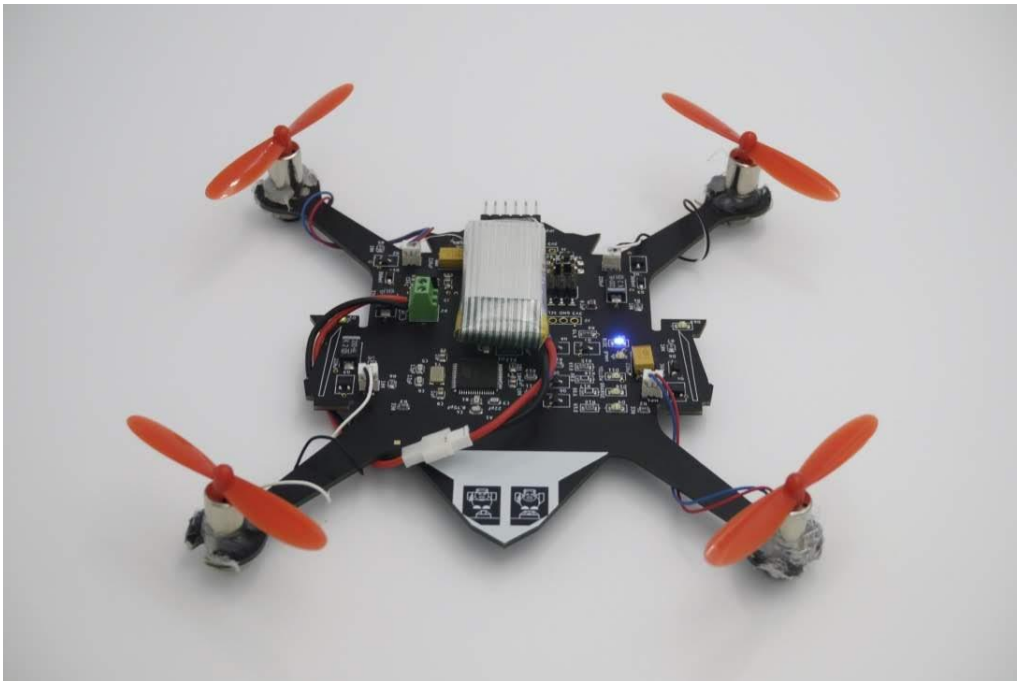
When Pigs Fly

Designing, Assembling, Programming, and Flying a Quadcopter Drone from Scratch



Robert Ketchum

With

Thomas Stuart

Abstract

A Printed Circuit Board (PCB) for a quadcopter was designed utilizing Autodesk EAGLE. The PCB was assembled using reflow and hand soldering techniques, and the microcontroller was flashed. Firmware for the remote and the quadcopter was written in C++. The control system of the quadcopter was designed utilizing ad-hoc PID tuning methods. The quadcopter was flown successfully. A video of the quadcopter can be viewed at: https://tinyurl.com/quadflopter. Arduino firmware and EAGLE quadcopter board files are available at: https://github.com/rketch/quadcopter.

Acknowledgements

Designing, assembling, programming, and flying a quadcopter in only 10 weeks seemed like an impossibility when we started out at the beginning of the quarter, but proved to be one of the most rewarding, fun, and stressful times of my undergraduate at UC San Diego.

My teammate Tom was responsible for much of the success of this project. Infinitely driven, his stories injected much needed levity during the long hours in the laboratory, on weekends at the coffee shop, and late nights in the library. I will always remember his patience with me as a relatively inexperienced (and messy) C++ programmer at the start of the class. I was extremely lucky to have him as a teammate.

This project would not be possible without the endless support, assistance, and motivation of Professor Steven Swanson. The success of any student of the class is built upon Prof. Swanson's extensive quadcopter knowledge as well as the rigorous course framework. All students of the class were provided course material including: the overall quadcopter electronics design, borrowed flight control and remote boards, Arduino and EAGLE libraries, and online laboratory instructions. The result is a class in which any motivated and technically minded UCSD student can learn the fundamentals of mechatronic design with ease.

I would also like to thank the TA for the class, Kunal Gupta. His printed circuit board thermal paste application and hand soldering tutorials were essential for assembling the quadcopters.

Lastly, the UC San Diego EnVision Studio staff generously provided students with a workspace as well as soldering lessons and materials. Their tireless effort to support student projects is greatly appreciated.

Table of Contents

Figures and Tables

Introduction

Quadcopter drones are the future. From transportation to entertainment, from land surveying to delivery services, drones are helping humans across aspects of life. Commercial drones usually have features such as autonomous navigation, cameras, and/or vectorized thrust for end user assistance or ease of control. However, these features all build and rely upon the fundamental drone functionality: flight.

To this end, this report focuses on the full design process of a pair of simple, barebones quadcopter drones which were assembled, programmed and flown by teammate Thomas Stuart and myself in the UC San Diego course CSE 176e: Robot Systems Design and Implementation (The Quadcopter Class).

For greatest use, supplement this document with the laboratory instructions for CSE 176e, which may be viewed at: https://github.com/NVSL/QuadClass-Resources/.

PCB Schematic

Overview

A circuit schematic presents a streamlined, electrical connection focused view of a circuit. This helps a designer build the individual components into a functional PCB without worrying about the appearance of the eventual physical circuit. Note that computer programs such as Autodesk EAGLE link the selection of a schematic component with a physical part used in the PCB layout, so care should be taken to ensure the correct version of a part is chosen. To aid in this process, the datasheet of each component provides the size, type, and connectivity requirements of each terminal, any additional components (capacitors, resistors, etc.) needed for functionality, as well as heat dissipation, voltage, power, and temperature requirements [1] [2].

The sections which follow discuss each subgroup of the quadcopter circuit broken down by functionality. Figure 1 below displays the full quadcopter schematic, which is also reprinted in Appendix B in higher resolution. See the Bill of Materials in Appendix D for the specific parts used in the quadcopter.

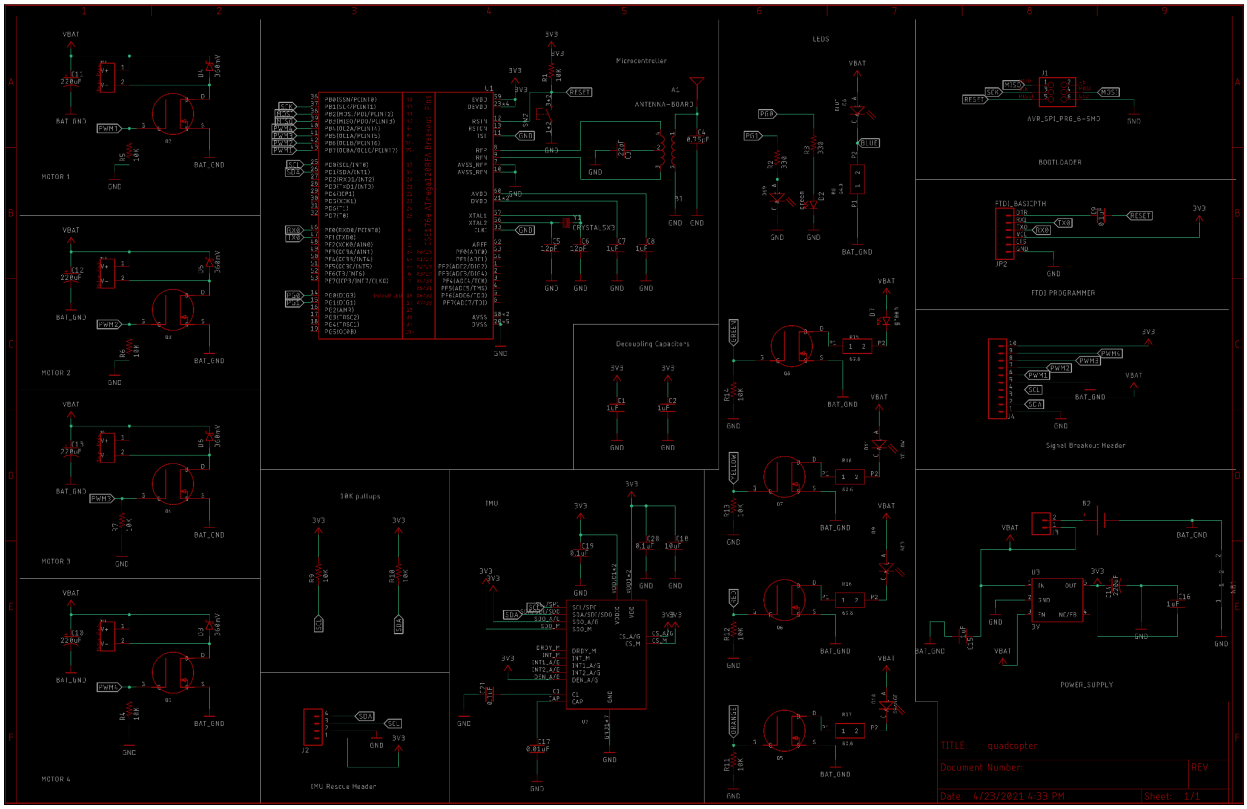


Figure 1: Quadcopter Circuit Schematic

Microcontroller

The microcontroller is the brains of the quadcopter. True to its name, it controls the entirety of the quadcopter board including: reading and processing IMU orientation data, writing motor throttle PWM signals, and maintaining radio communication with the remote control.

The microcontroller used in the quadcopter (and remote control) is the Atmega128RFA1. It features 128 kilobytes of flash memory, 4 kilobytes of EEPROM, and a single I2C line, used to communicate with the IMU. Most importantly for us, this microcontroller contains a pair of RFP/RFN pins which allow for easy radio communication from the remote control to the quadcopter [1]. See the **PCB Layout** section for more information on the radio.

The radio and microcontroller schematic is displayed in Figure 2. In compliance with the datasheet, a 16 MHz external crystal oscillator was used for nominal microcontroller and radio transceiver performance. A reset button was also added for ease of testing firmware.

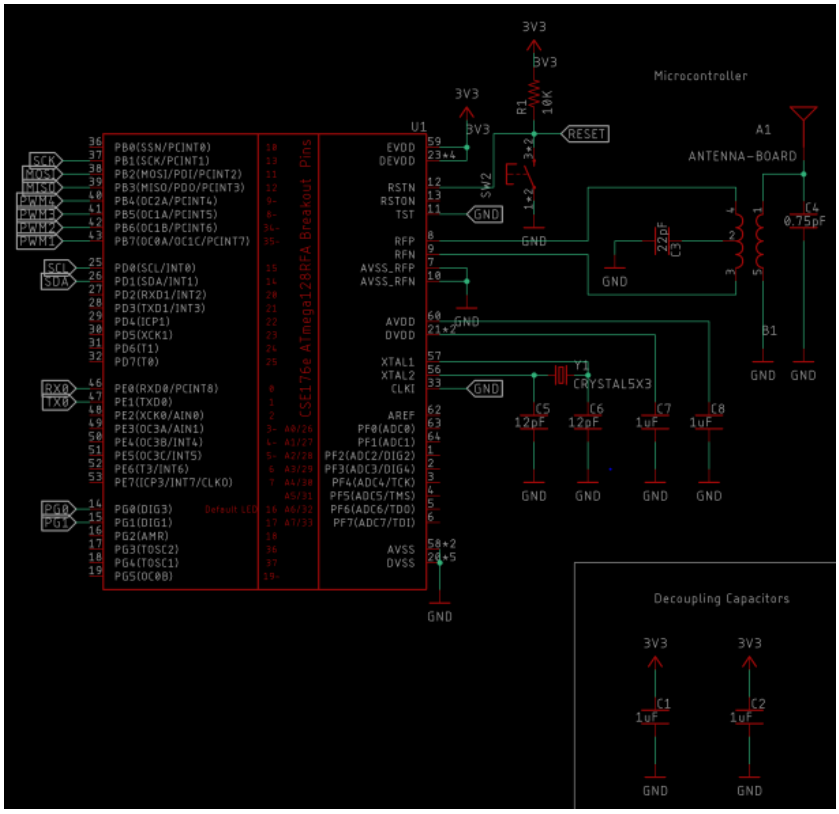


Figure 2: ATmega128RFA1 Microcontroller, Radio, Reset Button, and Crystal Oscillator Schematic

IMU

The Inertial Measurement Unit (IMU) is a sensor which may be used to read the spatial orientation of the quadcopter. The LSM9DS1 was chosen for this project as it has an accelerometer, gyroscope, and magnetometer which each provide the microcontroller with 3-Dimensional sensor data. It also provides the end user control over sampling rate, high pass and low pass sensor filters, and enable/disabling of each sensor via internal registers [2]. Testing revealed that the LSM is also less sensitive to the high frequency vibrations emitted by the quadcopter motors than other tested IMUs.

The IMU communicates with the microcontroller via I2C. As such, the serial clock (SCL) and serial data (SDA) signal lines require 10$k\Omega$ pullup resistors ensure accurate data transfer, as seen in Figure 3 [2].

The IMU rescue header ensures that a quadcopter with an incorrectly assembled IMU may still fly utilizing a breakout board. See the **Assembly** section for more details.

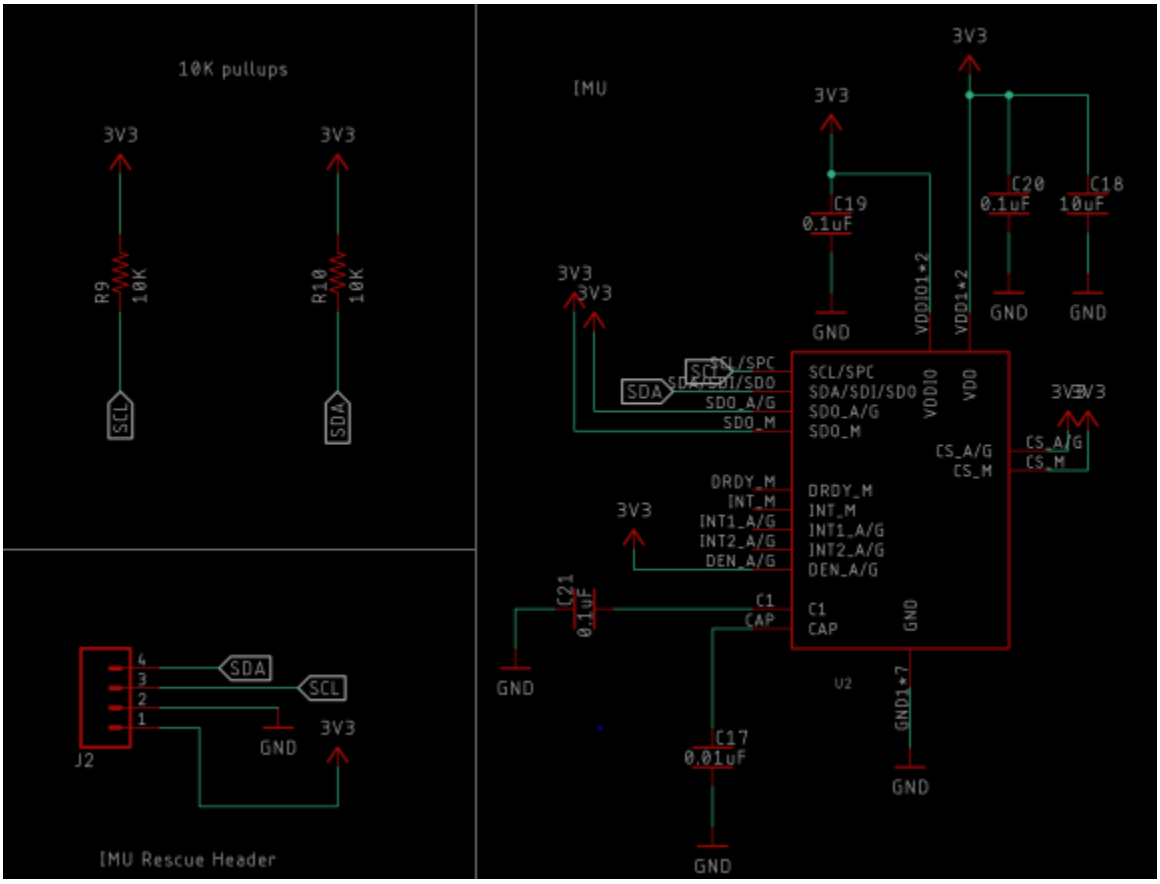


Figure 3: LSM9DS1 IMU Schematic

Power Supply

The quadcopter was powered by a single $3.7V$ $380mAh$ battery. The raw battery voltage was used to directly power the motors and LEDs, but a power regulator was required to step down the battery voltage to a nominal $3.3V$ for the Microcontroller and IMU. The schematic of the full power circuit is shown in Figure 4. The net bridge, NB1, ensures that the battery ground and digital ground are in electrical connectivity while retaining their respective names in EAGLE. See the **PCB Layout** section for further explanation of the importance of the Net Bridge. Also present is a two pin header, J3, which acts as a "key" for the quadcopter. Without shorting the two ends of the headers with a wire, the quadcopter will not turn on.

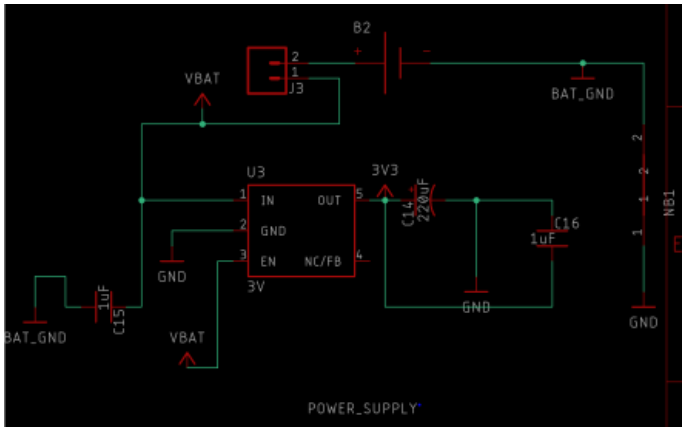


Figure 4: Power Supply Schematic

Motor Drivers

The full circuit schematic for a single motor driver is displayed in Figure 5. A field effect transistor, Q2, was used as a shunt switch to amplify the corresponding PWM motor throttle command from the microcontroller. When the PWM line is off, the gate and the source are at the same voltage, and the drain is held at VBAT. When the PWM line is turned on, the gate voltage is higher than the source voltage, dropping the drain voltage. This creates a voltage drop across the motors, which turns them on [3].

A flyback diode, D4, ensures that an accidental buildup of charge may be safely discharged to the battery, bypassing the motor.

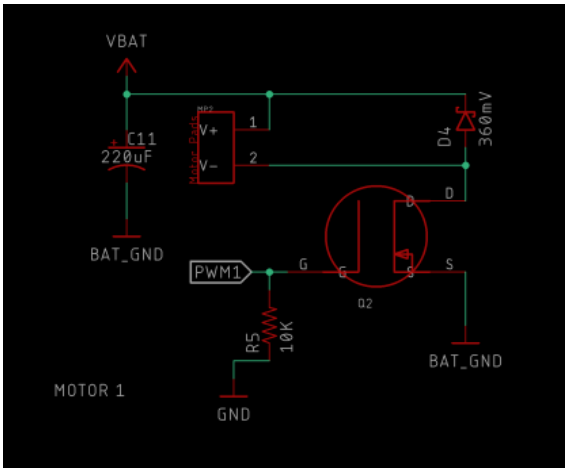


Figure 5: Motor Driver Schematic

ISP Bootloader and FTDI Programmer

A 6 pin ISP device was used to initially bootload the microcontroller such that Arduino sketches could be uploaded via FTDI unbalanced serial [4]. The ISP Bootloader and FTDI programmer schematic components are shown in Figures 6 and 7, respectively. See the **Firmware** section for further discussion of the programming architecture.

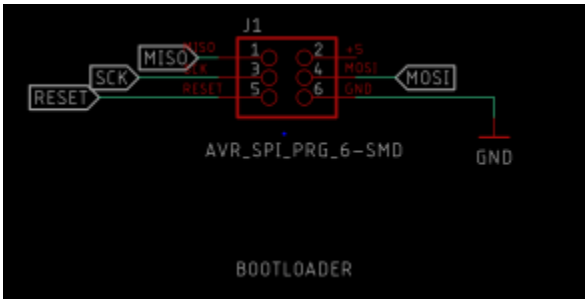


Figure 6: ISP Bootloader Schematic

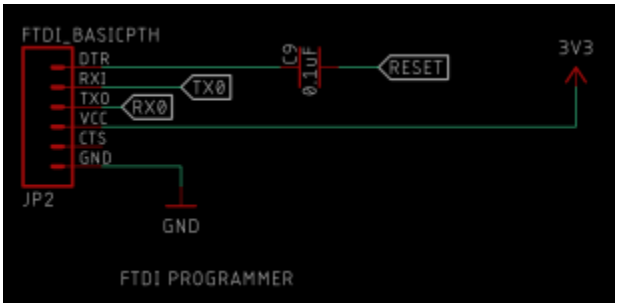


Figure 7: FTDI Programmer Schematic

LEDs

PWM-driven and simple ON/OFF LEDs were both used in this design, as displayed in Figures 8 and 9, respectively.

The first of these LEDs were PWM-driven and were purely aesthetic. True to the project name, the idea was that increasing the quadcopter motor throttle command would turn on each individual LED until liftoff, when a rainbow of LEDs would show that pigs were flying. Of the rainbow, only the blue LED was permanently illuminated, while the rest were PWM driven using the same FET switch design as the motor drivers. Unfortunately, the PWM lines were not connected to their corresponding microcontroller pins in the version of the board which was manufactured (hardware1.0), so only the blue light turned on. This error was fixed in an updated version of the board file (hardware2.0). For a partial fix, shorting the drain and the gate of the FET permanently illuminated the rainbow.

Two ON/OFF LEDs were also used for communicating the arming status of the quadcopter.

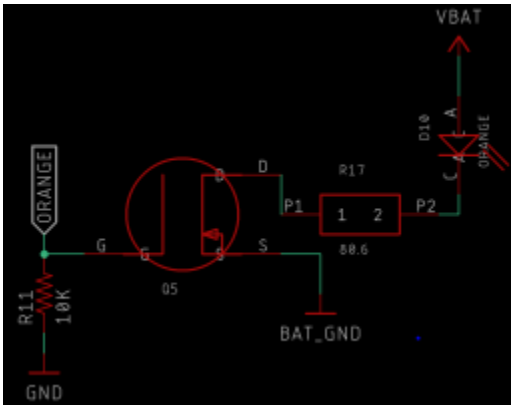


Figure 8: PWM-Driven LED Schematic

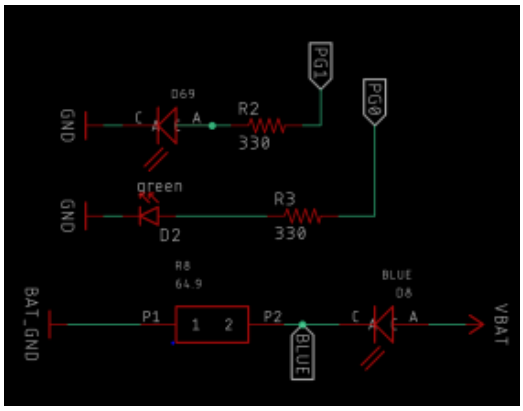


Figure 9: Status LED Schematic

Signal Breakout

For ease of testing electrical shorts and signal voltage, a signal breakout header was included in the design, as seen in Figure 10.

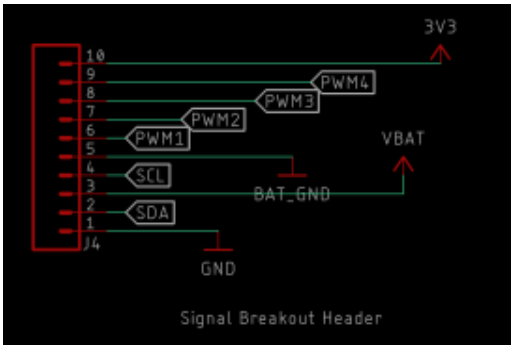


Figure 10: Signal Breakout Header

PCB Layout

Overview

Equally as important as the circuit schematic is the PCB layout, which determines how the physical circuit is manufactured. As seen below in Figure 11 (and in higher resolution in Appendix C), the PCB serves as the full body of the quadcopter, complete with metal arms to hold the four brushed DC motors.

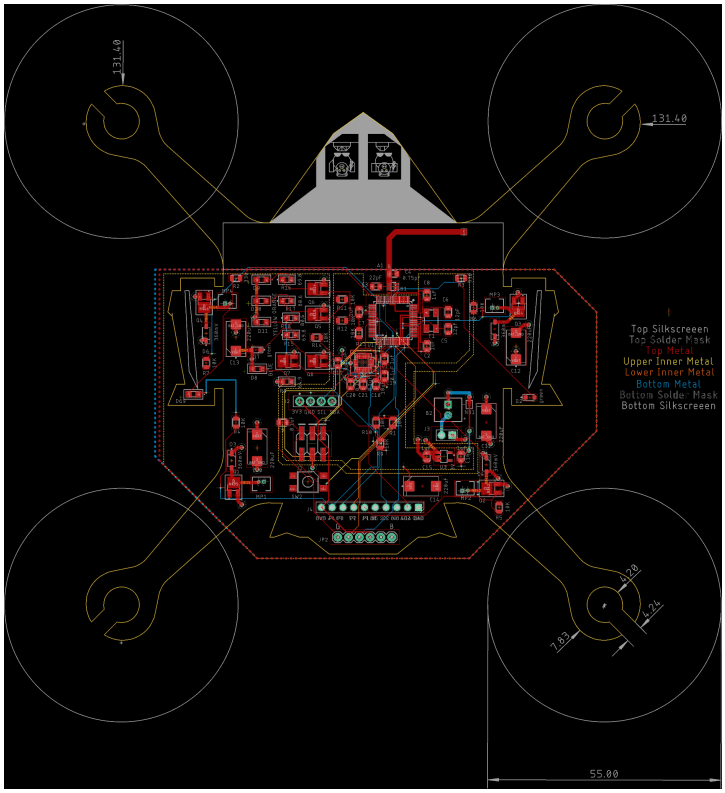


Figure 11: Full PCB layout

Printed Circuit Boards consist of layers of conductive metals and insulating composites. Using a computer program such as Autodesk EAGLE, circuit components are placed on the top or bottom metal layer, and electrical paths are routed to connect corresponding circuit component terminals [5].

Two types of circuit components were used in this design: surface mounted devices (SMD) and through hole devices. SMDs have the benefit of compact packaging, but are more difficult to assemble by hand and oftentimes cannot be altered once assembled. Through hole devices are larger and take up all layers of a PCB, but are easier to assemble and provide greater physical connection to the board [6].

PCB Copper Layers and Net Bridge

The PCB of the Quadcopter was designed using four layers of copper. Of these, the Top and Bottom Layers were used to route connections between circuit components, while the inner two layers were primarily used to distribute power to the motors, microcontroller, and IMU. Due to the presence of a voltage regulator in the circuit, there exist two different voltages in the middle layers. These two voltage levels can easily be seen below in the Lower Middle Layer, represented by the orange, dotted, U-shaped outline of the Battery Voltage and the more centralized 3.3*V* outline which surrounds the microcontroller and IMU in Figure 12. Note that these same outlines are mirrored on the Upper Middle Layer, except the Battery Voltage and 3.3*V* lines are replaced with the Battery Ground and Digital Ground lines, respectively.

While the two ground levels are in electrical connectivity due to the presence of the Net Bridge, the two levels separate the digital and sensor circuitry from the unregulated battery and motor circuit. This was done to ensure that the power and ground lines connecting the battery to the motors are as wide as possible to minimize the power lost to electrical resistance. The Net Bridge also prevents a surge of motor current from interfering with the digital circuitry.

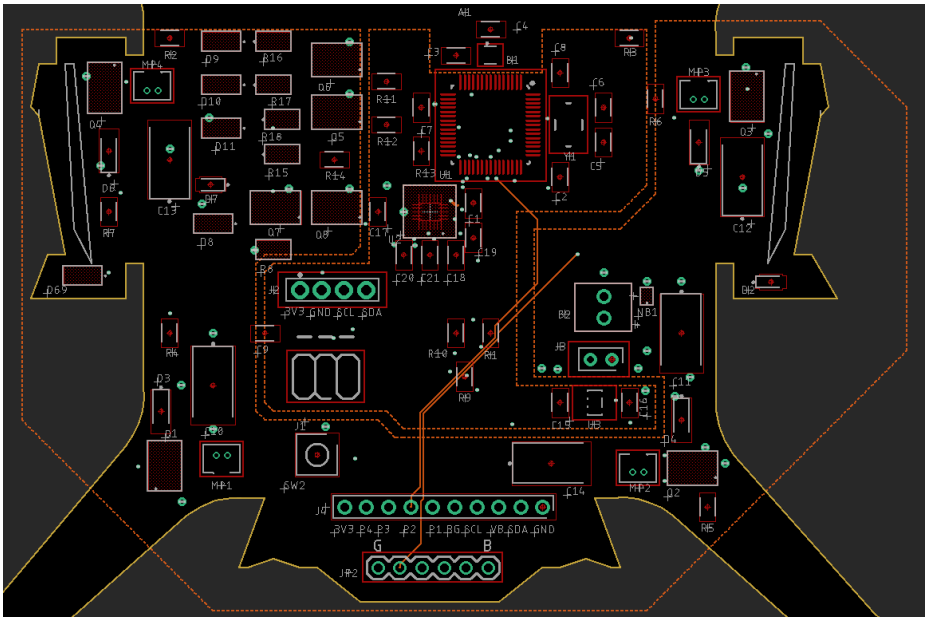


Figure 12: Battery power and 3.3*V* power planes. All of the circuitry requiring the battery power is on the outer part of the board containing the battery power plane.

Circuit Line Thickness and Net Classes

The width of a routing line is inversely proportional to its electrical resistance, and therefore the power dissipated as heat. Too high of a current will burn a thin routing line, causing irreparable damage to the PCB. For this reason, the routing lines are sorted into net classes, which each have their own rules for routing width and drill diameter. See Table 1 for more details.

Table 1: Net Class Rules

Net Class Name	Routing Line Width (mil)	Drill Diameter (mil)	Example Routing Line
Default (Signal lines)	5	7.88	SDA, SCL, RESET, PWM1
RFSIG (radio antenna)	50	7.88	ANT1
PWR	10	7.88	3V3, GND
HIGHCURRENT	30	24	VBAT, BAT_GND

IMU

The IMU was the first component placed into position on the board. This was done to ensure that the sensor was positioned at the centroid (coordinate {0,0}) of the quadcopter for an accurate reading of its orientation. Four decoupling capacitors of values 0.01 to 10µ*F* were placed close to the IMU to filter out electrical noise [7]. Headers to connect to a breakout IMU were placed near the center of the board as well, as seen in Figure 13 below.

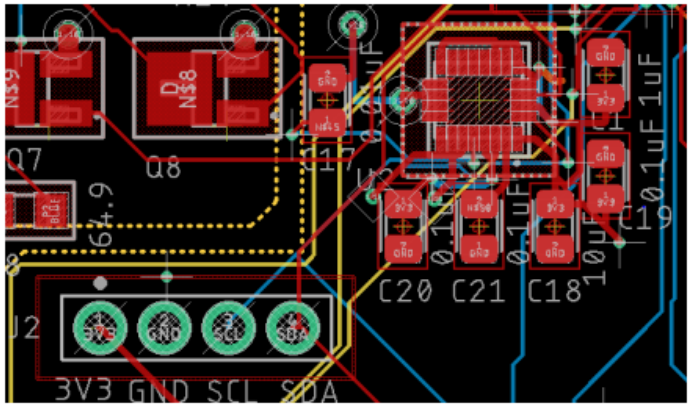


Figure 13: IMU and Breakout IMU Header Layout

Microcontroller and Oscillating Crystal

The microcontroller was the next circuit component placed on the layout, and was put near the IMU with its RFP/RFN pins pointed to the front of the quadcopter for use with the radio, as seen in Figure 14 below. Surrounding decoupling capacitors shield the microcontroller from electrical noise [7]. To the right of the microcontroller is its $16MHz$ resonating crystal, which the microcontroller uses to tell time by counting its oscillations. Both the decoupling capacitors and the crystal were placed as close to the microcontroller as possible to minimize resistance losses and signal travel time.

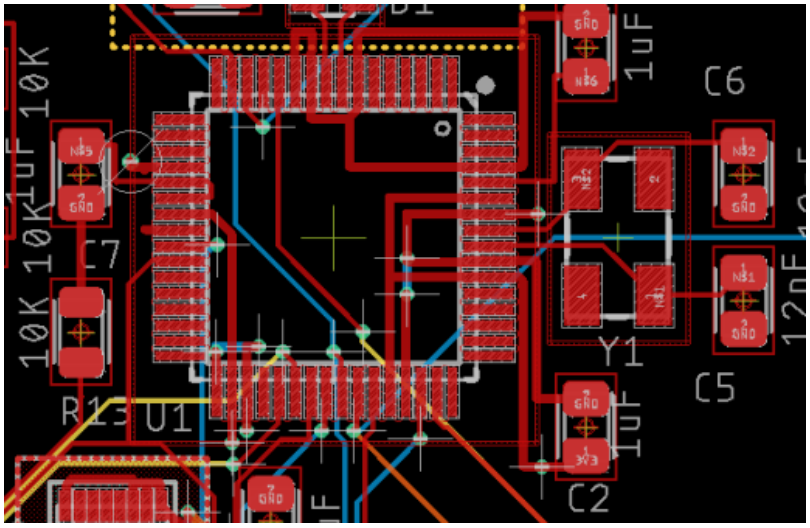


Figure 14: Microcontroller and Crystal Layout

Radio

The ATmega128RFA1 microcontroller was chosen primarily due to its RFP/RFN radio pins. These pins provide balanced signals that are mirror images of each other, one positive and one negative. The signal's message is encoded in the difference between their two voltages. This type of signal is known as a differential pair, and has the benefit of a reduction of signal noise compared to an unbalanced signal, provided that the noise affects both signals equally [8].

The balun is the 6-pin circuit component which connects the microcontroller to the radio antenna. It converts the balanced (differential) signal of the microcontroller to the unbalanced radio antenna signal. Here, "unbalanced" refers to the fact that the signal operates in relation to Ground [9].

Because the radio antenna is unbalanced, its length is required to be a quarter of the wavelength ($\frac{\lambda}{4}$) of the radio signal it is carrying. We may use the equation $\lambda = \frac{c}{fz}$ to find the wavelength, where c is the speed of light, and the radio transceiver frequency, fz, is $2.4\ GHz$. Plugging in the known values, we find that the wavelength is about $0.125\ m$, leading to an antenna length of $\frac{\lambda}{4} = 31\ mm$ [1] [10]. The radio antenna was built into the PCB as a routed signal line, as seen in Figure 15.

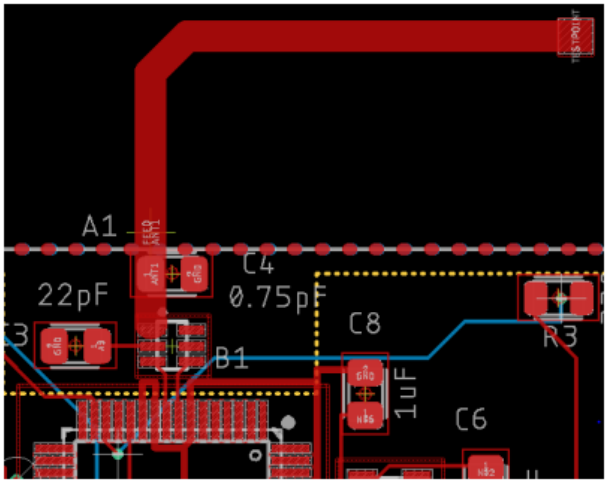


Figure 15: Radio Antenna and Balun Layout

Battery and Voltage Regulator

As seen below in Figure 16, the battery connector and voltage regulator circuitry was positioned on the edges of the battery and $3.3V$power planes, respectively, to minimize resistance losses in transferring power to each of the planes.

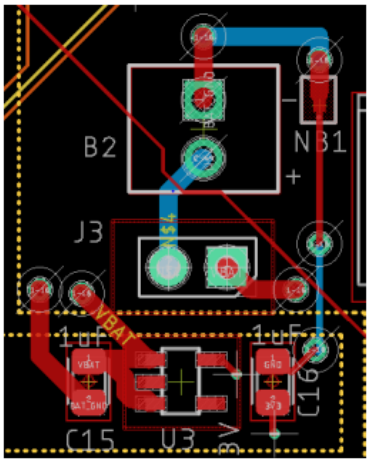


Figure 16: Voltage Regulator Layout

ISP Bootloader, FTDI Programmer, Reset Button, Signal Breakout Header

The layout of the FTDI programmer is seen in Figure 17 below. It was positioned at the bottom edge of the PCB so that its through-hole headers could be accessed for programming.

The ISP bootloader, the reset button, and the signal breakout header were all placed in easily accessible locations--not covered by an IMU breakout board or within the reach of a propeller.

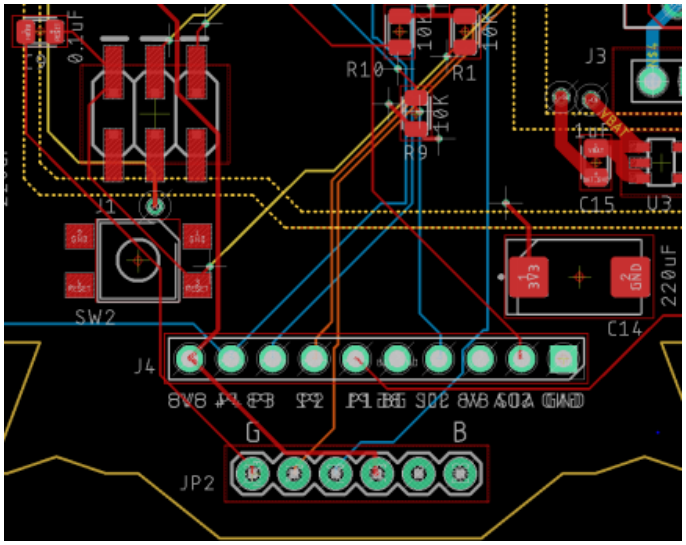


Figure 17: ISP Bootloader (top left), FTDI Programmer (Bottom), Signal Breakout Header (Center), and Reset Button (Left) Layout

LEDs

The LED circuitry was positioned starboard of the microcontroller, seen below in Figure 18. The LEDs themselves were placed from back to front in the following order: Blue, Green, Yellow, Orange, Red. As mentioned previously, the PCBs as manufactured did not allow the LEDs to be controlled by the microcontroller due to missing signal lines.

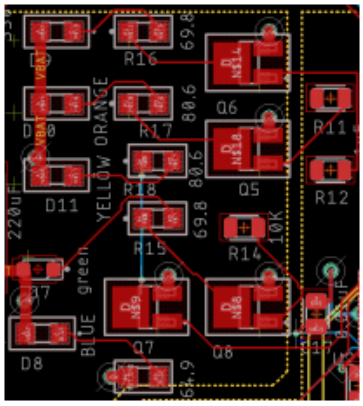


Figure 18: Rainbow Aesthetic LEDs (left column) Layout.

Motor Drivers and Quadcopter Arms

The four motor drivers and their corresponding motor pads were placed near each corner of the quadcopter so that the 80 mm motor wires could easily connect from the terminals to the arms [11]. A single full motor driver layout, including arm dimensions, is shown below in Figure 19.

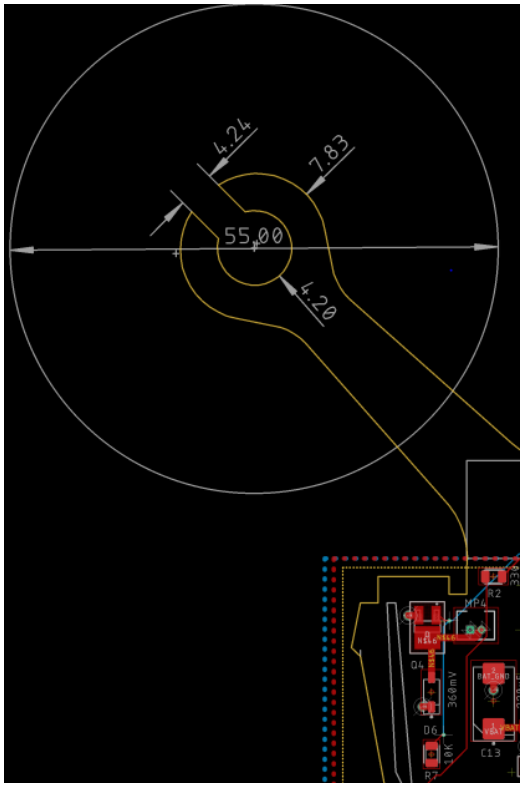


Figure 19: Motor Driver Layout and Quadcopter Arm.
The circle corresponds to the propeller reach.

Quadcopter PCB Assembly

The quadcopter PCBs were manufactured by JLCPCB, utilizing the exported EAGLE .CAM files [12]. A PCB stencil, which is a sheet of metal with holes cut into it corresponding to SMD pad locations, was also manufactured by JLCPCB. All circuit components used by the quadcopters were ordered on Digikey. See Appendix D for the Bill of Materials.

The SMDs could not be soldered by hand, for the simple reason that they required too much precision. For example, the entire IMU measures just 4mm by 4mm, and has a total of 24 pins which each need to be soldered to their corresponding copper pad without any connection between adjacent pads. Reflow soldering was therefore used to assemble the SMDs.

Reflow soldering assembly proceeded as follows. A single board was first placed on a flat surface, and leftover PCBs were taped adjacent to the board to prevent it from moving. The stencil was then positioned so that its holes aligned with the SMD pads. Leaded solder paste was then spread over the top of the stencil so that the pads were covered completely, as in Figure 20 below. Each SMD component was then carefully placed on its corresponding pads with tweezers, as pictured in Figure 21. Finally, the two quadcopters were baked in the reflow soldering oven seen in Figure 22. The reflow oven melted the leaded solder paste, causing the molten solder's surface tension to attract the component terminals to their corresponding copper pads.

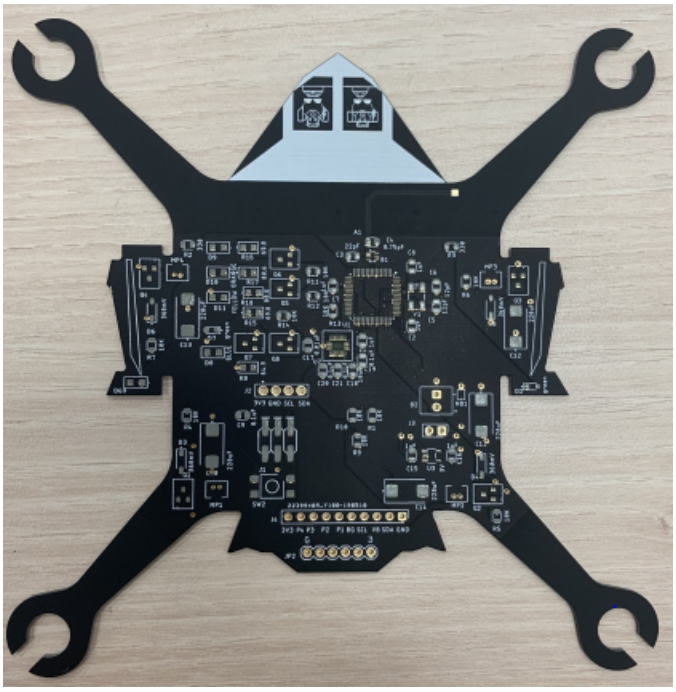


Figure 20: Pre-Assembly PCB with Applied Solder Paste

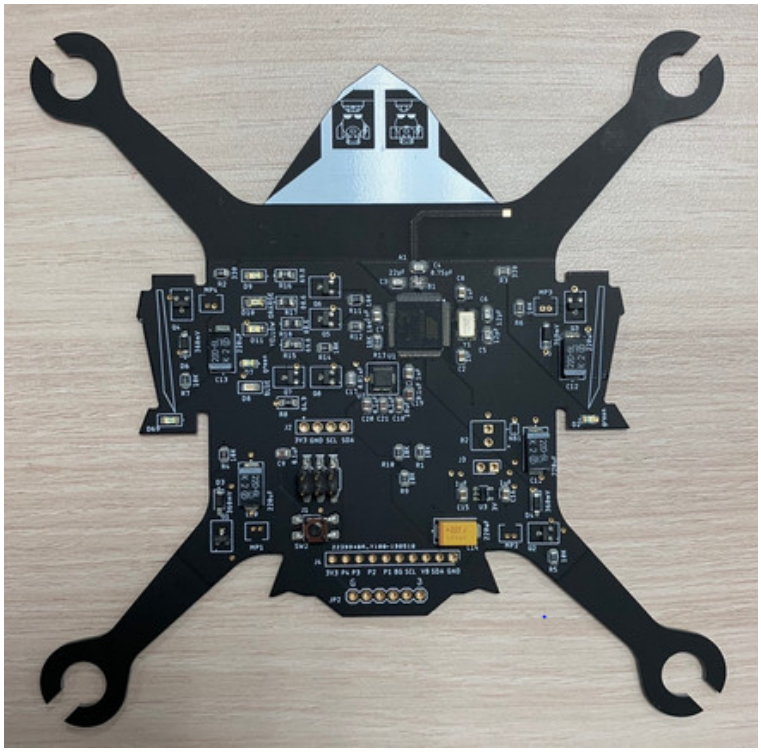


Figure 21: Post-SMD-Assembly PCB

Figure 22: Reflow Soldering Oven [13]

Once the quadcopter boards were taken out of the oven and allowed to cool off, the through-hole components were soldered on by hand, and the motors were attached to the arms using hot glue. The assembled quadcopter is pictured in Figure 23.

Once the quadcopter PCBs were assembled, connectivity between signal lines was checked using a voltmeter before powering the boards. Any connectivity would mean that there is a short in our system which could lead to a range of electrical problems, from rendering the IMU signal unreadable to frying the entire PCB. My personal quadcopter had a short between the SCL and SDA signal lines, caused by a solder bridge underneath the IMU itself. I was forced to remove the IMU to fix the short and use a breakout board attached via the IMU rescue headers. Because of this, it was decided that we would focus on getting Tom's correctly assembled quadcopter to fly for the remainder of the class.

Figure 23: Fully Assembled Quadcopter

Remote Control Hardware

The remote which controls the quadcopter, shown in Figure 24, was provided by Prof. Swanson. It contains the same microcontroller and radio circuitry as the quadcopter, so the two may be programmed to communicate with one another. Several buttons and a rotary knob on the remote PCB are intended for tuning the control system and trim of the quadcopter, the information of which may be displayed on an LED screen. The remote PCB also contains headers which connect to analog gimbals to control the throttle, pitch, roll, and yaw rate of the quadcopter.

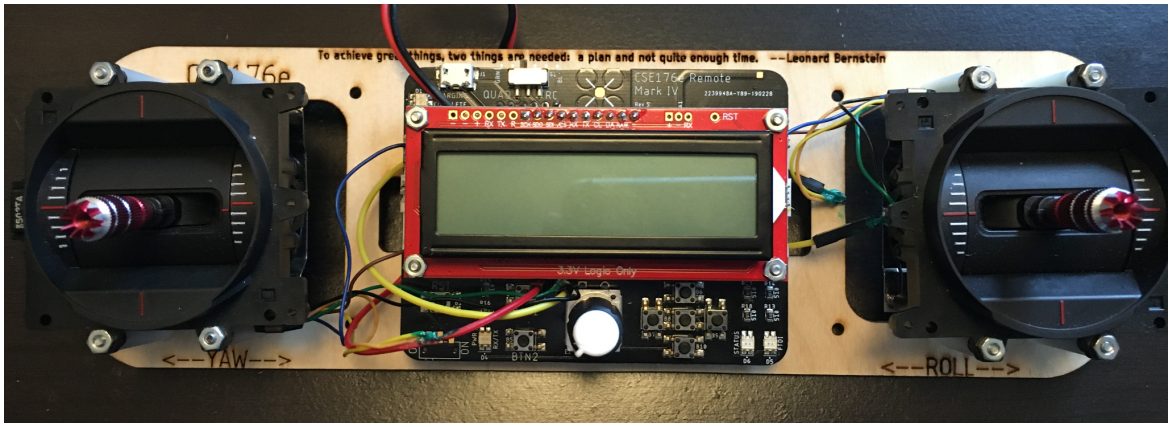


Figure 24: Assembled Remote Control

Firmware

Bootloading the Microcontroller

Once the quadcopter was assembled and checked for electrical shorts, its microcontroller was bootloaded using a Pocket AVR Programmer connected to the 6 pin ISP header [14]. The Programmer was used to install USBtiny for our specific microcontroller, via the CPU command line program AVRDude. USBtiny gives the microcontroller Arduino compatibility by allowing it to communicate to the CPU via USB serial, as seen by the architecture diagram in Figure 25 [4][15][16].

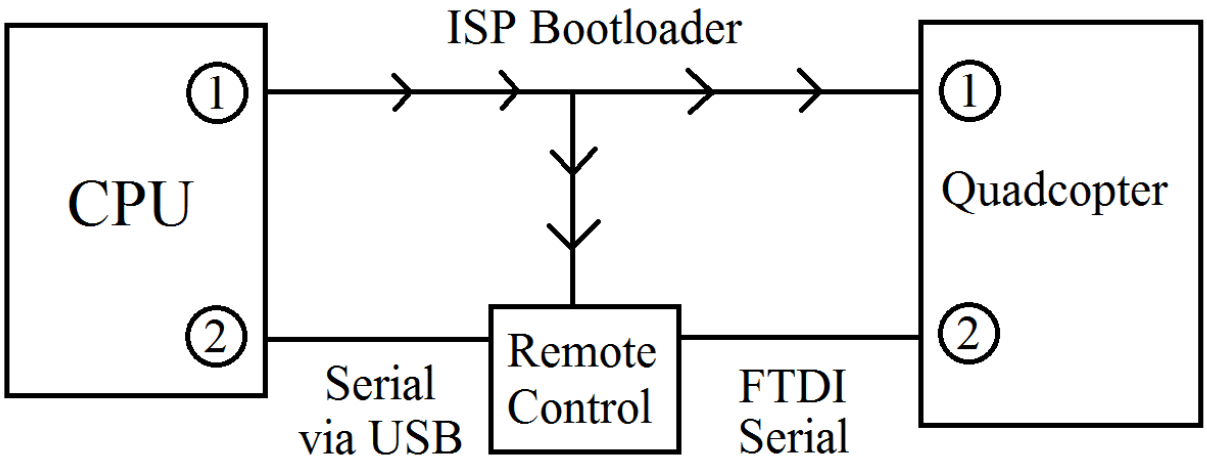


Figure 25: Programming Architecture Diagram

Programming via USB

Once bootloaded via ISP, the quadcopter and remote control microcontrollers may be programmed using USB. The remote control PCB has a Micro B USB 2.0 plug, and an FT232RL chip to convert the differential USB signal to unbalanced serial. The unbalanced serial lines are connected to the receiving (RX) and transmitting (TX) microcontroller pins for programming the microcontroller and sending data to the CPU, respectively. A simple switch connected to the Select Input pins of a multiplexer determine whether the unbalanced serial is routed to the remote microcontroller, or is routed to the quadcopter microcontroller via a 6-pin FTDI cable [17][18][19]. See Figure 26 for a picture of the quadcopter being programmed.

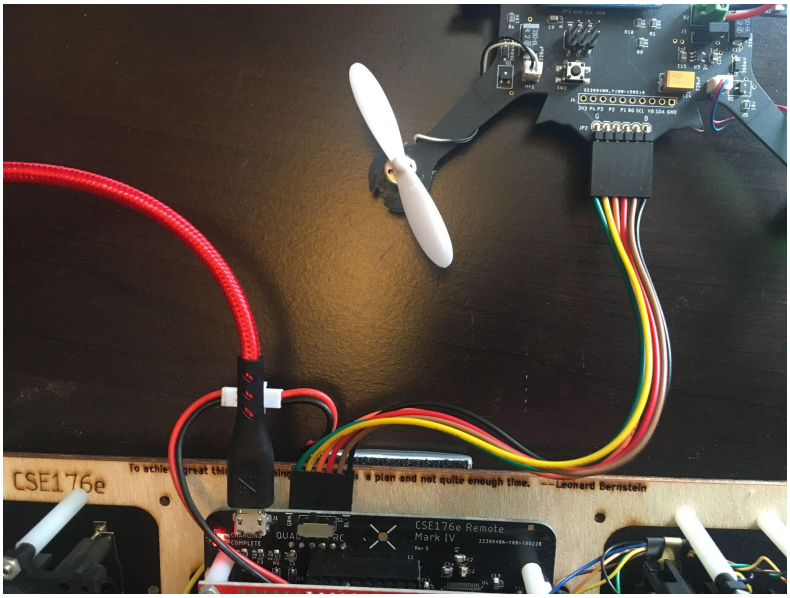


Figure 26: Programming The Quadcopter

Remote Control Firmware

The microcontrollers of the remote and the quadcopter were programmed in C++ programming language using the Arduino IDE. All firmware for this project is available under GNU GPL 3 at: https://github.com/rketch/quadcopter. See Appendix E for the remote and quadcopter firmware LaTeX Doxygen output.

Upon powering up the remote control, it enters into a setup function to initialize the LCD screen, radio, gimbal pins, rotary knob, and buttons. It also reads the control system and trim values saved in the microcontroller's EEPROM. By default, the remote initializes in the disarmed mode for safety, meaning that it prevents the quadcopter from turning on its motors. The firmware then enters into an endless loop function.

The loop function first reads the user gimbal values. These values are checked to see whether the gimbals are pointed down and away from each other, which is the user command that arms the quadcopter. For safety purposes, this gimbal positioning requires two independent motions which are unlikely to happen randomly. The information for tuning either the control system or the trim is then displayed to the LCD screen, and the remote control buttons are checked for user input. The control system and trim values are then stored in a data structure which also includes the current throttle, pitch, roll, and yaw rate commands, and whether the quadcopter is armed or disarmed. This data structure is then sent over radio to the quadcopter.

The state diagram for the remote firmware is seen below in Figure 27. When powered, the remote allows the user to tune its PID control system. Pressing Button 2 on the remote control switches to tuning the quadcopter trim. Arming the quadcopter keeps the same tuning state but alerts the user that the quadcopter is armed by displaying an "A" on the LCD screen.

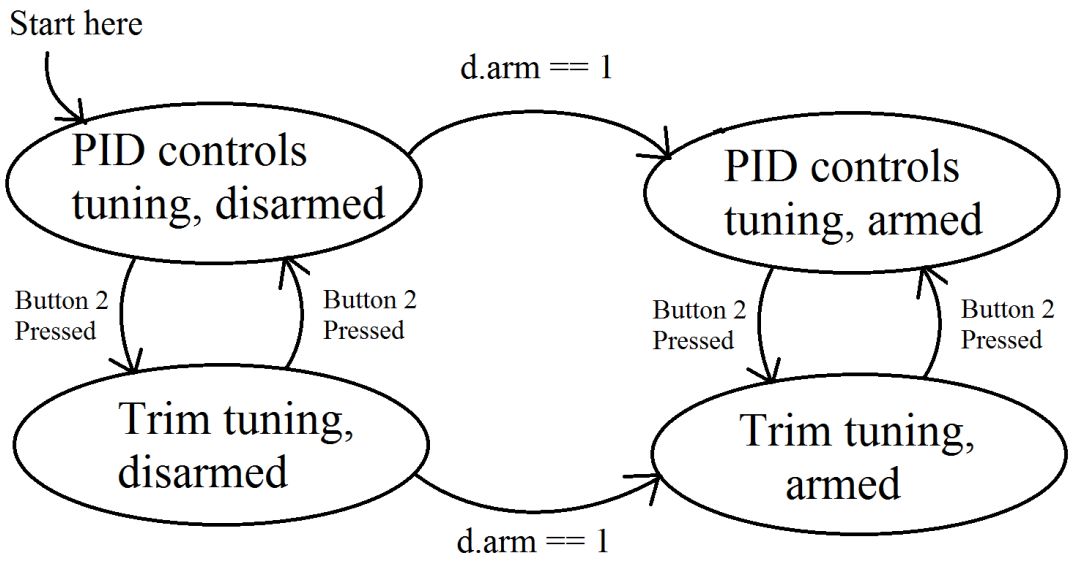


Figure 27: State Diagram of the Remote

Within the control system tuning state, there exist sub-states which determine what aspect of the control system is currently being tuned. The state diagram for tuning the control system is seen below in Figure 28. When powered on, the user is by default allowed to tune the Proportional value of the quadcopter's pitch angle, as shown by the pointing arrows in Figure 29. The orientation being tuned is switched via the left button on the D-pad, and the PID control value being tuned is cycled with the right button.

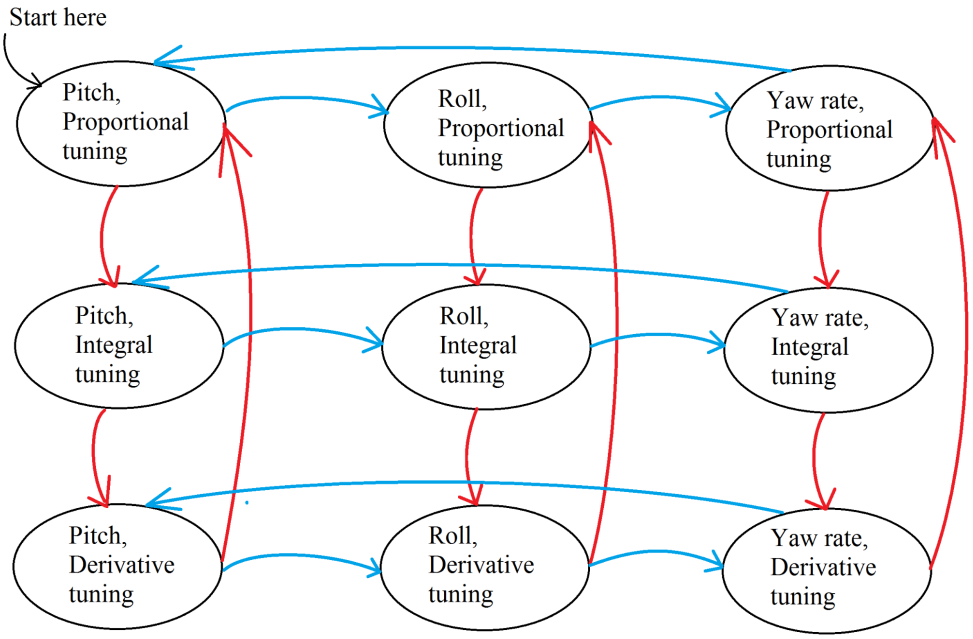


Figure 28: Control System Tuning State Diagram. The state is advanced via the blue or red arrow if the left or right button is pressed, respectively.

The control system number being tuned is split into its integer value and its decimal value with precision to the thousandths place. Each of the two is tuned independently, with lines above the number to tell the user which value is currently being tuned. Pressing the D-pad center button switches between the decimal and integer tuning values. A rotary knob may be used to increment the displayed value by 1, while pressing the up button on the D-pad increases the current value by 100. Pressing the tactile knob will trigger a button which resets the current displayed value, and pressing the down button saves all current control system and trim values to EEPROM. If the quadcopter is not in its armed state, the down button will also enter into a script which allows the user to calibrate the gimbals.

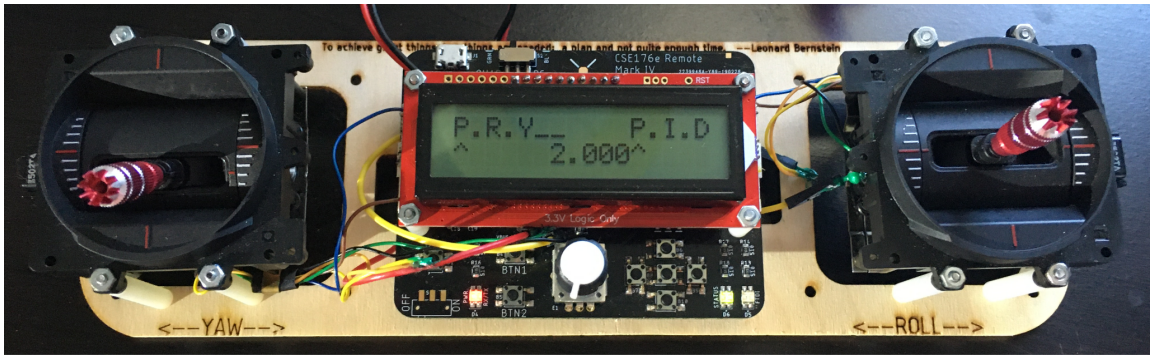


Figure 29: PID Tuning Menu. An “A” would appear on the screen if the quadcopter was armed

Tuning the quadcopter trim is a more straightforward process than the control system. A quadcopter with a correctly tuned control system may drift to one side when throttled up. Simply pressing any directional button on the D-pad will lessen the quadcopter drift in that direction. Pressing the center button will store the control system and trim values in EEPROM, and pressing Button 2 will return the user to the control system tuning menu. See Figure 30 for the remote screen while trim tuning, and the **Control System** section for more discussion about trim and PID control system functionality.

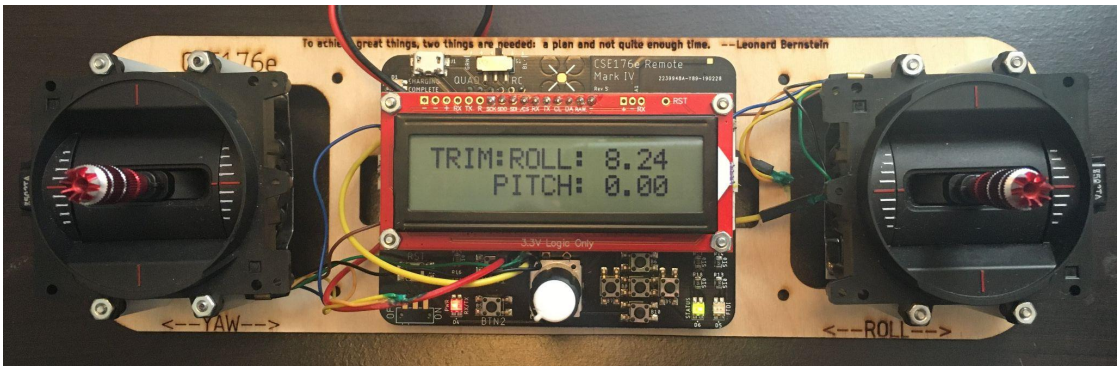


Figure 30: Trim Tuning Menu. An "A" would appear on the screen if the quadcopter was armed

Quadcopter Firmware

Prof. Swanson provided a Flight Control Board to our team for prototyping firmware in parallel with developing our quadcopter PCB design. The Flight Control Board was a quadcopter with similar overall functionality and the same microcontroller as our quadcopter design, and may be seen testing IMU filtering in Figure 31. Once our quadcopter was designed, manufactured, and assembled, it replaced the Flight Control Board for firmware testing.

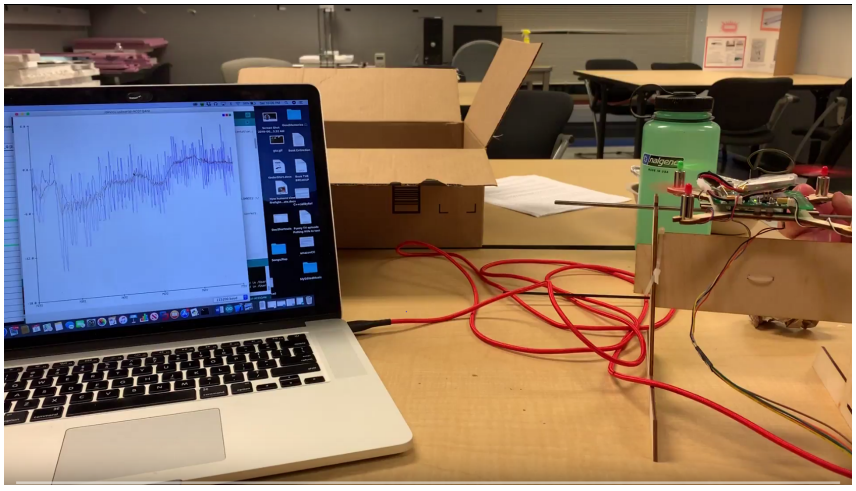


Figure 31: Flight Control Board on its Test Stand

Mirroring the remote, the final quadcopter firmware consisted of the standard Arduino setup and forever loop functions. The setup function initializes the IMU register values to ensure the correct filtering is used and that timely orientation data is fed to the microcontroller. The radio and motors are also initialized, and a timer is started for use in the control system and IMU filtering.

After setup, the infinite loop is entered. At the beginning of each loop, a function is called which reads and verifies the remote data communicated over the radio. The radio data tells the quadcopter whether to be in its armed or disarmed state, and is initially set to disarmed when powering on, as seen in Figure 32. Regardless of state, the quadcopter orientation is then read from the IMU utilizing a complementary filter, and a PID control system function adjusts the PWM throttle value to be sent to each individual motor.

When the quadcopter first receives the command from the remote to arm itself, it assumes that it is on a level surface and zeroes the current IMU values. It will then engage its motors with the sum of the commanded throttle value and the control system output for each motor. The time step is then updated and the firmware loops back to receive more radio data.

If the quadcopter is disarmed or the commanded throttle is zero, the time step is updated and the loop is repeated without writing the PWM values to the motors.

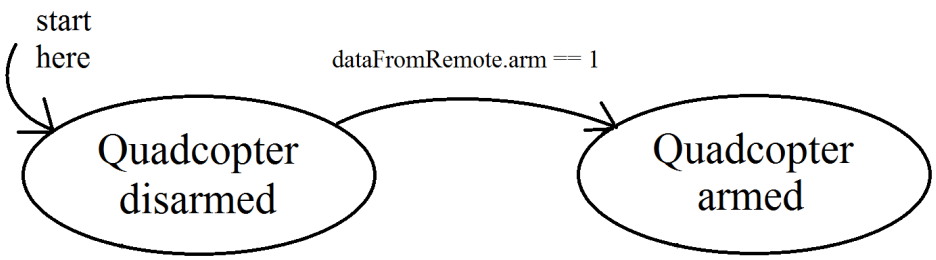


Figure 32: Quadcopter State Diagram

IMU Sensor and Orientation

Quadcopter Orientation Overview

The first step to achieving stable flight is having the quadcopter sense its orientation using its onboard Inertial Measurement Unit (IMU). Figure 33 shows the coordinate axes of the quadcopter, as well its pitch {θ}, yaw {ψ}, and roll {φ} angles. The arrows tell each angle's positive orientation; the quadcopter's nose pointed upward, port (white propeller side) pointed upward, and a clockwise rotation as seen from above correspond to a positive pitch, roll, and yaw, respectively [20].

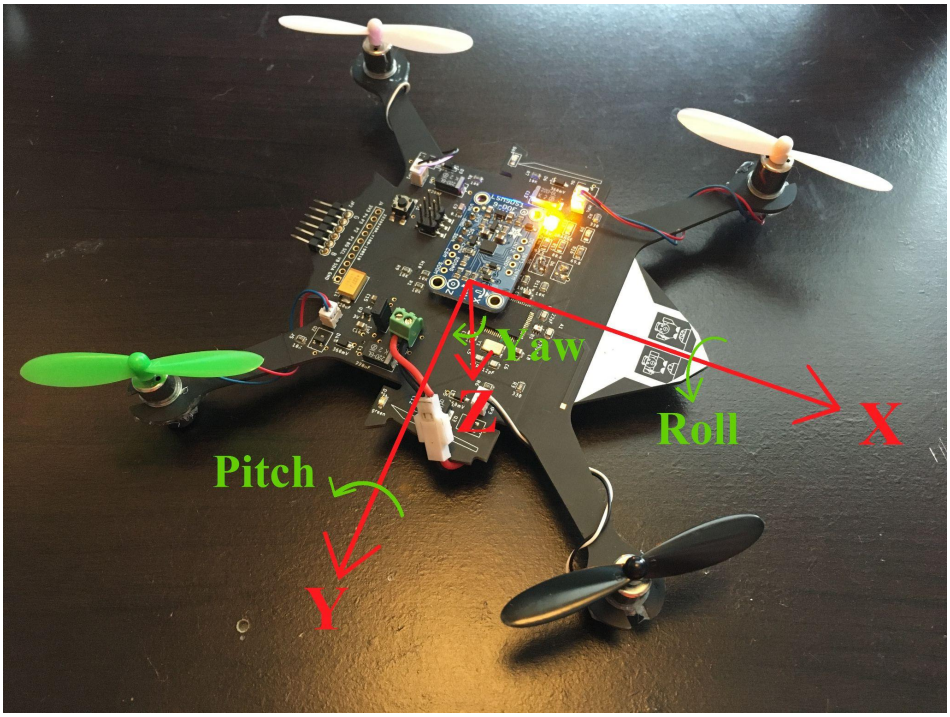


Figure 33: Quadcopter Coordinate Axes

All sensors have limitations for use, so to get an accurate orientation we use a combination of two sensors: an accelerometer and a gyroscope.

Accelerometer

The accelerometer reads three orthogonal acceleration vectors $\{A_x, A_y, A_z\}$, and may use the gravity vector of the Earth to read the orientation angles of the quadcopter [21]:

$$Roll \;=\; \varphi_{A_n} = tan2^{-1}\left(\frac{A_y}{A_z}\right)$$

$$Pitch \;=\; \theta_{A_n} = tan^{-1}\left(\frac{-A_x}{A_y\, sin\left(\varphi_{A_n}\right) + A_z\, cos(\varphi_{A_n})}\right)$$

Note that we do not care about the yaw angle, as it is more useful to control the yaw rate for a non-autonomous, remote controlled drone. The accelerometer is useful for telling the quadcopter's downward direction, which may be viewed as a low frequency signal. Unfortunately, the four motors transfer high-frequency vibrations to the body of the quadcopter, which is detectable by the IMU as a high frequency acceleration signal. We therefore utilize a low pass filter on the accelerometer signal, which takes the discrete-time form [22]:

$$Low\ Pass\ Pitch\ Filter \;=\; \theta_{A,\,Filt_n} = (1 - \alpha)\,\theta_{A_n} + (\alpha)\,\theta_{A,\,Filt_{n-1}}$$

This equation, and the other filtering equations which only specify the pitch angle, are also applicable for the roll orientation angle. The subscript "n" refers to the time step.

A discrete-time low or high pass filter utilizes a Filter Ratio, which allows the designer to control how much low or high frequency signal to filter. It is of the form:

$$Filter\ Ratio \;=\; \alpha \;=\; \frac{\tau}{\tau + dt} = \frac{1}{dt\,\omega_c + 1}$$

Where dt is the sampling period of the IMU, $\tau$is the time constant of the system, and ω_c is the filter's cutoff frequency, or the frequency at which half of the signal's power is attenuated [23] [24]. This was chosen to be 10 Hz, or the estimated rate at which a quadcopter would lose control if it were suddenly disconnected from power. The firmware loop frequency was roughly 100 Hz, leading to a filter ratio of 0.91.

Gyroscope

A gyroscope senses angular velocity. It is useful for accurately determining high-frequency angular changes of the quadcopter, but must first be integrated with respect to time to be usable. This is estimated via a zeroth-order hold of the gyroscope reading at the current time step [25]:

$$Pitch\text{: } G_{\theta_n} = \frac{d\theta}{dt} \Rightarrow \theta_n = \theta_{n-1} + \int_{t_{n-1}}^{t_n} G_{\theta_n}\, dt \approx \theta_{n-1} + G_{\theta_n}(t_n - t_{n-1})$$

The zeroth-order estimation leads to inaccuracies which manifest themselves as a near-constant "walk", or increase or decrease of angle. For this reason, a discrete-time high pass filter is used [26]:

$$High\ Pass\ Pitch\ Filter = \theta_{G,Filt_n} = \alpha(\theta_{G,Filt_{n-1}} + \theta_{G_n} - \theta_{G_{n-1}})$$

Complementary Filter

If the gyroscope's low pass filter and the accelerometer's high pass filter use the same filter ratio, their sum will lead to an overall gain of 1. A combined filter of this type is known as a complementary filter:

$$Complementary\ Pitch\ Filter = \theta_{C,Filt_n} = \theta_{A,Filt_n} + \theta_{G,Filt_n}$$

It may be arranged to take the more direct form [27]:

$$Complementary\ Pitch\ Filter = \theta_{C,Filt_n} = \alpha(\theta_{C,Filt_{n-1}} + G_{\theta_n}\, dt) + (1-\alpha)\,\theta_{A_n}$$

The pitch and roll outputs of the complementary filter are then used in the control system. The yaw angular velocity is used directly as given by the gyroscope.

Kalman Filter

Another form of state estimator is the Kalman filter. In theory, it reduces Gaussian noise of a system by minimizing the square of the error between the system's actual states and the Kalman filter's estimated states [28][29]. For a discrete-time system, the state and output equation are:

$$x_n = Ax_{n-1} + Bu_n + w$$

$$y_n = Cx_{n-1} + q$$

where w and q are the input disturbance to the system and measurement noise, respectively [29].

The Kalman filter has two stages: the unfiltered predictor, and the correction, which is filtered by the value β (not to be confused with the predictor-corrector method of solving ODEs).

$$Predictor: \hat{x}_n^- = A\hat{x}_{n-1} + Bu_n$$

$$Corrector: \hat{x}_n = \hat{x}_n^- + \beta(y_n - C\hat{x}_n^-)$$

The $\hat{}$ indicates that this is the Kalman estimated state.

We may combine the predictor and corrector stages to receive:

$$Combined: \hat{x}_n = A\hat{x}_{n-1} + Bu_n + \beta[y_n - C(A\hat{x}_{n-1} + Bu_n)]$$

For the quadcopter orientation pitch angle from the accelerometer and gyroscope sensors, our system has the state and output equations as follow:

$$\theta_n = \theta_{n-1} + G_{\theta_n} dt + w$$

$$\theta_{A_n} = \theta_{A_{n-1}} + q$$

The predictor method utilizes the gyroscope, and has the same form as the zeroth order integrated hold:

$$Predictor: \hat{\theta}_n^- = \hat{\theta}_{n-1} + G_{\theta_n} dt$$

The corrector incorporates the accelerometer reading:

$$Corrector: \hat{\theta}_n = \hat{\theta}_n^- + \beta(\theta_{A_n} - \hat{\theta}_n^-)$$

Combining the predictor and corrector:

$$Combined: \hat{\theta}_n = \hat{\theta}_{n-1} + G_{\theta_n} dt + \beta[\theta_{A_n} - (\hat{\theta}_{n-1} + G_{\theta_n} dt)]$$

Rearranging:

$$\theta_{K,Filt_n} = (1 - \beta)(\theta_{K,Filt_{n-1}} + G_{\theta_n} dt) + \beta\,\theta_{A_n}$$

If we choose $\beta = (1 - \alpha)$, we obtain the same transfer function as the complementary filter, demonstrating that the complementary and Kalman filters are equivalent for this system [30]:

$$Kalman\ Pitch\ Filter = \theta_{K,Filt_n} = \alpha(\theta_{K,Filt_{n-1}} + G_{\theta_n} dt) + (1 - \alpha)\,\theta_{A_n}$$

Control Systems

Control Blocks

The following applies for pitch and roll angles, and the yaw angular velocity. Each of these has its own independent PID values.

The quadcopter orientation is directly affected by the motor throttle set by the microcontroller. This relationship may be visualized as an open-loop control block, as seen in Figure 34.

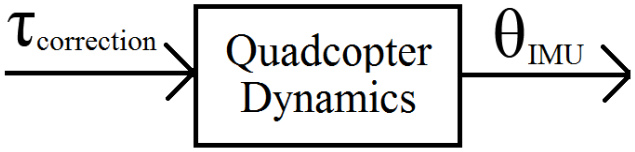


Figure 34: Open Loop Block Diagram

The user tells the quadcopter which orientation to have using the remote. This user orientation is known as the set angle. If the set angle differs from the "true" IMU angle of the quadcopter, the quadcopter must correct its orientation by the difference of these values, or the error angle. The quadcopter executes the correction maneuver by increasing the throttle of some motors, and decreasing the throttle of other motors. For example, if the pitch angle of the quadcopter was set higher than the current angle read by the IMU, the front two motors would have to increase throttle and the back two motors would decrease throttle to compensate. The exact amount by which we vary the motor throttle for a given error angle is determined by the PID controller.

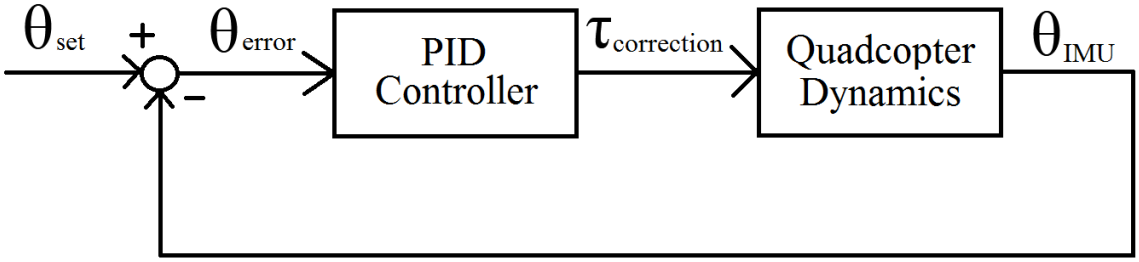


Figure 35: Closed Loop Block Diagram

PID Controller Overview

The control system used in the quadcopter was a discrete-time proportional, integral, and derivative (PID) controller implemented in the microcontroller's firmware. While each of these three types of controllers are dependent on the error angle signal, only the proportional controller is solely dependent on its current value. Instead, the integral controller is dependent on the total time history of the error angle, and the derivative controller is dependent on the current rate of change of the error angle. Each of the three controllers has a gain value, K, which affects the motor throttle's exact dependence on each controller type [31]. These behaviors are demonstrated in their corresponding equations in Table 2. The sum of the three throttle corrections was then added or subtracted to each corresponding motor throttle.

Table 2: PID Controller Equations [28]	
Proportional	$\tau_{correction,\,P,\,n} = K_P\,\theta_{error,\,n}$
Integral	$\tau_{correction,\,I,\,n} = K_I \sum_{j=0}^{n} \theta_{error,\,j}\,dt$
Derivative	$\tau_{correction,\,D,\,n} = K_D \frac{\theta_{error,\,n} - \theta_{error,\,n-1}}{dt}$

The effects of each component in a correctly tuned PID controller are best discussed in relation to a step response. In general, a step response is the system's response to a sudden change in input. For the quadcopter, this corresponds to how its IMU orientation would be affected by a sudden change in the set angle. A proportional controller quickly converges the system response with the input signal. An integral controller ensures that over time any remaining error dissipates, leading to an exactly matched IMU and set orientation angle. Finally, a derivative controller minimizes overshoot by acting in opposition to quick changes in the system.

Ad-hoc PID Tuning

While there exist arguably superior control theory techniques, such as lead-lag, state feedback linearization, or LQR, a PID controller has the advantage of easy ad-hoc tuning, or changing controller gain based on the current system response "feel" [32] [33]. The quadcopter system response for too much or too little proportional, integral, or derivative gain are shown in Table 3.

Table 3: System Response due to Varying PID Gains			
	Proportional	Integral	Derivative
Too much	Overshoot. Mid frequency oscillations	Integrator windup. Low frequency oscillatory overshoot	Noise amplification. High frequency Oscillations
Not enough	No convergence to setpoint	Slow or no convergence to setpoint	Underdamped overshoot.

The yaw control system was tuned first by tying fishing line on each motor arm, tying the four lines together, and increasing the proportional and integral values until the yaw rate was easily controllable.

Finding a suitable PID controller for pitch and roll was an iterative process which required close following of the rules tabulated in the above system response guide, as well as patience and a certain amount of luck. The quadcopter was mounted on a test stand, seen above in Figure 31, which was provided by Prof. Swanson to limit the quadcopter's degrees of freedom. However, this directly changed the pitch and roll equations of motion of the quadcopter, meaning that the PID values which controlled the quadcopter on the test stand needed to be re-tuned to stabilize the free flying quadcopter. The PID values which stabilized Tom's quadcopter are shown below in Table 4.

Table 4: Stabilizing PID values for Tom's Quadcopter			
	Proportional	Integral	Derivative
Pitch Angle	1.8	0.02	0.516
Roll Angle	1.8	0.02	0.428
Yaw Angular Velocity	6	1.01	0

Trim

One final tuneable control necessary to fly the quadcopter is its trim, or the orientation at which the pitch and roll are zero. The reason for this is that the zeroed IMU angles might be slightly off from the stabilized, motionless quadcopter orientation. This was implemented in two ways. First, when the quadcopter is armed, its current pitch and roll are subtracted from subsequent readings. Second, pressing a button on the remote control while tuning the PID control system will bring the user to a separate menu for tuning the trim. The trim can then be changed using the four directional buttons. See Figure 30 for the trim tuning menu as it appears on the remote screen.

Flight

With all of the hardware and firmware complete and the control system and trim properly tuned, the quadcopter can fly. More specifically, Tom's quadcopter was able to fly for five to ten seconds before it would hit a wall or the ceiling, which would end the flight. This limitation in flight time was more due to our lack of previous quadcopter piloting experience rather than any issues with the system, although a limitation of the throttle PWM may have prevented a few crashes. A demonstration of flight is viewable at the end of this video: https://tinyurl.com/quadflopter.

As for my quadcopter, at the end of the class it was completely unflyable, demonstrated by the slow motion clip in the above linked video. The reasons for the disparity between our two quadcopters are mainly due to the addition of the breakout IMU. Namely, the breakout IMU was not able to be mounted with all 4 corners to the quadcopter, increasing the amount of vibration noise affecting the accelerometer. It also increased the overall mass of the quadcopter, affecting the control system. Lastly, the IMU orientation on the breakout board was mounted at a 90° angle as compared to the surface mounted IMU, requiring a re-mapping of the controls.

Since the end of the class, I have spent time stabilizing the quadcopter by soldering the IMU down to prevent it from being dislodged, limiting the motor vibratory noise affecting the accelerometer using rubber dampeners, and tuning the control system. As of the time of writing in May 2021, my quadcopter can currently glide along a smooth surface such as a tile floor, and is pitch, roll, and yaw controllable.

Conclusion

All four pillars of mechatronic design: firmware, electronics, controls, and mechanical systems were necessary foundations for the quadcopter drone. A daunting primary objective, flight in ten weeks proved doable but the temptation to improve any pillar still exists.

I am currently working on tuning the control system for my quadcopter using a combination of ad-hoc and a MATLAB Simulink model. I will update this report as I make progress.

A future goal for the project is to rewrite the firmware for both the remote control and quadcopter to include task-oriented functions. This has the advantage of being able to choose when to run each task function individually with its own timing, making more efficient use of the microcontroller's computing abilities.

I would also like to re-design the entire quadcopter using lessons learned in this iteration. A rough laundry list of proposed changes is as follows: switching from the current DC brushed motors to brushless motors, not making the PCB the frame of the quadcopter, allowing an IMU breakout board to be mounted via screws to the frame, adding a dedicated battery holding assembly, upping the battery power and voltage, using a more powerful microcontroller, adding at least 1 camera for position sensing and autonomous flight, and perhaps switching to a premade hobbyist remote control.

My first foray into mechatronics consisted of a balancing robot. My sophomore project was a naturally unstable yet controllable flying machine. Control systems engineering truly proves that any sufficiently advanced technology and magic are one and the same.

Appendix A: References

[1] *ATmega128RFA1*. Atmel, ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8266-MCU_Wireless-ATmega128RFA1_Datasheet.pdf.

[2] *LSM9DS1 Datasheet*. ST, www.st.com/resource/en/datasheet/lsm9ds1.pdf.

[3] Administrator. "FET as a Switch: Working of MOSFET or JFET as a Switch." *Electronics Hub*, 31 May 2019, www.electronicshub.org/fet-as-a-switch/.

[4] "AVR Tutorial - Choosing a Programmer." *Ladyadanet Blog RSS*, www.ladyada.net/learn/avr/programmers.html.

[5] Sachin. "Printed Circuit Board (PCB) Materials." *Printed Circuits LLC*, Printed Circuits LLC, 2 Nov. 2020, www.printedcircuits.com/printed-circuits-materials/.

[6] *Through-Hole Versus SMD Components*. Vishay, www.vishay.com/docs/45242/throughholevssmdcomponents.pdf.

[7] *Capacitors*, learn.sparkfun.com/tutorials/capacitors/application-examples.

[8] "The Why and How of Differential Signaling - Technical Articles." *All About Circuits*, www.allaboutcircuits.com/technical-articles/the-why-and-how-of-differential-signaling/.

[9] Notes, Electronics. "What Is a Balun: RF Antenna Balun." *Electronics Notes*, www.electronics-notes.com/articles/antennas-propagation/balun-unun/what-is-rf-antenna-balun.php.

[10] Notes, Electronics. "Quarter Wave Vertical Antenna." *Electronics Notes*, www.electronics-notes.com/articles/antennas-propagation/vertical-antennas/quarter-wave-vertical.php.

[11] "8pcs 8520 Motor 8.5x20mm 15000KV Brushed Motors." *Amazon*, www.amazon.com/8-5x20mm-15000KV-Coreless-JST-1-25-Connector/dp/B07CFQMF1M.

[12] "PCB Prototype & PCB Fabrication Manufacturer." *JLCPCB*, jlcpcb.com/.

[13] Hasegawa, David. "PCB Reflow Oven". https://photos.google.com/share/AF1QipN9F1jDYG9cubRXA-CwWNmIePdVq3RI0pVtxd35Kb-5K7d3aRKYkmTJEXwJQgeCqA/photo/AF1QipNKU5nrpHLrn9KLGKl-BQNUg4QJ0XB9_hSVIN0q?key=cHRpcUZwaGp2eWJNbW5EVUltc3VJQ0NNVlZvN3FB

[14] *Pocket AVR Programmer Hookup Guide*, learn.sparkfun.com/tutorials/pocket-avr-programmer-hookup-guide/programming-via-arduino.

[15] *AVRDUDE - AVR Downloader/UploaDEr*, www.nongnu.org/avrdude/.

[16] *USBtiny*, dicks.home.xs4all.nl/avr/usbtiny/.

[17] "USB Pinout, Wiring and How It Works!" *ElectroSchematics.com*, 27 Mar. 2020, www.electroschematics.com/usb-how-things-work/.

[18] *FT232R Datasheet*. FTDI Chip, www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf.

[19] *NX3L2467 Datasheet*. NXP, www.nxp.com/docs/en/data-sheet/NX3L2467.pdf.

[20] "Aircraft Rotations." *NASA*, NASA, www.grc.nasa.gov/www/k-12/airplane/rotations.html.

[21] Non-Volatile Systems Laboratory. "NVSL/QuadClass-AHRS." *GitHub*, github.com/NVSL/QuadClass-AHRS.

[22] "Low-Pass Filter." *Wikipedia*, Wikimedia Foundation, 3 May 2021, en.wikipedia.org/wiki/Low-pass_filter#Discrete-time_realization.

[23] Colton, Shane. *The Balance Filter*. Massachusetts Institute of Technology, docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxteWltdWVzdGltYXRpb25leHBlcmllbmNlfGd4OjY1Yzk3YzhiZmE1N2M4Y2U.

[24] *Cutoff Frequency*. University of Massacusetts Lowell, www.uml.edu/docs/First-Cutoff_tcm18-190085.pdf.

[25] Van de Maele, Pieter-Jan. "Getting the Angular Position from Gyroscope Data." *Pieter-Jan*, www.pieter-jan.com/node/7.

[26] "High-Pass Filter." *Wikipedia*, Wikimedia Foundation, 11 May 2021, en.wikipedia.org/wiki/High-pass_filter#Discrete-time_realization.

[27] Van de Maele, Pieter-Jan. "Reading a IMU Without Kalman: The Complementary Filter." *Pieter-Jan*, www.pieter-jan.com/node/11.

[28] "The Kalman Filter [Control Bootcamp]." *YouTube*, YouTube, 6 Feb. 2017, www.youtube.com/watch?v=s_9InuQAx-g.

[29] "Kalman Filter." *Massachusetts Institute of Technology*, ocw.mit.edu/courses/mechanical-engineering/2-154-maneuvering-and-control-of-surface-and-underwater-vehicles-13-49-fall-2004/lecture-notes/lec20.pdf.

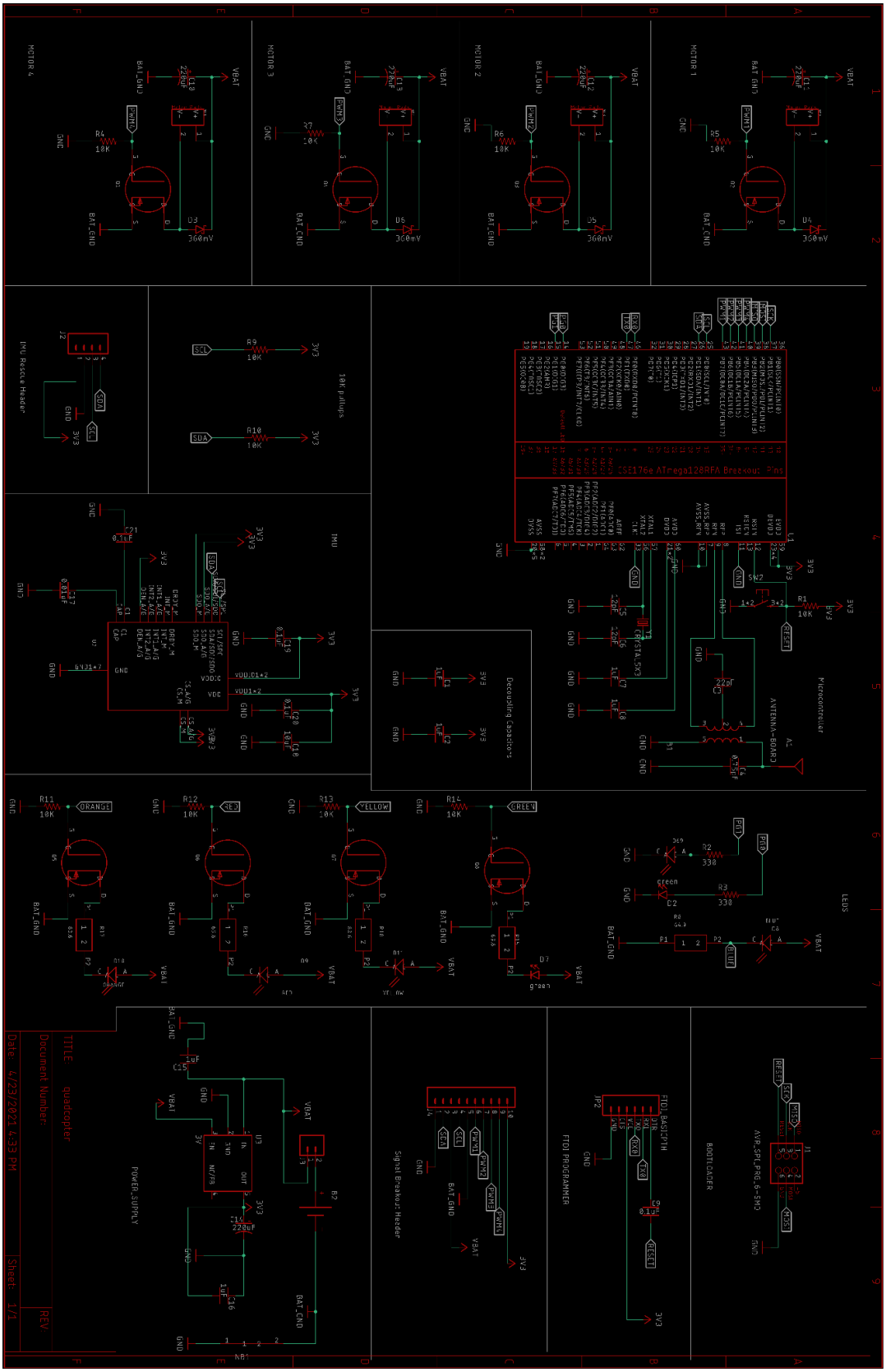
[30] "IMU Data Fusing: Complementary, Kalman, and Mahony Filter." *OlliW's Bastelseiten*, www.olliw.eu/2013/imu-data-fusing/#chapter21.

[31] "PID Controller." *Wikipedia*, Wikimedia Foundation, 19 May 2021, en.wikipedia.org/wiki/PID_controller#Discrete_implementation.

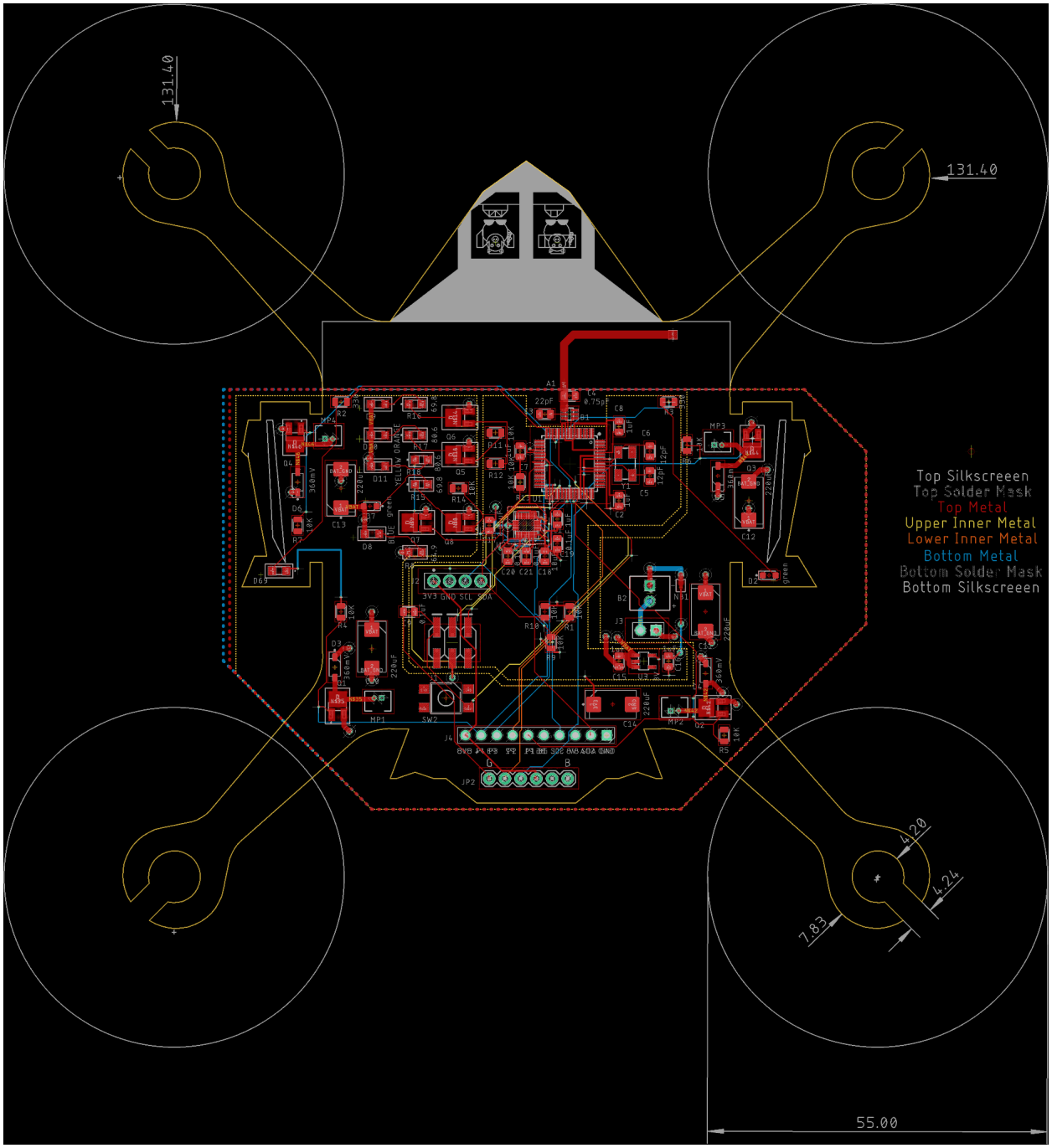
[32] "PID Controller." *Wikipedia*, Wikimedia Foundation, 19 May 2021, en.wikipedia.org/wiki/PID_controller#Manual_tuning.

[33] Bewley, Thomas. "Chapter 19: Classical Control Systems." Numerical Renaissance , First ed., Renaissance Press, 2018, pp. 586-588. http://numerical-renaissance.com/NR.pdf

Appendix B: Schematic 1.0



Appendix C: Layout 1.0



Appendix D: Bill of Materials

Part	Value	Device	DIST	DISTPN	CREATOR
A1	ANTENNA-BOARD	ANTENNA-BOARD	NA	NA	Swanson
B1	BALUN0805	BALUN0805	Digikey	732-2230-1-ND	Swanson
B2	BATTERY-SCREW-TERMINAL	BATTERY-SCREW-TERMINAL	DIGIKEY	ED10561-ND	Swanson
C1	1uF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-1UF	Digikey	587-1281-1-ND	Swanson
C2	1uF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-1UF	Digikey	587-1281-1-ND	Swanson
C3	22pF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-22PF	Digikey	490-5534-1-ND	Swanson
C4	0.75pF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-0.75PF	Digikey	490-3585-1-ND	Swanson
C5	12pF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-12PF	Digikey	490-5531-1-ND	Swanson
C6	12pF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-12PF	Digikey	490-5531-1-ND	Swanson
C7	1uF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-1UF	Digikey	587-1281-1-ND	Swanson
C8	1uF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-1UF	Digikey	587-1281-1-ND	Swanson
C9	0.1uF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-0.1UF	Digikey	445-1418-1-ND	Swanson
C10	220uF	CAPACITOR-POL_SMD-7043-D_TANTALUM-220UF	Digikey	478-8912-1-ND	Swanson
C11	220uF	CAPACITOR-POL_SMD-7043-D_TANTALUM-220UF	Digikey	478-8912-1-ND	Swanson
C12	220uF	CAPACITOR-POL_SMD-7043-D_TANTALUM-220UF	Digikey	478-8912-1-ND	Swanson
C13	220uF	CAPACITOR-POL_SMD-7043-D_TANTALUM-220UF	Digikey	478-8912-1-ND	Swanson
C14	220uF	CAPACITOR-POL_SMD-7043-D_TANTALUM-220UF	Digikey	478-8912-1-ND	Swanson
C15	1uF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-1UF	Digikey	587-1281-1-ND	Swanson
C16	1uF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-1UF	Digikey	587-1281-1-ND	Swanson
C17	0.01uF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-0.01UF	Digikey	1608-1458-1-ND	Swanson
C18	10uF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-10UF	Digikey	445-5984-1-ND	Swanson
C19	0.1uF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-0.1UF	Digikey	445-1418-1-ND	Swanson
C20	0.1uF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-0.1UF	Digikey	445-1418-1-ND	Swanson
C21	0.1uF	CAPACITOR-NP_SMD-2012-0805_CERAMIC-0.1UF	Digikey	445-1418-1-ND	Swanson
D2	green	DIODE-LED_SMD-2012-0805-GREEN	Digikey	160-1179-1-ND	Swanson
D3	360mV	DIODE-SCHOTTKY_SMD-SOD123-360MV	Digikey	CRS08QMCT-ND	Swanson
D4	360mV	DIODE-SCHOTTKY_SMD-SOD123-360MV	Digikey	CRS08QMCT-ND	Swanson
D5	360mV	DIODE-SCHOTTKY_SMD-SOD123-360MV	Digikey	CRS08QMCT-ND	Swanson
D6	360mV	DIODE-SCHOTTKY_SMD-SOD123-360MV	Digikey	CRS08QMCT-ND	Swanson
D7	green	DIODE-LED_SMD-2012-0805-GREEN	Digikey	160-1179-1-ND	Swanson
D8	BLUE	LED-BLUE	digikey	846-1207-1-ND	robntom
D9	RED	LEDRED	digikey	160-1178-1-ND	robntom
D10	ORANGE	LED-ORANGE	Digikey	754-1936-1-ND	ROBTOM
D11	YELLOW	LED-YELLOW	Digikey	754-1937-1-ND	ROBTOM
D69		LEDRED	digikey	160-1178-1-ND	robntom
J1	AVR_SPI_PRG_6-SMD	AVR_SPI_PRG_6-SMD	Digikey	952-1922-ND	Swanson
J2	HEADER-4POS-0.1IN-FEMALE	HEADER-4POS-0.1IN-FEMALE	Digikey	S7002-ND	Swanson
J3	HEADER-0.1IN-2POS-MALE	HEADER-0.1IN-2POS-MALE	Digikey	3M9447-ND	Swanson
J4		HEADER-10POSTH-254X10	Digikey	609-3250-ND	Swanson
JP2	FTDI_BASICPTH	FTDI_BASICPTH	Digikey	609-3327-ND	Swanson
MP1	MOTOR_PADS-MOLEX-0530470210	MOTOR_PADS-MOLEX-0530470210	DIGIKEY	WM1731-ND	Swanson
MP2	MOTOR_PADS-MOLEX-0530470210	MOTOR_PADS-MOLEX-0530470210	DIGIKEY	WM1731-ND	Swanson
MP3	MOTOR_PADS-MOLEX-0530470210	MOTOR_PADS-MOLEX-0530470210	DIGIKEY	WM1731-ND	Swanson
MP4	MOTOR_PADS-MOLEX-0530470210	MOTOR_PADS-MOLEX-0530470210	DIGIKEY	WM1731-ND	Swanson
NB1	NETBRIDGE	NETBRIDGE	N/A	N/A	tom+rob
Q1	NMOSFET-NMOSFET	NMOSFET-NMOSFET	Digikey	SI2302CDS-T1-E3CT-ND	rketch
Q2	NMOSFET-NMOSFET	NMOSFET-NMOSFET	Digikey	SI2302CDS-T1-E3CT-ND	rketch
Q3	NMOSFET-NMOSFET	NMOSFET-NMOSFET	Digikey	SI2302CDS-T1-E3CT-ND	rketch
Q4	NMOSFET-NMOSFET	NMOSFET-NMOSFET	Digikey	SI2302CDS-T1-E3CT-ND	rketch
Q5	NMOSFET-NMOSFET	NMOSFET-NMOSFET	Digikey	SI2302CDS-T1-E3CT-ND	rketch
Q6	NMOSFET-NMOSFET	NMOSFET-NMOSFET	Digikey	SI2302CDS-T1-E3CT-ND	rketch
Q7	NMOSFET-NMOSFET	NMOSFET-NMOSFET	Digikey	SI2302CDS-T1-E3CT-ND	rketch
Q8	NMOSFET-NMOSFET	NMOSFET-NMOSFET	Digikey	SI2302CDS-T1-E3CT-ND	rketch
R1	10K	RESISTOR_SMD-2012-0805-10K	Digikey	311-10KARCT-ND	Swanson
R2	330	RESISTOR_SMD-2012-0805-330	Digikey	311-330ARCT-ND	Swanson
R3	330	RESISTOR_SMD-2012-0805-330	Digikey	311-330ARCT-ND	Swanson
R4	10K	RESISTOR_SMD-2012-0805-10K	Digikey	311-10KARCT-ND	Swanson
R5	10K	RESISTOR_SMD-2012-0805-10K	Digikey	311-10KARCT-ND	Swanson
R6	10K	RESISTOR_SMD-2012-0805-10K	Digikey	311-10KARCT-ND	Swanson
R7	10K	RESISTOR_SMD-2012-0805-10K	Digikey	311-10KARCT-ND	Swanson
R8	64.9	RESISTOR-64.9OHM	Digikey	311-64.9CRCT-ND	ROBTOM
R9	10K	RESISTOR_SMD-2012-0805-10K	Digikey	311-10KARCT-ND	Swanson
R10	10K	RESISTOR_SMD-2012-0805-10K	Digikey	311-10KARCT-ND	Swanson
R11	10K	RESISTOR_SMD-2012-0805-10K	Digikey	311-10KARCT-ND	Swanson
R12	10K	RESISTOR_SMD-2012-0805-10K	Digikey	311-10KARCT-ND	Swanson
R13	10K	RESISTOR_SMD-2012-0805-10K	Digikey	311-10KARCT-ND	Swanson
R14	10K	RESISTOR_SMD-2012-0805-10K	Digikey	311-10KARCT-ND	Swanson
R15	69.8	RESISTOR-69.8OHM	Digikey	311-69.8CRCT-ND	ROBTOM
R16	69.8	RESISTOR-69.8OHM	Digikey	311-69.8CRCT-ND	ROBTOM
R17	80.6	RESISTOR-80.6OHM	Digikey	P80.6DACT-ND	ROBTOM
R18	80.6	RESISTOR-80.6OHM	Digikey	P80.6DACT-ND	ROBTOM
SW2	PUSHBUTTON_-4.5X4.5	PUSHBUTTON_-4.5X4.5	DIGIKEY	EG5350CT-ND	Swanson
U1	ATMEGA128RFA1	ATMEGA128RFA1	Digikey	ATMEGA128RFA1-ZU-ND	Swanson
U2	LSM9DS1TR9AXISIMU-ACCEL/GYRO/MAGI2C/SPI24LGA	LSM9DS1TR9AXISIMU-ACCEL/GYRO/MAGI2C/SPI24LGA	Digikey	497-14946-1-ND	rketch
U3	3V	TPS73633-DBVT3V	Digikey	LP3985IM5X-3.0/NOPBC	Swanson
Y1	CRYSTAL5X3	CRYSTAL5X3	Digikey	535-9122-1-ND	Swanson

Appendix E: Remote and Quadcopter Firmware Doxygen LaTeX Output

Quadcopter

3.2

Generated by Doxygen 1.8.16

Chapter 1

Quadcopter

1.1 Description

The C++ Arduino firmware for a quadcopter made as part of UCSD class CSE 176e

1.2 Hardware

Custom PCB utilizing the ATmega128RFA1.

- Quadcopter hardware by Thomas Stuart and Robert Ketchum. Documentation and EAGLE board files available here: https://github.com/rketch/quadcopter
- Remote PCB made by UCSD Professor Steven Swanson.

1.3 Libraries

All libraries available at: https://github.com/rketch/quadcopter

- Wire
  - Arduino I2C
- Arduino_LSM9DS1
  - Onboard IMU for quadcopter orientation.
- Adafruit_Simple_AHRS
  - Attitude and heading reference system for LSM
- radio
  - Radio library. Modified to include data structure sent via radio
- quad_remote
  - Custom CSE 176e library for the remote
- EEPROM
  - To save PID and trim calibration values to Non-volatile memory
- RotaryEncoder
  - To use the knob and button

1.4 Authors

Created by Thomas Stuart and Robert Ketchum, April 2019. Modified by Robert Ketchum, May 2021.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 quad_firmware/quad_firmware.ino File Reference

```
#include "radio.h"
#include <Wire.h>
#include <Adafruit_Simple_AHRS.h>
#include <Adafruit_LSM9DS1.h>
```

Macros

- #define RADIO_CH 17

  Radio Channel.
- #define M1 35

  Pin 35 is mapped to motor 1.
- #define M2 34

  Pin 34 is mapped to motor 2.
- #define M3 8

  Pin 8 is mapped to motor 3.
- #define M4 9

  Pin 9 is mapped to motor 4.
- #define FILT_RATIO 0.91

  Complementary filter ratio for a time constant of 10 Hz = 0.91.
- #define MS_TO_S 1000

  Conversion ratio.
- #define NEUTRAL_PITCH 43

  Neutral pitch gimbal position.
- #define NEUTRAL_ROLL 60

  Neutral roll gimbal position.
- #define NEUTRAL_YAW 60

  Neutral yaw gimbal position.
- #define MAX_ANGLE_FROM_NEUTRAL 10

  Maximum pitch or roll from neutral.
- #define MAX_TRIM_ANGLE 30

  Maximum pitch or roll trim angle.
- #define YAW_MAX_ANG_VEL 60

  Maximum yaw angular velocity (deg/sec)
- #define TRIM_HISTORY 3

  Number of median filtered trim values.

Functions

- Adafruit_Simple_AHRS ahrs & lsm (), &lsm.getMag(), &lsm.getGyro()
- void setupLSM ()
- void setup ()
- void loop ()
- void verifyRadioData ()
- void ComplimentaryFilter (double dt)
- void PID (double dt)
- void findGimbalOffsets ()
- void findTrimOffsets ()
- float median (float medArray[ ], int n)
- void sort (float unsortedArray[ ], int n)
- void calibrateAccelerometer ()
- void mixer ()
- void initializeMotors ()
- void disableMotors ()
- void engageMotors ()

Variables

- unsigned long time

  Current time since start.
- unsigned int last = millis()

  Last loop time since start.
- double timeDiff = 0.000000

  Implemented in complementary filter and controller.
- struct Data dataFromRemote

  Data structure received over radio.
- bool resetFlag = false

  For zeroing off orientation offset when armed.
- quad_data_t orientation

  Data structure with quadcopter orientation.
- float accHistoryArr [3][2]

  pitch, roll accelerometer reading history
- float accHistoryArrCopy [3]

  For median filter.
- float rawGyro [2]

  Raw gyro pitch, roll rate.
- float gyroAngle [2]

  Integrated gyro angles.
- float rawAcc [2]

  Raw accelerometer angles.
- float filteredAngle [2]

  Complementary filtered angle.
- float filtGyroAngle [2]

  Partially filtered gyro angle.
- float filtAccelAngle [2]

  Partially filtered accelerometer angle.
- float gyroAngleStepBack [2]

  Previous time step filtered gyroscope angle. Needed for complementary filter.

- float angleOffset [2]

  Accelerometer offset from neutral. Measured when armed.
- float trimAngle [2]

  Trimm offsets.
- float trimHistoryArr [3][4]

  Pitch, roll trim floats.
- float trimHistoryArrCopy [3]

  Trim History for median filter.
- float medOutTrimArray [4]

  Pitch, roll median filter outputs.
- float PID_output [3]

  PID output.
- float remoteSetAngle [3]

  Set angle.
- float currentAngle [3]

  Current quadcopter angle.
- float error [3]

  Error angle.
- float errorSum [3]

  Integrated error.
- float lastError [3]

  previous error (for derivative)
- float MotorValue [4]

  Implemented motor thrust variables.
- Adafruit_LSM9DS1 lsm = Adafruit_LSM9DS1()

  Create LSM9DS1 board instance.

3.1.1 Macro Definition Documentation

3.1.1.1 FILT_RATIO

```
#define FILT_RATIO 0.91
```

Complementary filter ratio for a time constant of 10 Hz = 0.91.

3.1.1.2 M1

```
#define M1 35
```

Pin 35 is mapped to motor 1.

3.1.1.3 M2

```
#define M2 34
```

Pin 34 is mapped to motor 2.

3.1.1.4 M3

```
#define M3 8
```

Pin 8 is mapped to motor 3.

3.1.1.5 M4

```
#define M4 9
```

Pin 9 is mapped to motor 4.

3.1.1.6 MAX_ANGLE_FROM_NEUTRAL

```
#define MAX_ANGLE_FROM_NEUTRAL 10
```

Maximum pitch or roll from neutral.

3.1.1.7 MAX_TRIM_ANGLE

```
#define MAX_TRIM_ANGLE 30
```

Maximum pitch or roll trim angle.

3.1.1.8 MS_TO_S

```
#define MS_TO_S 1000
```

Conversion ratio.

3.1.1.9 NEUTRAL_PITCH

```
#define NEUTRAL_PITCH 43
```

Neutral pitch gimbal position.

3.1.1.10 NEUTRAL_ROLL

```
#define NEUTRAL_ROLL 60
```

Neutral roll gimbal position.

3.1.1.11 NEUTRAL_YAW

```
#define NEUTRAL_YAW 60
```

Neutral yaw gimbal position.

3.1.1.12 RADIO_CH

```
#define RADIO_CH 17
```

Radio Channel.

3.1.1.13 TRIM_HISTORY

```
#define TRIM_HISTORY 3
```

Number of median filtered trim values.

3.1.1.14 YAW_MAX_ANG_VEL

```
#define YAW_MAX_ANG_VEL 60
```

Maximum yaw angular velocity (deg/sec)

3.1.2 Function Documentation

3.1.2.1 calibrateAccelerometer()

```
void calibrateAccelerometer ( )
```

Calculate the angle offset of the accelerometer

3.1.2.2 ComplimentaryFilter()

```
void ComplimentaryFilter (
            double dt )
```

Combine the gyroscope and accelerometer to get an accurate heading of the quadcopter

Parameters

dt	The time difference between calling the function (in ms)

3.1.2.3 disableMotors()

```
void disableMotors ( )
```

Ensure that the motors do not run when the quad is disarmed

3.1.2.4 engageMotors()

```
void engageMotors ( )
```

Write the motor commands to the respective motors

3.1.2.5 findGimbalOffsets()

```
void findGimbalOffsets ( )
```

Find the remote set angles from the gimbal positions

3.1.2.6 findTrimOffsets()

```
void findTrimOffsets ( )
```

Median the trim data and convert to a float angle.

3.1.2.7 initializeMotors()

```
void initializeMotors ( )
```

Initialize motor pins

3.1.2.8 loop()

```
void loop ( )
```

Loop forever. Receive radio data, find current quadcopter orientation, compute control system values, find current throttle corrections,write throttle values to motors if armed.

3.1.2.9 lsm()

```
Adafruit_Simple_AHRS ahrs& lsm ( ) &
```

3.1.2.10 median()

```
float median (
            float medArray[],
            int n )
```

Find the middle value of three raw inputs to filter out pertubations. Thanks geeksforgeeks

Parameters

medArray	An array containing the last n sensed values
n	The length of medArray

Returns

The median value

3.1.2.11 mixer()

```
void mixer ( )
```

Mix the user throttle value and the PID controller output

3.1.2.12 PID()

```
void PID (
            double dt )
```

Utilize PID controller to ensure stable quadcopter flight

Parameters

dt	The time difference between calling the function (in ms)

3.1.2.13 setup()

```
void setup ( )
```

Setup the quadcopter by initializing the radio, motors, IMU registers, and start clock

3.1.2.14 setupLSM()

```
void setupLSM ( )
```

set up our instance of the sensor with the wanted register values

3.1.2.15 sort()

```
void sort (
            float unsortedArray[],
            int n )
```

Sort an array into increasing numerical values. Thanks tsbrownie on youtube.

Parameters

unsortedArray	An array containing the last n sensed values
n	The length of medianFinder

3.1.2.16 verifyRadioData()

```
void verifyRadioData ( )
```

Read radio data received from the remote. If it is verified, save the data in a structure

3.1.3 Variable Documentation

3.1.3.1 accHistoryArr

```
float accHistoryArr[3][2]
```

pitch, roll accelerometer reading history

3.1.3.2 accHistoryArrCopy

```
float accHistoryArrCopy[3]
```

For median filter.

3.1.3.3 angleOffset

```
float angleOffset[2]
```

Accelerometer offset from neutral. Measured when armed.

3.1.3.4 currentAngle

```
float currentAngle[3]
```

Current quadcopter angle.

3.1.3.5 dataFromRemote

```
struct Data dataFromRemote
```

Data structure received over radio.

3.1.3.6 error

```
float error[3]
```

Error angle.

3.1.3.7 errorSum

```
float errorSum[3]
```

Integrated error.

3.1.3.8 filtAccelAngle

```
float filtAccelAngle[2]
```

Partially filtered accelerometer angle.

3.1.3.9 filteredAngle

```
float filteredAngle[2]
```

Complementary filtered angle.

3.1.3.10 filtGyroAngle

```
float filtGyroAngle[2]
```

Partially filtered gyro angle.

3.1.3.11 gyroAngle

```
float gyroAngle[2]
```

Integrated gyro angles.

3.1.3.12 gyroAngleStepBack

```
float gyroAngleStepBack[2]
```

Previous time step filtered gyroscope angle. Needed for complementary filter.

3.1.3.13 last

```
unsigned int last = millis()
```

Last loop time since start.

3.1.3.14 lastError

```
float lastError[3]
```

previous error (for derivative)

3.1.3.15 lsm

```
Adafruit_LSM9DS1 lsm = Adafruit_LSM9DS1()
```

Create LSM9DS1 board instance.

3.1.3.16 medOutTrimArray

```
float medOutTrimArray[4]
```

Pitch, roll median filter outputs.

3.1.3.17 MotorValue

```
float MotorValue[4]
```

Implemented motor thrust variables.

3.1.3.18 orientation

```
quad_data_t orientation
```

Data structure with quadcopter orientation.

3.1.3.19 PID_output

```
float PID_output[3]
```

PID output.

3.1.3.20 rawAcc

```
float rawAcc[2]
```

Raw accelerometer angles.

3.1.3.21 rawGyro

```
float rawGyro[2]
```

Raw gyro pitch, roll rate.

3.1.3.22 remoteSetAngle

```
float remoteSetAngle[3]
```

Set angle.

3.1.3.23 resetFlag

```
bool resetFlag = false
```

For zeroing off orientation offset when armed.

3.1.3.24 time

```
unsigned long time
```

Current time since start.

3.1.3.25 timeDiff

```
double timeDiff = 0.000000
```

Implemented in complementary filter and controller.

3.1.3.26 trimAngle

```
float trimAngle[2]
```

Trimm offsets.

3.1.3.27 trimHistoryArr

```
float trimHistoryArr[3][4]
```

Pitch, roll trim floats.

3.1.3.28 trimHistoryArrCopy

```
float trimHistoryArrCopy[3]
```

Trim History for median filter.

3.2 remote_firmware/remote_firmware.ino File Reference

```
#include "quad_remote.h"
#include "radio.h"
#include <EEPROM.h>
#include <RotaryEncoder.h>
```

Macros

- #define RADIO_CH 17

  Radio Channel.
- #define MAX_TRIM_ANGLE 30

  Maximum trim angle.
- #define TRIM_INCREMENT 10

  Increment by which trim is changed on remote.

Functions

- void knobPressed (bool)
- void knobsUpdate ()
- int calculateColPos (int, bool)
- void updateDisplayPIDElementChange ()
- void converToDecimalsBeforeSend ()
- float castIntPairToDecimal (int, int)
- void readPIDArrayFromEeprom (int)
- void writePIDToEeprom ()
- void writeGimbalsToEeprom ()
- void setup ()
- void loop ()
- void PIDTuningState ()
- void TrimTuningState ()
- void updateDisplayTrimElementChange ()
- void calculateTuningValues ()
- void readAndMapGimbals ()
- void isInArmState (Data &d)
- void calibrate ()
- void updateDisplayPIDNumChange ()
- void eepromRead ()
- void readTrimFromEeprom (int EepromAddress)
- void resetCoefficientValuesAndSigns ()
- void initializeButtonPins ()

Variables

- struct Data remoteToQuadData

  Structure for radio data. Struct "Data" initialized in radio.h.
- int gimbalRawValues [4]

  Values read by gimbal. Nominally 0 - 1024 but constrained by the potentiometer.
- int lowestGimbalValueReadSoFar [4] = { 0, 0, 0, 0}

  lowest value possible by analog stick gimbal
- int largestGimbalValueReadSoFar [4] = {303, 318, 320, 306}

  highest value possible by analog stick gimbal
- int saveLargestGimbalValuesSoFar [12]

  255 + 255 + 255 = 765 is the highest save value with EEPROM for gimbal
- bool knob_btn = 0

  Whether the knob button is pressed.
- bool armToken = true

  State variable for armed mode.
- bool isInTrimMode = false

  State variable for trim mode.
- int cpa [3][6]

  Coefficients Properties Array for displaying and storing PID coefficients for pitch, yaw, roll. 3 rows, 6 columns.
- int sign [9] = {1,1,1, 1,1,1, 1,1,1}

  Sign of PID tuning coefficients for storing via EEPROM.
- int PIDpos = 0

  PID value being tuned currently.
- bool changeLeft = 1

  Keeps track of whether we are tuning whole integer or decimal PID value.
- int row = 0

  LCD screen row to write to.
- int col = 0

  LCD screen column to write to.
- float trimAngle [2]

  Trim pitch and roll angles.
- String top = ""

  Initialize string to print on top row of LCD screen for PID tuning.
- int colPosStart [3]

  What column we are writing to LCD screen for PID tuning.
- bool knob_token = false

  trim or PID tuning variable
- int timesOverflowed = 0

  Keeps track of times an integer has overflowed for EEPROM writing.

3.2.1 Macro Definition Documentation

3.2.1.1 MAX_TRIM_ANGLE

```
#define MAX_TRIM_ANGLE 30
```

Maximum trim angle.

3.2.1.2 RADIO_CH

```
#define RADIO_CH 17
```

Radio Channel.

3.2.1.3 TRIM_INCREMENT

```
#define TRIM_INCREMENT 10
```

Increment by which trim is changed on remote.

3.2.2 Function Documentation

3.2.2.1 calculateColPos()

```
int calculateColPos (
            int pidPos,
            bool tuningWholeInteger )
```

Determines which cpa array column value we wish to edit

Parameters

pidPos	0->2: Pitch, Roll, Yaw
tuningWholeInteger	Whether we are tuning an integer or decimal

3.2.2.2 calculateTuningValues()

```
void calculateTuningValues ( )
```

Calculates the tuning values displayed and sent to the quad (float) from the values saved to EEPROM (int)

3.2.2.3 calibrate()

```
void calibrate ( )
```

Calibrates the gimbals and call function writeGimbalsToEeprom to save the new values. Scripted and hardcoded out of necessity.

3.2.2.4 castIntPairToDecimal()

```
float castIntPairToDecimal (
            int left_int,
            int right_int )
```

Takes the separated PID integers displayed on the LCD and combines them into one float value

Parameters

left_int	integer in the ones place on the LCD screen
right_int	integer in the decimal place on the LCD screen

Returns

castedPIDValue float value which may be used in computation

3.2.2.5 converToDecimalsBeforeSend()

```
void converToDecimalsBeforeSend ( )
```

Takes the PID user input as displayed on the LCD (as unsigned integers) and saves it in the structure "remoteTo↩ QuadData" (as floats), which will be sent over radio to the quadcopter. It also stores the sign of the PID values for EEPROM saving

3.2.2.6 eepromRead()

```
void eepromRead ( )
```

Read the values stored in EEPROM and store in a data structure

3.2.2.7 initializeButtonPins()

```
void initializeButtonPins ( )
```

Initializes all buttons on the remote

3.2.2.8 isInArmState()

```
void isInArmState (
            Data & d )
```

Arms the quadcopter if the gimbals are down and out

Parameters

&d	pointer to the the data structure containing throttle being sent via radio to the quadcopter

3.2.2.9 knobPressed()

```
void knobPressed (
            bool down )
```

Resets the current displayed PID or trim values

Parameters

down	the boolean which stores whether the knob is pressed

3.2.2.10 knobsUpdate()

```
void knobsUpdate ( )
```

Updates the stored knob value so that it agrees with the stored PID value displayed on the LCD screen

3.2.2.11 loop()

```
void loop ( )
```

Loop endlessly. Read gimbal positions, determine if quadcopter should be armed, determine which LCD tuning state the remote should be in, determine whether the user is inputing commands, and send data structure to quadcopter over radio.

3.2.2.12 PIDTuningState()

```
void PIDTuningState ( )
```

This function serves as a state machine. It is called when the remote is in the PID editing state

3.2.2.13 readAndMapGimbals()

```
void readAndMapGimbals ( )
```

Reads the analog gimbal positions and map to a value which may be sent over radio

3.2.2.14 readPIDArrayFromEeprom()

```
void readPIDArrayFromEeprom (
            int currentEEPROMAddressNumber )
```

Reads the PID values stored in EEPROM

Parameters

currentEEPROMAdressNumber	necessary to read correct data

3.2.2.15 readTrimFromEeprom()

```
void readTrimFromEeprom (
            int EepromAddress )
```

Reads the trim values stored in EEPROM

Parameters

EepromAddress	eeprom address to read correct data (should be at 43)

3.2.2.16 resetCoefficientValuesAndSigns()

```
void resetCoefficientValuesAndSigns ( )
```

Resets all PID coefficients.

3.2.2.17 setup()

```
void setup ( )
```

Setup the remote firmware by initializing radio, gimbal pins, LCD screen, knobs and buttons, and reading PID and trim values from EEPROM.

3.2.2.18 TrimTuningState()

```
void TrimTuningState ( )
```

This function serves as a state machine. It is called when the remote is in the Trim editing state

3.2.2.19 updateDisplayPIDElementChange()

```
void updateDisplayPIDElementChange ( )
```

Updates the entire LCD screen to the PID state. It should be called when an element changes on the LCD screen. For example: arming the controller or changing which pitch, roll, or yaw PID value is being tuned.

3.2.2.20 updateDisplayPIDNumChange()

```
void updateDisplayPIDNumChange ( )
```

Updates the numbers being displayed currently on the LCD screen. It should be called when a number changes.

3.2.2.21 updateDisplayTrimElementChange()

```
void updateDisplayTrimElementChange ( )
```

Updates the entire LCD screen to the tuning state. It should be called when an element changes on the LCD screen.

3.2.2.22 writeGimbalsToEeprom()

```
void writeGimbalsToEeprom ( )
```

Saves the calibrated gimbal values to the microcontroller's EEPROM

3.2.2.23 writePIDToEeprom()

```
void writePIDToEeprom ( )
```

Saves the PID and trim coefficients to the microcontroller's EEPROM

3.2.3 Variable Documentation

3.2.3.1 armToken

```
bool armToken = true
```

State variable for armed mode.

3.2.3.2 changeLeft

```
bool changeLeft = 1
```

Keeps track of whether we are tuning whole integer or decimal PID value.

3.2.3.3 col

```
int col = 0
```

LCD screen column to write to.

3.2.3.4 colPosStart

```
int colPosStart[3]
```

What column we are writing to LCD screen for PID tuning.

3.2.3.5 cpa

```
int cpa[3][6]
```

Coefficients Properties Array for displaying and storing PID coefficients for pitch, yaw, roll. 3 rows, 6 columns.

3.2.3.6 gimbalRawValues

```
int gimbalRawValues[4]
```

Values read by gimbal. Nominally 0 - 1024 but constrained by the potentiometer.

3.2.3.7 isInTrimMode

```
bool isInTrimMode = false
```

State variable for trim mode.

3.2.3.8 knob_btn

```
bool knob_btn = 0
```

Whether the knob button is pressed.

3.2.3.9 knob_token

```
bool knob_token = false
```

trim or PID tuning variable

3.2.3.10 largestGimbalValueReadSoFar

```
int largestGimbalValueReadSoFar[4] = {303, 318, 320, 306}
```

highest value possible by analog stick gimbal

3.2.3.11 lowestGimbalValueReadSoFar

```
int lowestGimbalValueReadSoFar[4] = { 0, 0, 0, 0}
```

lowest value possible by analog stick gimbal

3.2.3.12 PIDpos

```
int PIDpos = 0
```

PID value being tuned currently.

3.2.3.13 remoteToQuadData

```
struct Data remoteToQuadData
```

Structure for radio data. Struct "Data" initialized in radio.h.

3.2.3.14 row

```
int row = 0
```

LCD screen row to write to.

3.2.3.15 saveLargestGimbalValuesSoFar

```
int saveLargestGimbalValuesSoFar[12]
```

255 + 255 + 255 = 765 is the highest save value with EEPROM for gimbal

3.2.3.16 sign

```
int sign[9] = {1,1,1, 1,1,1, 1,1,1}
```

Sign of PID tuning coefficients for storing via EEPROM.

3.2.3.17 timesOverflowed

```
int timesOverflowed = 0
```

Keeps track of times an integer has overflowed for EEPROM writing.

3.2.3.18 top

```
String top = ""
```

Initialize string to print on top row of LCD screen for PID tuning.

3.2.3.19 trimAngle

```
float trimAngle[2]
```

Trim pitch and roll angles.

Index