MicroCART 2016-2017

Final Report

Group: May1716

Team Members

Brendan Bartels – *Controls Software Key Concept Holder*
Kristopher Burney – *Ground Station Key Concept Holder*
Joseph Bush – *Quadcopter Software Key Concept Holder*
Jacob Drahos – *Team Webmaster*
Eric Middleton – *Hardware Guru*
Tara Mina – *Team Communications Leader*
Andrew Snawerdt – *Control Systems Key Concept Holder*
David Wehr – *Team Leader*

Advisors

Dr. Phillip Jones
Dr. Nicola Elia

Team Email: *may1716@iastate.edu*

Team Website: *http://may1716.sd.ece.iastate.edu*

Contents

1 Definition of Terms **5**

2 Introduction **6**

2.1 Project Statement 6

2.2 Purpose 6

2.3 Goals 7

3 Design and Implementation **8**

3.1 High Level Diagram 8

3.2 Quadcopter 8

3.2.1 Directed Graph Based Calculations 9

3.2.2 Application-Platform Separation 10

3.2.3 LiDAR Integration 12

3.2.4 Optical Flow 12

3.2.5 GPS 13

3.2.6 Communication 13

3.2.6.1 WiFi as Communication Protocol 13

3.2.6.2 Asynchronous UART Receiving 14

3.2.7 Hardware Improvements 14

3.2.8 Virtual Quadcopter 15

3.3 Ground Station 15

3.3.2 Backend 16

3.3.3 Front End 17

3.3.3.1 Command Line Interface (CLI) 17

3.3.3.1 Graphical User Interface (GUI) 18

3.4 Control Structure and Tools 20

3.4.1 Mathematical Model 20

3.4.1.1 Communication System 20

3.4.1.2 Control System 21

3.4.1.3 Actuation 21

3.4.1.4 Sensor System 22

3.4.2 Controller Design 23

4 Testing Process and Results **26**

4.1 Quadcopter Software 26

4.1.1 Unit Tests 26

4.1.2 Functional Tests with Virtual Quad 26

4.1.3 Dedicated Hardware Tests 27

4.1.4 Live Flight Tests 27

4.2 Control Structure and Tools 29
4.2.1 Mathematical Model Testing and Results 29
4.2.2 PID Controller Testing and Results 34
4.3 WiFi Bridge 35
4.4 Hardware Improvements 36

5 Conclusion 37

Appendices 38
Appendix A: How to Fly the Quadcopter 38
Setup Infrared Camera System 38
Setup Ground Station 38
Setup Transmitter 38
Setup Quadcopter 39
Start the Ground Station (CLI) 39
Start the Quad 40
Appendix B: Changes to Original Design Plan 41
Appendix C: Suggestions for Future Work 42

References 43

Figures

Figure 1: *MicroCART Quadcopter* **6**
Figure 2: *High-Level System Diagram* **8**
Figure 3: *Zynq-7000 SoC on Zybo Board* **9**
Figure 4: *Example Graph for Directed Graph-Based Calculations* **9**
Figure 5: *Graph Chain for Autonomous X Control* **10**
Figure 6: *Application Platform Interface Design* **11**
Figure 7: *Quadcopter Software Re-Architecture* **11**
Figure 8: *LiDAR Sensor* **12**
Figure 9: *Optical Flow Sensor* **12**
Figure 10: *WiFi Module* **13**
Figure 11: *Inherited Quadcopter Hardware State* **14**
Figure 12: *Virtual Quadcopter Design* **15**
Figure 13: *Backend Daemon* **16**
Figure 14: *Command Line Interface Block Diagram* **17**
Figure 15: *Timing Diagram of a Request to the Backend* **18**
Figure 16: *GUI Control Graph Tab* **19**
Figure 17: *GUI Navigation Tab* **19**
Figure 18: *High-Level System Block Diagram* **20**
Figure 19: *Communication System Block Diagram* **21**
Figure 20: *Actuation System Block Diagram* **22**
Figure 21: *Sensor System Block Diagram* **22**
Figure 22: *PID Controller Block Diagram* **23**
Figure 23: *Nested-Loop PID Architecture* **24**
Figure 24: *Plots of Nested PID Outputs from "simplePlots.m" Script* **28**
Figure 25: *Calculated and Experimental Duty Cycle vs. Rotor Speed* **29**
Figure 26: *Error Between Predicted and Experimental Rotor Speed* **30**
Figure 27: *X, Y, Z Position Calculated by Simulink Model* **32**
Figure 28: *X, Y, Z Velocities Calculated by Simulink Model* **32**
Figure 29: *Complementary Filter Output of Model vs. Logged Flight Data* **33**
Figure 30: *X, Y, Z Position of Model vs. Logged Flight Data* **34**
Figure 31: *TCP vs. BlueTooth Latency Distribution* **36**
Figure 32: *Transmitter Connection Light* **39**
Figure 33: *RC Controller Operation* **40**
Figure 34: *Original Control Structure* **41**

Tables

Table 1: *Definition of Terms* **5**

Table 2: *PID Architecture Variable Definitions* **24**

Table 3: *Rise Time and Error Margin of Position Controllers* **29**

Table 4: *Rotor Speed Equation Variable Definitions* **35**

1 Definition of Terms

Below are some of the terms we use regularly in this design document:

Term	Description
CLI	Command Line Interface, will be used to take in user input to get data from the quadcopter and give it flight commands
ESC	Electronic Speed Controller, a component between the Zybo board and the motor, which takes in a PWM signal from the board and converts this to an amplified current to control the speed of the motor
GUI	Graphical User Interface, will be used to take in user input to get data from the quadcopter and give it flight commands
LiDAR	Light Detection and Ranging, a laser used to detect and measure distances (ranging)
MicroCART	Microprocessor-Controlled Aerial Robotics Team, our senior design team
PCB	Printed Circuit Board, will be designed to include a battery voltage regulator and power distributor for improved power management on the quadcopter
PID	Proportional-Integral-Derivative, a commonly used type of feedback controller
PWM	Pulse-Width-Modulation, a digital signal with varying duty cycle but constant period, which is received by each of the ESCs to control the speed of each motor
CI	Continuous Integration, a software design process where every change to the software triggers an automatic fresh compilation and running of tests

Table 1: Definition of Terms

2 Introduction

The MicroCART (Micro-processor Controlled Aerial Robotics Team) senior design project has been passed down from team to team since 1998, developing a quadcopter as a research platform. The MicroCART quadcopter (**Figure 1**) has been flying in the Distributed Sensing and Decision Making Laboratory, using the twelve-camera infrared tracking system for navigation.

Figure 1: MicroCART Quadcopter

2.1 Project Statement

We intend to create a modular platform for research in controls and embedded systems. In addition, we intend to advance the abilities of the existing quadcopter platform, including flying autonomously to a sequence of user-specified waypoints, as well as flying independently from the infrared tracking camera system.

2.2 Purpose

By creating a modular platform, any controls student should be able to design their own controller and test it with our system. Additionally, our project should represent and demonstrate the capabilities of the Electrical and Computer Engineering Department, in order to excite new students, offer a hands-on platform to learn about control systems, and better represent the talents of our department to visitors, interested students, and faculty members. In this aspect, our system will likely be utilized in the controls

systems course, EE 476, to help students develop a more intuitive understanding of PID controllers and provide a platform for students to test their controller designs.

2.3 Goals

Our advisors, Dr. Jones and Dr. Elia, had some high-level goals for what they wanted from our project, which can be summarized by the following points:

- Build a modular system for the quadcopter, so that each component can be easily removed and replaced by another of the same functionality, without breaking the operation of the entire system
- Design a control system for the quadcopter from a mathematical model representation of the quadcopter system, rather than from iterative testing procedures as done by previous teams
- Decrease the communication latency between the quadcopter and the ground station to be less than 10 milliseconds on average, to increase the stability of the system overall, which will most likely require WiFi communication rather than the current Bluetooth communication system
- Ensure that the communication system and controls system work together robustly, such that if any data packets are dropped, the quadcopter does not lose control and potentially damage itself
- Create a user-friendly GUI with an attractive multi-panel layout and click navigation features for the user to easily specify waypoints for the quadcopter to travel between
- Throughout the development process, create detailed and user-friendly documentation and tutorials for future MicroCART teams to follow and learn from, including video tutorials as well, and keeping the two-decade-long Wiki page up-to-date
- Advance the autonomous flight capability to support waypoint navigation
- Enable the quadcopter to fly without the ground station
- Enable the quadcopter to fly outdoors by integrating appropriate sensors onto the quadcopter

For autonomous flight, our goal was to develop a mathematical model of the quadrotor, and from this model, design several PID controllers for each direction of movement. We also wanted a user-friendly graphical interface with an attractive multi-panel layout and click navigation features for the user to easily specify waypoints for the quadcopter to travel between. We also hoped to develop the ability to fly without the infrared camera tracking system in the lab.

In terms of improving the modularity of the system, our goal was to provide a clear separation between controllers, application logic, and hardware implementation details. This was obtained through the use of a node-based structure for controller computations and defining abstract interfaces that interact with the hardware.

To generally improve the quality and stability of our system, we wanted to ensure that the communication system and controls system work together robustly, such as by preventing data packets from being dropped, which could cause the quadcopter to lose control and potentially damage itself. Additionally, we planned to improve the communication latency between the quadcopter and the ground station. Finally, we wanted to improve the state of the hardware, with better wire connections and removing unnecessary hardware components.

3 Design and Implementation

The design of our quadcopter system can be divided into three major components: the development of the quadcopter software, the ground station, and the PID controllers.

3.1 High Level Diagram

The system is primarily composed of 4 subsystems, the quadcopter itself, the ground station, the tracking system in our lab, and a RC controller. The quadcopter has been built from the ground up by previous teams. The ground station is simply a computer, running the ground station software. The tracking system in our lab is composed of 12 infrared cameras that allow us to determine the real-time precise location of the quadcopter. This position data is forwarded to the quad via the ground station, where it can be used in the control algorithms onboard the quadcopter. We can also manually control the quadcopter explicitly, using the RC controller. The ground station also features a GUI to control and monitor the quadcopter, such as giving the quadcopter waypoints to follow autonomously.

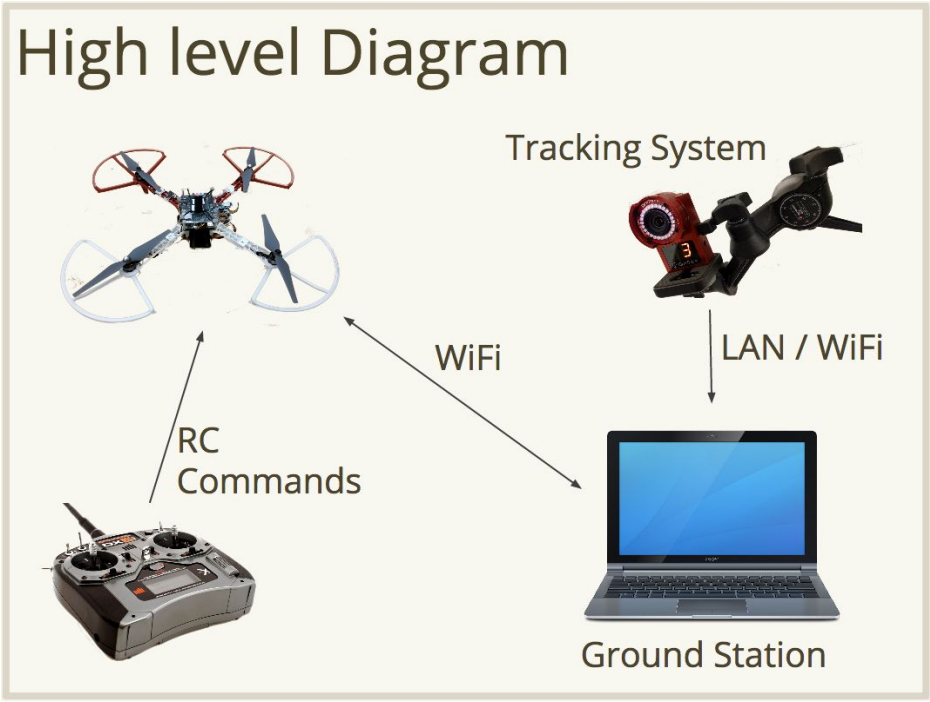

Figure 2: High-Level System Diagram

3.2 Quadcopter

We inherited both working hardware and software for the quadcopter from previous teams, which was used and modified for our purposes. After our contributions, the software underwent a significant architecture redesign and a re-implementation of the control algorithm. We also added the necessary software in order to support new devices we added to the quad, namely LiDAR, Optical Flow, and GPS.

The Zynq-7000 SoC, as shown in **Figure 3** below, runs the software on the quadcopter.

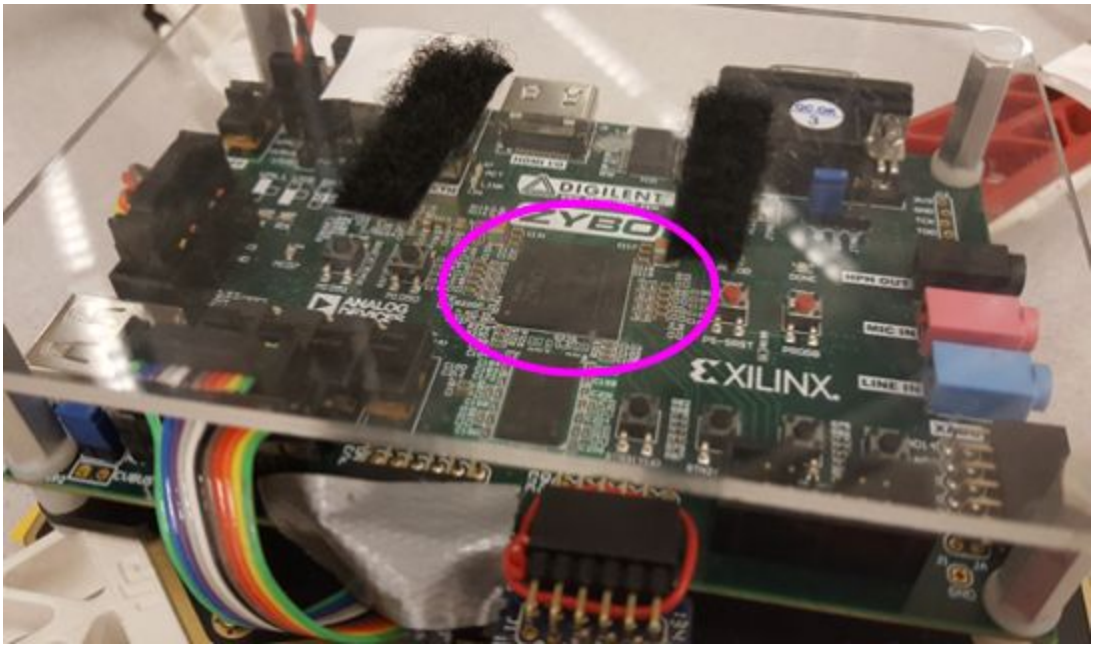

Figure 3: Zynq-7000 SoC on Zybo Board

3.2.1 Directed Graph Based Calculations

The original control algorithm was implemented in a traditional line-by-line software style, but this design presents a number of issues for a research platform:

- It inhibits fast and iterative changes to the control algorithm, since any change would require a recompilation of the software and subsequent transfer of the new code onto the quad.
- Adding new control algorithms and filtering techniques by a student or researcher would require detailed knowledge of the quadcopter's design and structure.

In order to overcome these issues, we re-implemented the control algorithm using directed graph based calculations.

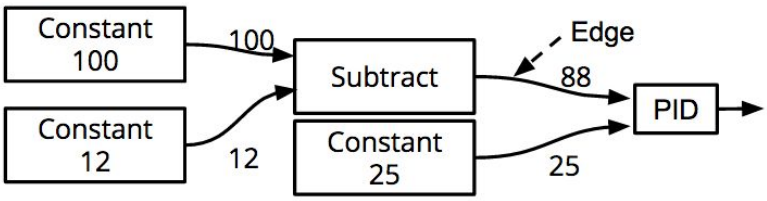

Figure 4: Example Graph for Directed Graph-Based Calculations

In a traditional graph, nodes typically represent some sort of data. However, in this type of design, each node represents a function, while the edges represent some value. Computation of the entire graph requires the use of a depth-first graph search algorithm, which proceeds as follows:

1. Begin search at the desired outputs; in our case, this would be the signal mixer, which computes the final motor values.
2. Recursively visit each parent, and once each parent has been visited, calculate the function for this node, using the outputs of its parents. Once the function has been computed, store the output and return, allowing its child to continue computation.

A typical computation graph is visualized in **Figure 5**. In it, the outputs of each node are visualized on the graph edges, which feed as inputs to other nodes.

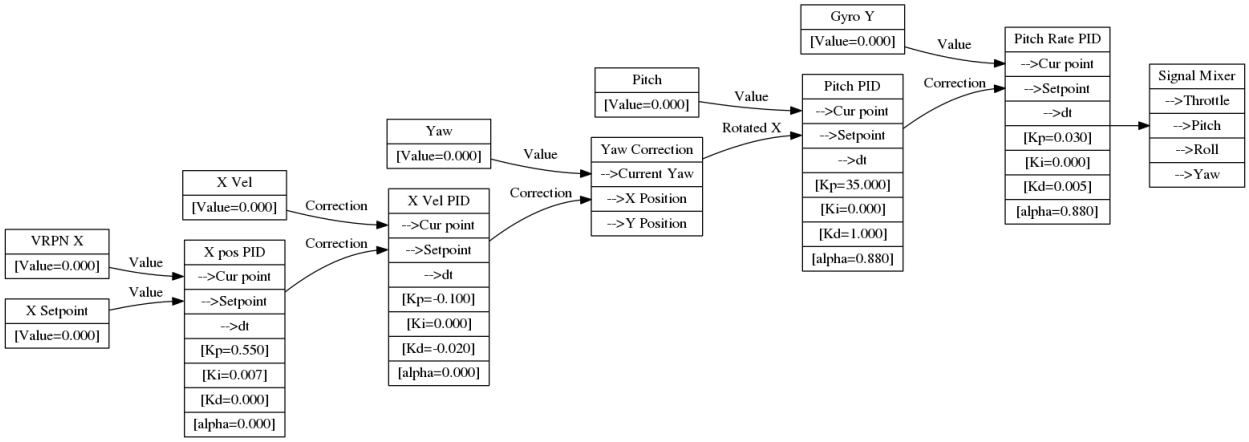

Figure 5: Graph Chain for Autonomous X Control

For our implementation, we have "value" nodes that represent constants and inputs to the graph. Inputs are either sensors, e.g. accelerometer, or user inputs, e.g. waypoint positions. In **Figure 5**, the nodes for autonomous X control are visualized. Data flows from left to right, with sensors such as VRPN data and gyroscope feeding into PID blocks that feed forward towards the signal mixer.

In addition to PID computation, sensor processing and conversion can also be implemented in the graph. Code for our filters, including the complementary filter, low-pass filter, yaw correction, and integration, can be implemented as reusable blocks and placed into the graph.

Structuring the graph using directed graph calculations allows for dynamic changes to the structure, automatic visualization of the controller, and ease of integrating new code, such as controllers and filters, that were developed externally of the MicroCART platform.

3.2.2 Application-Platform Separation

The quad software we inherited from previous teams was designed with tight coupling between the application layer and the platform layer. We consider the application layer to be any code that only requires standard C libraries and the platform layer to be code that requires platform specific libraries and functions. In our case, the platform of consideration is the Zynq processor and its associated libraries.

This tight coupling prevents effective testing of the quad software, since the platform-specific code dictates that the software can only be compiled for the processor on the Zybo (and hence, only able to be tested on the Zybo board). In order to be able to run unit tests on our laptops and on a continuous

integration server, the code must be able to compile in a Unix environment. Hence, we redesigned the quad software architecture in order to introduce a distinction between the application layer and the platform layer. We accomplished this division by implementing a driver layer between the application and our target platforms.

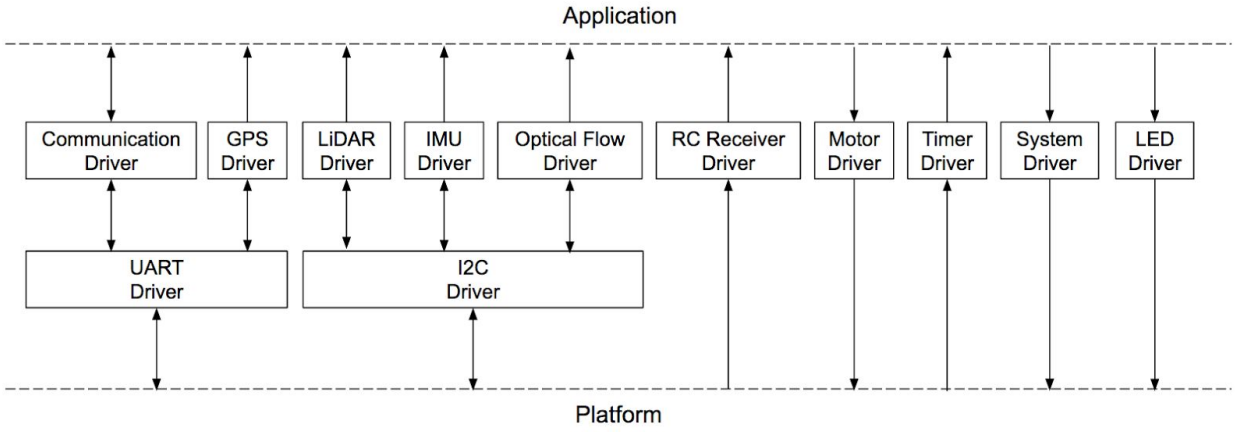

Figure 6: Application Platform Interface Design

Each driver could be consider an interface in an object-oriented sense, containing a number of unimplemented functions that represent certain hardware features and functionality provided by the platform. These include turning on LEDs, receiving data from I2C devices, outputting PWM signals for the motors, and other functionality provided by the platform.

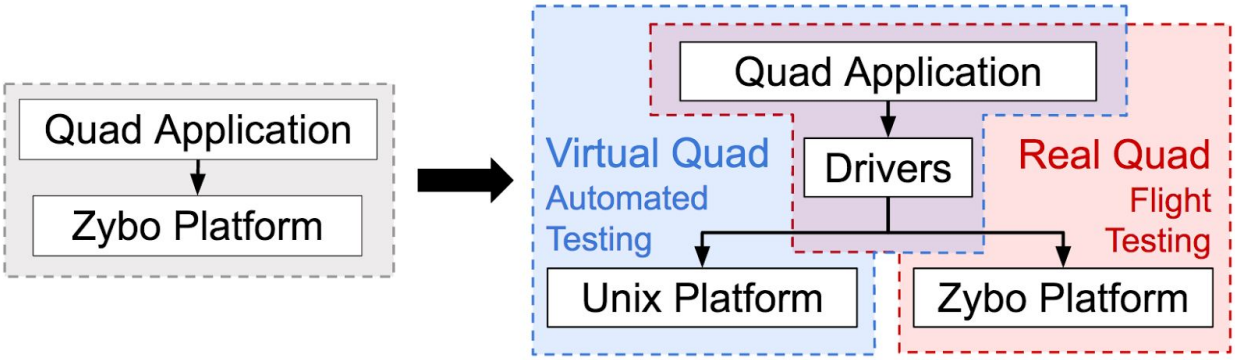

Figure 7: Quadcopter Software Re-Architecture

When selecting a particular platform (Zybo or Unix), these interface-like drivers are populated with functions appropriate for the platform of choice. In consequence, we can still compile our code for the Zybo by implementing these functions in order to interact with the Zybo board, but we can also implement these functions with mock behavior that uses no Zybo-specific libraries, which can compile and run in a Unix environment. We call this quadcopter with mocked functionality our virtual quadcopter, and it enables many forms of testing at the application layer.

3.2.3 LiDAR Integration

To achieve height determination agnostic of the camera system, we used a LiDAR sensor. The LiDAR sensor provides laser-based proximity sensing with an accuracy of ± 2.5cm, with 1cm precision. Originally, we planned to integrate this sensor onto the existing I2C bus on the Zybo board already configured by previous teams, but we encountered clock signal issues that we could not resolve. To maintain the reliability of the IMU sensor, a new I2C bus was created and on which this LiDAR sensor was added, and then an appropriate driver was implemented for this device to be used with the application.

Figure 8: LiDAR Sensor

3.2.4 Optical Flow

To achieve x-y coordinate determination agnostic of the camera system, we used an optical flow sensor. This sensor uses image processing of the ground below the quadcopter in order to calculate x-y velocity. We then integrate this velocity data within the quad application in order to determine relative position. This optical flow sensor was integrated onto the I2C bus with the IMU sensor, and a driver was implemented to be used within the quad software.

The specific sensor we used was the px4flow, a community-developed sensor designed for the PIXHAWK flight controller that has fully open-source hardware and software. We chose this particular sensor primarily because it was purpose-built for positioning on autonomous flying vehicles. Unfortunately, the software had several bugs (primarily related to I2C communication) that we had to find and fix to effectively use the sensor. We also found that the on-board SONAR sensor (which was used in the calculation of ground velocity) was inaccurate, so we chose to do the final conversion of pixel flow to ground velocities in our application using the LiDAR sensor.

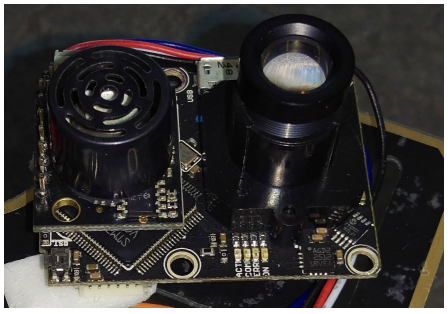

Figure 9: Optical Flow Sensor

3.2.5 GPS

While optical flow provides relatively accurate position within a short time span, absolute position determination over a long period of time requires the use of a GPS sensor. The GPS sensor provides position data accurate to a 1-meter resolution. We characterized the GPS module, and created a stub driver within the quad software, but full integration of the GPS module was not completed.

3.2.6 Communication

3.2.6.1 WiFi as Communication Protocol

The inherited system was configured to use Bluetooth as the communication method between the ground station and the quadcopter. Bluetooth has high latency, which presents difficulties for autonomous flight and running control algorithms on the base station. To improve this, we replaced Bluetooth with WiFi, using an ESP8266 microcontroller with integrated WiFi.

Figure 10: WiFi Module

The WiFi module acts as an access point, so the ground station can connect to the WiFi network hosted on the module without any extra configuration. To program the ESP8266, we used the esp-open-sdk, which provides a set of drivers and a simple TCP/IP stack. While the SDK supports both TCP and UDP protocols, we chose to use the TCP protocol because it more closely matches the reliability guarantees of the bluetooth system that was used the previous year.

To simplify the integration of WiFi with the rest of the quad application, the module simply forwards data from UART to WiFI, and forwards the data received over WiFi to UART. This way, no changes are required in the quadcopter software to use WiFi as the communication method. The module has no concept of the data protocol used between the quadcopter and ground station, which allows us to make changes to the communication protocol without modifying the firmware on the module.

3.2.6.2 Asynchronous UART Receiving

The WiFi bridge module passes data received over TCP to serial. In the previous system, reading from the UART was implemented as a blocking read once per main loop iteration. Therefore, if more data than the UART FIFO could hold (64 bytes) was received before the next read, data would be dropped, corrupting the packet, and potentially crashing the quadcopter.

To overcome these issues, we overhauled the communication system to improve robustness and throughput. The major component of this was switching from blocking UART reads to asynchronous reads. This involved implementing UART interrupts that would copy data out of the FIFO and into a queue, implemented as a circular buffer. Once per main loop, the circular buffer is processed and any complete packets are parsed and their callbacks executed.

We additionally updated the packet parsing and processing to gracefully handle corrupt data and eliminated memory leaks we inherited. This eliminated all communication-related crashes and increased our effective throughput from 12.8 kbps to 921 kbps.

3.2.7 Hardware Improvements

Because this project has been passed down over many semesters with the focus of each team being to add functionality to the quadcopter, the actual system hardware was in a non-ideal state. Wire connections were not very secure, some wire connections were being held together with tape, and some plugs were very tight. Along with the rewiring of the quadcopter in general, we redesigned the power distribution on the quadcopter system. The old system utilizes 4 AA batteries to power the Zybo board and sensors, which is unnecessary weight. These batteries also could not be fully charged as 4 AA batteries when fully charged is about 6 volts, when the board is powered off of 5 volts.

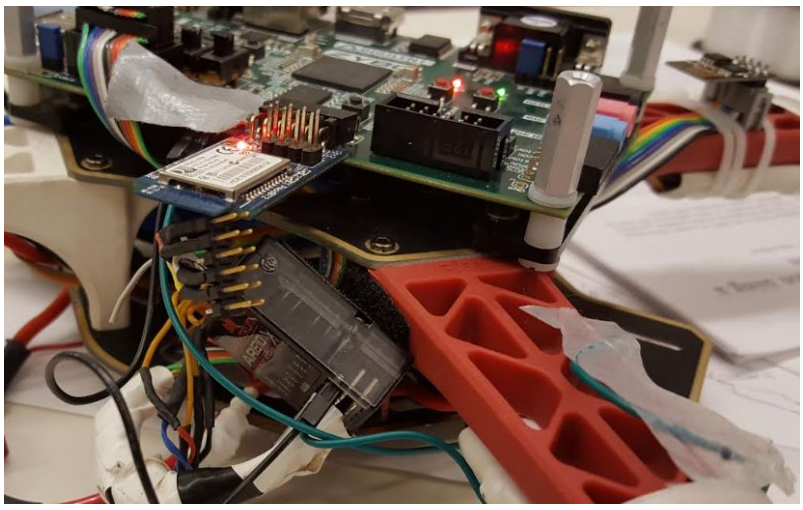

Figure 11: Inherited Quadcopter Hardware State

Overall, we made the following improvements:

- Replaced all tape-maintained wire connections with secure wire connections

- Rewired the I2C bus
- Regulate the LiPO battery to power the Zybo board using a voltage regulator:
    - From 11.1 volts to 5 volts
    - Up to 3 amperes
- Added a battery monitor to the quadcopter to prevent over-discharge

3.2.8 Virtual Quadcopter

Because the quad software was redesigned with an interface layer in order to separate the application layer from the platform layer, we were able to implement the driver interfaces with mock functionality in a Unix environment. For instance, instead of turning on a physical LED, we can print to the screen "LED turned on". This opens a wide range of behavioral testing at the application layer, where we can input certain values to the application and check if appropriate outputs are produced. We call this program our "virtual quadcopter."

The virtual quad runs as a process in a Unix environment. Its drivers are continuously reading values from a Unix shared memory space or FIFO, just as the real quad continuously read from physical sensors. We also direct all output to the shared memory space in order to observe the current output value. To interact with the virtual quad, one would need to either read from or write to Unix FIFOs or use the get/set commands of the virtual quad CLI. For instance, to emulate a quad command typically sent over WiFi, the same message can be written to the TX FIFO, and the virtual quad will receive and parse the message. To examine the values of the motors, the user can use the CLI, such as "get motor1" and the current value of motor 1 will be printed to the screen.

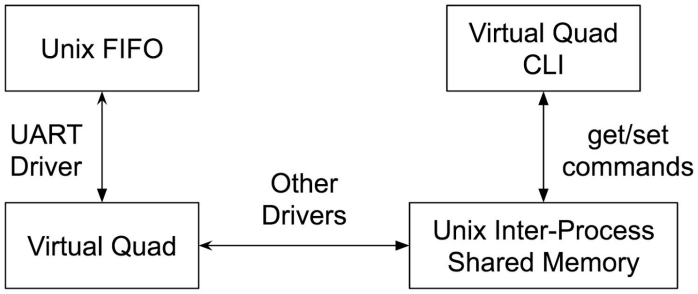

Figure 12: Virtual Quadcopter Design

3.3 Ground Station

The ground station software has two main components:

- Backend Component
- Frontend Component

The backend communicates directly with the VRPN system and the quadcopter. It also allows frontend clients to connect it to it via a socket interface, which allows the clients to call functionality available in the backend, and the backend to send updates to the frontend.

The frontend can be a simple command-line interface, or a more sophisticated graphical user interface.

3.3.2 Backend

The backend is designed to be a modular piece of the ground station. Any front end, whether it be a command line interface or a graphical user interface, connects to the back end via a socket connection. This backend accepts input through this socket and forwards it through another socket connected to the quadcopter.

Main Backend features:

- Complete connection to quadcopter through a back end
- Server-Client relationship between back and and front end
- Command pass through from front end to quadcopter

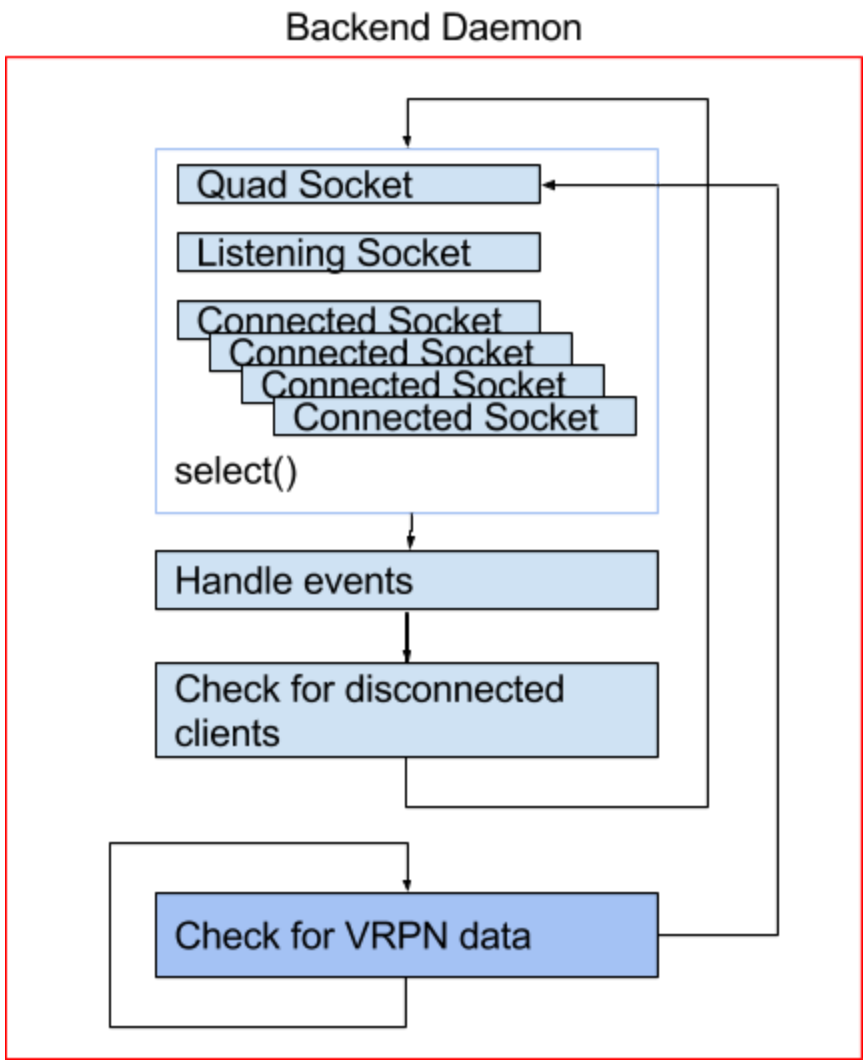

Figure 13: Backend Daemon

The backend daemon runs and manages connections with the quad. Communication with the quad uses a low-overhead binary protocol. It also connects to the tracking system and forwards tracking information to the quad. Frontends, CLI or GUI, connect to the backend and issue commands in an easy-to-parse, human readable ASCII protocol. These commands manipulate the state of the quad.

The backend consists of a headless daemon running at all times. This daemon will manage connection lifecycle with the quad and tracking system, as well as forward tracker data to the quad. The daemon provides a Unix-domain socket for clients (front-ends) to connect and issue commands.

3.3.3 Front End

Front-ends may exist in several forms. The first front-end is a simple non-interactive command line control program. This program, supporting several subcommands, is able to manipulate the state of the quad (active, manual, automatic, set coordinates, command waypoints), retrieve state, and download logs. This command runs, issues the command to the backend, and returns any information before exiting. The source code for this utility is structured to lend itself to reuse in later front-end implementations: the connection to the backend as well as the actual functionality to issue commands is isolated from the argument parsing logic.

The backend daemon allows one or many front-ends to operate simultaneously, and reusable command and communication code will make the creation of further front-ends a relatively trivial matter of acquiring user input, then calling the proper function to perform the desired operation. UX development work was completely isolated from the business logic of controlling the quad.

3.3.3.1 Command Line Interface (CLI)

A command line interface (CLI) provides the lowest level of front end ground station control. Users are able to set and request any relevant variables such as PID constants and pitch, roll and yaw set points. This low level bare bones control will allow for a fast testing environment as well as the ability to quickly check our understanding of the system as a whole. The CLI will connect via a socket to the modular back end below it.

The following command abilities for the user are implemented into the command line interface:

- Display the menu of commands available to the user
- Get current quadcopter information
- Set controller graph parameters
- Modify controller graph structure

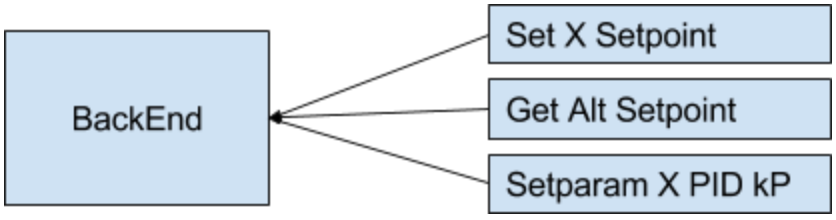

Figure 14: Command Line Interface Block Diagram

The design of the CLI is to create a Client-Server relationship with the BackEnd. The CLI is split up into separate programs. These programs will either create a connection and perform the task, or create a connection and hold it until the user is done with the task. This will allow tab completion in a shell, bash scripting to allow test scripts as well as a history of commands with BASH_HISTORY.

3.3.3.1 Graphical User Interface (GUI)

The GUI provides a convenient interface for common tasks and workflows. It is designed to support common tasks, including test flights of pre-programmed routes, as well as tuning or adjusting the control parameters. Because of the Client-Server model used with the CLI, the GUI is capable of coexisting with the CLI and being used simultaneously.

The GUI is written in C++, using the Qt application toolkit. The Signals and Slots facility of Qt is used to provide asynchronous access to the backend and avoid blocking the UI thread's responsiveness while a potentially long operation is occurring, such as a query to the quad. This also simplifies code reuse, as the library functions written for use with the CLI are all blocking. This is implemented by the creation of a "worker object", which lives in a separate thread. Actions that require backend queries are performed by posting a request signal to the worker object. The action is then performed and processed by the worker object (in a separate thread), and the completed result is posted back to the UI thread via another signal.

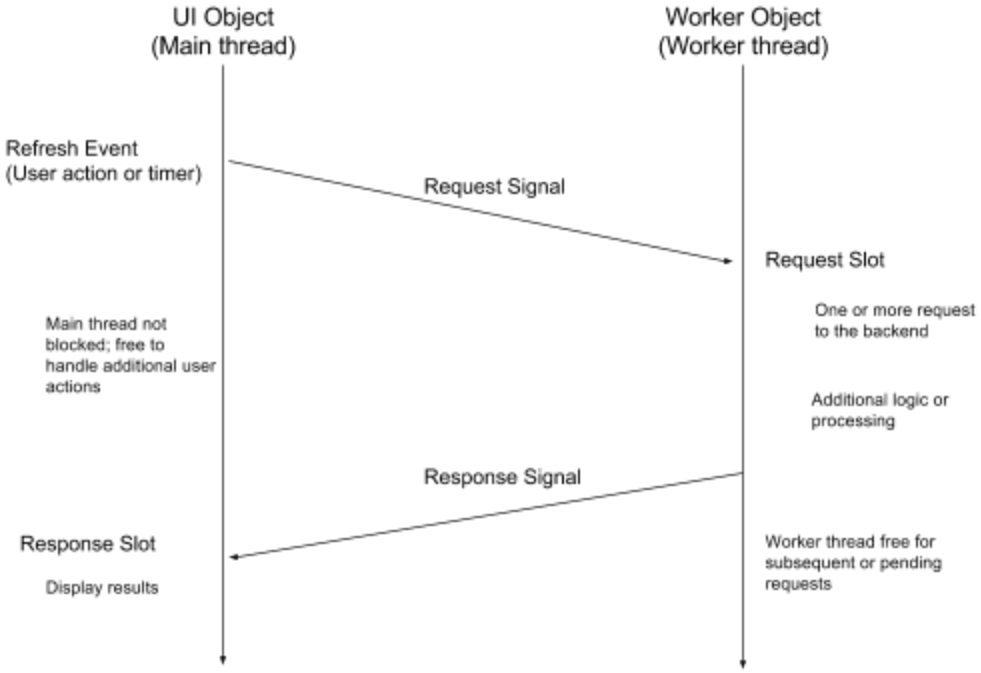

Figure 15: Timing Diagram of a Request to the Backend

The GUI workflow is separated into three tabs. The first tab is used to establish a connection to the backend. The second tab displays the control graph (obtained from the quadcopter) and can adjust the parameter values of any blocks in the control graph via a number of selection boxes. This tab also contains selectors for a number of important blocks used by other GUI functionality.

Because the control graph is dynamic, the GUI must be told which blocks define certain constants. For example, the X, Y, and Z setpoints can be adjusted from the Navigation tab. In order for that functionality to work, the nodes for those setpoint constants must be set on the Control Graph tab.

The Navigation tab displays the current position and attitude of the quadcopter, which may be obtained either from the tracking system (VRPN camera system) or queried from the quadcopter itself. Setpoints can be sent to the quadcopter either individually, or as a 3-tuple (X, Y, and Z/Altitude). Setpoint tuples can be saved in a list of waypoints, which can be modified-in-place, exported, and imported. The current position of the quadcopter can be saved as a setpoint or appended to the waypoint list.

Furthermore, Auto-navigation can be enabled with a configurable delay and distance threshold. When enabled, this feature will command the quadcopter to the next waypoint once it has reached the current setpoint. Additionally, to make the GUI more user-friendly, the Navigation tab provides a sprite animation of the current position and attitude of the quadcopter.

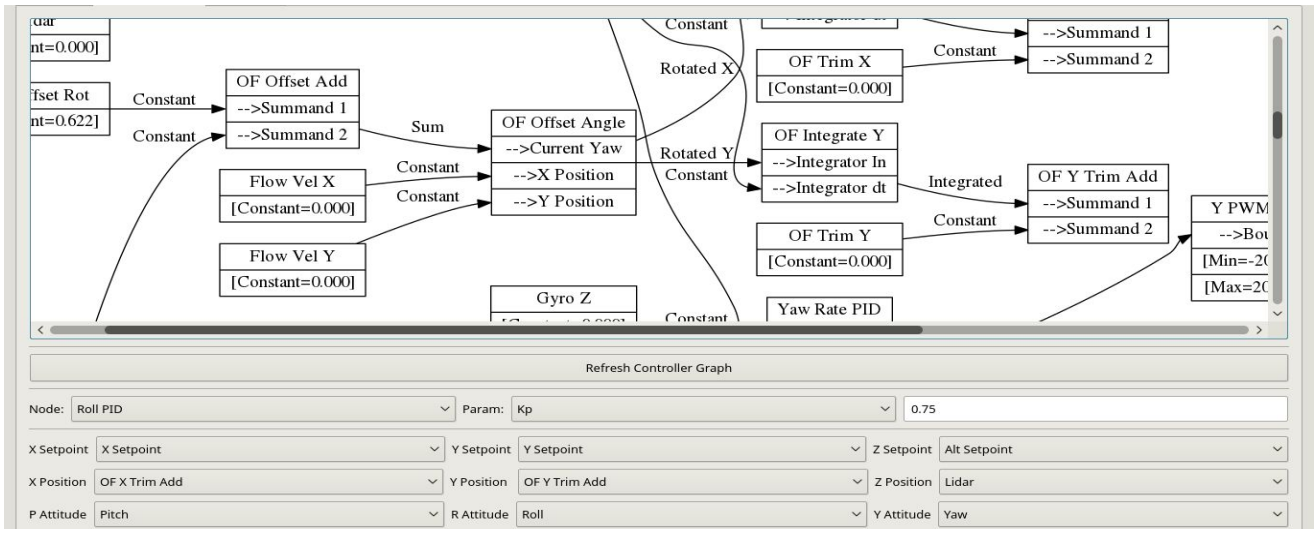

Figure 16: GUI Control Graph Tab

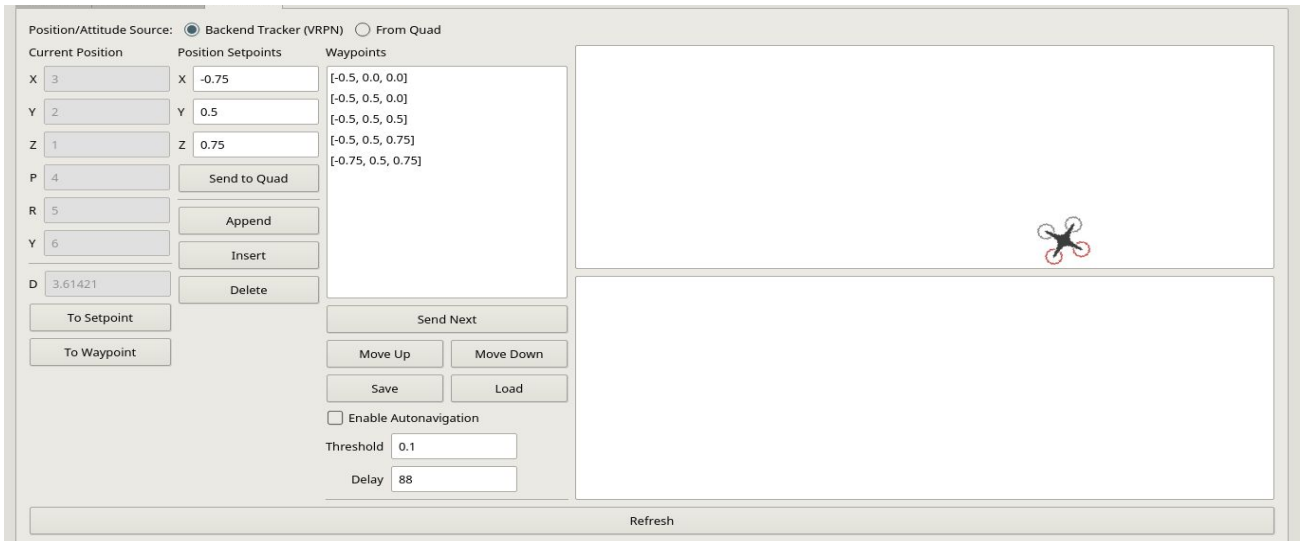

Figure 17: GUI Navigation Tab

3.4 Control Structure and Tools

3.4.1 Mathematical Model

Historically, the coefficients of the PID controller have been determined through iterative optimization. Since this approach has a couple of downsides, primarily with regards to scalability, our team's approach was to derive a mathematical model representing our entire system. To create this model, we did the following:

- By following the generic quadcopter model described and characterized in [5].
    - Performed system identification - measure of quadcopter parameters, including:
        - Moment of inertia about X, Y, and Z axes
        - Thrust and drag constants of quadcopter
        - Motor parameters (i.e. motor winding resistance)
        - ESC parameters: (i.e. motor “turn on” percent duty-cycle, maximum duty-cycle)
        - Data communication parameters: (i.e. round-trip latency for camera data)
    - Designed our model in Simulink using these parameters and the model in [5]
        - Includes 4 main blocks: actuator, sensor, communications, and controller
        - Implements calculations and equations in [5] for a physics model of the quadcopter
        - Characterized sensor noise
        - Followed and verified code structure, create a data flow diagram, and implement this logic in the controller and sensor block

The overall system is composed of four main components including the communication system, control system, actuation, and sensor system as shown below in **Figure 18**.

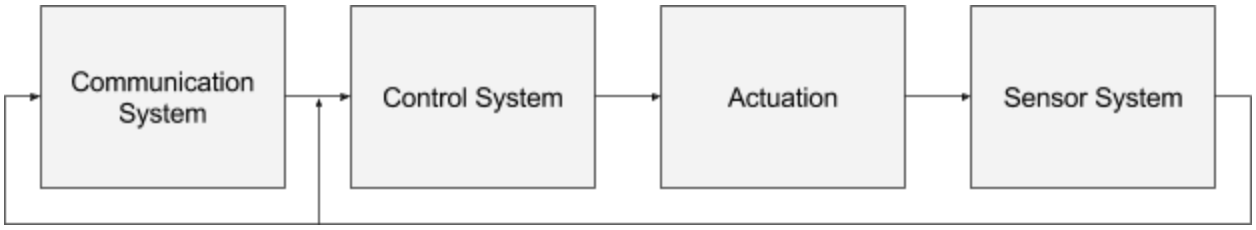

Figure 18: High-Level System Block Diagram

3.4.1.1 Communication System

The input to the control system from the communication system is dependant upon indoor or outdoor flight. When flying in the distributed autonomous and networked control lab, the x, y, z position of the quadcopter in space is determined from an OptiTrack camera system. This information must then be passed to the quadcopter through the ground station. However, during outside flight, the x, y position are determined from an onboard Optical Flow module, and the z position is determined with a LiDAR solution. This Optical Flow module and LiDAR solution will be a part of the sensor subsystem described in Section 3.2.3 and 3.2.4, however the model does not currently have these implemented.

Additionally, the communication system will also provide any user input from the command line interface (CLI/GUI), or controller during manual flight. This overall communication process is represented below in **Figure 19**.

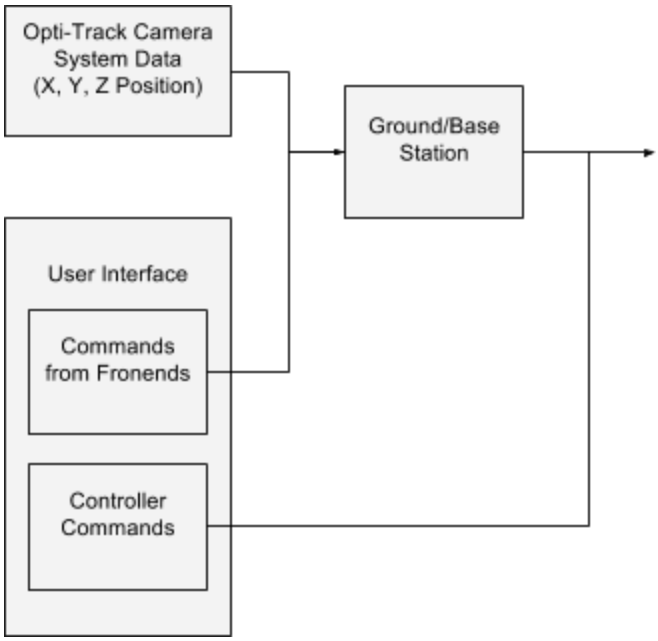

Figure 19: Communication System Block Diagram

3.4.1.2 Control System

The quadcopter is stabilized by multiple nested proportional-integral-derivative (PID) controllers. This will be discussed more in the subsection 3.6.2 Controller Design.

3.4.1.3 Actuation

The actuation, or mechanical movement of the quad occurs through the driving of each motor/rotor combination. As stated previously the output from the control system provides ESC motor commands, these commands are used by the ESC to coordinate what voltage to apply to each motor, represented as the vector V in **Figure 20**. The ESCs themselves are powered from a 11.4V LiPO battery (nominal voltage). From there the we can determine the actual angular velocity and acceleration of each rotor defined below:

$$M = \begin{bmatrix} \omega \\ \alpha \end{bmatrix}$$

where ω and α represent the angular velocity and acceleration respectively. From this we were able to derive the overall block diagram for the actuation of the quad.

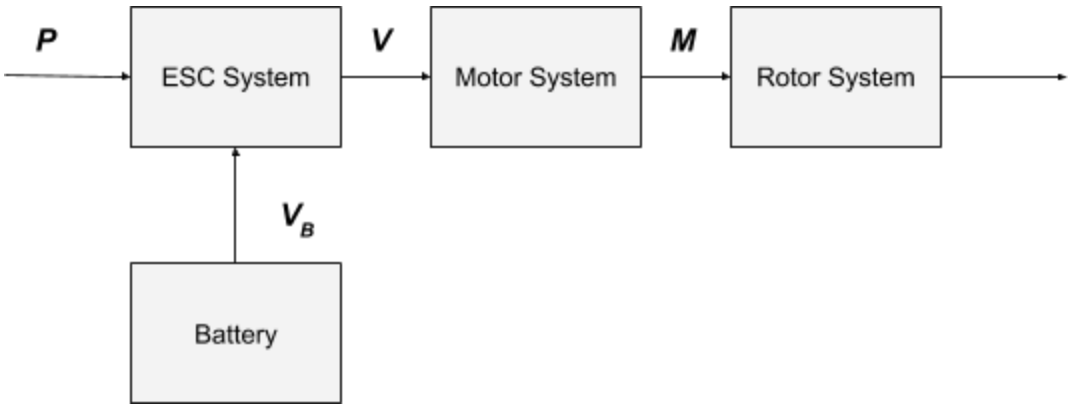

Figure 20: Actuation System Block Diagram

3.4.1.4 Sensor System

The sensor system is composed of the MPU-9150 IMU (Inertial Measurement Unit) which provides gyroscope and accelerometer data and is also an input to the control system. Alongside this, during outside flight the OptiTrack camera system is replaced with a dedicated Optical Flow module and LiDAR solution for determining the x, y, z position of the quadcopter. Data from the gyroscope and accelerometer can be used to find the yaw, pitch, roll angles, and angular velocities. With this and either the OptiTrack camera system or GPS module and LIDAR solution, we are able to provide all the required inputs to the control system.

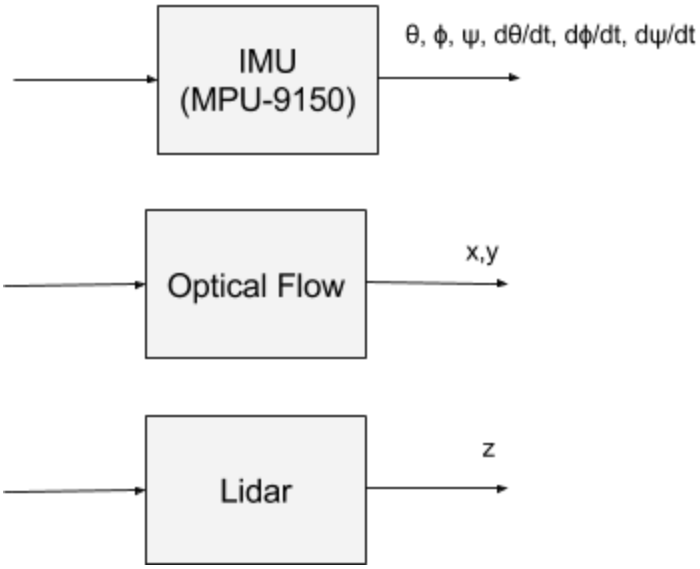

Figure 21: Sensor System Block Diagram

We have recently been looking into representing these systems in our Simulink model of our quadcopter in the best possible way. Currently our software code that takes the readings from the IMU to determine the pitch and roll angle of the position of the quadcopter is based on the a few equations. And though this makes intuitive sense, since we were able to rederive these equations using basic trigonometry, our graduate student advisors, Matt and Ian, have been explaining to us that some researchers may have

developed a better way to represent these angles of the quadcopter [2]. We will look into this to see if this is indeed a better method that will be worth switching to depending on how much more accurate it is as well as how much more effort it requires and complication it adds to the project.

3.4.2 Controller Design

The quadcopter is stabilized by multiple proportional-integral-derivative (PID) controllers.

The control system inputs come from both the communication system and sensor system. The entire control system is composed of multiple nested PID controllers. The composition of a PID controller in a feedback loop is shown below in **Figure 22**, where r(t) is the set point value, and y(t) is the measured output. The primary components of the PID controller include the K_p , K_i , and K_d terms, which denote the coefficients for proportional, integral, and derivative terms, respectively. These coefficients will be determined using our mathematical model of the quadcopter.

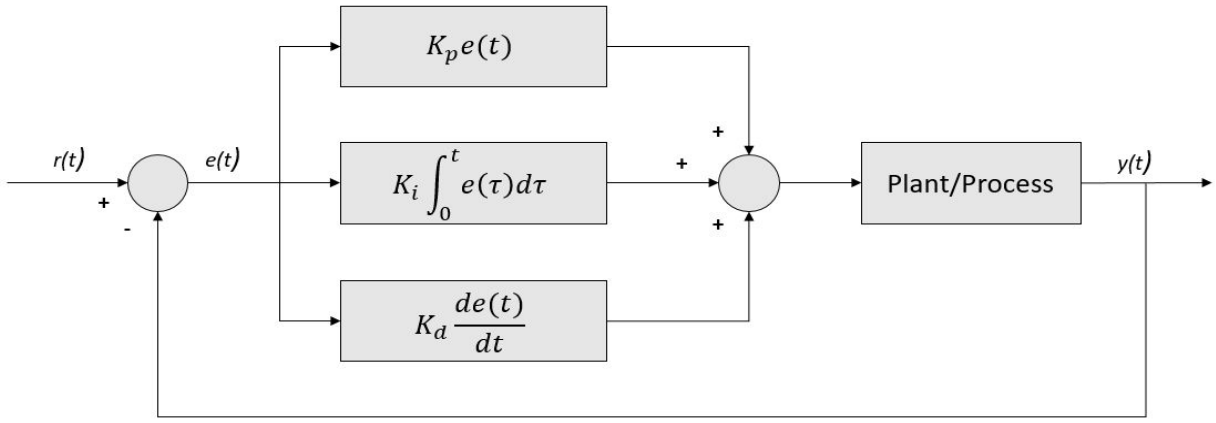

Figure 22: PID Controller Block Diagram

The overall control system is basically four controllers working in parallel: the height controller (z-axis), the longitudinal controller (y-axis), the lateral controller (x-axis), and the yaw controller. The latter three controllers have nested PID controllers embedded, as shown below in **Figure 23**.

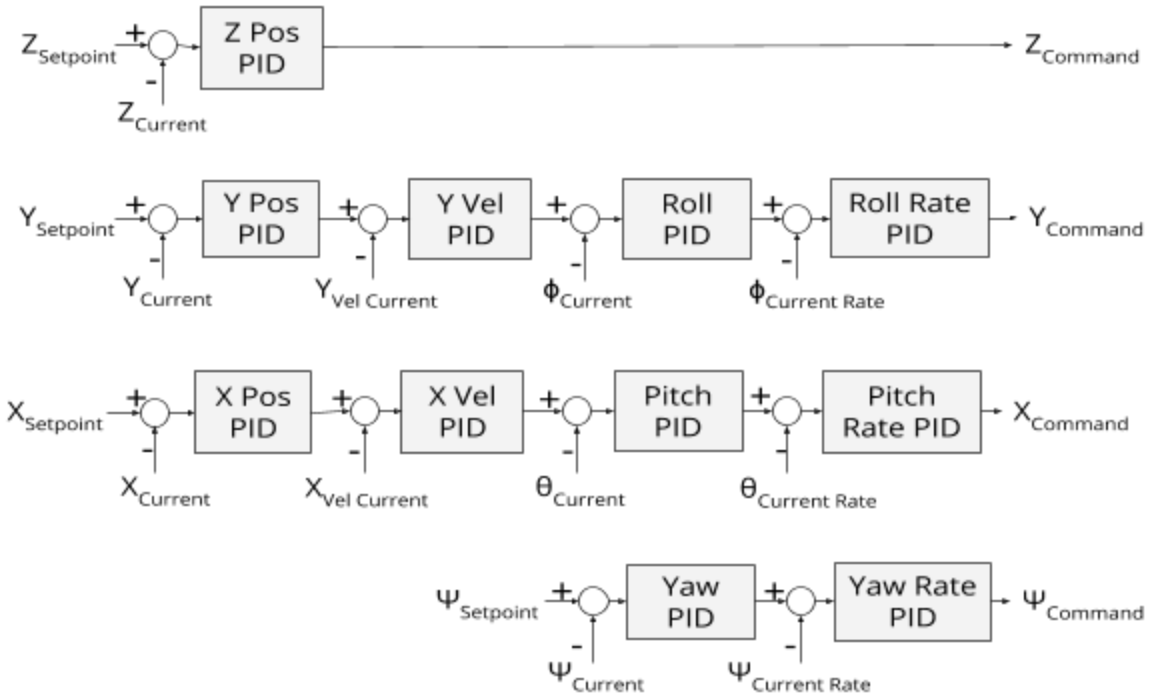

Figure 23: Nested-Loop PID Architecture

This architecture allows us to not only control the body frame position of the quadcopter, but its velocity as well. When describing things in terms of body frame we are referring to the frame of reference of the body of the quadcopter. This assumes that the origin of this axis lies at the center of mass of the body. The variables utilized in Figure 23 are defined in **Table 2** below. Note that in the above image variables denoted with the subscript "r" represent setpoint values.

Variable	Definition
x	Body frame x position
y	Body frame y position
z	Body frame z position
x_{vel}	Body frame x velocity
y_{vel}	Body frame y velocity
z_{vel}	Body frame z velocity
ϕ	Body frame roll angle
θ	Body frame pitch angle
ψ	Body frame yaw angle

ϕ_{rate}	Body frame roll angular velocity
θ_{rate}	Body frame pitch angular velocity
ψ_{rate}	Body frame yaw angular velocity
$Z_{command}$	Motor correction command from Z control loop
$Y_{command}$	Motor correction command from Y control loop
$X_{command}$	Motor correction command from X control loop
$\Psi_{command}$	Motor correction command from Yaw control loop

Table 2: PID Architecture Variable Definitions

Finally, we convert the actual output commands of the controllers to equivalent input commands for each of the four individual electronic speed controllers (ESCs). To combine the motor correction values from each of the four control loops, we utilize a signal mixer, defined by the following matrix:

$$M_g = \begin{bmatrix} 1 & -1 & 1 & 1 \\ 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{bmatrix}$$

4 Testing Process and Results

4.1 Quadcopter Software

The quad software is tested at 4 levels: unit testing of code, functional testing using the virtual quad, dedicated hardware tests for Zybo implemented drivers, and finally actual flight tests. Because the first 2 forms of testing do not require manual execution, we run our unit and functional tests within our continuous integration (CI) software development process, which runs these tests on every change to the code base, allowing us to catch errors immediately.

4.1.1 Unit Tests

Our unit tests are written in C, using a lightweight testing library that we implemented that runs all tests and presents the results. We primarily targeted critical sections of code to cover with unit tests, including the computation graph, the packet processing, and a queue library used by the UART interrupt system.

Here is an example of unit test output for the Queue library:

```
--------------------------------------------------------------------------------
Test results:

#  1: test_free   (passed)
#  2: test_add    (passed)
#  3: test_remove (FAILED)
#  4: test_size   (passed)
#  5: test_full   (passed)
#  6: test_empty  (ERROR!)

Total:   4 of 6   tests passed
--------------------------------------------------------------------------------
```

If any failure or error occurs in the unit tests, the CI test automation will fail, notifying the team of an error that needs to be fixed. At the time of this document, all unit tests pass.

4.1.2 Functional Tests with Virtual Quad

Using the virtual quad, we can verify application layer behavior, such as ensuring appropriate outputs are being produced for certain inputs. These tests have taken the form of Ruby scripts that start the virtual quad, and set certain inputs and check outputs using the virtual quad CLI.

Here is an example usage of the virtual quad CLI:

```
----------------------------
$ ./virt-quad get motor1
0.000000
$ ./virt-quad set rc_gear 1
$ ./virt-quad set rc_flap 1
$ ./virt-quad set rc_throttle 0.5
$ ./virt-quad get motor1
0.539014
$ ./virt-quad set rc_throttle 0.2
```

```
$ ./virt-quad get motor1
0.239014
--------------------------
```

We perform the following tests using the virtual quad:

- Safety Checks
    - Ensure motors cannot turn on unless certain conditions are met
    - Ensure motors can be killed by a single switch
- Motor Compensation Smoke Tests
    - Ensure that when the Quad is tilted according to the IMU, the correct motors get stronger to stabilize
- Communication Smoke Tests
    - Ensure the quad is able to receive bytes through the UART driver, process a packet, and return a correct response.
- Memory Integrity Checks
    - Ensure there are no memory leaks in the application layer.
        - We accomplish this by performing a virtual flight test with the virtual quad through Valgrind, a program designed to catch memory leaks.

Because these tests are executed with Ruby scripts, we can also run these tests on CI, meaning that we have these assurances on each change to our code repository, with a team notification going out if any one test fails. At the time of this document, all functional tests pass.

4.1.3 Dedicated Hardware Tests

In order to have test coverage for drivers with a Zybo implementation, we have manual tests written in C, designed to run with the Quad connected to the computer. These tests require manual verification for the most part.

Here is the complete list of manual hardware tests we have:

- A simple “blink” LED test, to verify that the LED and System drivers work correctly
- Tests that read each I2C sensor, to verify the IMU, LiDAR, Optical Flow, and I2C drivers
- Tests to examine PWM inputs, to verify the RC Receiver drivers
- A motor ramp tests, to verify the Motor driver
- Another LED blink test, to verify the Timer drivers
- A UART reflection test, where a Python script sends bytes over a USB cable to the UART device on the Zybo and checks that the same bytes are sent back. Used to verify the UART driver.

4.1.4 Live Flight Tests

For final verification of code and controllers, we need to perform flight tests using the real quadcopter and system. Tests using the quadcopter must follow specific procedures to ensure safety:

1. If major code changes have occurred since the last flight and all of our software tests pass:
    a. We load the code onto the quadcopter, but remove the propellers.
    b. At half throttle, we verify that the basic stabilization works by tilting the quadcopter in all four directions and monitoring the speed of each motor.

2. If step (1) verifies, then we tether the quadcopter to the ground and attach the propellers. We perform a manual flight to verify that the manual mode is still stable.
3. If our test involves autonomous flight changes, we will take one of the following approaches, depending upon the type of the changes and what we are testing.
    a. Have autonomous mode control pitch, roll, and yaw, but keep the throttle manually controlled. This lets us start the quadcopter on the ground, and slowly ramp up our throttle. Because we start low to the ground, if the controller is unstable, we will notice it while still low to the ground and the flight can be aborted without consequence.
    b. For some flights, starting low to the ground presents difficulties because the ground effect causes turbulence, making it more difficult for a controller to remain stable. If this is the case, we allow the quadcopter to fly higher in the air, but remain prepared to switch to manual mode and land should the quadcopter become unstable.

After performing flight tests, we analyze the flight logs using data analysis scripts in MATLAB. First, we use the DataAnalysis.m script to import the data. Then, we use our own, custom scripts to plot relevant data to help us understand why the system performed the way it did.

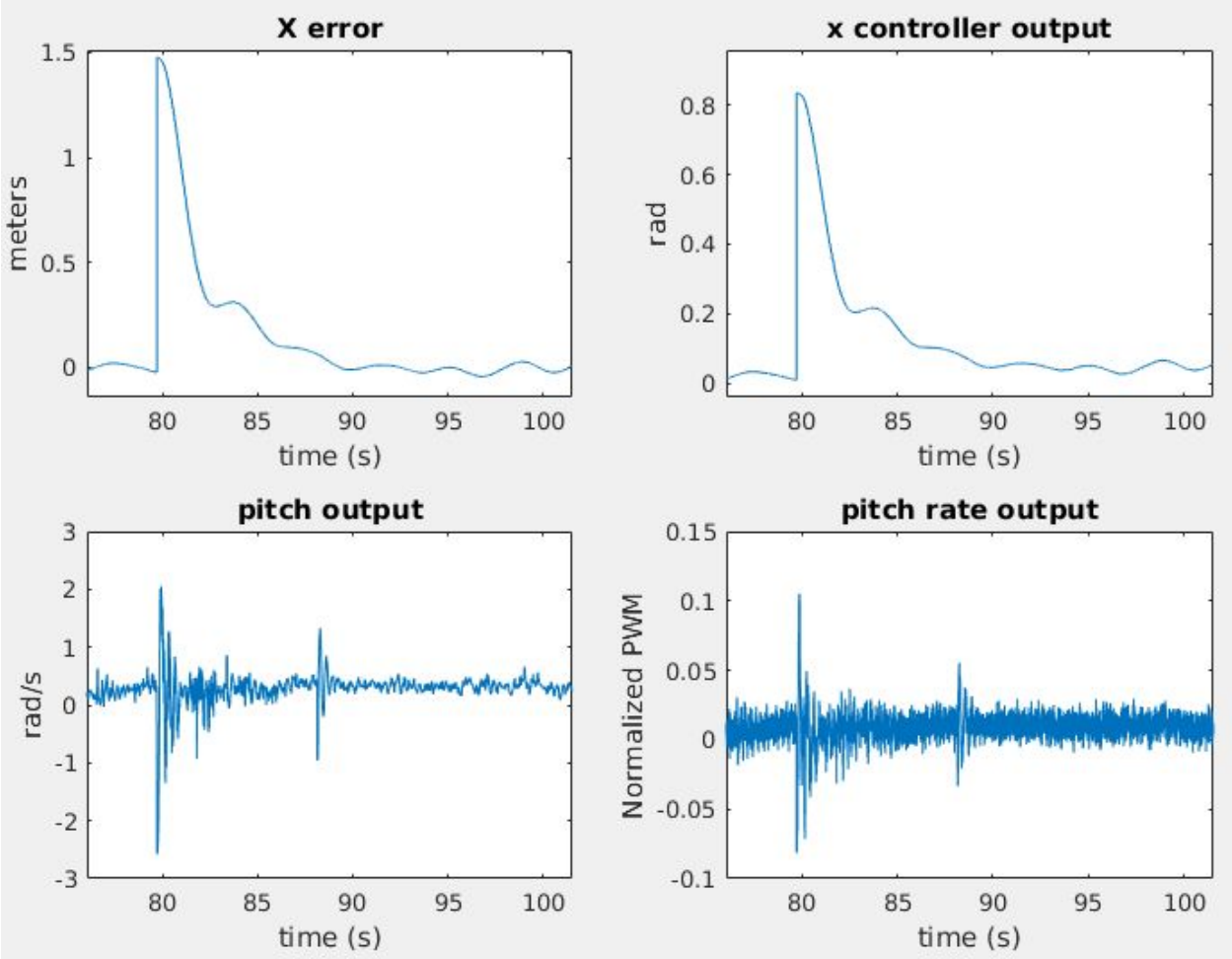

Figure 24: Plots of Nested PID Outputs from "simplePlots.m" Script

We analyzed the performance and found the following characteristics of our autonomous controller.

Axis	1-meter Step Rise Time (s)	Error Margin (m)
X/Y	3.1	± 0.05
Z	0.9	± 0.08

Table 3: Rise Time and Error Margin of Position Controllers

4.2 Control Structure and Tools

4.2.1 Mathematical Model Testing and Results

By taking experimental data of the angular speed of each rotor and then using equation 5.1 to find our estimated rotor speed based on the ESC percent duty cycles, we obtain the following graph:

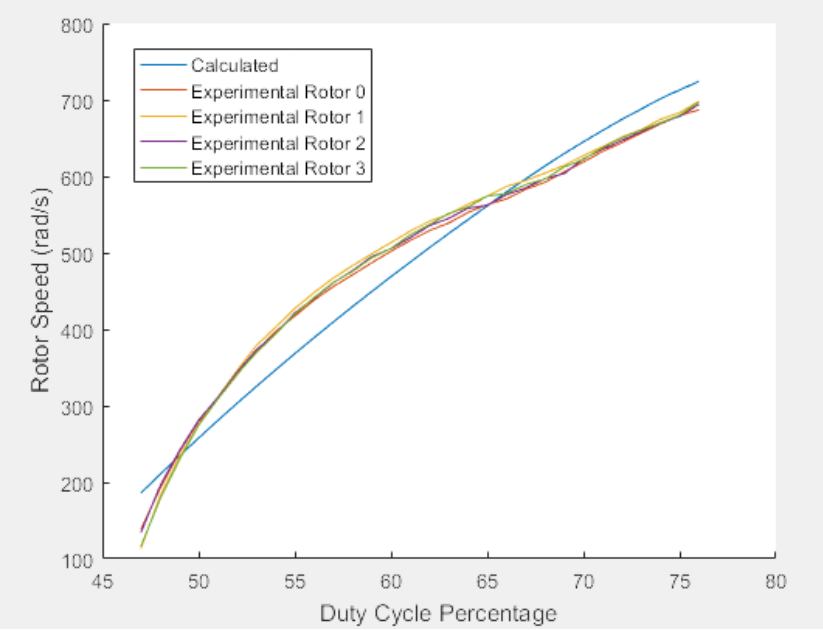

Figure 25: Calculated and Experimental Duty Cycle versus Rotor Speed

In the above graph the blue line represents our calculated rotor speed based off of equation 5.1, and the other lines represent the experimentally determined rotor speeds. From this we can calculate the error associated with our calculation:

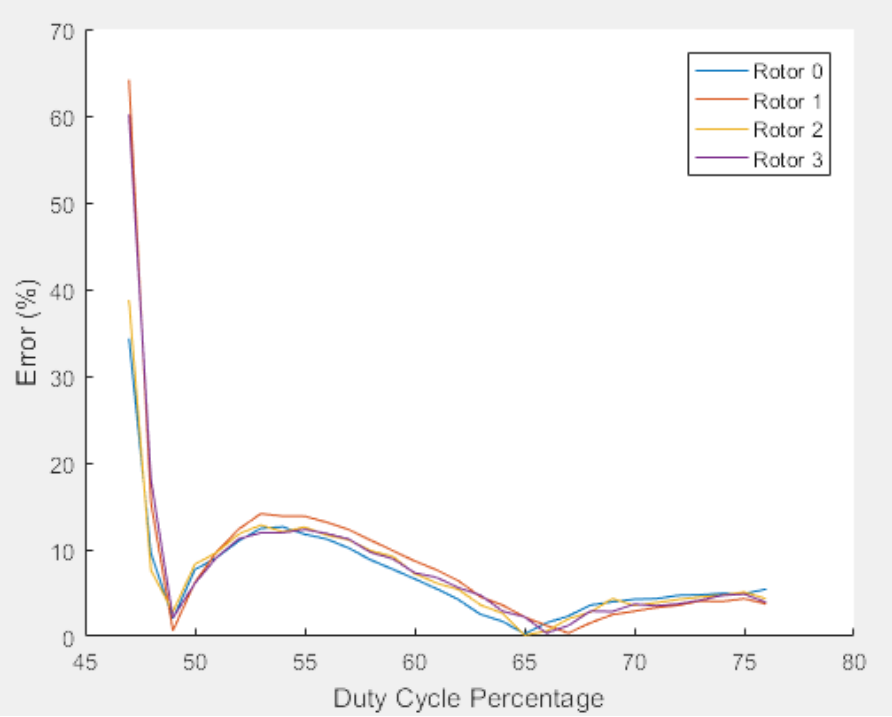

Figure 26: Error Between Predicted and Experimental Rotor Speed

From this we note that at low duty cycle percentages we have a very higher error; however, this is outside of our normal operating range. The typical error we would see within our operating range is between 5% and 15%, which is experimentally shown to be acceptable with the robustness of our controllers, as will be demonstrated next.

Additionally, we tested our Simulink model of the quadcopter on each of the four blocks. These tests will include the following steps, along with the corresponding results we expect to see in Simulink:

Test 1 for Simulink Model: Turning all quadcopter motors at the approximate hovering percent duty cycle - the percent duty cycle that directly counters gravity's downward force

- Should be approximately hovering
- X, Y, and Z position should all be constants
- X, Y, and Z velocities should be zero
- Roll, pitch, and yaw angles and angle rates should all be zero

Test 2 for Simulink Model: Turn quadcopter motors on maximum percent duty cycle per motor

- Should accelerate upwards
- Z position should be a negative parabolic function (Z is positive in the downward direction)
- X and Y positions and velocities should be at zero
- Z velocity should be a negatively-sloped line (Z is positive going down)
- Roll, pitch, and yaw angles and angle rates should all be zero

Test 3 for Simulink Model: Turning motors to right at a slightly smaller percent duty cycle than those to left - while setting the gravity vector to zero for simpler analysis

- Should roll in positive direction, flipping about its center, while going to the right
- X positions and velocities should all be zero

- Y should be a positively-biased sine function that diminishes in amplitude (starts at zero, goes positive when quadcopter circles right, then goes back to zero as the quadcopter circles around)
- Z should be a negative sine function (starts at zero, goes negative when quadcopter goes up, back to zero, goes positive when quadcopter goes down)
- Pitch and yaw angles and angle rates should be zero
- Roll angle should be a positive and increasing at a parabolic rate
- Roll angle rate should be a positively-sloped line

Test 4 for Simulink Model: Repeat Test 3 but set the motors to the left at a slightly smaller percent duty cycle than those to the right - should get the same results, except:

- Y position and velocity should be negative of what they were in Test 3
- Roll angle and angle rate should be negative if what they were in Test 3

Test 5 for Simulink Model: Repeat Test 3 but set the motors to the back at a slightly smaller percent duty cycle than those to the back - should similar results, but it will be analogous for the pitch rotation

- Should pitch in positive direction, flipping about its center, while going to the backward
- Y positions and velocities should all be zero
- X should be a negatively-biased negative sine function that diminishes in amplitude (starts at zero, goes negative when quadcopter circles back, then goes back to zero as the quadcopter circles around)
- Z should be a negative sine function (starts at zero, goes negative when quadcopter goes up, back to zero, goes positive when quadcopter goes down)
- Roll and yaw angles and angle rates should be zero
- Pitch angle should be a positive and increasing at a parabolic rate
- Pitch angle rate should be a positively-sloped line

Test 6 for Simulink Model: Repeat Test 5 but set the motors to the top at a slightly smaller percent duty cycle than those to the bottom - should get the same results, except:

- X position and velocity should be positive, not negative, of what they were in Test 5
- Pitch angle and angle rate should be negative if what they were in Test 5

Additional Tests: Try different combinations to predict what the quadcopter will do and if it makes sense intuitively. Not only will this be good as it is additional testing for the quadcopter model system, but it also helps the tester gain a better understanding for how the quadcopter system is supposed to work

From these tests, we tried applying an upward thrust with equal percent duty cycles for each motor, expecting to see the quadcopter going straight upward, which would be in the negative z-direction since z is positive downward. Then here were the graphs we got as a result:

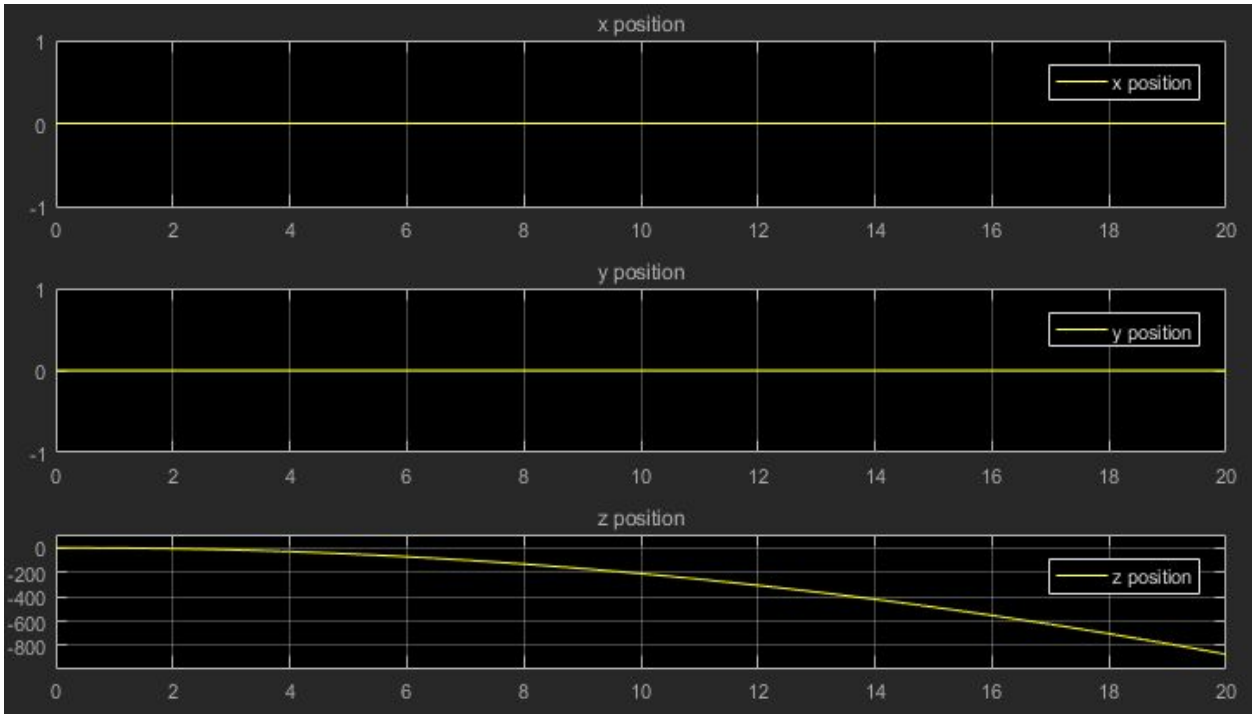

Figure 27: X, Y, Z Position Calculated by Simulink Model

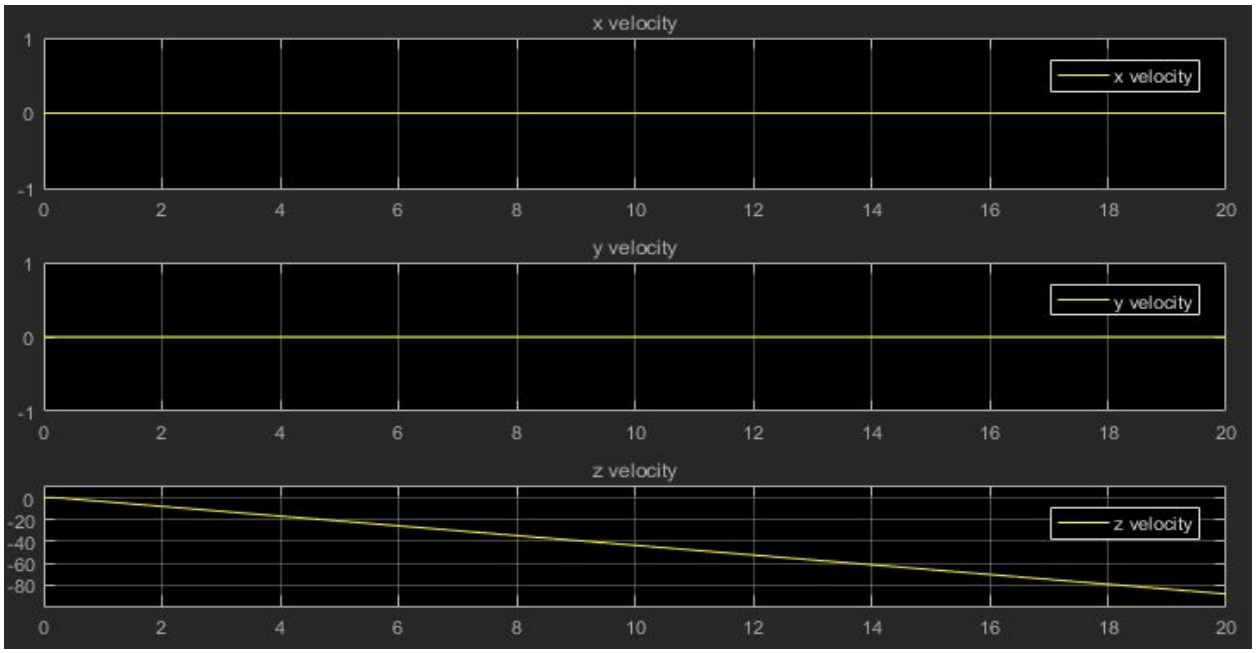

Figure 28: X, Y, Z Velocities Calculated by Simulink Model

As you can see, these calculate results make sense for the linear and angular position and velocities for our Simulink model. The X, Y, Z position make sense, because the quadcopter model should be going directly upward, which would make the X and Y position zero and the Z position a negative parabolic function with respect to time, since it should be accelerating upwards for there is a constant net force

upward, and the positive z-position is pointed downward. As a result, the first graph matches our expectations.

Additionally, the X, Y, and Z velocities of the second graph also make sense, since the quadcopter should not be moving in any direction in the X and Y directions, thus these velocity values should be zero, however it is accelerating upward, thus the Z-directed velocity should be increasing in magnitude, but negative in sign, since the Z axis is positive in the downward direction. Thus, these results make sense for our predictions and expectations for the behavior of our Simulink model.

We ran many tests like this on the Simulink physics model, as described in detail above, and examined the results in Simulink to verify the accuracy of our physics model.

We also implemented a mode where we input signals from logged flight data into each block of our Simulink model, then we compare the output signal of these blocks with that of the corresponding signal from the same logged flight test. Here are some example plots obtained to verify each block in our sensor subsystem of the model:

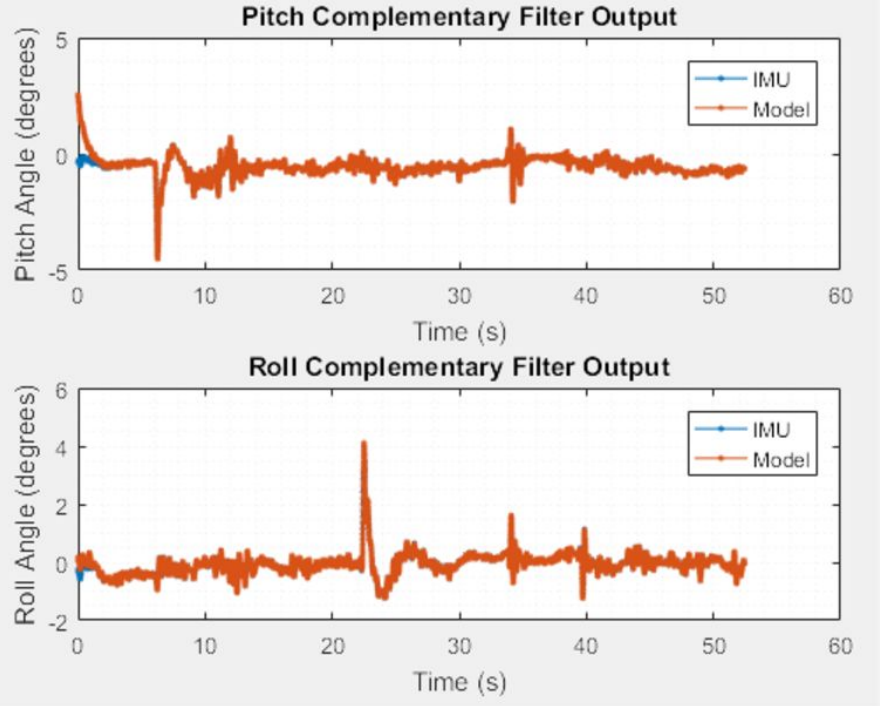

Figure 29: Complementary Filter Output of Model versus Logged Flight Data

We additionally, looked at the closed-loop system and input setpoints that we ran on the physical block, comparing it to what our model predicts for the position of the quadcopter. Here was the resulting plot:

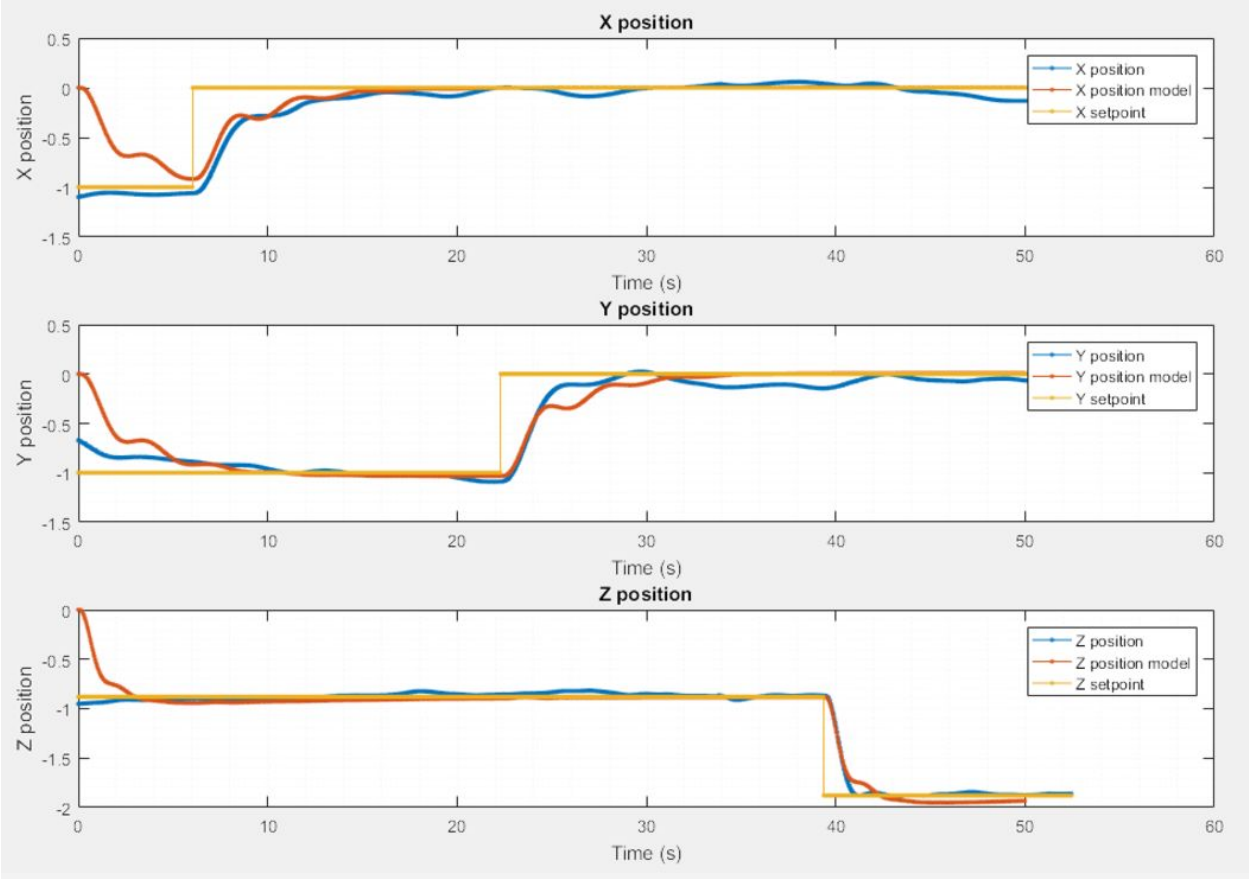

Figure 30: X, Y, Z Position of Model versus Logged Flight Data

As you can see, over a 50-second time period, the model is closely matching the position measured by VRPN with the different setpoints. The first few seconds are a bit different, but this is only because of the different initial conditions of our model, which always starts from zero, compared to the initial conditions of the position of our actual quadcopter, which is offset from the (0, 0, 0) point of the flying volume in the lab.

4.2.2 PID Controller Testing and Results

The current tests we have performed with regards to the PID controllers are primarily error calculations from the mathematical model of the quadcopter. To obtain accurate PID constants we need to model the quadcopter as best as we can to simulate a real scenario. One portion of this model involves the relationship between the electronic speed controllers (ESCs) and their relationship to the rotor speed. This relationship is as follows:

$$\omega_i = \frac{-1 + \sqrt{1 - 4R_m K_v K_q K_d (K_v R_m i_f - K_v u_{p_i} V_b)}}{2R_m K_v K_q K_d} \tag{4.1}$$

Variable	Definition
ω_i	Rotor Speed
R_m	Internal motor resistance
K_q	Motor torque constant
K_v	Motor back-emf constant
i_f	Motor internal friction current
μ_{pi}	ESC input duty cycle ratio
V_b	Nominal battery voltage

Table 4: Rotor Speed Equation Variable Definitions

For our testing procedure, we need to take experimental data of the angular speed of each rotor and then use the above equation to find our estimated rotor speed based on ESC input duty cycle percentage. With these data, we represent a relationship with our experimental data of the motor speed with respect to the percent duty cycle of the input PWM signal, and we can compare this with our predicted motor speed using the expression derived above. This test will be done for each of the 4 motors.

See section 4.1.4 Live Flight Tests for the testing procedure and results of the control system testing.

4.3 WiFi Bridge

The UART interface for the WiFi bridge can be tested by directly connecting the RX/TX lines on the WiFi bridge, and connecting to the WiFi module with a simple terminal(PuTTY on Windows, netcat on Linux). Every command sent over the terminal should be echoed back. This is a simple test to verify that we can send large and small amounts of data without loss.

For more robust testing, we have created a test script in Python that will send arbitrary sized packets, with different delays. It verifies the integrity of the packet, and records if any data was corrupted or lost.

We currently have implemented and tested the TCP portion of the WiFi bridge. We have run the test with various packet sizes and delays, to verify that the WiFi bridge is able to sustain the data rates that we will be sending. During regular flight, we will be sending roughly 32 bytes every 10ms. To verify that we are able to reliably send sufficient data, we have simulated 30 minutes of flight time, sending 350 bytes every 10ms, without losing any data.

We have done timings for round-trip between ground station and quadcopter response. **Figure 31** shows the latency distributions for WiFi vs. Bluetooth communication.

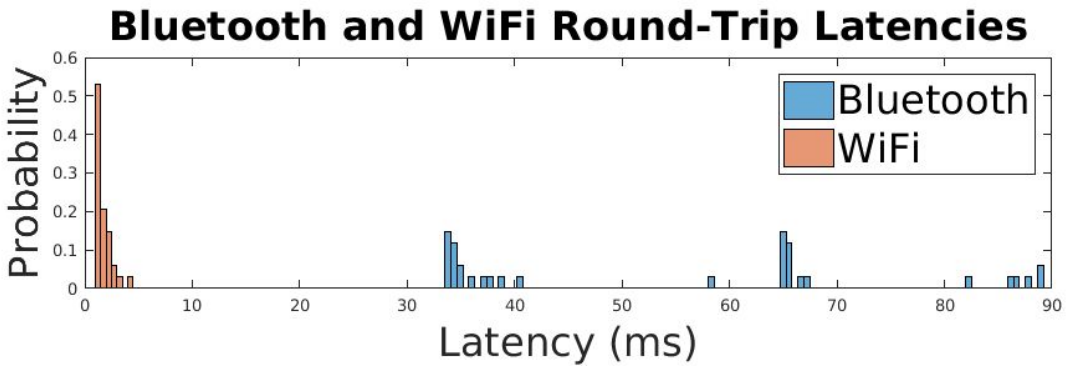

Figure 31: TCP versus Bluetooth Latency Distribution

4.4 Hardware Improvements

For the voltage regulator, we will first verify that the output voltage is within our required specifications. Then, we will increase the load on the system using high-power resistors until our maximum current rating is reached. At that point, we will verify that the output is still within the specifications.

After the voltage regulator has been fully tested, the last system to test are the connections for the ESCs and the WiFi module. For this, we will simply perform continuity tests between all connections to make sure that they are properly connected. Then, we will connect all of the devices and verify that all signals are being properly routed without being distorted. If all signals are correct, the board is functional.

5 Conclusion

This year (the 2016 to 2017 academic year), MicroCART has improved the current system by increasing modularity and designing PID controllers to manage the movement of the quadcopter in each axis of rotation and linear motion.

We have also implemented LIDAR and Optical Flow sensors to enable flights outside of the camera tracking system in the lab, as well as a more modular and extendable control structure.

Additionally, we were able to develop a mathematical model of the current quadcopter system to help future teams more easily find controllers that stabilize newer systems. This development, along with the new graphical computation library, offers future MicroCART teams and ECpE students a more manageable way for changing controller schemes quickly.

Alongside these major improvements, we also made progress in areas such as reduced latency using Wifi, better testing schemes for both hardware and software integration, and development of a user-friendly GUI. All of this was done in an attempt to create a system that could be used in the future as a learning platform for students interested in controls and embedded processing.

Appendices

Appendix A: How to Fly the Quadcopter

Follow these instructions to get the quadcopter up and running in Coover 3050.

Setup Infrared Camera System

1. To start up the camera system, log into the camera system computer (co3050-07) with the following username and password:

   username: `.\microcart`
   password: `microcart`

2. Once the OS is done loading, start up the program "Tracking Tools"
3. From the startup window, choose "Open existing project"
4. Choose "TrackingToolsProject 2017-01-13 5.30pm" (or a more recent project) in the "Optitrack_Configuration" folder
5. Then go to File -> Open and choose "Microcart" in the "Optitrack_Configuration" folder
6. This should create a "UAV" under "Trackables" in the Project Explorer on the left side of the screen
7. Now you should be able to move the quadcopter trackable around in the tracking area, and see it update in real-time on the screen.

Setup Ground Station

1. On the ground station computer (Co3050-microcart), log in with the following credentials.
   username: `ucart`
   password: `microcart`
2. Navigate to the ground station folder in a Terminal.

   ```
   $ cd {project_root}/groundStation
   $ ls
   BackEnd  Cli  logs  Makefile  obj  README.md  src  ucart.socket
   ```

3. If the project hasn't been built in a while, re-make the project:

   ```
   make vrpn
   make
   ```

Setup Transmitter

1. The RC transmitter is used to manually control the quad.
2. Ensure the transmitter has the following state before turning it on:
   a. "Gear" is set to 0

b. "Flap" is set to 1
c. Throttle is set to the lowest position

3. Turn on the transmitter.

Setup Quadcopter

This section assumes the quad already has a prepared boot image inserted into the SD card port and that a properly charged Li-Po battery is ready for use.

1. Prerequisites
    a. Make sure the connection to the motors from the main power line is disconnected.
    b. Make sure the previous setup sections have been done prior starting this section
2. Insert the Li-Po battery into the holder beneath the quad, and plug it into the quad.
3. Turn on the Zybo Board using the switch.
    a. The "PGOOD" Light should turn red.
    b. After the program has been completely loaded, the green DONE LED should turn on.
4. Ensure the quadcopter and transmitter has connected successfully.

Figure 32: Transmitter Connection Light

    a. The RC transmitter should have GAUI 330X selected and displayed on the screen. With the quadcopter and transmitter on, the unit on the quadcopter labeled Spektrum AR610 should have a blinking orange light or solid orange light (It is easier to see the orange light from the top of the receiver). If this is not blinking or solid, try restarting the quadcopter and transmitter with the transmitter closer to the quadcopter.

5. Plug connect the motors to the main power line.

Start the Ground Station (CLI)

Execute the following on the ground station from the root of the ground station folder.

In one terminal, run the backend

```
./BackEnd
```

Finally, in another terminal, export the socket path, and then execute any CLI commands that you like:

```
./Cli setparam 'X pos PID' 'Setpoint' 1.000

# ... other CLI commands
```

Start the Quad

1. Using the transmitter, flip the "Gear" switch to 1.
    a. You should see the green LED4 MIO7 turn on.
2. Start flying the quad. Below is a summary of how the manual controls work:

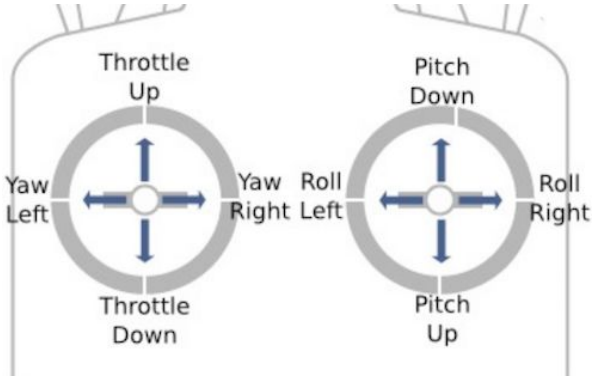

Figure 33: RC Controller Operation

Appendix B: Changes to Original Design Plan

Originally, we planned to make our controller structure for the X and Y loops have 3 nested PID controllers. Here was the original control structure we planned on having originally:

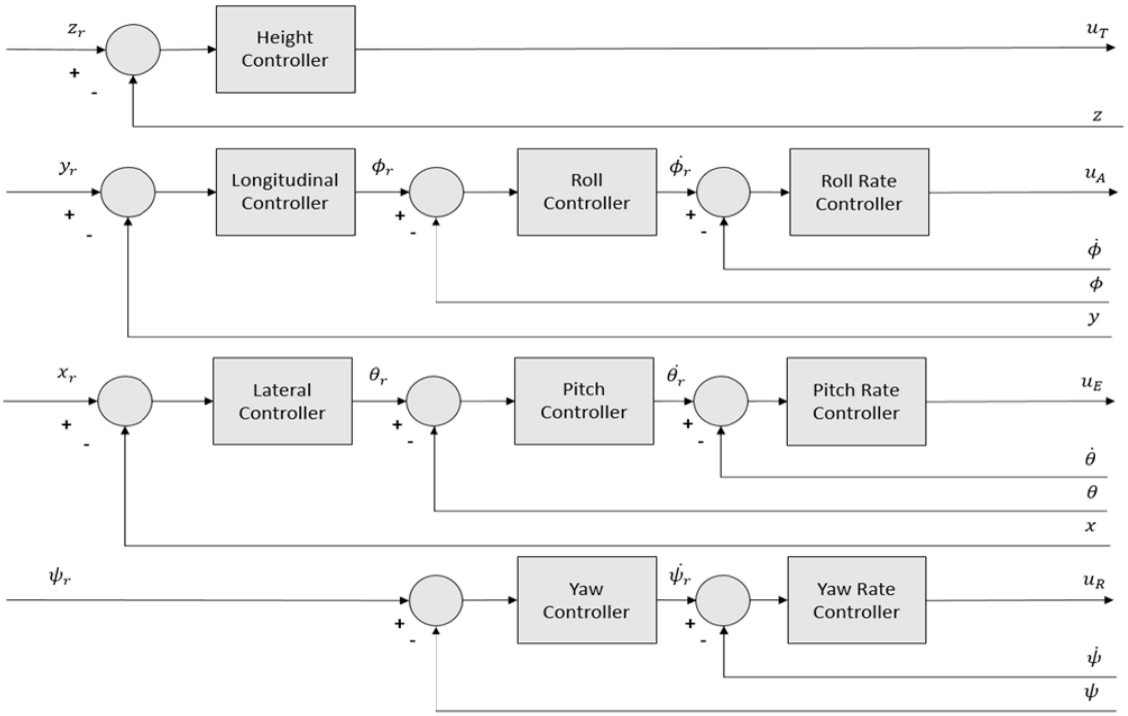

Figure 34: Original Control Structure

However, we found that we were able to find stabilizing controllers using a fourth nested loop, with a velocity PID controller between the position and pitch. Because of the better, more stable performance, we decided to change our control structure to include this fourth PID for the X and Y control loops.

Another addition to our design we added was the Optical Flow sensor. We included this because we realized that using GPS to obtain our X and Y position could become problematic if we are flying in a place with poor reception, such as in the courtyard of Coover. In these areas, the Optical Flow sensor can provide us with good values for the change in X and Y position of the quadcopter by integrating the velocity vectors found in these two directions.

Finally, another major improvement we made to increase the modularity of the entire system was to create a node-structure representation of the control system on the quadcopter. We did not think of this design initially, which is why we did not previously have it in our design plan, but when a member of our group thought of it, we realized how useful this structure would be to make the entire platform more modular, so continued this approach. With this implementation, each element of the control structure can easily be replaced with any block of C code that the user writes, and the connections between different nodes can easily be changed to change the control structure or to bring in data from different sources, rather than having to make large modifications to the code.

Appendix C: Suggestions for Future Work

Future improvements that can be made to the system, such as by future MicroCART teams:

- Create a PCB power board with the following features:
    - Voltage regulator for powering Zybo board
    - Reverse polarity protection
    - Sense current to determine power drawn
    - Battery voltage monitor
- Implement GPS position data acquisition (driver for GPS has been created)
- Edit Simulink model to include the Optical Flow and LiDAR sensor, which should include:
    - Noise characteristics of the sensors
    - Biases in the sensor data

References

[1] *Bluetooth vs Wi-Fi. (n.d.). Retrieved November 19, 2016*, from http://www.diffen.com/difference/Bluetooth_vs_Wifi

[2] Cavallo, A., A. Cirillo, P. Cirillo, G. De Maria, P. Falco, C. Natale, and S. Pirozzi. *Experimental Comparison of Sensor Fusion Algorithms for Attitude Estimation*. Thesis. Second University of Nepales, 2014. Aversa: ScienceDirect, 2016. Print.

[3] Ogata, Katsuhiko. *Modern Control Engineering*. 5th ed. Englewood Cliffs, NJ: Prentice-Hall, 1970. Print.

[4] "Products." *DJI Store*. DJI, 2016. Web. 12 Oct. 2016. <http://store.dji.com/>.

[5] Rich, Matthew. *Model Development, System Identification, and Control of a Quadcopter Helicopter*. Thesis. Iowa State University, 2012. Ames: Graduate Theses and Dissertations, 2012. Web.

[6] *Zynq-7000 All Programmable SoC Overview*. DS190 (v1.10). Xilinx. September 27, 2016