



UNIVERSITY OF CENTRAL FLORIDA

College of Engineering and Computer Science

Knight's Intelligent Reconnaissance Copter - KIRC



**EEL 4915 – Senior Design II
FINAL DOCUMENTATION
Spring 2014
Group #14**

**Nathaniel Cain
James Donegan
James Gregory
Wade Henderson**

Table of Contents

1. Introduction and Executive Summary.....	1
2. Project Description.....	2
2.1. Motivation & Sponsorship.....	2
2.2. Goals & Objectives.....	3
2.3. Requirements & Specifications.....	6
3. Research & Preliminary Design.....	8
3.1. Similar Projects & Products.....	8
3.2. Relevant Technology & Strategic Components.....	10
3.3. Specific Components and Decision Factors.....	13
4. Design Summary.....	22
4.1. Hardware Operation.....	22
4.2. Software Operation.....	30
5. Detailed System Design and Analysis.....	34
5.1. Detailed Block Diagrams.....	34
5.2. Flight Computer Software.....	42
5.3. Flight Stability Control System.....	54
5.4. Autonomous Navigation & Navigation Computer Software.....	63
5.5. Imagery & Network Architecture.....	72
5.6. Electric Design & Schematic.....	77
5.7. Ground Station Overview.....	84
5.8. Physical Design & Layout.....	89
6. Prototype Construction & Coding.....	94
6.1. Parts Acquisition.....	94
6.2. PCB Design and Assembly.....	96
6.3. Parts Integration.....	100
7. Prototype Testing and Evaluation.....	104
7.1. Microcontroller RTOS and Peripheral Driver Testing.....	104
7.2. Filtering Algorithms & Control Loops Testing.....	109
8. Operation Manual.....	115
8.1. Manual Flight.....	115
8.2. Using the Ground Station.....	117
9. Administrative Content.....	119
9.1. Milestone Discussion.....	119
9.2. Budget and Finance.....	123
9.3. Team Organization.....	126
Appendices.....	128
Appendix A – References and Other Projects Researched.....	128
Appendix B – Copyright Permissions.....	129
Appendix C – Datasheets Referenced.....	134

1. Introduction and Executive Summary

In this document, the design, analysis, operation, and testing results are detailed for the Knight's Intelligent Reconnaissance Copter (KIRC) project. In the last section, information about the team, the sponsors and the budget are given in detail. Copyright permissions, references, and the datasheets referenced are also included within this document in the appendices section.

KIRC is a UCF Senior Design project unofficially sponsored by the National Aeronautics and Space Administration (NASA) that includes two quadcopters that are able to provide imagery and telemetry to a ground station. This project was intended to test and evaluate the up and coming Delay Tolerant Networking (DTN) protocol for digital communication networks. DTN is a virtual networking protocol that excels in areas where communication networks can tend to have long delays or disruptions. The goal is to test the protocol in an active field setting in order to gain more experience with it as well as provide two quadcopters for later missions. Currently, the quadcopters developed in this project might possibly be used during some upcoming NASA missions such as the Pavilion Lake Research Project, in Canada.

The common goal of this project is to provide two quadcopters that can be reconfigurable on a mission-by-mission basis, test and evaluate DTN as a practical protocol for future missions, as well as develop a custom flight control system. As shown in figure 1 below. The two quadcopters as well as the ground station computer will form a three corner mesh network. With this arrangement, the goal is to illustrate DTN's ability to dynamically route messages and bundles of information to any member of the triangle either directly or indirectly as well as its ability to retransmit missing bundles.

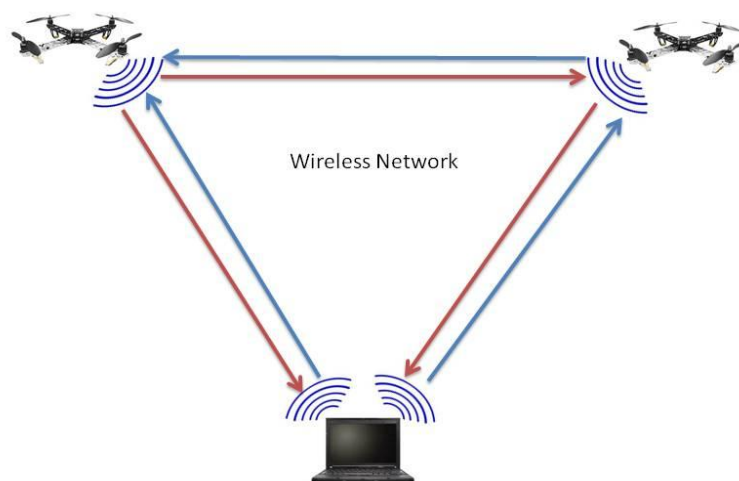


Figure 1: System Overview Diagram

2. Project Description

2.1. Motivation & Sponsorship

KIRC started as an idea during the summer of 2013 as a potential student project for a Co-op student working for the Center Planning and Development Office at NASA's Kennedy Space Center. It evolved into a senior design project as it grew more and more complex, and eventually a set of deliverables were developed. While this project is being partially funded by NASA, it remains an unofficial NASA project. The ideas behind the project were created to achieve multiple NASA goals, as well as provide future.

The main motivation behind this project was to test an experimental virtual communication protocol, DTN2, developed by University of California, Berkeley. This Delay Tolerant Networking, DTN2, architecture was designed for environments where communication is prone to delays and disruptions. During deep space missions, DTN2 is ideal for the long distances and the intermittent links between planets. For this reason, NASA is interested in implementing this software on much of its upcoming space missions, which is why they are motivated to learn more about it. Another main motivation, parallel to the first, behind this project is to provide two quadcopters for potential use in future NASA earth science missions. NASA decided to combine the two goals into one project, hence satisfying both goals in a single project. These quadcopters are able to patch together an image of an area cooperatively while transmitting telemetry over a wireless network using the DTN2 protocol. Control of these quadcopters is done manually, but it was developed such that the quadcopters could possibly be autonomous eventually. In order to keep the design of the project challenging, a custom flight control system was developed. This deliverable ended up being the most challenging to accomplish, but the team was able to get it done.

The team was provided unofficial sponsorship from NASA and an unofficial budget of about \$1000 for certain parts. With the unofficial budget provided, the team was tasked with the construction of 2 quadcopters with potential for full autonomous capability, a wireless DTN2 network, a custom developed flight control system, and the ability to send images to the ground for stitching. Over the course of senior design, the group researched, designed, built, and tested a system that performed the tasks set forth by NASA. This was difficult project at best, and required application in the areas of control, power, software development, operating systems, and computer networking.

One of the important details in the project was that the team needed to make two custom built quadcopters capable of autonomous flight. The reason behind making a custom built copter versus buying a commercial off the shelf (COTS) quadcopter was that the custom control would make the guidance portion of the project easier. With COTS items, many vendors will not make the software open source to the public. This complicates the design of the guidance control algorithm significantly, and also makes the software less predictable. Another reason is that COTS quadcopters parts tended to be quite expensive. With all this in mind, it was in the team's best interest to design a custom quadcopters for this project.

While the Pavilion Lake Research Project is the potential initial application of this project for NASA, it is also planned to be used in future projects as well. For this reason, the project system should be designed to be reconfigurable based on open ended mission needs. There are some possible long term goals of the project that include possible modular payload design, improved controller design, and autonomous development. This goal was not accomplished during the course of Senior Design, but will be a future application of the project.

Overall, this project was a fairly open ended one. It had several goals, which NASA and the team wanted achieve, all within a single project. The rest of this paper contains the design of the flight system developed and discuss the challenges associated with achieving NASA's goals. Having discussed NASA's sponsorship and motivations behind this project, the next sections will discuss the team's goals followed by the requirements.

2.2. Goals & Objectives

NASA's primary goal in this project was to have two autonomous quadcopters that can image an area based on commands entered over a wireless network with DTN2 as the network architecture. Based on these goals, the team devised sub-goals that were supposed to help them achieve these main goals. In this section, the goals and objectives of this project are outlined and how the team planned to meet these criteria. At the end of the section, a summary of these goals and objectives are provided in an outline form.

The primary goals and objectives of the project included the following: being lightweight, durable, adequate flight time, stable flight, ease of manual flight control, consistent autonomous flight, must use DTN2 application, two final quadcopters, and flexible software that can be easily be reconfigured based on mission requirements. These core objectives make up the description of how the quadcopters should perform as a polished product, and they are hard goals. This means that above all else, these are the goals that the team had to meet more than the others. At this point, it also may be useful to define what autonomous flight means in the context of this project. For this project, autonomous flight means that the finished quadcopters can fly to a user defined coordinate without need for a human pilot. It also means that the quadcopter can make changes to the flight plan, return home, or relay information without need for human intervention.

The quadcopters has several features that can be entered over the network via a user interface. There are commands that tell the copter to take a snapshot, or to clear its current database. Some of the larger commands are to input four coordinates and image the bounded area inside those coordinates, or to cooperative image an area. There are also safety commands such as return home, hover, or land quadcopter. The home location should be set every flight. All the commands are bound into a user interface program that is stored on the ground station computer. The parsed commands are then be sent to the quadcopters where they are interpreted as instructions.

The imagery portion of the project was designed to be done using a simple bread-crumbs trail GPS coordinate system. After the quadcopters' computers boot up, a user can input commands to autonomously image an area based on a four coordinate boundary. In order to do this, the plan was to have each quadcopter choose half of the boundary. From this, each quadcopter would make a flight path that would include locations to stop, hover, take a picture, and move on. The locations where the quadcopters would decide to stop depends on the horizon of the camera and the altitude of the quadcopter. Each of these locations can be calculated real-time based on these parameters.

The KIRC Project was split into three prototyping phases. Ultimately, the team planned to produce two completely autonomous reconnaissance quadcopters. The first phase was to create a manually controlled prototype that would allow the team to refine the flight stability algorithms before moving to an autonomous flight system. The second phase was to convert the quadcopter into KIRC, which used a PCB instead of a prototype board, and was capable of wireless telemetry. Phase three was to replicate KIRC, naming it SPOK (Secondary Partner of KIRC), and get them both autonomous. At the end of this project, the goal was to have both KIRC and SPOK working cooperatively to image an area. Unfortunately, due to time constraints, the autonomous portion of the project remains incomplete. This project did, however, achieve stable manual flight, telemetry transmission using DTN2, and two quadcopters with full flight computer PCBs were complete. The project demonstrated full capability of the DTN protocol as an operable networking protocol in real-time applications.

During phase 1, the plan was to achieve stable manual flight. This was done using a microcontroller as a means of digitally controlling the stability of the quadcopter. The microcontroller was using a Real Time Operating System (RTOS) in order to achieve a deterministic processing pattern. In order to make the quadcopter dynamically stable, the plan was to use a discrete-time proportional-integral-derivative (PID) loop to control its roll, pitch, and yaw. The inertial feedback of attitude data was achieved using a 10 degree of freedom (10DoF) IMU. Once this part of the project was done, a printed circuit board (PCB) was designed with all these previous parts integrated on one board.

During phase 2 of the project, the goal was to make the quadcopter integrated onto one PCB. The PCB was designed and tested with only a few issues. After the PCB was made, the navigation computers and new flight computers were mounted to the quadcopters. The navigation computer software was developed along with the ground station software for the user interface. The quadcopter was capable of taking images and displaying real time data on position, battery life, altitude, and state.

During phase 3 of the project, the primary objective was to replicate KIRC, naming it SPOK, and make them both autonomous. There was considerable programming of the autonomous features of both quadcopters during this phase, and unfortunately it wasn't able to get done. The goal by the end of the project was to have the quadcopters working cooperatively towards a common task, but that didn't happen.

Some of the objectives of this project were learning objectives. For instance, the microcontroller chosen was an ARM processor, and there had to be a Real Time

Operating System on it. Another example is that the flight control system had to be custom developed for this project. This was one of the challenging deliverables that prevented the team from completing the autonomous portion of the project.

In summary, the primary goals that the team wanted to achieve related to performance of the quadcopters were defined. The team also had smaller goals and objectives during this project, which were mainly driven by certain pieces of hardware or software that they wanted to implement for learning experience. Success in this project was determined by the ability for the team to meet the goals and objectives of this project by the Spring 2014 Senior Design II deadlines. A summary of the goals and objectives is listed below.

Goals & Objectives Summary:

- Lightweight and durable
- Ability to be manually controlled easily
- Dynamically stable in flight
- Flexibility of the software to be reconfigured on a mission-by-mission basis
- Consistent, accurate, and stable autonomous flight
- Two quadcopters that are identical to each other
- The user will also be able to address each copter individually for instructions to fly-to, image, or hover-at a coordinate
- Copters must individually and collectively be able to image a coordinate or area based on user input of four GPS coordinates representing a boundary to map
- Fail-safe commands to send to the quadcopter to land on the ground, or return home
- Ability to set home location easily
- The project should demonstrate the main features of DTN: data hopping over a mesh network, store and forward, and bundle handling.
- Use an imaging camera that is capable of reasonable clarity and is light and mobile enough to be mounted on a quadcopter
- Use an embedded Linux processor (Off the Shelf, not integrated as part of our PCB)
- Use an ARM processor for the flight stability computer
- Use DTN2 software on the Navigation and Ground computers for our Network Layer
- Use an image stitching software that can stitch together a composite image from multiple coordinate stamped images
- Use a Real Time Operating System (RTOS) on our ARM processor for deterministic behavior
- Implement digital PID loops to provide compensation to motors for control
- Develop or use an existing algorithm for filtering the noise from the IMU sensors
- Use PWM as the control signal going from the ARM processor to an Electronic Speed Controller (ESC) for each motor

2.3. Requirements & Specifications

Now that the major and minor goals and objectives have been defined, the requirements and specifications can also be defined. These include particular performance criteria, specific hardware, and software schemes that the team wished to use. A summary of all the parts, software, and performance requirements are given at the end of this section.

Performance of the quadcopter flight system is defined by the flight time per charge, durability of the system, maximum flight altitude, rate of climb, cruising speed, and wind requirements. The team wanted to aim for a reasonable flight time of 10-15 minute intervals before having to recharge. NASA did not provide any specific performance requirement for this project, but the team had some requirements that wanted to meet.

During flight, the quadcopters had to be durable enough to withstand dropping on the ground from a small height of about 3 to 4 feet. While being able to withstand a large drop would be desirable, it is not realistic within the design constraints of the project. The quadcopter's climb rate and cruising speed needed to be adequate enough that the navigation of a large area could be done without too many charging waits. The quadcopter had to be able to fly in light winds of 0-5MPH and still be stable enough for autonomous navigation. The maximum flight altitude had to be around 100 feet from ground level in order to provide adequate imaging range. All these requirements make up the performance characteristics for the quadcopters in our project. While these characteristics are important, some of these requirements are more important than others. The main hard requirements are the flight time, flight altitude, and wind requirements. All the other performance requirements are less critical as long as they appear to be adequate.

For this project, the team had certain requirements on the parts used as a result of the NASA part of the project (for a list of reasons for each part, see Research and Preliminary Design in the next chapter). The reason behind using some of these parts can be a requirement from NASA, or the team's requirement in order to gain more experience for example. NASA has the requirement that the team uses an embedded Linux platform as the navigation/imagery computer. This requirement is for two reasons, one being that the DTN2 software (required software, see next paragraph) only runs on Linux, and the other being that they want that part of the hardware to be interchangeable. Other hardware requirements include the use of a digital camera on board the quadcopter that interfaces to the embedded Linux computer. This piece of hardware was used for the imagery portion of the project, and was connected to the embedded Linux platform directly. One of the hardware requirements that came from an educational standpoint is the requirement to use a separate ARM processor for the flight stability and sensor processing. The team wishes to learn more about how to program on an ARM processor in order to gain more educational experience. The team also wishes to integrate the ARM processor, sensors, and support hardware onto a single printed circuit board (PCB).

Hard requirements for the software have also been defined. The team has certain software that they are required to use, or that they wish to use. NASA's software requirement is that the team uses DTN2 as the virtual network layer of our digital network. This

software will also be installed on the quadcopter's navigation computer as well as the ground station's laptop computer. NASA also requires that we find or develop a software the stitches coordinate stamped images into a large composite image for the imagery part of the project. As part of the team requirements, a Real Time Operating System (RTOS) must be used on the ARM processor. A digital control flight stability algorithm based on a discrete PID design was implemented, and a filtering scheme to reduce the sensor noise.

In conclusion, there were many requirements from both NASA and the team. Some of the requirements were hard requirements, while some were lighter ones that were less critical. The requirements were basically split up into three categories: performance, hardware, and software requirements. Success, again, was determined by the ability to meet these requirements and specifications, as well as the goal and objectives. A summary of requirements and specifications is given below:

Requirements & Specifications Summary:

- 10 to 15 minutes of flight time per charge
- Stable flight in winds of up to 5MPH
- Maximum altitude of about 150 feet from the ground
- Survive drops from 3-4 feet
- Integrate the ARM processor, sensors, and supporting hardware onto a single printed circuit board (PCB)
- Must use DTN software

3. Research & Preliminary Design

3.1. Similar Projects & Products

There were many different ways to implement a quadcopter design, and tough decisions on system design needed to be made. For the KIRC project, the team had to come up with a system and figure out how far down the design had to go. At the top level, the team could have bought a commercial quadcopter and called it their own, but that wouldn't satisfy the UCF engineering requirements. At the lowest level, the team could also have built most of the sensors and computer hardware themselves, but that would have been an unreasonable and unnecessary approach. So, in order to satisfy the requirements set forth by UCF and still remain a viable project, the team decided to define on what level was the design to be done and what type of system to implement. The project design level and system layout was determined by looking at what others have done and drawing ideas from their projects including successes as well as follies. To get a good feel for what other ideas people had implemented on their quadcopter designs, a formal market research needed to be done. In this section, similar products and projects were evaluated, and a formal project design level was established.

One approach to the project was to buy a commercial off-the-shelf (COTS) quadcopter from the market and adapt it to the requirements of the project. Indeed, there were a lot of quadcopters available on the market from various sources such as Parallax, DJI, and Parrot. Each of these companies produced quadcopters for commercial purchase, but buying quadcopters from them would not have worked well in this project. First of all, when buying a commercial quad-copter would make some parts of the project very difficult. Most commercial enterprises do not make their code or design documents freely available to the public which would've made customization very difficult to do. Another reason the team avoided going with a commercial quadcopter is that they tend to be quite expensive. Most quality quadcopters that were able to perform under the requirements of this project were at least \$500 or more. With the budget around \$1000 for everything, buying two commercial quadcopters would have decimated the budget and not left any room for the other parts for imaging, autonomous flight, and the printed circuit board (PCB). Unfortunately, these commercial quadcopters didn't really leave much in terms of technical ideas to draw from either. With much of the technical details of the quadcopters not available to the public, it made gathering information from these types of quadcopters impossible. The only thing that was listed that helped was the performance characteristics. Using this information, the team was able to make realistic design constraints on the performance of the quadcopter under certain conditions.

In the website for previous electrical and computer engineering senior design projects at the University of Central Florida, there were several teams of students who had made UAVs. One in particular had a single rotor and was encased in a wire cage which made for a very interesting design. Automatic Flying Security Drone, or AFSD for short, was intended to provide a UAV to security forces to 'patrol' an area autonomously. They had some novel designs that intrigued our senior design team. In order to maintain attitude control and steer the device, they used small rudders under the rotor to deflect the air and

create a small controllable force. With the cage design holding in all the electronics and keeping the motors out of reach, it made for a very safe design. The project also proved to be very innovative with its rudder design. While this project did have a few good points to draw from, unfortunately it wouldn't fare well in a higher altitude application. At higher altitudes, wind would become a large issue, and it could've created a potential problem where the cage design would have tumbled in the wind and the rudder for the motor wouldn't be able to compensate enough. This could've caused a potential crash of the copter system and may have resulted in the total loss of the system. Another drawback was the copter's limited cruising speed. The rudder design, while novel, didn't provide the force necessary to push the copter to higher speeds. For this project, the cruising speed needed to be enough such that large areas could be covered with a limited battery life. What could be drawn from this project was their use of control algorithms for flight navigation and specific parts used. Their use of state machines for their navigation control routine was a very viable option for the KIRC project. Using a GPS as a navigation feedback sensor, they were able to fly the copter between two points while using simple geometry to determine a trajectory. When taking off, flying, or landing there were states for each that were going to be integrated into the state machine. These ideas were able to be implemented in the final KIRC design.

Another design of interest also came from a UCF electrical and computer engineering senior design project. Autonomous Quadrotors Utilizing Attack Logic Under Navigational Guidance, or A.Q.U.A.L.U.N.G. for short, was another quadcopter project that utilized the use of autonomous flight. The original idea behind their project was to create a game whereby competitive laser tag players could play a laser tag game against multiple autonomous quadcopter opponents. There were many ideas from this project that were able to be implemented on the KIRC project. Their use of motors, batteries, frames, and computers were able to be potentially useful to draw ideas from. Their autonomous flight algorithms also made for a good place to start on KIRC's own autonomous flight algorithms. Their use of computer vision to provide obstacle avoidance, however, might prove too complex for this project though. A.Q.U.A.L.U.N.G. did provide a good resource of information on quadcopter design that the team kept in mind for design purposes.

While buying a quadcopter would've proved expensive and wouldn't have satisfied the requirements of the project, making everything from scratch would also be unnecessary. So, for the KIRC project, it was decided that the design would be done with the assumption that certain technologies were available: Microcontrollers, IMU sensors, Motors, Speed controllers, Frames, Propellers, Operating Systems, and all associated support hardware and software.

In summary, all the projects assessed in this section had portions that were implemented in KIRC. In the next sections, decisions and driving factors on KIRC parts and designs were given. These decisions were made based on the research given in this section and other research into cost and viability.

3.2. Relevant Technology & Strategic Components

In this section, the general technologies that were likely to be used in the KIRC project were given based on the research effort made by the team. Extensive research was done to find common parts, processes, and software used in other UAV projects, specifically ones that had to do with quadcopters. These technologies were integrated into the core of the KIRC design. The goal behind using common technologies in this project was to provide a system based on proven technologies that was known to work properly and be predictable.

One of the most important items in the KIRC project was the use of microcontrollers for the flight computers. The microcontrollers were critical to keeping the quadcopters stable in flight. In researching quadcopter designs, it was found that the majority of them used some form of microcontroller to control stability. Finding the right microcontroller was difficult, however, with so many to choose from. Usually, the microcontrollers most commonly used had the ability to do floating point arithmetic, support for interrupt driven software, and lots of peripherals such as UART, I2C, SPI, PWM, and A/D conversion. In order to satisfy UCF requirements, the team had to create a PCB which integrated a microcontroller. This put another requirement on the microcontroller, meaning that it needed to be available on a Launchpad for prototyping as well as a standalone chip for the final PCB prototype. This also means that the microcontroller needed to have support for JTAG in circuit debugging.

Sometimes, but not always, projects would implement a real time operating system (RTOS) as part of the software on the microcontroller. The advantage of this approach was that it prevented scheduling conflicts with the software, and also ensured that the processing of the data occurred in real-time. Unfortunately, RTOS software tended to be very large and required enough memory, and processor speed in order to run. The advantage, however, came in handy when implementing a digital control system.

Many of the quadcopter systems researched used a digital control system implemented on a microcontroller for stabilizing the quadcopter. Quadcopters were naturally unstable in flight and need an active attitude control system to maintain stability at all times. In researching flight control algorithms, many control topologies came up. Many people tended to use Proportional Integral Derivative (PID) loops to dynamically minimize the error from the roll pitch and yaw of their quadcopters. By stabilizing the roll, pitch, and yaw of the quadcopter, it became easier to control the quadcopter in flight and also made autonomous navigation possible. Even though there were many ways to implement a digital control system, one thing remained constant for all of them. Digital control systems only work under the assumption that the feedback and controller output occur at discrete time intervals. For this reason, the RTOS came in handy because it created a predictable time interval where control processing could occur. Using these windows of processing, all of the discrete math associated with the digital control system could be accurately done in the proper time.

Feedback in digital control of UAVs was usually done using inertial measurement units (IMU). This sensor group contained an accelerometer, rate gyroscope, magnetometer,

and altimeter, and used I2C as a serial protocol over a common line. The accelerometer and gyroscope were used to find an attitude state vector, while the magnetometer was used to find absolute heading and the altimeter found the altitude. Most IMU sensors tended to be noisy and biased when reading them directly, so frequently people used digital filters to ‘filter’ out the noise from the sensors. The filters most commonly used to get rid of the noise from these sources were adaptive filters. Adaptive filters actively changed based on optimization algorithms to most optimally get rid of noise. There were two common filters of this type most commonly used in practice: The Kalman Filter and the Least Squares Filter. Each of these filters had advantages and disadvantages that came into consideration when implementing them. First, the Kalman Filter, was a very accurate filter that provided excellent noise reduction for any application. The downside of this algorithm is that it is very computation intensive and could bog down a computer system rather quickly when implementing it in an embedded application. The other, the Least Squares Filter, used a computationally efficient algorithm that does a moderate job of getting rid of noise. While this filter is not as accurate as the Kalman filter, it was more ideal for computationally limited applications.

The input of the digital control system needed to come from somewhere. Usually while holding still in flight, the digital controller needed to just keep the error minimized while the input was zero. In reality, in order to move the quadcopter, there needed to be an input from a controller. For phase 1 of the project, the controller was planned to be a user with a RC controller device. The most commonly used type of RC controller for UAV applications was a hand held, joystick operated, controller device. These devices usually had settings for trim, offset, and mode. When looking for a RC controller, it was important to note the number of channels available to the user because this affected how many different signals the controller could send to the quadcopter at any one time. Each of the channels represented an input to the quadcopter control system that could change roll, pitch, yaw, throttle, etc. Another thing to keep in mind when buying the RC controller was the price. Unfortunately, RC controllers could tend to be quite pricey. The plan was to only buy one RC controller, as it was not planned to fly both quadcopters manually at the same time.

Pulse Width Modulation (PWM) was common method for motor control in UAV projects. PWM signals could be easily generated using a microcontroller, and many microcontrollers included built-in hardware for it. This control signal was ideal for motor control because it provided a modulated signal that contained load voltage percentage information. This meant that the PWM signal could change the output of a motor based on the duty cycle of its square wave. Electric motors, however, don’t usually take PWM directly as an input, so there had to be an intermediate circuit that could translate the PWM signal into a DC voltage for the motors. Electronic Speed Controllers, or ESCs, were circuits that could take a PWM input and send proper voltage and current to the motors to provide predictable motor output. ESCs needed to be chosen based on their current rating for motor support. This meant that if a motor is drawing 20 Amps, the ESC needed to be able to source 20 Amps without overheating, distortion, or causing an electrical failure.

Finding the right electric motors for the project was also an important task. The most commonly used type of electric motor in quadcopter projects was the brushless motor. In the process of looking up brushless motors, there were several specifications that came up. The Kv rating, the current rating, the power rating, the thrust, the weight, and the size were all things that were frequently listed among tech specs. For this project, motors had to be found that fit the frame of the quadcopter and had the capability to lift the whole system while not drawing too much current. The motors were definitely the most responsible for the current draw of the system, which directly affected the battery life and thus the flight time. Because the motors had such a direct effect on the flight time of the quadcopters, care needed to be taken in the decision of the motors for the project.

The closest future development for the KIRC project was the ability for the quadcopter to be autonomous. For that section, the team drew ideas from both the AQUALUNG and the AFSD project. The flight path and control loops would be calculated in real-time, while obstacle avoidance would not be taken into account. NASA desired the team to use a Raspberry Pi™ Model B, an embedded Linux platform, to test the DTN2 application as well as handle the image retrieval and telemetry relay. The Raspberry Pi could also be used to set flight paths and communicate (via UART) with the microcontroller that was being used for stability. Though the Raspberry Pi™ had more processing power than necessary for the project, it was necessary for the testing that NASA had in mind for the DTN2 protocol. The plan was to utilize the Raspberry Pi's™ extensive processing power to handle the option of future autonomous capabilities of the project.

In order to find the absolute position of the quadcopters, the IMU was not enough. The absolute position of the quadcopter was necessary for autonomous navigation. To find absolute position was relatively easy through the use of a global positioning system (GPS). There were inexpensive light weight GPS units available on the market that could interface to a microcontroller using a UART serial port. These units had a standard for communication called NMEA, which was named after the National Marine Electronics Association. This standard was almost always used in commercial GPS units, which made the code for parsing NMEA strings quite common. The software created for this project would have to include some NMEA parsing code in order to extract important information from the GPS.

Power management in quadcopter systems was critical to keeping flight time adequate enough. Many quadcopters, and indeed the ones evaluated in the previous section, used lithium polymer (Li-Po) batteries for their primary power source. The batteries for the KIRC project would have to be as lightweight as possible, and still able to hold enough charge, and provide enough energy to power all the devices and motors. This meant that the batteries would need to have a high energy density and low weight overhead. Lithium polymer batteries provided this performance in a small size and weight package. Compared to traditional alkaline batteries, Li-Po batteries had a much longer battery life and larger power capability while not being too much larger or heavier. Li-Po batteries usually came in configurations where multiple Li-Po cells had been linked in series to provide more voltage. Each cell for a Li-Po provided about 3.7V nominally when charged, and quadcopters usually required 12V sources to power the motors. So, for this

project, a 3 cell Li-Po battery was be ideal. Another thing to keep in mind when purchasing the battery was the battery charger that came with it. When the energy stored in the battery became depleted, the battery needed to be charged before using it again. So, after the right batteries were found for the project, a battery charger to go with them needed also be found.

In summary, all these technologies were ones that were likely going to be used in the KIRC project. The goal was to use common technologies in the project so that everything was guaranteed to work properly and predictably. In the next section, specific parts were given based on the general technologies given in this section.

3.3. Specific Components and Decision Factors

In this section, the specific components and software used in this project were given. The driving decision factors for each product is also given along with comparison of selected other products of interest.

The first component chosen for this project was the microcontroller due to the fact that it had the most impact on the software portion of the project. Based on the relevant technologies shown in the previous section, some requirements were made on what peripherals the microcontroller needed to have and performance criteria it needed to meet. A summary of the requirements is given below in a bulleted list:

- Needed to have a Floating Point Unit (FPU) to support control algorithm calculations
- Needed to have multiple UART ports for serial communication with the GPS and the Raspberry Pi
- Needed to have I2C ports for reading IMU
- Needed to have multiple PWM channels for motor control output
- Needed to have A/D converter
- Needed to have Real Time Operating System (RTOS) support
- 32 Bit Processor with fast enough clock rate to perform all control functions reasonably
- Needed to have a Launchpad/PCB available for prototyping
- Needed to have individual microcontroller available for own PCB development

With these factors in mind, research was made into different microcontrollers from different vendors. There were four options that stood out from a controller perspective that closely fit these requirements. In table 1 on the next page, the four microcontrollers were compared for the decision factors for this project.

Name	Vendor	Processor	RTOS Support	Availability	Price
UNO32	Digilent	PIC32MX320F128	No	Launchpad, Standalone	\$26.95
Piccolo Launchpad	TI	F28027F	Yes	Launchpad, Standalone	\$17.00
Stellaris Launchpad	TI	EK-LMF120XL	Yes	Launchpad only	\$13.49
Tiva C Launchpad	TI	TM4C123GH6PMI	Yes	Launchpad, Standalone	\$12.99

Table 1: Comparison of Select Microcontrollers

The final decision was to go with the Tiva C TM Launchpad from Texas Instruments for the prototyping part of the project and then obtain the chip and embed it in its own PCB with all the peripherals already integrated. While all the microcontrollers would have worked for the project, the Tiva TM C was decided based on the price and performance. The Tiva TM C is a very capable microcontroller with a 32 bit, 80MHz, ARM Cortex M4 Processor. It had plenty of flash ROM to hold the RTOS, with 256KB. It also had enough RAM to support any amount of data or variable necessary for the control algorithms, with 32KB available. The microcontroller also had support for multiple UART, I2C, SPI, PWM, A/D, and GPIO ports in hardware. One very attractive feature of this microcontroller was its available RTOS from Texas Instruments. The TI RTOS could be specifically optimized for this microcontroller. The microcontroller was also low power, drawing only 45mA in nominal mode according to the datasheet.

After a microcontroller was chosen, the next step was to finalize on the IMU. The IMU was critical to the flight stability control, and without it, the quadcopter would be flying blind. The reason for obtaining this part early was to start reading from it, developing filtering and control algorithms, and getting an early start on the software development. Choosing an IMU, however, could be very difficult. They could be very expensive, with prices being up to the thousands of dollars. On the other hand, going with an inexpensive unit could result in the compromise of quality, accuracy, and precision. With all these compromises evident, it was useful to provide a list of requirements specific to this project. Below is the list of requirement that the IMU needed to meet in order to satisfy the requirements of the project:

- Needed to be less than \$100 (preferably less than \$50)
- Needed to be I2C compatible
- Needed to have the following sensors: Accelerometer, Gyroscope, and Magnetometer
- Needed to work on 3.3V power and low current

- Needed to fit on through-hole mounting shield of size less than the microcontroller
- All on board sensors needed to be available individually from at least one vendor so that they could be incorporated into the PCB design

Using these requirements, the team searched for candidate IMUs for use in the project. There were several available from Ebay or even Amazon for a good price, but unfortunately, getting approval from NASA for these items proved difficult. There was also one from Sparkfun™ that fit perfectly, but had an expensive price of \$99.95 each. While this price did reach the limit of the budget for the part, it was an acceptable source to buy parts from. So, the 9 Degrees of Freedom – Sensor Stick from Sparkfun™ was the part of choice for the team.

The 9DoF sensor stick from Sparkfun™ had three sensors on board: the ADXL345 accelerometer from Analog Devices, the HMC5883L magnetometer from Honeywell, and the ITC-3200 gyroscope from InvenSense. These sensors were all available for individual purchase from their respective companies. All the sensors had support for I2C slave communication, and had a breakout header available for prototyping. The stick ran on a 3.3V source, which the microcontroller could source. The stick itself was also very small, taking up only 1.37” by 0.42” of space. Sparkfun provided a fair amount of documentation of this part for interfacing and troubleshooting.

The GPS was the most important part of the autonomous system. With this part, the quadcopters would know absolute position and be able to correct navigation. This part, just like the IMU, could tend to be very expensive. The GPS could come in varying sizes as well. For this project, a smaller GPS needed to be used that still had the signal strength to receive signal from even in the midst of the EMI caused by the motors. Below is a list of requirements for the GPS unit:

- Needed to have signal strength enough to overcome motor EMI
- Needed to be small enough to fit
- Needed to meet sensitivity requirements
- Needed to acquire fixed location within a short time
- Needed to be low cost

A big requirement was both the speed to connect with usable satellites and the number of satellites desired to use. Previously the standard was 16-channel, but for this project a 50-channel was better for precision. 50-channel was more precise due to the fact that the amount of channels correlates to the amount of satellites that could be used at one time. Though there would not be 50 satellites designated to global positioning during the design of this project, it would be able to use as many as were available. The connection to a larger amount of satellites correlated to a faster speed at measuring changes in location. When choosing a Global Positioning System, especially with a small target like a quad-copter and a need for precision, sensitivity was crucial to the outcome of the system. In order to maintain route and avoid missing images in the mapping process the sensitivity needed to be less than -160dBm during tracking and navigation.

Table 2 shown below contains a list of the GPS units considered. One of the concerns for the GPS was the startup time. This was known as the TTFF or time to first fix. The drain on the power source being the limiting factor on the flight time of this project brought the requirement that the TTFF be less than 30 seconds. With a startup procedure that could enable a low power part with long start up time as this one before powering more high power components would assist in the longevity of flight. This part would also need to have low powered, but with these components' power is usually not a real issue.

When it came to the budget of this project, being cost efficient was important. The budget set before this project was one of the major challenges due to the desire to replicate. For this reason all of the components used needed to be efficiently chosen to be cheap and up to our specification needs.

Name	Vendor	Power	Number channels	TTFF (seconds)	Sensitivity (dBm)	Price (\$)
GS407	S.P.K. Electronics Co.	3.3V@75mA	50	29	-160	\$89.95
GP635T	ADH Technology Co.	5V@56mA	50	27	-161	\$39.95
D2523T	ADH Technology Co.	3.3V@74mA	50	29	-160	\$104.00

Table 2: Comparison of Select GPS Units

The GP-635T produced by ADH Technology Co. Ltd. was not only slim and easily implementable, it was also cost effective at \$39.99 each, half the price of other competitors. The GPS receiver would work in coordination with the IMU in the process of flight, while the IMU would be completely responsible for flight equilibrium. This GPS receiver was accurate to -161dBm with a UART default baud rate of 38400 bps, and the implementation of digital I/O. Environmentally, this unit could maintain operational status between -40 and +85 degrees Celsius and less than 500 Hz of vibration, less than the amount expected under the worst of conditions. The accuracy and speed of communication with this receiver was crucial due to the nature of this project. Each image and, by extension, movement needed to be precise in order to get a good mapping of the area and accurate autonomous navigation in the future routine. The power needs were relatively small, but we sacrificed a small bit of battery life for the precision of such a component with 4.75 to 5.25 power supply needs with max of 56 mA current, which in comparison to the rest of the system would be close to nothing. As packaging went this component only sat at 8x35x6.55 mm, this assisted in keeping the payload of the quad-copter compact. At the price and precision of this component and its ease of implementation made it a clear choice for KIRC.

Electronically, the frame was of little importance, but it was a crucial piece part in the quad-copter when it came to packaging and flight abilities. When in actual flight the frame needed to withstand the environment, and affects speed, control, and maneuverability based on rotor separation. A frame that could be bought premade could come in a variety of dimensions, materials, and styles. Below is a list of requirements for the frame:

- Needed to be lightweight
- Needed to be agile in motion
- Needed to be rugged enough for most environments
- Needed to not hinder system capabilities

For this project it was between three different types of frames. Due to the fact that finding a frame made of a lightweight material is not an issue, part decision came down to agility, precision and system integration.

The first option that was considered was a frame done by a previous UCF senior design group known as the “Automatic Flying Security Drone”. In this project a cage in the shape of an orb was implemented. Though this design was creative, aesthetically pleasing, and well designed for the previous group’s requirements, it didn’t possess the agility and mobility that was expected for the missions KIRC will be involved in. In addition, this design would have issues in dealing with the environment at high altitudes due to the fact the aircraft would be blown off course in certain conditions.

The second option for a frame was a hex-copter frame. With this design we would have all of the agility, speed and control needed for completion of any mission set before the aircraft. This frame design was tossed, however, when system integration was considered. Using a frame that would require not only four motors, which were the main pull on the power source to begin with, but six motors made this frame unrealistic with requirements set forth for flight time.

After these considerations it was decided that a lightweight quad-copter frame would be the best for system integration into the KIRC project. This frame gave agility, and mobility enough to counteract any environmental issues seen, and had as little pull on the battery as necessary for mission purposes. The frame had reasonable size dimensions (450mm x 450mm x 55 mm) with motor mount bolt holes preset for ease of implementation, and lightweight (270g., 1.04 lbs.).

Motors carried a lot of weight when deciding the aeronautical capabilities of this aircraft. An analysis of flight and flight time capabilities with each of the motors available was the best way to narrow down the options. After the motor was chosen, then the ESC was decided by implementation abilities with selected motor.

- Needed to have thrust capabilities to hover payload at less than 50% thrust capacity
- Needed to be powered by 15 V or less
- Needed to be low priced, less than \$20
- Needed to adhere to the above requirements and maintain a flight time greater than 12 minutes with a 5 Amp/hour battery

After narrowing the search based on the price ceiling there were still many options readily available. The implementation of flight time calculator was implemented based on weight and thrust capacity of the motors. In order to do this, add the mass of each piece part including a tolerance for lightweight parts to be implemented on the PCB later. Then take the total thrust of the four motors (assuming they were all four the same type

of motor) and divide the total mass by this maximum thrust capability. Take this quotient is multiplied by the maximum total current drawn by the entire system. Finally, divide the life of the battery by this product, giving the total static flight time aiming for a minimum of 12 minutes. This analysis may be seen in the two equations and table 3 shown on the below for the chosen motor.

$$\%Thrust = \frac{m}{4 * Max Thrust}$$

$$FlightTime = \frac{Q_{bat}}{\%Thrust * I_{sys}}$$

m = mass of the entire system, in grams

$Max Thrust$ = max thrust for each motor, in grams

Q_{bat} = lifespan of the battery in Ampere Hours

I_{sys} = current draw of motors and electrical circuits

Part	Description	Part #	Mass (g)	Qty	Current (A)	Thrust (g)	Capacity (Ah)
Frame	Fiber Glass Quadcopter	Q450	280	1	0	0	0
Propeller Motors	Prop Drive Motor	NTM2830S-900	66	4	10.6	750	0
ESC	Afro 20Amp ESC	919200131-0	23	4	0	0	0
Battery	Turnigy 4s Lipo Battery	N5000.4S.45	552	1	0	0	5
Propeller	8x4.5 Propellers	17000055	15	4	0	0	0
Avionics	Controls, Processors, Sensors	Tiva/Raspi	200	1	1	0	0
	TOTALS	N/A	1448		43.4	3000	5

%Throttle to keep level	0.482666667
Current Drain (A)	21.46506667
Flight Time (Minutes)	13.97619698

Table 3: Flight Time Calculation Table

This calculation was done for all motors considered. Three NTM Prop Drive motors were considered above all others, an 800kV/300W, a 900kV/270W, and a 1000kV/235W. The analysis outcomes are shown in table 4.

	800KV/300W Motor	900KV/270W Motor	1000KV/235W Motor
%Throttle to Keep Static in Air	0.343809524	0.482666667	0.373532551
Current Drain (A)	25.4792381	21.46506667	23.26254002
Flight Time (Minutes)	11.77429242	13.97619698	12.89627013
Price	\$15.29	\$14.99	\$15.99

Table 4: Comparison of Select Motors

The 900kV/200W motor was the most cost efficient and allowed for the longest flight time. For this project, with the equipment so chosen, the NTM Prop Drive Series 28-30S 900kv drew enough current and supplied enough thrust to give the quad-copter the agility and mobility that was desired, as well as optimized the flight time, and cost.

Electronic speed controllers, ESCs, do the job of taking I2C PWM signal from the microcontroller and filtering the current to the motors in order to scale the thrust output of each motor.

- Needed to be easily integrated with the chosen motors
- Needed to be capable of drawing at least 15 amps
- Needed to implement I2C standard for communication

For this component only two Afro ESCs were considered, due to the fact that they were low cost and I2C standard. These were both 20 amp ESCs one being slim and the other a box package shape. Upon further search into the slim ESC it was found that the product was actually 30 amp capable, only being toned down by the 20 gauge wire being used. Do to this fact, along with the higher price and more difficult package to implement, the Afro ESC 20Amp Multi-rotor Motor Speed Controller was chosen as the project moves forward.

When it came to a flight system, it was only usable as long as the power source lasted. For our project being powered by a battery, the lifespan of this battery was the limiting factor on flight time. The best batteries as far as lifespan that were within the budget of this project were lithium polymer, LiPo for short. Requirements for the battery were as follows:

- Needed to be rechargeable
- Needed to have a rating of at least 4500 mAh
- Needed to be rated for at least 200 W
- Needed to be a 4 cell

In the decision of battery, the power vs. flight time table and calculations were implemented as when choosing the motor. When deciding whether to have a stronger battery or multiple batteries, the decision came down to price vs. addition to flight time. In addition this battery needed to be rechargeable, so that it could be put in the package and not have the necessity of being removed and replaced with each use. If a disposable battery was chosen, the budget of the project became time dependent, as the aircraft was used repetitively the cost of batteries added up quickly.

The calculation table used in the analysis for the motor selection was also implemented in this situation. The power source was the last thing chosen due to the fact that its requirements were dependent on the needs of the system. In this situation there were a couple of options when it comes to powering the system as a whole. The implementation of two batteries could be used or having a single battery with higher output.

The initial thought was to keep the budget as low as possible without diminishing the ability of the quad-copter's ability to complete its missions. In order to accomplish this a range of rechargeable 3 cell LiPo batteries were considered, centering on the range of 4500 mAh to 6000 mAh. The analysis of operation ability can be seen in Table 5 below.

	4500mAh	5000mAh	6000mAh
Flight Time (Minutes)	12.57857729	13.97619698	16.77143638
Weight (g)	513	552	623
Cost	\$57.15	\$65.03	\$64.75
Operating Temperatures (C)	45-90	45-90	25-50

Table 5: Comparison of Battery Size vs. Energy Capacity

Table 6 on the next page could be used in combination with table 5 above to consider additional aspects in design. The 6000 mAh batter lacked the range of operating temperatures that KIRC needed to be capable of performing in. Additionally, the weight of the battery didn't provide a large enough increase in flight time to encourage the loss in operating environments. The 4500 mAh battery was in the right range of temperature operation, but in terms of getting an additional minute and a half of flight time for the cost of \$7.88 only during replacement of the battery made the 5000 mAh battery the best power source. This extra time was useful during the longevity of some parts when cold starting.

When looking at the description in comparison with the rest of the system parts, it can be seen that the weight of the battery was much more than that of the other parts. For this reason, when two batteries were implemented an increase in flight time could be seen, but it wasn't a large enough increase in order to be worth the increase to 66% of the motor capabilities just to maintain static flight. A six minute increase in flight time could be seen, however this increase in weight made the drive on the motors too large, thereby taking away from its mobility and aerial agility. In addition the battery was one of the

most costly components in the budget. For all of these reasons it was decided that a single 3 cell, 5000 mAh, rechargeable battery was best suited for this project.

Part	Description	Part #	Mass (g)	Qty	Current (A)	Thrust (g)	Capacity (Ah)
Frame	Fiber Glass Quadcopter Frame	Q450	280	1	0	0	0
Propeller Motors	900KV/270W Motor	NTM2830S-900	66	4	10.6	750	0
ESC	Afro 20Amp ESC	919200131-0	23	4	0	0	0
Battery	Turnigy 4s Lipo Battery	N5000.4S.45	552	2	0	0	10
Propeller	8x4.5 Propellers	17000055	15	4	0	0	0
Avionics	Controls, Processors, Sensors	Tiva/Raspi	200	1	1	0	0
TOTALS		N/A	2000		43.4	3000	10

% Throttle to Keep Level	0.666666667
Current Drain (A)	29.26666667
Flight Time (Minutes)	20.50113895

Table 6: Flight Time Calculation Spreadsheet Example

4. Design Summary

4.1. Hardware Operation

Through the research and design phase of the project, the team had decided to divide the project into three phases with distinct hardware structures, as mentioned before. As previously mentioned, the three phases were as follows. All three phases of the team's quadcopter obviously required the essential mechanical components for flight such as a frame, four propellers, four motor drivers called electronic speed controllers, a battery, the microcontroller, and inertial measurement unit sensor components including an accelerometer, gyroscope, magnetometer, and barometric pressure and temperature sensor used to determine altitude. The first prototype phase consisted of the only the necessary components for stable, manually controlled flight and used a manufactured over the counter Launchpad microcontroller and an external sensor stick. The team used the first prototype phase to isolate any potential issues dealing with the stability algorithm and hardware issues related to flight. After the team was confident that the quadcopter achieved manual controlled flight capabilities, the team moved on to incorporating the PCB into phase two. The third phase was obviously much more complicated compared to phases two and three and had additional features such as a separate navigational computer, a camera to take pictures and fully autonomous flight capabilities. The navigational computer was implemented using a Raspberry Pi TM and uses information from a GPS sensor to determine its location at any given moment in time. Although the hardware of phases one, two, and three were somewhat different, the fortunate fact of the matter remained that each phase had been built by adding features the design of the previous phase. Although the components implemented in phases one and two were nearly identical, the team's integration of the microcontroller and the inertial measurement unit on the printed circuit board in phase two actually required less wire connectors, but introduced some interesting hardware challenges which had to be resolved.

The hardware operation essentially consists of a description of the specific hardware used, the connections between functioning hardware, and a general explanation of how each component communicates with one another specific to each phase implemented. As the design of the quadcopter project evolved, the team had gone through a number of different configurations, comparing and contrasting different ways to accomplish the same objectives. One large obstacle to overcome had been deciding exactly what tasks the Raspberry Pi navigational computer was in charge of and what tasks the microcontroller avionics computer had been designed to handle. This was important in the hardware design because the specific roles of both computer systems designated which components were connected to these computers. Significant time had been discussed concerning how the GPS is to communicate with the ground station laptop and the Raspberry Pi. There was some debate as to where and how the inertial measurement unit sensors were connected, and also some discussion regarding how the system was to read the battery voltage. The power distribution system of the hardware design had been pretty consistent throughout all of the three phases, however, when additional components were added such as the GPS sensor and the Raspberry Pi in phase three,

these electrical components required a ground and appropriate power line. The hardware operation concerning the power distribution is the first topic discussed.

A physical description of each component is provided to aid in visualizing the system. The battery chosen had 12.6 volts fully charged and 11.1 volts fully discharge, and is a three-cell lithium polymer nano-tech in the shape of a rectangular prism with three cables coming out of the top edge. One of the three protruding cables is a JST-XH connector which is used to recharge the battery after depletion. This recharge connector goes through an intermediate power adapter which is then plugged into a standard 110 to 120 volt wall outlet during charging events. The team decided to purchase extra batteries in order to have spare batteries charged at any given time. This strategy prevented the team from having to wait on charging batteries thus increasing the efficiency of the testing phase. Additionally, it was decided that having spare batteries during missions increases the potential area mapped out during the reconnaissance events. It should be mentioned that the team must land the quadcopters and manually switch out the batteries before resuming their paths. The other two protruding wires coming out of the battery are black and red in color and correspond to the positive and negative terminals of the battery. The team decided to use a standard convention and designated all ground wires to be black and all red wires to be red for all of the wires used connecting each of the components. This convention was consistent with the red and the black 4mm bullet power and ground cables coming from the battery. In phase one, two, and three the battery was connected directly to the electronic speed controllers; the electronic speed controllers have a built in 5 volt regulator circuit. In phase one, the team decided to power the microcontroller directly from the output of one of the ESC's 5 volt regulator. In phases 2 and 3, the team has integrated two separate voltage regulators, a 5 volt regulator and a 3.3 volt regulator. The team has decided to attach capacitors to the input and the output of the 5 volt and the 3.3 volt regulators in order to smooth out ripple and decrease noise. The team also decided to add diodes from the output to the input of each voltage regulator. The diodes protect each regulator in case the input power is cut off. In a case where the input power cuts off, the large capacitors at the output discharge their voltage, and the diodes make sure the voltage regulator is not destroyed if the voltage is significantly higher at the output node compared to the input node. This is accomplished because the diode provides a path flow from the output to the input, bypassing the regulator, for the current to flow. The team has decided to designate a 5 volt rail and a 3.3 volt rail from the output of each regulator in addition to the existing ground rail. With the PCB, the battery connects directly to the input of the 5 volt regulator and the 5 volt rails and 3.3 volt rails were used to power all of the on board electronics with the exception of the electronic speed controllers require the full voltage from the battery to power the motors.

In order to distribute power to each of the four motor controllers, the team purchased a splitter cable which splits the battery's single red wire into four red wires and the battery's single black wire into 4 black wires. Each electronic speed controller takes an input of one red wire and one black wire from splitter or breakout cable. The breakout cable's connectors are 4mm on the two wire side and 3.5 mm on the eight wire side. This proved to be an extremely convenient design due to the dimensions of the electronic speed controller's input cables. The breakout cable was a perfect liaison between the 4mm bullet connectors on the battery as well as the 3.5mm bullet connectors on the

electronic speed controllers. Each of the four electronic speed controllers have three input cables and three output cables. The manufacturer, considering the risks of reverse polarization, designed the two of the ESC's voltage input cables to be obviously red and black in color and correspond to positive and negative inputs from the battery. The single remaining input is a servo connector that would be attached to the microcontroller. The servo connector does not supply power to the electronic speed controller but instead provides a pulse width modulated signal. This pulse width modulation signal was designed to provide the electronic speed controller with a duty cycle which corresponds to the speed at which a motor should rotate. Three wires colored black, red, and yellow serve as the output of the electronic speed controller. One who does not have experience in robotics may be tempted to assume that the red and black wires connecting from the ESCs to the 3-phase brushless motors correspond to ground and high voltage lines, similar to that of the DC input coming from the battery to the ESCs. In actuality, the 3 phase brushless motors chosen take in an AC current input. This fact is important in understanding why the electronic speed controller is so vital in the team's quadcopter design. The electronic speed controller takes a DC current input and a PWM signal and creates an AC current output. Specifically, each of the three output wires from the ESC to each motor carry an alternating trapezoidal wave at different phases compared to one another at any given time. One may be inclined to believe the electronic speed controller controls the speed of the motor by varying voltage or current, but surprisingly it is controlled by the timing of these three phases with respect to one another. In contrast to the motors, the other electrical components on board are not powered by an AC signal from the ESCs but instead are powered by a DC voltage coming from the voltage regulator circuit. The first portion of the power system established concerns powering the four motors. Because the combination of the four total motors pull a relatively large amount of current compared to the other electronics on board, the motors are the main cause of battery depletion.

The next portion of the power system is to implement a voltage divider circuit. When designing the PCB, the team decided to include two different sources for the 5 volt power line. The first source comes from the ESC regulator as described above, and the second source comes from the output of the 5 volt regulator. Each of these two sources are mapped to two pins to the left and to the right of the main 5 volt pin. A jumper is used for convenient switching back and forth between these two sources. The logic behind having either source available is convenient for testing. For example, if the team wished to test the PCB, the team could simply use a power supply set at 5 volts and a low amperage as opposed to using the battery which may have varying voltage thus causing some uncertainty if problems were to arise. In fact, when the team first received the PCB, problems did arise; however, the team was able to pinpoint the issues and this feature helped expedite this troubleshooting process.

Initially in phase one's design, the microcontroller and the IMU sensor stick were connected using a perforated board. This perforated board was mounted to the bottom of the Tiva CTM launchpad and both were zip tied to the top plate of the quadcopter frame. This perforated-board design turned out to be very effective in keeping the IMU sensor stick level, necessary for initialization of control system algorithm. The control system was coded in a way that every time the system powers on, the IMU components read

their orientation at that moment and recognize this starting orientation the so called “level plane” to which the motors compensate to during flight. One disadvantage of this method was the team needed to make sure the quadcopter was level every single time the team powered the quadcopter on. The team also implemented a timer of about 2 seconds before any motors were enabled; this allowed the team member powering on the quadcopter to move out of the way in the off chance the quadcopter’s motors throttled on directly after powering the system thus minimizing injury to the team members. The perforated board was not necessary after the implementation of the PCB because the IMU sensor components are surface mounted directly onto the board.

The Tiva C Launchpad, the Raspberry Pi, and the GPS unit all take a supply voltage of 5 volts. The team decided to create a custom connector for the connection between the PCB and the Raspberry Pi. The custom connector was designed first in the schematic design. The team added through four holes in the layout of the PCB which have traces to the ground rail, the 5 volt power rail, and two through holes that connect from to the Tx and Rx UART pins on the microcontroller. These through holes were designed in such a fashion that the holes were equally spaced apart to fit a standard header pin format. After ordering the PCB, the header pins were soldered into the four through holes on the PCB. The team actually modified connectors taken from a rear projection TV (these connectors were initially used to connect boards together in the TV). The team used an X-ACTO blade to make a 12-pin connector into a 4-pin by cutting and off the unnecessary plastic to precisely fit the four pin header array. The team used a similar method to create custom connectors between the PCB and the ESCs and between the PCB and the RC controller receiver.

The IMU on the PCB obviously had traces to and from the 3.3 volt rail and the ground rail. In the phase one prototype, the IMU takes these two lines directly from pins on the Tiva C TM microcontroller launchpad. Had the team tried to directly connect the battery directly to these devices, they would be destroyed hence the need for a voltage regulator circuit. The other remaining components would be powered by either the microcontroller or the Raspberry Pi. There is in fact one more circuit that is included on the breadboard which is related to the power distribution system though not vital to the systems functionality.

The team decided to add a so called “battery life” feature to the quadcopter in phases two and three. In order to measure the amount of battery life in the power source, the team utilized one of the analog to digital converter pins on the microcontroller. The idea is as follows: the microcontroller periodically reads in a DC analog voltage from the battery. The microcontroller takes this analog input and converts it to a digital signal. The team sends this digital voltage from the microcontroller to the Raspberry Pi TM which then sends the battery voltage wirelessly to the ground station using the Wifi card. At the ground station laptop, the graphical user interface would display. The GUI updates in real time upon receiving the packets from navigational computer.

Because the analog to digital converter pin on the microcontroller cannot exceed a specified voltage, the team engineered a solution to step down the voltage from the battery to stay below the microcontroller’s voltage limitations. A voltage divider circuit

using two resistors and an operational amplifier was sufficient because the scaled down voltage is proportional to the total voltage on the battery and also below the voltage limitation on the A/D pin on the microcontroller. For this circuit, the input voltage, V_i , would be taken from the battery and the output voltage, V_o , would be wired to the A/D pin from the positive terminal of the unity gain buffer. The first resistor, R_1 , is between nodes V_i and V_o , and the second resistor, R_2 , is between node V_o and ground. The resistor values were chosen to be $100k\Omega$ resistor for R_1 and $400k\Omega$ resistor for R_2 . The equation below gives insight as to the rationale behind the resistor values chosen.

$$V_o = V_i \left(\frac{R_1}{R_1 + R_2} \right) = V_i \left(\frac{100k\Omega}{100k\Omega + 400k\Omega} \right) = V_i \left(\frac{1}{5} \right)$$

When the battery is at maximum power, the input voltage, V_i , is equal to 12.6 volts. The voltage divider circuit, with the given resistor values, forces the output voltage, V_o , to equal one fifth of 12.6 volts or 2.52 volts which is below the voltage threshold of the A/D converter pin on the microcontroller which is 3.3 volts. As the battery depletes, the output voltage, V_o , decreases proportionally with respect to the input voltage, V_i . When the microcontroller receives this stepped down voltage, V_o , there exists code which multiplies the value by 5 before sending the actual voltage to the ground station laptop. The team has gone through extensive analysis in testing the battery life. In this testing phase, the team recorded the voltage at the ground station laptop while exhausting the battery until depletion. From this data, the team generated countless graphs with the battery voltage on the dependent vertical axis and the amount of time elapsed on the independent horizontal axis. An accurate picture of how long the battery lasts is important during reconnaissance missions because since the quadcopter is manually controlled, the team must be prepared to land the quadcopter before the battery depletes. If the battery's voltage was to completely deplete during flight, the quadcopter's motors would not supply power to the propellers and the quadcopter would enter freefall and likely be destroyed upon crashing into the ground. Fortunately the team receives real time updates concerning the battery's voltage in the PCB integrated design, therefore giving the team an extra layer of information to avoid failure. In the future, if and when the team implements autonomous flight, the team would program the quadcopter to land automatically when the battery level gets too low. This feature was considered well in advance and was another reason this battery life circuit was implemented.

With a sound understanding of the power distribution system, it seems logical to move on to the next segment of the hardware design: the on board interconnections to and from each computer system. The hardware operation is heavily dependent on the communication method to and from each component. There are several different types of communication methods with specific connectors depending on the device of interest. The team implemented both wireless transmission from the RC controller to the receiver on board and wireless transmission from the Raspberry Pi to the laptop ground station. Although these two communication methods use the same radio frequency, 2.4 GHz, they use different channels in this frequency in order to prevent interference. In phase one, the RC controller receiver pins had female to female header pins to and from the microcontroller. In phases two and three, the connection between the RC controller receiver and the microcontroller was implemented with the custom connector. The

custom connector fit perfectly into the header pins of the PCB on one side and on the other side the wires were stripped, soldered, and electrical taped onto female header pins which connected to the appropriate pins on the RC controller receiver. Also on the PCB design, the IMU and the microcontroller were connected through traces embedded on the board. These copper traces on the board provide an ideal method of communication due to the minimal length of wire and low noise. For the team's quadcopter project specifically, there is a very high frequency of communication between the IMU and the microcontroller. The short copper traces provide the least amount of room for error and aid in stabilizing the quadcopter during flight. Having the hardware of the IMU and microcontroller on the same chip, mere millimeters apart, permits the fastest possible communication making the quadcopter more responsive to unforeseen disturbances.

The first on board communication relationship discussed may be the most important because the avionics computer's role was to control flight and stabilization of the quadcopter. It accomplished this task by taking in information from the IMU and sending signals to the input on the microcontroller. The microcontroller processes these signals to determine the orientation of the quadcopter. The IMU interacts with the microcontroller through a communication method called I2C. This method requires two lines between the microcontroller and the IMU. One of the two lines is referred to as the master slave line, and the other line is the clock. With information from the IMU, the microcontroller sends a pulse width modulated square wave signal to each of the 4 motor controllers. As mentioned above, each of the 4 motor controllers takes their PWM signal and converts it to an AC voltage which goes to each of the 4 motors on the quadcopter. Each of the 4 motor controllers have the capability of sending varying amounts of speed to manipulate each corresponding motor to first stabilize the quadcopter in flight then move the quadcopter in any direction in space.

One way to visualize this stabilization process is to picture one of the team's two quadcopters in midflight, hovering perfectly in one location in space. At this moment, the accelerometer in the IMU lets the avionics computer know that the quadcopter's velocity is zero and the gyroscope in the IMU lets the avionics computer know that the quadcopter is perfectly level. In this stable condition, the avionics computer is communicating to all 4 motor controllers to individually send approximately the same speed to each of the 4 motors to exactly counteract the downward force of gravity, thus keeping the quadcopter at a constant altitude. Keep in mind that the team's avionics computer is running through velocity and position calculations several times per second in order to correct any deviations to perfectly level as soon as possible. All of the sudden, a gust of wind blows the team's quadcopter out of its perfectly stable condition. For a split second after the gust occurs, two of the motors are slightly higher in altitude compared to the other two motors, and the team's quadcopter is now slightly tilted at an angle. The microcontroller, which is reading in information from the IMU several times per second, quickly realizes this slight tilt from the gyroscope's sensor as well as a slight change in velocity from the accelerometer's sensor. The avionics computer's microcontroller then quickly does calculations and sends appropriate amounts of speed to the 4 motor controllers to correct this tilt. In this specific example, the microcontroller could level out the quadcopter by providing slightly less speed to the two motors residing at a higher altitude, slightly more speed to the two motors residing at a lower altitude, or

a combination in order to make the gyroscope's sensor read perfectly level once again thus stabilizing the quadcopter to perfectly level.

Just as the avionics computer sends signals to each of the 4 motor controllers to stabilize the quadcopter in the air, the avionics computer can manipulate the signals to the motor controllers in order to move the quadcopter around in the air. The RC controller sends data to a receiver connected to the avionics computer. When a team member toggles a single joystick or both joysticks on the RC controller, the avionics computer takes this information from the receiver and sends signals to the motor controllers depending on how the joysticks are oriented. The RC controller has two joysticks. The team decided to use a common convention for commercial quadcopter joysticks. The left joystick on the controller controls thrust and rotation. Thrust is the forward to backward motion of the joystick, and rotation is the left to right motion of the joystick. The right joystick controls the pitch and the roll of the quadcopter. The pitch and the roll can be thought of as tilted deviations from a level plane. A useful analogy would be to think of a person standing upright. Hypothetically, a perfectly level wood board balanced on this individual's head makes up the so called level plane. The action of leaning forward, looking more toward the floor, or leaning backward, forcing the individual to look upward toward the sky, corresponds to a change in pitch given the level frame of reference. If this individual were to stand up straight again and either lean left or right, he would change the so called roll of the level plane. In this analogy, changing the so called yaw could be accomplished not by leaning forward, backward, right or left but instead by turning this person's head either left or right. This causes no tilt but changes the orientation of the plane to face a different direction. To sum up, the team's left joystick controls the thrust and the yaw, and the team's right joystick controls the pitch and the roll of the team's quadcopter.

The team's first goal was to have a fully manual RC controlled quadcopter functioning perfectly then the team would move on to program the autonomous flight mode. The RC controller would not be necessary after the team successfully programs both quadcopters to move autonomously, but the team planned to keep an RC controlled override feature just in case the team wishes to leave autonomous mode. This feature is accomplished with a switch on the RC controller to turn manual mode on or off. The team has decided to include the manual override feature to protect the quadcopter from flying away or crash landing during the testing phase. The team actually never accomplished this autonomous objective, but the hardware of the quadcopter allows this function possible in future embodiments. Now that a thorough understanding of the hardware of the avionics computer has been established, a discussion of the intricacies regarding the navigational computer's hardware will provide insight into how the two interact.

The Raspberry Pi TM serving as the navigational computer has several connections to and from its peripherals. The Raspberry Pi TM is connected to the avionics computer, to a camera module, to the wireless transmission adapter, and to an SD card. Although the quadcopter project does not utilize all of the external ports, the chosen Model B Raspberry Pi TM is a powerful machine with two USB ports, RCA video port, HDMI out, a 3.5mm audio jack, a 10/100 Ethernet port, 8 GPIO pins, a UART, i2c bus, and SPI bus with two chip selects. The SD card, the wireless adapter, and the camera module don't

require any hardware modification whatsoever; however, it should be mentioned that there was be a great amount of programming to incorporate all of these external devices. The SD card connects directly to the underside of the Raspberry Pi TM similar to plugging an SD card into a digital camera or any other electronic device. This SD card serves as extra data space and utilized when saving high definition images from the camera on board each quadcopter. Ideally, the Raspberry Pi would send the high definition pictures wirelessly to the ground station midflight. This feature would have been useful for the scenario in which, for whatever reason, the quadcopter gets lost during a reconnaissance mission and cannot return home, in which case the camera images taken during this flight would not be lost. Unfortunately, the team did not have be enough bandwidth to support this feature. Instead, the Raspberry Pi TM navigational computer receives pictures taken by the camera and stores them on the SD card attached directly to the SD port of the Raspberry Pi because the Raspberry Pi TM wireless communication the ground station did not have enough bandwidth to send these images. The camera chosen is conveniently made specifically for the Raspberry Pi. The camera module comes from the manufacturer with a 15-pin flexible ribbon cable connector attached to the board of the camera module. The flex connector attaches directly to the MIPI Camera Serial Interface 2 (CSI-2) interface connector without the need for purchasing any extra parts. The flexible connector simply connects to the Raspberry Pi board by pulling on two tabs on the top of the connector and snapping into place. Although the connections to the SD card and the camera module are relatively simple, the connection from the Raspberry Pi to the microcontroller proved to be more complex.

In the quadcopter design, the navigational computer on the Raspberry Pi TM uses a UART to communicate with the avionics computer on the Tiva C TM Launchpad. There are two lines for the UART communication which are known as a Clock line and a transmit/receive line. The team tied together input/output UART pins from the avionics and navigational computer to accomplish this task. Unlike standard USB cables, HDMI cables, or an Ethernet cables, the team must put effort when tapping into the GPIO pins of the raspberry pi and the Tiva C microcontroller. The team has decided to connect the UART pins of the microcontroller and the Raspberry Pi using single port female to female jumper wires. This method is ideal in that the team only used a small number of pins on either GPIO ports on both computers. The team decided against using a full ribbon cable for this very reason. The only remaining component left to discuss regarding the hardware operation is the wireless USB adapter connected to the Raspberry Pi.

The wireless USB adapter connects directly to one of the two USB ports on the Raspberry Pi TM. The ground station laptop has its own wireless adapter which communicates with the onboard Raspberry Pi TM on each quadcopter. The team planned to use the wireless transmission to the ground station as somewhat of a status indicator for both quadcopters simultaneously. The ground station is theoretically able to receive the following information from the both Raspberry Pis TM: GPS coordinates of both quadcopters' current location, both of their specific altitudes, and the battery voltage on both quadcopters. Although the avionics computer serves as the direct communicator to the global positioning sensor on board, the avionics computer sends the information from the GPS sensor to the Raspberry Pi which relays these coordinates to the ground station

via the wireless network. One may wonder why the GPS sensor is not directly connected to the Raspberry Pi. An explanation to be discussed in the next section 4.2., software operation, lies in the way the autonomous flight was planned to be implemented.

4.2. Software Operation

The software operation has gone through several changes as the design phase has progressed. As mentioned in the previous section 4.1., hardware operation, the team has gone through some debate as to exactly what tasks are to be handled by the Raspberry Pi navigational computer, the microcontroller avionics computer, and the ground station laptop. This section presents an overview of what the team has decided upon for the software to be programmed on each of these three computer systems.

To begin, the team first focused on the software to program the manual flight of the quadcopter. The team accomplished manual flight using an RC controller on land which sends signals to a receiver connected to the avionics flight computer. The team programed the flight computer to respond to movement of the two joysticks on the RC controller. In this context, it is to be understood the word respond entails sending signals to each of the 4 motors through the ESCs. Manipulating the left joystick upward successfully corresponds to an increase in throttle by increasing the speed of all four propellers. Manipulating the left joystick from left to right successfully correspond to a change in yaw. In order to understand how this was be accomplished, one must understand that two propellers spin clockwise and two propellers spin counterclockwise. Propellers spinning in the different directions are directly adjacent to one another and propellers spinning in the same direction are on the same axis on opposite ends of the quadcopter. A change in yaw was successfully accomplished by increasing or decreasing the speed of two propellers spinning in the same direction. The complexities of programming these functions become apparent as more details are exposed; however, these complexities seem trivial compared to the programming of the autonomous mode, a reach goal that was not accomplished throughout the project. Varying the right joystick up or down successfully changes the pitch of the quadcopter, tilting the “front” of the quadcopter downward or upward, respectively. The word front is placed in quotations because there was two ways of orienting the front of the quadcopter as shown in figure 2 below.

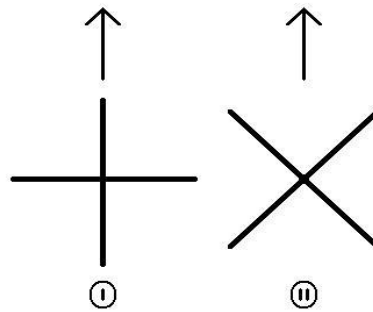


Figure 2: Quadcopter Orientation Diagram

In a typical helicopter or a plane, the front of the vehicle is obvious, but due to the symmetrical nature of quadcopters, the so called front of the vehicle is more ambiguous. The team considered each of the orientations and made the following conclusions. If the team designated the front to be on the axis of a single wing as indicated by I in figure 2, only two motors' speeds are manipulated in changing the pitch or the roll. If the team designated the front to be between two wings as indicated by II in figure 2, all four motors' speeds are manipulated when changing the pitch or the roll. An example is helpful in explaining how the pitch or roll is changed in the first orientation. Imagine the pilot presses the right joystick on the RC controller forward to tilt the nose of the quadcopter downward. In the first orientation, the two "side" motors perpendicular to the front axis are unchanged and the "front" motor's speed is decreased with respect to the "back" motor thus only the "front" and the "back" motors are used to change the pitch in this instance. Orientation II as labeled in figure 2, designates the front between two wings. If this were the orientation used in the example, the two front motors' speeds are changed with respect to the two back motors' speeds thus all four motors can be used to manipulate the pitch and the roll of the quadcopter in flight.

As one may imagine, pressing the right joystick on the RC controller from right to left also depends on how the front of the quadcopter is oriented. In orientation I, the two side motors' speeds are manipulated with to change the roll of the quadcopter. In orientation II, either the all four motors' speeds can be changed to roll left or right. The team has decided to choose orientation II because orientation II has the potential for greater torque during turns. This larger torque is possible because the team can program the quadcopter to utilize all four motors during turns in orientation II. In orientation I the maximum amount of motors utilized in turning is only two. These concepts were taken into consideration during the programming of the manual flight mode.

To program the autonomous flight mode, the team's avionics computer planned to use the Tiva™ C series ARM® Cortex™ microcontroller running an RTOS or a real time operating system. It should be mentioned that the team did indeed implement an RTOS for the manual flight for this reason. An RTOS is necessary for the information from the IMU to be transmitted rather quickly in order to quickly stabilize the quadcopter during flight. The avionics computer has a high frequency of checking altitude, orientation, and velocity to prevent the quadcopter from traveling off course or crashing. By contrast, the information coming from the GPS was not need to be checked as quickly because it's

coordinates in space are not nearly as important. The team programmed so called priority tasks on the RTOS or in other words, the RTOS ran the more important tasks in the avionics computer more frequently than the less important tasks. The navigational computer and the avionics computer of each quadcopter were programmed to provide the potential for autonomous flight. The avionics computer was inadvertently used to control the motors in flight, and the navigational computer was used to keep track of where the quadcopter is located at any given point in time. In the future the avionics computer was to use this location of the quadcopter along with the desired location to autonomously drive the motors to fly the quadcopter to a desired location. The avionics computer successfully took in data from the GPS sensor and sent this information directly to the navigational computer. When traveling, the team planned for the navigational computer to constantly compare the GPS coordinates of where it was currently located in space with the location it was to travel to. The navigational computer was to subtract the location of its current coordinate from the desired coordinate to generate a direction in which the quadcopter is to travel. The avionics computer was to receive this directional vector and compute some computations to extrapolate a specific thrust, yaw, pitch, and roll similar to an input command from the RC controller receiver. These instructions were to be used to send appropriate PWM signals to the electronic speed controllers just as the RC controller successfully does in order to move the quadcopter in space. The end result is a quadcopter reaching a desired destination for which the quadcopter is to hover and snap a photograph.

For the unfulfilled reach goal of autonomous flight, the navigational computer was to also be in charge of the paths of each quadcopter. The team would specify an altitude as well as latitude and longitude coordinate for 4 distinct points on the earth in the shape of a rectangle (or shape theoretically). The rectangle created by connecting these 4 points was to serve as the area in which the quadcopters would survey. The team would first perfect the reconnaissance with a simple rectangular geometry. Once the quadcopters were able to accurately image and create a map of this rectangular shape, the team would then move on to test other shapes. The team planned to program the navigational computer to take any 3 or more input coordinates and map out a map of any geometric shape. In the rectangular map, the paths of both quadcopters are somewhat simple, sweeping either from left to right or from top to bottom, side by side. The quadcopters would stop, hover, and take pictures during paths which are perpendicular and parallel to the outside lines of the rectangle. The paths and the locations at which the quadcopters would take pictures become more complex to program at more unfamiliar geometries. The navigational computer would determine the appropriate distance between photos taken based on the altitude specified. A higher altitude would permit more space between consecutive pictures due to a larger area covered per photograph.

The quadcopter would ideally maintain a specified altitude, and fly horizontally from one location to the next. It would be convenient if the quadcopters did not have to stop and hover to take pictures, but unfortunately the quadcopter cannot take pictures while traveling along its path without stopping since the camera is mounted directly on the bottom plane of the quadcopter. The team considered an alternate design in which the camera dangled directly below the quadcopter, but in this alternate design, the force of gravity on the camera would force the camera's vision to look directly downward at all

times. After much contemplation, the team decided against this design reasoning one major goal of the project aimed to have resilience toward disruption. A dangling camera would be subject of unsuspected outside forces such as wind or surprise collisions.

The last, and arguably most complicated, programming would be generating two quadcopters working in tandem. This task was impractical due to the implications of both quadcopters communicating with each other to avoid mid-space collisions, as well as coordinating the two copters' paths to optimize surveillance paths. In summary, the software algorithms throughout the duration of the project progressively got more and more complex providing an difficult challenge for the team to tackle, and all in all, the team was able to have one stable manually controlled quadcopter which successfully communicates with the ground station laptop.

5. Detailed System Design and Analysis

5.1. Detailed Block Diagrams

In designing the quadcopter project, the team decided to begin with general concepts and move into more detailed system design. In this section, a few block diagrams are discussed. Starting from basic block diagrams and moving on to more specific block diagrams paints a picture of exactly what subsystems were included in order to make the team's end goal of a quadcopter which images an area and sends real time updates to the ground station laptop a reality. The team created three basic block diagrams titled individual system block diagram, hardware block diagram, and software block diagram. After a thorough understanding of the basic hardware block diagrams, this section delves into two detailed system block diagrams called prototype block diagram and PCB block diagram. Lastly, this section discusses a software block diagram.

The first block diagram is shown below in figure 3. The first and the second block diagrams discussed are primarily used to obtain a general idea of the constituents implemented in the quadcopter project while the following two block diagrams are to expose the fine details and contain boxes for each electrical component and the connections between them included in the project. Specifically the prototype block diagram corresponds to the first phase in our quadcopter design, and the PCB block diagram corresponds to the second and third phases in our quadcopter design which have integrated the microcontroller and IMU sensor components onto a printed circuit board. The last block diagram, the software block diagram, provides insight into the programs which link the hardware components to each other.

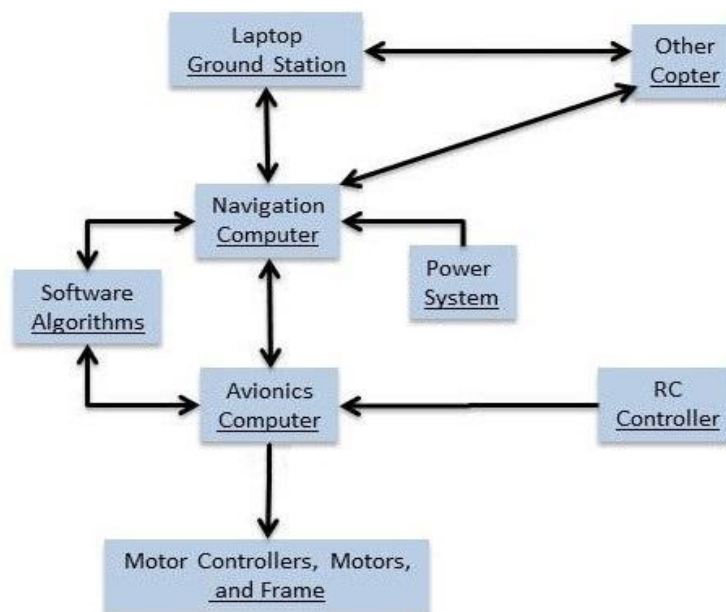


Figure 3: Individual System Block Diagram

The team's first block diagram, shown above in 3 titled "Individual System Block Diagram" shows how each of the individual systems interact. The individual systems included in this block diagram are as follows: a ground station controlled by a laptop, a navigation computer, software algorithms, a power distribution system, a computer controlling the avionics, a manual RC controller, motor controllers, motors, frame, and a separate quadcopter. The initial goal of the project was to create two autonomous quadcopters which work in tandem to image an area simultaneously. Unfortunately, the group had realized that this vision was quite ambitious, given the amount of time allotted to create the project. Because the team built the quadcopter from scratch including designing and coding the control system, the team did not have enough time to accomplish the autonomous flight objective. It should be mentioned that the team had this goal in mind throughout designing the quadcopter thus the quadcopter which the team created in phase three has all of the components required to achieve autonomous flight. Given a longer time frame, the team was confident that these two autonomous quadcopters, working together is feasible. In figure ???, the black lines with arrows on one or both sides, pointing from one box to another, give specific direction showing how each individual subsystem interacts with one another. For the black lines which have arrows pointing in both directions, to and from each two boxes, these two individual systems interact in a way that exchanges information. If serial communication is used for these two sided arrow interactions, the systems take turns sending data back and forth. The black arrow which points in only one direction, specifically between the avionics computer and the motor controllers, as one may guess the transfer of information is in one direction, and no information is sent back. In this case, the microcontroller sends a pulse width modulated signal to each of the four motor which corresponds to a percentage of full throttle for each of the four motors.

The diagram shows the other copter with an ability to communicate with the first quadcopter's navigational computer as well as the ground station laptop. By the end of the project, the team was able to use DTN protocol to communicate from one quadcopter to the ground station laptop, but the team was not able to implement the desired "three corner mesh network". It is worthwhile to mention that within the box labeled other copter also has its own avionics software, power system, RC controller, motor controllers, motors, and frame not shown on this block diagram. One can think of the individual system block diagram as the block diagram for each of the two quadcopters. A hypothetical block diagram for the other copter, if it had been created, would look identical to this individual system block diagram with one distinction being the text "other copter" would be changed to "initial copter". The rationale behind not including all of the components of the other copter in our block diagram of concern is to minimize complication. The block diagram's purpose is to give a simple representation that can be understood upon first glance. It is to be implied that the hardware and avionics software of the other copter is very similar to that of the first copter. Both of the quadcopters will send information to and from each other's navigational computer, and both of the quadcopters will send information to and from the ground station laptop. The individual system block diagram visually embodies the information presented in the previous section, section 4 design summary, where one can identify the connection of the avionics computer to the navigational computer directly through the hardware as well as through

the software algorithms. The individual system block diagram provides a solid foundation which can be expanded upon with the hardware block diagram.

The hardware block diagram shown below in figure 4 is similar to the individual system block diagram in that the team has chosen to use black arrows to show the relationship between elements in the diagram. One main distinction is that in the hardware block diagram, unlike the individual system block diagram, there is a specification of the communication method used to transport data from box to box. By looking at the diagram, one may notice the indications “USB”, “UART”, “PWM”, “I2C”, and “Wireless” above or to the right of each black arrow. All of these abbreviations represent different signal transmission protocols when communicating between devices. For the most part, each of the components considered in the research phase used a specific standard communication method, but the team did have some flexibility in choosing components which satisfied the desired communication protocol for reasons such as speed and compatibility. The IMU sends data from its sensors to the flight computer using I2C, and the GPS unit sends data from its sensors using UART. I2C is an abbreviation for inter integrated circuit and is a type of bus called a multi-master bus. UART is an abbreviation for Universal asynchronous receiver/transmitter and is used in this case for serial transmission or delivery. PWM is an abbreviation for pulse width modulation and will be the type of signal sent from the avionics computer to the motor controllers. Pulse width modulation turned out to be the ideal method of communication from the microcontroller to the ESCs because it is relatively easy to program a PWM signals to output pins of the microcontroller. The PWM signal from the microcontroller is a digital square wave with a varying duty cycle. The signal is programmed to be high and low for certain amount of times to each of the four motor controllers. The amount of time the signal is high divided by the amount of time for a single period is a fraction called the duty cycle. The duty cycle can vary from 0 percent to 100 percent and corresponds to the percentage of full speed each propeller spins.

The RC controller sends information wirelessly to an on board receiver. The RC controller receiver sends PWM signals to the microcontroller and is interpreted as the input to the control system code. A significant amount of software programming in the microcontroller was required to get the desired output at the motors from the input from the RC controller receiver.

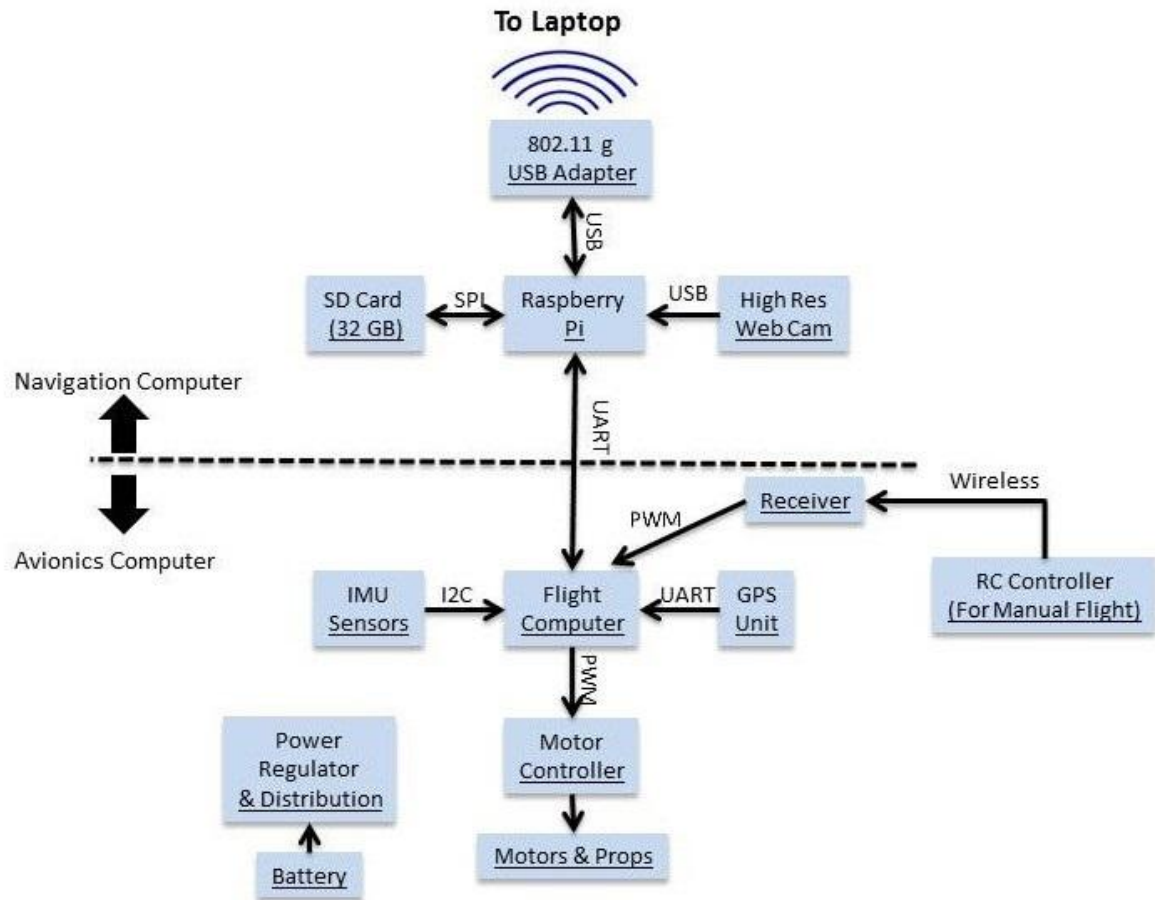


Figure 4: Hardware Block Diagram

The hardware block diagram shown in Figure 4 above is a visual representation of the information discussed in the previous section titled 4.1., hardware operation. According to the diagram, the navigational computer is connected to the flight computer, the camera module, the SD card, and the 802.11g wireless adapter. The navigational computer can be thought of as a separate mechanism, not necessarily needed for stability and manual flight. In practice this navigational computer was not implemented until phase three of the project. The separation between the avionics computer and the navigational computer is visually represented by a horizontal dotted line through the center of the hardware block diagram. In this diagram, it is implied that this power distribution system gives power to every electronic device on board in order to eliminate the need of drawing lines from the battery to the motor controllers and the voltage regulator circuit and from the voltage regulator circuit to the Raspberry Pi™, the flight computer, the IMU, and the GPS unit. The SD card, the wireless transmitter/receiver to the ground station laptop, and the Raspberry Pi™ camera module used all receive power directly from the Raspberry Pi™ thus these components do not require direct connections to the voltage regulator circuit. The concepts visually portrayed in the hardware block diagram are to be expanded upon in the following block diagram called prototype block diagram.

The prototype block diagram shown in figure 5 on the next page shows all of the connections to and from each of the electrical components in the prototype phase of the team's design. Because the subsequent block diagrams are extremely detailed, the team decided to use the individual system block diagram and the hardware block diagram as a prelude to the more meticulous block diagrams with arrows pointing each and every direction and even overlapping one another in some cases. The lines from the battery to the 4 ESCs and the voltage regulator circuit were implemented using two wires with a voltage difference equal to the battery voltage. The lines from each ESC to Motors one through four are composed on three wires due to the nature of the input power of each brushless outrunner motor. The lines from the Tiva C TM launchpad to each of the four ESCs are known as the radio connections and have three lines. The conventional nomenclature for these three lines are the positive, the negative, and the signal lines. In many radio controlled applications, the radio connections go directly into what is called a servo which communicates with an RC controller. The team's quadcopter project is significantly more sophisticated than an average hobbyist's design, and because the team has connected the radio connections directly to the microcontroller, these servos were not utilized. The design allowed for the quadcopter to have potential for autonomous flight, controlled by the Raspberry Pi TM. The arrow from the Tiva C TM launchpad to the sensor stick consisted of two lines required by the communication method called I2C. These two lines are the clock line and the master/slave line. The arrow between the Raspberry Pi TM and the Tiva C TM Launchpad as well as the arrow between the Tiva C TM and the GPS sensor also had two lines but will use UART as the communication method. The two lines utilized by UART are the clock and the transmission/receive line.

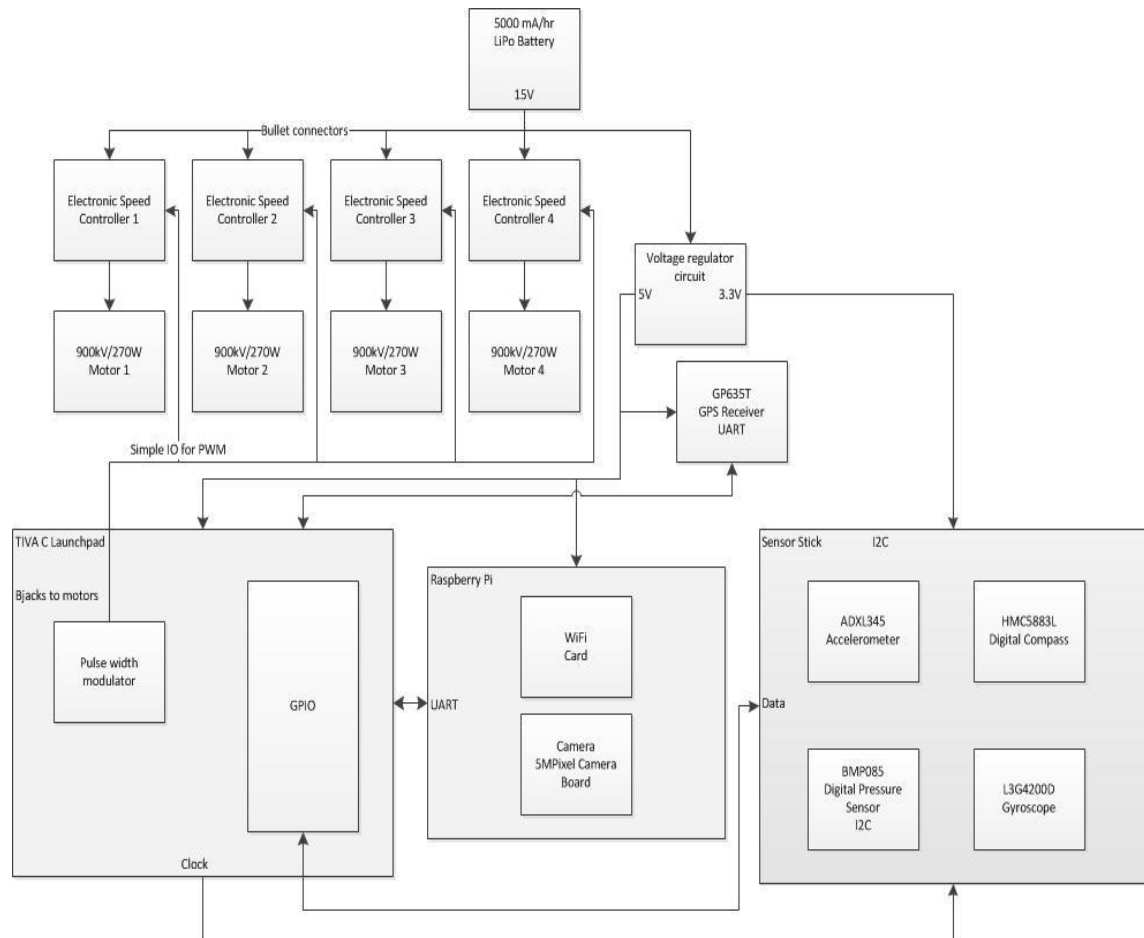


Figure 5: Prototype Block Diagram

The next block diagram shown in figure 6 on the next page titled PCB block diagram is very similar to the prototype block diagram with the exception of a few key distinctions. As one can see, the PCB box in the diagram contains all of the components from the sensor stick in the previous figure, figure 6, namely the accelerometer, the digital compass also known as a magnetometer, the gyroscope, and the digital pressure and temperature sensor which was used as an altimeter, and the voltage regulator circuit. The GPS sensor is external in all three phases. The use of a printed circuit board to embody all of the external components made the appearance of phase two and three's quadcopters more streamlined and significantly improved the performance of the system due to the proximity of the elements incorporated. Comparing the prototype block diagram and the PCB block diagram illustrates the advantages of the latter design. The remaining block diagram to be discussed regards the software implemented in the quadcopters.

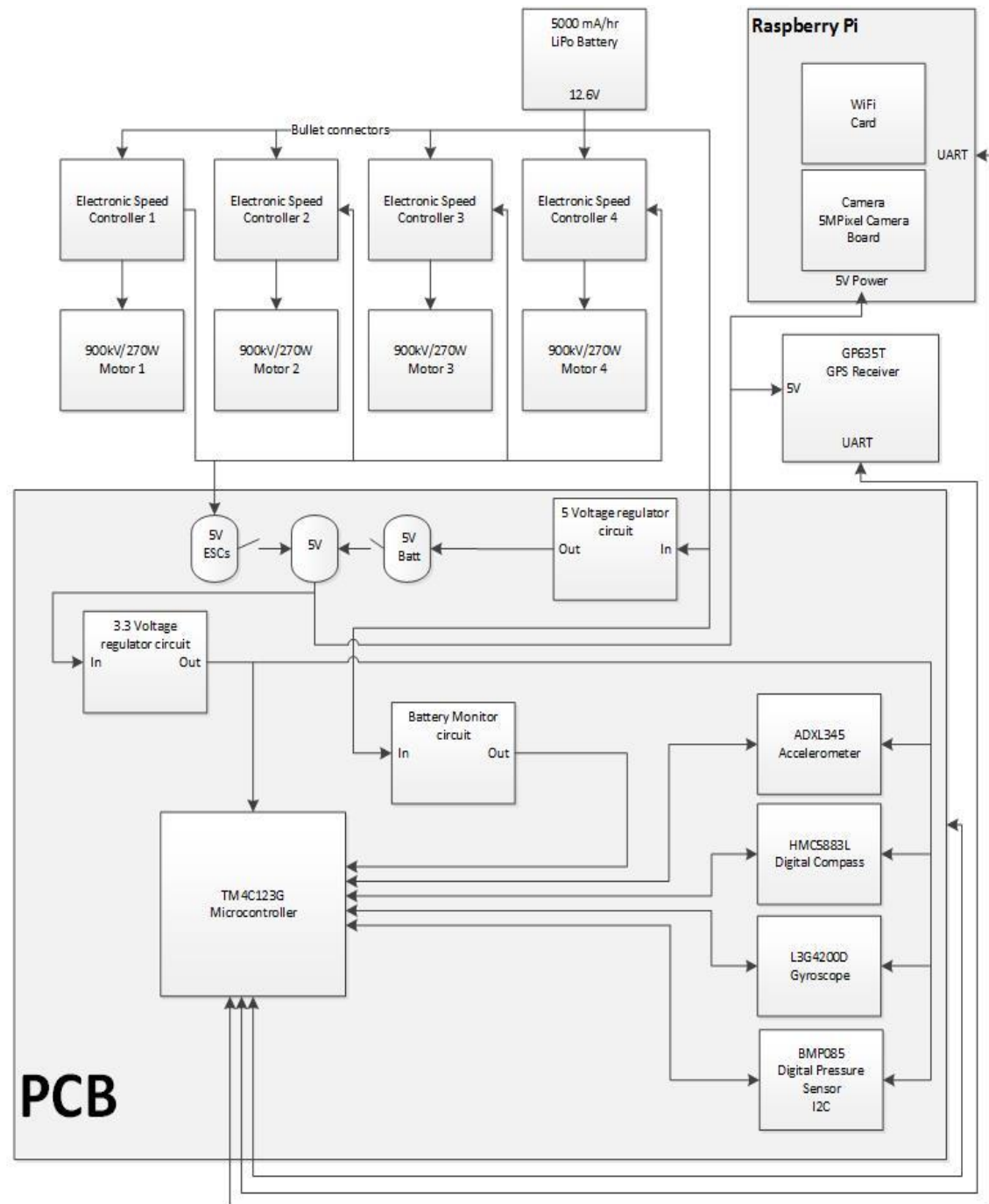


Figure 6: PCB Block Diagram

In the software block diagram shown below in **Error! Reference source not found.**7 on the ext page, there are three different computer systems, each of which have three components. Notice that every block on the diagram has a black arrow to and from the subsystem with which it is communicating. The laptop was equipped with a Linux Script operating system, specifically the OS called Ubuntu 12.04. This particular OS was useful in the amount of open source programs available to the public which can be found online. As previously mentioned, one of the team's end goals was to stitch together images from

the on board camera in order to create an aerial map of a specified area. The team has taken advantage of previously existing programs to help accomplish this task. Two other software components that were implemented on ground station laptop were the disruption tolerant networking 2 (DTN2) daemon program and Telemetry Script, both of were programed to directly communicate with the Ubuntu 12.04 OS on the laptop.

The Raspberry Pi TM used a different operating system called Raspbian. Raspbian is an operating system specific to the Raspberry Pi hardware and uses the same Linux script language for communication the Raspberry Pi will have 3 software components, namely DTN2 Daemon software, Telemetry Relay software, and software for the USB Camera.

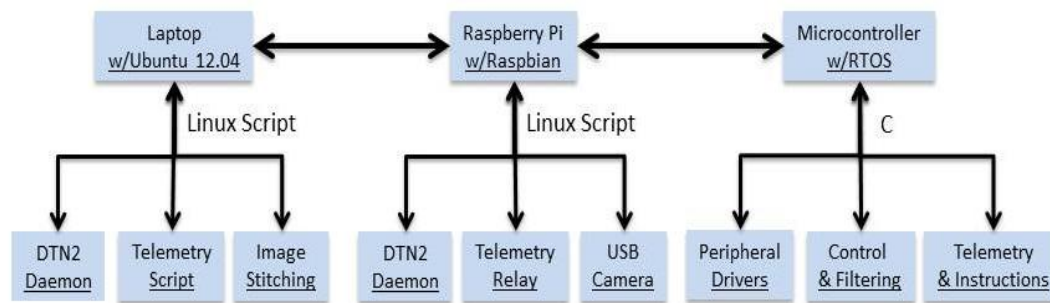


Figure 7: Software Block Diagram

The microcontroller will implement a real time operating software or an RTOS. Its software was programmed in the C programming language. Its software components included software for its peripheral drivers, control & filtering, and for telemetry & instructions. Programming the microcontroller during phase two proved to be a bit trickier compared to programming the quadcopter in phase one. In phase one, the team used a Tiva C TM launchpad with its peripherals connected to the sensors. The Tiva C TM Launchpad has two 64 pin processors, one in the very center of the board and the other centered near an edge on one side. The first processor in the very center of the Launchpad has contains all of the stability algorithm code and was programmed using the second processor on the Launchpad. Because the Launchpad was used to in phase one's prototype, the team merely used a USB to micro USB connector cable which came with launchad and simply plugs from any computer directly into the Launchpad. The program called code composer studio was used to debug and program the microcontroller with the control system which was written with the c programming language. Phase two's custom PCB does not include the second processor, and in this design, the team programmed the processor with a program called JTAG. This JTAG programming process required a separate device which programs the microcontroller through the JTAG pins created on the PCB.

5.2. Flight Computer Software

In this section, the details of the flight computer software are given. This includes discussion about the RTOS, peripherals, I/O, and software block diagrams. The software for this project was designed to be used on top of a Real Time Operating System, RTOS, from Texas Instruments. This RTOS was very useful when it came to the control system, as it allowed the processor to be used more efficiently during delay times, and allowed the control system to assume highest priority when running.

In order for a microcontroller to have any sensing of the outside world, it needs to have peripherals. These peripherals are the sensors that provide state feedback for the control system of the quadcopter for instance. The processor has to make decisions based on the information received from its sensors, and each data set needs to come in at a fixed rate. Since procedural programs are capable of entering into states where the programs can freeze or crash, the team decided to use a real time operating system provided at no charge from Texas Instruments called TI-RTOS. This RTOS provided several different facilities to prevent any task that crashed from crashing the whole system, as well all as provided APIs that allowed for more efficient use of delay time, and last but not least it provided task priorities to ensure that more critical tasks got executed on schedule.

The code for the RTOS from TI was designed to be modular such that certain features and segments could be added or removed by simply editing a configuration file. This allowed for major scalability in the software design for the flight computer. The sections that weren't necessary in the RTOS for the flight controller could be removed in order to save code space, compile time, and run time. Figure 8, shown below and reprinted with permission from Texas Instruments, provides some insight on how the software written for this project would run on top of the TI-RTOS. All the peripherals intended to be used in the flight computer software need to be declared in main () before the TI-RTOS BIOS (Basic Input Output System) is called. The drivers that will be used for this application such as I2C, PWM, UART, and interrupts will be declared in the main function before this call. This allows the CCS IDE to only compile the drivers that are needed. After the BIOS is called, the RTOS takes control of the software and all the tasks are initialized. When the tasks are initialized, any particular application that wants to use a driver it simply makes an API call to the RTOS which then queues the driver task to be executed. TI provides peripheral drivers built into software packages called TivaWare™ so that users don't need to convert all the interfaces to low level binary themselves.

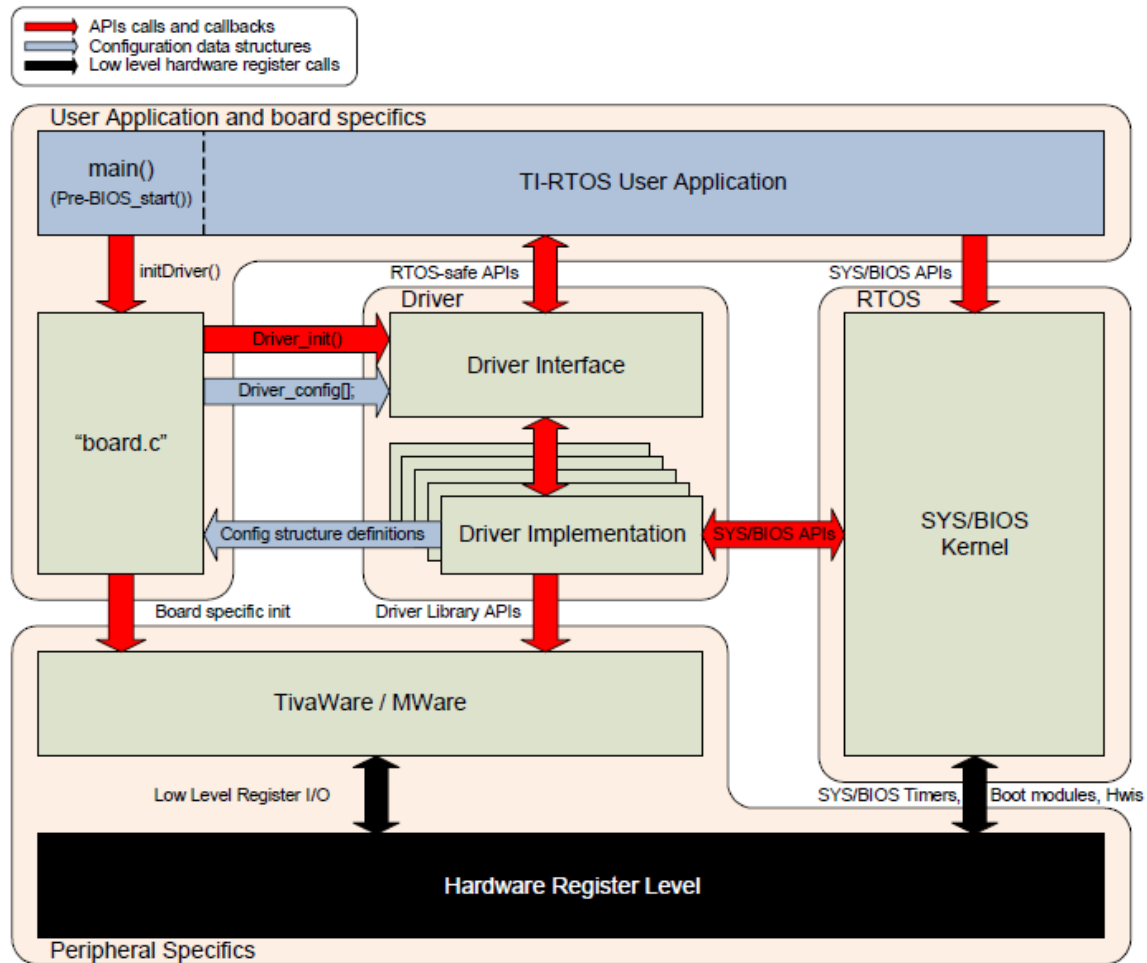


Figure 8: TI-RTOS Overview Diagram, Courtesy of Texas Instruments

The real time operating system is running on the Tiva™ C series ARM® Cortex™ M4F microcontroller, specifically the TM4C123GXL processor. This particular microcontroller, as mentioned before, provided the necessary I/O, memory, speed, and support necessary for the project at a reasonable cost. All the processing of the control system data and the sensor feedback needs to be done in real-time, meaning at a fixed rate. This was another motivation behind using the RTOS in the project. The flight computer system has several peripherals such as an accelerometer, magnetometer, GPS, altimeter and gyroscope that need to be checked at regularly scheduled intervals. Each sensor has a minimum time frame for the data to be sampled. The reason the project needs an RTOS is so it can read all of the sensors in an efficient manor without tying up the processor and waiting for a response. The most important features of the RTOS are scheduling, priorities, and preemption.

Since all of the sensors have to run on a predefined schedule, some of them have to have to be dedicated tasks. Each time a sensor needs to be read, a flag is sent to the operating system to access the appropriate driver. The most important sensors for the keeping the

quadcopter stable in flight are the accelerometer and the gyroscope sensors that make up the IMU system. Using these sensors, the orientation of the quadcopter can be estimated which is critical to the control system. Each of the sensors must be polled at a fixed interval in order to base integral and derivative calculations on the control system as well as attitude determination based on rates from the gyroscope and accelerometer. These sensors, however, have relatively high bandwidths which is why they need a dedicated task. For this project, the sample rate of the accelerometer and gyroscope were set to 100Hz, which seemed to be adequate for the control system. Another thing that is critical to the control system is the fusion of the IMU data, which takes a large amount of computation time. The RTOS has to allow for this computation time, and make sure that the process doesn't get preempted prematurely. Having only these two IMU sensors and the processed data associated with them is critical to keeping the control system satisfied, which enables the quadcopter to fly level.

The magnetometer is not as important for stabilization as the other IMU sensors. The magnetometer can be used to find the attitude just like the accelerometer and gyroscope, but for this project, it was figured that it wasn't necessary. The magnetometer is just used for finding the orientation of the quadcopter in reference to the earth. This is important for finding the heading of the quadcopter which can be used for autonomous flight. The RTOS doesn't need to read the magnetometer as much as the accelerometer and gyroscope, so the bandwidth for this sensor was set lower in order to keep the processing time down.

The altimeter uses temperature and pressure to estimate the altitude of the quadcopter relative to ground or sea level. This sensor is critical to keeping the quadcopter at a stable altitude during autonomous flight. The bandwidth of this sensor is also lower than that of the other IMU sensors. Since this one doesn't need to run as often, the RTOS can focus on other tasks while waiting for this sensor to collect data. This sensor is only important for stabilization of the altitude in flight.

The GPS receiver unit is set by default to take measurements once per second. For the autonomous control, however, it is necessary to increase the bandwidth as much as possible for real time flight path corrections. The unit chosen for this project has a maximum bandwidth of 5Hz, so the unit is corrected to sample at that bandwidth upon startup. Since the GPS data isn't as critical to the flight controller, the data received from the GPS can be read into a buffer and sent out to the navigation computer for processing later.

The analog to digital (A/D) pin on the microcontroller is used to measure the voltage of the battery (after having passed through a voltage dividing circuit). This is one of the lowest priority tasks of the RTOS in the flight computer software. The reason for this is that this data is mostly used for telemetry and small passive control decisions. The voltage of the battery is read whenever a GPS string is received, which makes the bandwidth of this sensor about 5Hz. This task was given a low priority in order to give the other tasks ample time for completion. for the battery will be scheduled to run at the lowest frequency since the remaining battery life cannot be changed quickly. The team is

expecting to have at least ten or more minutes of flight time so the battery only needs to be checked every so often to get a feel for how much flight time is left.

The flight computer software, as shown as an overview diagram is shown in figure 9. As shown below, all the major tasks of the processor are divided up for the RTOS to handle individually. All the tasks go in and out of the RTOS as they are scheduled automatically. The RTOS also handles situations where the processor would be idled, although this situation rarely occurs. Another thing to note is the SysTick module that provides timing services for the RTOS. This module allows any task within the RTOS to have a call-out of the processor relative time in micro-seconds as well as schedule delays for finite intervals. As mentioned before, when one task is put to sleep, the RTOS can switch to another task so that no processing time is wasted. Some of the tasks shown in the figure below run at varying rates and intervals as the demands of the tasks require.

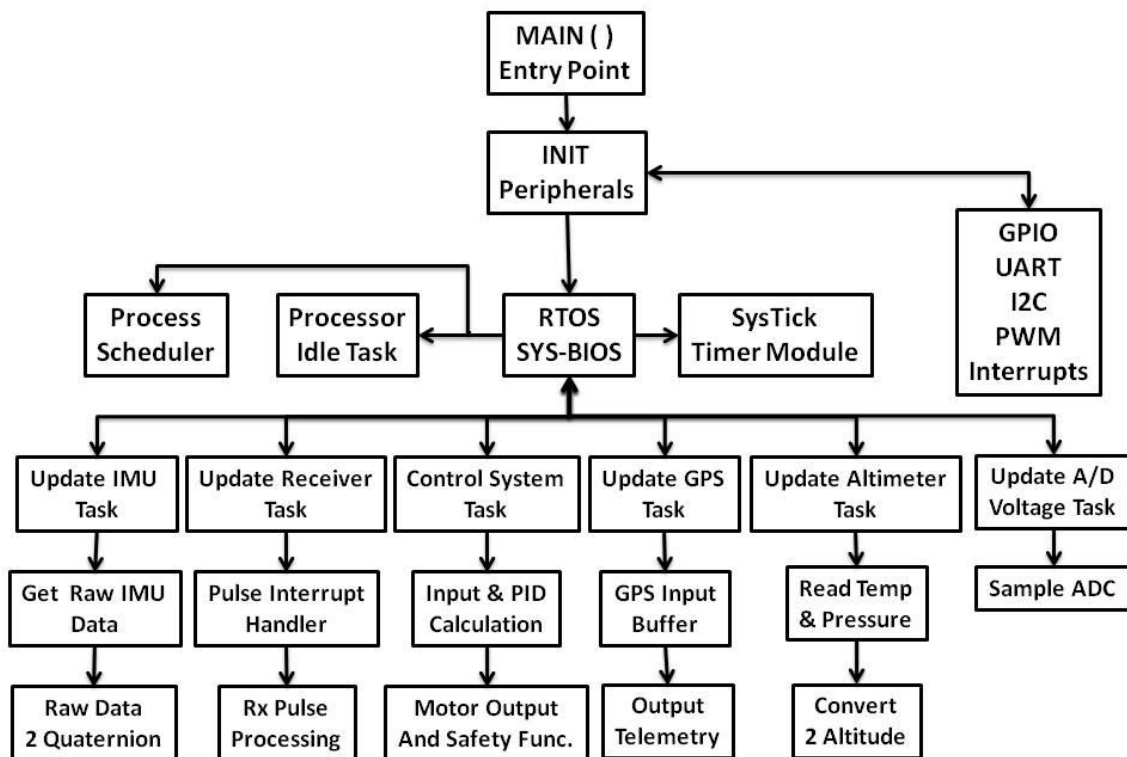


Figure 9: Flight Computer Software Overview Diagram

All of the tasks mentioned previously have a priority level assigned to them in the RTOS which allows the RTOS to make decisions as to what tasks to run first. Each task has a priority based on the importance of that task in regards to stabilization. Going off-course, not being at the correct altitude, or having a communication error with the Raspberry Pi TM was deemed less important than stabilization. Having an unsuccessful mission is much better than having the quadcopter crash due to a relatively unimportant task preventing an important sensor read task such as the accelerometer from being read. In

most cases if a quadcopter crashes it's due to stability, assuming it's in a controlled environment. In general it would seem intuitively appropriate to assign priority levels that match the frequency at which the task is scheduled since important sensors need to be read from more frequently. This is something that had to be experimented with in order to get right. The danger here is having an important task starve. For example, if the accelerometer was scheduled to be read from 20 times per second and each read takes 50 milliseconds, then the processor would be infinitely busy and when the task of reading the GPS is sent the processor, the operating system will suspend it since it has a lower priority than the accelerometer task. The quadcopter would not be able to use the GPS in this case. On the other hand if the GPS had a higher priority than the accelerometer and it came into the processor but took more than 500 milliseconds to perform then the accelerometer would not have been read in a long time and the quadcopter could become unstable and crash. In the first example, if there were only 10 reads per second then the situation would not occur. In the second example, the GPS would have to be quicker or scheduled differently. Figure 10 shows the peripheral's priorities in order with the top of the pyramid having the highest priority.

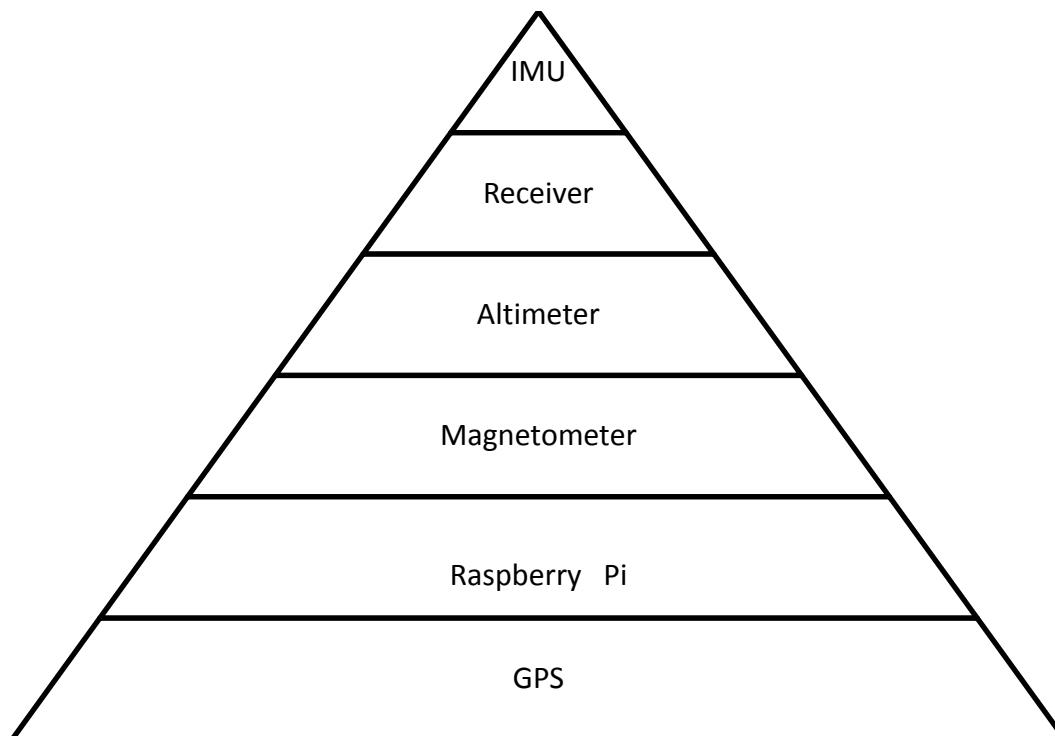


Figure 10: Peripheral Priorities

The preemption feature of the RTOS allows the RTOS to remove a process while it is being processed. This is useful when a process gets stuck, a peripheral stops responding, or when something more important needs to be processed immediately. A process can get stuck for many reasons. For example, if the code being executed enters an infinite loop, the processor would be held up processing the infinite loop and no other processes could run such as the gyroscope sensor, and this could cause the quadcopter to crash.

With preemption the RTOS allows the process to execute for a certain amount of time. If the process is not complete when the time is up, it will remove the process and allow the next scheduled task to execute. This would prevent one task from stopping all of the other tasks that need to be executed on time. Communication protocols such as I2C are being used to communicate to peripherals. With I2C there is a START bit and a STOP bit. If the peripheral device malfunctions during a read such as not providing a STOP bit then it could hold up the process. Since the RTOS has preemption, it is able to remove the process communicating with I2C and allow the next process to begin even if the peripheral device does not respond. Safety features could be added here when this happens to let the quadcopter descend gradually if it was not a critical sensor that failed. If it was a critical sensor such as the accelerometer, the microcontroller can try to restart the communication, but in that case the quadcopter might be lost anyway.

Preemption is also useful to have in the RTOS for this project to stop a process if a time limit was exceeded or an error occurs in the process. This can happen from time to time, and it is very critical that the RTOS be able to handle this situation. For example, if the quadcopter is flying and gets hit by a bird or runs into an object just before it is about to communicate to the Raspberry Pi™, the microcontroller would have to stabilize itself as soon as possible in order to return back to a stable condition. If the communication procedure with the Raspberry Pi™ normally takes one to two seconds, that might be too long for the quadcopter to wait in order to regain flight control. When a situation like this occurs, the operating system will cancel all nonessential tasks, preempt the current process, and focus on stabilization before trying to communicate with the Raspberry Pi again. In order for this to happen, code had to be written in the application layer for the essential drivers to take over based on the state the quadcopter is in.

The peripheral drivers were written with the help of the examples provided with the Tiva™ C microcontroller such as the I2C and UART examples. Each device uses similar underlying drivers to access the communication lines, but has its own APIs for reading and processing. Multiple devices are using I2C but each device has its own address and requires different processing techniques. This is why a separate API will be written for each device. The sensors used for this project are the ADXL345 Accelerometer, ITG-3200 gyroscope, HMC5883L magnetometer, BMP085 altimeter, and GP-635T GPS. The receiver unit is used to provide user input to the controller. The accelerometer, gyroscope, and magnetometer were all on the same sensor stick for prototyping, but were integrated individually into the PCB. The power saving features that some of these sensors provide were not needed because they needed to be fully active throughout the entire flight. The power consumption of the sensors was negligible compared to the motors and the Raspberry Pi™ anyway. All of the sensors that are using the I2C protocol communicate using a standard I2C bus speed of 400 KHz.

The microcontroller communicates with the accelerometer via I2C at the slave address 0x1D. The accelerometer has four resolution ranges: $\pm 2g$, $\pm 4g$, $\pm 8g$, and $\pm 16g$. Since the accelerations that the quadcopter experiences are relatively small, the $\pm 2g$ range is appropriate. This is done by setting the OFSX, OFSY, and OFSZ registers to 0x7F because the scale factor is 15.6 mg/LSB in two's complement form. When the whole system boots up, the microcontroller initiates the self-test on the accelerometer. This is

done by setting the data rate to 800 Hz (400 kHz transfer rate) which requires the rate bits D3 through D0 in the register BW_RATE to be 0x0D. The accelerometer needs to be in normal power mode by clearing the LOW_POWER bit. Bit D3 of the DATA_FORMAT register needs to be set to 1 so the accelerometer will output at full resolution and Bits D1 and D0 need to be set to 0 in the DATA_FORMAT register so the range is in $\pm 2g$. Finally Bit D7 needs to be set to 1 in the DATA_FORMAT register to enable the self-test. When the self-test has finished Bit D7 will be set back to zero. The Accelerometer Block Diagram is shown below in Figure 11.

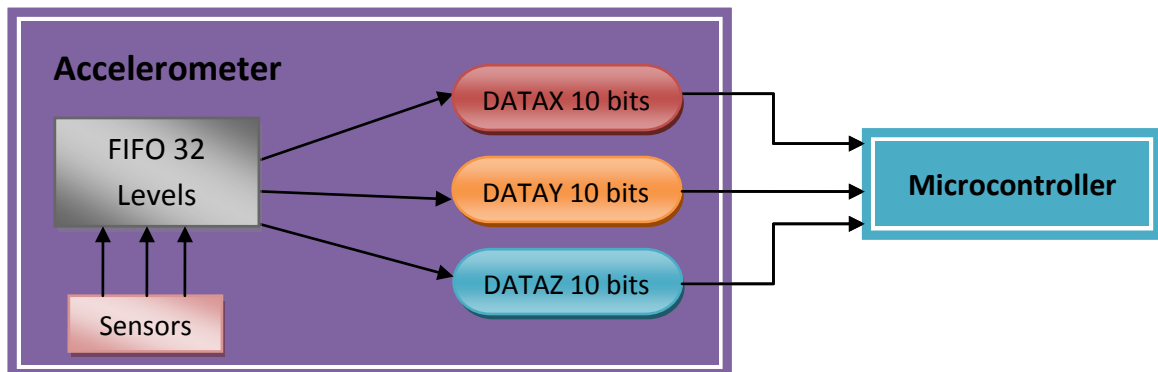


Figure 11: Reading from Accelerometer

The information that the accelerometer reads is temporarily stored in a memory management buffer called the FIFO. The FIFO is set in stream mode by setting the FIFO_CTL register bits D7 to 1 and D6 to 0. This fills the latest 32 measurements and overwrites old information in the FIFO as new information comes in. The reason why this mode was chosen is because the old data is obsolete as soon as new data is taken. It does not benefit the quadcopter to know what the accelerometer history was because it is essentially streaming the new data into the microcontroller since the events will all happen in real time. To read data from the accelerometer, the microcontroller accesses registers DATA_X, DATA_Y, and DATA_Z for the roll, pitch, and yaw types of motion respectively, from the accelerometer. This is shown in Figure 11 above. Once the registers are read, the FIFO automatically fills the registers with new data. The microcontroller must wait at least 5 microseconds before reading the sensor again to ensure the new data has been fully transferred to the registers.

The gyroscope's I2C address is 0x68 if the logic level on pin 9 is low or 0x69 if it is high. The slave address can be read from register zero in the gyroscope chip. The three-axis MEMS gyroscope has built in analog-to-digital converters for each axis that are 16-bits each. The maximum internal sampling rate is 8 KHz. To sample at this rate, the DLPF_CFG (digital low pass filter configuration) portion of Register 22 has to be set to 0x00 and the FS_SEL (full scale selection) has to be set to 0x03. Now the SMPLRT_DIV (sample rate divider) Register 22 has to be set. The formula below shows how to calculate the value needed between 0 and 255.

$$\text{Sample rate} = \frac{\text{Internal gyro sample rate}}{\text{SMPLRT_DIV} + 1}$$

Since a maximum sampling rate is desired, setting register 21 to 0 will result in a sample rate of 8 KHz. The table in Figure 12 shows the data output registers. These half-registers are all in 2's complement format since the reading can be positive or negative. Counterclockwise movement is positive in respect to each axis. The microcontroller uses all of them at the same interval.

<u>8-bit Registers</u>	<u>Description</u>
#29 GYRO_XOUT_H	Upper half of the roll (x axis)
#30 GYRO_XOUT_L	Lower half of the roll (x axis)
#31 GYRO_YOUT_H	Upper half of the pitch (y axis)
#32 GYRO_YOUT_L	Lower half of the pitch (y axis)
#33 GYRO_ZOUT_H	Upper half of the yaw (z axis)
#34 GYRO_ZOUT_L	Lower half of the yaw (z axis)

Figure 12: Gyroscope Data Registers

The magnetometer's I2C slave address is 0x1E. The microcontroller must send 0x3D to read from a register or 0x3C to write to a register. The register pointer is automatically incremented by 1 after a register has been read. This reduces the amount of communication the microcontroller has to do with the magnetometer since the microcontroller is always going to read all of the data registers in order. The magnetometer samples data in Gauss which is the unit of measurement for measuring how strong a magnetic field is. The magnetometer will only be used in continuous-measurement mode so new data will always overwrite the old data as measurements are continuously made just like the accelerometer. This is done by setting the Mode Register (MR) bits MD1 and MD0 to 0. The first thing that will be done with the magnetometer when the quadcopter boots up is to initiate the self-test. This is accomplished by setting bits MS1 and MS0 to 1 on Configuration Register A when testing positive bias and changing MS0 to 0 when testing the negative bias. The complete algorithm for the self test is covered in Section 7. The data registers shown in Figure 13 for the magnetometer are similar to the gyroscope because they are separated by two 8-bit registers, high and low. The difference is the order that roll, pitch, and yaw are in.

<u>8-bit Registers</u>	<u>Description</u>
#3 MSB	Upper half of the roll (x axis)
#4 LSB	Lower half of the roll (x axis)
#5 MSB	Upper half of the yaw (z axis)
#6 LSB	Lower half of the yaw (z axis)
#7 MSB	Upper half of the pitch (y axis)
#8 LSB	Lower half of the pitch (y axis)

Figure 13: Magnetometer Data Registers

For flight Configuration Register A, bits CRA5 to CRA6 are both set to zero so none of the samples are averaged per measurement output. Bits CRA4 to CRA2 will be b110 for a maximum data output rate of 75 Hz. The final two bits are both set to zero for normal measurement. Putting this all together the CRA register is set to 0x18. Configuration Register B controls the device gain. This will be set to 0xA0 since only CRB7 to CRB5 set the gain which is 5 followed by zeros. The output values range from 0xF800 to 0x07FF. This is be used in the default ± 1.3 Ga range since the earth's magnetic field does not exceed 1 Gauss. This is done by setting bits CRB7 and CRB6 to 0, and CRB5 to 1.

The digital altimeter is part of the IMU sensor stick, and is also accessed via I2C. The microcontroller uses slave address 0xEF to read from it and 0xEE to write to it. The mode used for this project is the ultra high resolution mode done by reading register address 0xF4 which requires the over sampling setting to be 3. This collects 8 samples, converts them in a maximum of time of 25.5 milliseconds, and has the lowest noise variance. Pressure is measured in steps of 1 Pa (equivalent to 0.01 hPa). The pressure data is 19 bits long. In order to calculate what the actual pressure is from the raw sensor data, the microcontroller needs to read all of the calibration data from the E2PROM. The pressure is then calculated by using the algorithm shown below in figure 19. This is important to include because the device would not be usable without this flow diagram. This has to be converted to a C program for the microcontroller to run it. The altimeter data is used as an absolute altitude. The quadcopter takes samples of the pressure when it is idle at the ground station. This is the reference altitude for the entire mission. After the pressure is calculated through the algorithm, it can be converted to meters with the formula below. The variable p_0 represents standard pressure at sea level which is 1013.25 hPa. The pressure calculated is p .

$$\text{Altitude (meters)} = 44330 * \left(1 - \left(\frac{p}{p_0}\right)^{\frac{1}{5.255}}\right)$$

Figure 14 is a summary of the actual figure in the BMP085 altimeter datasheet which is located in Appendix D, Figure 1. The steps of the algorithm will be copied into the code for the quadcopter microcontroller. The main purpose of each step is the same for both figures. The important thing to point out is that the raw data is not something that can be directly used. It has to be converted trough a series of calculations. The altitude relies on

the pressure data, which relies on the temperature data, which is all calculated with the calibration data.

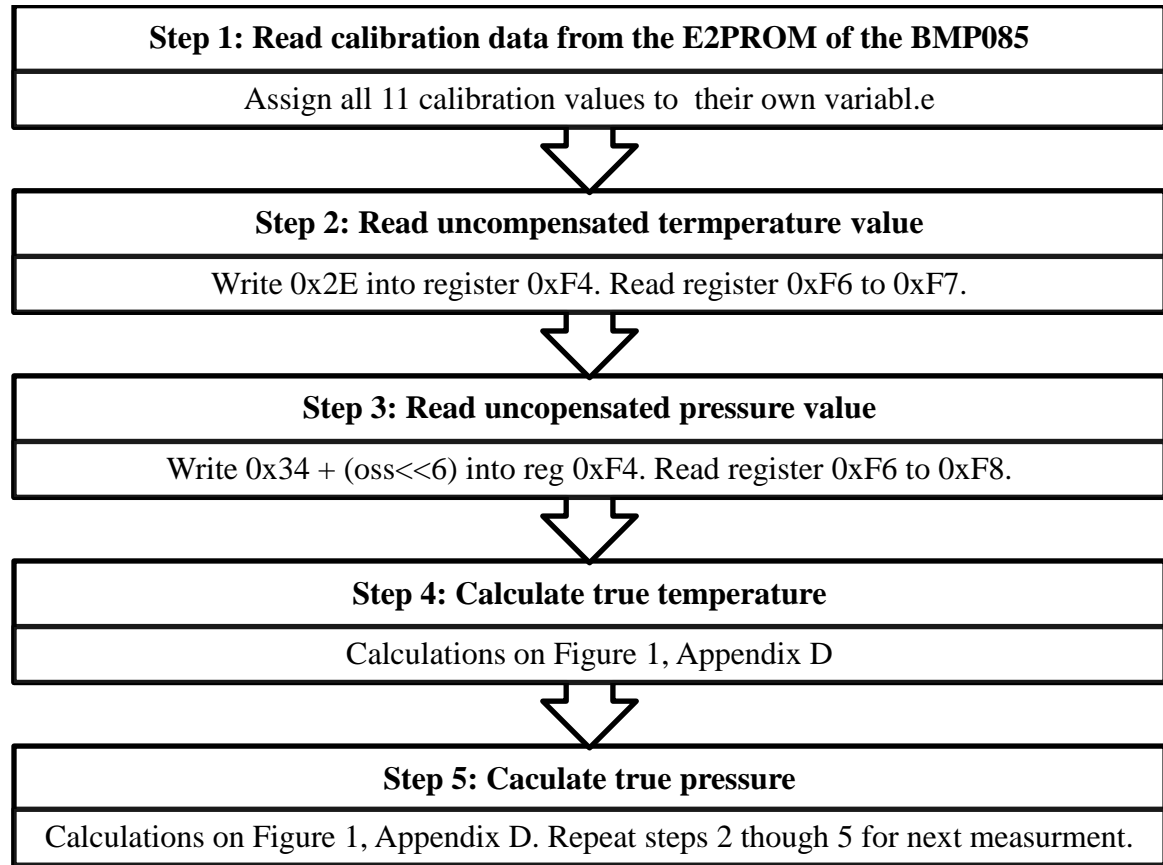


Figure 14: Temperature and Pressure Algorithm

The GPS sensor communicates with the flight computer via UART to the microcontroller at 115200 bps 8 data bits, 1 stop bit, and no parity. The cold start time for the GPS is 27 seconds so the quadcopter has to stay on the ground at least that long before the GPS is ready for flight. It provides GPS NMEA strings at a rate of 5 Hz. There are seven types of National Marine Electronics Association (NMEA) sentences described in Figure 15 on the next page. Each sentence is sent in the following format: \$GPXXX, X,X,X,X...[checksum]. The figure table below provides a definition for each number that would be output separated by commas for each sentence type.

<u>Sentence</u>	<u>Description</u>
GPGGA	Global positioning: UTC time, latitude, north or south, longitude, east or west, position indicator (0: invalid, 1: GPS SPS mode, 2: differential GPS, 6: dead reckoning mode), number of satellites used, horizontal dilution of precision, mean sea level altitude, unit, geoidal separation, unit, age of differential GPS data in seconds, differential reference station ID, checksum.
GPGLL	Geographic positioning: latitude, north or south, longitude, east or west, UTC time, data validity (A valid, V invalid), and mode indicator (A autonomous, D: differential GPS, E: dead reckoning), checksum.
GPGSA	Dilution of precision (DOP) and active satellites: mode 1 (M: manual, A: automatic), mode 2 (1: no fix, 2: 2-dimensional mode, 3: 3-dimensional mode), up to 12 channels displaying the number of satellites used, position DOP, horizontal DOP, vertical DOP, checksum.
GPGSV	Satellites in view: number of messages, message number, satellites in view, ID number, elevation, azimuth, signal to noise ratio, [repeat, starting at ID number...], checksum.
GPRMC	Recommended minimum specific transmit data: UTC time, data validity (A: valid, V: invalid), latitude, north or south, longitude, east or west, ground speed, course speed, date, magnetic variation, mode (A: autonomous, M: manual, D: differential GPS, S: simulation, E: dead reckoning, N: data invalid), checksum.
GPVTG	Course over ground and ground speed: course over ground degrees, reference (true), course over ground degrees, reference (magnetic), speed over ground, unit in knots, speed over ground, unit in km/hr, mode (A: autonomous, D: differential GPS, E: dead reckoning), checksum.
GPTXT	Message from u-blox. This is ignored.

Figure 15: NMEA GPS Output Sentences

It is important to know all seven sentences are sent from the GPS to the flight computer until it initializes. After GPS initialization, the GPS only sends the GPGGA string that contains all the necessary information. This string, along with the heading, voltage, and quadcopter state are sent to the Raspberry Pi TM navigation computer to be parsed and processed. All of this data is sent to the ground station along with the Wi-fi signal

strength and packet number. The ground station performs more analysis on the data if necessary.

The receiver does not have a standard communication protocol like UART or I2C. It has pins used for controlling individual servo motors. Since there was no useful datasheet for the receiver, each pin was tested individually with an oscilloscope in order to determine the best way to receive the data. The pins of the receiver output a pulse width modulated signal train where the falling edge of the current channel overlaps the rising edge of the next channel. In order to process this, all the pulses are setup on GPIO interrupt pins and the time is taken for each rising/falling edge in the pulse train. After the interrupts for each channel have been cycled, they are disabled and the pulse input buffer is saved, and a flag is set. Every so often, the receiver task will process the pulse input buffer and convert it into a control system input.

All of the peripheral drivers were made into functions that work with the RTOS and can easily be called upon anywhere in the application code for requesting data or sending commands. Figure 16 is a quick summary of all the peripheral drivers.

<u>Peripheral</u>	<u>Interface</u>	<u>Address</u>	<u>Self-test</u>
Accelerometer	I2C	0x1D	Yes
Gyroscope	I2C	0x68 / 0x69	No
Magnetometer	I2C	0x1E	Yes
Altimeter	I2C	0xEF / 0xEE	No
GPS	UART	N/A	No
Raspberry Pi	UART	N/A	No
Receiver	Custom	N/A	No

Figure 16: Peripheral Driver Summary

With the RTOS running and all of the peripheral drivers written, the software foundation is complete. Function and API calls are the primary interaction with the higher level flight stability and control code. For example, the process of getting the altitude from the altimeter involves interaction, delays, and calculations. When this is called in the application code, it's a simple function call like: "float height = getAltitude();".

In summary, the flight computer has the RTOS to keep all the tasks running coherently. The flight computer has to read all the sensors, filter the data, relay information to the Raspberry Pi TM, calculate control compensation for all axes, and output information to the motors. All these tasks are difficult to keep running in parallel without overlap, so clever coding techniques and special care was taken during flight computer software development.

5.3. Flight Stability Control System

In this section, the design and analysis of the flight control and stability system along with the details of the sensor filtering algorithms are given. In flight, multi-rotor copters are naturally unstable air vehicles that need to be dynamically stabilized in order to fly properly. This conundrum can be fixed by using a compensation control algorithm which has various control parameters that minimize the error on the roll, pitch, and yaw axes in order to produce stable, predictable motion. When using a compensation control algorithm, however, the gain parameters must be deduced based on the dynamic response of the system. Also, when using a compensation algorithm, however, some form of digital feedback is necessary in order to have an estimate of the orientation. Unfortunately, all digital sensors have inherent noise in the system which is unfortunately unavoidable. This noise, however, can be reduced by using the proper filtering algorithm. The control and filtering algorithms used in this project are analyzed in this section.

One of the first things to establish when designing the control system for the quadcopter is the orientation. The quadcopter must have an established orientation in order to base all the control assumptions on. Figure 17 (II) shows the assumed orientation of the quadcopter with respect to the direction of travel.

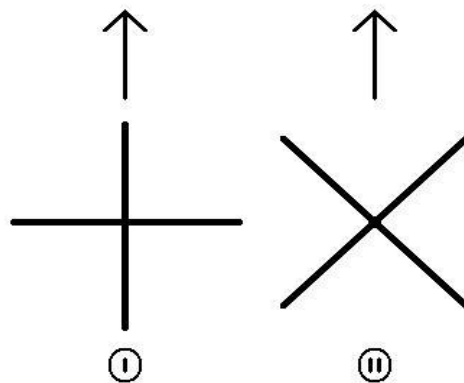


Figure 17: Quadcopter Movement Orientation

In three dimensions, the quadcopters attitude is a vector of its three Euler angles: Roll, Pitch, and Yaw. These angles will be denoted by the Greek letters: ϕ , θ , and ψ respectively. Figure 18 on the next page shows the orientation of the quadcopter in three dimensional space. These conventions will form the basis of the calculations for the control system.

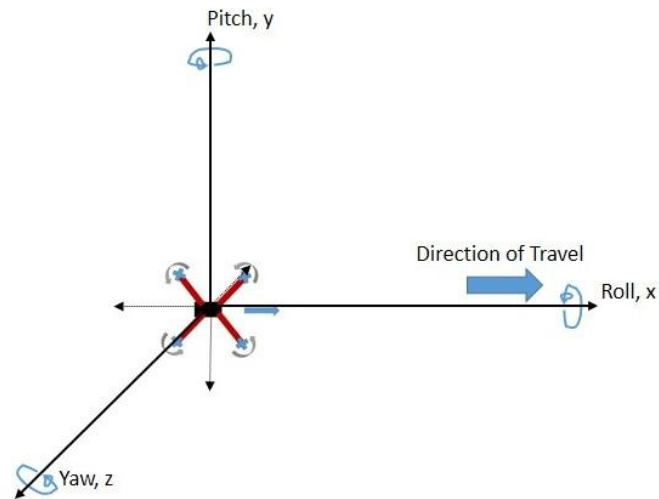


Figure 18: Orientation of the Quadcopter in 3-space

In order to move the quadcopter, a differential force must be applied on either side of the frame using the motors. This differential force creates a coupling moment that causes the quadcopter to spin along an axis. For example, referencing figure 22, if all the motors on the left produced more torque than the ones of the right, than the quadcopter would begin to roll right. Conversely, if the motors on the top half were producing more torque than the bottom half, the quadcopter would begin to pitch up. Changing the attitude of a quadcopter is critical to its motion, as the angle at which the quadcopter is oriented will produce a force vector pointing out which will create momentum along that axis. This principle, illustrated in figure 19, is what creates motion for the quadcopter along a specific axis.

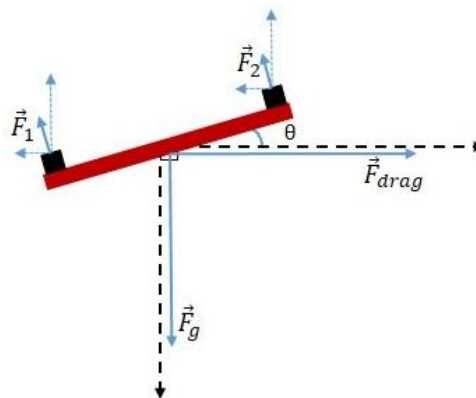


Figure 19: Quadcopter Free Body Diagram in 2-space

The roll and pitch are relatively easy to control on a quadcopter, but the yaw isn't. The yaw in a quadcopter system is caused by the torques from the spinning motors. One of the major things to take into consideration in the design of the quadcopter is that opposing motors should have the same spin (shown below in figure 20), but the two different pairs of motors should have opposite spin. The principle here is to counter the motor torques that can cause the quadcopter to spin out of control. This method does not completely get rid of the rotation, but it does reduce it enough so that the control system can eliminate the rest on the yaw axis. In order to make a controlled yaw spin in any direction, the speed of each motor pair can be adjusted in order to make a controlled torque deviation. This deviation will start the yawing the quadcopter in a controlled manner.

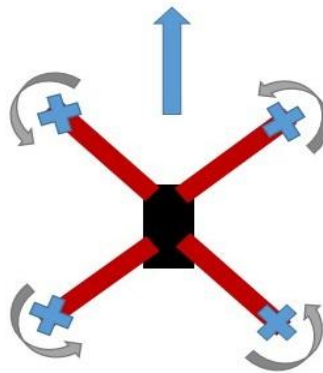


Figure 20: Quadcopter Spin Directions

With the roll, pitch, and yaw conventions established, the control system can now be explained. There are several areas of interest to the control system that include: Stability, Overshoot, Response Time, and Disturbance Rejection. The stability of a quadcopter is determined by its ability to cancel out any oscillation along any axis. This prevents the quadcopter from spinning out of control and crashing. The overshoot and response time of the control system is a fine balance between how fast the quadcopter will response to user input and the amount of time it takes to stabilize once it gets to a desired set-point. This is important for consideration because the control system must be responsive, but not so sensitive that small changes will make large oscillations. In order to control these parameters, a feedback loop will be put in place to control them. The feedback in the system will be done using the gyroscope and the accelerometer which read the angular rate and linear accelerations along three axes. Figure 21 below shows the feedback control system for attitude stability.

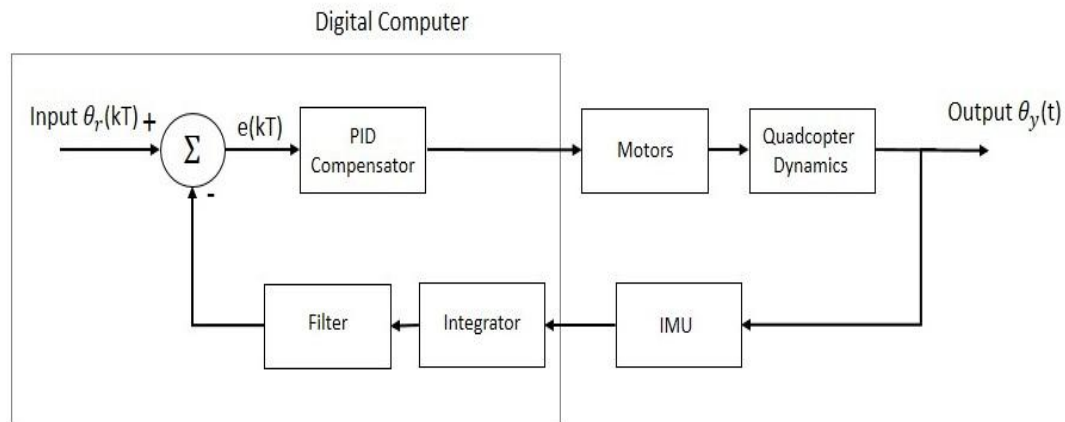


Figure 21: Attitude Feedback Controller

When the vehicle starts up, it will have to find its orientation with respect to earth. This orientation will be the initial state vector as well as the defined origin of the roll, pitch, and yaw. This means that when the roll, pitch, and yaw are all zero, the quadcopter is level with respect to the earth. So, in order to make the quadcopter stable in flight, the feedback controller will have to force the output towards the input. When keeping level, as mentioned before, the input will be zero.

Another thing to note is that the roll, pitch, and yaw each will have their own separate feedback control loop. Since each is a separate parameter, each will need its own loop. There will also be a feedback loop for the altitude of the quadcopter. This will not be done for the phase 1 prototype, but will be done when the focus shifts to the autonomous navigation.

The feedback controllers will be digitally implemented on the Tiva C [™] flight computers for this project. There are several advantages and disadvantages of doing it this way versus analog. The several advantages include robustness, adaptability, environmental stability, and cost. The disadvantage comes in the areas of delay and need for real-time processing. One assumption of most digital control algorithms is that the sample frequencies are always a fixed and never change. This is why the processing of the data on the flight computer must be done in real-time, which was the driving factor behind using the RTOS.

The digital sensors that will provide the feedback for the control system also need some filtering done before they can be used. Unfortunately, there is always noise within sensor system, and the must be filtered to improve accuracy and precision. In order to bring the noise down, there are a few adaptive filters that perform well but the computation time required for them severely limited the flight computer. As mentioned in section three, the

Kalman and the Least Squares filters are commonly used filters to get rid of digital noise. The first, the Kalman filter, is very accurate and greatly reduces noise but is computationally heavy and can hold up the processor for very lengthy periods of time. The other, the Least Squares filter, doesn't perform as well as the Kalman, but it can run somewhat more efficiently on a computer with limited processing power.

Another avenue for the project was to use a running average filter. While the performance of a passive running average filter (which is essentially a low pass filter) isn't as good as one of the adaptive filters explained above, it was found that the performance was adequate for the project. The running average filter is a simple one where the last several variables are saved and when the newest sensor data is retrieved, it is averaged with the last 'n' variables. After performing this action, the last several variables are updated. The principle behind this is to only keep the last 'n' variables so that the earlier data doesn't weight the estimate too much, but it does filter out significant noise.

The equation for this filter is given below, where d_n is the last 'n' measurements, N is the filter order (how many variables saved), and n is an index variable.

$$\hat{d}(n) = \frac{1}{N} \sum_{n=0}^N d_n$$

A block diagram of the operation is given below in figure 22 for the running average filter. This filter is relatively simple and can be implemented with minimal memory and processing power. Essentially what happens is whenever new sensor data is ready, the filter takes the last 'N' samples, averages them, and pushes back the last N samples. For this project, a running average filter of order N=5 was sufficient.

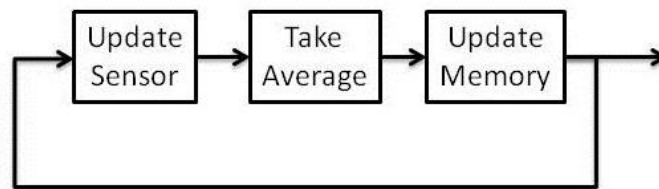


Figure 22: Running Average Filter Block Diagram

The sensors being used for the inertial measurement unit (IMU) include the altimeter, accelerometer, gyroscope, and the magnetometer. Each of these sensors measures specific quantities that are important to the feedback control system. The altimeter measures the altitude of the quadcopter using the pressure and temperature of the surroundings. The accelerometer measures the forces along all the axes using a piezoelectric resistance sensor. The gyroscope works in a similar way, but it measures angular rate over all the axes. The magnetometer measures the heading of the quadcopter by using the earth's magnetic flux as a compass.

The gyroscope being used for this project is a rate gyroscope, which means that it measures angular rate. For the control system, however, the roll, pitch, and yaw angles are used to stabilize the quadcopter, not the angular rates. This means that the rates coming from the gyroscope must be integrated with respect to time in order to get the attitude. Unfortunately, it isn't as easy as just integrating the angular rates because it isn't the same thing as the angular velocity. The angular velocity must be normalized before it can be used to find the attitude.

For this project, a quaternion state estimator was used in order to fuse the accelerometer, the gyroscope, and the magnetometer data into one estimate attitude. This quaternion estimate was then converted to an Euler angle estimate for the control system to use. A quaternion vector is essentially a four axis unit vector that accounts for rotation as the fourth parameter.

Essentially, by converting each of the sensors' data (accel, gyro, magn) into a unit vector, simple vector algebra could be used to add the three together and a specific weight could be given to each. After doing this, the value can be converted to Euler angles for use in the control system later.

After these values are solved for, and the input is filtered, the flight computer can start performing the control algorithms. The input of the control system of this project can come from two sources, either the autonomous guidance system, or the RC controller. The input can be used to throttle the motors up, tilt the quadcopter in any direction, or spin the quadcopter. Using these methods, as mentioned before, is the basis of how quadcopters can fly and maneuver.

The RC controller has a receiver on the quadcopter that has multiple channels that each represents an input to the system. Each of these channels can be used to control a flight parameter of the system. Using this as the input, the flight computer can subtract it from the feedback and obtain an error function that it can use as the compensator input. This can be expressed in equation form, shown below, with $r(kT)$ as the input, $y(kT)$ as the feedback, and $e(kT)$ as the error.

$$e(kT) = r(kT) - y(kT)$$

This is the input to the compensator, which is the heart of the control system. The compensator design is one that can be very tricky, and has many approaches. The compensator has to take the error input, magnify it in some way, and create an output that the plant can understand. The plant, in this case, is the quadcopter and the motors that are controlling the motion. The compensation can be dynamic or it can be static, depending on the needs of the project. For this project, however, the compensation will have to be dynamic so that it can have a fast and accurate enough response time, and damp any unwanted oscillations.

The compensator of choice for this project is the PID controller, otherwise known as the Proportional Integral Derivative controller. This controller gets its name from exactly what it does to the input. That is, this controller multiplies the error input by a

proportional value, integrates it, derives it, and sums them up. The equation below shows an equation for the output.

$$output = u(t) = K_p e(t) + K_I \int_0^t e(t) dt + K_D \frac{de(t)}{dt}$$

Each term in the equation brings some form of value to the compensator. For instance, both the integral and the proportional terms tend to drive the error to zero, but increases instability in terms of oscillations. The derivative term damps oscillations and makes the system more stable, but slows the response and creates steady state error. The integral term by itself also gets rid of any steady state error.

With each term contributing as a whole to PID controller, the brunt of the design is to determine the gains: K_p , K_I , and K_d terms of the controller. There are several methods to determine these terms, and all of them are right, but each has advantages and disadvantages. The team narrowed the search to just two methods, and eventually decided on one of them.

The first method, finding the plant dynamics and using classical control theory to find the values, can be very lengthy and messy, but it guarantees the most efficient control algorithm that works properly. Finding the plant dynamics, however, would involve having to solve all the kinematic and kinetic equations of the system for all the parameters of interest, and turning them into s-domain equations. The plant models can't neglect gravity and air resistance in the form of drag either. These factors play a big role in the dynamics of the system.

The other method, the Ziegler-Nichols method, is a well known method of finding the three parameters based on trial and error. This method, while much easier than the first, still requires patience and time. The trial and error process goes through several iterations before finally solving for all the parameters. The end result of this method isn't guaranteed to be as efficient as the other method either. This method also requires the use of a test setup where the system can be isolated to one axis at a time and be restrained from crashing. Figure 23 shows the process that the Ziegler-Nichols method uses to iteratively solve for each of the PID parameters (Ziegler et al, 759)

	Kp	Ki	Kd
1 st step	Ku	0	0
2 nd step	0.6Ku	2Kp/Tu	Tu/8Kp

Figure 23: Ziegler-Nichols Method Procedure for PID Tuning

Note that the first step in the Ziegler-Nichols method was to set the proportional and integral gains equal to zero. The purpose of this step is to find the ultimate proportional gain at which the output of the system oscillates at a constant amplitude. At this gain, the period of oscillation of the system is used along with the value of the gain itself to

calculate the values of the PID parameters. This process has to be done for every feedback loop for everything that is being controlled by the flight computer.

The PID controller needs to be converted to a discrete-time equation before it can be used on the flight computer. The process of doing that is relatively simple. First step is to assume that the discrete derivative of any arbitrary function used in the system. This is defined in the equation shown below, where T is the sample rate (Franklin et al, 66)

$$\dot{x}(k) = \frac{x(k) - x(k-1)}{T}$$

Using this equation, the PID control equation can be turned into the following discrete equation shown below which can be implemented digitally:

$$u(k) = e(k-1) + K_p e(k) + K_i T e(k) + K_d \frac{e(k) - e(k-1)}{T}$$

This discrete equation can easily be implemented in software on the Tiva C TM flight computer. The algorithm for realizing this transfer function is given below in Figure 24. The recursive PID function will be called for each control parameter, and will repeat every time a new sample comes in. There will be a PID function in the code memory that accepts parameter structures, and each parameter structure will have at the most recent value, the gains of the PID, and the sampling time.

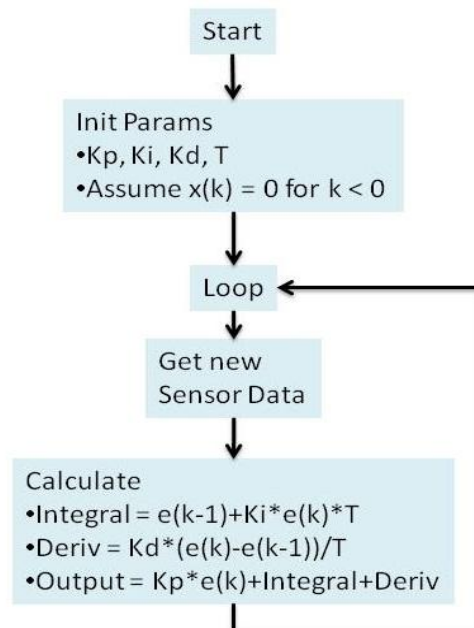


Figure 24: PID Control Algorithm

After the PID algorithm is done for each control input, the compensator output has to be sent to the motors. Unfortunately, the most microcontrollers, including the one used in this project, don't have a port for interfacing directly to a motor. Most microcontrollers do, however, have Pulse Width Modulation (PWM) which can be sent to an intermediate circuit which will translate the signal into a current that the DC motors can use. The intermediate circuit, called the electronic speed controller (ESC), sources a current to the motors that is proportional to the duty cycle of the PWM signal. The PWM principle is shown below in Figure 25.

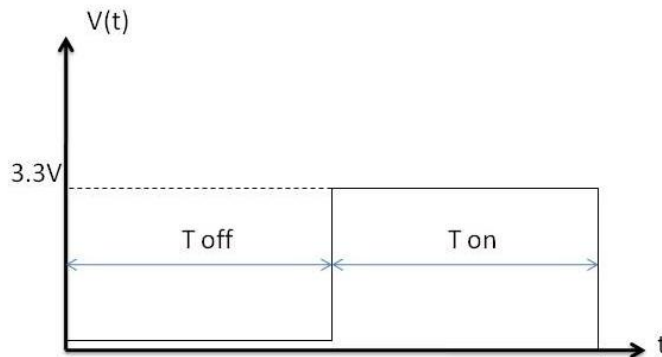


Figure 25: PWM Signal

The basic assumption is that the total time is always constant, and never changes. This means that the time on and time off the can vary, but the addition of the two times must always stay the same. This also means that the frequency is also held constant while the duty cycle changes. ESCs can take this signal and use some transistors and passive filters to turn the signal into a proportional current for a motor.

The AFRO 20A ESCs from Hobbyking.com will be the ESCs for the quadcopters in this project. These ESCs can source up to 20 amps, which is more way more than the motors being used in the project. Each motor needs its own ESC, so each quadcopter must have four ESCs. The PWM frequency of this particular ESC is 51Hz. This means that the PWM signal coming out of the flight computer must have the same PWM frequency.

During autonomous navigation, the control system will be taking input from itself essentially. This means that there will have to be pre-programmed flight maneuvers, as well as altitude feedback control. This will become taxing on the flight computer, as many calculations will have to be done in a very short amount of time. The guidance state machine, which will be explained in the next section, will handle which state the quadcopter is in so that the control system can be adjusted accordingly

In summary, there will be a feedback control loop for every axis and every parameter of attitude and movement. The attitude feedback controller should be able to perform well when the input is the RC controller or the autonomous navigation. The sensors on board the quadcopter will all have to filtered to get good data. The Recursive Least Squares

filter will be used for all the high noise sensors. The compensator of the feedback controller will have to output PWM signals to the motors so that they can be controlled digitally.

5.4. Autonomous Navigation & Navigation Computer Software

In this section, the plan for implementing the control system will be shown along with the navigation computer software. For each quadcopter, there will be a navigation computer which will be the Raspberry Pi TM embedded Linux platforms. The Raspberry Pi TM runs Raspbian, an ARM optimized variant of the Debian Linux distribution. Since the Raspberry Pi TM runs a Linux operating system, it is capable of running multiple processes simultaneously, which is ideal for computationally heavy tasks. These computers have more than enough memory, RAM, and processing power to do all the tasks required for the project. The Raspberry Pi's will be responsible for autonomous navigation, network communication, and imagery. The flight computer, aka the Tiva C TM, will also be assisting in the autonomous navigation portions of the project.

Since the Raspberry Pi TM runs Linux as an operating system, it is capable of multitasking. The multithreading features, and the powerful processor on board the Raspberry Pi TM will be very important to the outcome of the project. These features are especially important because the navigation computer will be handling the autonomous functions, imagery, and the digital network all at the same time. Each task will have a priority and will be running in parallel. The highest priority job is to navigate the quadcopter. Under that, the computer will have to keep taking images from the camera and storing them on the SD card. The lowest priority job will be to send data over the network using DTN2. All of these processes will make up the software of the navigation control system. Below, in figure 26, is the navigation computer software overview diagram. This shows how the operating system fits into the software for the navigation computer. In fact, the applications developed for the navigation computer will run on top of the operating system, while the operating system handles all priorities, resources, and storage.

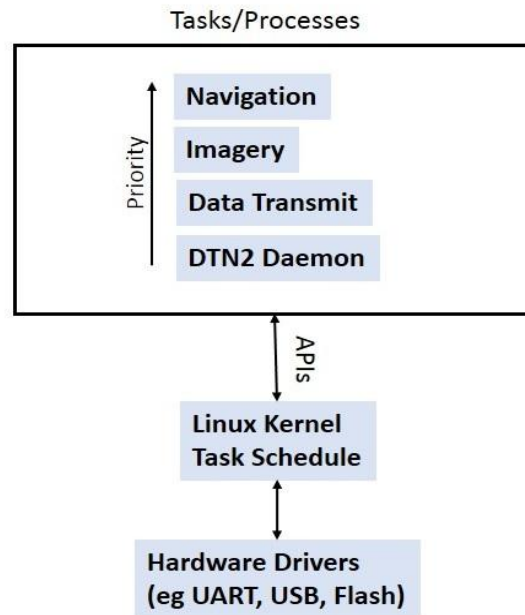


Figure 26: Navigation Computer Software Overview

The main reason behind using the Raspberry Pi TM is NASA's requirement to use the DTN2 software. The DTN2 application has only been developed to work on Linux so far. This means that the DTN2 application would not run on the Tiva C TM without major modifications. So, it was decided that the Raspberry Pi TM would be used for this function. Another goal behind using the Raspberry Pi TM is to make the Navigation computer interchangeable. When the Navigation computer receives information from the flight computer, the goal is to make the dynamics abstracted enough that the Raspberry Pi TM and supporting hardware will be able to be ported to any UAV application. It should be able to make navigational decisions regardless of the vehicle it runs on.

DTN2 is the network layer of the communication network being used in the project. While running DTN2, users can use APIs or user interface commands to send messages, files, or even stream data. Essentially, DTN2 is a virtual router that can be implemented on any computer. This powerful application can prevent data from being lost from bad transmission or delays. More about the DTN2 application will be explained in the next section. The daemon for DTN2 will be running in the background while the computer handles the other navigation and imagery functions.

In Figure 27, the startup process for the Navigation computer is shown. It is critical to make sure that all the applications being used are opened and assigned priorities. The peripherals for the project also need to be opened as well, so that they can be used by the Navigation applications. After a quick system check, the software then moves into a system script that opens and handles the command parser, Navigation, imagery, and telemetry applications. The Navigation computer will have the task of communicating with the flight computer as well as the ground station.

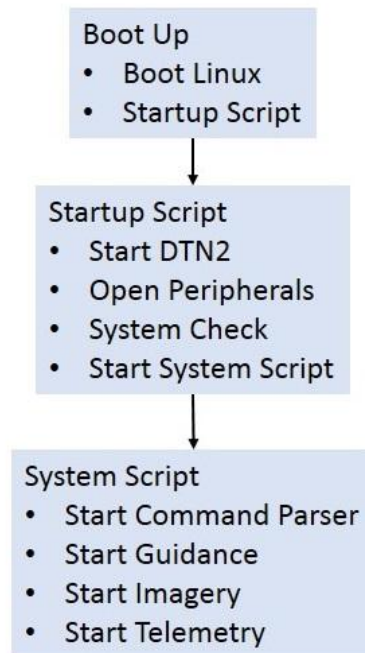


Figure 27: Navigation Computer Startup Process

Most of the software developed on the Navigation computer will be in the Linux native bash script. The reason behind doing it that way is to make interfacing with the peripherals and the DTN2 application much easier. The state machine, however, will have some parts that will be done using the C programming language. C is still far more useful than the bash script when it comes to the computation and logic for the Navigation system. The scripts developed will call on C programs to do computation or logic for certain tasks.

The Navigation computer will be communicating with the flight computer over a serial UART link at 115200 Baud. The Navigation and flight computer will be sending data back and forth to each other on things like state, battery life, GPS coordinates, and altitude. Both computers will have the UART connection interrupt driven, as the communication between the two needs to be able to be done at any time. The strings between the two will have a specific format to be determined at a later date. This will make parsing the data between the two relatively easy to do and should not require too much processing time.

The parser for the Navigation system communication from the ground station will be sent as commands for the system to be interpreted. These commands will be translated into instructions that the flight computer can understand in bite size pieces. Some of the commands will be single step, while others may be multiple steps. Commands such as hover, land, change altitude, return home, or fly to a location will be sent almost directly to the flight computer without much alteration. The single step commands are mostly setup as safety instructions that can change the state of the flight computer state machine in the case of an emergency. The image an area command will be done using multiple

steps. This will be done by making a bread-crumbs trail of coordinates to 'fly-to' that act as waypoints for the quadcopters. When flying both quadcopters, they may also communicate position and other information to each other while in flight so that each may optimize path dynamically. One of the commands that will not be translated for the flight computer is the take a picture commands. This command has no need to be routed through the flight computer, as the camera system is connected directly to the Raspberry Pi™.

While the Navigation computer is parsing the user commands, the system will also be processing the data from the camera and the flight computer. This data will be stored on the SD card as well as transmitted in real-time during flight. The imagery, however, will likely not be sent during flight, and will likely be post processed after flight. The user on the ground station will have access to GPS location, altitude, battery life, velocity, and state. These will be processed in real-time during flight on the ground station and be packaged in a graphical user interface (GUI).

The flight path determination of the Navigation computer is perhaps the most difficult of the Navigation computer software. The quadcopter will have to halt and hover during flight in order to take stable pictures. So, based on the altitude of the quadcopter, and the horizon of the camera, a reasonable assumption can be made as to how far apart the stops must be for imaging. The determination of these stops will be shown in the next section. In this section, however, it is important to note that stops must be made during autonomous flight navigation to allow for imaging. With this in mind, the flight path of the quadcopter will be determined based on what mode it is in. In the fly to location mode, the quadcopter will simply be instructed to take a straight path to its destination. This path will not include any stops for imaging, nor will it provide collision detection. It is important to note that since the quadcopters in the KIRC project DO NOT have object avoidance software, operators MUST NOT draw a path that has objects in the way. This, however, can be avoided, as the cruising altitude can be set above most objects during flight. In the image an area mode, the Navigation computer will set several pit stops along the path where it will stop and take in image. In order to scan a whole area, the Navigation computer will begin to scan the area in a serpentine pattern. Figure 28 on the next page shows how the flight path might look for a single quadcopter in image area mode.

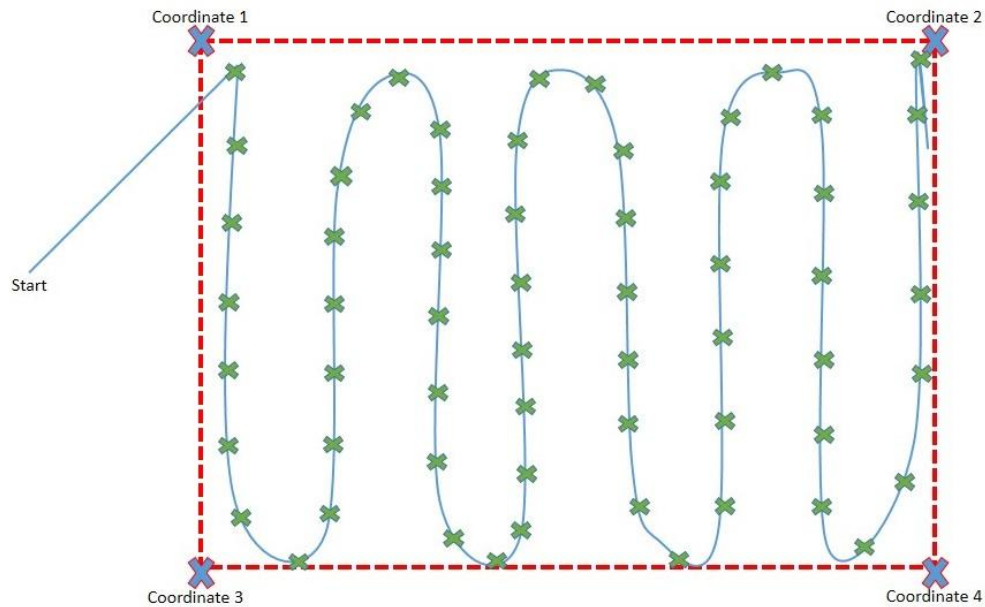


Figure 28: Example of Area Imaging Flight Path

When both quadcopters are being used, both will have the task of imaging an area. Each quadcopter will have half of the area to image. Cooperatively, they will have to decide how to take the area and divide it up. One of the quadcopters will have to be the master one that decides for both, as this will resolve any conflict where both quadcopters try to fly in the same area. During flight, the quadcopters will relay information to the ground about their respective states. Through the use of DTN, the quadcopters can also route through each other in the case where the ground station is out of range.

The Navigation computer essentially acts as an abstraction barrier for the flight computer that will be handling the brunt of the work. The flight computer, after the stability algorithms have been established, will be reprogrammed to include software support for reading GPS, sending telemetry strings to the Navigation computer, and processing a guidance state machine. The state machine, shown below in Figure 29, is the workhorse of the Navigation system. The navigation computer can command the flight computer to change states for the system at any time. The GPS strings will be sent from the flight computer directly to the Raspberry Pi TM. The Raspberry Pi TM will be doing all the processing of the GPS strings.

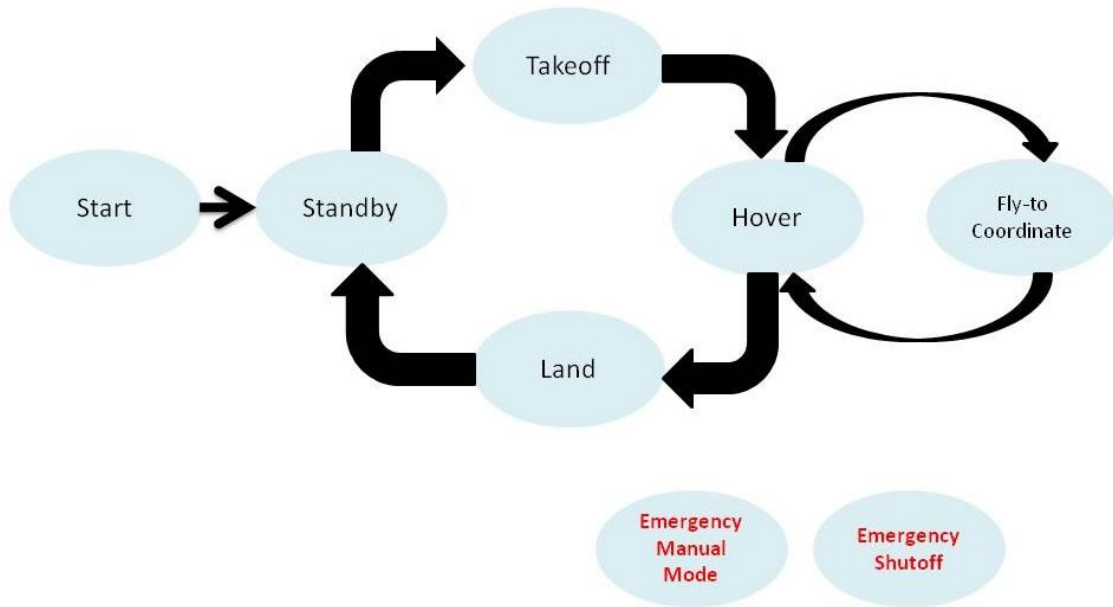


Figure 29: Navigation State Machine

When the flight computer first boots up, the quadcopter will be put into a standby mode while waiting for instructions from either a manual controller or the Navigation computer. In standby mode, the quadcopter rests on the ground with the motors throttled off. This mode acts as a default safety mode when the quadcopter is idling. After this mode, the quadcopter is set into takeoff mode, where the flight computer shifts its primary computing power towards getting to a set altitude while maintaining stability. The taking off and landing states are the most difficult states because the quadcopter is experiencing instability. Once the quadcopter has reached a desired altitude, the quadcopter will shift its state to hovering. In this state, the quadcopter solely tries to maintain altitude and attitude in order to stay level. The hover state is the default state during flight, as it is the most stable when off the ground. The next state can either be to fly-to a coordinate, or to land the quadcopter. When flying to coordinates, the default to return back to hover when the destination is reached. After the quadcopter has completed its mission, it will shift to a landing state, where it will use an on-board ultrasonic range-finder in order to land the quadcopter on the ground. The quadcopter will thereby enter the standby state and safe itself so that the battery can be removed and the flight data can be retrieved.

One of the most important features of the quadcopter guidance state machine is the emergency states that can be reached no matter what state it is in. The quadcopters can either have a complete shutoff or enter manual mode when in flight as an emergency precaution in the case where a flight isn't going as planned. The shutoff mode, planned only for absolute emergencies, will cut off signal to the motors and the quadcopter will drop to the ground from whatever height. The problem with this state is that it will likely destroy the quadcopter if dropped from considerable distances, which is why this mode is preferred not to be used. It is, however, necessary in the case that the quadcopter starts

taking a turn towards the wrong direction and starts going out of control. The other state, manual mode, automatically shifts control of the quadcopter to a user with a RC controller. This mode is the preferred way of dealing with emergencies over completely cutting the motors, which is why there should always be a skilled pilot available to stop the quadcopter from crashing just in case.

In all the states of the quadcopter's guidance state machine, the flight computer will still be constantly running the PID loop and stabilizing the quadcopter. The only exceptions to this, of course, are the startup and standby states where the quadcopters won't be flying. The other states will be running the loops and constantly be processing data while achieving the tasks of each state.

The most important part of the guidance system, the ability to find a path from one location to another, is critical to whether or not the quadcopter will be able to fly autonomously. The quadcopter will have to be able to take in a GPS coordinate and compare it to its current location, find a vector from its current location to the new one, and fly itself to that point (illustrated in Figure 30). The difference of distance between the two points will create a vector that will help guide the quadcopter to the proper location.

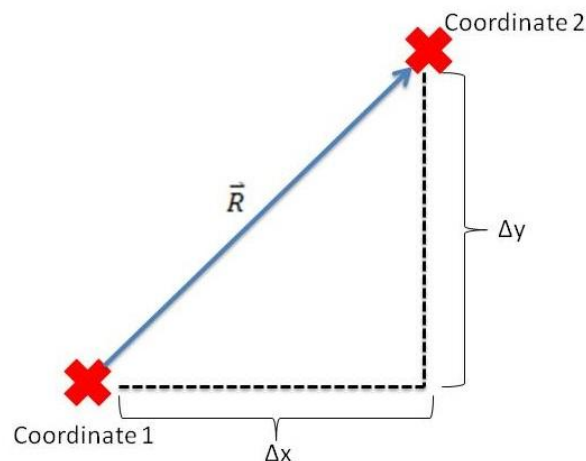


Figure 30: GPS Coordinate to Navigation Vector Transformation

When using GPS, the relative position between two objects can be found by subtracting the distances between the two points. If the GPS coordinates are already translated into UTM (Universal Transverse Mercator) coordinates, the two coordinates can be directly subtracted to find a vector between the two points in meters. This method, however, only works for relatively short distances, as it does not take into effect the roundness of the earth into consideration. Assuming short distances, it is possible to use this 'flat earth' model in order to simplify calculations considerably.

Each GPS coordinate comes with two parts, the north and the east coordinates. These two coordinates can each be stored as a floating point variable in RAM. To find the distance north that the quadcopter must travel in order to arrive at the destination, it must subtract the north coordinate of the current position from the north coordinate of the desired position as shown in the equation below.

$$\Delta_{north} \text{ (in meters)} = \Delta y = P_{2 \text{ north}} - P_{1 \text{ north}}$$

Similarly, the distance east can be found by subtracting the east coordinate of the current position from the east coordinate of the desired position as shown in the equation below.

$$\Delta_{east} \text{ (in meters)} = \Delta x = P_{2 \text{ east}} - P_{1 \text{ east}}$$

Using these two distances, a vector can be formed that will guide the quadcopter to the proper location as shown in the equations below.

$$\vec{R} \text{ (in meters)} = (\Delta x)\hat{x} + (\Delta y)\hat{y} = R\hat{u}$$

$$R = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

$$\hat{u} = \frac{(\Delta x)\hat{x} + (\Delta y)\hat{y}}{\sqrt{(\Delta x)^2 + (\Delta y)^2}}$$

With this, the vector can be used to find a heading and distance to travel which will be the basis of the control system that will be used to navigate the quadcopters between points. The distance between the two points is the error that will be minimized, as the quadcopter moves towards the destination, this is shown in Figure 31. The quadcopters will use the magnetometer to find the heading and make sure that the quadcopter is moving in the proper direction. The speed at which the quadcopter moves between the two points will be capped and that value will be determined at a later time. The control action to get to each point determined by the attitude stability control portion discussed a few sections earlier. While flying between two points, the quadcopter will also be using feedback to maintain altitude as well in order to prevent the quadcopter from falling.

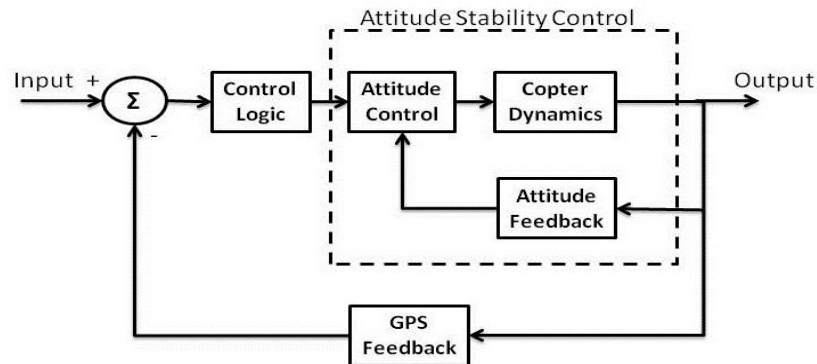


Figure 31: Navigation Feedback Diagram

Most GPS units have a low bandwidth, and a very low resolution for most commercial application. With feedback coming in at only at a few hertz, and the data being off by up to several meters, tolerances must be made in order to make it work. Since the current position of an object as determined by a commercial GPS unit can be off by several meters, it is assumed that the final destination is a range around a specific point rather than the specific point itself. This means that there is a bubble range where the quadcopter can say it has reached its destination. In order to counter the low bandwidth of the GPS, the maximum cruising velocity will be capped. Capping the velocity will help prevent overshoot and inaccurate headings. As the quadcopter approaches the target, the quadcopter will also begin to slow down. This will be accomplished by turning the quadcopter in the opposite direction of travel in order to counter the forward momentum. This will slow down the quadcopter.

Figure 32 below shows the control logic for the point-to-point Navigation system. This simplified logic diagram shows the continual looping to find the error vector and fly to that point. The diagram does not depict, however, the complex speeding up and slowing down portions of the logic.

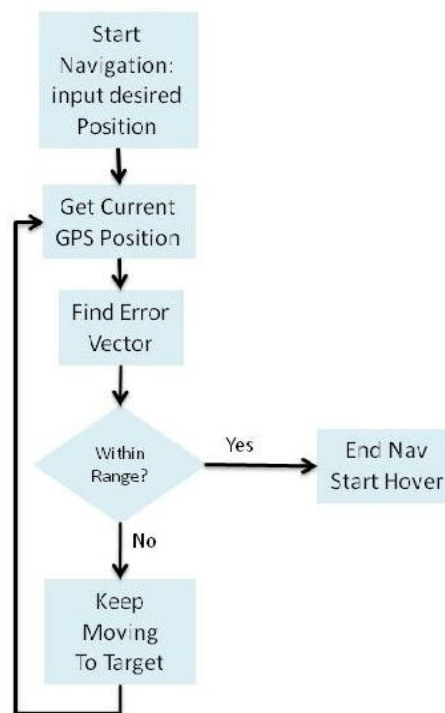


Figure 32: Control Logic for Navigation System

The guidance and navigation portions of the project can be very tricky. The plan is to use both the Tiva C TM and the Raspberry Pi TM to do these tasks respectively. The Tiva C TM will have the guidance state machine that helps guide the quadcopter to specific points while the Raspberry Pi TM will have the task of navigating the quadcopter through all the coordinate 'way points' and collect the data in the process.

5.5. Imagery & Network Architecture

In this section, the design of the imagery system and the network architecture are given. Although each of these topics is slightly different, they are both going to be implemented on the same system. The imagery system and the DTN2 application will both be running on the guidance computer (the Raspberry Pi™). Both are low priority applications that run when the processor has time for them. The DTN portion of this section includes the overview of how DTN2 works, how implemented in the project, and how it is called in a program.

One of the major bonuses of the Raspberry Pi™ is the never-ending availability of extra hardware add-ons. The Raspberry Pi™ has a camera system available to it that plugs straight into one of its ports and works relatively well. The five mega-pixel camera module was specifically designed to work with the Raspberry Pi™. The Raspberry Pi™ can even address it using the operating system file system of attached devices. There are built in APIs for accessing the camera and taking images and video which make interfacing to the camera module simple and easy. When the camera is mounted, images can be retrieved and stored on the SD card of the computer. Overall, the goal was to keep the imagery system as simple and cost effective as possible while still achieving the major goals of the KIRC project.

When setup properly, the camera will be able to be addressed in the /dev directory on the Raspbian operating system. The plan is to use this feature to take images remotely with the Raspberry Pi™ during flight. The cameras will be facing down and attached to the bottom of each quadcopter in flight. The mount will have to be done carefully as not to pose the risk of damaging the camera in flight. Figure 33 shows how the mount might look while in flight. The mount will be put under the battery which is underneath the frame of the quadcopter.

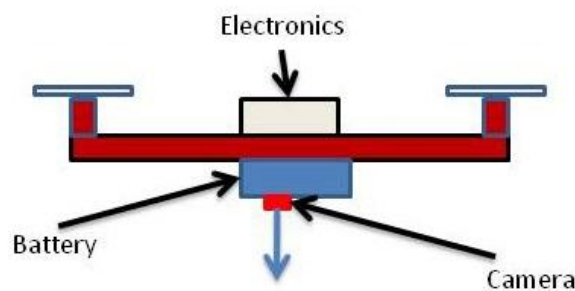


Figure 33: Quadcopter Imagery Mount Plan

During flight, the guidance computer software retrieves the images from the camera on demand, stores them on the hard drive, and if possible sends a sample set to the ground. Images will be saved in JPEG form so that they may be easily be viewed by any

computer. Image stitching is done on the ground computer, by post processing the images. Each of the images has a GPS coordinate stamped for processing later.

The ground station and the two quadcopters, KIRC and SPOK, will be connected on a digital wireless communication network. The Raspberry Pi TM uses an IEEE standard 802.11g wireless card in an ad-hoc configuration to form a mesh network. The DTN2 application acts as the virtual network layer for the system. Using this application gives the computers the ability to dynamically route links and store-forward bundles of data between any of nodes. DTN, or delay tolerant networking, is made to excel in situations where links are intermittent or opportunistic. The DTN2 protocol only works on the Linux operating system for now, so this was the major reason behind using Raspberry Pis TM for the project.

DTN2 is a networking protocol that was developed recently and has been steadily growing popularity in the space industry. It is an open source software freely available from dtnrg.com and is written in C++. The software is still under heavy development, but is operational at this point as a viable networking option. DTN2 is essentially an overlay virtual networking protocol that operates on the network layer as a virtual router. The protocol holds support for convergence layer adapters such as TCP and UDP, allowing it to be used with higher level applications. It also comes with a daemon for use on the command prompt and several APIs written in C++ for use in custom programs. For this project, the plan is to use the daemon in the command prompt, because interfacing with the APIs might prove to be too difficult for the team to manage. The overview of the DTN2 software from an academic paper is shown in Figure 34.

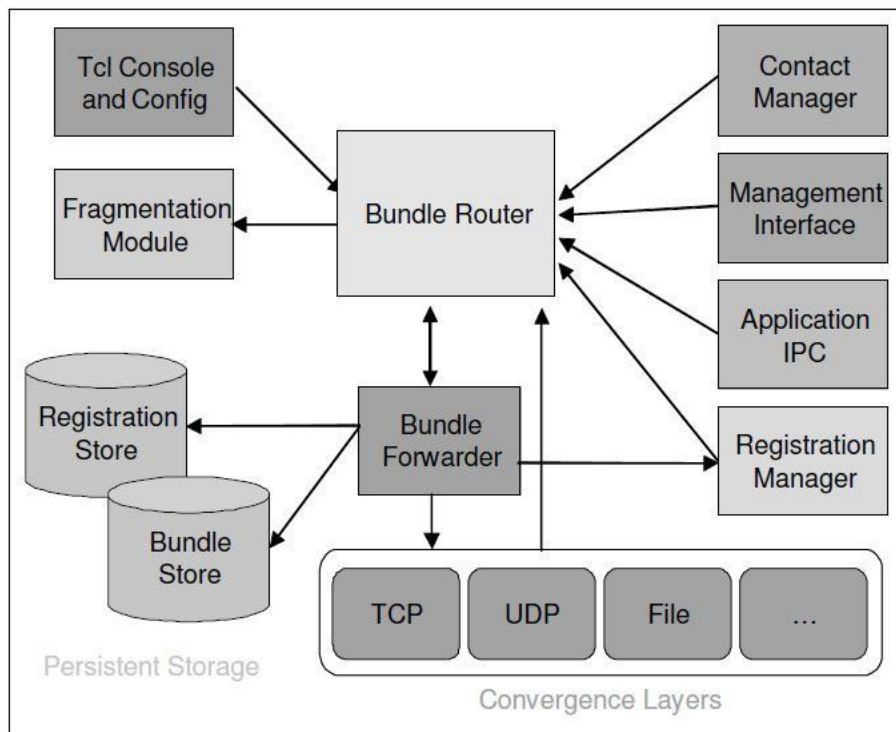


Figure 34: DTN2 overview diagram

DTN2 handles data in ‘bundles’, where each bundle can vary in size from a few bytes to even gigabytes. Each bundle to be sent is stored and forwarded to a destination, where the destination will acknowledge receiving the bundle. This exchange reaction is one of the features of DTN that makes it so durable in rough networking environments. When the sending node does not receive an acknowledgement from the receiving node, it keeps resending the bundles until an acknowledgment is received or the operation times out. The time-out period for DTN2 is set for 1 hour, but it is adjustable depending on the situation. For this reason also, TCP must be used as the convergence layer, as it has support for the send/ack structure of DTN. While DTN2 can be used with UDP, its feature set is limited when using this protocol. Figure 35 below shows a block diagram of the send/ack logic.

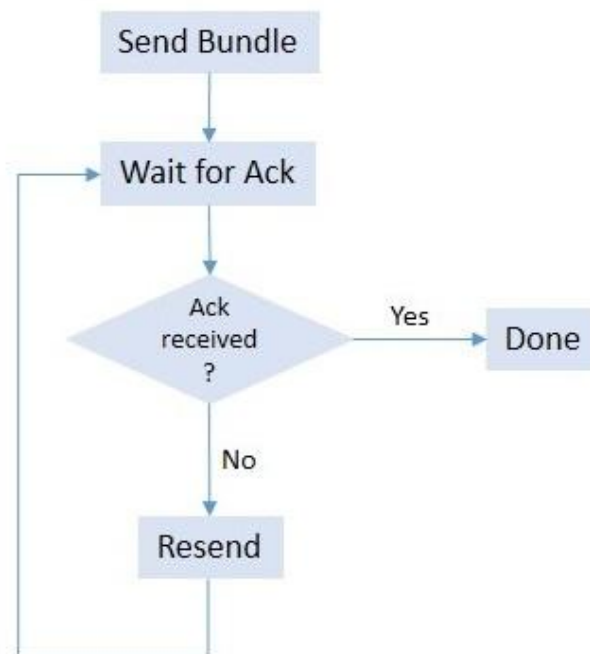


Figure 35: DTN Sending and Receiving Exchange Diagram

Another feature of DTN2 is its ability to dynamically route bundles based on routing table and opportunistic links. When configuring DTN2, a routing table must be established that has all the IP addresses, domain names, and convergence layer information of every node in the network. After all this information is given, DTN2 must be configured to work in a dynamic routing mode. Once setup properly, DTN2 has the ability to route through each node dynamically to create a path to the destination when a direct route is not there. This is especially ideal for situations when links between two nodes are cutoff or have intermittent connections.

While DTN2 is being used in this project, it isn't the only DTN software available. JPL has also developed a low overhead version of DTN called ION, which is short for Interplanetary Overlay Network. This name reflects NASA's future vision for DTN as a deep space telecommunications networking protocol. ION is written almost entirely in C,

and much like DTN2 is still very much in the development phase. The reason behind using DTN2 instead of ION is because DTN2 is more developed and has more features that are useful to this project.

Before using DTN2, it must be installed on the host computer running Linux. For the Raspberry PiTM, the process of installing DTN2 can take quite a long time, but it can be shortened by cross compiling the software on a faster machine. The software is available, as mentioned, from dtnrg.com. DTN2 comes with two separate packages that must be installed in order for the program to work. One is the DTN2 application itself, and the other is a support application called Oasys. Oasys contains the support library for DTN2 that has the utility functions that the program needs. While installing DTN2, there are several package dependencies, so it is recommended to install GCC 3.3 or newer, TCL 8.5, BerkeleyDB 4.8, and Xerces 2.6+ before installation. The install may also require some other packages depending on the machine it is running on. After configuring and compiling Oasys and DTN2, with Oasys being the first, the program is now ready to run.

Before running DTN2, there is a configuration file that must be edited. In the configuration file, the user must list his IP address, domain name, and connection information. The user must also do this for any computer on the network that will be using DTN2 also. This process creates a routing table for DTN2 that can be used to route bundles to the proper destinations. If any changes are made to the configuration file after DTN2 startup, the application must be restarted in order to incorporate the new changes.

DTN2 comes with a command prompt daemon that can be used on the Linux console. The daemon can also be run in the background while other programs can have time to run. The daemon can be opened by typing in 'dtn2' in to the Linux console with the appropriate arguments. The full list of appropriate arguments can be listed by typing in 'dtn2 -help'. Before running the daemon, new bundle database must be initialized. This can be done by using the command 'dtn2 --init-db'. It is recommended to clear and remove any old databases before initializing a new one though. After initializing the database, the daemon can now be run. For this project, the daemon must be running in the background, so the DTN2 application is started up by typing in the command 'dtn2 -d -o dtn.log' into the console. The dtn.log portion creates a log file for the daemon as it is running as a way of checking if there was an error. This process must be done anytime the DTN2 application is restarted.

Once DTN2 is running, there are several commands and applications that will be available to the user on the command prompt. One of the most basic commands is the 'dtnping' command. This command requires a destination argument, and when used it pings the destination repeatedly while waiting for responses. This command is very useful during debugging and helps check for working links within a network. Another command that is available is the 'dtnsend' command. This command sends messages, files, or timestamps to another node within a network. The only downside to this command is that the other computer must be using the 'dtnrecv' command before receiving any bundles. The 'dtnrecv' command is a blocking function that holds up the command prompt until the user decides to quit. There are two application commands, however, that are more suited to this project than the normal sending and receiving

commands. The commands ‘dtncp’ and ‘dtncpd’ are command console functions for sending and receiving respectively. ‘dtncpd’ is the listening function that can be set to run in the background while other tasks are performed. After receiving a file, it is put in a local directory. ‘dtncp’ is the complimentary function to the receiving one. It sends files of any type to any of the nodes on the DTN2 network. These two functions will be used extensively in this project as the standard communication functions.

From start to finish, there are several steps that must be done to configure DTN before it is ready to use. Figure 36 shows the summary of the initialization process of the DTN network for each computer. This process will be streamlined by packaging it into a script for the computer to run that will be called on startup. After running the script, users should be able to start executing other programs such as the guidance system, while the daemon will handle all the communications in the background.

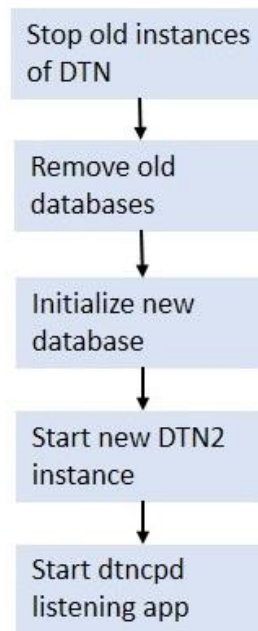


Figure 36: DTN2 Startup Script Overview

For the software on the guidance computers for the quadcopters as well as the ground station laptop, there will be scripts that will handle all the functions of the DTN software as well as the other tasks that need to be performed. A program on each of the guidance computers is used to periodically check if a file has been received, or if one needs to be sent and perform the appropriate action thereafter. The ground station has a similar structure, but it is actively reading the information from both quadcopters and displaying the information live in a terminal as well as a GUI.

In summary, the imagery systems as well as the DTN2 network architecture overview is given. Each of these systems are incorporated into the software of the guidance computers as well as the ground station. The quadcopters use the network as a means to

relay telemetry and control the quadcopter remotely over a user interface. The imagery is post processed by a third party application and is explained in a later section.

5.6. Electric Design & Schematic

In the initial schematic design for this project it was decided to begin with getting from the power supply to the test equipment including the Tiva C™ launchpad, IMU sensor stick, altimeter, GPS Sensor, and Raspberry Pi model B. The easiest way to maintain drop voltage to an amount for powering microcircuits was through the use of voltage regulators. The voltage regulator circuit is shown in figure 37 below.

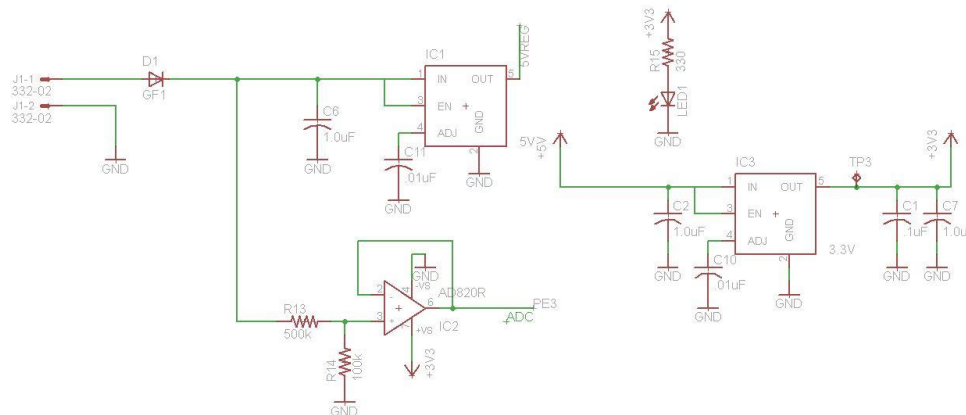


Figure 37: Voltage Regulators and Battery Monitoring

The 15 volt power came in through the J1 power plug, this came directly from the battery. In order to protect the battery a safety diode was implemented. This was followed by a split into two circuits, a voltage regulator to drop the voltage down to 5 volts and a voltage divider into a unity gain operational amplifier for battery monitoring.

The desire was to have the voltage going in to the non-inverting terminal to be less than 3.3 volts. The operational amplifier was implemented in order to cut out the current and send this low current, voltage signal to an available analog to digital converter on the microcontroller and measure the slope in output seen by the battery as the system was in flight. This monitoring was done in order to force a landing of the quadcopter in the event of dangerously low power.

On the other line coming from the safety diode goes into a 5 volt regulator. Just before this voltage regulator is a capacitor placed to maintain circuit in the case of a voltage drop and filter low frequency noise. The 5 volts then flows into a +5VREG net that goes to connectors to choose either the ESCs as the power source or the battery itself for powering other system components, as well as another voltage regulation circuit and verification light emitting diode, LED. In prototyping LEDs are important for signifying what stage of operation the system is in, which is why one is implemented in order to verify proper powering of the system.

This 5 volt jumper circuit was followed by a 3.3 volt regulator circuit. Due to the fact that most of the components on the board are powered by 3.3 volts, if this portion of the system doesn't work properly then none of the system would be functioning properly. For this reason the 3.3 volt regulator circuit was also given a power on LED. In order to keep the other circuitry and this one from being altered by low to medium frequency noise, capacitors were implemented on both sides of the 3.3 volt regulator.

As schematic design moved forward the best way of preparing for implementation was the use of any resources available. A big resource was for some of this design was the suggested circuit design developed for the sensor stick used in testing as well as those design suggestions in the datasheet of many of the components. In addition it made the system design more reasonable. The design started with simplicity and moved forward to more complex network design from there. The IMU sensor circuitry is shown below in figure 38. The circuits themselves were derived from some similar product from Sparkfun.

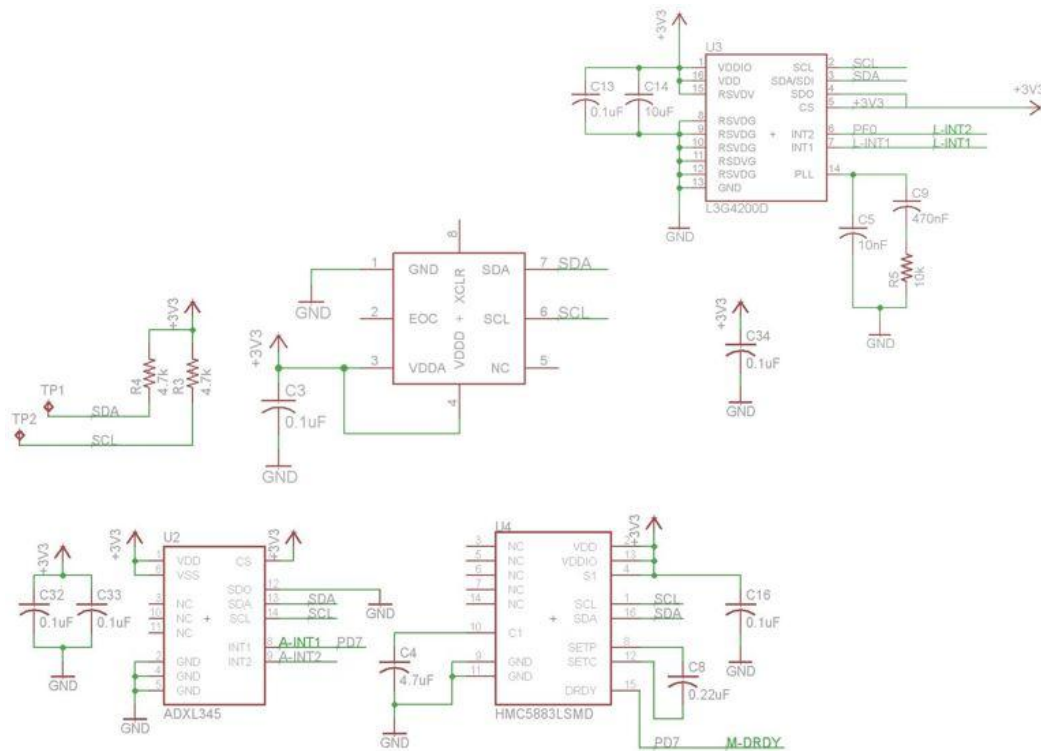


Figure 38: IMU Sensors

The next design to be implemented was the configuration of the sensor stick, short the altimeter. This circuit would be communicating through the use of I2C, see section 6.3, and with this communication configuration the data and clock lines leading to the microcontroller needed pull-up resistors for proper performance. In addition, when dealing with sensors the output and power needed to be stable. For this reason the VDD and any other powering signals were filtered for medium and high frequency noise, in addition to those mentioned in the voltage regulator circuit previously. In this system all of the sensors had different functions and for that reason had different pin-out signals and

configurations. In this system all signals were digital and for this reason all grounds were designated by the same node. For many of these sensors there is a built in standby mode that was not going to be taken advantage of in this project due to the fact that data would be continuously pulled from all of them and the desire was for all sensors to be constantly taking in measurements.

The accelerometer, ADXL345 as seen in lower left of Figure 38, had both VSS and VDD powered by the 3.3 volt signal. This configuration powered the device and put it into standby mode, waiting only for the command to enter measurement mode. By connecting the CS signal to the VDD the sensor was put into I2C mode as desired, but then the seven bit I2C address must be determined. This was accomplished by connecting SDO to ground giving this sensor the address of 0x53 hex preventing it from conflicting with the other sensors.

The magnetometer, HMC5883L as seen in the lower right of Figure 38 had a much different configuration than that of the accelerometer. It had the occurrence of 5 no contact pins that were most likely used in programming the sensor or in manufacturer calibration as well as a data ready or interrupt pin that was connected but no used. As suggested in the datasheet for this sensor, when using a single power supply for the sensor the VDD, VDDIO, and S1 signals were tied together, connected to the 3.3 volt power signal, and connected through a 0.1 micro farad to ground. The C1 value was set to 4.7 micro farads and connection between SETP and SETC was set to 0.22 micro farads in order to handle peak current pulses and low voltage drops seen between the control and the set/reset strap driver in the sensor. SETP was the set/reset strap positive signal to noise ratio capacitor, SETC the driver signal to noise ratio connector and C1 being the reservoir capacitor

The gyroscope, L3G4200D as seen in the upper right of Figure 38, also had a suggested configuration by the datasheet. The 3.3 volt power supply of the system powered the system by connecting to VDD as well as VLOGIC for simplicity and due to the fact that an external clock wasn't desired, and interrupts were unnecessary CLKIN was tied to ground and INT was no contact. Pins 2-7, 14-17, 19, 21, and 22 were no contact as specified within the datasheet as well as RESV-G signal was a reserved signal needed to be connected to ground. As suggest the regulator filter capacitor was set to 0.1 micro farads, the charge pump capacitor to 0.0022 micro farads, and a noise filter capacitance on the VLOGIC signal set to 0.01 micro farads.

The last sensor to be implemented was an altimeter, BMP085 in the middle of Figure 38, which was not associated with the sensor stick, but was used in this system in order to maintain height readings for imaging definition. This connector was the same I2C communication as the other three sensors and therefore had the same pull up resistors along the SCL and SDA lines. This sensor would be powered by the 3.3 voltage line and connected to both VDD and VDDA with the addition of a 0.1 micro farad capacitor to ground. As with the other sensors there was an NC contact that would be no contact along with XCLR which was a master clear signal, no contact allowing conversation, and EOC, which was an end of conversion signal or interrupt, that was going to be no contact as done with the other sensors.

The image shows a detailed PCB layout for a Raspberry Pi 4B board. The layout includes the following components and connections:

- Power Management:** A 3.3V regulator (P10) is connected to the 3.3V pin of the Raspberry Pi 4B module. The regulator is powered by a 5V input (P1) and has a 10k resistor (R10) and a 240pF capacitor (C41) connected to its output. The output of the regulator is connected to the 3.3V pin of the Raspberry Pi 4B module.
- Microcontroller:** A TM4C123GXL microcontroller (U1) is connected to the Raspberry Pi 4B module. The microcontroller is powered by a 3.3V input (P1) and has a 10k resistor (R10) and a 240pF capacitor (C41) connected to its output. The output of the microcontroller is connected to the 3.3V pin of the Raspberry Pi 4B module.
- USB-to-UART Bridge:** A USB-to-UART bridge (P1) is connected to the Raspberry Pi 4B module. The bridge is powered by a 5V input (P1) and has a 10k resistor (R10) and a 240pF capacitor (C41) connected to its output. The output of the bridge is connected to the 3.3V pin of the Raspberry Pi 4B module.
- Passive Components:** Various capacitors (C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, C20, C21, C22, C23, C24, C25, C26, C27, C28, C29, C30, C31, C32, C33, C34, C35, C36, C37, C38, C39, C40, C41, C42, C43, C44, C45, C46, C47, C48, C49, C50, C51, C52, C53, C54, C55, C56, C57, C58, C59, C60, C61, C62, C63, C64, C65, C66, C67, C68, C69, C70, C71, C72, C73, C74, C75, C76, C77, C78, C79, C80, C81, C82, C83, C84, C85, C86, C87, C88, C89, C90, C91, C92, C93, C94, C95, C96, C97, C98, C99, C100) and resistors (R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, R64, R65, R66, R67, R68, R69, R70, R71, R72, R73, R74, R75, R76, R77, R78, R79, R80, R81, R82, R83, R84, R85, R86, R87, R88, R89, R90, R91, R92, R93, R94, R95, R96, R97, R98, R99, R100) are used for various purposes, including decoupling and signal conditioning.

The layout is labeled with various components and their values, and includes a legend for the components.

When doing the design for this circuit there were a lot of resources that could be taken advantage of. The most useful of these resources was the Tiva C™ launchpad that would be implemented during testing of the system. When moving from testing with the implementation of a manufactured board to a designed circuit without the excess capability given on the launchpad it was wise to keep the system as close as possible so as not to cause back tracking in development. However, this program would not be implementing two microcontrollers as done on the launchpad, simply because the system should only have the processing power to perform its mission causing understanding of the datasheet.

80

and GNDX (Pin35). This allowed for all of the powering of the system to be powered by a single voltage source. The TM4C123GH6PM is a 3.3 volt powered microcircuit that is connected at VDDA (Pin 2), VDD (Pin11, 26, 42, 56), and VBAT (Pin37). VDDC (Pin25 and 56) was used for internal logic system and were connected to each other as well as across a capacitance equally 3.4 uF.

Along with most of the ADCs there were more signals and abilities that were not going to be taken advantage of. Due to the fact that the microcontroller would be at work for the entire flight time it was unnecessary to prepare for this quad-copter to go into hibernation mode. For this reason the hibernation function involved with Pin32-37 would be voided. HIB' will be no contact because there is no need to show it was in hibernation when it will never happen. Similarly the hibernation oscillation module input and output (XOSC0 and XOSC1 respectively), which would usually contain a lower frequency crystal other than the system uses under normal operation conditions, was no contact at XOSC1 and XOSC0 will be tied to the hibernation oscillator ground, GNDX. VBAT signal would normally be a lower powered input due to the lightening of processing while hibernating will now be attached to 3.3 volts as the other power signals are and GNDX was connected to the same ground plane as all other GNDs in the system. WAKE' was an active low signal that signified for the microcontroller to come out of hibernation mode. In order to keep the system from ever falling into hibernation mode the WAKE' signal as connected to the ground plane.

The microcontroller didn't have an internal timing mechanism. For that reason an external oscillator crystal was needed to be implemented in order to maintain synchronous timing. The suggested oscillation frequency of the crystal was the 16 MHz. For this reason a 20 pF crystal oscillator is connected to OSC0, the main oscillator crystal input and OSC1, the main oscillator crystal output P40-41 respectively. This was seen in center right of Figure 39.

When powering the microcircuit it was important to filter as much noise as possible on the power source line. For this reason there was a chain of capacitors, seen in Figure 40 that range from 1.0 micro farad to 0.01 micro farads. The larger of the capacitances were used to filter the low frequency noise and protect the source from a slow drop in voltage. The lowest capacitances were used to assist in maintaining a consistent power while experiencing high frequency additive white noise.

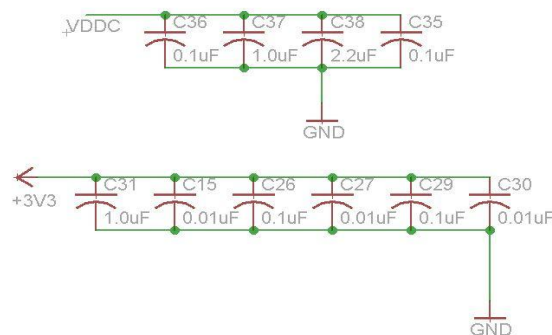


Figure 40: Noise Filtering Capacitors

Another small circuit deemed necessary for the system was a system that could reset the system in two ways, both manually and via JTAG programmer. For the manual switch circuit seen in Figure 41 TARGETRST' was connected to Pin38 of the microcontroller and resets it when the signal goes low. The manual push button was spring loaded in order to consistently keep the signal charged with a capacitor to keep the noise from driving the signal low randomly. The 3.3 volt was connected to a 10 k Ω resistor. This was done in order to keep constant feedback on the JTAG RST' pin.

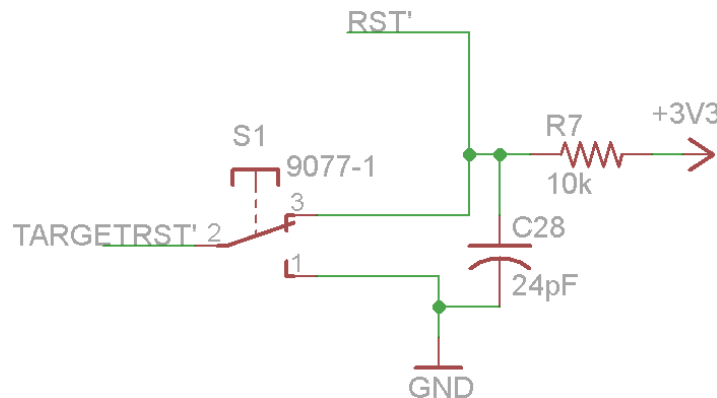


Figure 41: Microcontroller Reset Circuit

The JTAG connector used ten pins to program the microcontroller, seen on the center left of Figure 41. The JTAG test clock (TCK) at J2-3 and the JTAG test mode select (TMS) at J2-7 inputs signals were each connected to the 3.3V net through pull-up resistors, and connect to PC0 and PC1 (Pin52, 51) respectively on the microcontroller. The reference grounds for this connector were located at J2-5,9,8. The test data in signal at J2-6 and the test data out at J2-2 were shorted to PC2 and PC3 (Pin50, 49) respectively on the microcontroller.

The microcontroller received all the data sent by the GPS through UART as seen in figure 42. This connector was not only used for receiving and transmitting data, but also for powering the sensor itself. The reference was linked to the same ground plain in the circuit at J5-1. The GPS's main power was sent as a 5 volt signal via J5-2 and a 3.3 volt signal through J5-5 as a backup power source. The transmit signal TXA coming from the GPS J5-3 was connected to the PA0 net and connected to the U0RX receiving port Pin17 on the microcontroller. Similarly the microcontroller transmitted from Pin 18 the U0TX along the PA1 net to the RXA pin J5-4. J5-6 was a power control signal that could be used to turn off the GPS if desired. Due to the fact that all sensors would be sending and transmitting data throughout the entire flight time this pin was disconnected in order to leave the sensor powered on.

The other UART connection was going to connect to the Raspberry Pi this was a much simpler connection due to the signals needed to be sent are so few, seen in Figure 42. It will need to be powered by this PWB via the output of the 5 volt regulator output through X1-3 and the ground through X1-1. The microcontroller is very useful in that it has an opportunity to transmit through a universal serial bus or a UART signal transmission. In

this case it is better to send simple receive and transmit signals via PD5 and PD4 net, U6TX and U6RX respectively.

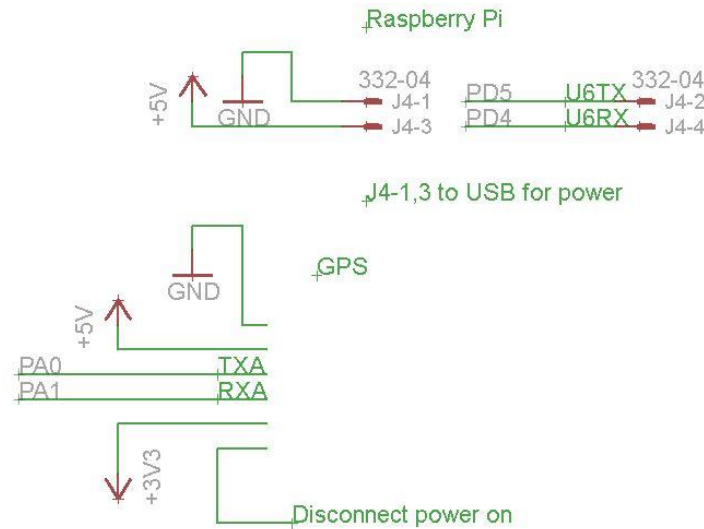


Figure 42: UART Connections

The last signals that were sent from the microcontroller off of the PCB were pulse width modulated signals that were to be sent to the ESCs. Due to the fact the ESCs were pulling power through to the motors directly from the battery, the power signals were only received as a reference 5V for the PCB during flight. There were four ESCs being implemented as there were four sets of three pin output pins. One signal was designated for receiving 5 volts, a second for a ground reference and the third signal was the modulated signal, see top left of Figure 42. The microcontroller had two motion control module with the ability to send 7 PWM signals each. It was decided that two PWM signals would be sent from one module, M0PWM0 and M0PWM1, and two from the second module, M1PWM2 and M1PWM3. These signals correlated to nets PD0, PD1, PA6, and PA7 or pins 1, 4, 23, and 24 on the microcontroller, respectively.

The other PWM signals that were used in this system came from the RC receiver. In order to have the ability to user pilot the quadcopter, the use of a RC controller was implemented and the microcontroller needed to receive the signals via a receiver unit. In order to implement easier programming and testing of the system 6 receiver ports were implemented. Each of these signal sets had three headers. These signals included receiving PWM signal, ground, and a 5 volt power for the receiver itself. These signals were linked to the microcontroller as follows: RPWM1 to PA2, RPWM2 to PA3, RPWM3 to PA4, RPWM4 to PA5, RPWM5 to PA7, and RPWM6 was only used for power with the option of later implementation or jumping of a signal as desired.

Due to this being the entire electrical design section it was necessary to explain a portion of the cable design that was needed to be done for implementation. The first issue was seen as the voltage comes from the battery jacks and went into a splitter. The splitter went from two to eight banana jack connectors for the ESCs. The PCB implemented header pins to the cell monitoring connector on the LiPo battery in order to receive the

voltage reference, not needing the large amount of current that would be going to the ESCs. In actual flight mode the pcb connection straight to the battery will have to sole purpose of monitoring the battery voltage level. Then came the connection from the PCB to the Raspberry Pi. This was done by simple header connections. These contained receive, transmit, ground, and 5V power that plugged straight into the GPIO of the RPi. The plugging into the RPi, however, needed a lot of care due to the fact that plugging the 5V power into any of the other GPIO caused the card to burn out, which was a problem that arose during testing.

For the testing and development portion of the project it was decided that being able to pull power from either the battery itself, short the ESCs would be useful so as not to burn out the speed controllers. In order to accomplish this feat a jumper system was implemented as can be seen in the top left Figure 42 that either jumped the output of the on board 5 volt regulator to the 5V line or, for flight, jumped the ESC 5 volt power to the system 5V line.

At this stage in design the schematic would take into account all of the circuitry necessary for a proper working system while stripping unnecessary components and circuitry. After testing and development was completed, the implementation of this altered circuit posed little issue.

5.7. Ground Station Overview

The main purpose of this project was to have a map generated from an aerial view. The ground station was the final step to creating this map. As KIRC and SPOC fly within the specified bounds of the designated area to observe, they both take pictures facing downward in reference to the quadcopter and store images on the SD card in the Raspberry Pi TM. The ground station is a laptop with an 802.11 card to communicate to the Raspberry Pi TM with both quadcopters on the ground and during flight. This is also where the quadcopter lifts off and lands. During the flight, each quadcopter sends information back to the ground station about its status. The laptop has a custom interface used to track everything that the quadcopters are doing. When the mission is complete, the SD cards are removed from each Raspberry Pi TM and inserted into the laptop which then processes the picture data into a single image. This is done using open source software. The team can also send all of the pictures over the DTN at once by hitting a button on the ground station GUI.

Missions always start at the ground station with the laptop. Once the quadcopters are booted up they connect via Wi-Fi with DTN running on top to the laptop. Next they fly to the area that is being mapped. During the mission each quadcopter sends its status back to the laptop every second. The information is stored on a text file in the Raspberry Pi TM that is sent DTN through the Peer-to-Peer network between the quadcopters and the ground station. This includes its current GPS coordinates, altitude, battery voltage, signal strength, and the state the quadcopter is in. The packets received, and packet number will also be displayed from the ground station. The interface only displays the latest packet. A screen shot of the interface is shown in Figure 43. The file drop down menu provides another option to exit the program. The edit drop down menu allows a team member to

select the computer he is using. This switches between a Windows machine, Linux on the same machine, and Linux on a netbook. The DTN parameters are different for each computer so this allows for one executable of the program to be used for all of the team member's computers that were used for the mission and testing.

The state that the quadcopter is in is the state machine used for the Tiva C TM microcontroller. This is also the interface that is used to start the mission. The user enters the four coordinates of the total area that quadcopters have to map. The positions entered will display on the image through an API call to Google MapsTM. As seen in Figure 43 they are shown as markers in the image of the field being mapped. This is useful for estimating what size areas at certain distances can be mapped. When the enter key is pressed or the Update Map button, the marker moves to the corresponding location. The actual location of KIRC and SPOK are shown on the map with markers "K" and "S" respectively. These are updated on the map every second with each packet automatically so the Update Map button does not have to be used. The Map Center text field is used to enter the location to view on the map. This is adjusted by using the four arrow keys on the right of the display. Zoom In and Zoom Out allow the user to adjust the view of the map. When the team is ready to start a mission, they press the Log Time button which takes the time and date from the laptop and stores it. KIRC and SPOK Take Picture buttons allow the user to take a picture on demand. When the mission is over the team clicks the Save Log button which puts the time that was logged on the top of a text file and inserts all of the packets received into one file. This also removes all of the individual files from the folder. The Send Images button sends all of the pictures taken to the ground station. This is optional since the team can also get them from the SD card in the Raspberry Pi TM. The Clear Pi's button deletes all of the status files and pictures stored on the SD card remotely.

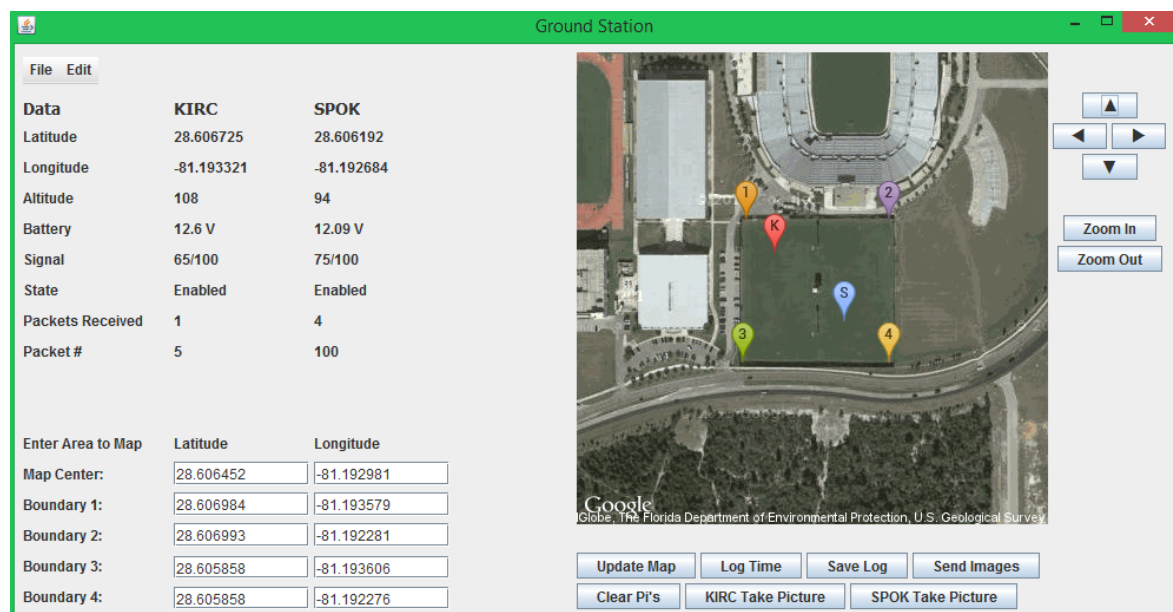


Figure 43: Ground Station User Interface

When the quadcopters have landed after successfully completing their mission, the images are removed from the SD card and placed onto the laptop's hard drive. The pictures are stitched together using open source software called Hugin. This creates a map with the pictures in a mosaic form since the quadcopters are translating in reference to the ground. Most image stitching software is made for panoramic photography so it requires each image to be set as its own lens in order for Hugin to properly stitch the images in a 2-dimentional fashion. The user enters advance or expert mode. From there the user right clicks each image and select new lens. This is shown in Figure 44. This is time consuming since each picture is individually clicked on and set as a new lens. The projection is set to rectilinear and positioning is done in mosaic mode. Each image is optimized separately since it was taken independently from the other images. The final image is stitched together and saved as a JPEG image file type.

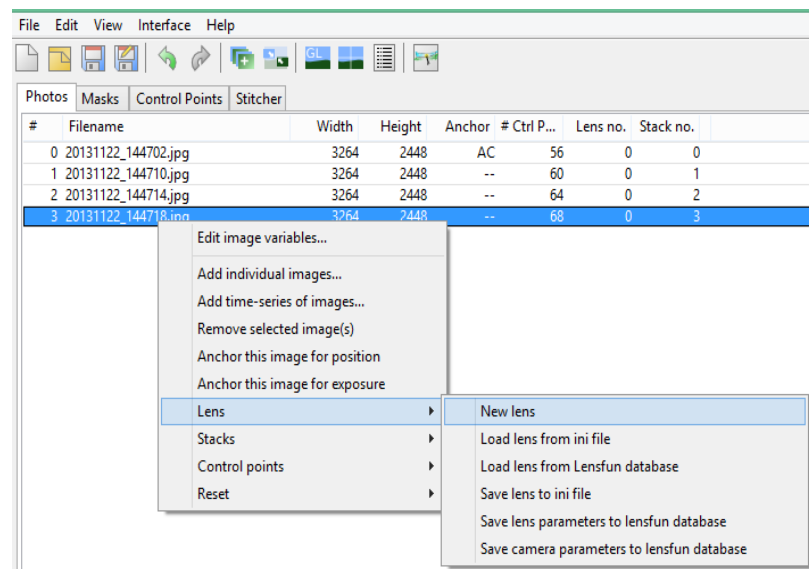


Figure 44: New Lens Setting

Figure 45 shows six individual pictures taken of a fence in a backyard. Each picture was taken at a different location on the ground. The angles are also different intentionally which was not the case with the quadcopter but it was done in this example to show that Hugin can adjust the pictures accordingly, regardless of the different angles. Also this object is much closer than the objects that were taken during the mission. The overlap of these pictures is noticeable if you look for distinctive objects. Hugin looks for distinctive patterns in the picture that match.

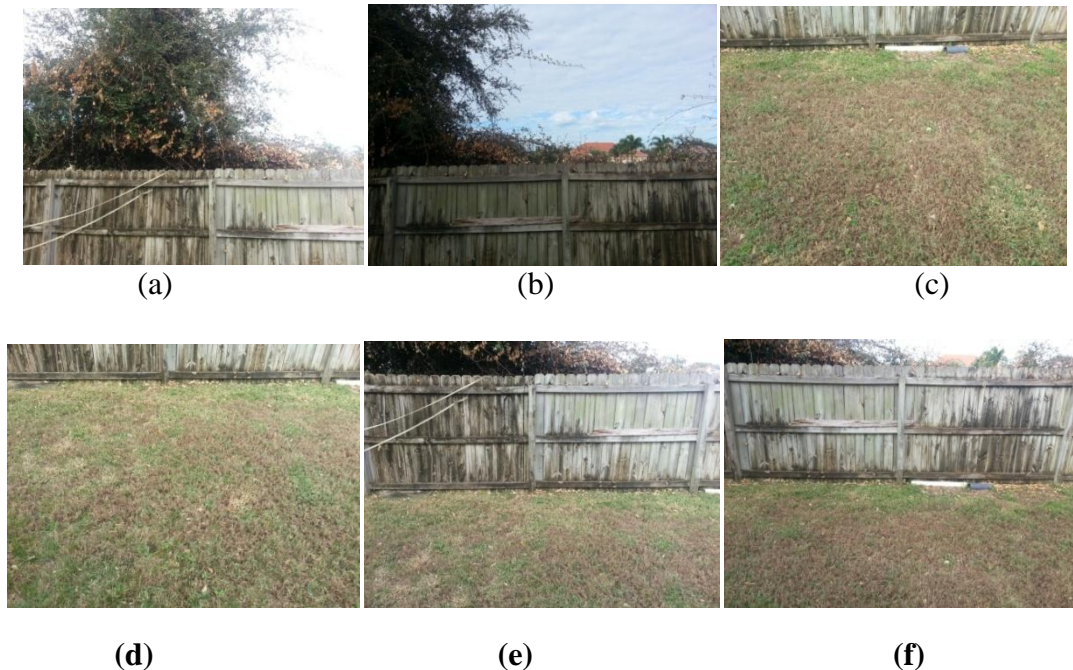


Figure 45: Individual Pictures (Stitching Software Example)

It is essential to have the pictures overlap. Figure 46 shows what image stitching would ideally be on the left. This is the most efficient way to cover the largest area possible. Theoretically this is possible if the pictures are all taken in order and there is no under or overlap. Since this is not possible with the quadcopter at 100 plus feet, the images will have to be overlapped. The stitching software makes control points which represent the points on picture A that correspond to the same points on picture B. The overlapping of the pictures can vary significantly based on focus, angle, and lighting. Since the program uses an algorithm to try and find matching points, this means the more overlap that exists between two pictures, the more points can be found. For this project the team uses around a 50% overlap on every picture in every direction like shown on the right side of figure 46. This reduces the amount of theoretical area that the quadcopters map, but the quality is much better than pictures with gaps and misalignments. Since the software is just using overlapping as the only method to determine where each picture belongs, it is not necessary to keep track of the order that the pictures placed or taken. The downside of this is the larger the area is, the more pictures will need to be taken. This is something that not only affects the total flight time but the stitching time as well. Each picture that is added exponentially increases the amount of time for the program to place each picture. This is because Hugin has to compare each picture with every other picture to look for control points.

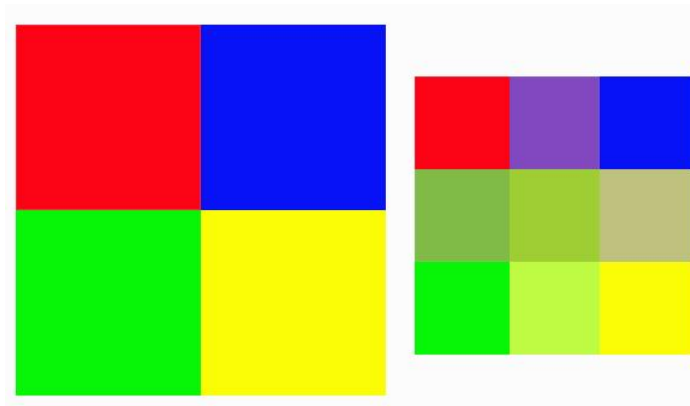
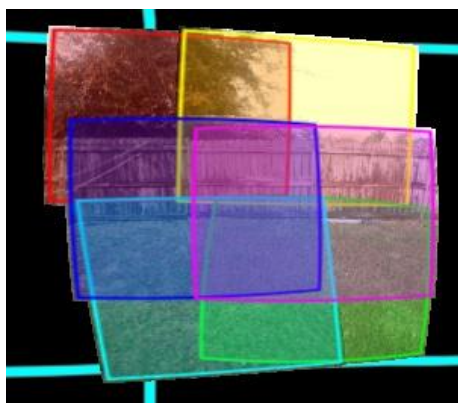


Figure 46: Example of Capturing an Area (Overlap)

Figure 47 is an image preview in Hugin that shows all of the individual pictures together with the overlaps. Each picture is approximately overlapping the neighboring pictures by 50% just like the example in Figure 47 shows the final render of all the pictures stitched together. It is not entirely perfect since there are so many overlaps, the program has to pick which picture takes priority in each overlap that matches the surrounding overlaps the best. Overall it produces a decent uniform picture.



(a) Placement



(b) Final

Figure 47: Pictures Stitched Together

The ground station is where the mission starts and ends. It is a convenient way to start a mission since no wires are needed to hook up. The intuitive interface allowed the team to track the entire progress of the mission while it was happening. When the mission was over, the laptop put all of the images together on the spot so everything needed to do a

mission (or even multiple missions) was present. The team also had the option to stitch all of the images together later and just get the photographs while out on the field. The controller can optionally be located at the ground station during testing. It only comes with one receiver so only one quadcopter can be overridden and manually controlled during the mission. This is not essential to the functionality of the ground station or project.

5.8. Physical Design & Layout

The physical design and layout of each quadcopter consisted of integrating the mechanical components such as the frame, the propellers, and the motors with the electrical components. This section will discuss the geometry of the parts as well as how they were physically pieced together. The descriptions will be supplemented with pictures of models created using the Solidworks™ modeling program. A brief narrative of the modeling process provides an initial platform to develop the physical design and layout. Each of the components in the design was modeled individually on Solidworks™. The team decided to put forth great determination in creating a model that was completely to scale. By modeling each component individually and combining all of the elements into an assembly, a visual representation of the prototype would assure that all of the parts chosen would fit together in a reasonable manner. If there were any sizing issues in the model, for example if the battery was too large for a chosen frame, the team could take proper actions accordingly by either choosing a battery of different dimensions or choosing a different frame with a larger clearance. Another option would be to rearrange the components in the assembly to fit a desired part. After assembling all of the components and piecing them together in a functional Solidworks™ model, the team made the decision to keep all of the components chosen in the design phase. Figure 48 below shows a diametric view of the assembled Solidworks™ model. The orientations of the components in the assembly were chosen by the group in an effort to maximize performance.



Figure 48: Solidworks Model of Full Setup

The meticulous details of the assembly were difficult to see in the zoomed out picture seen in figure 48 above therefore more pictures were provided to reveal the assembly in further detail. Figure 49 below on the next page shows one of the wings and the parts attached to said wing. Each of the 4 wings were nearly symmetric with the exception of the color. Two adjacent wings were colored red, and the remaining two adjacent wings were colored white. The purpose of this color difference made the orientation of the quadcopter more obvious and was especially advantageous when the quadcopter was flying at a high altitude. As one can clearly see in Figure 49, each wing had a motor and propeller pair mounted at the center of the circular section toward the outer end of the top face of the wing. Also on the top face of the wing, toward the inner end, resided a purple electronic speed controller. One may have noticed three small cylinders colored black, yellow, and red facing radially outward toward the motor. These three cylinders represented the starting points of the three wires feeding power to the motor. There were also three cylinders facing radially inward on the right hand side of the ESC in this figure. The two cylinders colored red and black, located on the top and the bottom in this figure, corresponded to the wires going to the battery. The middle cylinder with a larger diameter was a model of the polar capacitor on the ESC. Although the three wires facing radially outward appeared to be cut off after a short distance, it should be made clear that these wires in fact extended the entire length of the wing and connected to the three wires coming from the motor which were not shown. Similarly, the black and the red wires facing radially inward would not be cut off, but in practice the wires would run to the positive and negative terminals of the battery. It was common practice in mechanical engineering to leave out the wires when modeling. The team decided it impractical to model wires going to and from every element. This task would prove especially difficult for wires going from each ESC to the battery in which each asymmetrically swoop from the top plane of the frame to one of the top edges of the battery.

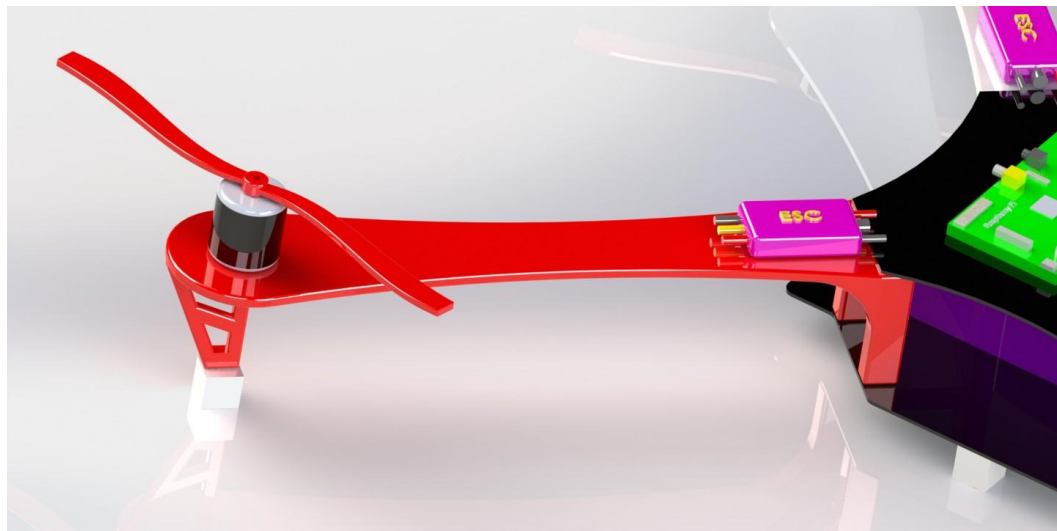


Figure 49: Solidworks Wing Model

The battery was difficult to see in the first two figures, but its position became more obvious after viewing the quadcopter from a different angle. Besides the four wings, the

frame consisted of two parallel plates, a top plate and a bottom plate. These plates make up the center portion of the quadcopter where the avionics computer, sensors, and the navigational computer would be mounted. The frame came with 24 identical screws. Each wing will use 4 screws to mount to the top plate and two screws to mount to the bottom plate. By inspecting Figure 50 shown on the below, one could see the front face of the battery. In the figure, said face is the green with purple letters in between the vertical supports of the wings. The battery is to be mounted to the bottom plate of the frame. The group justified positioning the battery on the bottom plane in order to maintain a lower center of gravity. This heavy base design, as opposed to a more top heavy design, had a much lower probability of flipping over during flight or during emergency landings. As one could see from the figure, there was little separation from the top of the battery to the bottom of the top frame, nonetheless, any separation would aid in keeping heat dissipating off the battery and away from the electrical components on the top frame. The underside of the bottom plate had a white box connected to each of wing supports.

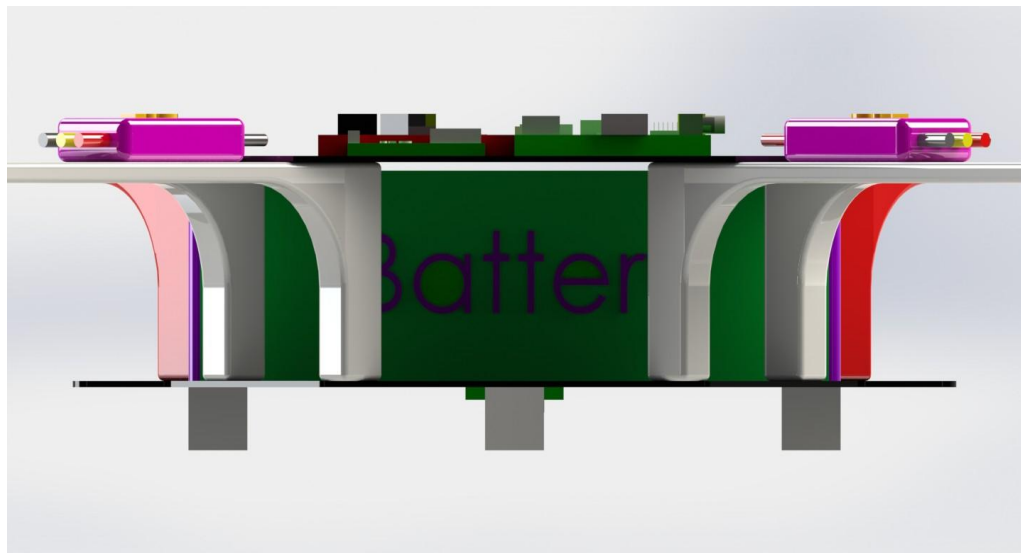


Figure 50: Solidworks Front View

Figure 51 shown below is a zoomed in top view of the prototype version of the quadcopter. It was worthwhile to mention the entire Solidworks TM model was for the phase one portion of the team's project. Remember, phase one corresponds to our design before implementing the PCB, using the launchpad at the system of stabilized flight control. For this reason, one could see the IMU and the GPS separated from the microcontroller. Again, it was obvious that the connectors between components were not shown on the model. In actuality, there were a plethora of wires from the microcontroller to the sensors, ESCs, and Raspberry Pi TM.

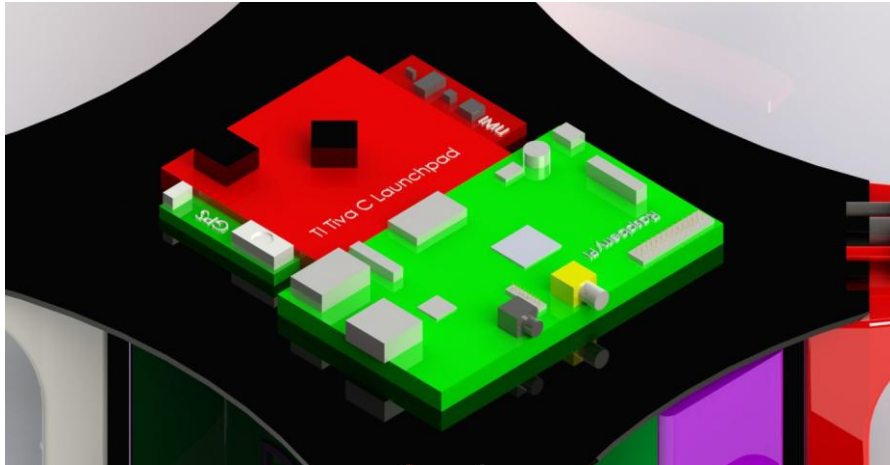


Figure 51: Solidworks Top Panel

Also not included in figure 51 is the ribbon cable from one rectangular prism on the Raspberry Pi TM stretching downward around both the top and bottom plate of the frame to the camera mounted to the underside of the bottom frame. This camera would be pointing directly downward and served as the camera used to take aerial photographs during the surveillance missions. Figure 52, shown below, is an underside view of the quadcopter. The green rectangular prism represented the board on which the camera was mounted and the black cylinder was the camera itself. The four white boxes mounted to the underside of the bottom plate of the frame were made of styro-foam. The team realized since the bottom plate of the quadcopter was almost level with the ground the camera's lens was actually protruding further than the legs on each wing. The addition of styrofoam buffered to the bottom plate of the frame as well as on each support leg kept the bottom of the quadcopter frame elevated off the ground enough to protect the camera from getting damaged by the ground.

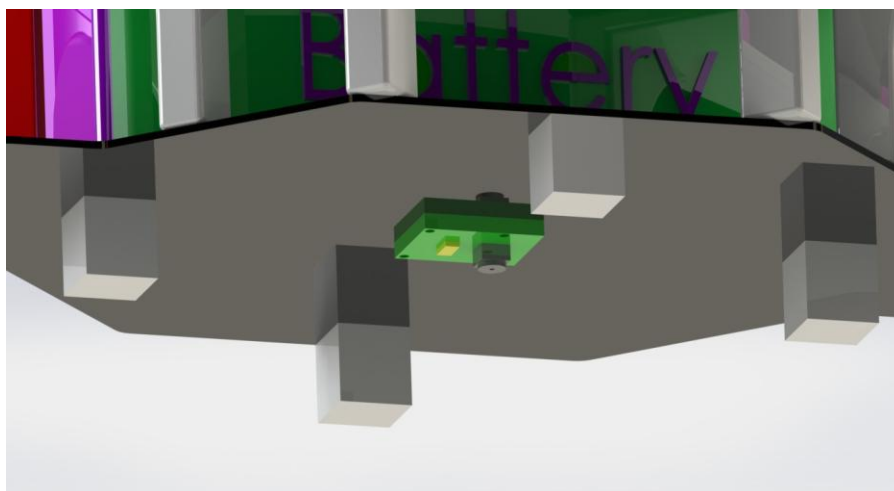


Figure 52: Solidworks Camera Model

To get a general idea of the dimensions of our quadcopter, figure 53 below shows a side view of the frame from wing to wing. The width of the quadcopter was 450 millimeters in length or approximately a foot and 5.7 inches. The height including the buffers, the frame, the motors, and the propellers was 90 millimeters or a little over 3.5 inches. This was a relatively large quadcopter. The large quadcopter design was beneficial for stability aiding in remaining steady when the camera took pictures at high altitudes.

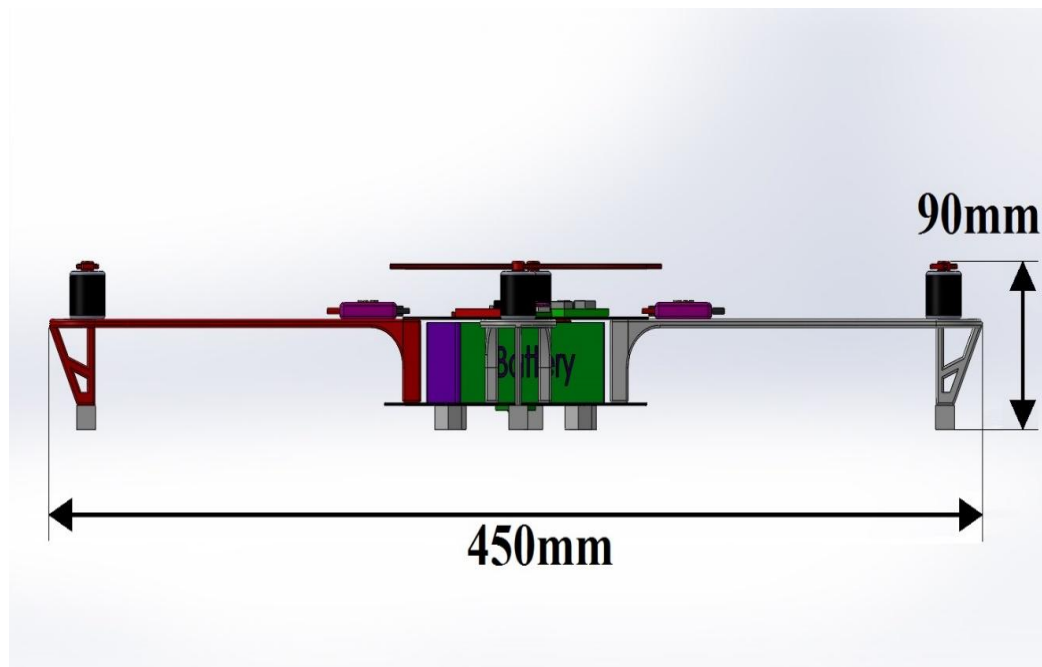


Figure 53: Quadcopter Dimensions

6. Prototype Construction & Coding

The following sections will proceed to explain part acquisition, PCB design and assembly, part integration, and a brief coding overview. Part acquisition includes any distributors and resources that will be used in order to gain access to needed passive and active components. In part integration the off board interfacing is discussed, for PCB communication see the PCB design section in the design analysis chapter and assembly will describe the layout, size of board and optional manufacturers to fabricate the board. Lastly, the code overview describes the operating systems and compilers to be used as well as a description of the software to be tested.

6.1. Parts Acquisition

When keeping a project to a low budget, not only was part selection an important factor, the distributor became a resource to compare. This project was unofficially sponsored by the National Aeronautics and Space Administration, NASA, and for that reason this organization would be supplying all funds for part acquisition with few exceptions. An advantage of being involved with such a prestigious organization was the ability to have many passive and active components on hand for use, as well as the ability for board development, if necessary, was a capability handled internally by NASA.

The first part chosen was the microcontroller, a TM4C123GH6PM, produced by Texas Instruments. Texas Instruments Incorporated, TI, was a useful resource for many aspects of the design phase of prototyping the project. They provided resources including Launchpad and expansion options, a large technical support team, and large level of production.

One of the benefits that was taken advantage of was the opportunity of having a Launchpad available for immediate implementation of testing and development. This resource was valuable in that it provided an excess of ability that would not be needed in the final prototype, but was of great use in the time between design and PCB fabrication. This ability to continue forward in development instead of having down time while components were being placed and soldered was highly advantageous for a faster schedule with more time in the event of unforeseen issues arising.

Secondly, TI was known for maintaining a very capable engineering staff filling the technical support department. In the event of misunderstanding or confusion in the use of a product TI developed, this department was available for extended office hours with a more in depth knowledge of TI's products than could be simply read from a user manual. In addition to the assistance in the implementation of products, these engineers were also an available resource for discussing troubleshooting options that may have been overlooked.

Due to the fact that Texas Instruments is such a largely used company they consistently produce a large amount of each popular product. In the case of the TM4C123GH6PM microcontroller, the large quantity of supply allowed for lower cost to the consumer for both the microcontroller and the Tiva C Launchpad. This allowed not only for low cost

reproduction of circuit design, but also a hasty part replacement in the event of one being damaged.

In the case of many of the mechanical components being implemented in this design, Hobby King was one of the best resources available. Hobby King has a specialized inventory of parts solely for the development of multi-rotor copters. This offered the opportunity to compare prospective parts quickly and efficiently in terms of price, and specification. In addition there was also some additional suggestions for easily compatible parts that they also provided. The frame, motors, electronic speed controllers, propellers, lithium polymer battery, and controller were all procured from Hobby King.

One of the many benefits of Hobby King was the interface of the website. The ease of navigation and the quick representation of important data with the option of additional information quickly narrowed the list of capable parts. With the benefit of the initial search supplying not only an image, name, and short description, it also rendered a price. This allowed for a budget, which is of great importance to this design as any engineering design for that matter, to be maintained without a waste of time searching into overpriced piece parts.

Due to the fact that this design included a lot of opportunity for mechanical design components it was a great resource to have the opportunity to use a supply such as Hobby King that made these specifications easy and straightforward for understanding. Upon selection of a part the buyer was immediately met with multiple images on different angles of the part, a brief description and an organized, summary table of part specifications including, but not limited to, dimensions, and weight. When choosing the motors for this design, there was no necessity of undergoing a study of dynamics in order to decide on propeller size; Hobby King provided current pull associated with different propeller lengths. In addition, due to the fact that this was equipment specifically used for development of aeronautical systems, the specifications on each part were isolated to only the data needed for implementation in such a system.

Sparkfun was another useful resource in the acquisition of electronic components and testing cards. Both the four sensors used (gyroscope, altimeter, digital compass, and accelerometer) and the testing sensor stick was found on Sparkfun. When it came to design, Sparkfun had many beneficial additions that simply ordering from some large part distributor would not have provided.

When inquiring into different parts, Sparkfun had the additional benefit of giving a brief bit of background and critical choosing points. When looking into different altimeters a description of the interfacing abilities were given. In order to avoid wasting time on a component that only used USB communication instead of the desired I2C, making reading through pages of datasheet while verifying other specification needs, Sparkfun immediately stated that this altimeter interfaced with I2C as well as USB.

The sensor stick that would be implemented for initial testing and board development was to be acquired from Sparkfun. In addition to providing the datasheets for all of the sensors that were implemented on the breakout board, Sparkfun also provided a full schematic of the card. As the project moved forward with the design and development of

the PCB, having a basis for schematic design for most of the sensors fast tracked the process.

If supplying the schematic weren't enough, Sparkfun also supplied an Eagle file, Eagle being popular PCB design software, of the breakout board itself. By supplying this file to the customer Sparkfun allowed for an easy integration of parts not already included in the library. If such a file were not provided and the parts used were not provided in the Eagle library, i.e. the sensor chips, the designer was forced to go through a painstaking process of adding the part to the library, and in the case of the school provided libraries, not additions to the library were saved overnight.

Another useful distributor of components such as the Launchpad was Amazon.com. Not only did this website have a large quantity of components readily available, they also had a quick shipping time. It could take as little as overnight to 3-5 business days. This 3-5 business days shipping time was free of shipping and handling charges for students. This was extremely useful as this project moved forward into testing and implementation.

During testing and troubleshooting it was common that a couple of small components or piece parts could be damaged here and there. With this in mind, ordering and shipping small and cheap parts became expensive relative to the budget when the cost of shipping is more than the cost of the actual component.

Another benefit in the use of Amazon.com was their return policy. When using components that were sensitive to electrostatic discharge packaging and handling were damaging when not done properly. Before purchasing a product on Amazon the return policy of the seller was readily available, saving the buyer from misrepresentation by a seller.

At the University of Central Florida, senior design projects were done completely by the students, but the university had readily available piece parts on hand for students to do testing with. They provided many piece parts in order to prevent the students from needing to purchase small quantities while readily available in bulk to the university. This was very useful when beginning implementation and testing of the design because the students would be able to use lab equipment for development before final design of the PCB was confirmed and fabricated. Mr. David Douglas is readily available for students to present their needs that he has in his stockroom at all times. Another benefit was the renting of soldering irons without which the group would have been at a standstill at times.

6.2. PCB Design and Assembly

The printed circuit board, PCB, was the last stage in the development of design. This was because the circuit needed to be proven to work properly before packaged for final prototype. In order to avoid wasting funding and ruining the budget the circuit needed to be tested in as many small parts as possible. For this project beginning with using the launchpad, GPS, IMU sensor stick, Raspberry Pi TM, Li-Po battery, and small regulation circuit. Stepping away from the manufactured ICs, save the Raspberry Pi TM, and toward the completed design on a breadboard as the last step before sending the design for

fabrication. In Figure 54 is the PCB as seen in EAGLE 6.2.0 designed up through Senior Design 2. Additionally, Figure 54 shows the physical, developed PCB used in the final design for presentation. Note, however, that this design changed over time doing to problems found in the first version of the PCB.

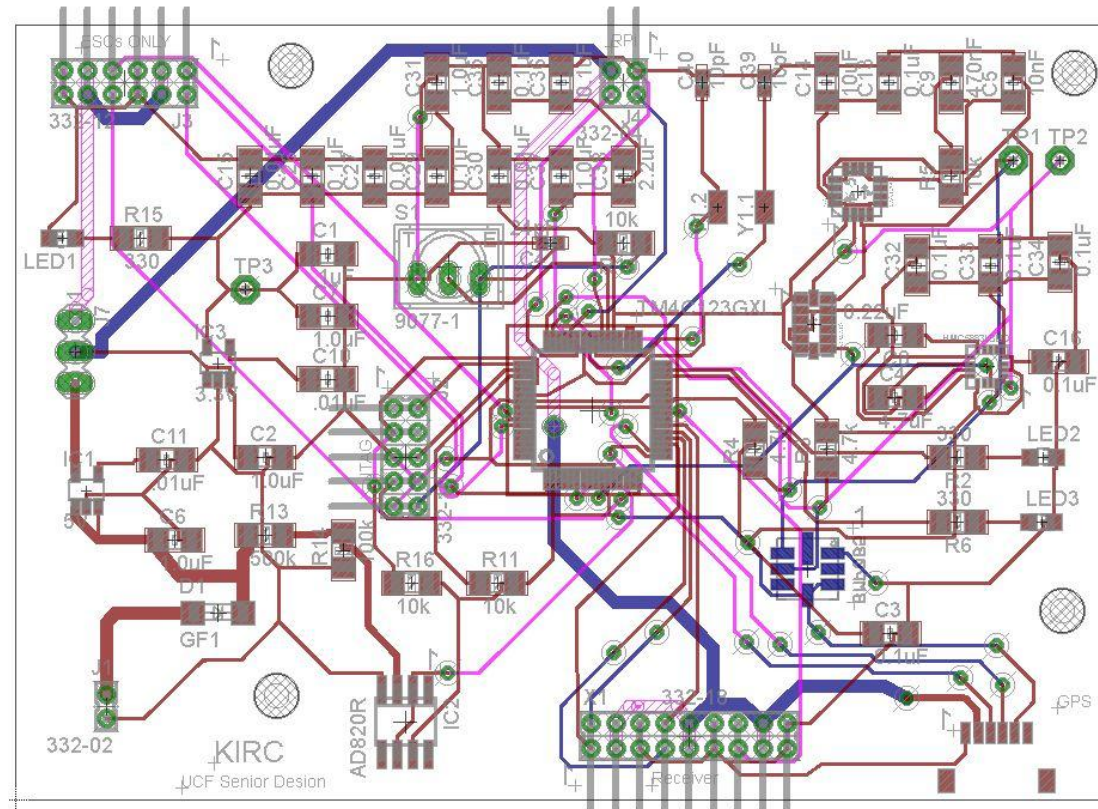


Figure 54: Printed Circuit Board

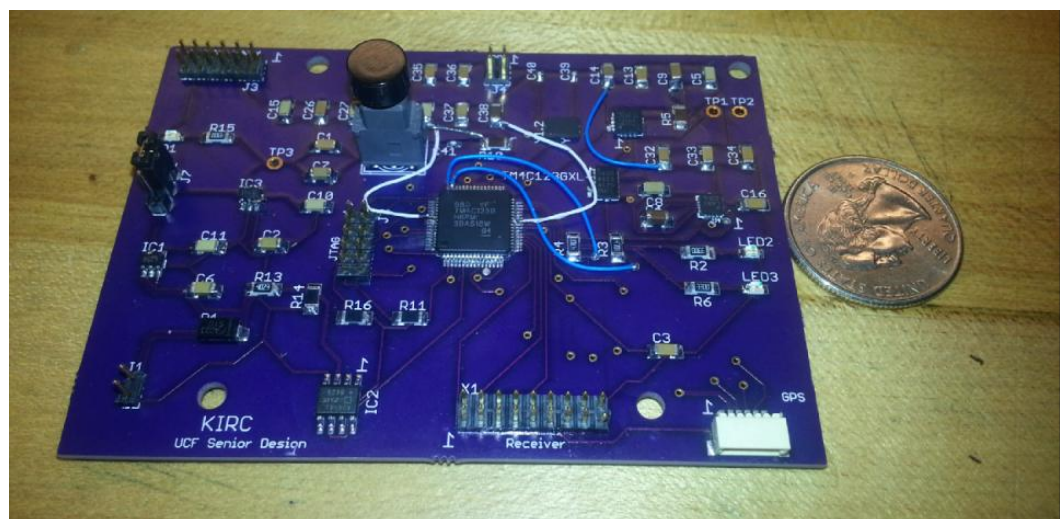


Figure 55: Physical Printed Circuit Board

On this PCB design, as with most, all parts were organized in sections and oriented for an optimal use of space. The organization strategy was based on purpose, location of implementation, and amount of power. This organizational strategy assisted in keeping the trace lines more logical and would require less layers for the PCB assembly.

First in the organization was the location of the nets connected as compared to the component use. There was no sense in running fifteen volts around circuitry that wasn't powered by it in order to cram as many components as possible onto the smallest board possible. For this reason as the components moved farther from the power connector, in the bottom left as seen in Table 7, the components were powered by lower voltages and manage less current. For this reason the voltage regulator circuit and the battery monitoring components, operational amplifier and higher power resistors, took the lower left corner board and near as possible to their terminating locations. The 5 volt regulator that was only implemented in UART connections was organized close near to the connectors the signals were transmitted from. That same logic placed the 3.3 volt regulator close to the 5V input of the board due to the fact most components were powered by its output

The net oriented organization was taken into consideration when placing the passive components. Due to the size of these components being so small and so popular it could be seen how the crossing of so many of these lines would cause unnecessary need for additional layers. For this reason the IMU sensors were set with those capacitors used for noise compensation and pull-up resistors used. The IMU sensors were also kept together due to the fact that by being implemented on the same sensor stick these sensors do not affect the performance of one another and they will all be using the same communication lines. Additionally the accelerometer as well as the gyroscope need to be placed as close to center of the copter for best yaw and pitch/roll stabilization. The TM4C123GXL microcontroller was placed as close as possible to the center of the board with the orientation chosen to minimize cross lines and those traveling under the controller to reach the necessary pin, i.e. the clock oscillator, IMU sensors and connectors.

As for the purpose coordinating to the location, this mostly referred to connectors, with a few sets of other components. When implementing this PCB in the final, full system the connectors were as close as possible to the location of termination to prevent voltage drop. It should also be clear that signals crossing over the board for any reason could have had effects on sensor data as well as desired outcomes in circuitry. The PWM signals were going out to the ESCs therefore needed to be on the outskirts of the board and the UART connectors for the Raspberry Pi, hence location of J2 and X1. However the GPS was on the opposite side of the board as the Pi, hence the location of J5.

When it came to the fabrication of the board itself there were a couple of options presented. The most logical option was to have NASA develop the PCB internally. The second option was to outsource the board to a low price fabrication company known as OSH Park. Both of these were viable options that would have the ability to produce a product up to expectations that would work as designed, in addition to the fact that both had great incentives to issue them this business.

OSH Park is and has been a popular PCB fabrication center for UCF senior design students for many years. This is mostly due to the fact that they implement fabrication of three boards, allowing for option of ruining boards and still having extra, at a lower price. In many of these senior design projects a large amount of layers in the PCB were not necessary, including the design for the PCB implemented on this quad-copter and for this reason OSH Park offered a four layered board for \$10 a square inch and a two layered board for \$5 per square inch. With space on the racks of the frame being small in size, a four layered board will be used.

Another incentive for the use of OSH Park for fabrication was the haste with which they completed the board. For a 2 layer board, as basic as possible, there was a maximum of 9 day turnaround time and for the 4 layer it was slightly greater than two weeks, depending on the quantity of work placed before them. In a situation where there will be test equipment available for continuation of testing while the PCB was being made and populated, the lead time wasn't a great issue, however any unnecessary time that wasn't taken advantage of to verify performance of the PCB was a hindrance on the program.

Doing a project for a company like NASA allowed for many benefits because of the strength of resources they could provide, both personnel and material. Due to the fact that NASA was mostly sponsoring this program the price wasn't really a large issue. When taking into account the cost of fabrication for this board would be materials and the amount of time that it takes to fabricate, hindering the development of other internal projects. For this reason timing was an obstacle.

In order to avoid hindering other program development the PCB needed to be ready for fabrication early and on a moment's notice. If it so happened that the schedule allowed for a break long enough for fabrication it would be critical to get the board developed before anything could alter it. This presented another issue, however. If this plan of action was implemented and the PCB design was yet to be fully tested with quality mission success it could turn out to be a waste of time to make the PCB. With this in mind there was also the option that the developed design would have no issues, or at most a couple vias could be implemented as a fix.

In addition to the waiting on availability of the fabrication center, there was an additional lead time on the build. Due to the fact that NASA commonly does very complicated boards compared to those involved in this program the average turnaround was five to six weeks. This was not considering that with this project being a student design NASA would be more accommodating to deadlines.

After comparison of the two options for PCB fabrication centers, as the project moved forward OSH Park was the chosen manufacturer. This is in the case that NASA was unavailable to develop the PCB when prototyping has reached that stage. If NASA had the availability to produce the board it would be unlikely that they would fund OSH Park doing the work.

After the manufacturing of the PCB it was found that certain aspects of the design were incomplete and needed to be routed using white wires. These were issues that occurred due to net name labeling mistakes as well as misunderstandings of the powering of the

microcontroller. After these simpler jumper wires were implemented there were no longer any issues seen in the implementation of the PCB into the final design.

6.3. Parts Integration

Through the use of this microcontroller gave an opportunity to control as many components as could be possibly need for system control and performance. In any design the part integration is a big decision based on communication speeds and more often capability. The Tiva TM4C123G-H6PM microcontroller was capable of implementing Universal Asynchronous Receiver/Transmitter (UART), Synchronous Serial Communication (SSI), Inter-Integrated Circuit (I2C), and Joint Test Action Group (JTAG) communication with the pins specified in Table 7 below.

GPIO Pin	Default State	GPIOAFSEL Bit	GPIOCTL PMCx Bit Field
PA[1:0]	UART0	0	0x1
PA[5:2]	SSI0	0	0x1
PB[3:2]	I2C0	0	0x1
PC[3:0]	JTAG/SWD	1	0x3

Table 7: Microcontroller Communication GPIO

JTAG, a common name for the IEEE 1149.1 standard test access port and boundary-scan architecture, would be the source of communication for programming the microcontroller. JTAG was a fairly common choice for programming and communicating with specialized integrated circuits due to the fact that it was original implemented in boundary scanning for testing and troubleshooting PCBs. With the Tiva microcontroller and the Launchpad that was being used for initial testing already JTAG integrated, it assisted in an ease of integration as the project moved forward. In the final programming implementation the Launchpad was actually used as the JTAG programmer of the PCB microcontroller with no unforeseen difficulty.

The GPS communicated with the Tiva microcontroller through the implementation of UART serial transmission. Due to the speed and desire for precision in the data received from the GPS unit this was a useful tool. With the GPS being a unit that was to be purchased from a third party and not running off the same oscillators as the microcontroller the ability to configure transmission speed was necessary. Also, the use of a real time operating system (RTOS), which was based on a series of priorities, made the ability to only allow data passage via a ready to receive bit prevented the loss of useful or skewed data.

UART was also implemented in communication between the Tiva Microcontroller and the Raspberry Pi model B. UART is a form of serial communication to be implemented by way of the RS-232 standard commonly implemented by computers and integrated

circuits to peripherals devices. The UART was a very useful tool due to the fact that data formats and transmission speeds were configurable. The Raspberry Pi TM contained a powerful processor and was in control of designating to the microcontroller the location of the quad-copter's next way point via a UART serial port.

The Raspberry Pi TM would additionally be communicating with the ground station that was used for the alignment and meshing of the grid images sent from the Quad-copter. Due to the fact that the mapping was not of great importance while in flight and was only represented as a whole, the Raspberry Pi TM was communicating with the assistance of a WiFi card. While in range all images were sent back to the ground station for image processing, and when it should happen that the aerial vehicles fell out of this range the Raspberry Pi TM would store the image internally until a strong enough connection that could ensure no data loss occurred. The implementation of Wi-Fi was the best option for this means of communication, not only because it was easily implemented on the Raspberry Pi TM, but also because it had the ability to quickly send large files, allow for higher quality imaging.

The Raspberry Pi TM would be taking these images using a specialized 5 megapixel camera that was made specifically for implementation with the Raspberry Pi TM itself. Communication and data transfer between the camera PCB and Raspberry Pi TM was implemented by way of a 15 way ribbon cable. This communication was done by way of camera serial interface, CSI-3, which allowed for the higher resolution imaging.

I2C communication, commonly referred to as I-squared-C, was critical in the communication with all sensors, save the GPS. I2C has commonly been implemented in communication when managing low speed peripherals. The fact that this interface is based on a master-slave coordination between an open I/O address on a microcontroller to a peripheral device, and add in that the Tiva C TM microcontroller has designated I/O that is unused elsewhere made I2C interfacing the most promising means of communication as this project moved forward.

The ESCs controlled the motor speeds and were controlled by pulse width modulated signals (PWM). Another great benefit of using the Tiva TM4C123G-H6PM microcontroller was the built in PWM ability. This made the ESCs easily controlled by surface mounted short form pins that were straight memory mapped I/O from the microcontroller.

The chosen motors were three phase brush motors that were controlled by the ESCs. The ESCs produce an analog signal in each of the three lines leading to the motor. The motors took this signal and based on how out of phase these signals were, produce a certain amount of thrust.

The power system was fairly simple for this project due to the fact that it was digital, most components were low voltage and controlled from the PCB. The power was distributed from the battery to the ESCs, followed by the motors, using simple banana jacks. Though this was simple and basic way to distribute power it was effective. The PCB took in 5 V from the ESC and distributed to the proper locations. It also took in 12.6 volts by way of header jacks and was put through an amplifier to drop the voltage, then

sent to the microcontroller for battery monitoring (for further information please see section 5.7).

Due to the fact that this project would not only contain PCB design and interfacing, but would also have a testing configuration that needed to be address. As far as the interfacing off of the PCB, nothing changed. However, the PCB would initially be a more powerful Tiva™ C Series TM4C123G Launchpad and a sensor stick containing the altimeter, gyroscope, digital compass, and accelerometer that will later be implemented in the PCB.

The sensor stick was a useful tool for initial implementation due to the fact that its operational ability had been verified. Secondly the sensor stick came with a present communication set up of I2C. With this able to be worked on in the initial testing phase reaffirmed the decision made to move forward with the I2C interfacing.

The launchpad described was a Texas Instruments product that contained two microcontrollers, on board in circuit debug interface and USB for computer interfacing. On top of this it also provided preloaded RGB application forty on board I/O, switch selectable powering, 2 user switches, a reset switch, and test LEDs that were useful in troubleshooting, but most weren't implemented on the PCB.

6.4. Coding Overview

As this project was a big undertaking from a programming stand point the software used needed to be something that the team was comfortable with. Due to the fact that the microcontroller being used is produced by Texas Instruments, Code Composer Studio version 5 is the best integrated development environment for this project. Code Composer Studios is not only the most comfortable compiler for the team, due to the fact that much of the programming done by the team has been through CCS, but it also provides debugging support and JTAG based development. The microcontroller has a JTAG programmable ARM processor in a reduced instruction set microprocessor powerful enough to perform all needs of the system. An added bonus is that Code Composer includes a real time operating system, or RTOS for short.

A real time operating system is used to implement real time requests and return data output from the system. This in most cases is able to operate without lag or buffering delaying the data flow. Through the implementation of this program the testing will be more thorough. Due to the complexity of the sensor data that is used to maintain the entire flight system an ability to readily read the sensor responses is crucial as the project moves forward. The ability to register real time data fed through the system allows for a better understanding of what will be the output to other components as well as the ability to maintain a stream of control data in order to diagnose any flight stability issues before test flight.

The system is implementing a navigation computer that handles the autonomous flight and imaging as well as transmission of these images to the ground station. This system will be maintained on a Raspberry Pi model B. This runs on a Linux Raspbian operating system, a Debian variant operating system that is optimized for Raspberry Pi™

hardware. Through the use of this powerful system, communication is be attainable with the microcontroller and the camera. Programs for this system have the ability to run straight out of the command prompt, and even upon boot-up. It falls upon this operating system to test out the DTN software during flight.

All code written on the quadcopter will be written in C. This is a general purpose programming language that is one of the most popular programming languages used. It is a mid level language above assembly that CCS can interpret and compile into hardware language before sending to the device. With this language being the most dominant programming language as far as team understanding and implementation, it was an obvious choice with which to move forward.

For the ground station GUI, the software was written in Java. There are several GUI builders available through the Eclipse developer IDE. The software lead on the team is experienced most with this language and development environment, so this was the most logical decision. The user interface was designed using the interface tool, but the background mechanics were programmed using the Java language itself.

7. Prototype Testing and Evaluation

7.1. Microcontroller RTOS and Peripheral Driver Testing

Making sure that the software for the Tiva C™ software was ready for flight required several phases of testing in order to show the predictability and accuracy of the software created and used. There are several elements of the software that were tested individually before integrating them into a complete system. The Real Time Operating System (RTOS) and peripheral drivers are the subjects of interest for this section. The RTOS was running and tested before integrating the peripheral drivers.

The reason why an RTOS was used for this project was to prevent a single process from holding up the processor which would prevent other tasks from running. Making sure that all of the real time events that occur when implementing a digital control system do not get interrupted or blocked is critical to its operation. Before anything was done with any control algorithms or peripheral drivers, the RTOS must be running and tested. After installing the RTOS support software within Code Composer Studio™, the things that were tested was its ability to schedule tasks, handle priorities, and do preemption. Each of these features was tested separately using programs that forced the operating system to do just these tasks.

Scheduling needs to be tested so that tasks can run at fixed intervals. Testing this involved more than one task. Initially one task was created and be set to run at a certain interval, such as once every second. The team did this by toggling an LED. The interval in which the LED was on and off matched the specified schedule. This was confirmed with an oscilloscope. The next thing that was done was adding another task for the operating system to manage. This was done with another LED set to toggle at a different rate. Again, the accuracy of the tasks were measured with an oscilloscope. Different intervals were also tested and a scheduling conflict was tested as well. The team tested certain intervals that overlap and cause conflicting schedules. This is so the team could observe the behavior of the TI-RTOS in order to understand what would happen in this situation.

Priorities are tested at every level. The complete number of priority levels the RTOS supports was tested. The highest priority task would always come before any other priority. Priorities are important when multiple tasks come in at the same time and the RTOS has to decide which one will get executed. Figure 56 shows a illustrative example of three processes: GPS, accelerometer, and magnetometer, being placed in the RTOS (represented by a funnel) at the same time. The one with the highest priority comes out first. This was tested by creating two or more tasks with different priorities. Task one has the highest priority and turned on the red LED built into the evaluation board for one second. Task two has a lower priority and turns on the green LED for one second. They both were scheduled to launch at the same time. This was easy to confirm just by seeing that the red LED came on first. The same test was repeated with both tasks at a lower priority level. The result was the same. When both tasks have the same priority, the RTOS had to pick one even though they were scheduled at the same time. This was another behavior that was observed and documented.

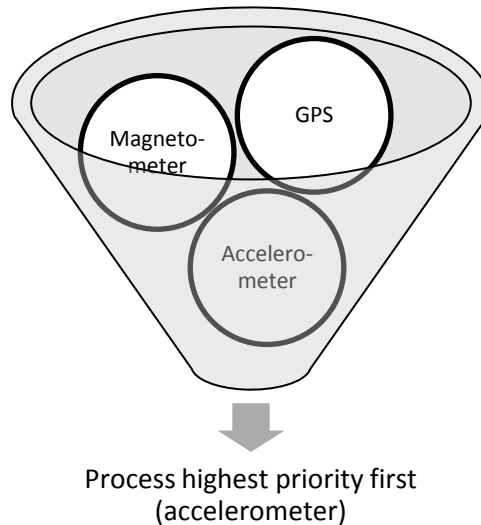


Figure 56: RTOS Priority Funnel

Preemption was tested similarly to the priority testing above. The difference was the lower priority task (task two) turned on the green LED for five seconds and the higher priority task (task one) turned on the red LED for one second. Task two was scheduled first and task one was scheduled one second after task two. Since the RTOS preemption worked right, the green LED turned on for one second, then switched to red for one second, and finally back to green for the remaining 4 seconds. This test was also ran again with task two at a higher priority than task one to make sure that preemption does not occur when the task with the higher priority is already running. The last test was to have both tasks at the same priority level. The operating system just ran the task that came first.

Peripheral driver testing was done through software and physically. At this point the microcontroller was already tested during the RTOS test since the team used the evaluation board to test it. Software had to be used to test the peripheral sensors since they provided feedback through digital communication. The team used the microcontroller for all of the peripheral devices. The accelerometer, gyroscope, magnetometer, and altimeter all communicate via I2C. Each slave device responded when a message was sent to it to retrieve data. None of the other devices on the I2C line sent any data when their address was not called from the master. The GPS and the Raspberry Pi TM communicated via UART. The 2.4 GHz remote control receiver is proprietary and setup like described in Section 5.3. The accelerometer, gyroscope, and magnetometer are on the same board. These were tested separately when communicating with them and had to be tested together physically. The altimeter, GPS, and Raspberry Pi TM were physically independent. Each device was tested to make sure they match the needed specifications for this project.

The accelerometer was setup on the microcontroller to forward data to a laptop. After connecting the accelerometer to the microcontroller, the self test was initiated. The accelerometer produces an electrostatic force that acts on the internal sensor in the same

way that acceleration would. The self-test output is scaled based on the supply voltage. Using a 3.3V power source scaled the output by 1.77 on the X and Y axis, and 1.47 on the Z-axis. Since the accelerometer was set in the ± 2 range and the power supply was 3.3V, the output fell within the following scaled parameters:

- $88.5 \leq X \text{ axis} \leq 955.8$
- $-955.8 \leq Y \text{ axis} \leq -88.5$
- $110.3 \leq Z \text{ axis} \leq 1286.3$

When all of the axes were within the boundaries, testing was continued. If they were not then the test would have been repeated. When the self-test continues, the other three gravitational force ranges were checked to see if something may have been wrong with the initial parameters. The accelerometer would have been replaced if the required ranges were not satisfied. Next MATLAB was used to graph the data in real time. All three axes were tested. The magnitude of each motion was documented in order for the team to fine tune the sensor data into realistic values for the microcontroller to make determinations from. This test was analyzed with movements in a similar magnitude that the accelerometer would feel on the quadcopter. The new data that comes through the FIFO and placed in the registers was updated at a rate of 800 Hz.

The gyroscope does not have a self-test feature. It was tested similarly to the accelerometer by having the microcontroller forward the data to a laptop and viewing the results in MATLAB. This data was read from the upper and lower half registers of the X, Y, and Z axes. All directions of movement were tested and verified by rotating the gyroscope on each axis. The range and magnitude of each movement was documented for fine tuning when it was on the quadcopter. The sign was also verified. When the gyroscope was rotated clockwise on an axis it was negative, and it was positive when rotated counter-clockwise.

The magnetometer has a self-test option that was more involved than the accelerometer. The magnetometer internally generates its own magnetic field by pulling 10mA through the offset straps. The self-test sends a pulse, takes a single measurement, sources 10mA to generate about 1.1 Gauss, then takes a second measurement. The two measurements are then subtracted and stored in the X, Y, and Z output registers. The process of doing a positive self-test is shown in Figure 57. CRA and CRB stand for Configuration Register A and B respectively. The MR is the Mode Register. During self-test and flight, register 2 (MR) was always in continuous-measurement mode. When self-testing, the gain would start at 5 and if the data registers were not within the required limits, the gain was increased. When the gain was increased, the next measurement was skipped. This is why there were two delays if the gain was increased. If the gain exceeds 7 then one or more axes would not have passed the self-test. This test was done for both positive and negative biasing.

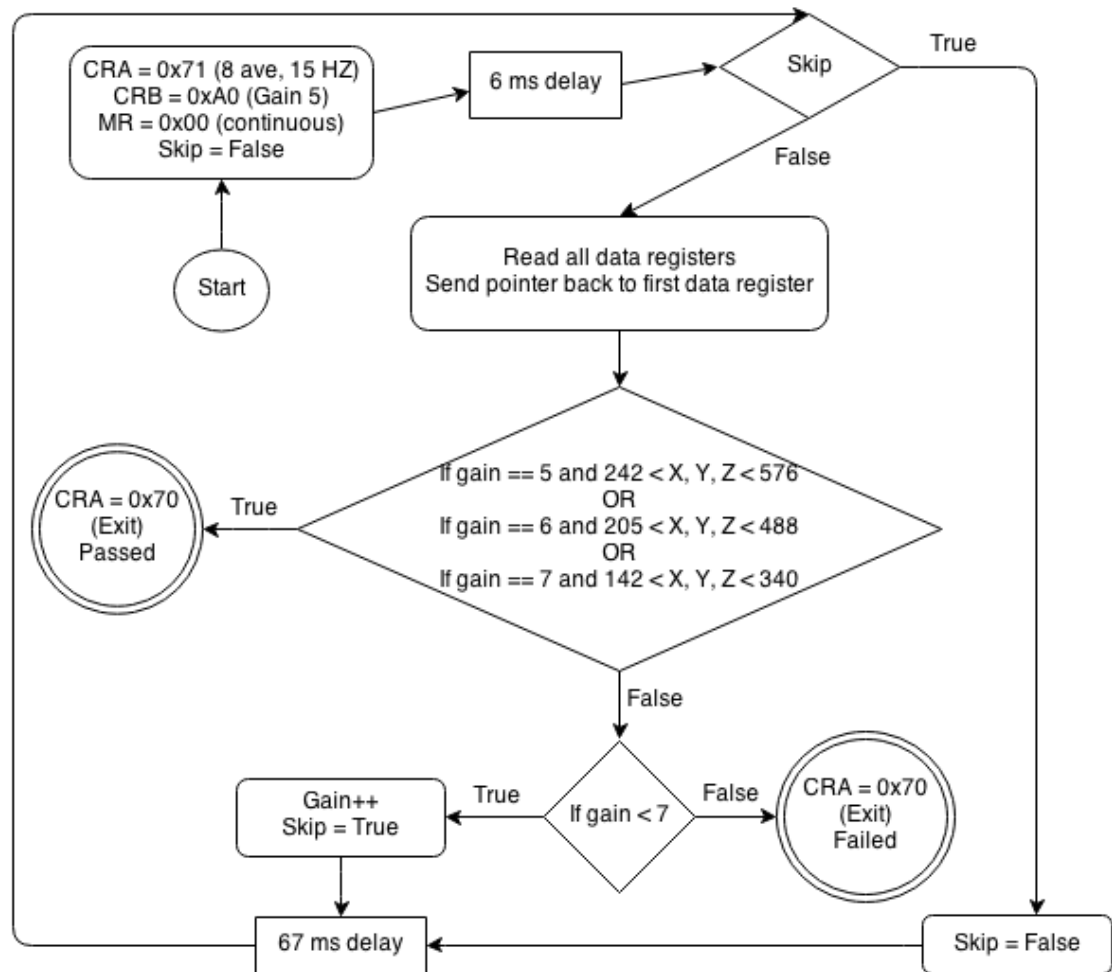


Figure 57: Magnetometer Self-test flow chart

Once the self-tests passed, the digital compass was tested against a real compass. The values sent to the computer would show the magnetic direction of the earth. These would also change as the sensor was moved or rotated. The sensor detects a magnetic field in three dimensions so the sensor was rotated in every possible direction. North was consistent throughout the test.

The altimeter was tested the same way via I2C. This required the team to use a tall building in order to see if the data provided from the device was accurate. The first thing that was tested with the altimeter was to read the calibration data from the E2PROM. In order to use the altimeter, the algorithm discussed in Section 5.3 was implemented. Once this was all working, the pressure was compared to the actual pressure in the room with another device and by looking up the approximate pressure for the area the sensor was being tested in. The pressure then was converted into height. At ground level the pressure was read and set as a reference point of 0. The device was moved up a floor to see a significant difference. This was compared to the actual altitude and repeated a few times to get a set of data to represent how precise the measurements were. It showed a lower pressure the higher the device was and vice-versa. This was good for general flight

but the most critical time that altimeter would be needed is during landing if the quadcopter was autonomous. If the quadcopter descends too slowly, it would take several minutes to land. If it is too fast then the quadcopter would be damaged. The ideal solution would have been for the quadcopter to slow down its decent as it gets closer. The altitude that was tested the most was in the 5 to 0 foot range because that was when the quadcopter would be about to land. Measurements were taken every inch from 0 to 5 feet to see how accurate the sensor was at such a low altitude. The altimeter's datasheet claims that the altitude noise is 0.25m which this was verified inside and outside.

The microcontroller also forwarded the UART data from the altimeter to a laptop just like it did with the I2C testing. The GPS was mainly tested outdoors so it had a line-of-sight signal to the satellites. This was tested against the GPS in an Android phone and the data provided from Google Earth TM. It was within ten feet of accuracy and tested in various locations. The response time was tested on a clear day and a cloudy one. The output also was observed when there was signal. This checks if incorrect data was supplied when there was no signal. The six main NMEA sentences were verified with each other to make sure the data the GPS was providing was consistent. The frequency at which each one occurs also was noted. This helped decide which sentences to read the most or to reference from more frequently. The GPS was tested when it was stationary so the latitude and longitude stay static and the speed was zero. It was then tested while moving. This was done in a car so the testers can use the speedometer as a reference. The things that were checked during these tests were the actual current location, speed, direction, changes in dilution of precision, and active satellites. Based on all of the testing, the team found the optimal sentences to use for each state the quadcopter was in during the mission.

The Raspberry Pi TM communicated through UART, but was not a sensor. The only thing that was tested for the microcontroller was sending and receiving messages with the Raspberry Pi TM. This was easily done by sending a number to the Raspberry Pi TM and having it double the number and return it. The camera was attached directly to the Raspberry Pi TM and accessed ground station. It was tested by the Raspberry Pi TM taking a picture with it every five seconds. The camera was facing a clock with a visible second hand. The pictures were stored on the SD card. The SD card was removed and read from a computer. Each picture showed a time that was five seconds apart. This test proved the Raspberry Pi TM can take pictures on command.

The receiver had individual wires connected to the microcontroller. The handheld transmitter controller had to be used to test this part. A program had to be written for the microcontroller to send information about what was being received from the receiver. Each movement with the joy sticks on the controller were detected by the microcontroller and sent to the laptop. All ranges of motion with the controller worked with any speed of movement. There were no glitches. This was important to test for reliability and accuracy since these movements were all mapped to what the microcontroller sends to the ESCs.

The final step in this section was to integrate the peripherals in the real-time operating system. Each peripheral had a different schedule and priority, as discussed in Section

5.3. Out of all the peripherals, the accelerometer and gyroscope had the highest priority and scheduled rate. The receiver was in second place followed by the altimeter. The UART Raspberry Pi™ task had the second lowest priority but was still important since it tells the ground station where it is and what it's doing during the mission. The actual frequency of each access to the peripherals was done experimentally. The integration of all this was running through both the RTOS and peripheral drivers test again with the peripheral drivers test within the RTOS test. Preemption, scheduling, and priorities all worked with the peripherals. This whole test section may seem inefficient since the RTOS and peripheral drivers could have been tested at the same time. The reason for this method of testing was to avoid any uncertainty when something does not work right. This alone saved time and prevented potential problems.

7.2. Filtering Algorithms & Control Loops Testing

In this section the testing of the filtering algorithms and the control loop testing will be given. These tests were performed after the RTOS and peripheral drivers were proven to work properly. The least squares filter was shown to not have reduced noise from the sensors much and was not used to provide a proper estimate of position based on data since the data was already very clean. The control algorithm was also tested as well as tuned.

For the least squares filter, data coming out of the sensors was retrieved and plotted in order to show that the filter was working properly. The sensors were read by the microcontroller where they were sent serially to a computer to be processed in real-time as well as post-processed. As the data was coming in, there are two methods that were employed to process the data. In order to test for accuracy and response time, MATLAB was used to plot the data in real-time. Using this method, even small amounts of noise could be seen on the plot as well as any inaccuracies with the data output, even though there weren't many. The other method was to represent the data that was coming out of the sensors as a three dimensional cube. This method only worked for the gyroscope, but it was good for determining if there was a drift in the sensor, or jitter. The ability to track movements was also well represented using this method. Using these two methods, the filter was shown to not improve the quality of the data, while not increasing latency since the data was very clean to start with.

After the filters were decided to be unnecessary, the PID control algorithms were tested. This was done by using various steps to show that the control loops were providing the right compensation based on the input. There are three parameters for each axis that were tuned in order to make the quadcopter stable. The proportional gain, the integral gain, and the derivative gain for each axis was tuned. Before assembling the quadcopters, the control algorithm was tested by reading the controller's PWM outputs on an oscilloscope and verified that they were showing the right control action based on the input. The full range of the PWM was tested.

To tune a quadcopter with a PID controller, the team found the method that worked best was to set P, I, and D to zero and start with proportional with pitch. The quadcopter was then tethered between two objects restraining one axis at a time. Proportional was

increased until the quadcopter started to over-shoot its correct position. The gain was then backed off until the overshooting stopped. Next the derivative was increased until fast oscillations were seen, and backed off a little bit until the fast oscillations stopped. Proportional was then revisited the same was as before, this time the team was able to raise it higher than last time. The integral was the last gain to tune. This was tuned similarly to derivative and ended up with a much smaller value. Next roll was tested the same way, and then yaw by tethering the quadcopter from above.

When the quadcopters were ready, the control algorithms were first be tested while the each quadcopter was tethered. The goal was to get the algorithm working first and then transfer the program to the other quadcopter for use later. The quadcopter was tethered for the initial testing in order to make sure that the system wouldn't destroy itself when it flies for the first time. When the algorithms were tuned enough so that the team felt comfortable enough to fly it without a tether, un-tethered flights began.

7.3. Hardware Testing

The hardware testing involves all of the components and systems that were not programmable like the microcontroller, or semi-programmable like the sensors. This includes the motors, propellers, electronic speed controllers, lithium-polymer battery and charger, power breakout cable, Wi-Fi on the Raspberry Pi TM and the laptop, and the power distribution system. In general throughout the hardware test plan, one component was added at a time. Each test was built upon another test and that tests the components in series like shown in Figure 58.

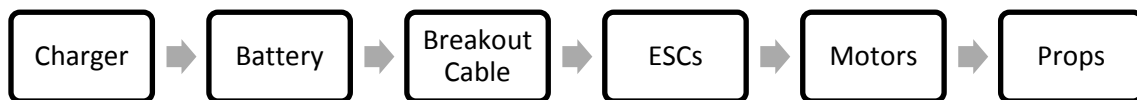


Figure 58: End-to-end Power Testing Sequence

The battery was a 3 cell lithium polymer 5000mAh battery. The main plug fits into the power breakout cable. The plugs on the end of the breakout cable all fit in the ESCs. The battery also fits the plug of the charger. This was needed in order to charge the battery. Using a voltage meter, a reading was obtained when the battery was drained and the voltage was less than 11V. Next the team plugged the battery into the charger and allowed it to charge until it was complete. After unplugging the battery from the charger, it had a voltage of 12.6V. The temperature was something that was monitored when first using the battery. This was done while the battery was charging and discharging. If the battery got above 50°C when it is discharging, then the load would be pulling too much current for this battery. A higher rating battery was not necessary to obtain. Charging was done at a slower rate than the rate at which the quadcopter discharged it so the battery did not get very hot. If it did then it would have been due to overcharging. Some cheap battery chargers are known to do this. If it was overcharged too long the battery would have expanded which causes permanent damage.

The motors and the electronic speed controllers (ESCs) were tested together. The motor controllers were tested using pulse-width modulation from the microcontroller. The motors were tested the entire voltage/speed range. The ESC was able to drive the motor at full speed and completely stop the motor. This tested the functionality of the ESCs and the motors since they should work at any speed in-between the supported ranges. The total current of one motor and ESC combined did not exceed 18 amps with or without the propellers attached. The speed and torque was verified and nearly identical for each of the four motors with ESCs. The current they pulled at each speed was observed through the entire voltage range with the blades on so the team has a current profile to calculate minimum and maximum current consumption and runtime. The thrust was measured using a weight and a digital scale. The motor and propeller were attached to the weight to hold it down. Once everything was on the digital scale the total weight was subtracted by using the built-in tare feature on the scale. The thrust was negative. Each motor was compared for consistency.

The USB Wi-Fi dongle in the Raspberry Pi TM was connected to the laptop through its Wi-Fi card. This required the laptop to host a network and the Raspberry Pi TM joined it. Communication was tested in both directions. The Raspberry Pi TM was able to send statuses and other information periodically. Next the range between the two devices was tested. This test did not require DTN. The Raspberry Pi TM remained at ground level. One team member walked away with the Raspberry Pi TM connected to the laptop via Wi-Fi and saw he could go about 50 feet before the connection was lost. This was the approximate distance that the quadcopter can fly away from the ground station.

The Raspberry Pi required a 5V power source and the Tiva C TM microcontroller required a 3.3V power supply. The only source of power on the quadcopter is the 12.6V Li-Po battery. The ESC's provided a 5V source up to 500 mA each, totaling 2 amps. These were tested in order to prevent a brown or blackout for the processors. They were first tested by connecting them to a fully charged battery. Next they had a test load that pulls a little more than what the maximum power consumption was on the microcontroller and the Raspberry Pi TM. The voltage remained at 5V. The battery was then discharged down to 11V. This did not cause the voltage to dip on the output of the regulators. This was important to test because having a low battery would cause the flight microcontroller to turn off and the quadcopter would crash.

All of these components were tested rigorously to ensure that there would be no problems during flight. The power train was the main system in this section that was tested. When dealing with large currents it was not a bad idea to check on how warm or hot all of the components were getting. Generally if something was wrong the component would be too hot to touch. The battery would have been disconnected immediately and the problem would have been investigated. Sometimes it's a faulty component, other times it is the implementation.

7.4. Manual Flight Tests

A manual flight was conducted with a RF controller that transmits movement commands to the quadcopter. This is how a typical remote control quadcopter that one could buy

from a hobby shop is controlled. At this stage the quadcopter had all of the sensors it needed to stabilize itself. This includes the accelerometer and gyroscope. Other sensors such as the GPS, magnetometer, and altimeter were not required for self-stabilization. This test demonstrated that the quadcopter could self-stabilize, move in a desired direction on all three axes of motion, function with an interrupted or weak signal from the controller, and fly outdoors. After that all of the peripherals were tested on the quadcopter during flight. Figure 59 shows the hierarchical format of this test phase.

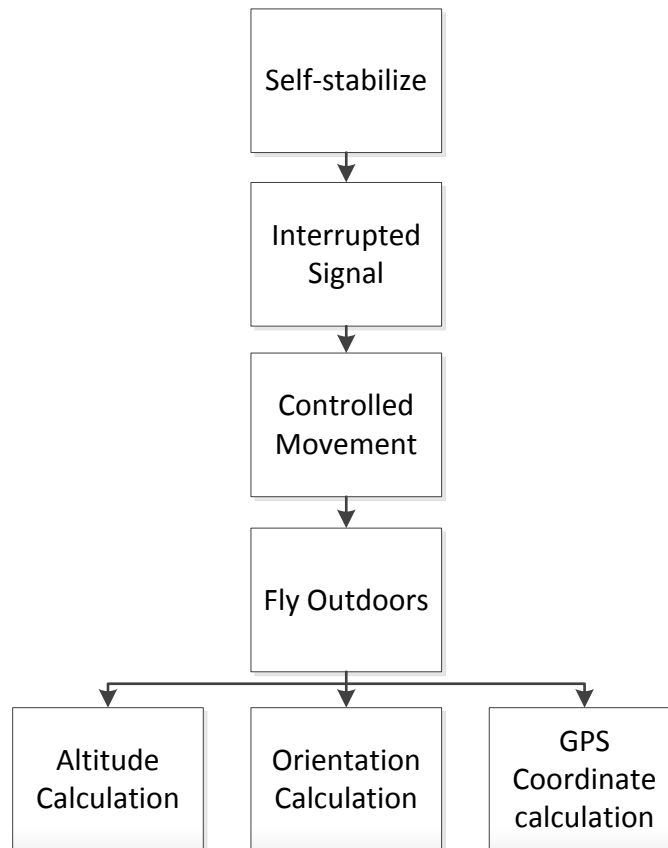


Figure 59: Manual Flight Test Plan

Self-stabilization is something that the quadcopter does regardless of whether or not the controller is transmitting movement instructions to it. The system continuously makes fine adjustments to the motor speeds in order to stay level in one position at the current altitude (not precisely since the altimeter was not required) based on the accelerometer's feedback when the controller was idle. This was first tested indoors with provisions for crashing. This was tested by turning on the quadcopter, providing thrust in order to get the quadcopter to lift off the ground and then backing off the thrust to keep the quadcopter from gaining altitude. The user with the controller did not have to stabilize the quadcopter; he only had to move it in any desired direction including yaw. When nothing was being operated on the controller, the quadcopter stayed somewhat in its position in a controlled environment. It was normal for the quadcopter to start drifting as long as it was gradual and not drastic.

Movement was controlled with the two joysticks on the controller. Each movement was tested. This was first tested indoors with provisions for crashing. The left joystick controlled altitude and yaw. When it was pressed up, the quadcopter would move up the z-axis. When was pressed down, the quadcopter moved down the z-axis. This was mainly be used for lifting off and landing. When the joystick was pressed right, the quadcopter would begin to spin clockwise and when it was pressed left it would spin counter-clockwise. The right joystick controlled leaning forward, backward, left, and right. Leaning in a certain direction caused the quadcopter to move in that direction. Pressing the joystick in any direction on the x-y plane caused the quadcopter to correspondingly move in that direction. Once all of the directions were confirmed functional, multiple movements at the same time were tested. This means the quadcopter would be able to perform x, y, and z axis movement at the same time, even with rotation. This was possible but at times made the quadcopter seem a little unstable.

When all of the above tests were successfully performed and all of the revisions have been finalized, the quadcopter was flown outside. This was done to test the quadcopter's ability to fly in situations where wind or other environmental changes were present. First the team flew the quadcopter outside with some wind. The quadcopter was pushed around by the wind, but did not become unstable since the stabilization system compensated for any involuntary movements. Next the team confirmed that the quadcopter could fly at a decent altitude and distance from the starting location. Range testing was done at ground level. The quadcopter was at rest on the ground and the team member with the controller walked away. After about every 50 feet, the user was able to spin the blades of the quadcopter to confirm that the connection was not broken. Once a distance was reached where the controller would no longer communicate with the quadcopter, the user would then walk 50 feet towards the quadcopter and see if he could fly the quadcopter to his location.

Now that the quadcopter could function as it should, the team tested the other devices that were used on the quadcopter. This included the altimeter, magnetometer, and GPS. The altimeter allowed the microcontroller to know how high the quadcopter was flying. With this integrated on the board with the microcontroller, it was able to read and adjust to it. The first test was to see if the quadcopter could hover more accurately when commands are not being sent by the controller. Next, the team hardcoded an altitude to reach such as 50 feet and see if the quadcopter could recognize that it reached that height by turning on a red LED when it was flown to that height. The LED stayed on for visibility purposes. The next test was to simulate a landing. This will simply blink an LED at a fixed rate when the quadcopter is above 5 feet. When the quadcopter descends, the LED will blink faster the closer it gets to the ground. When the altimeter detected that the quadcopter was within a foot off the ground, the LED turned off.

The magnetometer was easy to test since it only involved the quadcopter's rotation on the z-axis (yaw). When the copter was flying it was setup to log the orientation of the flight the whole time. One of our team members then wrote down the time and orientation every 30 seconds. This was also logged every 30 seconds by the microcontroller. Once the flight was over the team then verified that all of the readings were correct. The next step was to hardcode the orientation that the quadcopter should be at. The team set the

orientation of the quadcopter to always face north. Next the person with the controller then spun the quadcopter on the z-axis to see that the green LED was only on when the quadcopter is facing north.

The GPS allowed the quadcopter to know where is on the earth. This was tested by manually flying the quadcopter around and collecting GPS coordinates the entire time. When the flight was over the team then verified that all of the data was correct. The team then hardcoded in GPS coordinates for the quadcopter to be manually flown to. This was done within range. As long as the quadcopter was not at the correct coordinate, the red LED did not turn on. Once the quadcopter arrived at its destination, a green LED turned on and the red LED turned off.

All of the tests that were performed demonstrate all of the necessary functionalities that the quadcopter needed for this project. They were tested multiple times in different ways and areas. Self-stabilization was the basic function that had to be mastered before anything else can be developed for the quadcopter.

8. Operation Manual

8.1. Manual Flight

In this section, the overview of how to fly the quadcopter manually is given, along with an explanation of the safety features of the quadcopter. Quadcopters, especially ones of the size developed in this project, can tend to be very dangerous if users aren't careful. Special care must be taken around the propellers and motors to avoid getting cut. When flying, special care must also be taken to avoid crashing the quadcopter. Any crashing may cause mechanical failure of the frame or the propellers.

Controlling the quadcopter manually is done using the RC controller unit for controlling various parameters. Figure 60 shows how the RC controller was setup for this project. Notice that the joystick knobs on the left are used for controlling throttle and yaw, and the joystick knobs on the right are used for controlling pitch and roll. Next to the joystick knobs are little switches that are used for trim of each of the control parameters. This is built into the receiver unit as a function for fine tuning of the parameters. The controller also has a few switches on top of the controller as well as potentiometer knobs. These are used for some of the safety functions of the quadcopter as well as the tuning process.



Figure 60: Turnigy Controller Unit Diagram

Booting up the quadcopter is a multi-step process that must be done carefully. The first thing to be done is to TURN ON THE TURNIGY CONTROLLER UNIT. This is important because while the quadcopter is booting, the user wants to make sure the quadcopter doesn't receive any errant signals that would cause the quadcopter to behave unpredictably. There are built in functions to prevent this from happening, but it is still good practice to always turn on the controller first. The second thing to do is to place the quadcopter on a level surface and plug in the quadcopter battery to the ESCs and immediately remove all objects from the propeller area. This is more of a precaution just in case anything were to happen. The third thing that happens is a power LED and a user LED turns on, the quadcopter is ready to fly after the second user LED starts blinking. This signifies that the sensors are done initializing and calibrating. It is important to note that the quadcopter must be perfectly still and on level ground when initializing, or else this could ruin the sensor calibration and cause unpredictable results. After all these steps have been completed, and the LED is blinking the quadcopter is ready to be armed and flown.

The quadcopter has several safety features that are used to protect the quadcopter and more importantly, the people using it or watching it. The only way for the motors to be throttled is if the motors are armed. There is a state machine that determines whether to arm the motors or not that serves as the primary safety function of the system. The first requirement of the system is to make it through initialization properly. Neither the user nor the flight control software is capable of arming the motors until after the sensors have been initialized properly. After this point, the user must deliberately use the safety switch on the controller in order to arm the motors. The throttle must also be above a certain threshold to arm the motors too. After these requirements have been met, the quadcopter will be armed and the propellers can start spinning. The quadcopter PCB displays an "armed" signal to the user by turning of the user LED one turns off. The last safety function is a timeout function used to determine when the last set of good data from the receiver came in. If the user hasn't received data in over 500 milliseconds, the flight software automatically disarms the motors. This can cause the quadcopter to crash in flight, but rather than having an out of control quadcopter it is better to drop it and recover what is left.

Flying the quadcopter is not an easy task, and some prior quadcopter flight experience is necessary. When the motors are armed, the user can throttle up the motors in order to gain some altitude, and use the roll, pitch, and yaw sticks to maneuver that quadcopter.

In conclusion, the quadcopter can be difficult to fly as a beginner, but with some practice and experience it is possible to fly the quadcopter with proficiency. There are several safety function built into the flight computer software in order to keep the user and others safe when flying the quadcopter. It is important, however, to make sure that the pilot always is responsible when flying the quadcopter as to not endanger others or the quadcopter.

8.2. Using the Ground Station

In this section, the overview of how to use the ground station user interface is given. The ground station interface was built to display the telemetry of each quadcopter, show live quadcopter location on a Google maps overlay, and be able to send commands to each quadcopter.

The ground station interface, shown below in figure 61, is quite self explanatory for most parts. On the left is the live telemetry display, which shows the latest info on the position, state, altitude, battery, and packet information. The packet number and packets received entries are important to have in order to gauge relative data latency and rate. This can help to determine if packets are actually still coming in even if the GUI isn't updating. On the right is the Google maps interface. There are buttons available for zooming in and out, and moving around in the map. The map can't be dragged like a lot of Google map interfaces, but that is due to the limitations of the free software that was picked. On the bottom of the screen is the command interfaces for sending and receiving information from the quadcopters. The user can command any individual quadcopter to take an image and send it, or send all the images, or clear any databases. When the quadcopter is done, the quadcopter can save a log of all the telemetry packets received into a composite, time stamped log file. Other features of the GUI include the ability to switch between Windows and Linux easily under the 'edit' tab. This feature was built in during the development and debugging stage as a way to switch easily without having to recompile the code.

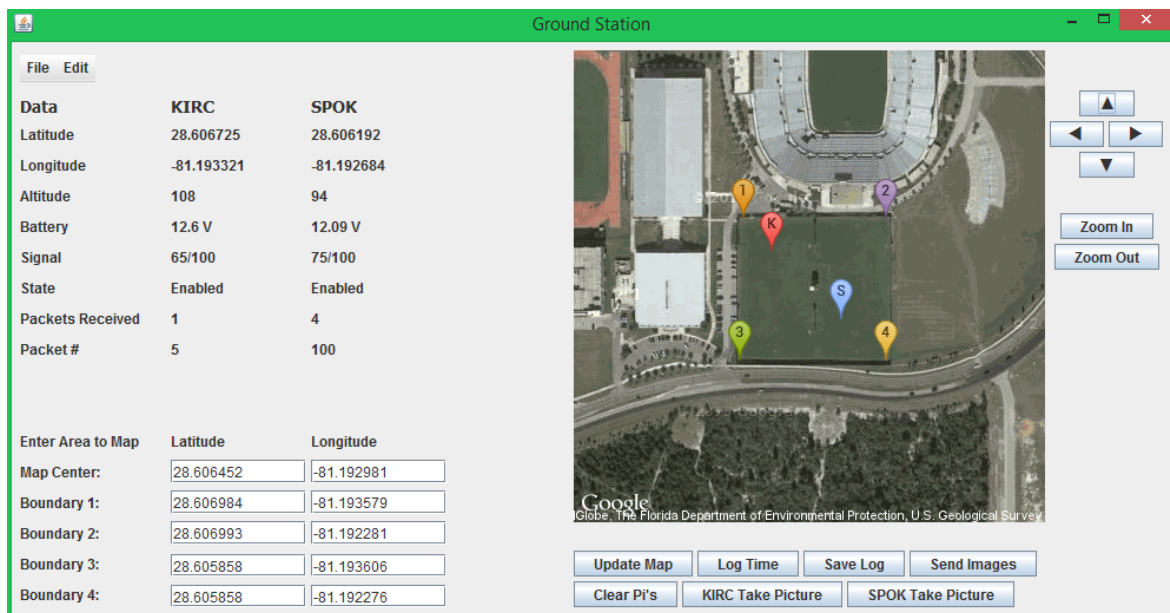


Figure 61: Ground Station User Interface

Before the ground station starts receiving data, the quadcopters must be initialized first. It takes some time for the Raspberry Pis TM to boot up before they start transmitting data, so it is recommended to wait until packets are being received on the GUI before flying, which could take a few minutes. It is also necessary to log into the Ad-Hoc, peer-to-peer

network before turning on the quadcopter. This ensures that the Raspberry Pi will always connect properly to the network. As long as these steps are done first, the user should be able to wait for a few minutes, and the interface will automatically start updating.

It was intended to make the quadcopters in this project autonomous, and the user interface was designed to allow for that in future developments. The user interface can be reconfigured to add or rearrange some buttons that would allow the user to input a boundary to be mapped or a coordinate to fly to. Safety functions can be easily added that can force the quadcopter to take off, or land. So it was left pretty open for future development to allow for more functions and features for autonomous flight.

In conclusion, the user interface was intended to be easy to use and easy to reconfigure. The interface allows users to view telemetry and current location which creates a fairly interactive experience. It is functional in all its aspects, as the user can actually send the quadcopter commands to take images and send them and they work.

9. Administrative Content

9.1. Milestone Discussion

As mentioned in the project description section, the Knight's Intelligent Reconnaissance Copter (KIRC) project is split into three prototyping phases. Each of the three phases represents a logical progression of how the team thought the project should be broken up. Throughout all the phases, there was research, design, construction, and testing. Each of the phases was evaluated based on the team's abilities to achieve the goals within each respective phase. In this section, the goals, milestones, and schedules are defined for each phase is covered in detail for both Senior Design 1 and Senior Design II.

The first phase was longest phase of the project, as it ran the length of Senior Design 1 entirely. In this phase, one of the goals was to begin research into different quadcopter designs, parts, and software algorithms. Once an adequate microcontroller, inertial measurement unit (IMU), and a GPS were found, we decided to do a first round parts order with these parts in late September of 2013. When the parts arrived, the team began coding of the microcontroller software that controlled the motors and the stability algorithms that made the compensation. During this, research was also done for the physical quadcopter parts, such as the frame, motors, props, and batteries. Once the software was written, and the quadcopter parts came in, the team began integration of the quadcopter. After being fully integrated, the first prototype quadcopter will undergo rigorous stability, maneuverability, and durability testing under various conditions in manual mode only. The details of the testing phase, and the test plans associated with it are given in chapter 7 of this document. The goal is to have the Tiva C™ microcontroller software completed and thoroughly tested before Senior Design II starts.

The second phase of the project started at the very beginning of Senior Design II. With the quadcopter now capable of stable manual flight, the team will shift its focus towards making the quadcopter integrated by creating a custom printed circuit board (PCB), and interfacing a camera to the quadcopter. For this part, the team will be split up into two separate task groups, which will hopefully speed up the progress during this phase. Two of the group members will be focused on creating a PCB, while the other two will be focusing on creating software to make the quadcopter able to send data with the embedded Linux computer. There was also significant work towards researching an image stitching software for use on the ground computer, and also development of a ground station interface. At the end of this phase there was doing more testing of the quadcopter, but this time on perfecting flight controller.

The third and last phase of the project will be to replicate the first quadcopter and make them autonomous. Starting midway through Senior Design II, the team will replicate the first quadcopter, and begin simultaneous flight testing. Also during this phase the team finalized the user interface, attempted to thoroughly test the dual copter system, and provided proof that DTN2 is working as NASA expected it to. There was two weeks of buffer time during this phase to allow for scheduling delays.

Below is a list of milestones and dates that was developed at the beginning of Senior Design II. This list does not include lost time, schedule delays, or other disturbances.

- | | |
|--|--|
| I. Phase 1 (Senior Design 1): Build 1 st Generation Copter | Aug 19 th – Jan 6 th |
| a. Research and Design I | Aug 19 th – Oct 18 th |
| i. Team Formation and Organization | Aug 19 th – Sept 6 th |
| ii. Software Design | Sept 1 st - Oct 1 st |
| iii. Hardware Design | Sept 1 st - Oct 18 th |
| iv. 1 st Round Parts Acquisition * | Sept 16 th – Oct 1 st |
| v. 2 nd Round Parts Acquisition ** | Oct 1 st – Oct 18 th |
| b. Software and Hardware Implementation | Oct 4 th – Nov 15 th |
| i. Software: RTOS Implementation | Oct 4 th – Oct 25 th |
| ii. Software: Peripheral Drivers | Oct 4 th – Oct 25 th |
| iii. Software: Sensor Filtering & Control | Oct 18 th – Nov 15 th |
| iv. Hardware: Assemble Frame & Motors | Oct 18 th – Nov 1 st |
| v. Hardware: Make electronics mount | Oct 18 th – Nov 15 th |
| vi. Use microcontroller to drive motors | Oct 18 th – Nov 15 th |
| vii. Use microcontroller to read RC controller | Oct 18 th – Nov 15 th |
| c. 1 st Gen Copter Integration | Nov 15 th – Dec 10 th |
| i. Assemble the Copter | Nov 15 th – 22 nd |
| ii. Tethered Flight Testing | Nov 15 th – 22 nd |
| iii. THANKSGIVING WEEK OFF | Nov 25 th – Dec 1 st |
| iv. Buffer Time | Dec 1 st – Dec 10 th |
| d. 1 st Generation Copter Testing | Dec 10 th – Jan 6 th |
| i. Test Manual Flight | |
| II. Phase 2 (Senior Design 2): Build 2 nd Generation Copter | Jan 6 th – March 14 th |
| a. Research and Design II | Jan 6 th – Feb 3 rd |
| i. Design PCB and send for fabrication | Jan 6 th – Feb 3 rd |
| ii. Research image stitching software | Jan 6 th – Feb 3 rd |
| iii. Ground Station Software Design | Jan 6 th – Feb 3 rd |
| iv. Navigation Computer Software Design | Jan 6 th – Feb 3 rd |
| b. Software and Hardware Implementation II | Feb 3 rd – March 14 th |
| i. Software: Ground Station | Feb 3 rd – March 14 th |
| ii. Software: Navigation Computer | Feb 3 rd – March 3 rd |
| iii. Mount PCB and Navigation Computer | March 3 rd – 14 th |
| c. 2 nd Generation Copter Testing | March 14 th – 31 st |
| i. Test Autonomous Flight | |
| III. Phase 3 (Senior Design 2) Replicate Copter and Testing | March 14 th – May 2 nd |
| a. Copter Integration | March 14 th – 31 st |
| b. Copter Testing | April 1 st – April 14 th |
| i. Test Joint Autonomous Flight | |
| c. Buffer Time | April 14 th – May 2 nd |

* 1st Round Parts Acquisition Includes Microcontroller, IMU, and GPS part

** 2nd Round Parts Acquisition Includes Quadcopter Physical Parts

*** This Milestone Outline Does Not Include Most Recent Dates and Scheduling

The Schedules for Senior Design I and Senior Design II were organized from the milestone outline above into the two Gantt charts given below in Figure 63 and Figure 64 below respectively. In Figure 62, we have the Gantt chart for Senior Design I, which has been modified from the milestone outline above with more updated dates from our schedule used. In Figure 63 we have the projected schedule for Senior Design II. The major points that the team wanted to achieve are: the construction of a stable prototype by the start of Senior Design II, a PCB with integrated sensors and processor by the end of February, a working autonomous quadcopter by mid-March, and two working quadcopters by mid-April. This, however proved over optimistic, and the autonomous section was never finished.

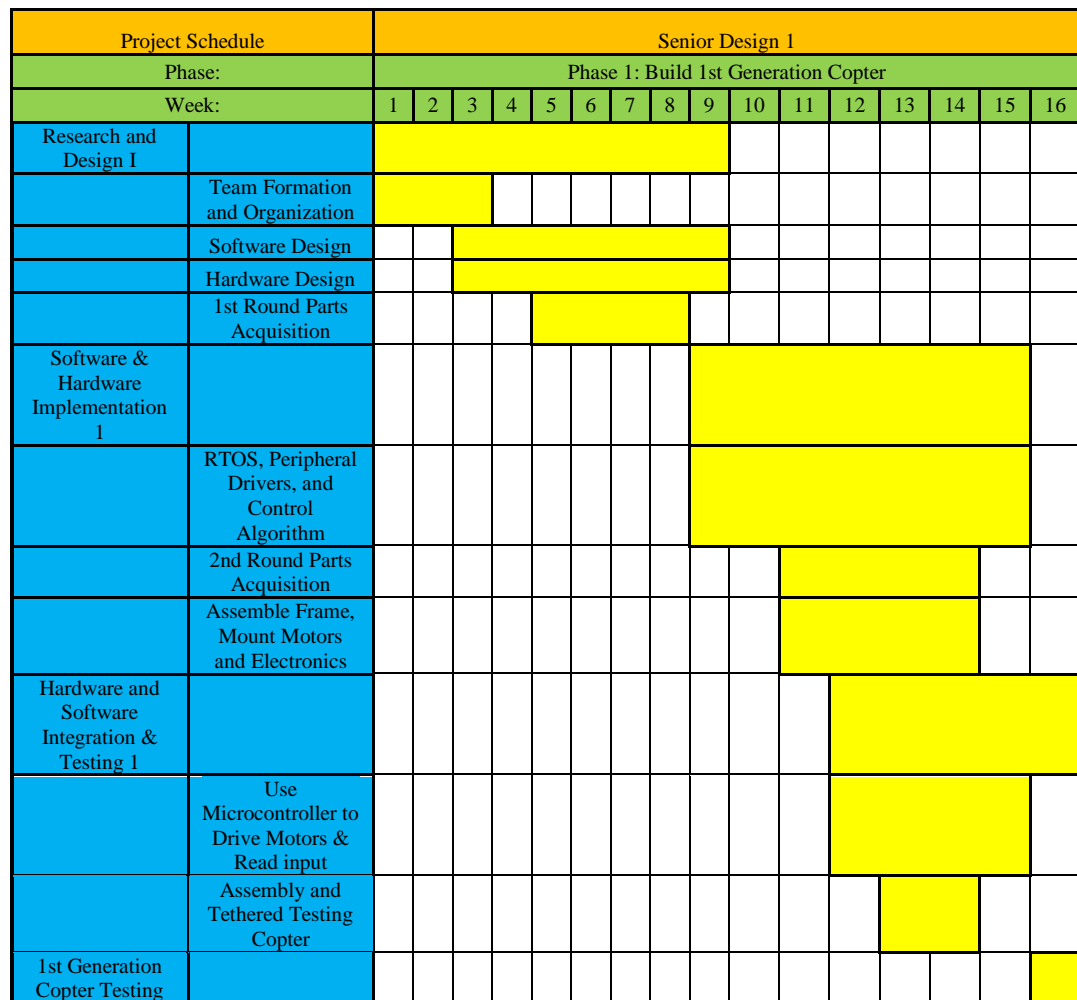


Figure 62: Senior Design I Schedule (Modified based on changes to schedule)

Project Schedule		Senior Design 2															
Phase:		Phase 2: Build 2nd Generation Copter						Phase 3: Replicate Copter and Testing									
Week:		17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Research and Design II																	
	Design PCB																
	Ground Station Software Research																
	Navigation Computer Software Design																
Software & Hardware Implementation 2																	
	Ground Station Software Implementation																
	Navigation Computer Software Implementation																
	Fabricate PCB																
	Mount PCB & Navigation Computer																
2nd Generation Copter Feature Testing																	
Replicate Copter and Testing																	
	Copter Integration																
	Dual Copter Testing																
	Buffer Time																

Figure 63: Senior Design II Schedule

The schedule has not gone completely as planned, as expected. One of the major events thus far that has slowed down the progress on the project was the government shutdown from October 1st through October 16th. With NASA providing most of the budget for the team, the government shutdown slowed down the acquisition of parts that was to occur from late September through late October. Some of the critical parts the team ordered themselves, as they did not wish to delay the project any further. Parts such as the Tiva C microcontroller and the 10 Degree of Freedom Inertial Measurement Unit were critical to the start of the software development so they decided to get the specific parts with their own money. The team planned for another possible government shutdown looming in late January by planning to have all the necessary parts acquired by then so that the schedule will not be hindered by the possible shutdown.

In summary, the first phase, building a first generation quadcopter, covers from researching quadcopters to coding and testing manual mode. Phase two, making the quadcopter autonomous, begins during Senior Design II. During this phase, we would like for our quadcopter to have an embedded Linux computer on board and be able to autonomously fly to location(s). The last phase, replicating the quadcopter, begins midway through Senior Design II. In this phase, our goal is to completely replicate the first copter and have them cooperatively image an area. Not everything went to plan, as things caused delays such as the government shutdown in October and several setbacks such as the control system tuning. This was not too detrimental to the progress of the project however, and only proved to be a minor setback over the long run. Even though not all the deliverables were achieved, the concept of the project was shown as a viable one, and there were enough deliverables completed to make both NASA and Dr Richie happy. Overall, though, the team has tried to stick to the schedule as the guideline of where the team should be at what point, and it has kept them on track to complete most of the deliverables.

9.2. Budget and Finance

Most of the budget for the KIRC project came from an unofficial NASA sponsorship. The project has been given a rough budget of about \$1000 to order parts necessary for outfitting the quadcopter, but the team has to buy the quadcopter parts themselves. These parts were eventually donated by Marc Seibert, the team's advisor from NASA. With the budget set by NASA, and the donated parts from Marc Seibert, the team has to allocate enough money for all parts. In this section, the budget allocation, the most up to date bill of parts, and any issues encountered so far with acquisition of parts.

With NASA's allocation of \$1000 dollars, the team decided to allocate about \$200 for avionics and control parts, \$400 for quadcopter physical parts, and the rest of the budget for other miscellaneous parts. The goal was to make the cost of the quadcopters as little as possible while not limiting the functionality of the quadcopters themselves. In

Figure 64 64 below is the preliminary budget for the entire project. This budget does not include the exact parts and their costs, but it was based on a preliminary web search of parts and costs based on things needed for the project. Parts already acquired or freely available (via free samples), were not added parts into the total cost of the project. Each price estimate for each thing on the budget was rounded up to the nearest \$5 and a quantity of each was given. Based on this estimate, the team came up with a rough cost of about \$800 for the whole project. This included all the physical parts, the batteries, the avionics, the computers, and receivers. Keep in mind, however, that this is not the actual budget (provided later), but a preliminary estimate and allocation table based on initial estimates and research. Figure 65 on the next page shows the breakdown of the parts overall.

Category	Item	QTY	Price Ea. (\$)	Total \$	Need By Date	Status
Quad:ControlSys						
...	Microcontroller Launchpad	2	\$15.00	\$30.00	9/30/2013	To be acquired
...	IMU Sensor Unit	2	\$25.00	\$50.00	9/30/2013	To be acquired
...	GPS Unit	2	\$50.00	\$100.00	9/30/2013	To be acquired
Quad:FlightSys						
...	Speed Controller	8	\$10.00	\$80.00	11/4/2013	To be acquired
...	Motors	8	\$20.00	\$160.00	11/4/2013	To be acquired
...	Props	12	\$4.00	\$48.00	11/4/2013	To be acquired
...	Frame	2	\$15.00	\$30.00	11/4/2013	To be acquired
...	Li-Po Battery (4-5 A-h)	2	\$40.00	\$80.00	11/4/2013	To be acquired
...	RC Controller & Reciever	1	\$50.00	\$50.00	11/4/2013	To be acquired
Quad:GuidSys						
...	Embedded Linux Processor	2	N/A		N/A	Acquired
...	Power Cable	2	N/A		N/A	Acquired
...	SD Cards	2	N/A		N/A	Acquired
...	802.11G Wireless Card	2	N/A		N/A	Acquired
...	High Resolution Webcam	2	\$50.00	\$100.00	1/6/2014	To be acquired
Ground:GndStat						
...	Laptop	1	N/A		N/A	Acquired
Quad:PCBHardW						
...	Microcontroller Standalone	2	\$10.00	\$20.00	2/3/2014	To be acquired
...	Accelerometer	2	\$5.00	\$10.00	2/3/2014	To be acquired
...	Gyroscope	2	\$5.00	\$10.00	2/3/2014	To be acquired
...	Magnetometer	2	\$5.00	\$10.00	2/3/2014	To be acquired
...	Altimeter	2	\$5.00	\$10.00	2/3/2014	To be acquired
...	Passive Components		Free		N/A	Acquired
...	Active Components		Free		N/A	Acquired
TOTALS	All			\$788.00	N/A	N/A

Figure 64: Generic Preliminary Budget Allocation

The parts will be ordered through a NASA contact, and for some parts there is paperwork that needs to be filled out. The team had to take into account, the purchase request process at NASA and allot at least a month of time before expect to receive any part requested. Parts arrived at NASA KSC in the shipping and receiving building where they were picked up by the NASA contact and be put in the team's custody for the remainder of the project. Parts bought by NASA needed to be handled with care not lose or destroy them, as they are still technically government property.

The most recent bill of parts is listed below in Figure 65, with status of each part, vendor website, and exact cost. Some of the parts were purchased directly from Texas Instruments, while others were from third party vendors such as Sparkfun or Amazon. Most of the quadcopter physical parts were acquired from Hobbyking, an online hobbyist retail store. The total cost so far is about \$671.22 which was less than the initial estimate.

Category	Item	Part #	QTY	Price Ea. (\$)	Total \$
Quad:FltContlSys					
...	Tiva C Launchpad	TM4C123G	2	\$12.99	\$25.98
...	10 DoF IMU Sensor	Misc	1	\$25.90	\$25.90
...	50 Channel GPS Reciever	GP-635T	2	\$39.95	\$79.90
Quad:FltSys					
...	Electronic Speed Controller	9192000131-0	8	\$12.49	\$99.92
...	Prop Drive 900kv/270W Motors	NTM2830S-900	8	\$14.99	\$119.92
...	8x4.5 Propellers	17000055	12	\$3.53	\$42.36
...	Q450 Glass Fiber Frame (450mm)	297000028-0	2	\$7.99	\$15.98
...	4S 5000mAh Li-Po Battery	N5000.4S.45	2	\$65.03	\$130.06
...	Turnigy Rx/Tx 9ch controller	TX-9X-M1	1	\$54.00	\$54.00
...	HXT 4mm to 4 X 3.5mm bullet Multistar ESC Power Breakout Cable		2	\$3.65	\$7.30
Quad:GuidSys					
...	Embedded Linux Processor		2	N/A	
...	Power Cable		2	N/A	
...	SD Cards		2	N/A	
...	802.11G Wireless Card		2	N/A	
...	RasPi 5MP Camera Board	7757731	2	\$34.95	\$69.90
Ground:TelStation					
...	Laptop	N/A	1	N/A	
Quad:PCB_HW					
...	Microcontroller Standalone		2		
...	Accelerometer		2		
...	Gyroscope		2		
...	Magnetometer		2		

...	Altimeter		2		
TOTALS	All				\$671.22

Figure 65: Detailed Budget, Bill of Parts

In the course of the project, there have been some unforeseen issues associated with the acquisition of some parts. One, as mentioned in the previous section, was the government shutdown that delayed the procurement of the avionics parts. Another issue was the federal ban on the government purchase of parts from China. Unfortunately, this really limits what technological parts the team can purchase. One of the ways the team worked around this is that the ban is specifically related to computer parts manufactured in China, but there is an approved list of manufacturers. For the computer parts we used, specifically from Texas Instruments and Sparkfun, we filled out a request for investigation form for approval of parts from these companies. These forms were approved right before the government shutdown, and we were able to order these parts after the shutdown ended.

In summary, the budget allotment of \$1000 from NASA is more than enough to cover the cost of the project. The goal to remain less than the allocated budget seems very realistic and attainable. The initial estimate for the total cost of the project was \$778 and so far, the bill of parts comes out to under \$700 for everything. The reason for the cost coming up much less than the initial budget is because the team found some vendors that have the parts they want for less than allocated for, and they also found some parts for free if sampled with the manufacturing company. The other cause is because the team also overestimated the budget on purpose in order to allow for any unforeseen cost. So far, though, the project has gone as planned from a budget perspective.

9.3. Team Organization

The KIRC project team is a group of four UCF seniors in engineering, with three electrical engineers and one computer engineer. The project has been divided into four areas, one for each group member. The areas of this project have been split into: Hardware, Software, Control Systems, and Team Lead. In this section, each team member's job is explained and how the team has handled communication and collaborative file editing. Member profiles and brief bios are provided in the appendices for reference.

Nathaniel Cain, an electrical engineering senior, is the team lead. He is responsible for providing administrative leadership to the project as well as document tracking and editing. He is also responsible for the integration of the systems and subsystems that make up the project. Nathaniel is the NASA liaison for the project as is the NASA co-op/intern. His mentor is providing the budget for this project as well as guidance when necessary.

James Donegan, also an electrical engineering senior, is the hardware and power systems lead. He is responsible for quadcopter assembly, power distribution, wiring, and PCB design. James' role is important in the assembly and integration of the quadcopter parts.

James Gregory, another electrical engineering senior, is the control systems lead for the project. His tasks include reading input from the RC controller, setting up a feedback controller, filtering the IMU sensors, and providing PWM output to the motors. This job is mostly software, and will require much collaboration with the software lead. James Gregory is also the backup hardware person behind James Donegan.

Wade Henderson, the team's only computer engineering senior, is the software lead for the project. He is responsible for software version management and organization, RTOS implementation, and peripheral drivers. Most of the other team members will have to coordinate their tasks with Wade because this project is mostly software. Wade is also the team lead backup, which means that he is team lead in the absence of Nathaniel.

During the first week, the team formed and exchanged phone numbers and email addresses in order to ensure contact with each member in the group outside of class. The team also formed a private Facebook™ group where they could post updates and links amongst the members as an alternative form of communication. Another online tool utilized was Google Drive, a free online data storage service. This tool enabled the team to share files from a central online point and keep file management more organized. Despite all the technology making contacting and file sharing easier, the team still needed a weekly time to meet and share project updates. From the start, the team decided to meet on the UCF campus every Tuesday and Thursday from 8:30-9:00AM during Senior Design I. During Senior Design II, the team met weekly during the weekends in the senior design lab in on weekends.

In order to help keep the software organized, the group used a GitHub account. This software allowed the group to keep track of revision management, perform source control, as well as merge code branches. This software was instrumental to the flight computer software development. It allowed the team to have multiple branches of code, while still being able to keep track of each one, and be able to commit changes and merge sections. Since the code went through several revisions before being finalized, having the old revisions easily tracked allowed the group to efficiently work through several coding phases.

Overall, the team was well organized into task groups that are able to tackle the problems and challenges associated with the project. The team's four members, each with their own diverse skill sets, have their own portions of the project that were integrated into the final project. The team had multiple forms of communication, file sharing, and meeting options that are able to keep them organized and our information synchronized. In the end, it is the team organization, diverse skills, and the ability for the team to efficiently communicate and share information that helped them be successful.

Appendices

Appendix A – References and Other Projects Researched

- 1.) Hayes, M. H. "9.4: Recursive Least Squares." *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons. 1996..
- 2.) Ziegler, J.G and Nichols, N. B. "pages 759-768." *Optimum settings for automatic controllers*. 1942.
- 3.) Franklin, G. F. et al. "3.3: PID Control." *Digital Control of Dynamic Systems*. California: Addison Welsey Longman, Inc. 1998.
- 4.) <http://eeecs.ucf.edu/seniordesign/fa2012sp2013/g04/Final.pdf>
- 5.) <http://eeecs.ucf.edu/seniordesign/fa2012sp2013/g17/Senior%20Design%202%20Report.pdf>

Appendix B – Copyright Permissions

We were not able to write Bosch an email, we had to use this online form from:

http://www.bosch-sensortec.com/en/homepage/contact_1/contact/formedit/contact_form

Country *

Telephone Number


Requested Information * ☒ Data sheet
☐ Prices
☐ Distributors
☐ Other

Product * ☐ 3-Axis Sensors
☐ 6-Axis Sensors
☐ 9-Axis Sensors
☒ Environmental Sensors
☐ Software

Your message to us

Subject

Message

Captcha 

Text

Verification

[Not readable?](#)

The actual text was:

Hello Bosch Sensortec,

I am a senior majoring in computer engineering at the University of Central Florida. My senior design team is currently writing our technical document for our quadcopter project using a BMP085 altimeter. We wanted to request permission to use the calculation of pressure and temperature figure on page 13 of the BMP085 altimeter document number: BST-BMP085-DS000-05 revision 1.2 our academic paper.

*Best regards,
Wade Henderson*

The response:



Saeed Afzal (BST/SNA) <Afzal.Saeed@us.bosch.com>
Mon 11/25/2013 8:54 AM

To: fotonphorces@knights.ucf.edu;

Action Items

Hi Wade,
Thank you for your interest in Bosch Sensortec products. Please go ahead and use the flow chart example on pg 13 in your technical paper. Also keep in mind that BMP085 is now EOL. The replacement device is BMP180 for your designs moving forward.

Thanks,
Afzal

Afzal Saeed
Senior Sales Manager NA
Bosch Sensortec
Email: afzal.saeed@bosch-sensortec.com
Cell: 847-867-3795
Visit us at: www.bosch-sensortec.com

Actual text:

*Hi Wade,
Thank you for your interest in Bosch Sensortec products. Please go ahead and use the flow chart example on pg 13 in your technical paper.
Also keep in mind that BMP085 is now EOL. The replacement device is BMP180 for your designs moving forward.*

*Thanks,
Afzal*

Afzal Saeed
Senior Sales Manager NA
Bosch Sensortec
Email: afzal.saeed@bosch-sensortec.com
Cell: 847-867-3795
Visit us at: www.bosch-sensortec.com

TI RTOS figure request:



fotonphorces
Mon 11/4/2013 11:52 AM
Sent Items

mark as unread

Hello Texas Instruments,

I am a senior majoring in computer engineering at the University of Central Florida. My senior design team is currently writing our technical document for our quadcopter project using a Tiva™ C series microcontroller. We are using the TI-RTOS to control it. We wanted to request permission to use the RTOS figure (User Application and board specifics) on page 56 of the TI RTOS 1.10 user's guide, Literature Number: SPRUHD4C in our academic paper.

Best regards,
Wade Henderson

Actual text:

Hello Texas Instruments,

I am a senior majoring in computer engineering at the University of Central Florida. My senior design team is currently writing our technical document for our quadcopter project using a Tiva™ C series microcontroller. We are using the TI-RTOS to control it. We wanted to request permission to use the RTOS figure (User Application and board specifics) on page 56 of the TI RTOS 1.10 user's guide, Literature Number: SPRUHD4C in our academic paper.

*Best regards,
Wade Henderson*

Response:

Permission to use figure from TI-RTOS user's guide



Bassuk, Larry <l-bassuk@ti.com>

Tue 11/26/2013 5:19 PM

To: fotonphorces <fotonphorces@knights.ucf.edu>;

Action Items

Thank you for your interest in Texas Instruments. We grant the permission you request in your email below.

On each copy, please provide the following credit:

Courtesy Texas Instruments

Regards,

Larry Bassuk
Deputy General Patent Counsel &
Copyright Counsel
Texas Instruments Incorporated
214-479-1152

Actual text:

Thank you for your interest in Texas Instruments. We grant the permission you request in your email below.

On each copy, please provide the following credit:

Courtesy Texas Instruments

Regards,

*Larry Bassuk
Deputy General Patent Counsel &
Copyright Counsel
Texas Instruments Incorporated
214-479-1152*

DTN figure request:



cain.n

Fri 11/22/2013 2:04 PM

Sent Items

mark as unread

To: demmer@cs.berkeley.edu;

Hello Michael Demmer,

I am working on an engineering senior project with a team of students at the University of Central Florida. We are using DTN2 in our project and need some background information on how DTN is implemented. We stumbled on your paper from 2004 called "Implementing Delay Tolerant Networking" and was wondering if we could have permission to use some information and diagrams from it. Specifically, we would like to use the block diagram in figure 3 on page 6 in our report.

Thanks!

Nathaniel

Actual Text:

Hello Michael Demmer,

I am working on an engineering senior project with a team of students at the University of Central Florida. We are using DTN2 in our project and need some background information on how DTN is implemented. We stumbled on your paper from 2004 called "Implementing Delay Tolerant Networking" and was wondering if we could have permission to use some information and diagrams from it. Specifically, we would like to use the block diagram in figure 3 on page 6 in our report.

Thanks!

Nathaniel

Response:



demmer@gmail.com on behalf of Michael Demmer <demmer@cs.berl

Fri 11/22/2013 2:54 PM

Sure - no problem.

Actual Text:

Sure-no problem.

Appendix C – Datasheets Referenced

- 1.) <http://www.ti.com/tool/ek-tm4c123gxl>
- 2.) <http://www.ti.com/product/tm4c123gh6pm>
- 3.) <http://www.raspberrypi.org/technical-help-and-resource-documents>
- 4.) http://www.hobbyking.com/hobbyking/store/_43709_Afro_ESC_20Amp_Multi_rotor_Motor_Speed_Controller_SimonK_Firmware_.html
- 5.) http://www.hobbyking.com/hobbyking/store/_25081_NTM_Prop_Drive_Series_28_30S_900kv_270w_short_shaft_version_.html
- 6.) http://www.hobbyking.com/hobbyking/store/_46308_8045_SF_Props_2pc_Standard_Rotation_2_pc_RH_Rotation_Red_USA_Warehouse_.html
- 7.) http://www.hobbyking.com/hobbyking/store/_24172_Q450_Glass_Fiber_Quadcopter_Frame_450mm.html
- 8.) http://www.hobbyking.com/hobbyking/store/_20791_Turnigy_nano_tech_5000mah_4S_45_90C_Lipo_Pack_USA_Warehouse_.html
- 9.) http://www.hobbyking.com/hobbyking/store/_8991_Turnigy_9X_9Ch_Transmitter_w_Module_8ch_Receiver_Mode_1_v2_Firmware_.html
- 10.) http://www.hobbyking.com/hobbyking/store/uh_viewitem.asp?idproduct=25483&aff=588847
- 11.) http://www.amazon.com/Raspberry-5MP-Camera-Board-Module/dp/B00E1GGE40/ref=wl_it_dp_o_pd_S_nC?ie=UTF8&colid=3AHXZEF7P8ZM&coliid=I2D9TFB0V615PM
- 12.) <https://www.sparkfun.com/products/10724>
- 13.) <https://www.sparkfun.com/products/11571>
- 14.) https://www.sparkfun.com/datasheets/Components/General/BMP085_Flyer_Rev.0.2_March2008.pdf