus the output gear and is very important when selecting a geared DC motor. In general, the higher your gear ratio is then the more torque you receive; the lower the gear ratio, the more speed you will receive. It was also important to choose a motor that provided the greatest efficiency within our budget.

As our research showed, a typical DC motor is only between 40 and 75 percent efficient and adding gear ratios reduces this further due to the energy lost at each contact point of the gears. Table shows the average gearbox efficiency versus several gear ratios. Based on this table we can estimate that a DC motor operating at 50% efficiency that utilizes a 100:1 gearbox will have an actual efficiency of about 33%.

6:1	81%
30:1	73%
75:1	66%
100:1	66%
180:1	59%
300:1	59%
500:1	59%
800:1	53%
1000:1	53%
3000:1	48%

Table - Gear Box Efficiency Versus Gear Ratio
(Table created by our group)

From our research we have also found that it is possible to approximate the speed, in miles per hour, you will achieve from a motor, based on the wheel diameter and the rotations per minute of the motor. This equation can be seen in the calculation below. Based off this equation we can conclude that to get our desired speed of 20 miles per hour with a robot utilizing 5-inch diameter wheels, we will need a motor having a speed of at least 1344 rpm.

$$\text{Speed (mph)} = (\text{Diameter of wheel (in)} \times \pi \times \text{rpm of motor} / 60) \times .0568$$

Supply voltage was another important factor we considered when selecting motors. We found that typical supply voltages for robots were 6, 12, and 24 volts. While we will discuss power to the motors later in the power supply section of this document, it is worth noting here that larger supply voltages generally allow for greater motor speeds at the cost of more expensive battery packs and power supply equipment.

In our search for DC motors, we also came across another type of motor that has recently emerged as a popular choice among RC enthusiasts. This type of motor is known as a DC brushless motor and differs from a normal (brushed) DC motor in a number of ways. Unlike a conventional DC motor where the brushes make contact with the commutator, forming an electrical circuit between the DC source and armature-coil causing the armature to rotate on its axis, the brushless DC motor (BLDC) utilizes stationary electromagnets, or stators, with rotating permanent magnets and a static armature. Figure shows the cross-sectional differences between the two motor types. The power distribution in a BLDC is controlled by an electronic control rather than the commutator/brush system of a brushed DC motor.

In comparison of a traditional DC motor, the BLDC motor offers many advantages including higher efficiency, reduced EM interference, increased lifespan due to no brush erosion, increased power, and the elimination of ionizing sparks from the commutator. BLDC motors are also advantageous because they require no internal airflow due to the fact that electromagnets are attached to the motors casing, and are thus cooled by conduction. This also means that the motor can be completely sealed to be protected from the environment. The main drawback of BLDC technology is its relatively high cost, which currently prevents them from being implemented everywhere that brushed DC motors are currently being used. The increase in price is due to two factors. The first is that BLDC motors require complex and expensive electronic speed controllers to run. The second is that because BLDC motors have still not gained a lot of popularity in the commercial sector, the motors are often hand-wound, whereas brushed motors armature coils are inexpensively machine wound. From our research we have found BLDC motors to be as much as three times the price of a traditional brushed DC motor, with the electronic speed controllers (ESC) to be upwards of $150 dollars.

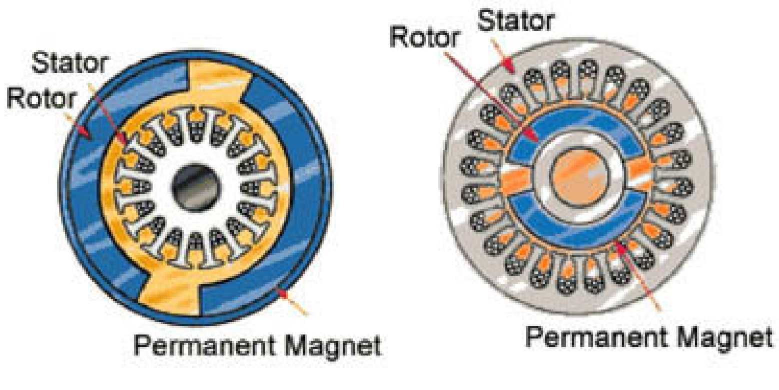


Figure - Burshed DC Motor (left) and Brushelss DC Motor (right)
(Figure printed with permission by Wikipedia GNU)

3.4.0 Power Management System

We knew that our project was going to be using several electronic components, all requiring different amounts of voltage. To be able to obtain these specific voltages we figured we would have to implement a number of different valued voltage regulators that were fed from the batteries. Our group had a limited understanding of this technology, so research was done in regards to the operation of a voltage divider.

Luckily, the theory behind voltage regulator circuits is relatively easy to understand. The only decision we needed to make regarding the regulators was whether we were going to build them ourselves or to purchase pre-made voltage regulator solutions. For purposes of adding design elements to our project, we chose in favor of constructing the regulators ourselves.

From our research we learned that the only components needed for voltage regulation construction were two electrolytic capacitors, PCB boards, and oddly a voltage regulator. The capacitors in the voltage regulation circuit are necessary to filter the ripple from the voltage regulator and to balance the load to ensure a constant output voltage. In order to achieve the different value for constant voltage output, we will also need several variations in voltage divider transformers. Figure illustrates the circuit layout we plan to implement to obtain our voltage regulation.

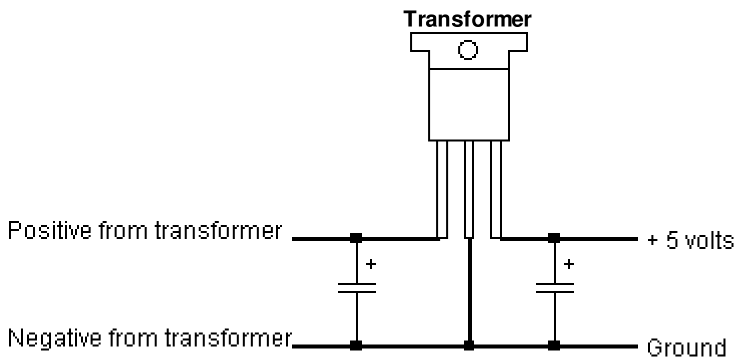


Figure - Voltage Regulator Design for the ARMORD System
(Figure created by our group)

3.5.0 Wireless Communication

We will need wireless communication to talk in between the host computer/iPhone and our robot. The first thought was going to use 802.11b/g/n, which has great speed and great distance, but due to their being so many different 802.11 devices always around we were worried about interference between our host and the robot. After reviewing forums and different websites we saw a lot of people using either Zigbee or EnOcean for wireless communication.

After carefully reviewing what both technologies have to offer, we discovered the main differences between Zigbee and EnOcean are that Zigbee uses a 2.4GHz band while EnOcean uses 315MHz/868MHz. Another difference is that EnOcean doesn't need a battery to power it; instead it uses heat dissipated by other electronics around it. EnOcean says that 2.4GHz is overused and that it may cause interference with your device, we believe that we won't have this issue. Even though the Zigbee has two setbacks compared to the EnOcean product, Zigbee has been widely used and there is more information, projects, and source code for Zigbee online, which will be very helpful to us, seeing we have never used Zigbee before. Figure is a diagram of how we are going to use the Zigbee module.

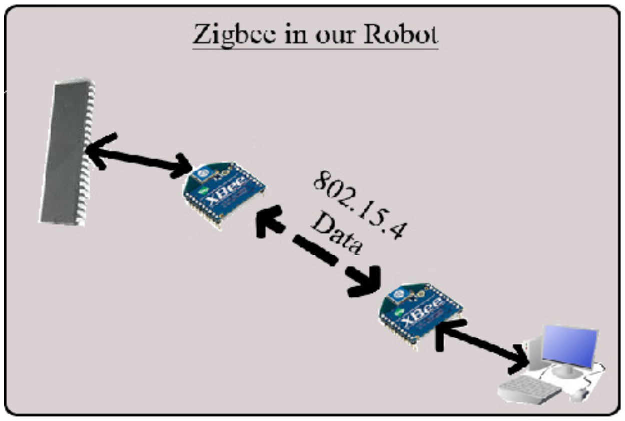


Figure - Zigbee in our Robot
(Figure created by our group)

Many different companies take the Zigbee standard and use it to make their own modules. In our case we will be looking at a company named Digi for our Zigbee modules, because they are a high quality distributor. They have two main types of modules that our team will be looking at: the XBee and XBee Pro. Some of the major features that are good for our project are that it's a small form factor, it has a long battery life, provides for high data reliability, and the fact that the product is easy to replace in case of failure. Table shows the differences between the XBee and the XBee Pro.

XBee Comparison Chart		
	XBee	**XBee Pro**
Indoor Range	133 ft	300 ft
Outdoor Range	400 ft	1 mi
Transmit Power	1.25 mW	50 mW
RF Data Rate	250 Kbps	250 Kbps
Supply Voltage	2.1-3.6 VDC	3.0-3.4 VDC

Transmit Current	35 mA	294 mA
Receiver Current	38 mA	45 mA
Power Down Current	< l uA	< 10 uA
FCC Approved	YES	YES

Table - XBee Comparison Chart
(Table created by our group)

From the table above, you can see the XBee Pro gets about three times more distance compared to the XBee, with the drawback of needing more power and more current. Even though the XBee Pro needs more voltage and current, we believe it will be better for us because of the communication distance. These modules can run for a very long time even if it transmitting and receiving constantly because of low power requirements.

3.6.0 Chassis

When we first conceived the idea for this robot, we knew that construction of the chassis would be one of the most daunting yet important tasks. The strength and durability of the chassis is the only line of defense between extreme physical forces and our expensive electronics. The research presented in this section represents all aspects of chassis construction, including the drive train and the mounting of electrical components.

3.6.1 Robot Platform

Going into this project we were almost certain a custom built chassis was going to be required due to the level of durability needed; however, as a precaution we researched several robotic platform solutions. The most promising of these platforms were a set of all-terrain robot kits from SuperDroid Robotics Inc, the Standard ATR (Figure). This kit stood out to us because of its “dual-sided” design that we had decided upon, as well as its seemingly solid construction of 1/8” aluminum. The drawbacks to this platform were its price point and mounting area for electronics, as this platform was designed to mount electronics and sensors on the exterior of the robot (Figure). While we could tailor this robot to suit our needs, the extra expenditures plus the base price of $600 for the robot did not seem practical and was dismissed as an option.

H x W x L: 6.75” x 12.25” x 17.25”

Figure - SuperDroid Standard ATR
(Figure printed with permission by SuperDroid)

Figure - Intended Mounting of Peripherals to the ATR
(Figure printed with permission by SuperDroid)

In continuing our search for a promising robotic platform we looked to an unlikely source: our childhood. In the early 1990’s Kenner, a popular toy company of the time released a remarkably popular children’s RC car known as Ricochet (Figure) that featured the same “dual-sided” we wanted. While this platform contained almost certainly antiquated electronics and a flimsy plastic enclosure, the design was analogous to what we had imagined. If nothing else we figured we could at least perform preliminary testing with it after some slight modifications. After some online searching, we were able to find several of these cars for under $40.

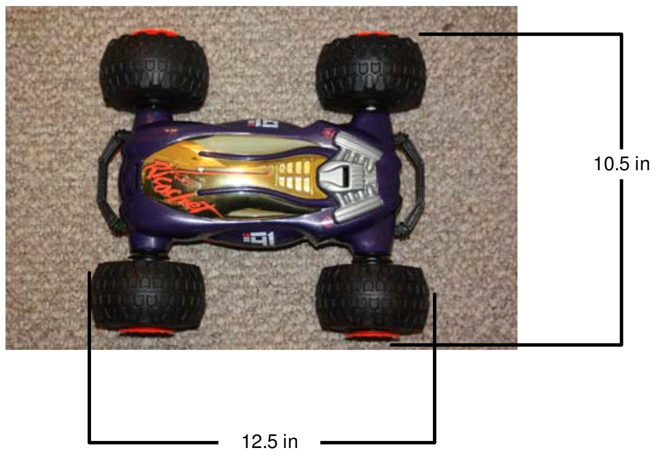


Figure - Kenner Ricochet
(Figure was created by our group)

From these findings we agreed that it was best to stick with our initial plan to construct a custom chassis to suit our needs; Although we did agree to buy at least one Ricochet RC car to aid in tests and to be used as a possible construction aid for our final design.

3.6.2 Drive Train

The method of locomotion also had a crucial role in this project. Not only did we need a reliable system for moving our robot in a harsh environment, but we also needed it to stand up to the abuse it would receive from being thrown and dropped. Based on our best guess, the wheels and axles would take as much, if not more abuse than the robots enclosure. The first thing that had to be decided was the method of locomotion by which to have our vehicle travel. The two options we took under consideration were tank treads and wheels.

After researching drive trains, tank treads seemed to be the ideal way to make sure your robot has enough torque. This is a reason why the military uses treads on their tanks on the battlefield, because they need to move heavy equipment quickly. Tank tread are a large outer shell with teeth like indents on the outer shell, which dig into the ground and help push the vehicle. The main drawback from treads is that when they make a turn, the treads skid, causing wasted energy.

The robot from which we got our inspiration, the Dragon Runner, used a set of all terrain wheels to do its moving, which we decided was the best option for our robot as well. Our reasoning behind this was that we did not want the tank treads slipping off when our robot underwent severe trauma such as large falls or high-speed collisions. Once we decided wheels were the best method of locomotion for our robot, we needed to find a good set of all-terrain wheels and tires that provided a good balance between durability and cost.

We also had to choose the best method for connecting the motors to the wheels. The most common methods we encountered were belt driven systems and direct coupling of motors to wheels. Research showed that similar robotic platforms used the latter method because it provided a more solid connection, whereas a belt drive could come loose upon impact. The one benefit of a belt-driven system is the ability to allow for a significant amount of “give” and resilience to vibrations. Figure shows a common motor and wheel coupling in a similar robotic system to ours, with three sets of wheels. This will illustrate our intended approach to motor placement.

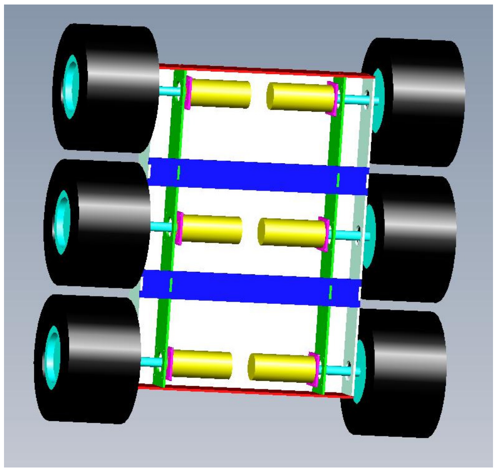


Figure - Common Drive System in a Typical All-Terrain Robotic Platform
(Figure printed with permission by Wikipedia GNU)

3.6.3 Wheels

After researching wheels, we found that most suppliers sell tires and wheels separately, thus driving up the cost. We also found that most off-road robot wheels were actually just RC car wheels being sold by robot store retailers. With this information we were able to expand our search for wheels to other retailers dealing in RC cars in hopes of finding a cheaper solution and maybe a tire and wheel combo pack. In our research we found there were many styles of off-road tires designed for different terrains. A few of these styles can be seen in Figure .

Another factor we had to consider when choosing an appropriate wheel was size. From what we found, most RC wheels ranged between 3 to 5 inches in outer diameter. For the purposes of our robot, we decided it was best to find a wheel with at least a 5-inch diameter in order to provide us with the maximum amount of space to build our robot; remember, our robots dual-sided design means that we have equal tire availability on both the top and bottom of the robot. If we allowed at least 1-inch of tire height for each side of the robot, we would be left with about 3-inches for the body and frame. Our research also indicated that wheel circumference (π * wheel diameter) correlates directly with speed and torque; the smaller the wheel, the greater the torque, whereas the larger the wheel, the greater the speed. Thus it was a balancing act to choose a wheel diameter that was perfectly sized for our robot.

Figure - Common Types of All-Terrain Tire Treads
(Figure printed with permissions by Superdroid)

3.6.4 Raw Materials

When selecting raw materials to construct the chassis and its components, the three factors we wanted to really take under consideration were durability, price, and weight, in that order. The robot design that we based our idea on is constructed out of a number of proprietary and military specific materials that are out of reach for our budget; however, we feel we can achieve similar results with common construction materials.

The outer enclosure of our vehicle will be taking a significant amount of the abuse. For this we needed to find a material that was extremely durable and could provide protection from the elements. As an added precaution, we also wanted a material that would be fairly inexpensive so that bulk quantities could be bought, incase that we needed to build more enclosures because of breakage or design changes. The three materials we researched to be used as a suitable enclosure material were fiberglass, carbon fiber, and aluminum. A comparison of the basic mechanical properties and cost between these materials can be seen in Table . In this table we present two measures of tensile strength: the ultimate strength and yield strength. Ultimate strength is a measure of the maximum stress a material can withstand when subjected to tension, compression, or shearing before it fails. Yield strength is the stress at which a material can strain before it deforms permanently. It should be noted that both fiberglass and carbon fiber do not have a yield strength because they are chemically brittle and do not strain-harden which means that their ultimate strength is essentially their breaking point.

	Fiberglass	**Carbon Fiber**	**Aluminum**
Ultimate Strength (MPa)	3,450	5, 650	40-50
Yield Strength (MPa)	N/A	N/A	15-20
(Shock Strength)			
Density (g/cm^3)	2.57	1.75	2.7
Average Price ($/ft)	.91	13.80	7.50

Table - Comparison of Raw Materials for Outer Enclosure
(Table created by our group)

Carbon fiber initially looked like a good solution because of its high tensile strength and extremely light weight, but was quickly dismissed due to its high price per foot and poor “shock strength”. The analogy we constantly ran into with carbon fiber was that it is extremely difficult to stretch or bend, but when hit with a hammer it will easily shatter. Aluminum was also considered as an enclosure material, as many platforms we encountered were constructed out of at least 1/8” aluminum sheet. Aluminum shows good tensile strength as well as the ability to distort and return to its original shape (yield strength) and, relative to other metals and alloys, is one of the lightest we’ve seen. The problem we foresaw with aluminum was not really an issue with the metal, but more in our own abilities to manipulate the metal. Unlike other metals, aluminum requires special equipment to cut and weld it. A special type of metal cutting blade must be used to properly cut aluminum, without this the aluminum will tend to melt or scorch and could possibly cause damage to the cutting instrument. Welding aluminum is also extremely difficult, as it requires extensive preparation of the metal as well as special welding wire and welding guns. It is also necessary to weld aluminum with a TIG welder, which is both more expensive and difficult to use. This is compounded by the fact that none of the members in our group have welded before, which requires any welding work to be contracted out to a third party, which would increase our costs and cause time delays; however, for some of the robots more simplistic components aluminum is still a viable option.

One of our group members at one time had worked in the auto industry and suggested fiberglass as an enclosure material. In researching the material we, found it showed similar characteristics of carbon fiber as a lightweight solution. Unfortunately, like carbon fiber, fiberglass also has a poor “shock strength”, however fiberglass is easily layered thus reducing this weakness. In order to build with fiberglass, a resin of either polyester or epoxy base must be mixed with a catalyst or hardener and applied over fiberglass mat of cloth. The resulting material of this mixture is known as glass-reinforced plastic, or GRP, and is how fiberglass gets its strength. We also found that the price of fiberglass matt is extremely inexpensive when compared with other researched materials. Like carbon fiber, fiberglass is bought by the yard, and from the several sources we found, could be purchased for as little as 50 cents a foot. The fact that we also had a group member experienced in fiberglass construction also meant that we could do all the work ourselves and save on time and costs, which also meant we could rapidly create more enclosures for our needs. Another interesting fact that we came across, while researching fiberglass, is that it is permeable to most radio frequencies and has low signal attenuation properties. Because of its RF permeability, the telecommunications industry often uses glass-reinforced plastic as a method for shrouding the visual appearance of antennas. Based on this research, fiberglass was determined to be the best solution for the construction of our robot enclosure.

After deciding upon glass-reinforced plastic as the material to construct our robot from we continued searching for ways to add additional durability to the enclosure. This research led us to discover the chemical compound Polyurea, or more commonly known as truck-bed liner. This chemical compound is known to be extremely durable and resistant to the elements. From our research we have found that there are several cases of the military using Polyurea as a method to armor several of its vehicles. We also found that a popular Discovery Channel show, Smash Labs, had conducted a series of experiments in which they tested several ballistics and explosives against the compound, with every case indicating no damage to the compound (Figure). Specification sheets from the most notable manufacturer of Polyurea, Rhino Linings, indicate a tensile strength of 41 MPa, making it a significantly tough coating. The other benefit to Polyurea is its quick reaction time and ability to build it up to any desired thickness, thus increasing strength.

Figure - Damage Sustained to Polyurea from Large Grade Explosives
(Figure printed with permission by Wikipedia GNU)

3.7.0 GPS System

We wanted to install a GPS System to be able to track the location of our unit at all times incase our unit became lost or to just get our current location. We will connect our GPS to a UART connection of our PIC microcontroller. We have identified a few products that will make integration easier. We will be able to get speed, 3D Position, and more out of our GPS module.

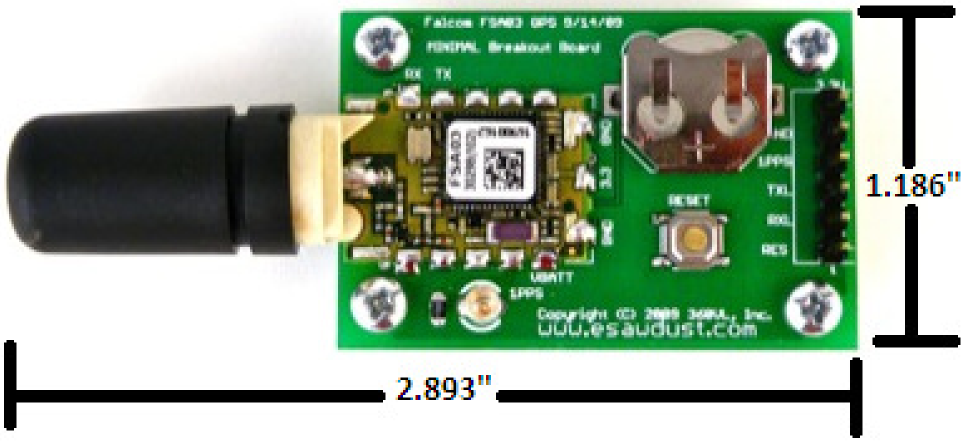


Figure - FalcomA03 GPS
(Figure printed with permission by SparkFun)

We first started to look at Sparkfun's Copernicus GPS dip module. This GPS will help us out a lot due to the fact we can attach it to a breadboard and test our setup. This unit is also relativity cheap compared to most GPS modules, plus it only needs around 20mA to power. The drawback was that you had to buy a separate antenna.

After consulting with a friend I was told of another GPS which has a prebuilt antenna attached to it. The name of this product is called the FalcomA03 GPS breakout board. The main advantage to this GPS unit is that it sends 3.3V level signals for transition and reception, which will work perfectly with our microchip. Other great features of this GPS unit is easy access to the pins, a reset button built on board, battery backup power, and a LED to tell when we have a lock with a satellite. Some of the technical features this product is that it only draws 100mA of power and when it locks on to a satellite it draws 20mA of current. A picture of this unit is above in Figure .

Global Positioning System receivers passively receive satellite signals, they never transmit. The functionality of a GPS depends greatly on the accurate time reference when the GPS receiver receives a message. Every message the satellite sends out has its location and time. These messages are sent constantly with time codes based off of atomic clocks. The GPS receiver can determine how far the satellites are by seeing how long it takes to receive the signal. When a receiver can get the distance of four satellites it can then calculate its position in a four dimensional coordinate system, or time-space.

3.8.0 Power Supply

We are planning on using at least two batteries to power our robotic system, one for our motor circuit and another for the other electronics and peripherals. In choosing power supplies we had to consider the electronics that they would be powering. After researching motors, it was learned that most of the high power motors we are interested in require 24V for operation. For the peripheral circuit, we planned on implementing a 12V source to allow for headroom and expandability, as out highest voltage draw in this circuit would be a 9V camera. We also needed to choose our battery based on its capacity in mA/h. Based on our motor research, the motors we are interested in draw roughly 900 mA. Under perfect conditions, a battery rated at 1800 mA/h would power two motors, in parallel, for an hour. Since these are not ideal conditions, we have to allow for some headroom in case of emergency. A battery in the 2000 to 4000 mA/h range would be best suited if you wished to operate the robot for only an hour.

It was also necessary to decide on what type of battery to use: Nickel Metal Hydride (NiMH) or Nickel Cadmium (NiCd). These are currently the two most common battery types being used in RC and robotic applications. The main differences between the two batteries include capacity, memory effects, and environmental friendliness. Table lists advantages between the two battery types, while Table provides a more numerical comparison between them.

	NiMH Batteries' Advantage Over NiCd Batteries
PERFORMANCE	NiMH batteries will greatly out-perform standard NiCd batteries in high-drain applications.
CAPACITY	The amount of energy stored by the battery. NiMH has more than twice the capacity of standard NiCd. **NiMH batteries have much longer runs times**
MEMORY EFFECT	NiMH batteries can be charged or "topped-off" at any time without affecting battery life. In order to achieve optimum performance from NiCd batteries, they must be fully discharged before recharging. Unlike NiCd batteries, **NiMH has No Memory Effect**.
ENVIRONMENTALLY FRIENDLY	NiMH batteries have no Cadmium added. Cadmium is hazardous to the environment. **NiMH is much more environmentally friendly**
VOLTAGE	The power produced by the battery. Both NiMH and NiCd

	have virtually the same voltage.

Table - Advantages of NiMH vs. NiCd Batteries
(Table was created by our group)

	NiCd	**NiMH**
AA Capacity (mAh)	600 - 800	1,800 – 2,000
AAA Capacity (mAh)	200	600 - 700
Service Life	Up to 750 cycles	Up to 1,000 cycles
Voltage	1.25V	1.25V

Table - Numerical Comparison between NiCD and NiMH Batteries
(Table was created by our group)

3.9.0 A/V System

The audio/visual system portion of our system is going to be sending video and audio to our gateway. We are hoping to get 10 to 30 frames per second out of our camera at minimum. Also instead of sending video and audio through our XBee due, to the amount of data that we would be sending, we have decided to buy a camera with an integrated transmitter that we will pick up with a receiver on the other side with a receiver. We were worried about price of these cameras because we need a small camera for our robot, but after researching we found a few cameras that would be able to transmit about 100 feet which will do the job for our robot, that aren't very expensive.

Another area of research needed in the A/V system, was the method of how to capture the live video feed from the camera and display this on the iPhone. The audio/video signal that comes out of the camera is composite video on an RCA cable. We decided that the easiest way to capture this signal was with a video capture card, which would be attached to our computer. The video capture card will allow us to get the audio/video feed from the camera onto a computer, for distribution to the iPhone.

Once connected, the video capture card is accessible via a variety of open source programs, such as Videolan's VLC or ffmpeg. Both of these programs are an optimal choice to capture the video source because not only will they allow us to view it on the computer, but we can also stream it over a variety of protocols. Determining which protocol to use is dependent on what protocol the iPhone supports. To determine which program to use, we will look at which one can produce the best results for our project. We are looking for real-time encoding and streaming of the video feed. Whichever program can do this better will be the one we use.

The iPhone's built-in media player supports only a select few video types: .mov, .mp4, .mpv, and .3gp. These file formats are only good for pre-recorded video. We needed a solution that would display live video. After further research we found that Apple recently released a new method to deliver live video over HTTP. This "HTTP Live Streaming" method, takes an MPEG2 Transport Stream, chops it into multiple smaller pieces, and then creates a playlist file for the smaller pieces. The playlist file contains the most recent few chunks of the stream, and the client reloads this file to get the most recent videos. Fortunately, Apple provides a command line application, mediastreamsegmenter, which will take in an MPEG2-TS video feed and dump the corresponding file chunks and playlist file to an HTTP server. From the HTTP server, the video stream can be distributed to any client that supports the .m3u8 playlist scheme and the encapsulated video. For the iPhone, the video must be encoded using H.264, and the audio in AAC. This solution seems like it would work out well for our project. The only drawback is that this technology is only a few months old so we would not be able to find extensive documentation on it.

Another possibility that may be easier to implement would be to use a direct mp4 stream from the gateway machine to the iPhone. The tricky thing about this is that the iPhone is very specific in the format that it plays. It must be the correct resolution and have the proper encoding, if any factor is off the iPhone will refuse to play the video. However, if we can get this method to work, then we can avoid the hassle of using the HTTP live streaming method.

3.10.0 Software

This section talks about all of our software components of our robot. Due to the fact that we need to transmit and receive from both the robot and controller (iPhone), we have broken up each software portion into subsections.

3.10.1 Software Communication

In the design of the robot, we decided that we wanted an iPhone to be the controlling device. This introduced challenges of how we were going to get the iPhone to talk to the robot. There are three main elements of data that needs to be communicated across the devices. The GPS data and video feed going from the robot to the iPhone, and the control signals (to control the robots movement), going from the iPhone to the robot.

We decided to use XBee modules, which operate on the Zigbee (IEEE 802.15.4) network, to communicate the control signal and GPS data, as this is common in the robot building community. A drawback with this decision however, was that interfacing an XBee module with the iPhone would require using the proprietary data port that Apple uses. Apple allows companies to send them devices to be approved for use with iPod and iPhone devices. Unfortunately, to get a device certified as “Works with iPhone”, you need to send an actual working device to Apple and get approved. We decided against this option.

Without being able to connect an XBee module directly to the iPhone, we decided to use the iPhone’s 802.11 WLAN connection to talk directly to the board on the robot. We looked at a variety of embedded Linux boards, which would have worked for us, however there wouldn’t be enough design in our project so we had to drop this idea also.

Finally, our third, and final, communication system design included a mix of the first two. We decided to use the XBee modules and the 802.11 WLAN on the iPhone, with a computer, essentially a server, linking the two systems. We decided on having the iPhone talk to the server over 802.11, and then have the server translate that data and send it to an attached XBee module. That XBee module would then send its data to the paired module sitting on the robot. The XBee network will handle the communication of GPS and control data. For the video feed, the Zigbee protocol would not suffice due to low bandwidth so we took a different approach.

For the video feed, we decided to look at pre-built wireless video transmitter/receiver systems and then interface that with our system. We plan on purchasing a video camera, with an antenna on it, and mounting these on the robot. The receiver module will then be connected to the server, via a video capture card or similar, and the video feed broadcast over an mp4 stream to the iPhone.

3.10.2 iPhone Development

The iPhone application will be the primary controlling device for the robot. By making this decision, we had some difficulties in finding a good way to have the user control the robot from the iPhone and even more trouble in finding a way to display the live video feed from the robot onto the iPhone.

There are two main ways to build applications for the iPhone. First, we could have built a web app. Web apps are just websites that are formatted specifically for the iPhone. To access web apps, the user would have to launch Safari, the iPhone’s native Internet browser application, and then navigate to the site. In our case, the site could have been hosted by a 3rd party and be available on the web. Or we could have just hosted a website off of a computer, and have it only available on our local network. Web apps require in-depth knowledge of web design and do not harness the full capabilities of the iPhone. They do not have access to the iPhone's multiple touch gestures and/or the accelerometer, or any other hardware feature that the iPhone offers. This was a major setback as we would have liked to have the user slide their fingers, thereby using the iPhone's multi-touch hardware features, to control the robot. A web app would not allow us to do this. One positive thing about developing a web app, however, was that they were free to develop, and could be developed on any platform compared to the native apps.

Building a native iPhone application is the second approach that we had. Native apps allow access to the iPhone's multi-touch gestures, and hardware functionality that a web app cannot access. One drawback with this approach however was the fact that you needed to develop on a Mac as the SDK for the iPhone is only available on Mac OS X. A second drawback was that for you to get your application installed onto a physical iPhone, we would have to purchase a $99 iPhone developer account. After looking at both options, we decided that we needed the power and flexibility of a native iPhone app. We dropped the idea of a web app.

Now that we had decided to build a native iPhone application, we had to decide on how we wanted to design the GUI. There are a few possibilities that the iPhone is capable of, and each had its strengths. The most complicated way, but also most powerful way, would have been to make an OpenGL ES application. This would allow us full customization of creating anything we wanted for our GUI. However, nobody in our group had experience with OpenGL and the platform is mostly made for game design, not for menus. We quickly dropped this idea and moved onto the next.

The second option was to make a Utility app. This type of application is provided as one of the templates when you create a new iPhone app in XCode. Utility apps are basic applications that contain essentially a "card" that has two sides (Figure). One side displays the main data, and has a small button in the corner to flip the card. When the user taps this button, the card has a smooth animation that flips it around and on the back there are usually settings/configuration options for the app. This type of application looked very slick and had a nice simple interface; however, it was too simple for our needs. We needed something more involved and more of a menu based application. Again, we had to discard this option, as it was not as flexible as we needed it.

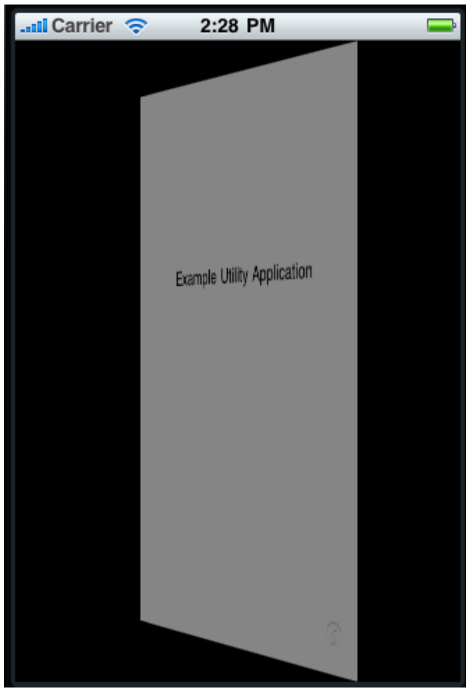


Figure - "Card" Feature of Utility App
(Figure created by our group)

The third option was to make a view based application. This option, like before, was a template provided by XCode; however, it was more flexible than the utility template. The view based app allowed us to create multiple views, each view being essentially a "screen", and the user could navigate from screen to screen by tapping various buttons. Although this option was more complicated to design than the utility application, it provided us with the flexibility that we needed in our app. This option was decided to be the most appropriate for our application.

Now that we had chosen the type of application we wanted to build, we needed to find a solution for how we wanted to control the robot. We came up with three different solutions to get the control scheme working on the iPhone. The first solution was to place a UIImageView control (from the standard Cocoa Touch library), in the center of the screen and have two UISliders on either side of the UIImageView component (Figure)

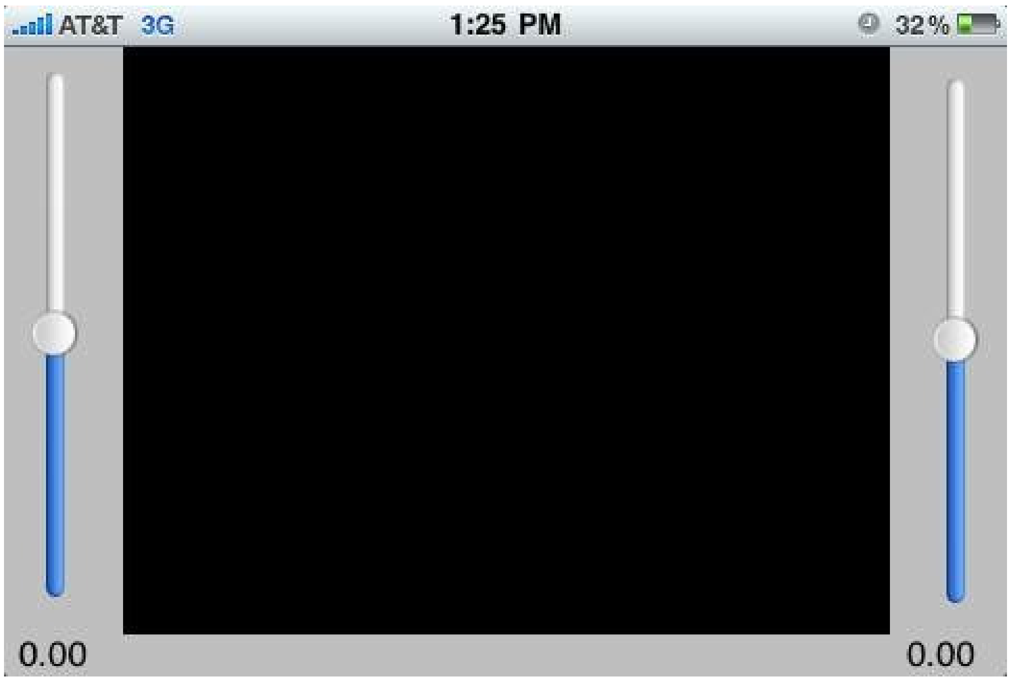


Figure - Prototype of the First iPhone App
(Figure created by our group)

The UIImageView would display the video feed from the robot. We planned to have this control constantly update itself with new jpg images from the server at about 30 frames per second. That frame rate would give a smooth feed and make it appear as if the user was looking at a video. The design worked on the iPhone simulator (which runs on a computer) giving a smooth video feed. However, when we tested it on the actual iPhone device, the images were only updating at around 2-3 fps. This was unacceptable and we had to look for a more appropriate approach.

The second possible solution that we came up with was to use the built-in movie player, MPMoviePlayer that comes with the Cocoa Touch library to display an actual mp4 video stream. However, this video player cannot be embedded in a view; it has to take over the full screen. Knowing this, we decided that we could replace the UISliders from the first design, with the accelerometer built into the iPhone. The accelerometer on the phone was extremely powerful. It could detect the orientation of the phone with respect to all three axes. In Figure we see the orientations of the axes that iPhone reports in. Note that in this figure, the positive Z axis is pointing out of the page. We could easily detect the amount of tilt on the phone, and move the robot left, right, forward, or in reverse, depending on how the user tilted the phone. This design worked with smooth video; however, the control scheme was not as fluid as the first design. To keep the robot still, you had to balance the phone properly. Any axis that was not perfectly balanced would result in the robot moving and tilting the screen while watching the video feed did not feel right. It was not a very fluid approach.

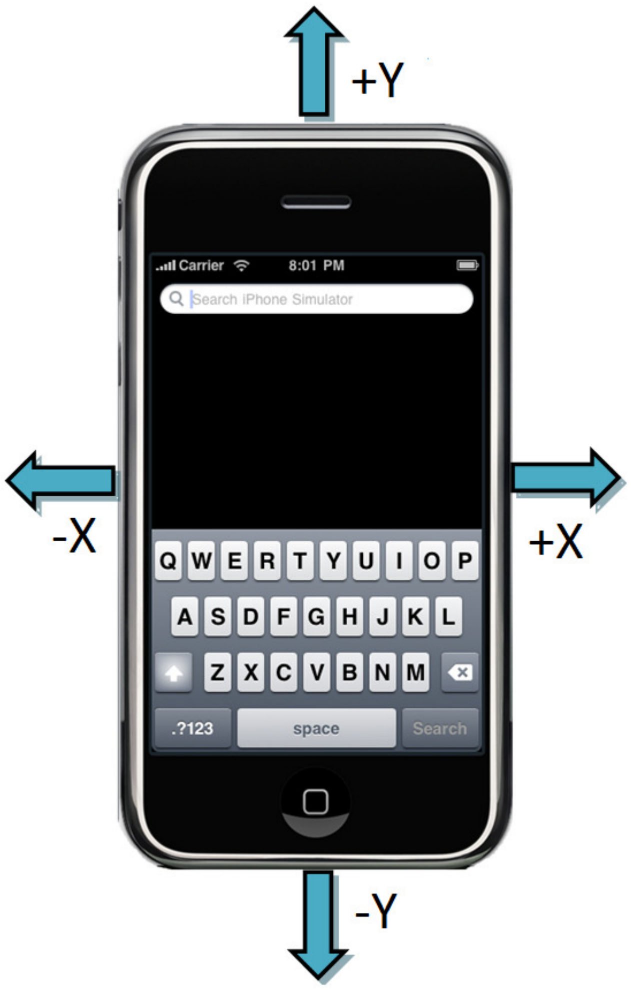


Figure - iPhone's Accelerometer Values
(Figure created by our group)

Finally, we came up with a third solution that combined both previous solutions. After doing more research into the MPMoviePlayer class, we realized that we could overlay our own controls on top of the video while it was playing. This worked out by allowing us to keep the smooth video from the second design, while using the sliders in the first design. Figure illustrates what the final iPhone control design should look like.

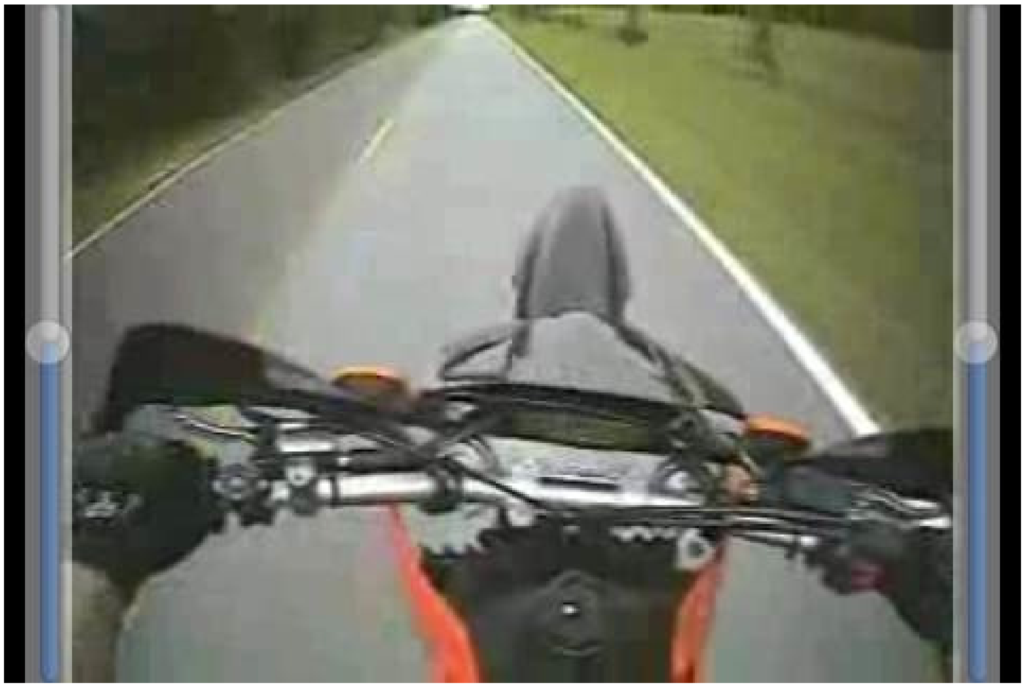


Figure - Prototype of the MPMoviePlayer Class
(Figure created by our group)

In addition to the video feed and control scheme, we also wanted the iPhone app to contain a map that can drop a pin on the location of the user and the robot and then compute the distance between the two. There was only one reasonable option to choose here, and it was the built-in MapKit framework on the iPhone provided by Google and Apple. This component can drop pins on a map based on latitude and longitude coordinates. The provided map GUI component also provides the swipe to scroll, and pinch to zoom features. This functionality satisfies what we were looking for in mapping the locations of the user and robot.

In addition to UI decisions, we wanted to give the ARMORD system some security restrictions. We discussed a variety of options including a keypad like, numeric password system, a username/password combination, or some type of biometrics-based security. We looked at what the iPhone could provide and immediately determined that biometrics was out of our scope. To implement this, we would need to have some type of facial recognition algorithms implemented to process pictures taken with the iPhone’s camera. This was out of our groups’ capability and so we had to decide between the keypad or login options. The keypad based security seemed like a good option at first, however, we wanted something more dynamic that could support multiple users and give them different permissions. A single numeric password was not the right choice for this. The username and password login combination seemed like the best approach to our security requirement and we decided to implement this solution, as it would provide us with multiple users having different security levels.

3.10.3 Gateway Application

The purpose of the gateway application is to link the iPhone to the robot. Since we are using XBee modules to send and receive data, we needed a computer to connect the XBee modules to, as there is no easy way to have an iPhone talk directly to an XBee module so we decided on having a computer link the two items together.

The gateway application must be able to send and receive data over a TCP connection from the iPhone, and then relay that data over a serial port to the XBee modules. To accomplish this goal, there are a variety of programming languages that would allow us to do this. Microsoft's C# and Sun Microsystems' Java languages were the top two choices as our group members had experience with both, and they both could accomplish our needs for the gateway application.

Looking at the two languages, each are object oriented and provide existing classes that would allows us to easily access and send data over a TCP connection and a serial port. They both would suffice in building the gateway application. However, we needed a way for the gateway application to communicate to the XBee modules. One of our group members has had prior experience with sending data to XBee modules using basic C++ so that seemed like a good language to write the application in, however, after further research, we found that there were existing XBee API's, written in both Java and C#, which would allow us to communicate with the XBee modules easily. With the API's, we would not have to worry about the lower level serial communication, and instead we could work at a higher level, just sending data over the XBee modules with the entire configuration done for us. Our decision had to be between interfacing the XBee modules with C#, Java, or C++. We based our decision after looking at each API and deciding which one would be easier to use in our project.

Comparing the API's, we looked at the documentation for both and saw that the Java API was more matured with much more complex features than the C# API. However, the C# API had a better documentation and full examples that we understood completely after one read. The Java API was not as easy to understand. We ruled out the Java API as an option. Next we looked at the C++ method to interface with the application. This seemed like the easiest method as the group had previous experience with this method and we could easily get a program up and running successfully. Making a C++ program to talk to the XBee and a C# program to talk to the iPhone seemed like it would be the easiest way to connect the system. This however brought new challenges of how we were going to interface the C# application to the C++ application. We needed to send the control data from the iPhone to the C# app, and then to the C++ app, then to the XBee, which would then finally send it to the robot. Getting the C# application to talk to the C++ application was another area that we needed to research.

Looking at Microsoft's document on Interprocess Communications, there were several viable options to look at for getting two processes to talk to each other. The first method that they mentioned was the system clipboard. It was very simple and something that we didn't realize could be used to make two processes talk to each other, but after some thought, we designed a system that could have worked out to let the two programs talk to each other. This system would be a simple messaging system pasting data into the clipboard with a timestamp and sender. We could let the C# application paste in a timestamp, the left wheel speed values, and right wheel speed values to the clipboard. Then have the C++ application poll the clipboard at regular intervals (most likely somewhere in the millisecond range) to get the most recent data. However, the C++ application would still need to send GPS location data to the C# application. This could also be done by using the clipboard but then brings in the problem of one application over-writing the other application's data. However, having both of the apps cut the data out of the clipboard and leave nothing in it could easily solve this problem. This would allow an app to check the clipboard for data before writing to it. If it has data, then wait a few milliseconds and try again.

Another method that was mentioned on the document was DCOM. This method is a Microsoft standardized way to send data between compatible applications. After doing some research into this field, which nobody in our group had prior experience with; we decided that it was too complicated and essentially overkill to implement this solution. In fact, we quickly decided that doing any sort of interprocess communication would be overkill.

With this option thrown out, we had to make a choice between using only one application, either a C#, C++, or a Java program. The Java option was the least favorite as we had the least experience with it and the API for the XBee modules was somewhat complicated. Between C++ and C#, we knew that it would be easier to create a GUI with C# as Visual Studio provides a simple designer component that allows you drag and drop controls, and creates the code on the fly. One complication with a GUI however, is that the application would need to be multithreaded. One thread manages the GUI, and another thread would work in the background to communicate with the XBee and iPhone systems.

With the .NET framework, this is easily implemented using the BeginReceive and EndReceive methods in the TcpListener class. The TcpListener class allows an application to start listening for a connection on any port, and then once the connection is established; it can begin to wait asynchronously for data, via the BeginReceive method. In detail, this method takes a slew of parameters; one of the parameters specified is an AsyncCallback delegate. This delegate will call the provided function (any user created function that returns void and has one parameter of IAsyncResult) when the process is complete. In our case, this is what the code would look like the code snippet in Figure .

```
sock.BeginReceive(dataBuffer, 0, dataBuffer.Length, SocketFlags.None,
newAsyncCallback(OnDataReceived), null);
```

Figure - Code to Begin Waiting for Data Asynchronously

As you can see, the OnDataReceived method would be called when there is data ready to be read. This method is working as a callback function. This means that it will be executed on a new thread once it is launched. A tricky thing that would happen here is that we are receiving the data on this background thread, but we would want to display it on the GUI, via the GUI thread. Due to the dangers of multithreading, the .NET framework does not allow any other thread to modify the GUI (i.e. display data on the screen), other than the thread that created it. So for us to get the data from the background thread onto the GUI, we would need to set up a custom delegate (function pointer), to allows us to execute a method on the GUI thread. This method would be invoked by the background thread with the data as a parameter, but then executed on the GUI thread. This allows safe updating of the GUI components and complies with the multithreading standards.

As you can see between C++ and C# we felt that they were equally fit to complete the job. C# had its advantages of a GUI, but then a drawback of a multithreaded GUI. Also, we could still create TCP connections in C++, so we could still connect to the iPhone via a C++ application. We had enough experience in both languages, but our final decision was to go with C#. We chose this as it would allow us to easily access the COM port, create a simple GUI, and working with the .NET framework would make programming much easier.

3.11.0 Computer Hardware

The ARMORD system requires a computer and router to ease the communication from the robot to the iPhone. Here we will look at the different types of hardware available to us, and compare them to see what would be most appropriate for our project. The middle system between the robot and user is called the gateway; this is where everything is processed through.

3.11.1 Gateway Hardware

Once we had decided on using C# as the primary gateway application language, we had another decision of what kind of hardware we wanted to run it on and the operating system. Looking at the hardware, we could choose between a desktop, server type machine, or a laptop. A server was out of the question, as we had no need for that much computing power. The gateway application does not need much power and can run on a low power machine. Between a desktop and a laptop we looked at how our robot would be used. Our main idea was that the robot would be remotely controlled from an iPhone and be easy to move around. To keep in with this idea we felt as though a laptop would be a more appropriate choice as a gateway computer due to its mobility. A desktop would require too many external peripherals to be mobile. However, looking at additional hardware needs, we realized that we needed a computer that had a serial port. Unfortunately most laptops do not come with that port, so a desktop was the only choice remaining. Although choosing a desktop may have cut down the portability of the system, we still would have needed to plug our router into a wall outlet, which would effectively also cut down the mobility. A desktop was the final choice.

Now that we had chosen the hardware we had to look at the different operating systems available to us. We had to choose between Mac OS X, some variant of Linux, or Windows. We had machines running each so the decision was not based on what we had available, it was what would work best for our project. We looked at Mac OS X first and realized that none of us had any experience of running C# code on a Mac. We also quickly realized that the MacBook did not have a serial port, so it would not be able to connect to the XBee module. We scraped the MacBook as an option and moved onto Linux and Windows. Both of the machines we were using, had Linux and Windows running on them as well as having ports available. Comparing the two operating systems, both would allow our program to create a TCP connection, which was necessary to connect to the iPhone, and both would allow us to access the serial port, which was necessary to communicate with the XBee modules. In the end we decided to choose a Windows machine, as that is what the group was most familiar and comfortable with. We decided that since hardware resources was not an issue for the gateway application computer, that we did not need a low-power nor high-performance machine. A windows machine would work fine.

3.11.2 Router

We needed a router to create the 802.11 wireless network between the gateway computer and the iPhone. There are a variety of networking standards available to choose from, ranging from 802.11 A, B, G, and the newest one N. Ideally we would have liked to implement 802.11N into our system as it has the best data transfer rates and the longest range. However, this is not possible because the iPhone does not have the appropriate radios internally to support this new standard. If the iPhone does not support it, or rather if any component in our system doesn't support it, there is no benefit from obtaining a wireless N router. The next best routing technology we could choose was one step down to 802.11G. This is an extremely common standard nowadays, which the iPhone supports, and almost every laptop has a compatible radio. This was our final decision. One benefit of choosing the popular 802.11G standard is that since it is so widely used, we can essentially use any home router and incorporate it into our communication system.

3.11.3 Capture Card

The use of a capture card in our product is vital. Most graphic cards and some motherboards can take video input directly, but there is a lot of processing overheard involved in this process. That is why we will be buying a standalone capture card. With capture cards you have the option of either buying a PCI card or a USB card. If you buy the PCI capture card you will be limiting yourself to a desktop and we don't want to be limited. So we have been researching USB capture cards. The process of the capture card is pretty simple, we take video in from the video receiver and take the output of it and connect to the capture card. After that the capture card, which is connected to the USB port of the gateway, takes it from there. Figure is the capture card we will be using for our project to capture live video.

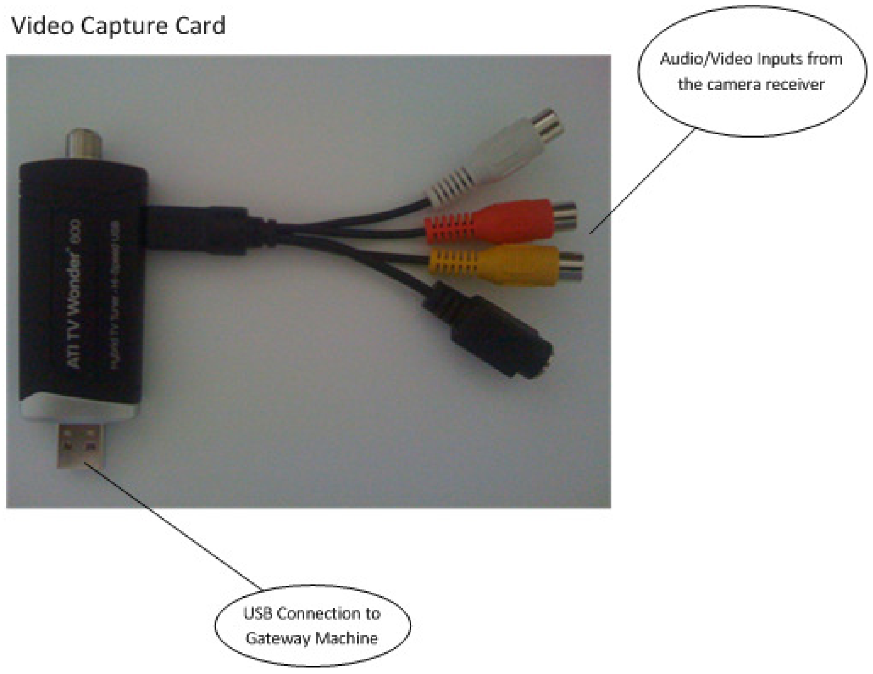

Video Capture Card
Audio/Video Inputs from the camera receiver
USB Connection to Gateway Machine
ATI TV Wonder 600

Figure - ATI TV Wonder 600 Video Capture Card
(Figure was created by our group)

Some of the requirements we needed out of the capture card were a wide selection of video outputs, this was due to the fact we weren't sure what the iPhone was going to take as input. So we made sure to buy a capture card that had at least .mp4, which the iPhone can play. We also wanted to make sure that our capture card had an audio input because our camera outputs both audio and video. We also wanted to make sure we could control the buffering of the encoder, so that we could make sure we are streaming as fast as possible to stop any lag.

4.0.0 Design

This section of our document goes over the detailed steps in the design of each of our subsystems. Seeing we all had different time schedules, we worked on our projects mainly at home. This was not a problem seeing that we had all the tools necessary to build our sections of the robot.

4.1.0 Board Components

This section of design encompasses all electronic board components for our system. We either made our own boards with PCB's or we purchased the unit as is. One of the hardest parts was getting all of these units to talk together. In the coming subsection we break down how we assembled and linked these units together.

4.1.1 Microcontroller

The microcontroller we used is installed on a standard printed circuit board using a retention contact. We use the retention contact so that we don't solder the microcontrollers' pins directly to the printed circuit board. This is in case something happens to our microcontroller we will be able to switch it out easily without having to solder everything again.

The main part after the microcontroller is soldered to the board is powering the device. We need to connect a voltage source that is between 3.3 to 5 volts to the VCC pin on our microcontroller and we need to connect the GND to the ground in on our voltage source. You can't just connect any power to the microcontroller due to the signal not being clean and it probably not being a constant voltage, which is not good for the microcontroller.

The process of cleaning up the signal is first, taking in our higher voltage, which we will say is 9 volts, and passing it through a filtering capacitor. Next we take our filtered signal and put it through a voltage regulator in order to knock down the voltage to what we want. Figure below shows the completed circuit for our power supply. The output voltage should be within 100 mV, which is acceptable.

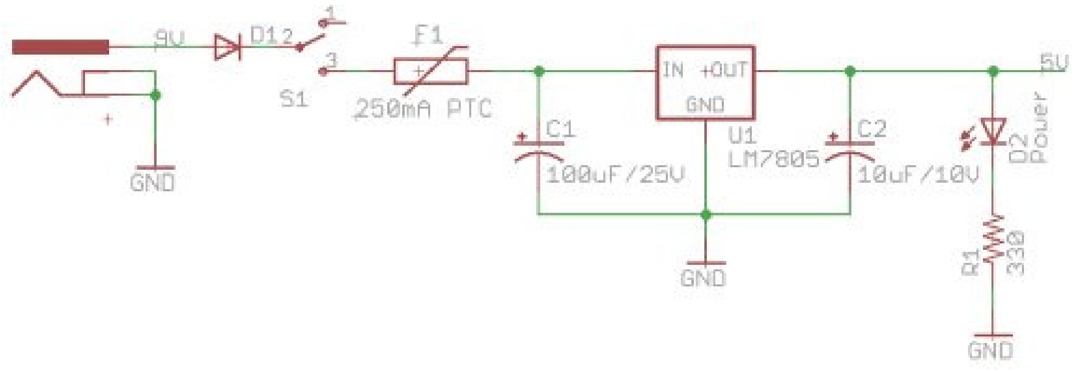


Figure - Power Supply to Microcontroller
(Figure printed with permission by Sparkfun)

4.1.2 Motor Controller

We could have bought just a normal H-Bridge to be our motor controller but we decided against this because the movement of our robot is crucial. So we decided to buy the Sabertooth motor controller to insure stability and reliability with our robot.

The motor controller we purchased allows you to control the motors with analog voltage, radio control, serial, and packetized serial. The connections for the motors are very simple. We simply connect motor 1 to M1A and M1B terminals and motor 2 to M2A and M2B terminals. It also has two terminals to connect your battery maximum of up to 24V using terminals B+ and B-. Two great features of this motor controller are that when the robot motors are not being used it actually recharges the batteries and allows you to make very fast stops and reversing. With the setup we need, which will be non lithium mode, microcontroller R/C input, independent mode with exponential control our switch settings on our robot is show in Figure .

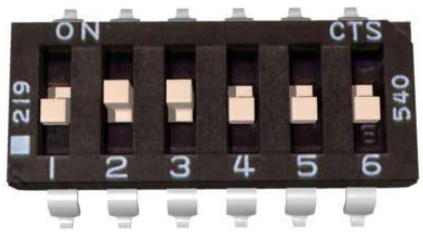


Figure - Motor Controller Settings
(Figure printed with permission by Sparkfun)

Once we have our motor controller setup to the motors, we will then connect our microcontroller to the motor controller. Due to this being a widely used product in robotics, we will modify existing code to be compatible with our motor control setup. We will be using packetized serial to send data between our microcontroller and the motor controller. We will connect the output from our robot to the S1 input terminal of the Sabertooth. I have included a picture of the Sabertooth to show the pin out connections. It also has LED's to show the status of the motors at all times and errors. Figure shows the Sabertooth pin out diagram and shows how easy it is to connect your microcontroller.

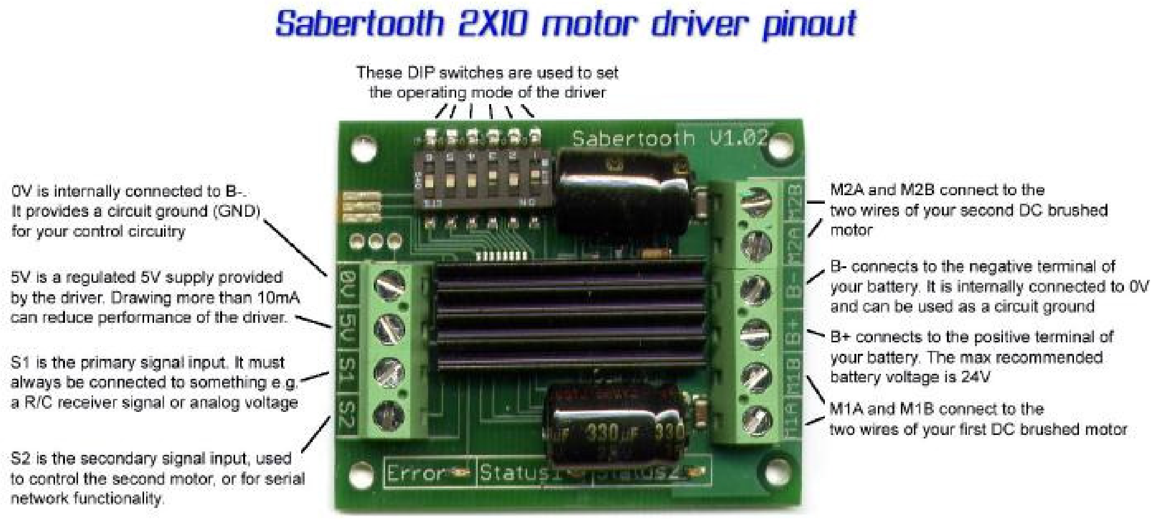

Sabertooth 2X10 motor driver pinout
These DIP switches are used to set the operating mode of the driver
0V is internally connected to B-. It provides a circuit ground (GND) for your control circuitry
5V is a regulated 5V supply provided by the driver. Drawing more than 10mA can reduce performance of the driver.
S1 is the primary signal input. It must always be connected to something e.g. a R/C receiver signal or analog voltage
S2 is the secondary signal input, used to control the second motor, or for serial network functionality.
M2A and M2B connect to the two wires of your second DC brushed motor
B- connects to the negative terminal of your battery. It is internally connected to 0V and can be used as a circuit ground
B+ connects to the positive terminal of your battery. The max recommended battery voltage is 24V
M1A and M1B connect to the two wires of your first DC brushed motor
Sabertooth V1.02
0V
5V
S1
S2
M2B
M2A
B-
B+
M1B
M1A
Error
Status1
Status2

Figure - Sabertooth Pin-out Diagram
(Figure printed with permissions by Sparkfun)

4.1.3 XBee Controller

The XBee controller we purchased has two different versions; both will be needed. The first is called the XBee Explorer, which will be connected to our robot; the second is called the XBee Explorer USB, which will be connected to our gateway.

The XBee explorer is a great solution to be able to add a Zigbee module to a robotic system. The XBee explorer allows easy access to the pins needed to receive, send, and give power to the Zigbee module. The controller also has LED's in order to show what is going on with the device, which is really helpful. Another great feature of this controller is that it can take in 5V and regulate it to the 3.3 voltage necessary to power the device. We plan on connecting the D_{out} of the controller to the Receiving line (Rx) of the microcontroller and we will connect the D_{in} of the controller to the Transmit line (Tx) of the microcontroller. Figure below shows the pin outs of this device. This will allow us to send data in between the microcontroller unit and Zigbee module.

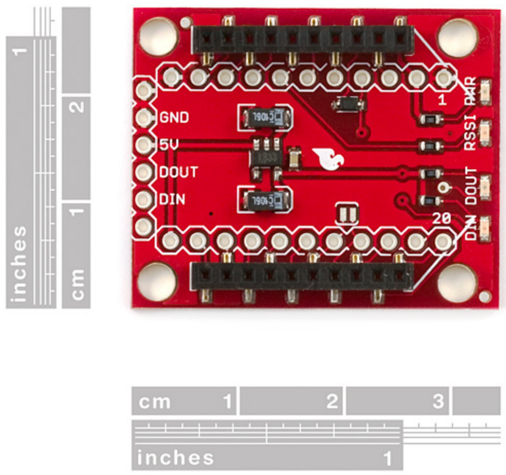


Figure - XBee Explorer
(Figure printed with permission by Sparkfun)

The XBee USB Explorer is very similar to the first controller except it has a USB port instead of pin outs. Figure below shows the XBee USB Explorer unit. This makes it easy for us to connect to a computer and be able to program as a communications port. This controller is also used for configuring the Zigbee modules using their XCTU software. This software allows you to write certain settings to the device needed to change the PAN ID, End Address, and Start address.

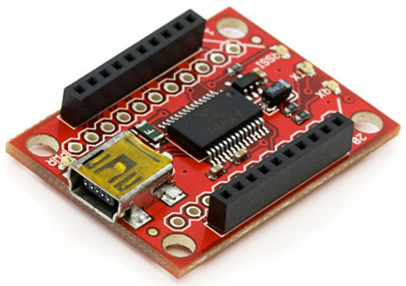


Figure - *XBee USB Explorer*

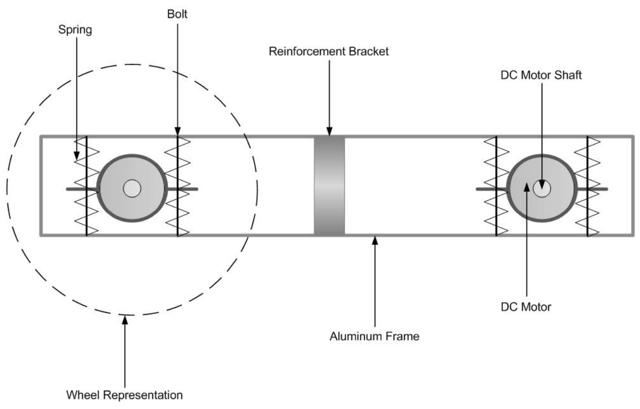
(Figure printed with permission by Sparkfun)

4.2.0 Chassis

This portion of our design section has to do with the body of our robot. This includes the drive train, outer body, inner body, and shock absorbers. We could have bought a pre existing chassis, but we wanted to build our own for a custom fit on our electronic devices.

4.2.1 Drive Train

ARMORD will implement a four-wheel drive system. The motors we will be using to accomplish this task will be four IG42P 24V DC motors. These motors are rated for 255 rpm with a gear reduction of 1:19.This setup will allow for 360 degree turn radius as originally desired. By our estimates, the wheels and axel are going to take the brunt of the impact from drops, thus it was important to design a reliable drive train and suspension system to stand up to the forces expected. Figure illustrates the design of our suspension system and Figure illustrates our drive train. The idea behind this design is that by implementing the springs, we can absorb a good deal of the force taken by falls and alleviate stress on the motors and motor shafts. To alleviate radial load felt by the motor shaft we are mounting the wheels as close as possible to the motor. The aluminum frame sandwiching the suspension system can further be used to mount our electronics.



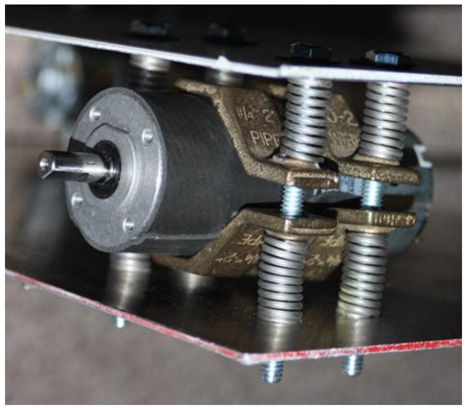


Figure - ARMORD Suspension System
(Figure created by our group)

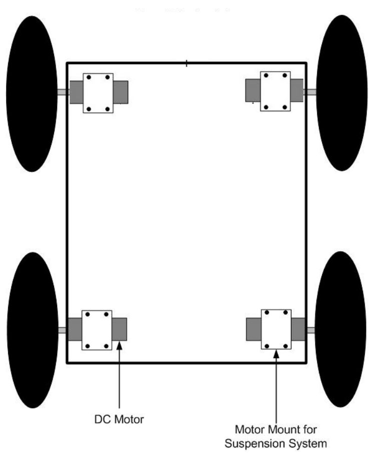


Figure - ARMORD Drive Train
(Figure created by our group)

4.2.2 Body Design

The body is important to our system as it is the first line of defense against impact, debris, and the elements; simply put it is the armor of ARMORD. In designing our body, we had to carefully consider all the peripherals being implemented on our system. It would be easy to just build a box to encase our frame, but we needed to accommodate cameras, ventilation, antennas, etc. We also had to address the more basic functions, such as accommodating for wheel clearance and meeting height restrictions. In designing the enclosure, it was also important to design an efficient system to remove the enclosure in order to make repairs and charge batteries.

The enclosure was constructed in two parts, a top half and a bottom half, both almost identical. The body was constructed by first layering several sheets of extruded expanded polystyrene and sanding them to a basic rounded shape that fit our size requirements. This foam piece would be used as our “plug” for our fiberglass mold. Once we had the desired shape, we laid mold release agent and tin foil over the foam followed by a coat of polyester resin. Once the resin became sufficiently “tacky” we laid two light layers of fiber glass mat onto the plug. Once this dried, we removed the new fiberglass mold. This mold could then be used to create the top and bottom halves to our system. The inside of the mold was then treated with mold release agent and another layer of resin and fiberglass were applied. We used a total of 5 layers of fiber glass mat and one layer of fiber glass woven cloth, for extra strength.

Once we had the initial enclosure pieces we could begin to make modifications for the peripherals and extremities of the system. Small holes were cut to accommodate the motor shafts for attaching the wheels. The next step was to cut the holes for the camera and the ventilation system. To simplify the fabrication, we cut openings on only one half of the enclosure. Because the surface was mostly curves, once the holes were cut, we had to build up the area where fans were to be mounted with additional layers of fiber glass and resin. After this stage, we could then spray our pieces with Polyurea for toughness.

The opening we cut for the camera hole was then fitted with 1/8" piece of Lexan (Plexiglass) and sealed with weather stripping to protect from the elements. The ventilation fans were also fitted with steel mesh to prevent large debris from entering the vehicle. We decided that the best method for attaching the enclosure was to bolt it directly to the frame on the reinforcement straps we incorporated. The method for attachment can be seen in Figure . The enclosure was then fitted with weather stripping along its perimeter to prevent debris and water from entering the vehicle. Figure shows the completed design of the enclosure.

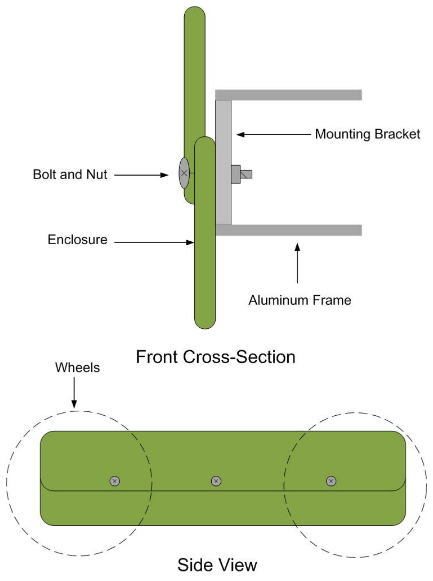


Figure - Attaching ARMORD's Enclosure to the Frame
(Figure created by our group)

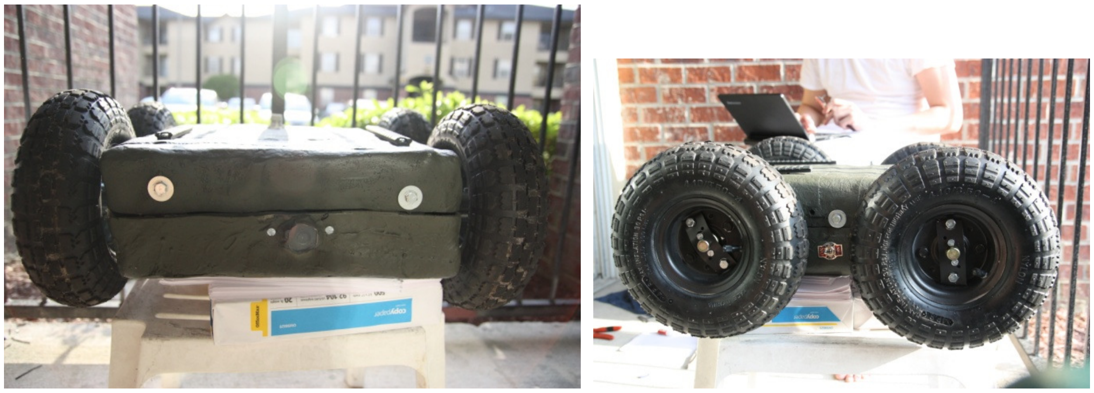


Figure - Front and Side View of Enclosure Design
(Figure created by our group)

4.2.3 Shock Protection of Electronics

In an effort to protect the electronics contained within our system, we have implemented a system to dampen the forces experienced by internal components. One method we have utilized to reduce vibration is using rubber washers at all places components are screwed into the frame. Our most promising electronic protection feature is our unique method of mounting the electronic boards onto the robots aluminum frame, as seen in Figure . This method requires an opening to be cut in the surface of the aluminum sheet about an inch larger than the size of the board. Springs are then connected to contact points on the board and secured across the gap to the aluminum frame with bolts and rubber washers. This method helps to protect the electronics by absorbing much of the force sustained from falls and collisions. Much of the aluminum plating will still be left for secondary components to be mounted.

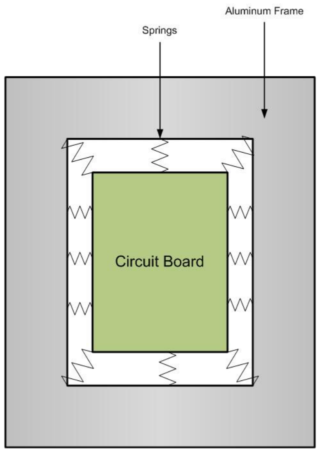
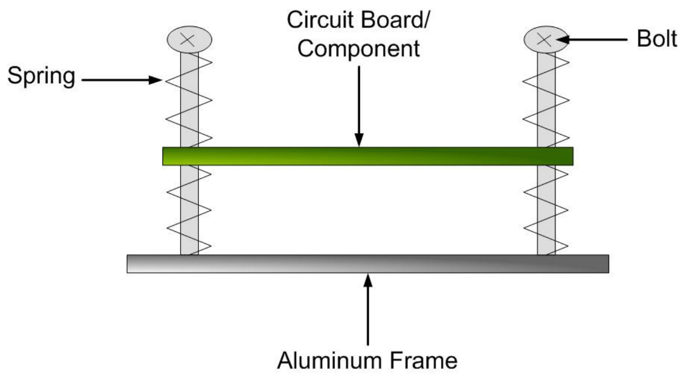


Figure - Method to Mount Electronic Boards to the Frame
(Figure created by our group)

We will also be implementing a secondary mounting system for smaller electronics and components. While not as dramatic as the previous method it will still provide appropriate levels of shock absorption. This mounting solution utilizes bolts surrounded by springs that are embedded into the top and bottom of the aluminum chassis to suspend the components. Figure illustrates this mounting solution. This solution can also be used to mount batteries, voltage regulators, and other secondary electronics.



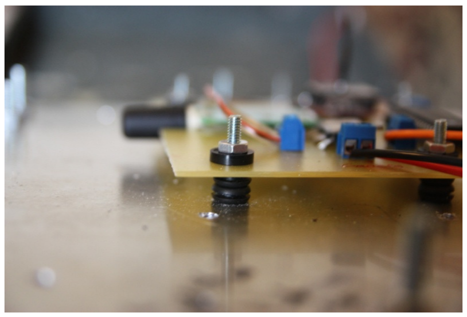


Figure - Illustration of Secondary Mounting Solution
(Figure created by our group)

4.2.4 Cooling and Ventilation System

ARMORD will utilize total of four mini PC fans for its ventilation and cooling system, two intake and two outtake fans. Each fan is 12V and will be powered by the battery hooked to the control circuit. A separate switch will be included for independent control over the ventilation system, thus helping to save power during idle testing times where cooling is not required. Another benefit of independent ventilation control is the ability to more stealthfully navigate an environment; however the fan units for our design are rated for ultra quite operation. Figure illustrates the ventilation systems ability to cool internal electronics and motors.

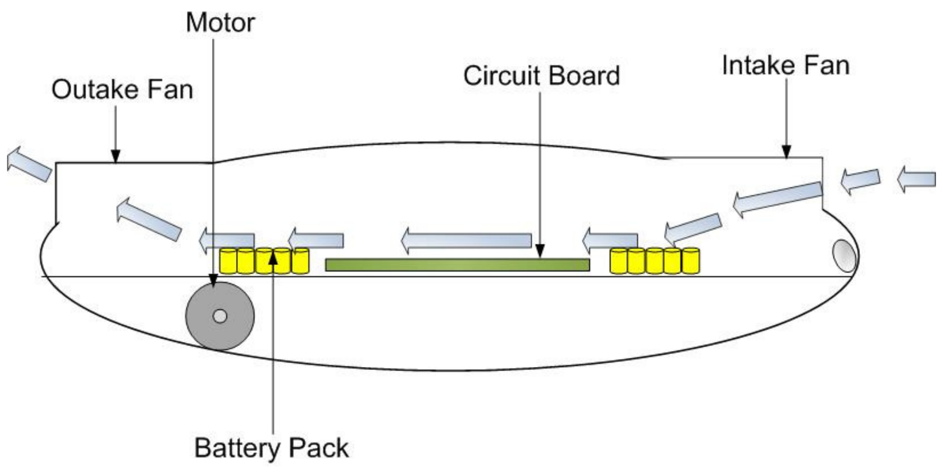


Figure - Illustration of ARMORD's Ventilation System
(Figure created by our group)

We will monitor internal temperature by implementing three DS18B20 digital temperature sensors in the front, back, and middle of the inside of our robot. These units provide accuracy of 0.5˚ Celsius from -55˚ to 125˚ Celsius. We will be able to take an average of these three temperature sensors to get an accurate measurement of the temperature inside our robot. This information will allow us to make sure the internal temperature does not reach a dangerous level for our sensitive electronics. Figure provides an illustration of the temperature sensor while Figure demonstrates correct implementation, within a circuit, of this device.

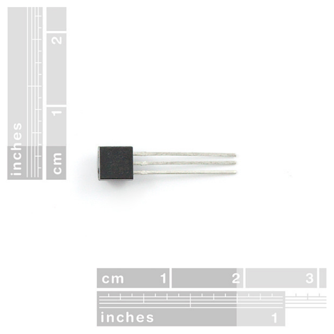


Figure - DS18B20 Digital Temperature Sensor
(Figure printed with permissions by Sparkfun)

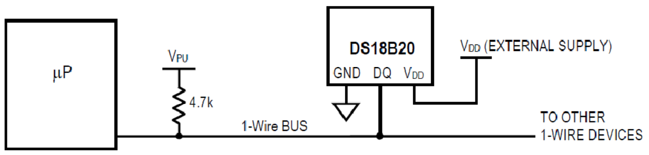


Figure - Correct Circuit Implementation of DS18B20
(Figure printed with permissions by Sparkfun)

4.3.0 GPS System

Our GPS system will be both used for locating our robot in case of a communication failure and for following it on a map application.

4.3.1 GPS Module

The GPS unit we have picked out is the Falcom FSA03 unit; we purchased it with its own breakout board for ease of use. We will be connecting our 3.3Volt source to the VCC pin of the break out board and our Ground to its GND pin. We will then connect the TXL (transmit line) to our USART receive line and the RXL (receive line) to our USART transmit line. This will allow data to flow between both the microcontroller and the GPS unit seamlessly. The pin layout for the breakout board is show in Figure below.

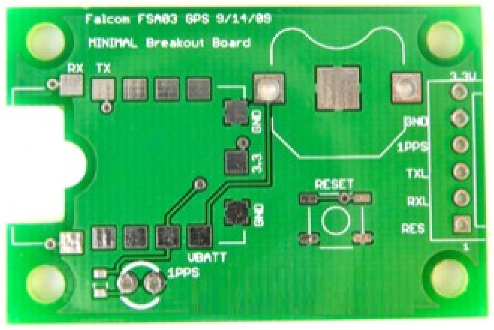


Figure - GPS Breadboard Layout
(Figure printed with permissions by Sparkfun)

4.4.0 A/V System

The audio and video system for our project will consist of getting a live video/audio feed from the robot, to the iPhone in real-time. The method that we took in accomplishing this resulted in a combination of other smaller systems. Figure gives a general overview of how our A/V system works. As it shows, the audio/video stream begins from the camera on the robot. It then transmits the data to its paired receiver station. The receiver station outputs RCA cables which we then feed into a video capture card. The card allows us to process the video/audio on the computer, and prepare it to be sent to the iPhone. Finally once the video is encoded correctly, the data is sent over 802.11 to the iPhone.

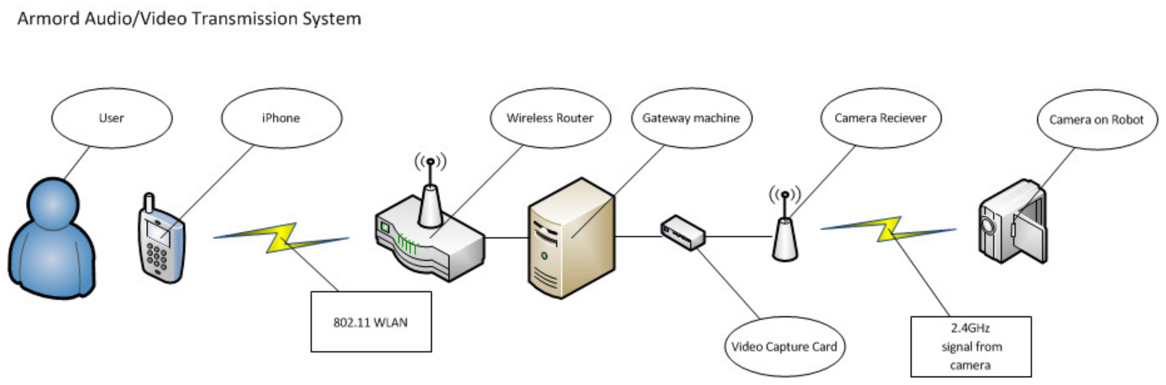


Figure - Overview of the AMORD AV System
(Figure created by our group)

4.4.1 Camera

We were originally going to have our Zigbee module transmit our frames from our camera but due to the communication overhead being too much we decided to buy a camera that came with its own transmitter and receiver. This made us not have to worry about the communication overhead. We decided on this camera because it was the smallest available, while providing a good balance between range and price. Figure illustrates the size and representation of the camera we will be implementing. Another great feature of this camera is that it transmits audio as well as video.

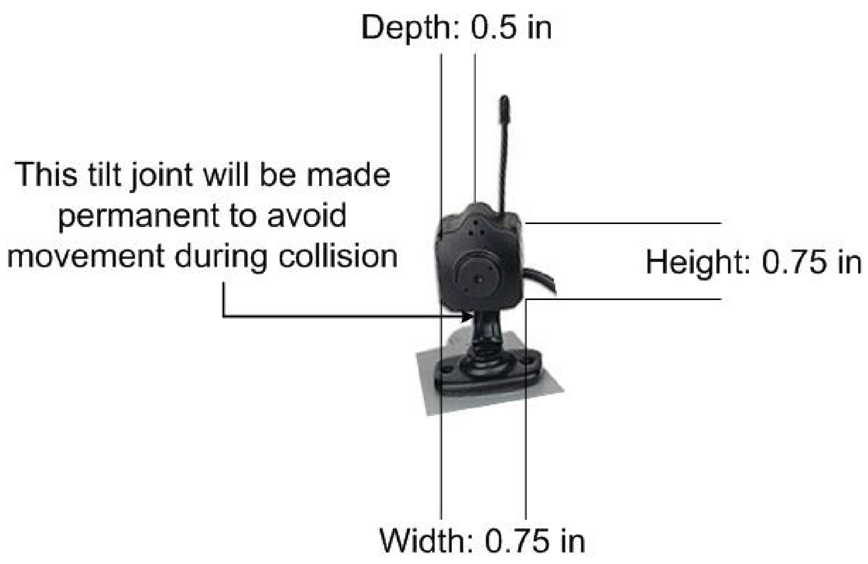

Depth: 0.5 in
This tilt joint will be made permanent to avoid movement during collision
Height: 0.75 in
Width: 0.75 in

Figure - Camera Size
(Figure created by our group)

Another benefit of this camera is the ability for it to run on 9V vice 12V. This makes testing the camera easier, in that all we need is a 9V battery. The lower voltage can also indicate that the camera will draw less current, thus increasing the overall battery life. One short coming found with this camera is its communication range of a 150 foot radius. This limitation encroaches on our requirement to maintain communication up to a 300 foot radius. On the other side we will connect the receiver to the computer where we will use a universal serial bus capture card to the receiver the video. A capture card is made for capturing live video and either storing the video, to a storage device, or for streaming live video. We are using it to stream live video to the user interface or iPhone. We will need to ensure we have lowered the buffering to a minimum otherwise we will have delay. We will not be able to reduce all delay in the system, but we need to get the delay to a minimum, so viewing the robot is not a problem. Once we have streamed the video we will be either picking it up via self programmed application or we will show it on a webpage that will be easily accessed.

Mounting the camera will prove to be challenging in the fact it cannot be mounted in the same fashion as out other components as the added suspension will provide unwanted and “jerky” motion to come through the video. The camera will be mounted directly to the aluminum top plate with bolts. Rubber mat will be sandwiched between the camera and aluminum to increase shock absorption without providing too much bounce. It will also be important to permanently fix the cameras tilting joint at a static angle to reduce any movement it could sustain from falls or collision.

4.4.2 Video Processing

Once the video signal is received by the video capture card, the video processing begins. We obtain the video/audio feed from the capture card using Videolan's VLC application. This open source program allows us to easily access the data from the capture card and stream it to any other computer in a variety of protocols. In our case, we encode the video in H264 and the audio in AAC. This is then encapsulated in an MPEG-2 Transport Stream (MPEG2-TS), which is then streamed to the local machine over a UDP Unicast stream (specifically 127.0.0.1:1234). We stream the output back to the local machine because more processing is needed for the video/audio to be played by the iPhone.

As mentioned before, we decided to use Apple's new HTTP Live Streaming protocol, as this seemed like the only method that would allow us to send a real-time feed to the iPhone. To implement this protocol, we used Apple's "mediastreamsegmenter" command line tool. This tool receives the MPEG2-TS that was output by VLC and prepares it to be distributed to the iPhone. It accomplishes this by segmenting the live stream into smaller video files and creating a main playlist file so that the client knows which files contain the latest video/audio stream. To run this application we execute the command line:

```
mediastreamsegmenter –f ~/Sites/stream –D –t 2 – s 3 – S 1 127.0.0.1:1234
```

This command will read the UDP stream incoming at 127.0.0.1:1234, and segment it according to the HTTP Live Streaming protocol. It will then dump the segments and index file to "~/Sites/stream", which is where our HTTP web server is located. The "-D" parameter will delete the old/unused segments as it deems necessary. "-t 2" will set the length of each segment to 2 seconds. "-s 3", will keep three segments listed in the index file at all times. "-S 1" specifies that the program should wait until 1 file is written before creating the index file.

Once this application is launched, the stream is actively segmented and the index file is updated constantly. Now any client that supports this protocol can download the .m3u8 index file from the web server (over HTTP) and begin to watch the live stream. This entire process is summarized in Figure below.

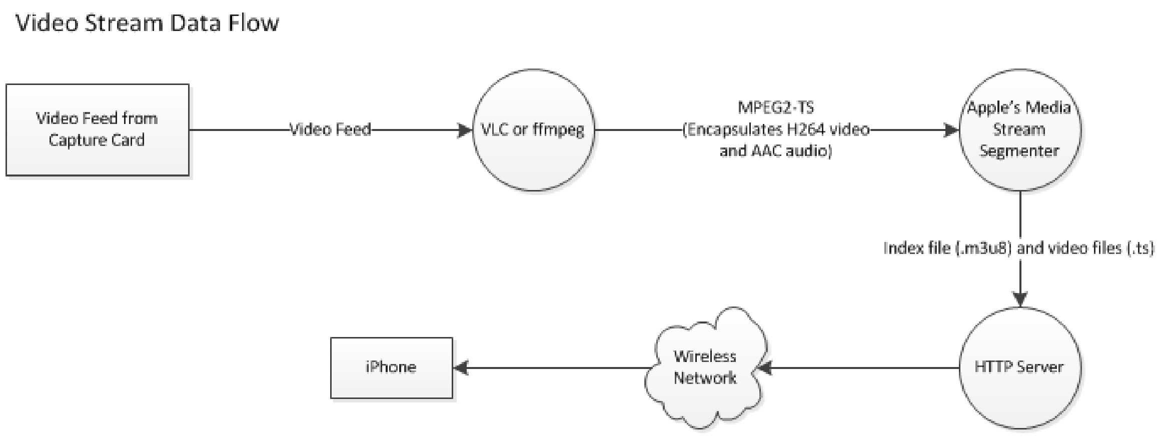

Video Stream Data Flow
Video Feed from Capture Card
Video Feed
VLC or ffmpeg
MPEG2-TS (Encapsulates H264 video and AAC audio)
Apple's Media Stream Segmenter
Index file (.m3u8) and video files (.ts)
HTTP Server
Wireless Network
iPhone

Figure - HTTP Live Streaming Used in our Project
(Figure created by our group)

Also, a side advantage of using this method is that since the stream is placed on an HTTP web server, any client can access the feed, not just the iPhone.

Once we put implemented this process, we encountered several problems. The video format we expected to send to the iPhone was not of the proper type. The iPhone is very specific on what video format it plays, and we could not get the iPhone to play the video. We were able to view the video feed from another computer, but that feed had a 10 second delay, which was too much delay for our specifications.

Our solution to this was to purchase a separate video subsystem. It had its own camera with a paired receiver that broadcast over the 900 MHz frequency. To display the video, we created a custom controller that housed the iPhone, with a 7" LCD monitor attached to it. The LCD monitor displayed the video feed.

4.5.0 Power Supply

ARMORD will utilize two batteries, each powering a separate circuit. Our drive circuit will be powered by a single 24V 4500 mA/h 20 cell NiMH battery and our control circuit will be powered by a single 12V 4000 mA/h 10 cell NiMH battery; both of these batteries are fully rechargeable. The drive battery will be wired in conjunction with the control circuit to the inputs of a double pole, double throw switch. In the "On" position of this switch, the drive circuit will be wired in parallel with two inputs of the motor controller to provide enough power for our two 24V DC motors. In this state, the control circuit will be wired in parallel to several voltage regulators in order to provide appropriate power to several peripherals and electronic control devices. In the "Off" position, the both circuits will be connected in parallel to a charging plug to provide easy hookup for an AC charger. The schematic of our power system can be seen in Figure .

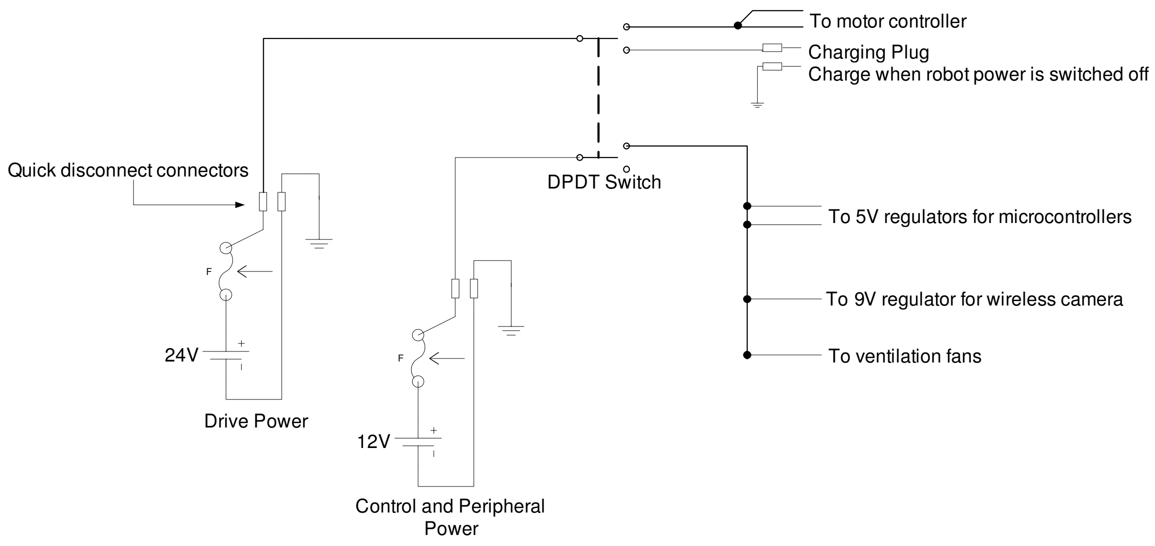

To motor controller
Charging Plug
Charge when robot power is switched off
Quick disconnect connectors
DPDT Switch
To 5V regulators for microcontrollers
To 9V regulator for wireless camera
To ventilation fans
F
24V
Drive Power
F
12V
Control and Peripheral Power

Figure – Schematic of Power Supply and Distribution
(Figure created by our group)

4.6.0 Software

There are three main components in the robot communication system, the iPhone application, the server, and the robot itself. These three components all link together in different ways to send data to and from the robot. An overview of this system is illustrated in Figure .

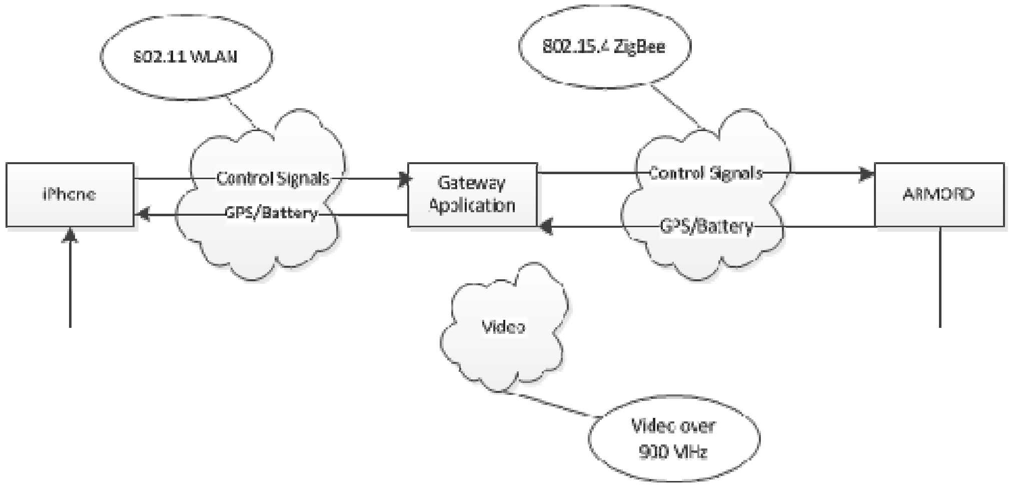


Figure - Data Flow Throughout the System
(Figure created by our group)

The iPhone app is the primary method to access the robot. The iPhone app can view a map of the robots location and control the movement of the robot. It has two sliders on the screen whose values control the left and right wheel speed of the robot. When these slider values are changed, each value is sent over 802.11 WLAN to an application running on the server. The messages sent from the iPhone to the server only contain the left/right slider values. These values, by default, are given in two decimal places. This was too much precision, and resulted in an excessive amount of data being transferred over the network. It was modified so that only values that are multiples of 10 were sent across the network. Not only did that change result in less network traffic and faster performance, but it also solves a problem with the touch-based user interface. This problem is that when a user thinks that they have both thumbs at an equivalent location on the screen, their thumbs may be off by a small amount, resulting in the robot turning slightly when the user wanted it to go straight. By only transmitting values multiple of 10, the user has a “buffer-zone” of 10 units where their thumbs can be at different points on each slider, but it will give them the desired result of moving the robot in a straight line. The actual packet that is sent is just a simple string of three characters. The first character is either ‘L’ or ‘R’, indicating which wheel/slider was changed. The next two characters represent the current value of that slider, ranging from 0 to 90.

Once the application on the server receives a value, it then pushes that data along to the attached XBee module. The XBee module then talks directly to the paired module sitting on the robot. The board on the robot then handles the data, moving the robot in the desired direction.

In addition to controlling the robot, we needed a way for the robot to send video feedback to the iPhone app so that the user can see where the robot is going. The video feed starts with the camera mounted on the robot. This camera transmits its video feed wirelessly to its paired receiver. The receiver, which has standard RCA outputs, is then plugged into a video capture card on the server. Finally, the server streams the live feed to the iPhone app over the network.

4.6.1 iPhone Application

The robot’s primary controlling device is an iPhone application. We used the standard Apple iPhone development library (Cocoa Touch), which is written in Objective-C, and the XCode IDE to create the app. The application has two key features, a live video feed with slider controls used to move the robot, and a map with the GPS location of both the device and the iPhone. All of this data is obtained/sent over an 802.11 WLAN network.

The actual GUI of the iPhone app is composed of multiple view controllers and a navigation controller that pushes the views onto the screen. The multiple screens guide the user through the connection process, until they get to the main screen, where they can look at the map, or begin to control the robot. A prototype of the iPhone GUI can be seen in Figure .

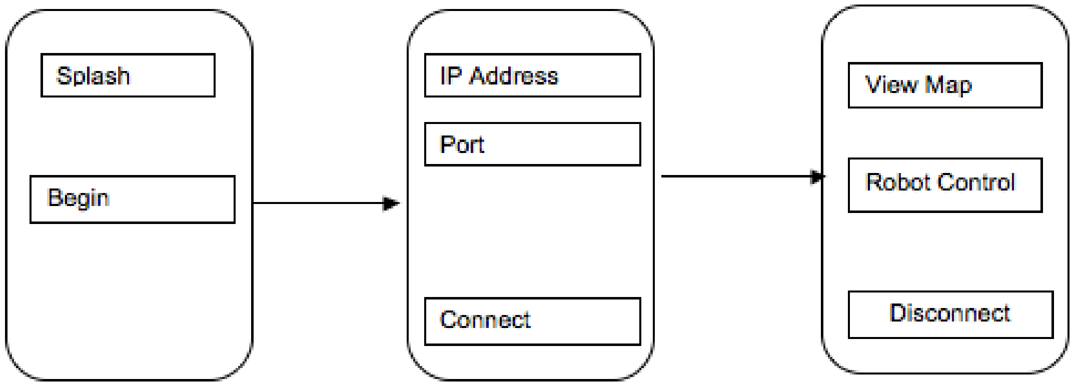


Figure - Prototype of the iPhone GUI
(Figure created by our group)

The initial screen is just a splash screen with a “Begin” button. Clicking this button results in the connection screen view being loaded. Here the user can specify the IP address and port of the server. When the connect button is pressed, the iPhone attempts to connect to the server on the specified port using a standard TCP connection. If the connection times out, an appropriate pop-up message is displayed.

Once the connection is made, the iPhone sends an initial packet of data, which the server then verifies by sending back a similar packet. If the iPhone does not receive a data packet back in the correct format, it knows that something is wrong and terminates the connection. If a successful connection is made, the iPhone app then brings up the main view, which gives the user the option to either see where the robot is on a map, or connect to its drive system and begin controlling it.

The map view uses the provided Cocoa Touch class, MKMapView, to display the map. Once the map has loaded the user’s current location is obtained via the Core Location framework and the robots position is reported via its on board GPS unit. These coordinates are then processed and the mapview then zooms into an optimal position where both points are on the screen. Finally, it drops two pins on each coordinate, and then displays the distance on an overlaid UILabel object.

The robot control scheme uses two components from the Cocoa Touch libraries. The first one is an MPMoviePlayer, which displays the stream that the robot's camera is broadcasting. The second one is two UISliders, which are overlaid on either side of the video. The sliders allow the user to control the robots left and right wheels.

Applications on the iPhone are put together using the Model-View-Controller (MVC) pattern. This results in each "screen" on the iPhone having two separate files. The first file, the XIB file, contains the graphical element of screen, with the GUI components and layout. This is the "View" in MVC. The second file is the controller file, which is the objective-c code that is linked to the XIB. This controller file responds to actions/events that occur on the GUI and performs whatever is necessary. This is the "Controller" in MVC. The Model part of MVC is where all the data is stored. In this application there is no need for much data storage as all data is sent/received over the network.

For our iPhone application, each screen will need its own UIViewController subclass tied to a XIB file. Each view controller, GUI component on that controller, and the corresponding event callback methods are listed in the UML class diagram in Figure . As you can also see on the diagram, there is an ArmordAppDelegate class, which implements the UIApplicationDelegate protocol. This class is essentially the root class of our app. The main application view is loaded from this class, and the iPhone OS will communicate with the application through this class. Another thing to note is the UINavigationController object found in the ArmordAppDelegate class. This object has something like a stack of "screens" and allows us to easily push/pop new UIViewController's onto the iPhone's screen. This object allows seamless navigation throughout the app, and is widely used in iPhone app development.

iPhone Application Class Diagram

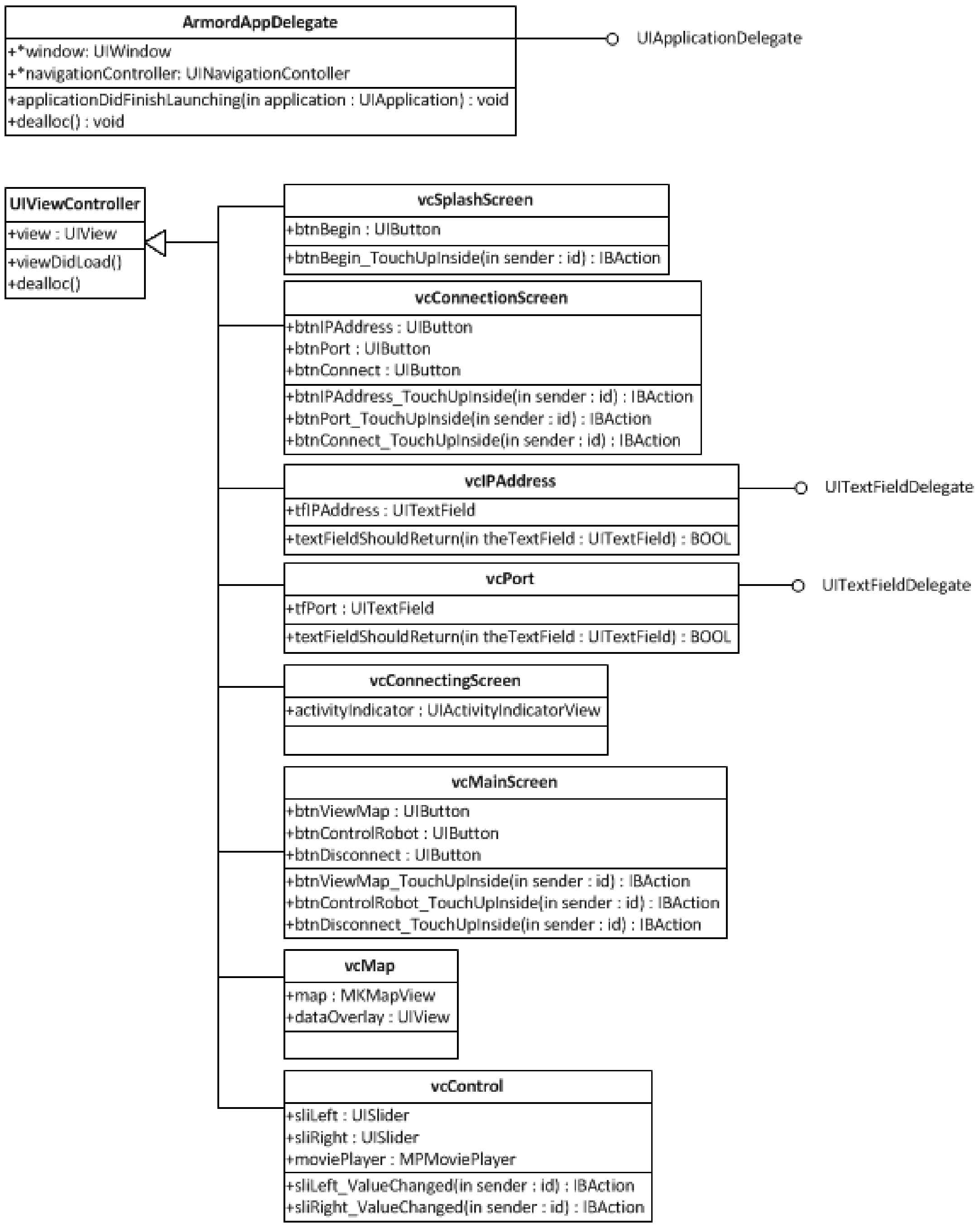

ArmordAppDelegate
+*window: UIWindow
+*navigationController: UINavigationContoller
+applicationDidFinishLaunching(in application : UIApplication) : void
+dealloc() : void
UIApplicationDelegate
UIViewController
+view : UIView
+viewDidLoad()
+dealloc()
vcSplashScreen
+btnBegin : UIButton
+btnBegin_TouchUpInside(in sender : id) : IBAction
vcConnectionScreen
+btnIPAddress : UIButton
+btnPort : UIButton
+btnConnect : UIButton
+btnIPAddress_TouchUpInside(in sender : id) : IBAction
+btnPort_TouchUpInside(in sender : id) : IBAction
+btnConnect_TouchUpInside(in sender : id) : IBAction
vcIPAddress
+tfIPAddress : UITextField
+textFieldShouldReturn(in theTextField : UITextField) : BOOL
UITextFieldDelegate
vcPort
+tfPort : UITextField
+textFieldShouldReturn(in theTextField : UITextField) : BOOL
UITextFieldDelegate
vcConnectingScreen
+activityIndicator : UIActivityIndicatorView
vcMainScreen
+btnViewMap : UIButton
+btnControlRobot : UIButton
+btnDisconnect : UIButton
+btnViewMap_TouchUpInside(in sender : id) : IBAction
+btnControlRobot_TouchUpInside(in sender : id) : IBAction
+btnDisconnect_TouchUpInside(in sender : id) : IBAction
vcMap
+map : MKMapView
+dataOverlay : UIView
vcControl
+sliLeft : UISlider
+sliRight : UISlider
+moviePlayer : MPMoviePlayer
+sliLeft_ValueChanged(in sender : id) : IBAction
+sliRight_ValueChanged(in sender : id) : IBAction

Figure - UML Class Diagram of the iPhone Application
(Figure created by our group)

The application lifecycle of an iPhone application is unique. Other operating systems allow multiple applications to be running at once. However, the iPhone OS only allows a single application to be used at a time. Once the application is run, it is essentially running in a sandbox, contained in its own environment. As

Figure below illustrates, we can see that once the user launches the application, the iPhone OS fires up the application. Next, the Armord App Delegate class, which conforms to the UIApplicationDelegate protocol, creates its main window and loads the initial view onto the screen. In our case, this is our splash screen. All user touches are passed down from the OS onto the current displayed screen, where each screen handles the touch event in its own manner. After an event is handled, the application goes back into its event-wait loop, where it just waits for more user events.

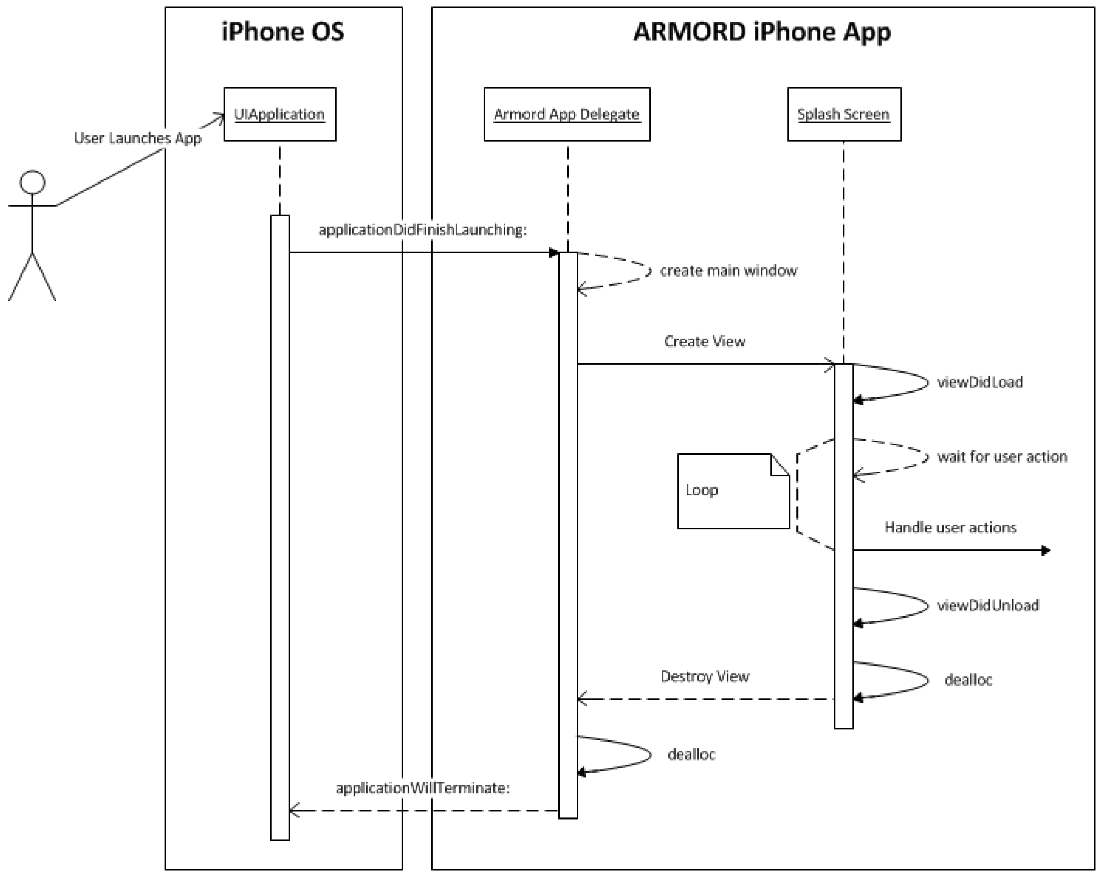


Figure - UML Sequence Diagram of iPhone application
(Figure created by our group)

As the diagram also illustrates, each screen that is loaded has a viewDidLoad, which is called when the view loads. They also have a viewDidUnload and dealloc methods, which are called when the view is being destroyed. The purpose of these methods is for the view controller to perform initialization, or finalization code when the user is navigating around the application. In our case, we use these methods to initialize fields, such as the IP Address and Port of the gateway application, when the view is loaded.

An example of pushing a view controller onto the navigation controller is shown in Figure below. The view controller is created and initialized in the first line.

Since it is tied to a XIB file, the corresponding GUI elements are put in place when the “init” method is called. The next line then calls the “pushViewController: animated:” method on the applications navigation controller, which actually pushes the “vc” view controller onto the screen. The newly created view controller is passed in as the new view to be pushed and we set the animated Boolean parameter to true, which will give a smooth animation of sliding the new view controller onto the screen.

```
vcSettings *vc = [[vcSettings alloc] init];
[self.navigationControllerpushViewController:vc
animated:YES];
[vc release];
```

Figure - Pushing a View Controller onto the Screen

The last line in the figure above releases the local copy of the view controller, as the “pushViewController:animated” method retained its own copy. Once the view controller has been pushed onto the screen, we can “pop” it off the stack/screen by calling the “popViewControllerAnimated:” instance method of the UINavigationController object. This method will pop the current view off the screen, and return the user to the previous screen.

Another option to push view controllers onto the screen is to use the “presentModalViewController:animated:” instance method that the UINavigationController class provides. The difference between this method and the previous one is that modal view controllers are displayed for temporary, or configuration purposes. They are slid up from the bottom of the screen, instead from the left, and this is commonly used in iPhone applications to represent a quick new view where a user can make an edit or do a side task to the main program. The modal view controllers will be used when the user taps the IP Address or Port buttons on the main screen. A simple view with a textbox and keyboard will slide up from the bottom of the screen and allow the user to enter the necessary information. A “done” button in the top right corner of the modal view will dismiss the current modal view controller by calling the method “dismissModalViewControllerAnimated:” method of the parent navigation controller. We have to call this on the parent view controller as modal views are considered children to the main views. If we were using the pushViewController method, we would be creating a sibling to the current view controller.

Another key element of the IP Address and Port fields is that they will retain the last used IP address and ports that the user entered. This is a convenience feature so that every time the user starts the app, they don’t have to reenter the IP address and port of the server. The iPhone OS allows applications to save their data in the <Application_Home>/Documents folder. This data will persist when the application is closed, and not be overwritten when the app is re-installed or upgraded. The main way that we chose to store data was in a plist file. Plist, or property list, files are just XML files that store key-value pairs of data.

The data is restricted to basic types of strings, integers, arrays, dates, and data. The reason for this is that plist files are frequently loaded into NSDictionary or NSMutableDictionary objects for easy read and write operations. Once data is loaded into a dictionary, it can easily be re-written to the plist file and saved in the applications documents folder for later use. In our case, the plist file will just contain two key-value pairs, one for the IP address and one for the Port (Table). The application will load the path to the documents folder using the NSSearchPathForDirectoriesInDomain method provided by Apple's Foundation framework. We can provide the method with a parameter specifying that we want the documents folder, then it will return the path to that folder. Once we have the path, we can check to see if our plist file exists and create it if it does not. If it does exist, we load the data out of the file into a dictionary and then fill the appropriate data structures.

Property	*Type*
IPAddress	*NSString*
Port	*NSString*

Table - Configuration of the Connection Information Plist File

One main feature of the iPhone application is the mapping feature. When we collect the GPS data from the robot, we use the latitude and longitude to plot the position of the robot onto the map. We also plot the location of the user. To accomplish this, two key frameworks are used, the MapKit.framework and the CoreLocation.framework. The MapKit framework provides access to the MKMapView class, an easy to use Google maps object. It has a slew of properties that allow a satellite, map, or hybrid view, just like on Google's maps website. It also reacts to slides/pinches to move and resize the map. The CoreLocation framework opens up the GPS location features of the iPhone and allows access to the latitude and longitude coordinates of the user's iPhone. In order to plot locations on the map, we need to add "annotations" to the map. These annotations can range from the standard pins that are used in the official iPhone maps application, to any sort of customized UIView. To add an annotation to a map, we needed to create a custom annotation class that conforms to the MKAnnotation protocol (interface). This protocol requires that an annotation has at least a CLLocationCoordinate2D coordinate, which is just a data structure that holds latitude and longitude position. Once we created our class, we need two instances of it, one for the robots location and one for the user's location. The two annotations can be plotted on the map, and then the distance can be calculated from their coordinates and displayed. Adding an annotation to a map is very simple. Figure shows an example of how we do this.

```
MapAnnotation *ann = [[MapAnnotation alloc] init];
ann.title = @"Annotation Title";
ann.subTitle = @"Annotation Subtitle";
```

```
            ann.coordinate = currentLocation;
            [mapview addAnnotation:ann];
```

Figure - How to Add an Annotation to the Map

As you can see in the figure, we create a MapAnnotation object in the first line. The MapAnnotation class is a custom class created by us. It conforms to the MKAnnotation protocol and also implements the optional title and subtitle fields. Lines 2 through 4 set the properties of the annotation. The "currentLocation" variable is a CLLocationCoordinate2D global class variable that holds the current location of either the user or the robot. The final line adds the newly created annotation to the mapview object, which is our MKMapView. When updating the current location for the user, the iPhone tends to have some initial warm up time before locking in on an exact location. To handle this, the current location is constantly updated until it starts constantly reporting the same location. Once this happens we can add the annotation to the correct position.

Another main feature of the iPhone application is the movie player that is used to display the video feed. The movie player is provided by the framework and is called MPMoviePlayerController. This class allows us to load a movie (or stream) from a URL, whether the file location is local or remote. Once it is loaded, we can start playing the move (assuming it is of the right type), by simply calling the "play" method. When this is called, the movie player takes over the full screen and begins to play its movie. In our case it will begin to play our live feed of the robots camera. We also want there to be two sliders on top of the movie player so that the user can navigate the robot while watching where it is going. When the movie player begins, we need to grab the window that it is playing on. We can do this with the code in Figure below. As you can see, we get an instance of the singleton class UIApplication, and get the "keyWindow" property of it. The keyWindow is a property that always contains a UIWindow reference to the window currently in view, in our case it is the movie player.

```
UIWindow *moviePlayerWindow = [[UIApplicationsharedApplication]
keyWindow];
```

Figure - Code Snippet to Grab the Current Window

Once we have a reference to the movie player window, we now need to add our own UISliders to the frame. The UISliders were created in the XIB file (via Interface Builder) and we have references to them in our code called leftSliderOverlay and rightSliderOverlay. We need to position the sliders at the left and right sides of the movie player. However, since the movie player is launched in landscape mode, we actually need to position the left slider at the top of the screen and right slider at the bottom of the screen. The snippet of code that achieves this is listed in Figure below.

```
// set the position of the left and right sliders
CGRect frame;

frame = [leftSliderOverlayframe];
frame.origin.y = kSliderGap;
leftSliderOverlay.frame = frame;

frame = [rightSliderOverlayframe];
frame.origin.y = 480 - kSliderGap -
          rightSliderOverlay.frame.size.height;
rightSliderOverlay.frame = frame;
```

Figure - Code to Set the Position of the Sliders

As you can see from the code above, we declare a CGRect object. This object is a temporary variable used to manipulate the properties of each UISlider. In line 2, we get a reference to the current leftSliderOverlay's frame. The frame is the actual rectangle shape of the view that the slider is in. Once we have the frame, we set the y position of the frame to a constant, kSliderGap. This constant is our gap between the side of the screen and the start of where we place the slider. After setting that, we change the leftSliderOverlay's frame to the newly modified frame.

With the rightSliderOverlay, we have essentially the same procedure but we also have to take into consideration the height of the slider and the move it down to the bottom of the screen. The iPhone is 480 pixels tall, so we subtract out the height of the slider and the kSliderGap to get our final position for the rightSliderOverlay. After we've modified the frames for the sliders, we finally need to add them to the current window, which we grabbed earlier. We can do this with the code snippet in Figure below.

```
[moviePlayerWindow addSubview: self.leftSliderOverlay];
[moviePlayerWindow addSubview: self.rightSliderOverlay];
```

Figure - Code to Add the Sliders to the Main Window

As you can see, the code is simple; we call the "addSubview:" method of the keyWindow (in our case the movie player), and then provide the two slider overlays that we modified as the parameters. The final result is the movie (or stream) will be playing in the background, with the two sliders permanently on top of that view.

Other UI elements that we used in the iPhone application included a UIAlertView. This class provides a message box popup that can be used to grab the user's attention immediately for warnings, errors, or very important messages. Figure contains an example of a UIAlertView popup. As you can see, it can display basic information to the user, and it forces the user to respond to it immediately as it blocks the entire screen.

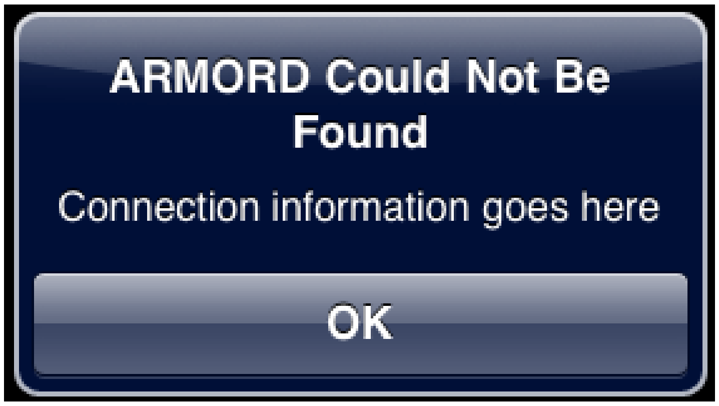


Figure - Example UIAlertView Error Message
(Figure created by our group)

The code necessary to display this screen is extremely simple. All we have to do is allocate a UIAlertView object, and then in the init method, we can customize the entire popup. We can set the title, message, and even customize the number of buttons on the alert, and what each says. Once all of that has been setup in the init method, we can then display the popup by calling the “show” instance method of our UIAlertView object. One important thing to do is appropriately release the object right after calling the “show” method or else we are causing a memory leak. In our application, we use UIAlertView's to display connection error messages when the iPhone application cannot find the server, or when the server disconnects from either the iPhone or the robot.

Another UI element that we use to enhance the user interface is the UIActivityIndicatorView. This simple class displays a spinning activity indicator that is a symbol to the user that the application is doing something. It’s the equivalent of a progress bar, but it has to progress value. It is in an infinite loop that just ends whenever the work is done. To use this class we first need to create a UIActivityIndicatorView object and position it on a view. Once it is has been created, we can start and or stop the gear from spinning by calling the “startAnimating” and “stopAnimating” functions. This is a really simple class to use and we implemented it by spinning a gear whenever the iPhone application is attempting to connect to the server. This is useful because the user knows that the application has not frozen, but is doing work in the background. Figure shows our implementation of this object.

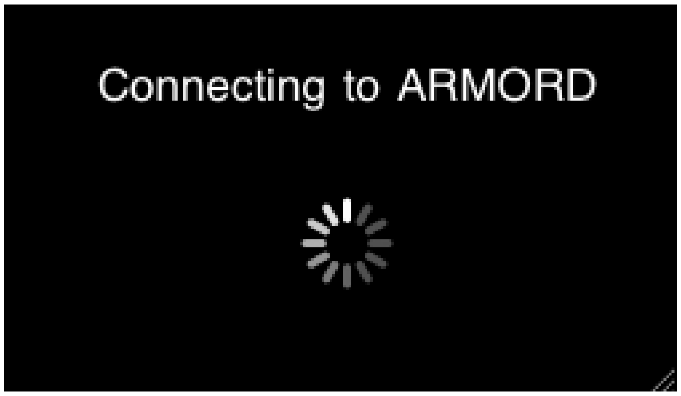


Figure - UIActivityIndicatorView on our UIView Overlay
(Figure created by our group)

One other thing that we used in our design was a small UIView (pictured above), which would allow us to show the connecting status on the main home screen. This view is displayed over the home screen when the user hits the connect button to indicate that the application is attempting to connect to the gateway application.

That concludes the how the iPhone application is built, however, there is one more topic related to the iPhone app. We need to get the iPhone app binaries installed onto a physical iPhone. This process is fairly simple once the initial configuration is done. To get an iPhone application onto a physical iPhone (not the simulator), we first needed to get a Development Certificate. This certificate is obtained from Apple, after paying the $99 developer fee, and is used to sign all code that we produce and wish to run on the iPhone. Once they provided us with that certificate, we then needed to register our device with our account by providing its UDID, which is a unique number on each iPhone. Lastly, we needed to create an App ID, to identify our app uniquely. This App ID can be re-used for development purposes and so we only needed one App ID and we could use it for multiple test applications.

Finally, we finished setting up the configuration to get an app onto the iPhone. The last thing we had to do was create a Provisioning Profile. This profile ties together Developers, iPhone devices, and Applications to be used for testing. The provisioning profile is installed on the iPhone and then any apps that have been signed with the same credentials can now be run on that test device. If we wanted to distribute the application, then we would need to create a distribution provisioning profile. It is essentially the same thing as the development one, but just made for ad hoc distribution.

The iPhone application also has a user login feature. This security measure was implemented to prevent unauthorized access to the ARMORD system. Before any connection can be made to ARMORD, the user must provide proper credentials via a username and password. The username and password will be stored in the application itself within a credentials file. However, we will not store it as plain text; we will instead hash the password and stored the hashed string. Hashes are one way encryption and cannot be decrypted. This means that when the user types in their password, we will hash their provided password and compare it against the correct hashed password previously saved in our application. The iPhone SDK has provided us with libraries for implementing an MD5 hashing algorithm so we can implement this feature with little effort. If a user fails to enter their password correctly after three attempts, we will lock that user out of the system for some amount of time. This can be tracked by adding more entries to the credential file, in addition to the username and password, a login attempt count, and a next allowed login date. The login attempt count will track how many times the user has incorrectly attempted a login. The ‘next allowed login’ date entry will hold a date and time when a locked out user is allowed to re-login. These two entries will be reset to 0 and null, respectively when the user successfully logs in. The only time that the ‘next allowed login date’ entry will be set is after a user fails to successfully login after three attempts.

The credentials file will be a plist, or properly list file. This XML based file is commonly used in iPhone development and is also used to store other data in our system. The setup for the credentials plist file will be as described in Table below.

Property	*Type*
Username	*NSString (plain text)*
Password	*NSString(MD5 Hashed)*
Login Count	*NSInteger*
Login Date	*NSDate*

Table - Configuration of the Credentials Plist File
(Table created by our group)

Similar to the connection information plist file, the credentials file will also be stored in the <Application_Home>/Documents folder. Storing the password in an XML text file on the hard drive may seem like a very insecure concept, however that is not the case. Inside the file, we will store the application password in a MD5 hashed form. This cannot be reversed to obtain the original string.

4.6.2 Microcontroller Code

The microcontroller code is written in C++. This is a lot easier than writing in assembly or MIPS. The main purpose of this code is to control the robot and send data back to the gateway.

The microcontroller will have two states with a semaphore type of function. If one state is running, the other will not be able to run. This prevents data corruption and both states entering the critical section of the program. This works by having two concurrent threads, which means that we have two sets of functions running at the same time, but due to the semaphores only one will run at a time. You might think that this will cause delay, but due to this application running millions of instructions per second this is not an issue. Figure below shows visually how the system works.

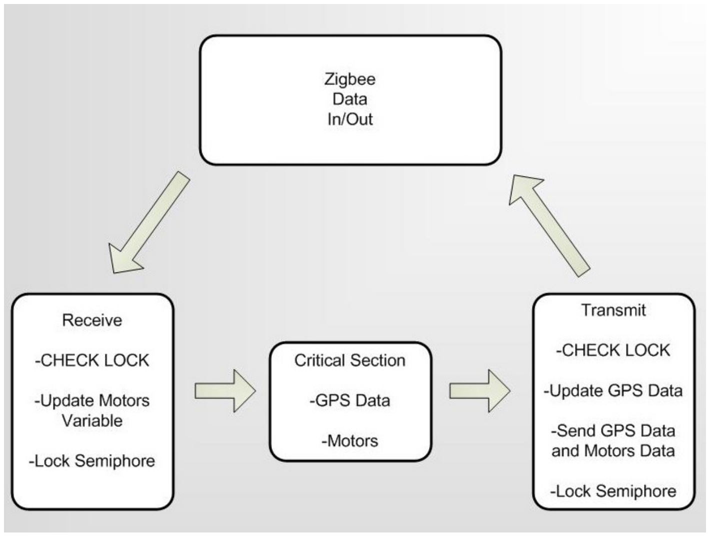


Figure - Microcontroller Code
(Figure created by our group)

The first state of the program will be the Receive state, which will take in from the user whether it needs to move one of the sets of wheels or both. This will be done with an easy increment variable, which will slowly move its way back to zero when it is not being incremented. After these variables have been sent they will be sent to the motor controllers, which will send pulses to the motors in order to make them move.

The second state of the program will be the Transmit state, which will send the position of the unit, which we get from our GPS unit and the current state of the motors back to the Zigbee device.

The critical section of our microcontroller code is what holds all of our important values. We have a variable called GPSLocation, which is a string that holds the GPS location of our robot at all times. We also will have a structure that contains two integers, called LeftMotor and RightMotor. They variables will hold how much we need to accelerate our motors or decelerate. If we receive a value of 0 we know that motor was not asked to move so we exponentially will have the motor slow to a stop.

This process will keep our system up to date constantly, throughout execution of our program. The critical section of our microcontroller consists of a structure that contains all of the variables we need to update the motors and the GPS. Figure shows what our structure looks like, which consists of integers, strings, and Booleans. Most of this structure has been previously explained, the only portions that haven't been explained are the Booleans. The variable Running is going to return to us true if the robot is in running mode, and false otherwise.

```
Struct CriticalSection
{

        Int LeftMotor;
        Int RightMotor;
        String GPSLocation;
        String GPS Time;
        Boolean Running;
        Boolean CommLink;
        String RobotName;

}
```

Figure - CriticalSection Structure

4.6.3 Gateway Application

The server application is written in C++. The main purpose of this program is to ease the communication between the iPhone and the robot, by providing an easy way to send data to the XBee module.

When the application is started, it hosts a TCP listener on the port of a local machine. It then waits for a connection on that port, and when it receives one, waits for an initial message to be sent from the iPhone. This initial message sent from the iPhone is then answered with a similar message from the server application, creating a "handshake" between the two of them so they can begin to transfer data. This handshake is useful because if the iPhone does not receive the initial packet from the server, it knows that something went wrong and that it cannot connect to the robot. This allows the iPhone app to handle the situation gracefully with a pop-up message instead of just sending packets to nothing.

After the handshake has been established, the application is then ready to start receiving data from the iPhone. The messaging structure between the iPhone and the gateway application will just be strings encoded in UTF-8. To send data through a socket, the string will be needed to be broken down to a byte stream, sent across the network, and then recombined into a string. Once we get the string, we parse through it to make useable data. This is necessary, as sometimes the iPhone will send multiple slider values in one packet. Once we have useable data, we then pass the data along to the XBee related functions, which prepare and send data to our XBee module.

The format that the iPhone sends its control data in is a string of a letter, indicating left (‘L’) or right (‘R’), and then a value right beside that, ranging from 0 to 90. The gateway application then parses through that data to make the appropriate data structure to send them to XBee module.

In addition to sending data to the robot, the gateway application also receives data. It receives GPS data from the XBee module, specifically a latitude/longitude coordinate and converts that data into a string for the iPhone. It sends one packet at a time, consisting of the latitude, followed by a semicolon delimiter, and then the longitude is concatenated to that string. It then breaks the UTF-8 encoded string down into a byte stream, transports it across the network to the iPhone, where the iPhone will convert the byte stream back into a readable string, and then take appropriate actions.

5.0.0 Testing

In order to evaluate our specifications and requirements we have developed several testing procedures for key aspects of our system.

5.1.0 Communication Range

This test will evaluate the communication range of our robot. We will perform this test in both ideal and poor radio frequency environments. For the ideal location we will use a vacant mall parking lot, as this will allow for unobstructed line of sight and low amounts of RF interference. Our poor RF location will be the University of Central Florida campus where we have multiple obstructions and large amounts of RF signals in the air. To perform this test, we will find a stretch of at least 100 ft of ground in both environments. We will then proceed to drive the robot in a straight line until we lose communication with the vehicle and it stops moving completely. The distance traveled will then be measured and recorded. This test should be preformed several times at each location in order to obtain an average communication distance.

Results:

In both optimal and RF-heavy environments we observed communication distances in excess of our required 100 ft range. We have 100% confidence that this device will perform without failure in any environment.

5.2.0 Video Communication Range

This will evaluate the distance the robot can travel without losing its video signal. This evaluation will take on the same format as the communication range test, only with video communications from the wireless camera.

Results:

Video communication was poor in any environment we tested in. Our first test was in utilizing 2.4 GHz communication to transmit video. The problem with this was that our control communication, the XBee also utilized this same protocol. We then had to implement a 900 MHz video solution that proved to be less effective. Our proposed solution would be to utilize an onboard computer with our device and transmit video via WiFi communication.

5.3.0 Speed Test

This test will evaluate our speed on both improved and unimproved surfaces. For this test we will run our robot on pavement, grass, and sand. We will carefully section of a 40 foot stretch of ground over each medium. This area will then be sectioned into two 20 foot sections. The vehicle will then be driven straight down the track and at the first 20 foot marker, an evaluator with a stopwatch will record the time it takes the robot to reach the end of the track. The purpose of this is to let the vehicle reach a top speed before evaluating it. The speed can then be determined by dividing the distance traveled by the vehicle (20 ft) by the time it took to complete the section. Multiple trials will be run on each surface in order to obtain average values. It is important to conduct these tests on a full battery charge to avoid decreases in speed caused by low battery power.

Results:

The mean result based on four speed time trials was roughly 5.5 mph. This aligned perfectly to our specified 5 mph requirement.

5.4.0 Battery Life

Battery life will be tested under both full load and idle conditions. To test the battery life under full load conditions, the robot will be suspended above ground to prevent it from moving. We will then connect voltmeters to the batteries and run a custom program that will spin the robots wheels at full speed and turn on the robots camera. The time it takes for the batteries to be completely depleted will then be recorded. The same test will then be preformed again only without spinning the wheels, to determine the battery life under idling conditions. Finally, we will compare the results of the test to our requirements to validate our design.

Results:

For the control battery we observed a battery life of approximately 5 hours under a full load. Our control battery under an idle load proved inconclusive as battery life remained after 10-hours of constant observation. The drive battery lasted approximately 20 minutes under a full load, while idle load also proved to be inconclusive.

5.5.0 Durability

To test ARMORD's capability to sustain damage from high elevation and collision we will conduct a series of impact tests.

5.5.1 Drop Test

Before we can evaluate our vehicle, we need to find a control in which to base our tests around. For this we will use a scrap piece of silicon computer board, as this is what we are trying to prevent from breaking inside our vehicle. To test our control, we will outfit the piece of silicon with several Shockwatch G-Force indicator labels and drop it from a height of three stories. These labels are commonly used for packing insurance use by shipping companies. An image of the Shockwatch label can be seen in Figure . The labels come in varying g-force magnitudes and by outfitting the board with several values of labels, we will be able to gauge the force required to damage it. Once the force needed to break a silicon board is found, we can begin to outfit our vehicle with the same indicator labels. It is important in this test, to remove all electronic components from the robot, leaving only the frame and enclosure; after all we are testing whether it is safe for the electronics. The vehicle will then be dropped from the same height of 3 stories and the labels will be checked to see which ones have been set off from the fall. As long as the labels that broke during the control have remained intact, then the vehicle is safe for the electronics.

Figure - Shockwatch G-Force Indicator Labels of Varying Magnitude
(Figure printed with permissions by Wikipedia)

Results:

The drop test proved to be more difficult then initially anticipated. We were unable to produce any Shockwatch labels, as the cost proved to be too high. Instead, we preformed a series of supervised drops and observed damage from the impact. We found that because of ARMORDS increased weight, damage proved to be significant. On one of the drops, almost the entire weight of the device landed on a single axle, which snapped the motor in half.

5.5.2 Shock Test

The collision test will be performed in an almost identical fashion as the drop test. The only difference with this test is that instead of dropping the vehicle, we will plan on striking it with a blunt object to simulate running into walls and having debris dropped on top of it.

Results:

The shock test revealed that ARMORD was extremely well adapted at taking collisions and directed blows to its enclosure. We were able to strike the device several times with a hammer, which only caused minor abrasions to ARMORDs surface. We found that direct collisions with walls did nothing to harm the device or its internal collisions.

5.6.0 Software Test

In order to verify that our software components are working as expected. We can break the entire software system into the three main components, the iPhone application, the gateway application, and the microcontroller code. Each application has its own set of functions that it needs to do for the entire system to be considered working.

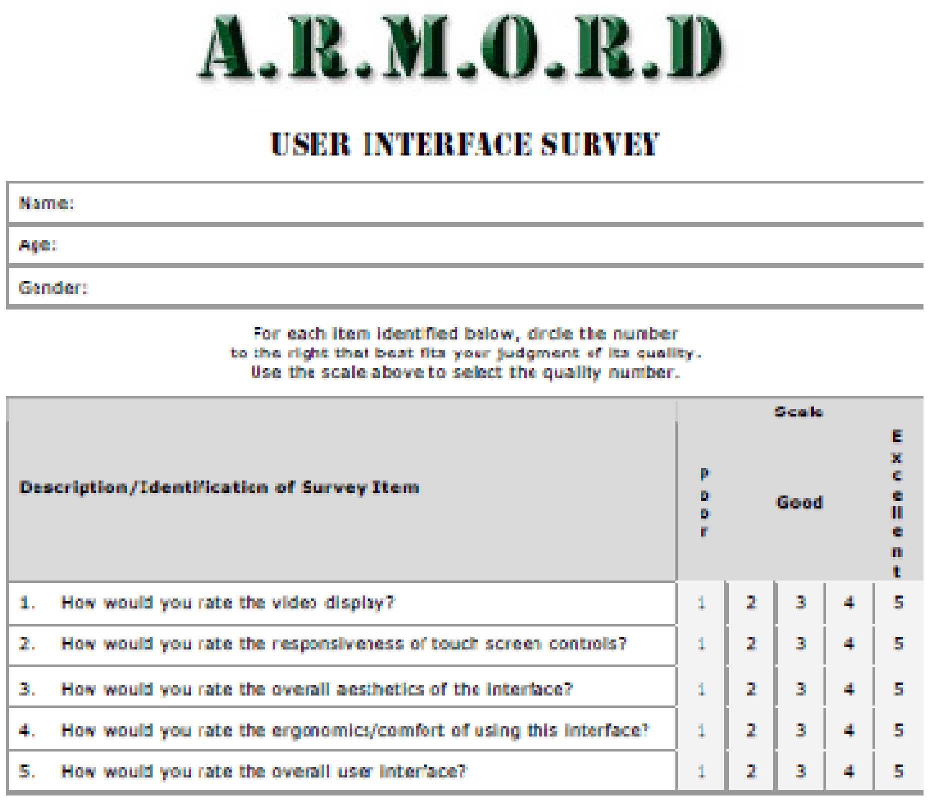
The iPhone application can be considered working once it completes its primary purpose of controlling the robot, displaying a video feed from the robot, and plotting the robots position on a map. By controlling the robot, we want the user to be able to use the touch-interface of the iPhone to move the robot in any direction desired. They should be able to control the direction of each set of wheels (left and right), the speed that each set is going, and effectively navigate the robot around with ease. To test the video feedback, we want the user to be able to see where what the robot sees in essentially real-time. The user should be able to navigate the robot around objects by using the video feed to see where the robot is and what objects are around it. To test the mapping functionality of the iPhone, we want the user to be able to bring up a map containing the current position of the robot and the current position of the user (based on the iPhone's location). The map should also display an approximate distance from the user to the robot.

The gateway application can be considered working once it successfully relays data to and from the iPhone and the XBee modules successfully. To test this we would first want to connect the XBee modules and the iPhone to the gateway application. Next we want to verify that when the user moves a slider control (to move the wheels), that the appropriate slider values are captured from the iPhone, and then relayed successfully to the attached XBee module. In addition to testing the control signals, the gateway application needs to be able to move GPS location data from the robot to the iPhone. To test this we would also need to have the XBee modules and the iPhone connected to the gateway app, then have the robot's GPS receiver send data across the microprocessor, to the XBee system, and then be captured by the gateway application. Once this data is captured, we can say the gateway app is working successfully by verifying the GPS location data is sent and captured by the iPhone.

The code on the microcontroller can be verified as successfully working once we verify that it constantly captures and transmits GPS and control (left wheel and right wheel) data. To test this, we would need to have the GPS module wired up the microcontroller, and verify that it captures the correct data, then transmits it to the XBee module successfully. This is also the same test case for the control data.

5.6.1 User Interface Test

This test will evaluate the ease of use for our user interface. To test this we will conduct a survey by sampling 20 random people on campus at the Student Union and ask them to rate the difficulty of controlling our robot with the iPhone, having no knowledge of the interface beforehand. The survey will include several questions regarding ease of use, responsiveness and ergonomics. The location of the student union was chosen because it provides a good mix of individuals from various backgrounds and professional fields. Based off this feedback we will be able to tweak our interface and control to better suit the "general population". Figure is a copy of the survey we intend to give to participants.



A.R.M.O.R.D

USER INTERFACE SURVEY

Name:	
Age:	
Gender:	

For each item identified below, circle the number to the right that best fits your judgment of its quality. Use the scale above to select the quality number.

Description/Identification of Survey Item	Scale				
	Poor		Good		Excellent
1. How would you rate the video display?	1	2	3	4	5
2. How would you rate the responsiveness of touch screen controls?	1	2	3	4	5
3. How would you rate the overall aesthetics of the interface?	1	2	3	4	5
4. How would you rate the ergonomics/comfort of using this interface?	1	2	3	4	5
5. How would you rate the overall user interface?	1	2	3	4	5

Figure - Interface Survey for Use in Testing Ease of Use
(Figure was created by our group)

Results:

ARMORDs user interface received mostly favorable reviews. The first item in the survey, the video display question, received the most negative marks because of its inability to maintain a constant video signal. This item received a 2 average score. For every other item in the survey we received almost perfect scores, due to the responsiveness and quality of the touch screen system.

5.7.0 GPS Positioning Test

We will use this test to validate that the onboard GPS module is providing an accurate reading. To perform this test we will drive to several points around the city of Orlando with our robot. Using our GPS interface we will verify that each actual location of our robot matches closely with the virtual representation of our location. We expect to see at least a few meter discrepancy due to the low resolution of consumer grade GPS.

Results:

To save on time, we instead went to several locations around the UCF campus to verify our GPS accuracy. We found that we were able to approximate ARMORDs location down to within 3 feet.

5.8.0 Cold Start Test

This test will evaluate our robots cold start time. To test this, we will leave the robot un-powered for a period of 12 hours. After this time, we will power the unit up and time how long the system takes to become fully functional in all aspects, including iPhone startup and communication initiation. It would be preferable to perform this test multiple times to obtain an average start time.

Results:

We found that the start time of the GPS varied between the environmental location of the GPS device. We observed almost instant start times in outdoor, open environments. In indoor, covered locations we found start times of 20 to 40 seconds.

5.9.0 Video Stream Delay

This test will evaluate the time delay present in our video stream. To perform this test, we will obtain two clocks running on exactly the same time. We will place one of these in front of our onboard camera that is streaming the video and another in front of the user watching the video from the iPhone. The difference in times between these two clocks can then be calculated thus obtaining our video delay. This test will be preformed both while our robot is stationary and again when it is moving at full speed. This will allow us to test the delay when other communications are going on. Again, the test will be preformed multiple times in order to obtain an average delay time.

Results:

ARMORD did not experience any delay in its video transmission that could be easily detected by the human eye.

5.10.0 Ventilation and Cooling Test

This test will validate that the internal temperatures within our robot are within the recommended operating temperatures of our components. Before we can perform this test, we need to know what these recommended operating temperatures are, thus Table was created. This table lists several key components of our system and their recommended temperatures. From this table we can find the components that have the lowest temperature values in either direction. These values will be used as our testing criteria; if our tests indicate internal temperatures greater (in either direction) than these testing criteria, we will then have to re-design our cooling capabilities.

Component	Recommended Operating Temperature
IG32P DC Motor	-10˚ C to 60˚ C
DS18B20 Temperature Sensor	-55˚ C to 125˚ C
XBee Pro Wireless Module	-40˚ C to 85˚ C
PIC18F452 Microcontroller	-40˚ C to 125˚ C
Wireless Mini Camera	0 ˚C to 40˚ C
NiMH DC Batteries	-20˚ C to 60˚ C
Sabertooth Motor Controller	-40˚ C to 85˚ C
Falcom FSA03 GPS	-40˚ C to 85˚ C

Table - Recommended Operating Temperatures of Components
(Table created by our group)

Based off this table, the shortest amount of temperature tolerance is our onboard camera with an operating temperature range of 0˚to 40˚ Celsius. These values will serve as the lowest acceptable values for our test. If the temperature is higher than these values, our camera could start functioning incorrectly. We will use our embedded temperature sensors and algorithm to measure the current temperature inside our robot during different scenarios. The first scenario will be to measure the temperature while the robot is idling and the ventilation is turned on. This scenario will be preformed both outside and inside. The next scenario will be to measure the temperature while the robot is moving forward at high speed with the ventilation turned on. We will also perform this scenario indoors and out. Finally, we will test the worst case scenario of the robot idling with the ventilation system turned off. If any of these tests fall below 0˚ C or above 40˚ C, then we will need to look at alternative ways to cool or heat sink our system.

6.0.0 Budget and Financing Analysis

Part Name	Quantity	Price	Total Cost
12V 2200mAHr NIMH	1	$23.90	$23.90
12V 4000mAHr NIMH	1	$53.60	$53.60
24V 4500mAHr NIMH	2	$124.80	$249.60
Falcom FSA03 GPS	1	$59.95	$59.95
24V 252 RPM Geared Motor	5	$44.90	$224.50
IPhone Grip	1	$25.50	$25.50
Sabertooth 10A Motor Controller	2	$79.99	$159.98
7 in. LCD	1	$79.99	$79.99
2.4GHz Mini Wireless Camera	2	$42.50	$85.00
900 MHz Mini Wireless Camera	1	$60.00	$60.00
Truck Bed Liner	2	$9.95	$19.90
XBee Pro	3	$37.95	$113.85
XBee USB Explorer	2	$24.95	$49.90
XBee Breakout Board	1	$9.95	$9.95
PIC18F4520	3	$4.50	$13.50
Misc(Body, Copper Clad, Etc.)	NA	$400.00	$400.00
Developmental Tools	NA	$350.00	$350.00
Shipping/Handling	NA	$300.00	$300.00
Total			**$2429.14**

Table – Budget

After thoroughly researching the many different areas of our project, we have decided on the parts that we will be using for our end product which are listed below. In our budget we show what we projected the cost of a product to be and the actual cost of that product. Table below shows the breakdown of each item.

We have included Figure below to visually show the differences between the actual and estimated costs. For most of the items we needed we estimated very accurately, our final estimated price was $662.00 and the actual end price was $739.00. The only difference in these numbers that could change is that when we test out our robot, something breaks which we will have to replace.

Even though the price of this robot came to $739.00 this is really nothing compared to what most robots cost today. The Dragon Runner robot cost over $32.000 dollars, yes they had a lot more features to their robot. Our robot still has a lot of the same of the same features as the Dragon Runner. I think we ffectively created and used our budged wisely.

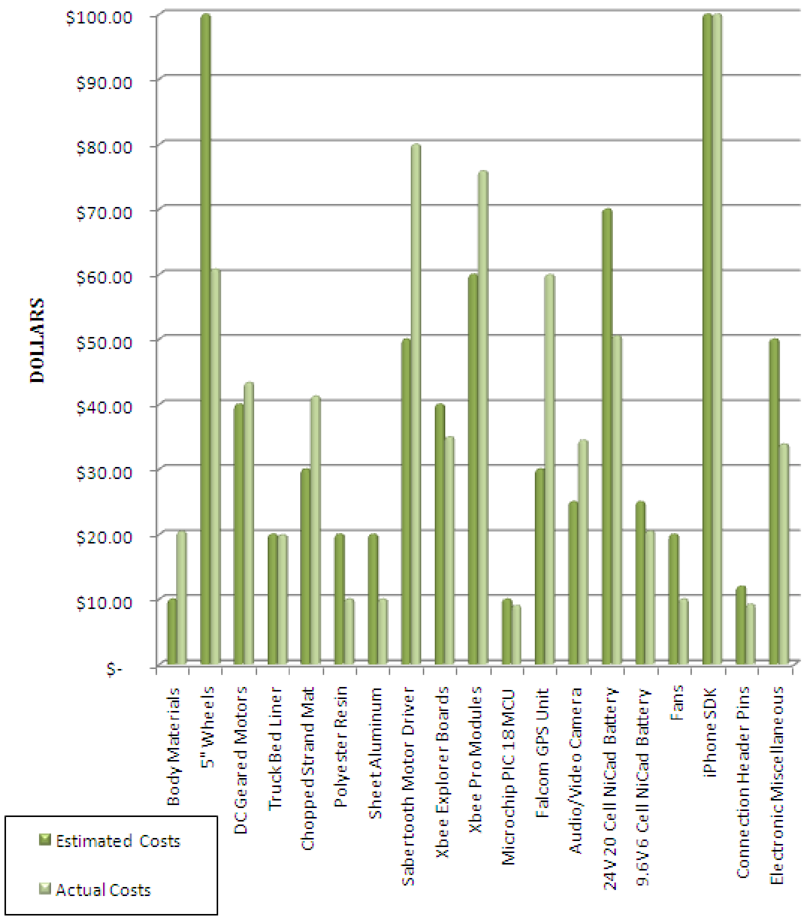


Figure - *Budget Analysis*

We sent many email's about being financed in our project, but either we ended up getting no response back or they decided no longer to finance the projects. We will be continuing to look for sponsors for our project, so that we can reimburse ourselves for the out of pocket money we have used.

We decided before we found any sponsors that the worst case scenario would be that we would split the costs of our project up three ways, so that it would be fair. This also played a factor in to how robust and how many features our robot would have and if we were all financially well off; we would have liked to put more money into this project to be able to have more features.

If we were to end up selling our project, we definitely think it is highly profitable. From the Dragon Runner costing over $32,000 dollars and we have built ours for under $1,000 dollars, we believe that an asking price of $5000 per unit would be acceptable. That price would include 200 hours of work in both software engineering and hardware engineering. Due to our robot being cheaper than Dragon Runner we definitely think we would have a market for people not wanting to spend that type of money. We also believe that due to our technology being so cheap that a defense company would either buy our invention or want to pay us to research other possibilities with our robot. We rather as a group do the later, of the two possibilities.

7.0.0 Final Design Review

Overall we liked our design in many aspects, but if we had a chance to go back and change some things we definitely would. When we started this project we went with really no knowledge on many of the different subsections of our robot.

For the hardware side of our robot the first design that would have been changed would definitely be the board we used. We used a microcontroller because we wanted to show what we have learned in our classes throughout the years. Alternatives would have been using boards such as the Arduino, which has multiple connections right on the board itself. This would of allowed us to get rid of the Zigbee controllers, microcontroller, motor controllers, and even the camera that had its own transmitter. The Arduino has USB built in so you can connect USB cameras and USB network adapters to it. You can even run Linux on Arduino which would have helped us stream live video via server type application straight from the robot. This board costs around $150.00 but the upside of this board is amazing, and this is why many robotic applications today use it. We also would of like to have someone custom make our outer and inner shell of our robot. For this we wish we could of hired a company to make a custom built shell, that we would not of had to worry about the strength of the shell and could of moved on with our project.

The software side wasn't as bad as the hardware; the only improvement we wish could have added was how we controlled the robot. The iPhone SDK was not as easy to stream video to unlike streaming to a computer. Windows has portable handheld computers that consumers use as laptops; they are extremely small which would have been the perfect solution for us. It would be able to run a simple windows application, which we are very proficient at compared to using the iPhone SDK.

8.0.0 Extensions

We are planning on completing our project early, and wanted to have goals in case that happened. These goals were to add more features to the robot to increase performance and usage; this was if time and money were available.

The first extension, if time allows, would be adding an IR sensor/camera. This IR sensor would be able to detect people up to 20 feet away. Once a person has been recognized our robot will alert the user and then go to a quiet mode. In quiet mode, the robot only allows the motors to rotate so much to make the minimum amount of noise. The camera portion will allow us to see in the night without having any lights. This would be a very useful if the robot was on reconnaissance mission and didn't want to be spotted by having lights on the front of it.

Another extension would be to attach a silenced weapon to the front of the robot. If the robot was being controlled by military on a mission that someone possibly could lose their life, it would be better to have a robot go in that mission and not have to risk a human life. Their many types of robot's that do this type of action already, but due to the size of our robot and quickness it would make a great add on to ARMORD. We would have it to the barrel of the gun could be able rotated in a 45 degree angle.

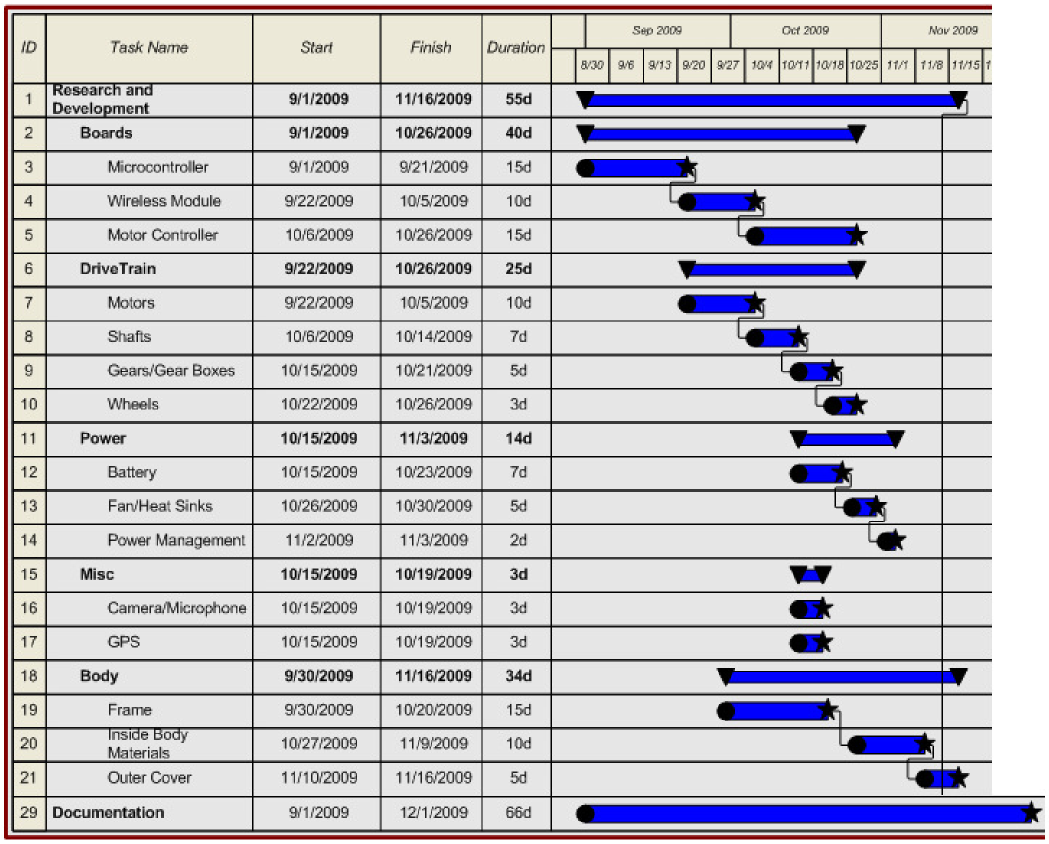
We also wanted to be able to add three more cameras to our robot. We were going to first install one on the back of the robot, so that we could be watching the front and the back of the robot with videos. Then we wanted to install two cameras, one on the top of our robot and one on the bottom that could be angled so that we could see what’s going on around the robot other than eye level.

9.0.0 Project Milestones

This section of our documentation is a breakdown of each semester's milestone. We set our milestones so that we would accomplish enough each term that we wouldn't have to do anything last minute.

9.1.0 Senior Design 1

During our Senior Design 1 term, we had two goals. The first goal was to research everything we were going to need to build our robot. We got on research the day we decided what our project was going to be. The second goal was to have our paper done before the due date, so we could go back several times and check our spelling and grammar. We easily completed this with hard work all term, and Figure below shows our Gantt chart.



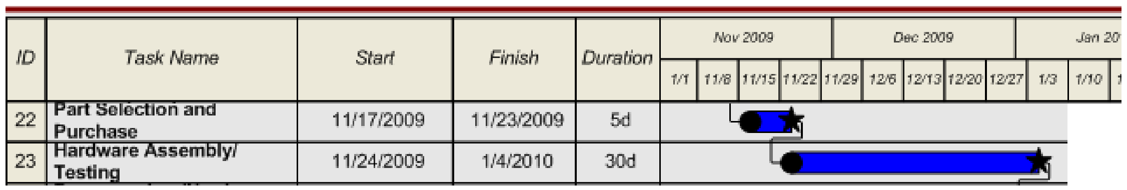
ID	Task Name	Start	Finish	Duration
1	**Research and Development**	**9/1/2009**	**11/16/2009**	**55d**
2	**Boards**	**9/1/2009**	**10/26/2009**	**40d**
3	Microcontroller	9/1/2009	9/21/2009	15d
4	Wireless Module	9/22/2009	10/5/2009	10d
5	Motor Controller	10/6/2009	10/26/2009	15d
6	**DriveTrain**	**9/22/2009**	**10/26/2009**	**25d**
7	Motors	9/22/2009	10/5/2009	10d
8	Shafts	10/6/2009	10/14/2009	7d
9	Gears/Gear Boxes	10/15/2009	10/21/2009	5d
10	Wheels	10/22/2009	10/26/2009	3d
11	**Power**	**10/15/2009**	**11/3/2009**	**14d**
12	Battery	10/15/2009	10/23/2009	7d
13	Fan/Heat Sinks	10/26/2009	10/30/2009	5d
14	Power Management	11/2/2009	11/3/2009	2d
15	**Misc**	**10/15/2009**	**10/19/2009**	**3d**
16	Camera/Microphone	10/15/2009	10/19/2009	3d
17	GPS	10/15/2009	10/19/2009	3d
18	**Body**	**9/30/2009**	**11/16/2009**	**34d**
19	Frame	9/30/2009	10/20/2009	15d
20	Inside Body Materials	10/27/2009	11/9/2009	10d
21	Outer Cover	11/10/2009	11/16/2009	5d
29	**Documentation**	9/1/2009	12/1/2009	66d

Figure - Senior Deisgn 1 Gantt Chart

As you can see we were researching multiple topics at a time, which was possible because we broke up the different parts and assigned them to different team members. This helped moved the process of finding the correct parts faster. We also had meetings in order to update team members on what was going on with each section. We did so that all of us would be skilled in each area.

9.2.0 Winter Break

During our winter break we did not want to waste any time because, we all knew our last semester before we graduate was going to be rough. For this reason we continued our project as if the term didn't end. During winter break we set goals for hardware purchasing and for assembly. Our winter break Gantt chart is shown below in Figure .

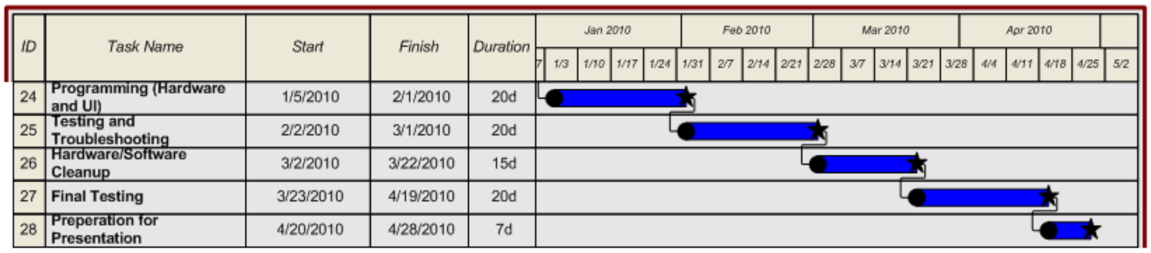


ID	Task Name	Start	Finish	Duration
22	Part Selection and Purchase	11/17/2009	11/23/2009	5d
23	Hardware Assembly/ Testing	11/24/2009	1/4/2010	30d

Figure - Winter Break Gantt Chart

9.3.0 Senior Design 2

For our last term, all we had left to do for the design was program the hardware and create the software side of our project. Our remaining tasks were to Test/Troubleshoot, Hardware/Software Cleanup, Final Testing, and Final Presentation. This seems like a lot of tasks to do but, since we made it so we have a lot of time on each of these steps to make sure we finish every part of this project correctly. Our Senior Design 2 Gantt chart is shown below in Figure .



ID	Task Name	Start	Finish	Duration
24	Programming (Hardware and UI)	1/5/2010	2/1/2010	20d
25	Testing and Troubleshooting	2/2/2010	3/1/2010	20d
26	Hardware/Software Cleanup	3/2/2010	3/22/2010	15d
27	Final Testing	3/23/2010	4/19/2010	20d
28	Preperation for Presentation	4/20/2010	4/28/2010	7d

Figure - Senior Design 2 Gantt Chart

9.4.0 Milestone Conclusion

Throughout senior design we knew when things had to get done, with our Gantt chart. At first, we thought we are able to just keep dates in our heads and because we saw our team members daily we would be able to push each other. This didn't work and after a week we realized we need some way to organize ourselves as a group, so we decided to do what they have taught us in our Project for Object Oriented Software class, which was make a Gantt chart.

This helped us organize time effectively and made sure we kept on track otherwise we would risk falling behind. I believe that our Gantt chart gave us that a decent enough amount of time to complete each task, and due to the fact we gave each section so much time we actually had more time to finish other tasks before they even started on our Gantt chart. We definitely now understand why you need to manage your time with charts and schedules, otherwise you have to rely on your knowledge of remembering everything, which usually doesn't work.

10.0.0 Conclusion

Our group knew that this project wasn't going to be easy when we picked it, but one thing we did know is that we would enjoy it and that is what mattered to us the most. To our group robotics is fun and interesting and was a big part of why we couldn't say no to the opportunity of building our own robot. ARMORD was quite an amazing project for us because it included mechanical, electrical, and computer engineering aspects. We used all the skills we have learned from our courses as well as additional material we learned from the Internet, to get our project to where it needed to be. When we didn't know about something we researched it fully to be able to understand the process, which made it easy to design later on. Our final design definitely was what we imagined when we first started this project, so we were happy with the fact we could create something that we pictured in our heads.

Most people say that you shouldn't join a group of your friends in Senior Design because you might end up losing them. Well, we three proved that theory wrong; all it takes is for everyone to be understanding and know how to listen and not just talk. Most people want to think they are right all the time but the problem you need to be able to listen to someone else and be ok with the fact you won't always be right or have the best idea. In our case, we had three people who knew how to be understanding and knew how to listen. We always didn't see eye to eye but that is normal, and the most important part of that is that we could sit down as group members and come to a conclusion when needed. This is what made us a great group of engineers, the fact we had great communication between each other.

Looking back on our project we really don't have any regrets. There are definitely more than a couple of things we wish we could have changed. This includes the amount of money we wish we could of put into this project, having a fourth group member in our team, and having a bit more time all around for our project. As for the money aspect of our project we really wish we could have had more money so that we could have added a lot of more features to our robot. Even though we work, we all have bills to pay and that comes first. Also, it would have been interesting to see what our project would have turned out if we had one more engineer in our group, which we would have preferred to be mechanical due to the dynamic chassis of our robot. We definitely could have used more time for everything on our project but in between work and school there is not much we could all do, but with the time we had we definitely used it wisely and efficiently.

We definitely learned a lot throughout this project, whether it was teamwork or just common robotics knowledge. We all got a reality of what it takes to be able to work fundamentally in a team and show both that you are able to lead and follow during appropriate times. We also learned that we don't know much regarding real life applications, which as the years go on and our experience level grows we will be able to more complex projects than our robot. We all got to find out what is behind the math that we learn in our electronics classes and what it takes to create a complicated circuit. We also learned that programming in class and programming in real world applications definitely takes more planning and research.

I think we could all agree on the fact that our group definitely has a lot to bring to the table and we all hope that in the future we will be able to work together somehow, whether day to day or at least helping each other solving problems in future projects. In conclusion, even though our robot wasn't perfect we learned more in this experience, than from most of our other courses combined.

11.0.0 User's Manual

System Requirements:

Gateway with Bonjour Setup Installed

Available USB port

Laptop with WiFi Card

FTDI Driver Installed

AD-HOC Network Setup

Run Procedures:

1. Flip power toggle switch to "ON" position
2. Connect iPhone to Ad-HOC network
3. Connect XBee USB to Gateway
4. Start Gateway application
5. Select correct COM port from Gateway application
6. Click "Connect" on Gateway application
7. Click "Connect" on the iPhone application
8. ARMORD is now controllable via iPhone interface slider controls
9. Additional ARMORD information can be accessed by double tapping GPS screen

Power-Down Procedures:

1. Turn ARMORD Power Switch to "Off"
2. Hit "Home" Button on iPhone to exit ARMORD Application
3. Exit Gateway Application

Appendix A: Works Cited

"HTTP Streaming Architecture." Apple Inc. 17 Nov 2009.<http://developer.apple.com/iphone/library/documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/HTTPStreamingArchitecture/HTTPStreamingArchitecture.html#//apple_ref/doc/uid/TP40008332-CH101-SW2>

"Interprocess Communications." Microsoft Corporation. 12 Nov 2009. <http://msdn.microsoft.com/en-us/library/aa365574(VS.85).aspx>

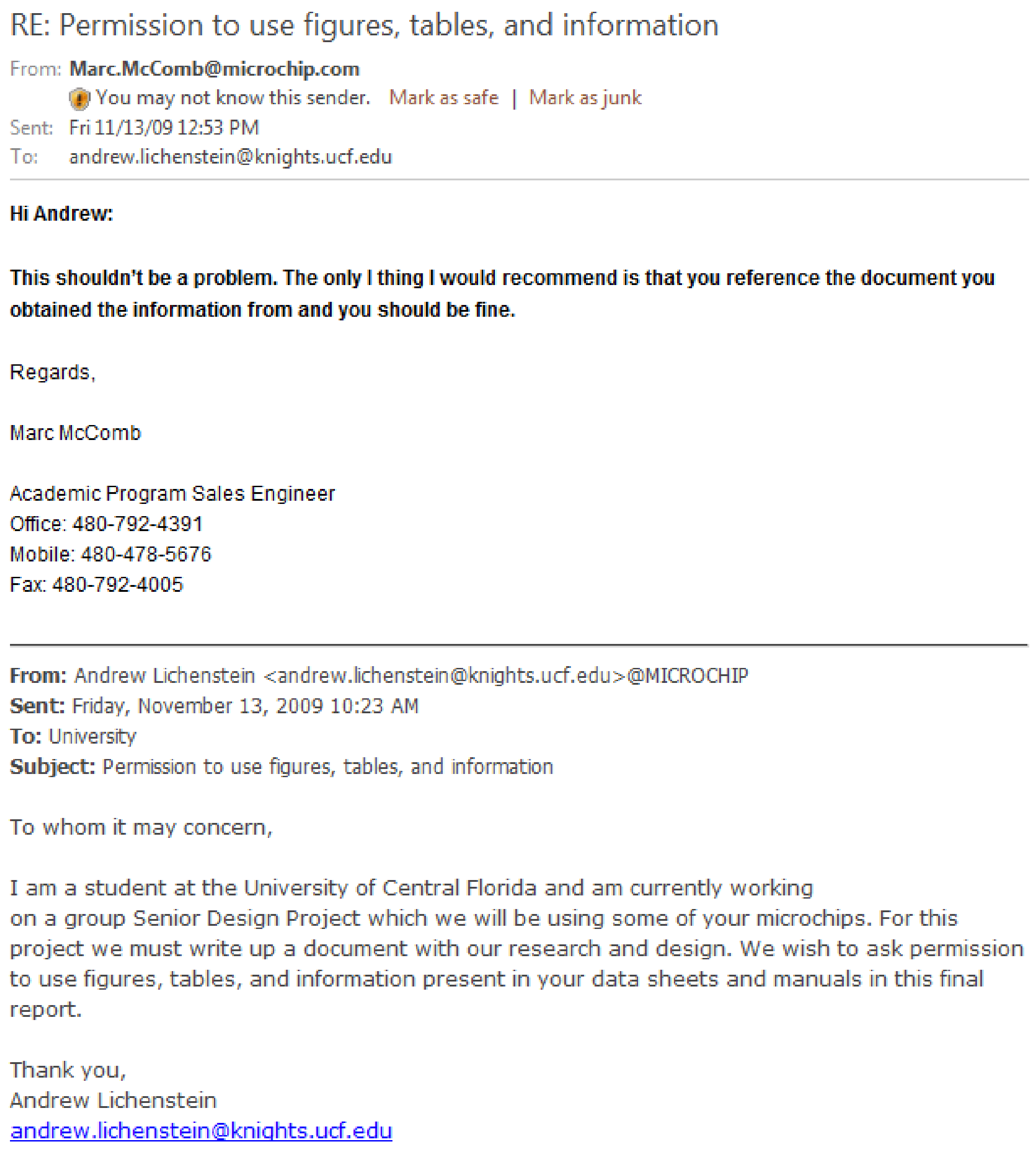
"Files and Networking." Apple Inc. 19 Oct 2009. <http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/FilesandNetworking/FilesandNetworking.html#//apple_ref/doc/uid/TP40007072-CH21-SW6>

"UIAcceleration Class Reference." Apple Inc. 27 May 2008. <http://developer.apple.com/iPhone/library/documentation/UIKit/Reference/UIAcceleration_Class/Reference/UIAcceleration.html>

Clark, Dennis and Michael Owings. Building Robot Drive Trains. New York: McGraw-Hill, 2003

Singer, P.W. Wired for War: The Robotics Revolution and Conflict in the 21st Century. New York: Penguin Press. 2009

Appendix B: Literature Permissions



RE: Permission to use figures, tables, and information

From: **Marc.McComb@microchip.com**
You may not know this sender. Mark as safe | Mark as junk
Sent: Fri 11/13/09 12:53 PM
To: andrew.lichenstein@knights.ucf.edu

Hi Andrew:

This shouldn't be a problem. The only I thing I would recommend is that you reference the document you obtained the information from and you should be fine.

Regards,

Marc McComb

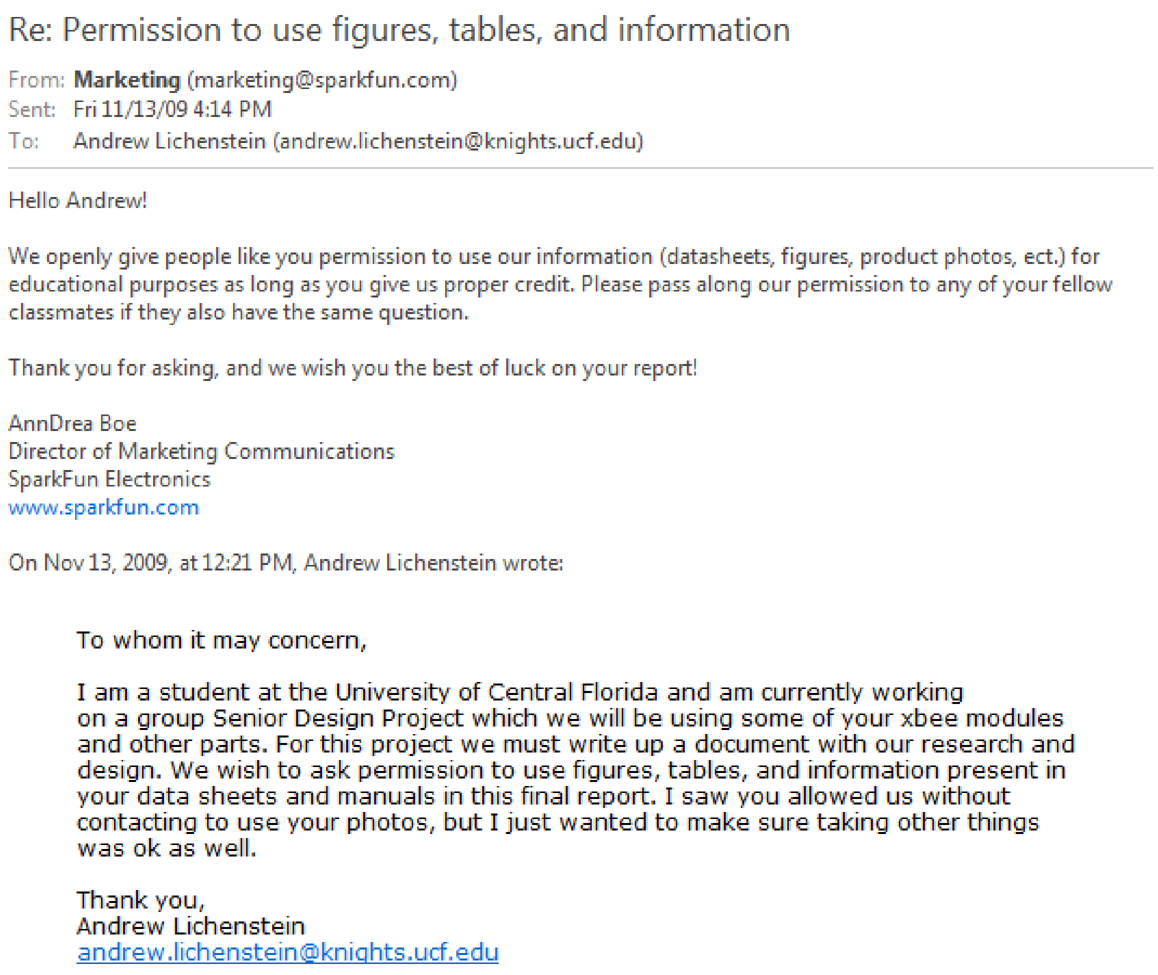
Academic Program Sales Engineer
Office: 480-792-4391
Mobile: 480-478-5676
Fax: 480-792-4005

From: Andrew Lichenstein <andrew.lichenstein@knights.ucf.edu>@MICROCHIP
Sent: Friday, November 13, 2009 10:23 AM
To: University
Subject: Permission to use figures, tables, and information

To whom it may concern,

I am a student at the University of Central Florida and am currently working
on a group Senior Design Project which we will be using some of your microchips. For this project we must write up a document with our research and design. We wish to ask permission to use figures, tables, and information present in your data sheets and manuals in this final report.

Thank you,
Andrew Lichenstein
andrew.lichenstein@knights.ucf.edu



Re: Permission to use figures, tables, and information

From: **Marketing** (marketing@sparkfun.com)
Sent: Fri 11/13/09 4:14 PM
To: Andrew Lichenstein (andrew.lichenstein@knights.ucf.edu)

Hello Andrew!

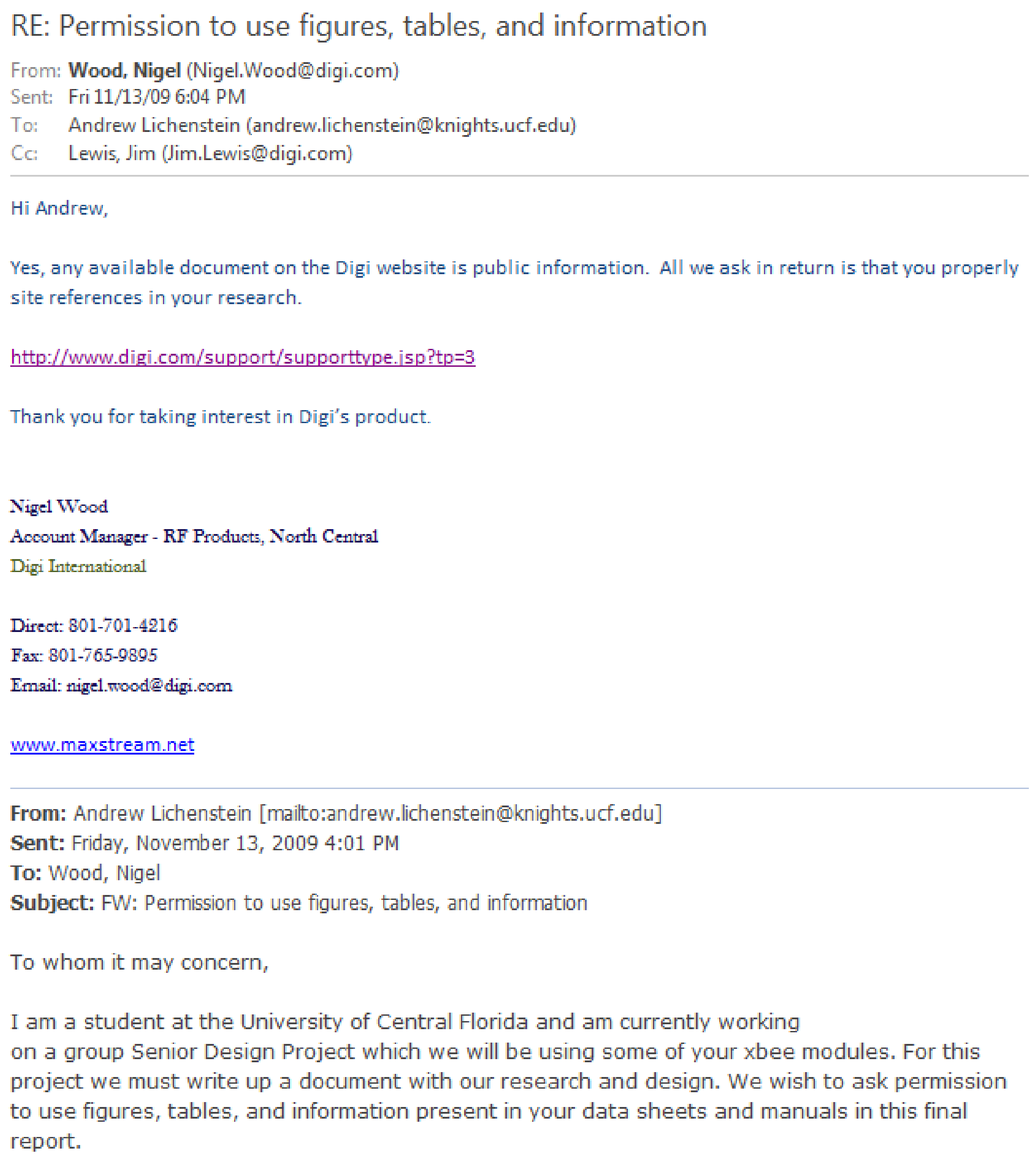
We openly give people like you permission to use our information (datasheets, figures, product photos, ect.) for educational purposes as long as you give us proper credit. Please pass along our permission to any of your fellow classmates if they also have the same question.

Thank you for asking, and we wish you the best of luck on your report!

AnnDrea Boe
Director of Marketing Communications
SparkFun Electronics
www.sparkfun.com

On Nov 13, 2009, at 12:21 PM, Andrew Lichenstein wrote:

> To whom it may concern,
>
> I am a student at the University of Central Florida and am currently working on a group Senior Design Project which we will be using some of your xbee modules and other parts. For this project we must write up a document with our research and design. We wish to ask permission to use figures, tables, and information present in your data sheets and manuals in this final report. I saw you allowed us without contacting to use your photos, but I just wanted to make sure taking other things was ok as well.
>
> Thank you,
> Andrew Lichenstein
> andrew.lichenstein@knights.ucf.edu



RE: Permission to use figures, tables, and information

From: **Wood, Nigel** (Nigel.Wood@digi.com)
Sent: Fri 11/13/09 6:04 PM
To: Andrew Lichenstein (andrew.lichenstein@knights.ucf.edu)
Cc: Lewis, Jim (Jim.Lewis@digi.com)

Hi Andrew,

Yes, any available document on the Digi website is public information. All we ask in return is that you properly site references in your research.

http://www.digi.com/support/supporttype.jsp?tp=3

Thank you for taking interest in Digi's product.

Nigel Wood
Account Manager - RF Products, North Central
Digi International

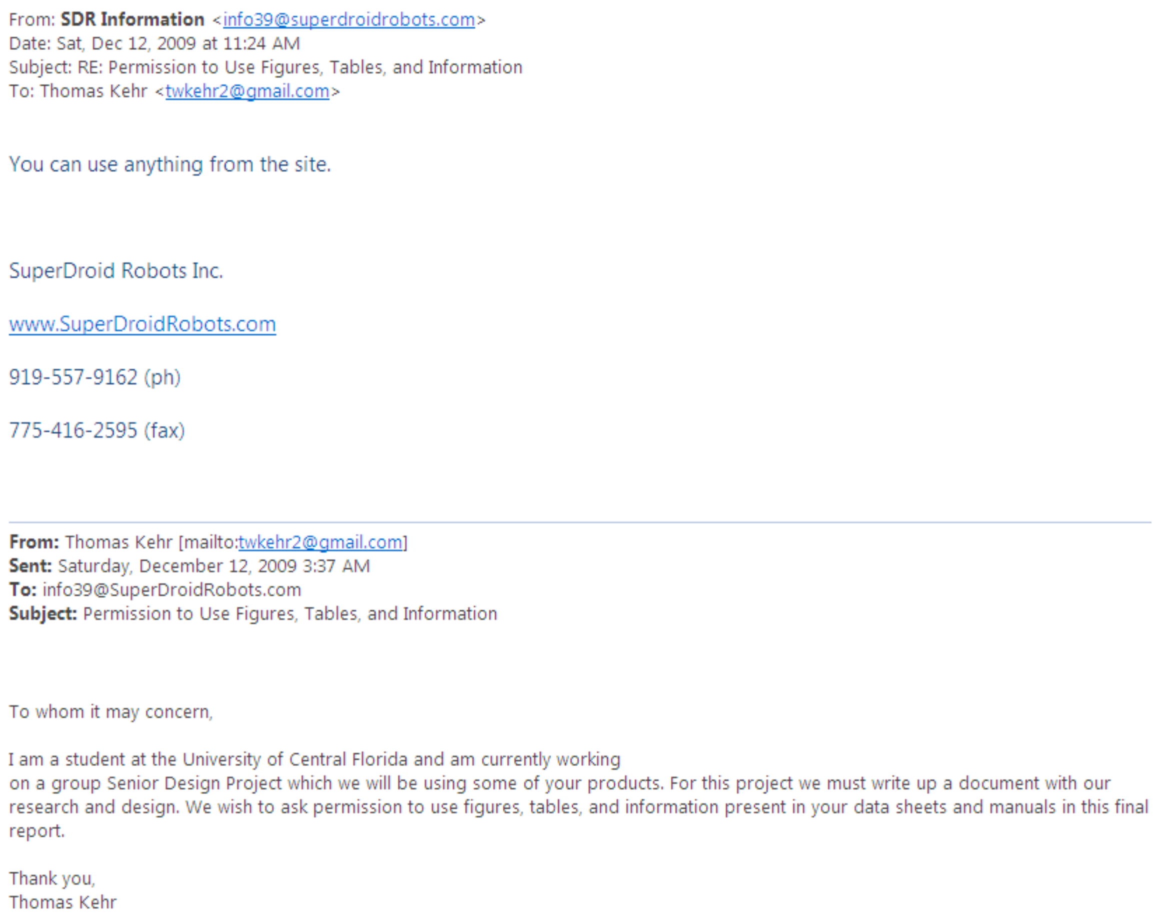
Direct: 801-701-4216
Fax: 801-765-9895
Email: nigel.wood@digi.com

www.maxstream.net

From: Andrew Lichenstein [mailto:andrew.lichenstein@knights.ucf.edu]
Sent: Friday, November 13, 2009 4:01 PM
To: Wood, Nigel
Subject: FW: Permission to use figures, tables, and information

To whom it may concern,

I am a student at the University of Central Florida and am currently working
on a group Senior Design Project which we will be using some of your xbee modules. For this project we must write up a document with our research and design. We wish to ask permission to use figures, tables, and information present in your data sheets and manuals in this final report.



From: **SDR Information** <info39@superdroidrobots.com>
Date: Sat, Dec 12, 2009 at 11:24 AM
Subject: RE: Permission to Use Figures, Tables, and Information
To: Thomas Kehr <twkehr2@gmail.com>

You can use anything from the site.

SuperDroid Robots Inc.

www.SuperDroidRobots.com

919-557-9162 (ph)

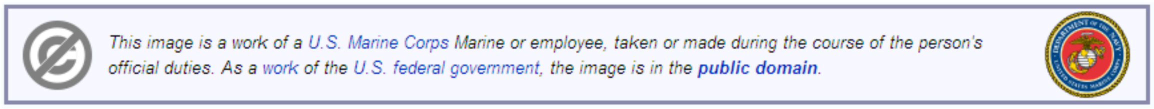
775-416-2595 (fax)

From: Thomas Kehr [mailto:twkehr2@gmail.com]
Sent: Saturday, December 12, 2009 3:37 AM
To: info39@SuperDroidRobots.com
Subject: Permission to Use Figures, Tables, and Information

To whom it may concern,

I am a student at the University of Central Florida and am currently working
on a group Senior Design Project which we will be using some of your products. For this project we must write up a document with our research and design. We wish to ask permission to use figures, tables, and information present in your data sheets and manuals in this final report.

Thank you,
Thomas Kehr



This image is a work of a U.S. Marine Corps Marine or employee, taken or made during the course of the person's official duties. As a work of the U.S. federal government, the image is in the ***public domain****.*