Autonomous Quadcopter

By

Andrew Martin, Baobao Lu, Cindy Lee

Group 37

TA: Katherine O'Kane

December 9, 2015

Abstract

The goal of this project was to design an autonomous quadcopter that is capable of balancing on its own and flying from an origin to a goal using preset paths and directions. The quadcopter implemented PID control, an ultrasonic sensor to detect obstacles, and an autonomous flight algorithm. This final report includes a general introduction of the project and discusses the components and subsystems implemented in this project. It examines the design goals, the costs of the project, and the ethical and practical implications of this project.

Table of Contents

1. Introduction

1.1 Purpose

Quadcopters are a popular project for hobbyists due to their relative low complexity and cost to build. However, due to their growing popularity, there is also an increase in number of new quadcopter builders and flyers. Thus, accidents are more likely to occur due to their inexperience in handling or controlling a quadcopter during the building phase or the test-flying phase. The FAA has imposed strict laws on quadcopter flight in an effort to reduce quadcopter accidents.[1]

Our intention is to create a quadcopter that can fly independent of any user control from the ground, on a low budget. Autonomous flight of quadcopters is a difficult and expensive process, usually involving the interaction of GPS, high quality accelerometers/gyroscopes/sensors, and the control of an expensive flight controller. We believe we can create an autonomous flight system that can be preprogrammed to follow a path and carry out collision avoidance, but without using any of these expensive features. To accomplish autonomous flight, we intend to mimic the signals of a transceiver/receiver system by using a simple ARM microcontroller. By preprograming a set of flight signals onto the ARM microcontroller and passing them to a cheap flight controller, we can preset a simple flight routine. To accomplish collision avoidance, we intend to integrate onto the ARM microcontroller a simple ultrasonic sensor. If the ultrasonic sensor detects an object close to the quadcopter, the ARM microcontroller modifies the flight path to keep the quadcopter safe from collision. This report will provide a general outline of the system and will discuss in detail the results of our team in attempting to meet these goals.

1.1.1 Defining 'Autonomous'

The question of whether the aforementioned design constitutes an 'autonomous' quadcopter has been discussed and debated throughout the course of this project. Merriam-Webster defines autonomous as "existing or acting separately from other things or people: having the power or right to govern itself" [http://www.merriam-webster.com/dictionary/autonomous]. We feel that this definition describes our quadcopter build. Autonomous systems "exist and act", to the extent of their ability to be aware of their surroundings. We have pushed the boundaries of how the quadcopter exists and acts with the limited usable information we could acquire from the ultrasonic sensor.

Criticisms that the quadcopter does not follow a spatial path exactly as described in the code fall outside the definition of autonomy as it pertains to this project. The quadcopter has the power to act on a set of instructions, and executes them to the best of its ability. Further, we believe the question of precision in flight is ambiguous, and since it was directly addressed in our requirements and verification, we feel it does not invalidate our claim that the quadcopter is autonomous. Criticisms that the quadcopter "flies blind" are reasonable but simplistic, as the quadcopter has the capability to react to its surroundings (decrease throttle and disarm), even if the capability is extremely limited in the given setup. These points will be addressed further in the ethical considerations section (5.3) of the report.

1.2 Block Diagram

Figure 1 outlines a top level design of the quadcopter.

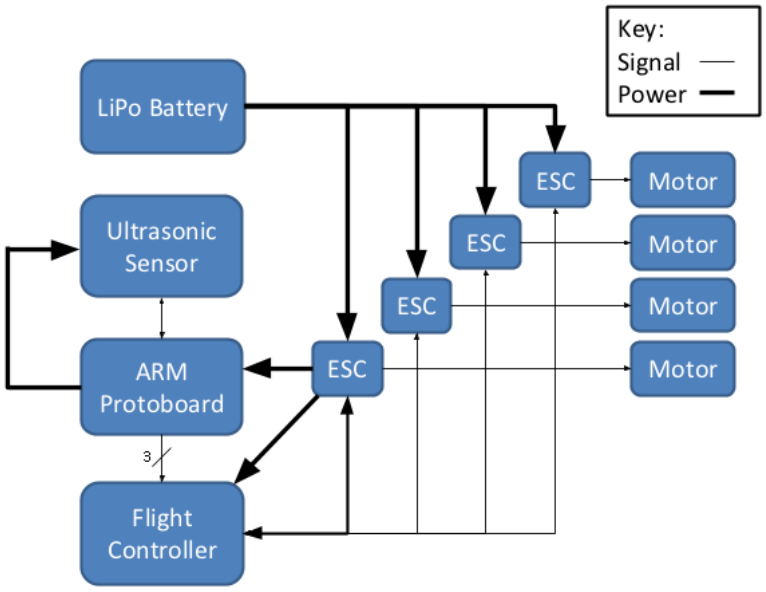

Figure 1: Block Diagram of quadcopter

2 Design

2.1 Design Procedure – Block Diagram Components

2.1.1 LiPo Battery [Turnigy 3 cell 3300mAh]

This is the source of power for entire quadcopter. The 11.1V battery routes power directly to the ESCs. The ESCs divert power to the motors, flight controller, and ARM microcontroller.

2.1.2 Electronic Speed Controllers (ESC) [F-20A Fire Red Series Simonk]

These controllers supply power to the motors by translating signals from the flight controller to electrical energy. Inputs are control signals from the flight controller and 11.1-12.4V from the LiPo Battery. Outputs include a 5V battery eliminator circuit (BEC) power supply that can power a flight controller or ARM microcontroller, and up to 220W of power for the motors. They are rated to pull up to 20A of power from the LiPo Battery. The amount of electrical energy supplied to the motors is based on the input signals from the flight controller.

2.1.3 Motors [HT-450 Brushless Multicore]

The motors receive 220W from electronic speed controllers (ESCs) and are used to drive the propellers. Speed tuning of the motors is achieved through PID tuning of the ESCs via CleanFlight software.

2.1.4 Flight controller [Copter Controller 3D]

The CC3D flight controller enables aircraft stabilization and flight. The CC3D board is a quadcopter flight controller that normally runs the OpenPilot firmware. This firmware takes inputs from the receiver and the onboard gyroscope and calculates the optimal motor speeds to produce the desired input from the quadcopter operator. It is powered by the 5V supply source from one of the ESCs and controls the ESC's transmission of power to the motors. It can be integrated with any airframe and is configured and monitored using a variant of OpenPilot software called CleanFlight. The CC3D also has an integrated PID control system that can be tuned in the CleanFlight software.

2.1.4.1 PID Control

We chose PID control because it is relatively simpler in complexity than other types of control systems. Additionally, CleanFlight integrates PID tuning, which makes PID tuning easy to control and directly implement.

The basics of PID control can first be shown in the following diagram by Oscar Liang.[2]

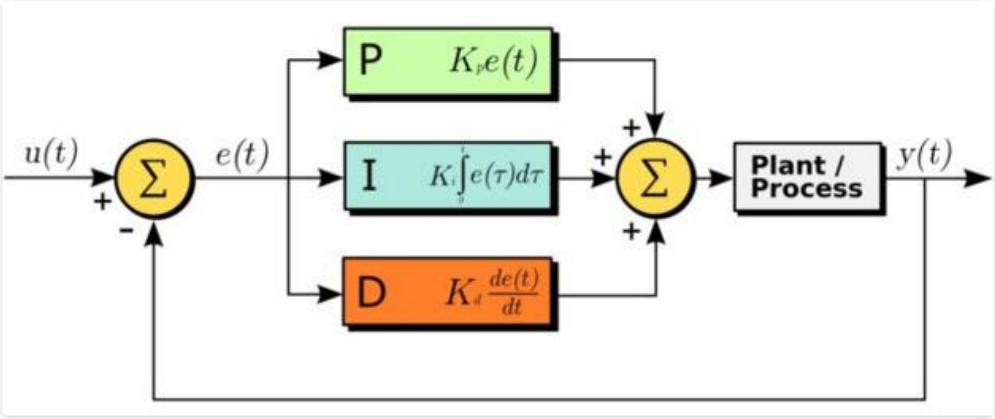

Figure 2: PID control outline for quadcopter[2]

When the feedback control signal is linearly proportional to the system error, it is called the proportional feedback, which is the P term in the diagram. The I term is the proportional plus integral term to get the automatic reset result. The last term D is derivative control, which has an important effect of giving a sharp response to suddenly changing signals.

PID, overall called the proportional integral derivative is a closed loop control system that is useful in getting the actual result of quadcopter control closer to the desired result by adjusting the input the quadcopter system. Many quadcopters use PID controller to achieve stability.

2.1.5 ARM Microcontroller [LPC1114, 32-bit ARM Cortex]

This is a replacement for the traditional receiver, which is normally used in conjunction with a transmitter to supply throttle and direction controls to the quadcopter flight controller. In a traditional setup, the receiver produces 3.3V PWM pulses that reflect the inputs of the transmitter it is paired with. Our ARM microcontroller replicates these 3.3 PWM pulses and, like the receiver, sends them to the flight controller. The ARM microcontroller is a development platform that takes input from ultrasonic sensor and charts an autonomous flight path by replicating receiver signals. It alters course by modifying the PWM pulses it sends if input from sensors indicates a nearby object in the path of the quadcopter.

2.1.6 Ultrasonic Sensor [HC-SR04]

The HC-SR04 sensor is in charge of detecting obstacles and outputting distance data to the ARM microcontroller. It is triggered by a pulse input from the ARM microcontroller. It has ranging accuracy of up to 3mm. The sensor has wire connections that consist of 5V supply, ground, trigger pulse input and echo pulse output. The working current is 15mA, and the working frequency is 40Hz. Figure 3 outlines the expected TTL operation of the trigger and echo pins of the Ultrasonic sensor.

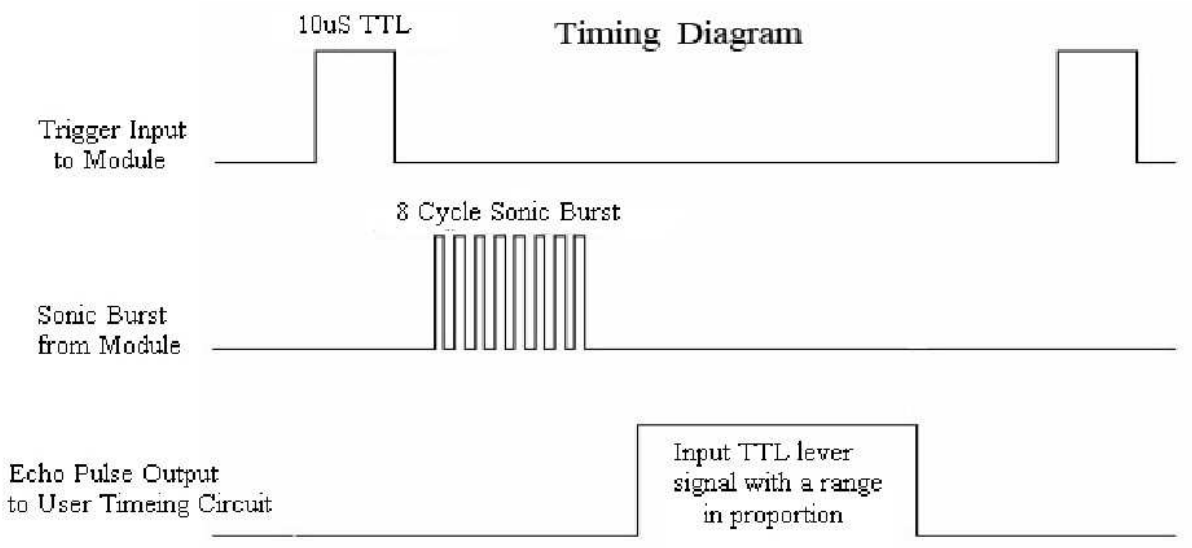

Figure 3: TTL Timing Diagram for HC-SR04 Ultrasonic Sensor[3]

2.1.6.1 Ultrasonic Sensor Power Supply

Because the echo output of the ultrasonic sensor is sensitive to the quality of the input 5v power supply it receives, a dedicated power regulator was designed to ensure quality output from the sensor. This hardware component uses a simple LM7805 power regulator to regulate an input 11.1 volt source from the LiPo battery. It features a female pin header used to connect both the outside voltage source input and the ultrasonic sensor output. It also features pads for connecting the trigger and echo pins to the ARM microcontroller circuitry.

2.2 Design Details

2.2.1 Power supply chain (LiPo Battery, ESCs, Motors)

In order to ensure the battery was powerful enough to fly the quadcopter, gravitational physics calculations were made to determine the power required to make the quadcopter hover:

$$F = M * a; \tag{1}$$

$$m = 43*4+11+5.7+278+40+200 = 0.706\,kg; \tag{2}$$

$$Fgravity = m * g = 0.7067 * 9.8 \approx 6.9\,N \tag{3}$$

$$F > Fg \approx 7\,N$$

$$W = F * d = 7 * 1\,m \tag{4}$$

$$P = W \div t = 7/1\,s = 7\,Joules/second \tag{5}$$

It can be shown that the LiPo battery, which is intended to operate from 11.1-12.4 volts, can support a draw of at least 80A to power all four motors. Thus, the power chain can produce 11.1V * 80A = 888 J/sec, which is more than sufficient to make the quadcopter hover and fly. This value is corroborated by the motors, which are each rated for 220W of power, which when

combined suggests that the quadcopter is rated to use 880 W.

If it is assumed that each motor draws 20A and that the additional components on the quadcopter (ARM microprocessor, CC3D, ultrasonic sensor) require one Amp to power overall, we can estimate a safe maximum flight time for the quadcopter. It can be shown that for a 3300mAh battery:

$$Flight\ Time\ (\text{min}) = 80\%\ rule * \frac{mAh}{1000} * \frac{1}{current\ draw * flying\ load} * \frac{60\ minutes}{1\ hour} \quad (6)$$

$$= 80\% * \frac{3300}{1000} * \frac{1}{(4*20+1)*30\%} * \frac{60\ minutes}{1\ hour} = 8.1\ minutes \quad (7)$$

Figure 5: Estimation of total flight time of quadcopter

Here the 80% rule is a safeguard to prevent overtaxing the battery, and the flying load is the estimated percent of maximum capacity carried by the quadcopter during flight.[4] Thus the quadcopter can be safely run for up to 8 minutes before the battery needs to be recharged.

2.2.2 Flight controller

The CC3D is a hardware flight controller. The board runs on OpenPilot firmware, and it is configured to run CleanFlight software. The CC3D is connected to the computer using USB port and CleanFlight is run on the computer in order to monitor the aircraft. The software is used for PID tuning, calibration, and for acquiring controller values. Signals from the CC3D flight controller are then sent and translated by the ESC and are used to power the motors. Since the ARM microcontroller acts as a receiver, the firmware in the CC3D takes inputs from the ARM microcontroller and the onboard gyroscope and calculates the optimal motor speeds to produce the desired input from the quadcopter operator. The ARM microcontroller replicates the 3.3 PWM pulses and, like the receiver, sends them to the flight controller. The ARM is connected to the receiver ports of the CC3D flight controller. The length of the PWM pulse specifies the output and throttle positions, which are interpreted by the CC3D as transceiver/receiver controls. CleanFlight can take controller values from 1000 to 2000, but the user is able to set different limits.

2.2.2.1 PID Control

The effects of PID tuning in CleanFlight can be visualized when the propellers are installed on the quadcopter with the motors rotating. Since the position and the resistivity of change of direction could be detected when we change the values of the PID terms. From further research, we found out that increasing the value of P would stabilize the quadcopter, and therefore creating a strong resistive force when someone attempts to move the motor. Decreasing the value of P on the other hand, will give the quadcopter less resistive force when one attempts to stop or move the motor. Increasing the value of I, could increase the ability of the quadcopter to hold overall position, and thereby reducing drift due to unbalanced forces. Decreasing I, could improve reaction to changes, but also make the quadcopter prone to drifting in random directions. Increasing the value of D, will add damping on the quadcopter, and make it react slower to fast changes.

Decreasing the value of D, provides less damping to the quadcopter, and thereby results in the quadcopter reacting faster to changes.

The default PID values are given in Figure 4 below:

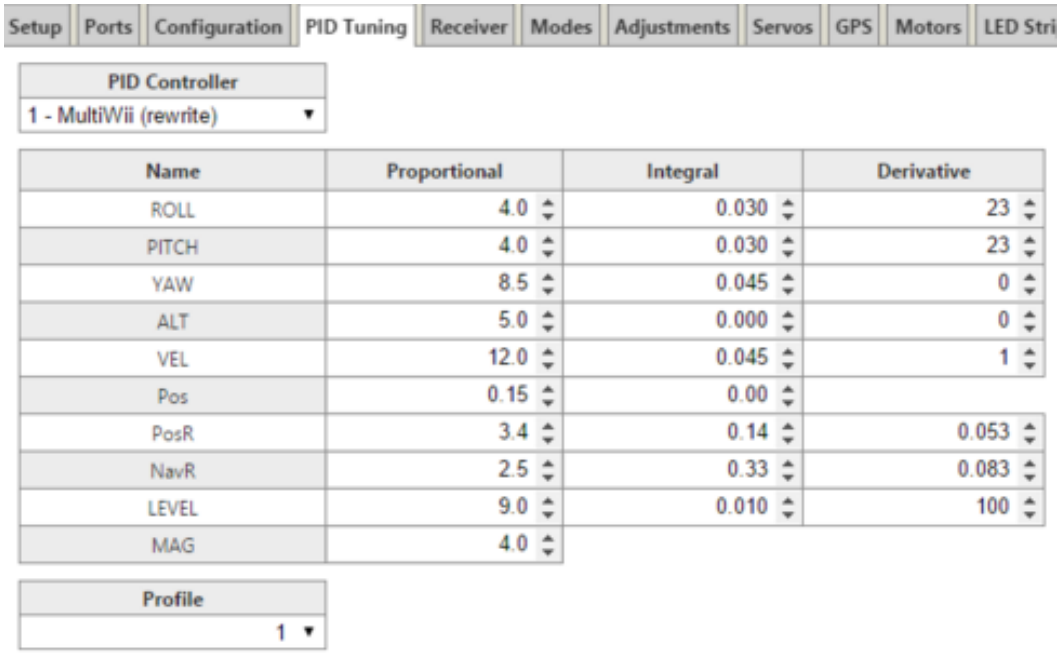

Figure 4: PID values for typical quadcopter setup[5]

In order to detect noticeable behavior changes in the motor, the values needed to be changed drastically from the default values. For example, when changing P, if you start from the given value, which is 4, the motor oscillations are not visible at all, and if you change it to 8, the effect is still barely noticeable, if you keep increasing the value to 12, oscillations begin to become audible in the motors. If P is increased to 17, oscillations are visible. So in order to detect visible changes in the motor due to PID parameters, one must greatly increase the values.

The table below is obtained after numerous tests in order to get the most stable flight. We first started with the default values and through trial and error incremented or decremented the constants in order to find the most optimal values.

Table 1: Final PID Values for Quadcopter

Name	Proportional	Integral	Derivative
Roll	4.0	0.06	23
Pitch	4.0	0.06	23
Yaw	8.5	0.045	0

2.2.3 ARM Microcontroller

Figure 5 below demonstrates the train of pulses and individual lengths of pulses generated by the Transceiver/Receiver, which the ARM attempts to emulate. Because the Transceiver/Receiver system can only produce pulses with widths between 1.12 ms and 1.92 ms, The ARM microprocessor is only required to generate PWM pulses between these two widths. The 20ms split between pulses remains constant regardless of the width of the PWM pulse.

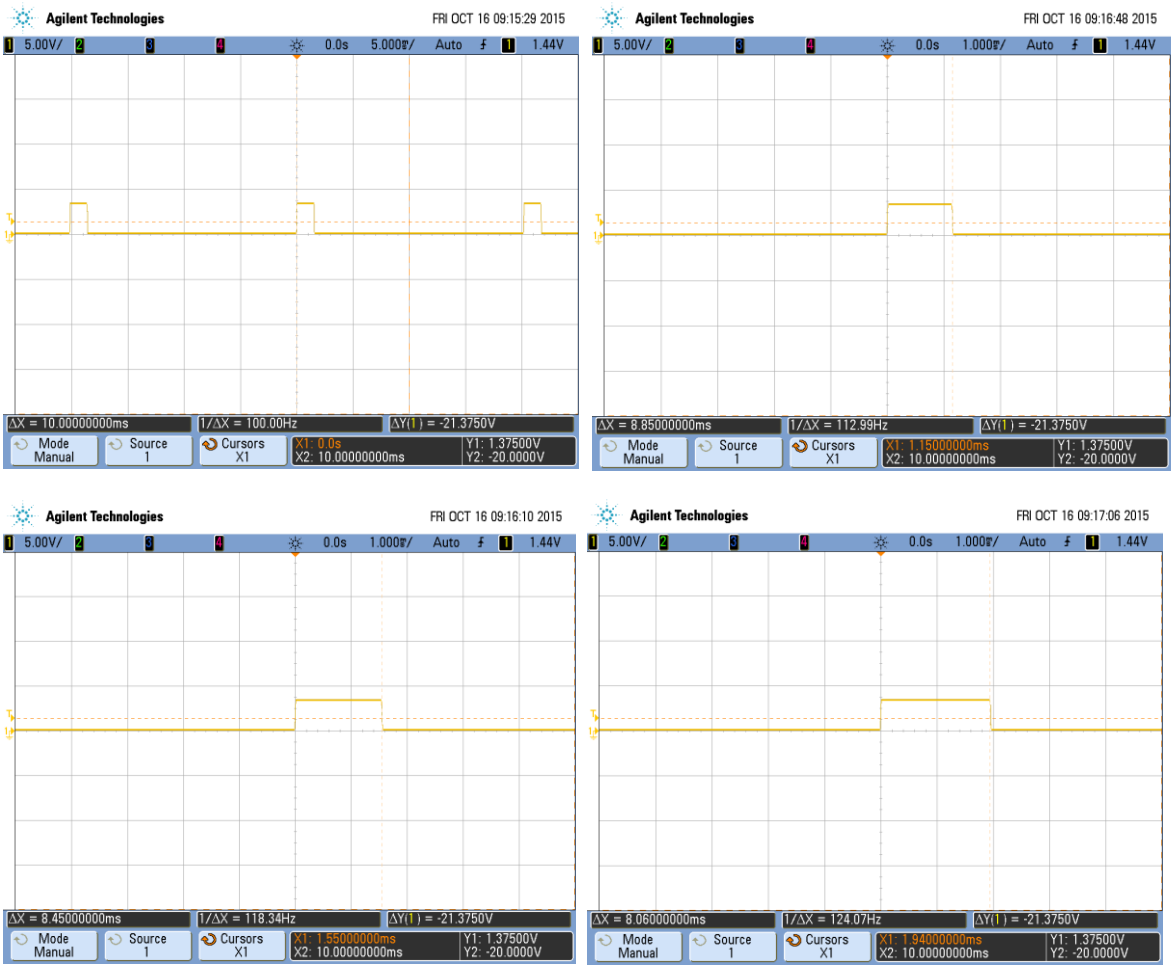

Figure 5: Clockwise from top left: 1.5ms pulse with 20ms split; Minimum length pulse, 1.12 ms; Maximum length pulse, 1.92ms; Medium length pulse, 1.5ms

To generate PWM signals at selected GPIOn pins[6], the frequency and the pulse width of a PWM signal is stored as data. Both are stored in increments of 10 microseconds for simplicity. The timer is also set to increment at each 10 microseconds. Then at time, t = 0, the signal is set to HIGH (or 3.3V) by ORing the GPIOn pin with a one. The signal stays HIGH until t equals the pulse width where the signal is masked by a zero bit, resulting in a LOW (or 0 V). In our case, the frequency is 50 Hz (equivalently stored as 2000*10 microseconds) and the pulse width could go between 1.0 ms to

2.0 ms (equivalently stored as 100*10 microseconds and 200*10 microseconds). The value of time t is also bounded by the frequency so it has values from 0 to 2000 so t resets to 0 when t reaches 2000. For example, we have pulse width of 1.55 ms (156*10 microseconds) at GPIO5 pin. Then the PWM signal would be processed as follows:

t = 0, GPIO |= (1 << 5), (<< 5, stands for shifting the bit by 5)

t < 155, no change

t = 155, GPIO &= ~(1 << 5); (~ stands for inverting the bits)

t > 155, no change

2.2.4 Ultrasonic Sensor

The ultrasonic sensor's trigger pin should output an ultrasonic wave that travels at a speed of sound.[3] Hence; the distance should be related to the wave's duration of travel as follows:

$$distance = \frac{duration \times speed\ of\ sound}{2} \tag{8}$$

To verify if the ultrasonic sensor Echo pin outputs a high at the right duration after the Trigger Pin sends a wave, the ultrasonic sensor was connected to a RedBoard programmed with Arduino as shown in Figure 6. A solid object was placed in front of the ultrasonic sensor at certain distances and was detected by the ultrasonic pulse. Code was put into the RedBoard to obtain the duration between trigger and echo that correspond to certain distances. Results of these tests are shown in Appendix C.

From the results obtained it can be seen that the average measured duration has a small percentage error that gets no larger than 4% when the distance is 7 cm and above. It has a high percentage error when tested using 2cm, 4cm and 6cm and for values past 75cm.

In addition to the values between 2cm-75cm tested in Appendix C, the ultrasonic sensor was pointed at a wall more than 75cm away. The ultrasonic sensor failed to detect the distance accurately; the data it outputted suggested the wall was at a distance around 50cm. Thus in order to ensure the sensor produced reasonable values, code for the quadcopter only implemented detection of objects at distances less than 50cm from the sensor.

During testing, it was noted that the ultrasonic sensor requires a steady 5V at the VCC input to produce reliable data. Because the RedBoard was only connected to the computer through USB, the RedBoard put out slightly less than 5V to power the sensor, which caused irregularities in testing. While we were able to use a benchtop power supply to hold the ultrasonic sensor voltage stable to obtain reliable data, it was decided that a standalone power supply would be necessary on the quadcopter. To that end, an ultrasonic power supply was fabricated to provide the 5V needed by the sensor. Ultimately, though, power from the battery eliminator circuit (BEC) of one of the four ESCs was used instead. Lastly, the code for operating the sensor on the RedBoard was ported to the ARM microcontroller and integrated into the flight control code.

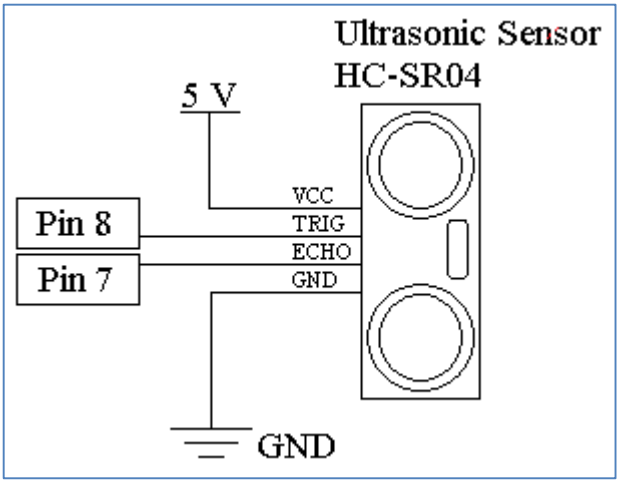

Figure 6: Circuit connecting Ultrasonic Sensor to RedBoard

2.2.4.1 Ultrasonic Sensor Power Supply

Although fully realized, the power supply of the ultrasonic sensor was not integrated into the final quadcopter build. An LM7805 was used as the primary component for the power supply. [7] The voltage regulator is rated to operate between 7-25V, so the introduction of an 11.1V power supply should have fallen within the bounds of a reasonable power source. However, the datasheet for the regulator indicates that for the given voltage and current schema, output voltage could range from 4.8V to 5.2V, which was observed in our experimental setup. Thus, the voltage regulator's variability in output made it insufficient for use in this application. Further diagrams of the board schematics for this part can be found in Appendix B.

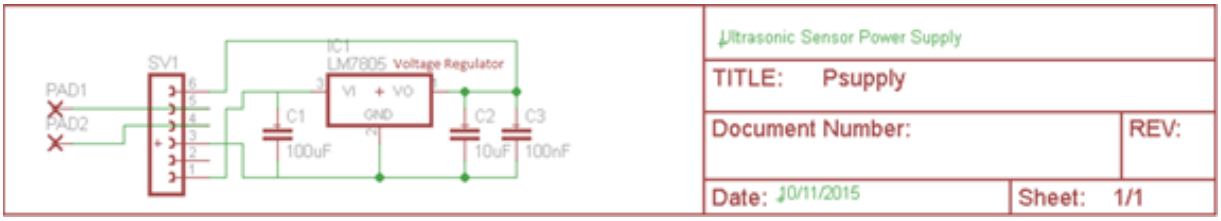

Figure 7: Hardware design schematic.

3. Verification

3.1 Power Chain Verification

In order to verify the correct power outputs of the LiPo Battery and ESCs, A multimeter was used to check voltages against those specified in the requirements and verification table (Appendix A). Although it was possible to verify the synchronous startup of motors after flight controller tuning, finding the maximum speed of each of the motors was not accomplished. No reliable method of measuring motor spin speeds was conceived or executed. Ultimately, all voltage and current related requirements were verified on the quadcopter through analysis with a multimeter.

3.2 Flight Controller Verification

In order to verify the function of the flight controller, the motors and accelerometer were calibrated on the flight controller. To calibrate the motors, the CC3D must be connected to the computer with CleanFlight running. Then under the Motor section of CleanFlight, the tab must be set to a maximum value for motors. Then the battery must be powered on. Afterwards, the motor tab needs to be put down to the minimum value. Then the battery and flight controller can be disconnected. The ESCs use the maximum and minimum values from CleanFlight to determine the proper input values for which the motors should be turned on.

To calibrate the accelerometer, the CC3D is simply set on a flat surface while connected to CleanFlight and the button to calibrate the CC3D is pressed. The CC3D consistently calibrated to within ±0.5° in all accelerometer directions on a perfectly flat surface.

3.3 ARM Microcontroller Verification

In order to ensure we could first program with the ARM microcontroller, it was necessary to fabricate a simple circuit utilizing the ARM microcontroller on a breadboard and attempt to load simple programs on it. In order to confirm that the ARM microcontroller worked, the experimental setup depicted in Figure X was constructed. This experimental setup schematic is documented in Appendix C. The first ARM microcontroller tested was shown to output about 1.1 volts on a line that should have output 3.3 volts. Further, the µVision software designed for programming the ARM microcontroller could not successfully program the ARM microcontroller. After several days of study, it was determined the microprocessor being used was broken. The ARM microcontroller was switched out and was found to program correctly, and output the correct 3.3 volts on the appropriate pin. The program was tested again after the ARM microcontroller was transferred to the completed PCB and was found to perform the same as on the breadboard.

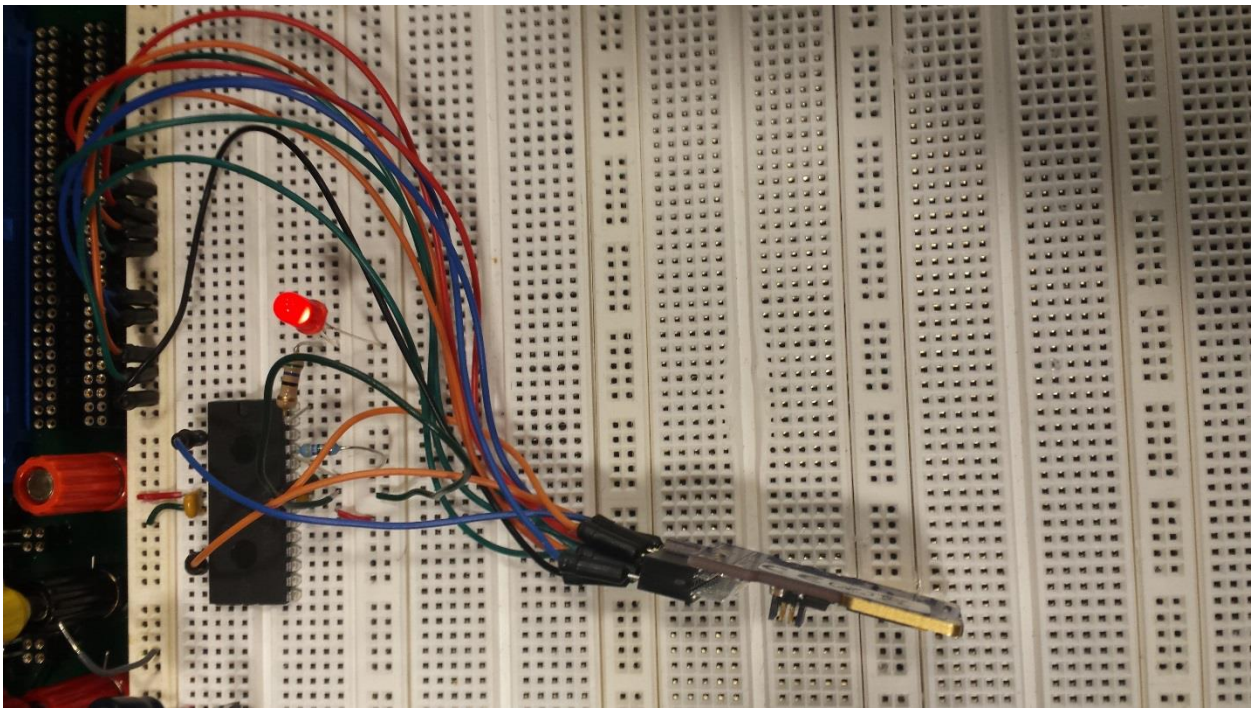

Figure 8: Image of the quadcopter processor (lower left) powering an LED. The JTAG programmer (lower right) is used to translate and send code from the µVision software to the ARM microcontroller.

3.4 Ultrasonic Sensor Verification

As part of testing, it was important to determine whether it was possible to receive correct output from the HC-SR04 sensor given ideal input and power conditions from a benchtop setting. These considerations included powering the sensor at Vcc of 5 volts and sending a 10uS TTL pulse to the trig pin. The following settings were used to produce a 10uS TTL pulse from the signal generator:

- Freq: 20kHz
- Amplitude: 1.25 Vpp
- Waveform: Square
- Offset: 1.25 Volts
- Duty Cycle: 25%
- Burst: On
- Trigger: Single

By implementing these settings, a 10uS TTL pulse was produced that can be triggered by pressing the trigger button on the signal generator.

The resultant output for two scenarios can be seen in Appendix C. The green line in both cases indicates the length of the trigger input, while the yellow line indicates the output from the echo signal. In the first graph, the object being detected is placed close to the sensor, about 1 ft. The second graph shows the results of putting an object roughly 6 inches further away. The first graph shows an echo result, which is shorter in length than the second signal. This correctly indicates that the echo takes less time to travel in the first case versus the second case, because the object is closer in the first case. Further tolerance analysis conducted to test the limits of the sensor can be found in Appendix C.

4. Costs

4.1 Labor

Table 2: Labor

Name	Hourly Rate	Total Hours Invested	Total Labor = Hourly Rate x 2.5 x Total Hours Invested
Andrew Martin	40.00	100	10,000.00
Baobao Lu	40.00	100	10,000.00
Cindy Xin Ting Lee	40.00	100	10,000.00
		Total	$30,000.00

4.2 Parts

Table 3: Parts

Parts	Unit Cost (USD)	Quantity	Total
Flip Sport Quadcopter Frame	65.00	1	65.00
HT-450 Motor - House	15.00	4	60.00
F-20A Fire Red Series Simonk-(RapidEsc)	8.00	8	64.00
HQ 9X4.5 MM Fiberglass Composite Propeller	3.98	4	15.92
Open Pilot CC3D	15.53	1	15.53
Turnigy 3300mAh 3S 30C Lipo Pack	25.80	1	25.80
IMax B6 Digital RC Lipo NiMh Battery Charger	22.72	1	22.72
Turnigy TGY-i6 Transmitter	49.00	1	49.00
Turnigy TGY-i6 6CH Receiver	Bundled with Transmitter	1	0.00
Breadboard PCB	8.50	1	8.50
ARM LPC1114FN28	5.28	1	5.28
USB MINIJTAG Debugger/Emulator	13.00	1	13.00
UA78MC-UIC 3.3V Voltage Regulator	1.00	1	1.00
R4.7/25 47uF Capacitor	0.15	1	0.15

COM-08375 0.1uF Capacitor	0.25	2	0.50
COM-00533 3mm Red LED	0.35	2	0.70`
CF1/2W472JRC 4.7k resistor	0.09	2	0.18
HC-SR04 Ranging Distance Sensor	10.00	1	10.00
		Total	$358.28

4.3 Grand Total

Table 4: Grand Total

Labor	$30,000.00
Parts	$358.28
Total	$30,358.28

5. Conclusion

5.1 Successes and Uncertainties

The quadcopter was able to follow autonomous paths as promised. It was demonstrated that it was possible to acquire PWM data normally sent by a transceiver and receiver. It has been shown that it was possible to duplicate this PWM data on an ARM microcontroller and deliver it to the CC3D. It was shown that the CC3D could receive control data for thrust, pitch, roll, and yaw and use the data appropriately to navigate.

The ultrasonic sensor was also successfully utilized in the design. It was shown that the quadcopter ESCs could provide stable 5 volts and that the ARM microcontroller could trigger the operation of the sensor. The ARM microcontroller successfully measured inputs from the ultrasonic sensor and could determine if objects were in the path of the sensor. The ARM microcontroller was shown to lower the throttle and ultimately shown to disarm the quadcopter if an object was placed in the path of the sensor.

However, some uncertainties existed in the final design. Despite the fact that the ARM microcontroller could pilot the quadcopter autonomously, it was not possible to assess the location of the quadcopter in its flight path accurately. Drift was a significant contribution to deviations in the planned path, and made it difficult to determine the final position of the quadcopter.

Uncertainties also existed in the ultrasonic sensor. The sensor frequently was shown to produce false positives of object detection when put in operation. To counteract these false positives, a program was implemented to take many samples and execute a disarm only if a certain threshold of positives were met. However, the ultrasonic sensor still failed to detect objects correctly in these cases, and was considered unreliable. Additionally, it was not shown that the ultrasonic sensor could be used in an in-flight autonomous situation. It remains to be seen as to whether autonomous flight and ultrasonic object detection can be integrated together into a final product.s

5.2 Ethical Considerations

IEEE Code of Ethics[8]	Relevance in Design
"1. to accept responsibility in making decisions consistent with the safety, health, and welfare of the public, and to disclose promptly factors that might endanger the public or the environment"	The purpose of this project is to produce a design that will increase the safety of quadcopters and decrease the risk of those flying them.
"2. to avoid real or perceived conflicts of interest whenever possible, and to disclose them to affected parties when they do exist"	The university has strict rules about the operation of quadcopters on university property. All flight-testing was done outside of university property in accordance with these rules.

"3. to be honest and realistic in stating claims or estimates based on available data"	Our claim that the quadcopter operates in an autonomous fashion has been disputed by outside sources. The claim must be re-assessed and arguments against it must be carefully considered to see if the claim is valid. An argument in support of the claim is presented in section 1.1.1 of this paper.
"4. to reject bribery in all its forms"	Bribery to change the quadcopter design would distance the resulting product from the goal.
"5. to improve the understanding of technology; its appropriate application, and potential consequences"	Full documentation of the design of the quadcopter will ensure that others that attempt the build will avoid the mistakes made during the design process.
"6. to maintain and improve our technical competence and to undertake technological tasks for others only if qualified by training or experience, or after full disclosure of pertinent limitations"	The design of this project was chosen to challenge but not exceed the abilities of those working on it, to ensure the result was achieved in the safest way possible. To this end, sacrifices to complexity may have been taken in order to allow a reasonable, safe pace of development throughout the course of the project.
"7. to seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others"	The group and the persons in it did not fail to acknowledge the help provided by peers and staff.
"8. to treat fairly all persons and to not engage in acts of discrimination based on race, religion, gender, disability, age, national origin, sexual orientation, gender identity, or gender expression"	The group and the persons in it did not discriminate in the process of seeking help and support in the development process.
"9. to avoid injuring others, their property, reputation, or employment by false or malicious action"	The quadcopter was specifically designed to avoid injuring others and their property. In cases where the components of the quadcopter could not perfectly comply with these standards, the design was changed to meet these standards.
"10. to assist colleagues and co-workers in their professional development and to support them in following this code of ethics"	Colleagues will be supported in their professional endeavors, and the code of ethics will always be integrated into that support.

5.3 Future Work

The immediate future goal of this project would be to test and integrate the autonomous flight of the quadcopter with the ultrasonic sensor capabilities of the quadcopter. In order to improve the object detection capabilities of the quadcopter, it would be helpful to implement more ultrasonic sensors, and to implement higher quality ultrasonic sensors in the design. The current sensor in use is not reliable enough to be used in future versions of this project. Additionally, with only one sensor, objects in only one direction can be detected. With multiple sensors, objects can be detected in multiple directions, ensuring complete protection of the quadcopter during flight.

Additionally, it would be useful to implement further refinements to the flight paths of the quadcopter. In order to improve the flight path quality, additional PID tuning should be carried out to reduce drift in the quadcopter. Further PID tuning will make it possible to keep better track of quadcopter position and orientation, which will make navigation around obstacles (and not simply deactivation upon detection of obstacles) a possibility.

Lastly, if the autonomous setup will be used extensively, it would be helpful to implement a GUI to allow for users to easily design and implement flight paths. As of now the only way to develop a flight path is to hardcode it into the ARM microcontroller, which is impractical for the target product group (casual flyers).

References

[1] Federal Aviation Administration. *H.R.658 - FAA Modernization And Reform Act Of 2012*. 2011. Print.

[2] O. Liang. “Quadcopter PID Explained and Tuning.” Available at: http://blog.oscarliang.net/quadcopter-pid-explained-tuning/, Oct 13, 2013.

[3] Micropik. “Ultrasonic Ranging Module HC-SR04.” HC-SR04 datasheet.

[4] multicopter.forestblue.nl. “lipo battery calculator.” Available at: http://multicopter.forestblue.nl/lipo_need_calculator.html.

[5] J. Case. “CleanFlight PID Tuning.” Available at: http://open-txu.org/home/special-interests/multirotor/cleanflight-pid-tuning/.

[6] NXP Semiconductors N.V.. “LPC1110/11/12/13/14/15 32-bit ARM Cortex-M0 microcontroller; up to 64 kB flash and 8 kB SRAM.” LPC1110/11/12/13/14/15 datasheet., 16 Apr. 2010 [Revised Mar. 2014]

[7] Fairchild. “LM78XX/ LM78XXA 3-Terminal 1 A Positive Voltage Regulator” LM78XX/ LM78XXA datasheet.

[8] Institute of Electrical and Electronics Engineers, Inc.. “Code of Ethics IEEE.” Available at: http://www.ieee.org/about/corporate/governance/p7-8.html. 2006.

Appendix

Appendix A: Requirement and Verification Table

Requirements	Verification	Verified?
Power: 1. LiPo must charge to and provide 11.1V±1.5V and 12A continuous, 16A burst to ESCs 2. ARM Microcontroller, Flight Controller, and Ultrasonic sensor must receive steady 5V ±0.5V	Power: 1. Monitor initial charge with IMax B6 Digital Battery charger 2. When plugged in to ESCs, monitor voltage with multimeter by measuring from LiPo-ESC connector in parallel, measure amperage by connecting multimeter to same connector, but in series	Y Y
ARM Microcontroller: 1. Must duplicate PWM receiver signals normally sent to flight controller and send to flight controller (Duty cycle accurate within ±5% of desired receiver signals, Period accurate within ±1% of receiver signals). 2. Must receive data from ultrasonic sensors by supplying a pulse and modify flight path if proximity conditions are met	ARM Microcontroller: 1. If receiver signals are correctly duplicated, motors will turn in the same way as if they had been controlled directly by the receiver. 2. If ultrasonic sensor data is correctly received and processed the flight path will change and an LED indicator will light up on the protoboard. Sending of a pulse can be verified by hooking the appropriate output pin to an oscilloscope.	Y Y
Flight Controller: 1. Must calibrate accelerometers so that tilt on all axes is 0° ±1° when placed on a flat surface. 2. Receive 5V, process receiver or ARM microcontroller signals correctly and send appropriate signals to ESCs.	Flight Controller: Accelerometers will output to cleanflight the tilt angles of the flight controller. Motors will turn when signals are sent to the flight controller if functional.	Y
Sensors: Must send ultrasonic distance data to ARM Microcontroller once triggered via a TTL pulse. TTL pulse must be 10µS ± 2µS. In order to prevent trigger signal conflicting with echo signal, measurement cycle must have a delay between pulses of at least 60 ms.	Sensors: Externally powering the sensor via power supply and looking at the echo output on the sensor via oscilloscope once a pulse trigger is supplied should show the signal generated by the return of the ultrasonic pulse to the sensor. The ARM microcontroller will supply this pulse once attached to the sensor.	Y

ESC: 1. Must provide up to 5V ±0.5 V to flight controller and ultrasonic sensor and 10-12 A to motors 2. Must output the correct power (0-220 Watts) to correspond to the input signals.	ESC: Voltage and current outputs can be tested by hooking up a multimeter in parallel and series respectively. Motors will turn if system is fully assembled and controlled via ARM microcontroller or receiver.	Y
Motors: Must turn when powered from ESC and run at comparable speeds after PID tuning. When motors all run at max speed, difference between motor speeds cannot exceed ± 500 Srpm. Tolerance for motor speed differences should scale with speed (i.e. at half of max speed, motors can have at max a ± 250 Srpm difference in speeds)	Motors: 1. Plug in the transceiver/receiver and pushing output throttle to the maximum 2. Measure the output speed in Hz. In order to meet requirements, Hz must not vary between the four motors by more than 600* ±500/24000 = ±12.5 Hz.	Y N

Appendix B: Large Schematics

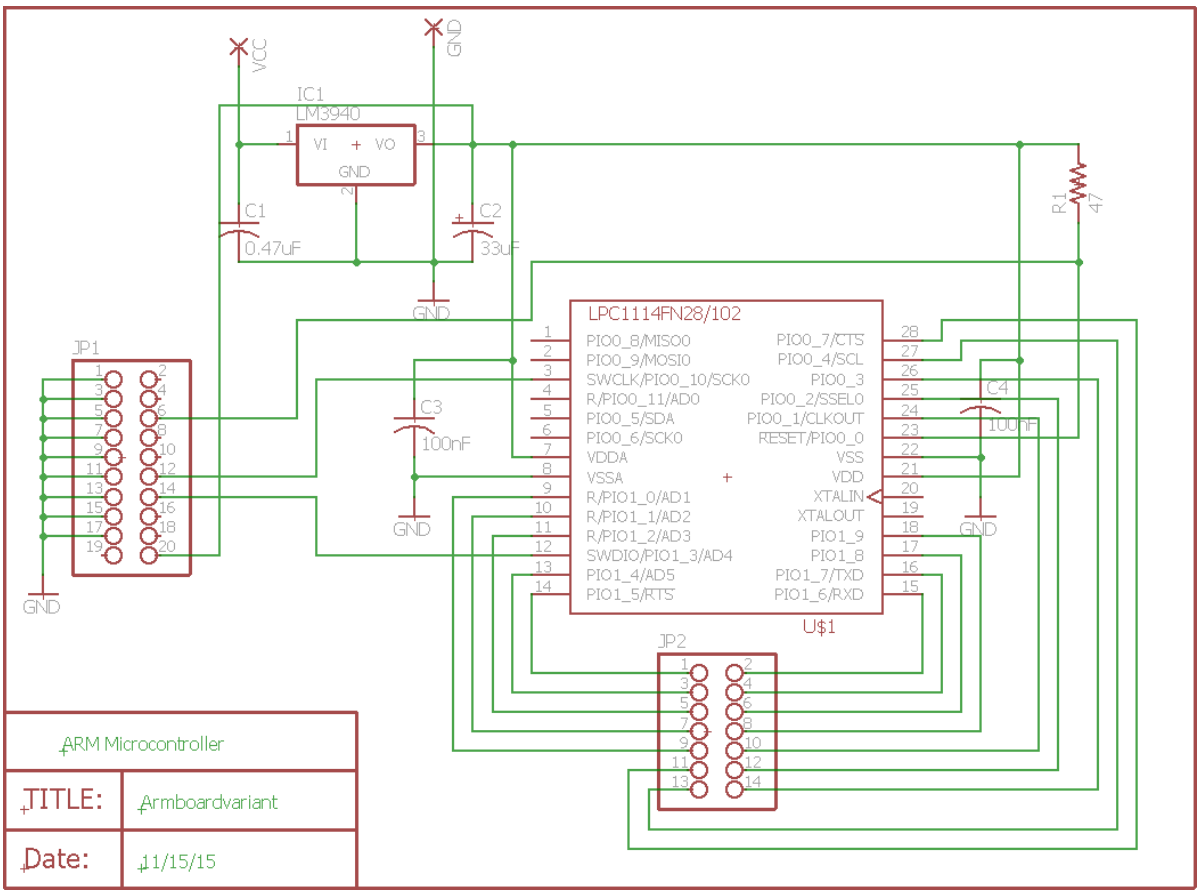

Figure 9: Board level schematic for ARM Microprocessor

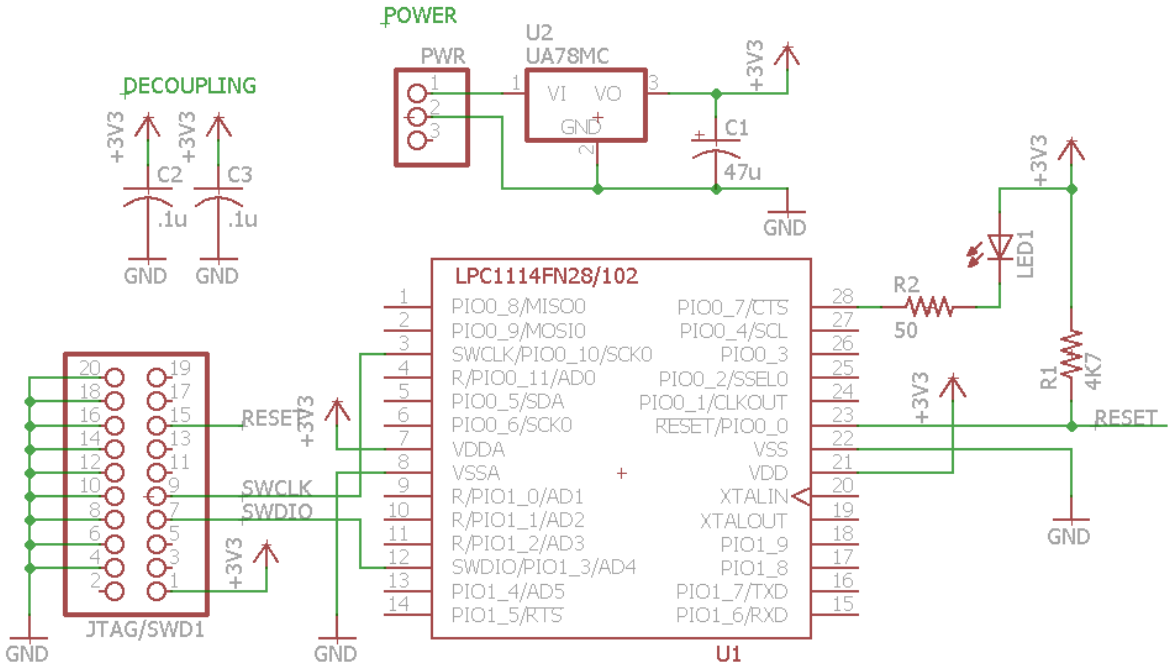

Figure 10: Board level schematic for ARM Microprocessor in breadboard verification stage

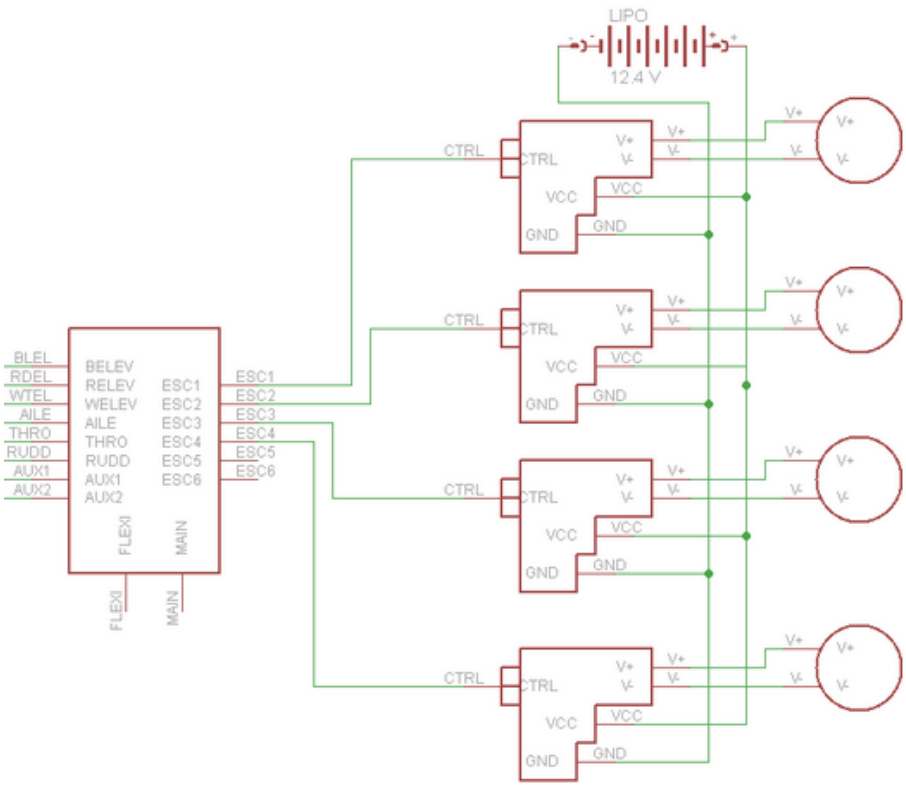

Figure 11: From left to right: CC3D, ESCs, Motors. 11.1-12.4 V LiPo batter at top

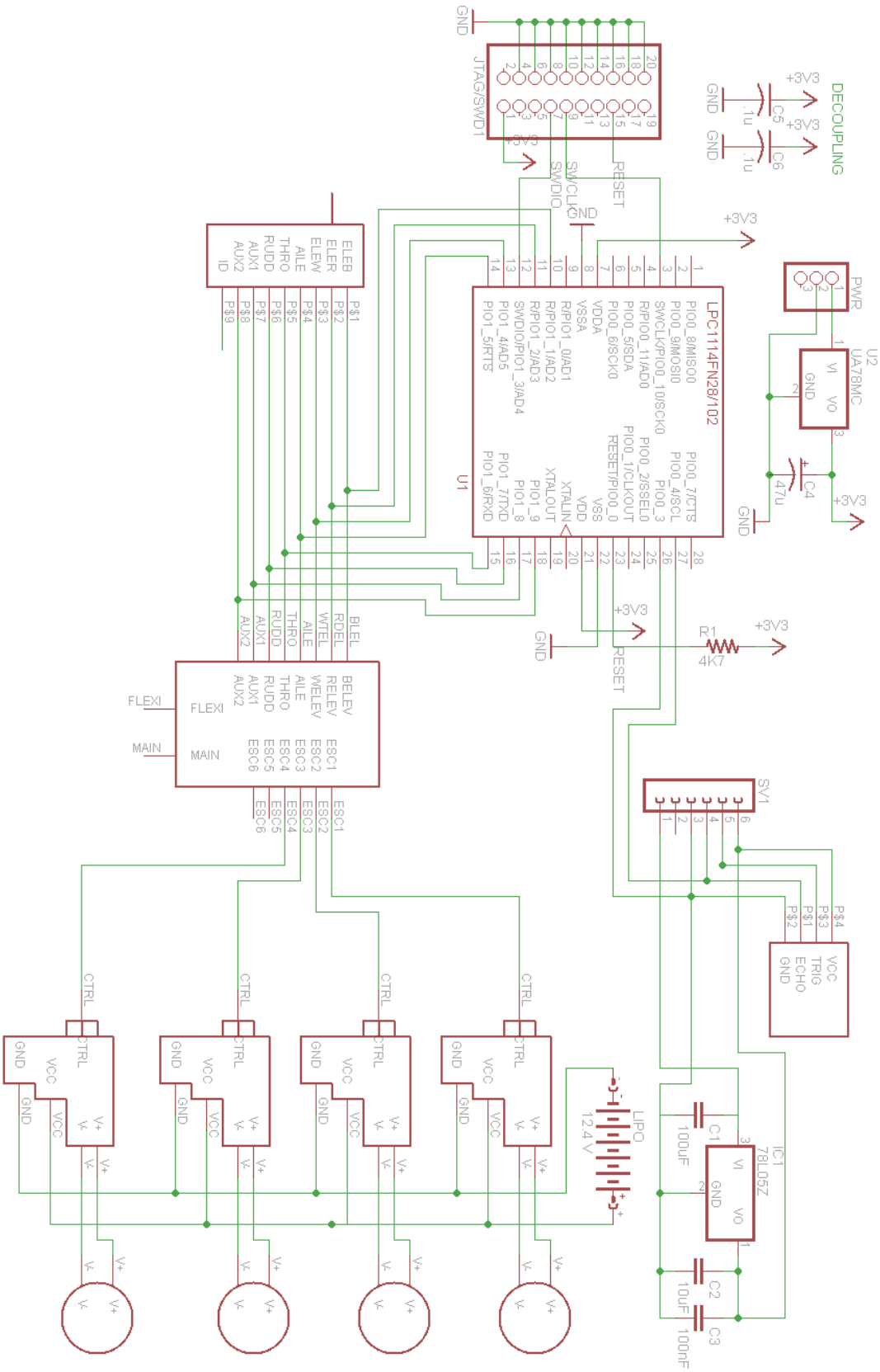

Figure 12: Full quadcopter schematic with obsolete power circuitry (replaced by BECs)

Appendix C: Acquired Ultrasonic Data

Table 5: Acquired Ultrasonic Distance Data

Distance (cm)	Expected Duration (µS)	Obtained Duration(µS)						
		1	2	3	4	5	Average	% Error
2.00	116.4	239.0	238.0	232.0	237.0	237.0	236.6	50.80
4.00	232.8	321.0	328.0	326.0	344.0	326.0	329.0	29.24
6.00	349.2	376.0	376.0	383.0	382.0	405.0	384.4	9.16
7.00	407.4	414.0	408.0	409.0	409.0	408.0	409.6	0.54
7.50	436.5	436.0	435.0	436.0	431.0	434.0	434.4	0.48
8.00	465.6	466.0	473.0	473.0	472.0	471.0	471.0	1.15
20.00	1164.0	1197.0	1203.0	1203.0	1228.0	1203.0	1206.8	3.55
50.00	2910.0	2965.0	2966.0	2989.0	2965.0	2966.0	2970.2	2.03
55.50	3230.1	3330.0	3281.0	3330.0	3304.0	3299.0	3308.8	2.38
75.00	4365.0	4448.0	4424.0	4426.0	4449.0	4455.0	4440.4	1.70
100+	>5820.0	Outputs random number possibly due to lack of power from arduino						

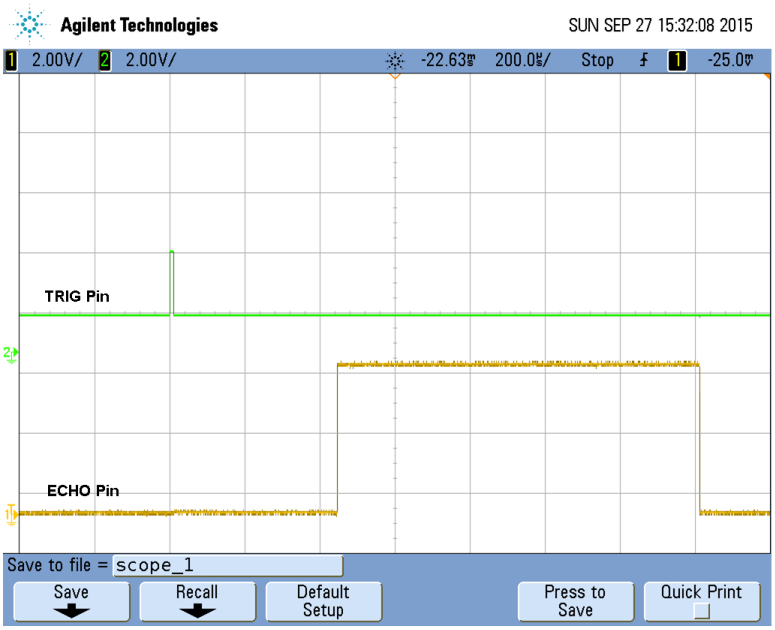

Figure 13: Waveform plot for short distance case

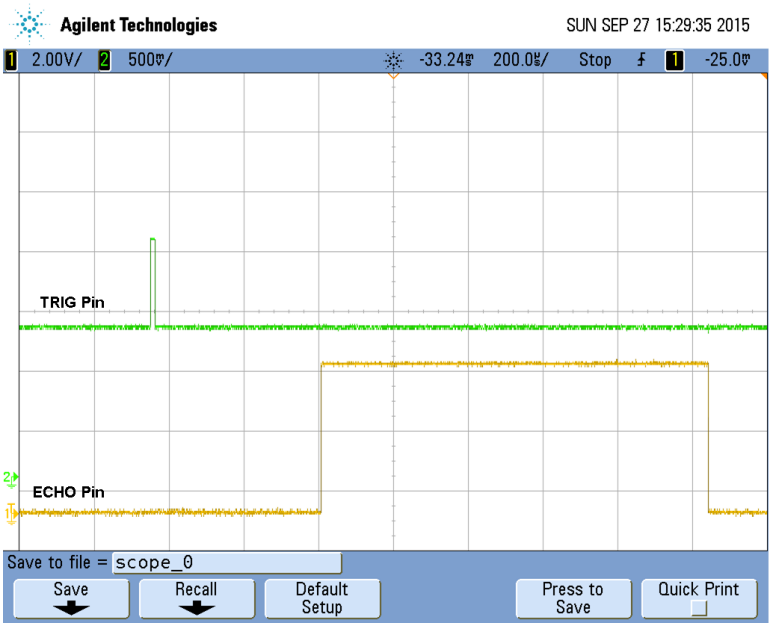

Figure 14: Waveform plot for long distance case

Appendix D: ARM Microcontroller code

ece445.h File

```
#ifndef ECE445_H_
#define ECE445_H_
#include <stdio.h>
#include "LPC11xx.h"
#define MULT 1
#define SYS_CLOCK 48000000
//#define ACCURACY 10000
#define ACCURACY 100000*MULT // 1/(10 micros)
#define SYS_TICK_RELOAD SYS_CLOCK/ACCURACY
#define THROTTLE_PIN 1UL<<1
#define ROLL_PIN 1UL<<2
#define PITCH_PIN 1UL<<4
#define YAW_PIN 1UL<<5
#define AUX1_PIN 1UL<<8
#define AUX2_PIN 1UL<<9
#define ALL THROTTLE_PIN|ROLL_PIN|PITCH_PIN|YAW_PIN|AUX1_PIN|AUX2_PIN
#define HOLD ROLL_PIN|PITCH_PIN|YAW_PIN
#define NUM_RECEIVER 6
#define MIN_PWM 100*MULT // 1.0 ms
#define MAX_PWM 190*MULT // 1.9 ms
#define TOT_PWM_SPACE MAX_PWM-MIN_PWM+1
#define PWM_WIDTH 2000*MULT // 20 ms
#define DEFAULT_PWM 150*MULT
#define UP_THROTTLE 167*MULT
//#define UP_THROTTLE 170
#define HOLD_THROTTLE 164*MULT
//#define DOWN_THROTTLE 144
#define DOWN_THROTTLE 158*MULT
#define R_PITCH 153*MULT
#define L_PITCH 144*MULT
#define HOLD_PITCH 150*MULT-1
#define R_ROLL 153*MULT
#define L_ROLL 145*MULT
#define HOLD_ROLL 150*MULT
#define HOLD_YAW 150*MULT
#define HOLD_PITCH_BALANCE 1*MULT
// Ultrasonic Sensor Part Settings
#define US_MAX_TIME 0x02DC6C00 // 1/(1/(48*10^6)) = 4.8e7, value in ticks
#define MIN_DIST 6 // in cm
#define MAX_DIST 100 // in cm
#define NO_OBJ 2 // Value to return if no object cant be more than MIN_DIST
#define OBJ 1
#define SENSOR_ERR -1 // Same as NO_OBJECT but cannot be equal to NO_OBJECT
#define US_STOP 1500 // us_sensor stops checking after 1500 micros (equivalent to
254 cm)
#define US_SLEEP 4000 // us_sensor is not usable for 40000 micros
// US_sampling Settings
#define US_SAMPLES 20
#define US_S_THRESHOLD 16
// others
#define _5_SECS 5*ACCURACY
#define _10_SECS 10*ACCURACY
```

```
#define _30_SECS 30*ACCURACY
#define _MS_TO_10MICROS 1000/ACCURACY
enum pwm{
       MIN,
       MAX,
       DEFAULT, // default
 UP, // up_throttle
       HD, //hd_throttle
       DOWN, // dn_throttle
       FRONT, //r_pitch
       HD_P, // hd_pitch
       BACK, //l_pitch
       RIGHT, // r_roll
       HD_R, // hd_roll
       LEFT, // l_roll
       HD_Y, // hd_yaw
       NUM_PWM
};
enum control{
       THROTTLE = 1,
       ROLL = 2,
       PITCH = 4,
       YAW = 5,
       AUX1 = 8,
       AUX2 = 9,
       SIZE = 10
};
enum us_sensor{
       OFF_SENSOR,
       ON_TRIG,
       WAIT_ECHO,
       DONE // stays in this state for a while before off
};
enum us_sampling{
       US_S_OFF,
       US_S_SAMPLING,
};
enum _test{
       _START,
       _FLY, // Flying phase, fly in small steps
       _FLY2, // Hover phase, hover a while to use us_sampling
       _HOVER,
       _DROP,
       //_REPLAN, // Revise decision
       _FINISH
};
void init(void);
void initGPIO(void);
void initControlPins(void);
int ARM(void);
int DISARM(void);
void loop(void);
void setpintime(enum control pin, enum pwm t);
unsigned int armed(void);
void ledOn(void);
void ledOff(void);
```

```
int up(double dist);
int front(double dist);
void us_sensor(void);
void us_sensor_s(void);
#endif
```

ece445.c File

```
#include "ece445.h"

unsigned int count = 0;
unsigned int arm_state = 0;
unsigned int controls[SIZE]; // Store PWM time
unsigned int pwm_states[TOT_PWM_SPACE+1]; // Store values to be sent to the
receiver pins
unsigned int pwm_values[NUM_PWM];
double x,y,z;
double goal_x = 0.0;
double goal_y = 0.0;
double goal_z = 25.0;
unsigned int time_taken = 0;
unsigned int curr_time = 0;
double frame_time = ((double)PWM_WIDTH)*1000.0/(double)ACCURACY; // 20ms

// throttle variables
int throttle_done = 0; // active low
int throttle_wait = 1;      // active low
enum pwm throttle_state;
double throttle_vel = 0.05; // 0.5 m/s = 0.05 cm/ms, throttle_vel1=
1/(throttle_vel*_MS_TO_10MICROS)
double throttle_vel1; // converted to multiplier to get count required
double throttle_start;
double throttle_sleep;
double throttle_dist = 0;

// pitch variables
int pitch_done = 0; // active low
int pitch_wait = 1;  // active low
enum pwm pitch_state;
double pitch_vel = 0.5; // 1 m/s = 0.1 cm/ms, pitch_vel1=
1/(pitch_vel*_MS_TO_10MICROS);
double pitch_vel1; // converted to multiplier to get count required
double pitch_start; // counter
double pitch_sleep; // counter
double pitch_dist = 0;
double pitch_extra_delay = 0; // this adds to pitch_sleep for extra delay
int HOLD_PITCH_RATIO1 = 3;
int HOLD_PITCH_RATIO2 = 2;
//int HOLD_PITCH_TOTAL = HOLD_PITCH_RATIO1 + HOLD_PITCH_RATIO2;
int pitch_hold_switch_count;
int pitch_hold_switch_type = 0;

// roll variables
int roll_done = 0; // active low
int roll_wait = 1;   // active low
enum pwm roll_state;
```

```
double roll_vel = 0.05; // 1 m/s = 0.1 cm/ms, pitch_vel1=
1/(pitch_vel*_MS_TO_10MICROS);
double roll_vel1; // converted to multiplier to get count required
double roll_start; // counter
double roll_sleep; // counter
double roll_dist = 0;
double roll_extra_delay = 0; // this adds to pitch_sleep for extra delay
int finish = 1;

// Test variables
enum _test test_state  = _START;
int hold_sleep = 0;

// ultrasonic sensor variables
enum us_sensor us_sensor_state = OFF_SENSOR;
int us_on_sensor = 1; // active low
int us_running = 1; // active low
int us_trig_sleep = 1; // active low
int us_done = 1; //active low
int us_echo_time; // timer
int distance = 0;
int us_sleep = 0;
int us_ready = 0; //active low

// sampling variables
enum us_sampling us_s_sampling_state = US_S_OFF;
int us_s_did_sampling = 1; // check if recently did sampling, active low
int us_s_on_sampling = 1; // start sampling, active low
int us_s_running = 1; // currently sampling, active low
int us_s_n_samples; // number of samples done
int us_s_positive_samples;
int us_s_curr_time = 0;
int us_s_time_taken = 0;

void SysTick_Handler()
{
	count = (count + 1) % PWM_WIDTH;

	if ((count) < MIN_PWM)
	{
		LPC_GPIO1->DATA |= ALL;
	}
	else if ((count) <= MAX_PWM)
	{
		if (pitch_state == HD_P && (count & (HOLD_PITCH|HOLD_PITCH_BALANCE))
&& pitch_hold_switch_type == 1)
		{
			LPC_GPIO1->DATA &= ~(pwm_states[(count) - MIN_PWM] | (1UL <<
PITCH_PIN));
		}
		else if (pitch_state == HD_P && (count & HOLD_PITCH) &&
pitch_hold_switch_type==0)
		{
			LPC_GPIO1->DATA &= ~(pwm_states[(count) - MIN_PWM] | (1UL <<
PITCH_PIN) );
		}
```

```
                else
                {
                        LPC_GPIO1->DATA &= ~(pwm_states[(count) - MIN_PWM]);
                }
        }

        if ((LPC_GPIO1->DATA & 0x00000001) == 1)
        {
                ++us_echo_time;
        }

        if (pitch_hold_switch_count == 0)
        {
                pitch_hold_switch_type ^= 1;

                if (pitch_hold_switch_type & 0x00000001)
                        pitch_hold_switch_count = HOLD_PITCH_RATIO2;

                else
                        pitch_hold_switch_count = HOLD_PITCH_RATIO1;
        }

        --us_trig_sleep;
        --us_sleep;

        --throttle_sleep;
        --hold_sleep;

        ++us_s_curr_time;

        ++curr_time;
        --pitch_hold_switch_count;

        return;
}
void initGPIO()
{
        // Note: Copied from blinky code. Need to modify!
        LPC_SYSCON->SYSAHBCLKDIV = 1UL;


        //enable clocks to GPIO block
        LPC_SYSCON->SYSAHBCLKCTRL |= (1UL <<  6);
        LPC_SYSCON->SYSAHBCLKCTRL |= (1UL <<  16);

        // Special Case, IOCON not initially PIO
        LPC_IOCON->R_PIO1_0 |= 0x1;
        LPC_IOCON->R_PIO1_1 |= 0x1;
        LPC_IOCON->R_PIO1_2 |= 0x1;
        LPC_GPIO0->DIR |= (1<<3); // Trigger pin
        LPC_GPIO0->DIR |= (1<<2);//16B0Cap0, LED test pin
          LPC_GPIO1->DIR |= (1<<9);
                LPC_GPIO1->DIR |= (1<<8);
                LPC_GPIO1->DIR |= (1<<7);
                LPC_GPIO1->DIR |= (1<<6);
          LPC_GPIO1->DIR |= (1<<5);
```

```
              LPC_GPIO1->DIR |= (1<<4);
          //LPC_GPIO1->DIR |= (1<<3);
              LPC_GPIO1->DIR |= (1<<2);
         LPC_GPIO1->DIR |= (1<<1);
        LPC_GPIO1->DIR &= ~(1<<0); // Same as echo pin, unusable
              LPC_GPIO1->DATA = 0;
}
void initControlPins()
{
       int i;

       //initialize pwm_values
       pwm_values[MIN] = MIN_PWM - MIN_PWM;
       pwm_values[MAX] = MAX_PWM - MIN_PWM;
       pwm_values[DEFAULT] = DEFAULT_PWM - MIN_PWM;
       pwm_values[HD] = HOLD_THROTTLE - MIN_PWM;
       pwm_values[UP] = UP_THROTTLE - MIN_PWM;
       pwm_values[DOWN] = DOWN_THROTTLE - MIN_PWM;
       pwm_values[FRONT] = R_PITCH - MIN_PWM;
       pwm_values[HD_P] = HOLD_PITCH - MIN_PWM;
       pwm_values[BACK] = L_PITCH - MIN_PWM;
       pwm_values[LEFT] = R_ROLL - MIN_PWM;
       pwm_values[HD_R] = HOLD_ROLL - MIN_PWM;
       pwm_values[RIGHT] = L_ROLL - MIN_PWM;
       pwm_values[HD_Y] = HOLD_YAW - MIN_PWM;

       // initialize all controls to 1.5ms
       for (i = 0; i <= SIZE; ++i)
       {
              controls[i] = pwm_values[DEFAULT];

       }

       throttle_state = DEFAULT;

       for (i = 0; i <= TOT_PWM_SPACE; ++i)
       {
              pwm_states[i] = 0;
       }

       pwm_states[pwm_values[DEFAULT]] = ALL;

       throttle_vel1= 1/(throttle_vel*_MS_TO_10MICROS);
       pitch_vel1= 1/(pitch_vel*_MS_TO_10MICROS);

       pitch_hold_switch_count = HOLD_PITCH_RATIO1;
}
void ledOn()
{
       LPC_GPIO0->DATA |= 0x00000004;
}
void ledOff()
{
       LPC_GPIO0->DATA &= ~(0x00000004);
}
unsigned int armed()
```

```
{
	return arm_state;
}
void init()
{
	//	ledOn();

	x=y=z=0.0;

	SystemInit();
	initGPIO();
	initControlPins();
	ledOff();

	if (SysTick_Config(SYS_TICK_RELOAD))
	{
		ledOn();
		while(1);
	}

	us_sensor_state = OFF_SENSOR;
	//ledOn();
}
int ARM()
{
	int i;



	if ((count) > MAX_PWM)
	{
		for (i = 0; i < 50; ++i)
		{
			while ((count) > MAX_PWM);

			while ((count) <= MAX_PWM);
		}

		setpintime(THROTTLE,MIN);
		throttle_state = MIN;
	setpintime(YAW,MAX);
	setpintime(AUX1,MIN);

		for (i = 0; i < 50; ++i)
		{
			while ((count) > MAX_PWM);

			while ((count) <= MAX_PWM);
		}

		//setpintime(THROTTLE,DOWN);
		setpintime(ROLL,HD_R);
		setpintime(PITCH,HD_P);
		setpintime(YAW,HD_Y);
	//setpintime(YAW,DEFAULT);
		//setpintime(AUX2,MIN);
```

```
		arm_state = 1;

		return 0;
	}
	else
		return 1;
}
int DISARM()
{
	int i = 0;

	if ((count) > MAX_PWM)
	{
		setpintime(THROTTLE, MIN);
		throttle_state = MIN;
		setpintime(YAW, MAX);
		setpintime(AUX1, MAX);

		for (i = 0; i < 50; i++)
		{
			while ((count) > MAX_PWM);

			while ((count) <= MAX_PWM);
		}

		//setpintime(THROTTLE,DOWN);
		setpintime(YAW, DEFAULT);
		arm_state = 0;

		return 0;
	}
	else
		return 1;
}
void setpintime(enum control pin, enum pwm t)
{
	if (pwm_states[controls[pin]] == pwm_values[t]) // already set to t
			return;

	pwm_states[controls[pin]] &= ~(1UL << pin); // zero the pin in previous t
	pwm_states[pwm_values[t]] |= 1UL << pin;
	controls[pin] = pwm_values[t]; // set new t of receiver pin

	return;
}
int up(double dist)
{
	int time_taken;

	switch (throttle_state)
	{
		case UP:
			if (throttle_sleep <= 0)
			{
				setpintime(THROTTLE,HD);
				time_taken = throttle_start - throttle_sleep;
```

```
                        throttle_state = HD;
                        z += time_taken/throttle_vel1;
                        return 0;
                }
                break;

        case DOWN:
                if (throttle_sleep <= 0)
                {
                        setpintime(THROTTLE,HD);
                        time_taken = throttle_start - throttle_sleep;
                        throttle_state = HD;
                        z -= time_taken/throttle_vel1;
                        return 0;
                }
                break;

        case HD:
                if (throttle_wait || throttle_sleep <= 0)
                {
                        throttle_wait = 1;
                        if (dist > 0)
                        {
                                setpintime(THROTTLE,UP);
                                throttle_state = UP;
                                throttle_start = ((double)dist)*throttle_vel1;
                                throttle_sleep = throttle_start;
                        }
                        else if (dist < 0)
                        {
                                setpintime(THROTTLE,DOWN);
                                throttle_state = DOWN;
                                throttle_start = -
1.0*((double)dist)*throttle_vel1;
                                throttle_sleep = throttle_start;

                        }
                }
                break;
        default:
                setpintime(THROTTLE,HD);
                throttle_state = HD;
                throttle_sleep = (int)throttle_vel1;
                throttle_wait = 0;
                break;
    }

    return 1;
}
int front(double dist)
{
    int time_taken;

    switch (pitch_state)
    {
        case FRONT:
```

```
                        if (pitch_sleep <= 0)
                        {
                                setpintime(PITCH,DEFAULT);
                                time_taken = pitch_start - pitch_sleep;
                                pitch_state = DEFAULT;
                                x += time_taken/pitch_vel1;
                                return 0;
                        }
                        break;

                case BACK:
                        if (pitch_sleep <= 0)
                        {
                                setpintime(PITCH,DEFAULT);
                                time_taken = pitch_start - pitch_sleep;
                                pitch_state = DEFAULT;
                                x -= time_taken/pitch_vel1;
                                return 0;
                        }
                        break;

                case DEFAULT:
                        if (pitch_wait ||    pitch_sleep <= 0)
                        {
                                pitch_wait = 1;
                                if (dist > 0)
                                {
                                        setpintime(PITCH,FRONT);
                                        pitch_state = FRONT;
                                        pitch_start = ((double)dist)*pitch_vel1;
                                        pitch_sleep = pitch_start;
                                }
                                else if (dist <= 0)
                                {
                                        setpintime(PITCH,BACK);
                                        pitch_state = BACK;
                                        pitch_start = -1.0*((double)dist)*pitch_vel1;
                                        pitch_sleep = pitch_start;
                                }
                        }
                        break;
                default:
                        setpintime(PITCH,DEFAULT);
                        pitch_state = DEFAULT;
                        pitch_sleep = (int)pitch_vel1;
                        pitch_wait = 0;
                        break;
        }

        return 1;
}
int right(double dist)// roll
{
        int time_taken;

        switch (roll_state)
```

```
	{
	case FRONT:
		if (roll_sleep <= 0)
		{
			setpintime(ROLL, HD_R);
			time_taken = roll_start - roll_sleep;
			roll_state = HD_R;
			y += time_taken / roll_vel1;
			return 0;
		}
		break;

	case BACK:
		if (roll_sleep <= 0)
		{
			setpintime(ROLL, HD_R);
			time_taken = roll_start - roll_sleep;
			roll_state = HD_R;
			y -= time_taken / roll_vel1;
			return 0;
		}
		break;

	case HD_R:
		if (roll_wait || roll_sleep <= 0)
		{
			roll_wait = 1;
			if (dist > 0)
			{
				setpintime(ROLL, RIGHT);
				roll_state = FRONT;
				roll_start = ((double)dist)*roll_vel1;
				roll_sleep = roll_start;
			}
			else if (dist <= 0)
			{
				setpintime(ROLL, LEFT);
				roll_state = BACK;
				roll_start = -1.0*((double)dist)*roll_vel1;
				roll_sleep = roll_start;
			}
		}
		break;
	default:
		setpintime(ROLL, HD_R);
		roll_state = HD_R;
		roll_sleep = (int)roll_vel1;
		roll_wait = 0;
		break;
	}
	return 1;
}
void us_sensor()
{
	switch (us_sensor_state)
	{
```

```
		case OFF_SENSOR:
			if (us_on_sensor == 0)
				us_on_sensor = 1;
				us_sensor_state = ON_TRIG;
				LPC_GPIO0->DATA |= 0x00000008;
				us_trig_sleep = 1;
				us_done = 1;
				//us_ready = 1;
				us_running = 0;
				break;

		case ON_TRIG:
			if (us_trig_sleep <= 0)
			{
				LPC_GPIO0->DATA &= ~(0x00000008);
				us_echo_time = 0;
				us_sensor_state = WAIT_ECHO;
			}
			break;

		case WAIT_ECHO:
			if (((LPC_GPIO1->DATA & 0x00000001) == 0 && us_echo_time != 0)
|| us_echo_time >= US_STOP)
			{
				//ledOn();
				// 2/((340.27m/s)*10^(2 - 6)) = 58.78 micros/cm
				distance = (us_echo_time*10) / 59.0;

				if ((distance < MIN_DIST))
				{
					distance = NO_OBJ;
					//ledOff();
				}
				else if ((distance > MAX_DIST))
				{
					distance = NO_OBJ;
					//ledOff();
				}
				else
				{
					distance = OBJ;
					//ledOn();
				}
				us_running = 1;
				us_done = 0;
				us_sensor_state = DONE;
				us_sleep = US_SLEEP;
			}
			break;
		case DONE:
			if(us_sleep <= 0)
			{
				us_sensor_state = OFF_SENSOR;
				us_ready = 0;
			}
			break;
```

```
	}
}
void us_sensor_s()
{
	switch (us_s_sampling_state)
	{
		case US_S_OFF:
			if (us_s_on_sampling == 0)// && throttle_state == HD &&
(pwm_states[pwm_values[DEFAULT]] & HOLD))
			{
				us_s_on_sampling = 1;
				us_s_n_samples = 0;
				us_s_positive_samples = 0;
				us_done = 1;
				us_on_sensor = 0;
				us_s_running = 0;
				us_s_sampling_state = US_S_SAMPLING;
				us_s_curr_time = 0;
			}
			break;

		case US_S_SAMPLING:
			if (us_done == 0)
			{
				// update sampling data
				us_s_n_samples++;

				if (distance == OBJ)
				{
					us_s_positive_samples++;
				}

				us_done = 1;

				if (us_s_n_samples == US_SAMPLES)
				{
					// determine result of sampling
					if (us_s_positive_samples < US_S_THRESHOLD)
					{
						distance = NO_OBJ;
						ledOff();
					}
					else
					{
						distance = OBJ;
						ledOn();
					}

					// Wrap up
					us_s_did_sampling = 0;
					us_s_running = 1;
					us_s_time_taken = us_s_curr_time;
					us_s_sampling_state = US_S_OFF;
				}

				else
```

```
                        {
                                // restart sampling
                                us_on_sensor = 0;
                        }
                }
        }
}
void loop()
{
        switch (test_state)
        {
        case _START:
                us_s_did_sampling = 1;
                us_s_on_sampling = 0;
                test_state = _FLY2;

                break;

        case _FLY:
                if (throttle_done == 0)
                {
                        if (z < goal_z && x != goal_x && y != goal_y)
                        {
                                us_s_did_sampling = 1;
                                us_s_on_sampling = 0;
                                test_state = _FLY2;
                        }
                        else
                        {
                                test_state = _HOVER;
                                hold_sleep = _30_SECS;
                        }
                }
                break;

        case _FLY2:
                if (us_s_did_sampling == 0)
                {
                        us_s_did_sampling = 1;

                        if (z < goal_z)
                        {
                                if ((((goal_z - z) / 5) < 1) && (((goal_z - z) / 5) > -
1))
                                {
                                        throttle_dist = goal_z - z;
                                }
                                else if (goal_z - z > 0)
                                {
                                        throttle_dist = 5;
                                }
                                else
                                {
                                        throttle_dist = -5;
                                }
```

```
                        throttle_done = 1;
                }

                if (x != goal_x)
                {
                        if ((((goal_x - x) / 5) < 1) && (((goal_x - x) / 5) > -
1))
                        {
                                pitch_dist = goal_x - x;
                        }
                        else if (goal_x - x > 0)
                        {
                                pitch_dist = 5;
                        }
                        else
                        {
                                pitch_dist = -5;
                        }

                        pitch_done = 1;
                }

                if (y != goal_y)
                {
                        if ((((goal_y - y) / 5) < 1) && (((goal_y - y) / 5) > -
1))
                        {
                                roll_dist = goal_y - y;
                        }
                        else if (goal_y - y > 0)
                        {
                                roll_dist = 5;
                        }
                        else
                        {
                                roll_dist = -5;
                        }

                        roll_done = 1;
                }

                test_state = _FLY;
        }
        break;

    case _HOVER:
        if (hold_sleep <= 0)
        {
                throttle_dist = -goal_z - 10; // 10 is just some offset for
extra assurance
                throttle_done = 1;
                test_state = _DROP;
        }
        break;

    case _DROP:
```

```
			if (throttle_done == 0)
			{
				DISARM();
				test_state = _FINISH;
			}
			break;

		case _FINISH:
			break;
		}

		if ((us_on_sensor == 0 && (pwm_states[pwm_values[DEFAULT]] & HOLD)) ||
us_running == 0)
		{
			us_sensor();
		}

		if (us_s_on_sampling == 0 || us_s_running == 0)
		{
			us_sensor_s();
		}

		if ((count) > MAX_PWM)
		{
			//ledOn();

			if (throttle_done)
			{
				throttle_done = up(throttle_dist);

				if (throttle_done == 0)
				{
					throttle_dist = 0;
					//us_on_sensor = 0;
				}
			}

			if (roll_done)
			{
				roll_done = right(roll_dist);

				if (roll_done == 0)
				{
					roll_dist = 0;
					//us_on_sensor = 0;
				}
			}

			if (pitch_done)
			{
				pitch_done = front(pitch_dist);

				if (pitch_done == 0)
				{
					pitch_dist = 0;
					//us_on_sensor = 0;
```

```
                        }
                }
                //ledOff();
        }
}
```

main.c File

```
#include "ece445.h"
int main ()
{
        int i;

        for (i = 0; i < 10000000; i++);
        //for (i = 0; i < 10000000; i++);

        init();
        while(ARM());

        while(armed())
        {
                loop();
        }

        while(1);
        return 0;
}
```