School of Engineering

Design of Multifunctional Quadcopter System

Sami Khaled Nofal (0138086) **Ahmad Mohammad Al Zayyoud (0134171)**

Amer Jamal Radwan (0142740)

Supervised by: Othman MK Alsmadi

Submitted in partial fulfillment of the requirements of B.Sc. Degree in Electrical Engineering

August 20, 2019

Students Statement

We, the undersigned students, certify and confirm that the work submitted in this project report is entirely our own and has not been copied from any other source. Any material that has been used from other sources has been properly cited and acknowledged in the report.

We are fully aware that any copying or improper citation of references/sources used in this report will be considered plagiarism, which is a clear violation of the Code of Ethics of the University of Jordan.

In addition, we have read and understood the legal consequences of committing any violation of the University of Jordan Code of Ethics.

Sami Khaled Nofal (0138086) **Ahmad Mohammad Al Zayyoud (0134171)**

Amer Jamal Radwan (0142740)

Supervisor Certification

This to certify that the work presented in this senior year project manuscript was carried out under my supervision, which is entitled:

"**Design of Multifunctional Quadcopter System**"

Sami Khaled Nofal (0138086) **Ahmad Mohammad Al Zayyoud (0134171)**

Amer Jamal Radwan (0142740)

Dr. Othman MK Alsmadi,

Examining Committee Evaluation Guidelines

Abstract and Introduction	**L M H**
Abstract is complete, concise, specific and *Self-sufficient*	)
Stating *problem definition* and *objectives* of work	)
Providing *background* information	)
Presenting *methods used in solution*	)
Including *previous work* and *citation*	)
Methodology	
Effectiveness of the proposed solution	)
Logical sequence or flow of report information	)
Technical *content accuracy* and engineering soundness	)
Results and Discussion	
Results presentation or demonstration	)
Results analysis and interpretation	)
Sample calculation, *error analysis, and trend* analysis	)
Use of *graphs and illustrations*	
Conclusions and Future work	
Stressing the significance of the work	)
Interpreting the results and the implications of the results	)
Including directions or actions to be taken for future work	)
ABET and References	
Impact of Eng. solutions: global, economical, environmental and societal	)
Implementation of citation / referencing and references quality	)

Examining Committee Member

Remarks and Notes

Dedication

We dedicate this project to our parents who helped and motivated us to complete the senior year project. To our supervisor Dr Othman MK. Alsmadi who supported us during these two semesters of planning and design. To my sister Noor Nofal and her spouse Ahmad Ziadeh who assisted us in delivering most of the components all the way from America – We are happy you guys didn't get in trouble at the airport gate. To my two best friends Osamah Madie and Sammy Saleh who supported me and made fun of my project – "I'll make sure the drone hunts your dreams" – Sami Nofal. To my brother Omar Nofal for advising and guiding me through my engineering journey.

Sami Khaled Nofal (0138086) **Ahmad Mohammad Al Zayyoud (0134171)**

Amer Jamal Radwan (0142740)

Acknowledgment

We would like to express our deepest appreciation to all those who provided us the possibility to complete this report. A special gratitude we give to our final year project supervisor, Dr Othman Alsmadi, whose contribution in stimulating suggestions and encouragement, helped us to coordinate our project especially in writing this report and investing his full effort in guiding the team in achieving the goal.

Furthermore we would also like to acknowledge with much appreciation the crucial role of the engineering staffs who gave us the permission to use all required equipment and the necessary materials to complete the project. A special thanks goes to Mitch Tabian an Android developer whose online tutorials helped us in the completion of the application. Last but not least, many thanks go to Drs Raghad Hemiemat whose effort in providing clear documentation guidelines helped us managing the senior year project report professionally. We have to appreciate the guidance given by other supervisors as well as the panels especially in our project presentation that has improved our presentation skills thanks to their comments and advices.

Sami Khaled Nofal (0138086) **Ahmad Mohammad Al Zayyoud (0134171)**

Amer Jamal Radwan (0142740)

Abstract

The development and investigation of autonomous flight systems have increased lately because of the increasing amount of applications of unmanned aerial vehicle (UAV) in military fields such as intelligence, surveillance, and reconnaissance missions, and in civil fields like aerial surveillance, aerial photography and video, firefighting, and many others that are emerging. The platforms that have caught our attention for investigation projects are multirotor helicopters, in particular quadcopters. Quadcopters have a simple, lightweight structure and a great flight capacity and maneuverability, such as vertical take-off and landing and hovering flight. These advantages allows quadcopters to be used in complex, tight, and dangerous environments.

The goal of this project is to design and build a semi-autonomous multifunctional quadcopter capable of self-sustained flight via wireless communication while utilizing a microcontroller. The quadcopter should be able to collect numerous ambient data about the environment and report them back to the user. Due to the noticeable growth of digital technology and the integration of smartphones, developing an application dedicated for the quadcopter has been one of the main goals from the start, and with such a user can have more controllability and functionality over the rotorcraft than ever before allowing complex computation to be done within the smartphone itself for testing and evaluation purposes if needed on the go.

It is hoped that this study will allow us to have a better insight on the working principle of multirotor helicopters and how we can apply our concepts and methods we acquired in the fields of control, electronics, and communication courses.

KEYWORDS: Drone/Quadcopter, Multirotor, Quadrotor.

Table of Contents

Chapter Page

List of Tables

CHAPTER 1

INTRODUCTION

1.1 Brief History of Quadcopter

Drones and quadcopters have revolutionized flight. They help humans to take to the air in new, profound ways. Drones belong to a class of aerial vehicles known as Unmanned Aerial Vehicles (UAVs). These vehicles can take to the air without pilots. Essentially, this makes drones a flying robots. Encompassing both planes and quadcopters, they have software-controlled flight algorithm integrated into their systems. These systems work with Global Positioning Technology (GPS) to guide and track their movements. The quadcopter is a newer UAV. As its name suggests, this rotorcraft (a craft lifted by spinning rotor blades) depends on four quick-turning rotors to give it thrust. Two spin clockwise and the other two, counter-clockwise. Two sets of identical, fixed pitch propellers help the process. Pilots achieve control of the craft by using remote control transmitters to change the speed of its rotor discs. Quadcopters were among the first vertical take-off and landing vehicles (VTOLs). Earlier helicopters used tail rotors to counterbalance the torque, or rotating force, generated by a single, main rotor. This was wasteful and inefficient. Later on engineers developed quadcopters to solve the problems that helicopter pilots had with making vertical flights. The first was the Oehmichen 2, invented in 1920 by the French engineer Etienne Oehmichen. [1]

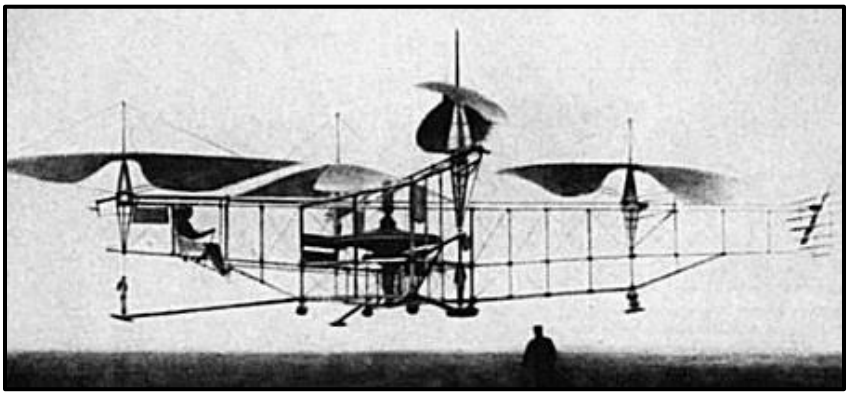

Figure 1.1 – Oehmichen No.2, 1920

Around the same time Dr. George de Bothezat and Ivan Jerome developed the de Bothezat helicopter aircraft, with six-bladed rotors at the end of an X-shaped structure. Two small propellers with variable pitch were used for thrust and yaw control. The vehicle used collective pitch control. Built by the US Air Service, it made its first flight in October 1922. About 100 flights were made by the end of 1923. The highest it ever reached was about 5 m (16 ft. 5 in). Although demonstrating feasibility, it was underpowered, unresponsive, mechanically complex and susceptible to reliability problems. Pilot workload was too high during hover to attempt lateral motion.

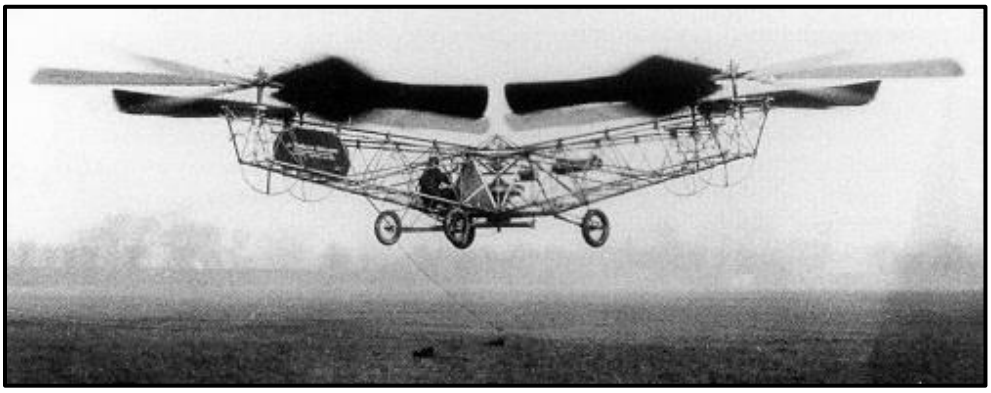

Figure 1.2 – De Bothezat Helicopter, 1923

Early quadcopters would typically have the engine sitting somewhere centrally in the fuselage of the copter, driving the 4 rotors via belts or shafts. Belts and shafts however are heavy and importantly, subject to breakage. As the 4 rotors of a quadcopter are all slightly different from each other, a quadcopter is not naturally stable, simply running 4 rotors at the same speed, while producing enough lift to hover the copter, does not produce stable flight. On the contrary, quadcopters have to be constantly stabilized. In the absence of computers, this meant a monumental workload for the pilot. [2]

As a result, multicopter designs were abandoned in favor of single, or on rare occasions for very large transport helicopters, double rotor designs.

Technology has advanced quadcopters dramatically with the advent of electric motors and especially microelectronics and micromechanical devices, a few years ago it became

possible to build reliable and efficient multirotors. Modern multicopters have an electric motor mated to each rotor, sitting directly below or above it. A flight computer constantly monitors the orientation of the copter and corrects for instability by changing not the pitch of the rotors but simply the rpm of the individual motors/rotors. This fixed pitch design is much simpler than the complex swash plate mechanics that are required for single rotor helicopters. [3]

1.2 Project Overview

This project includes designing and building a multifunctional quadcopter that can be used as a replacement for examining dangerous areas that otherwise would be difficult for an individual to investigate on his/her own by providing an aerial view of a given situation, disaster monitoring and surveying hazardous locations; additionally, the quadcopter has a custom-built dedicated Android application that permits the user to have more controllability and functionality to evaluate, test, and display data acquisition graphs within the smartphone itself. The application also enables the user to use the phone as means of controlling the movement of the drone incase an RF controller is not available.

1.2.1 Scope of the Problem

Multirotors on paper may seem to be the ideal solution in aerial flights; however, like many things in life nothing is perfect. In designing a quadcopter, one should take account the main problem that he/she will face during the flight simulation and that is the stability of the device. Multicopters are aerodynamically unstable and absolutely require an on-board computer and sensors in order to provide zero error steadiness. A simple negative feedback controller will not be able to achieve a smooth, even flight and may be susceptible to collisions and/or flight crashes.

To understand the problem of our goal, a well-documented mathematical representation of the quadcopter must be introduced to our design to detect the flaws of the system and how to

eliminate them; therefore, we challenge ourselves to realize how to solve real world problems using engineering experiences and skills obtained.

1.2.2 Objectives

The main objective of this project is to develop a quadcopter that allows itself to be capable of self-sustained flight using both mathematical models and software and hardware development to achieve the desired result. The project revolves around how to apply control theory rules to manage the quadcopter to be stable during an aerial flight, understand the laws of aerodynamics that governs the quadcopter, acknowledge how a PID controller works and how to implement it into the software branch, showcase our engineering skills in both electronics and communication fields, and design and develop a custom application to allow the user to have a more seamless and functional drone.

1.2.3 Method of Approach

Up until now we discussed the problems and the key solutions to our target goal. To approach the issues and complications we first begin to understand how to transform the design into a block diagrams which then can be used in multiple simulating programs to learn how to overcome the obstacles. Simulations such as the one given in MatLab Simulink can interpret the system into different stage blocks using transfer functions to conduct the experiments and evaluate the problem. From here a student can manage to perfect the system to his/her desire and hence to implement the logic into a readable code that can then be executed into the microcontroller.

It is required to have basic programming knowledge of embedded systems in order to achieve an efficient, clean, and fast executable code to prevent any logical or hardware errors during the time of flight.

It should be noted that the flight time depends on multiple factors including the efficiency of the power supplied to the circuitry. A well designed power electronics circuit must be taken into account to produce an effective, and energy-saving solution for the maximum amount of flight time the quadcopter can handle.

CHAPTER 2

MODELING AND DESIGN

2.1 Quadcopter Dynamics

In order to understand the basic working principle of quadcopters, a representation of quadcopter dynamics and control concept should be covered before any further advancement to design and control algorithms. A multirotor craft consisting of four motors can have either an ‘x’ (cross) or ‘+’ (plus) formation depending on the desired design. Both cross and plus designs bare resemblance with each other; however, they differ in terms of flight maneuverability. In order to understand more on how these two formations work, Figures 2.1 to 2.4 summarize the basic flight movements for each one as shown below:

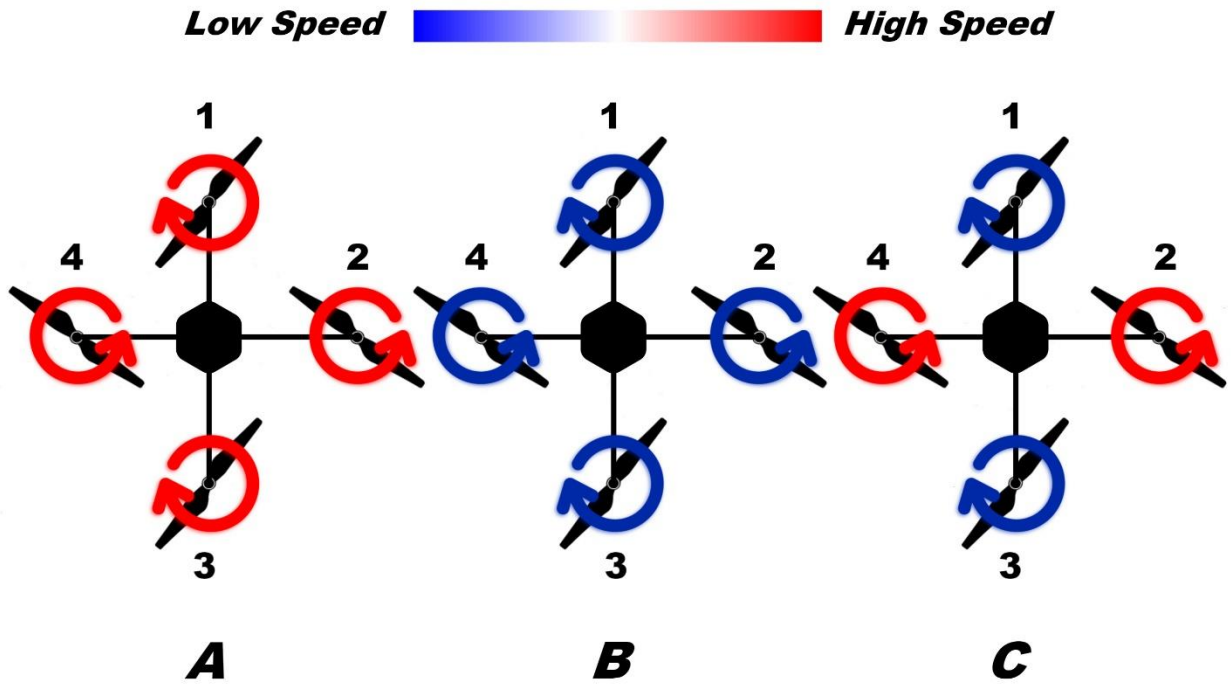

Figure 2.1 – Plus Formation Flight Movements. A: Upward, B: Downward, C: Clockwise.

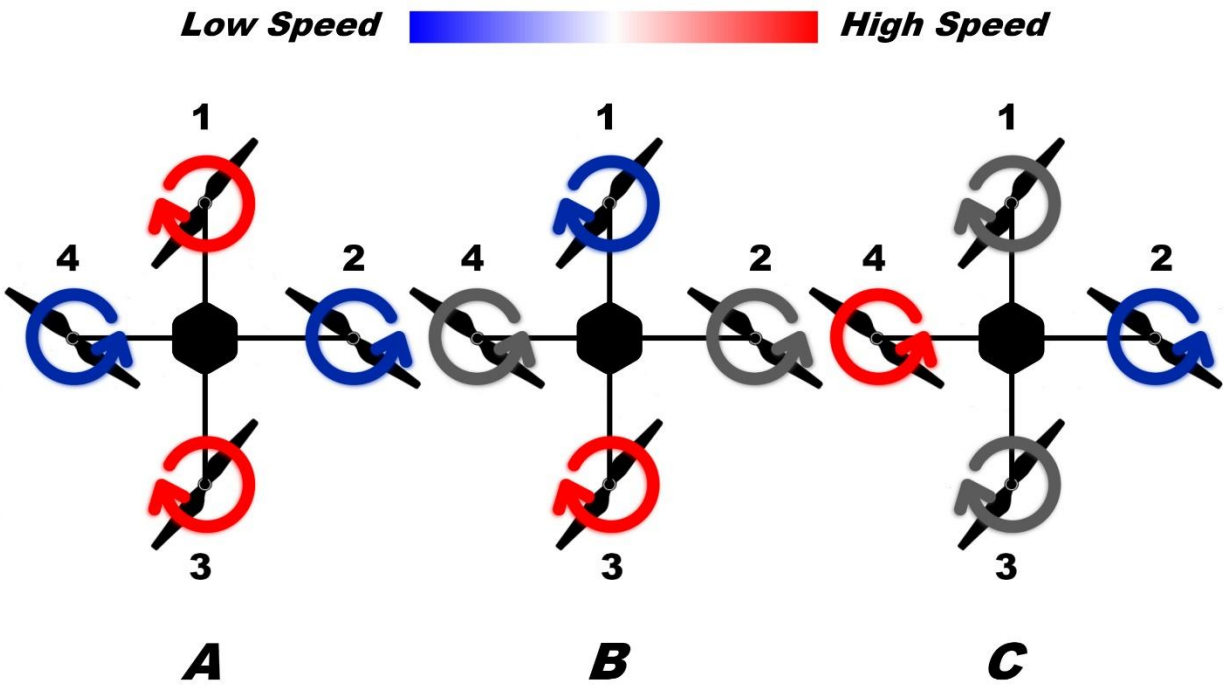

Figure 2.2 – Plus Formation Flight Movements. A: Anticlockwise, B: Forward, C: Right.

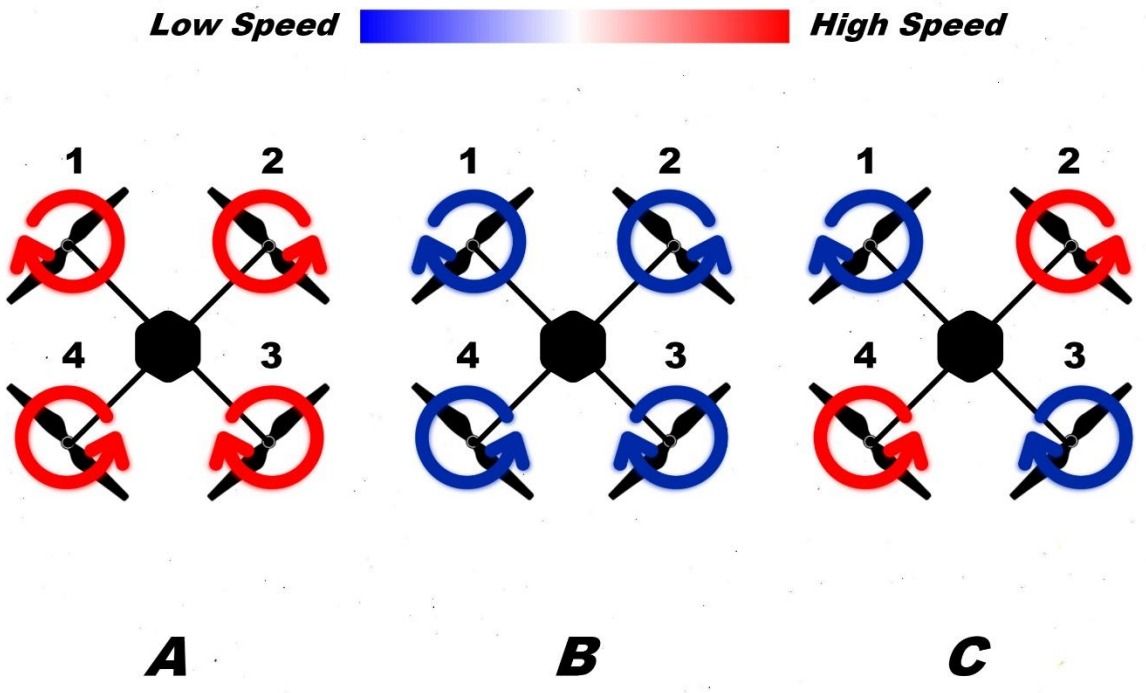

Figure 2.3 – Cross Formation Flight Movements. A: Upward, B: Downward, C: Clockwise.

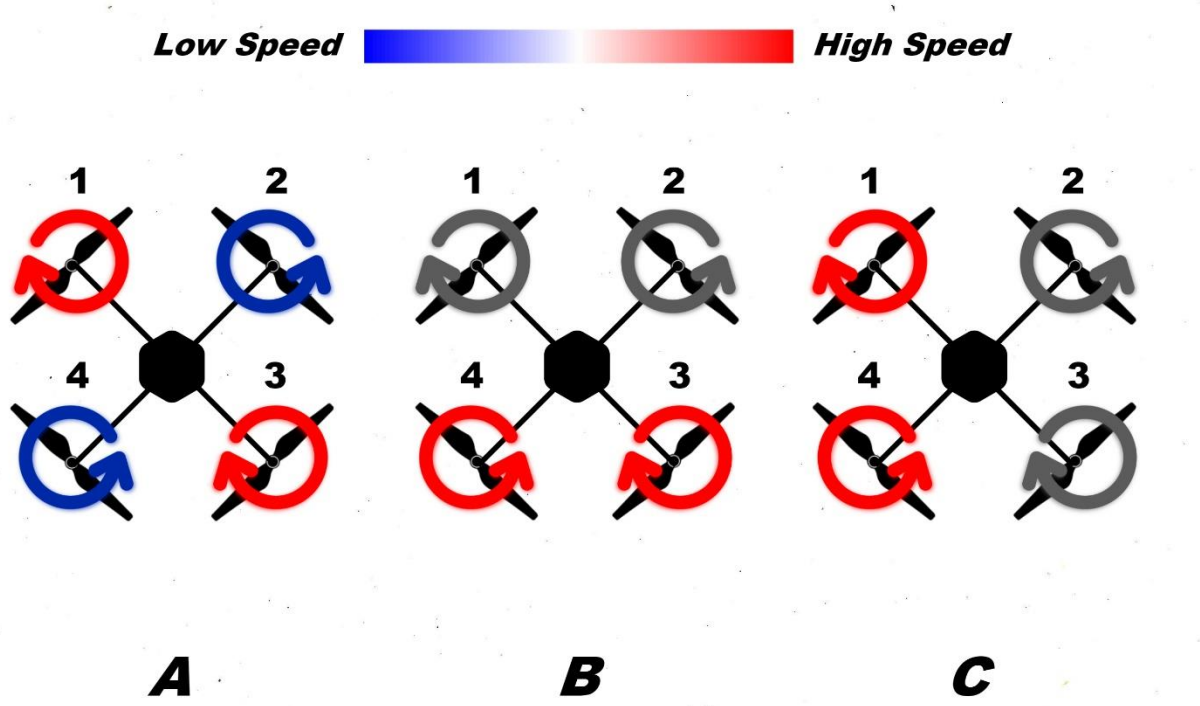

Figure 2.4 – Cross Formation Flight Movements. A: Anticlockwise, B: Forward, C: Right.

As it can be seen from these figures that a quadcopter's motors rotation direction is important in order to achieve an uplift, for example, in both cross and plus formation each parallel motors rotate in the same direction while each adjacent motors rotate in the opposite direction. A controller can control the movement of the drone by adjusting the rotation speed, hence by adjusting the thrust of each motor to achieve the user's desired movement. Despite the fact that quadcopters are simpler in mechanical design than most hovercrafts; however, they suffer from large instability due to the lack of unequal power distribution to each motor, asymmetry in distributing the total weight around the device, and the fact that drones in general are none linear devices. To understand the forces acting upon a quadcopter, a coordinate frame is given to provide a clear 3D model to be used as a basic reference to derive mathematical equations. Figure 2.5 shows the standard quadcopter coordinate system.

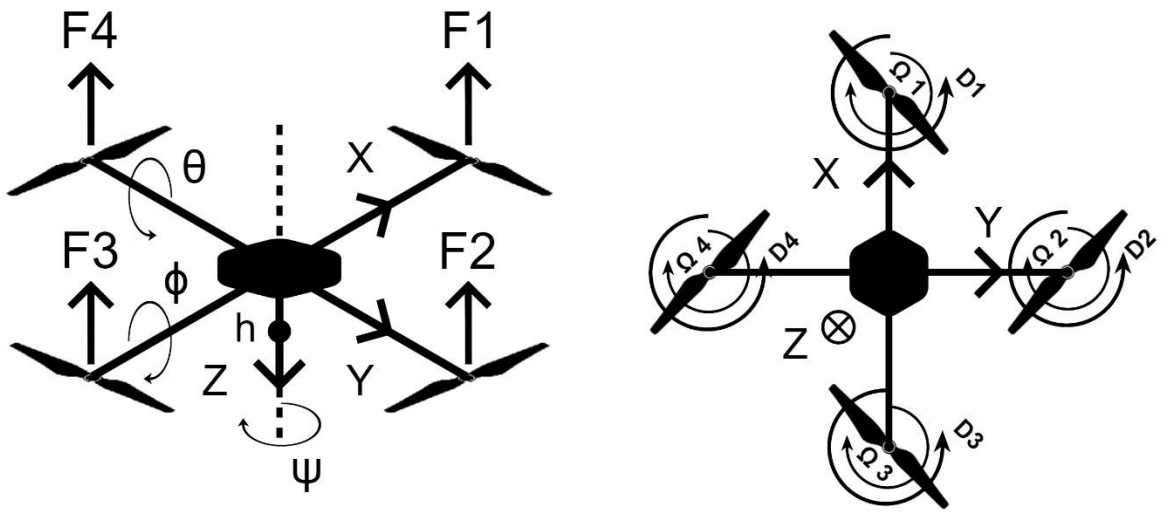

Figure 2.5 – Quadcopter Coordinate System.

Before designing a quadcopter a few points should be taken into account including that the structure of the frame and the propellers should be rigid and light, the structure is supposed to be axis symmetrical, the center of gravity and the body fixed frame origin are assumed to coincide, thrust and drag are proportional to the square root of the propellers speed, and the rotation matrix is defined to transform the coordinates from body to earth coordinates using Euler's angles φ – roll angle , θ – pitch angle, ψ – yaw angle. From Figure 2.5 the thrust forces acting on the quadcopter are F1, F2, F3, and F4, respectively. The quadcopter's angular velocities are (D1, Ω1), (D2, Ω2), and (D3, Ω3) where Ω and D are the clockwise and anticlockwise rotations. For a cross formation quadcopter the angular velocity rotations are Ω1, Ω3, D2, and D3. For simplicity the angular velocities will be denoted as ω. As propellers rotate they create a reaction moment M_i about the z-axis that is proportional to the square of angular velocity. Thrust produced by opposite propellers results in moments M_x and M_y, these moments are given by the difference between forces of opposite propellers multiplied by the quadcopter's arm distance L. Additionally, the force of gravity should be considered which acts in the downward direction of the quadcopter. The forces and moments equations for the quadcopter are represented in the following:

$$F_i = Kf \times {W_i}^2$$

$$M_i = Km \times {W_i}^2$$

$$M_y = (F1 - F3) \times L$$

$$M_x = (F4 - F2) \times L$$

$$Weight = mass * gravity$$

Where

$$Kf = Tangential\ velocity = r * m$$

$$Km = Moment\ Coefficient$$

The motion of quadcopter can be analyzed by Newton second law of motion. For linear motion:

$$Force = mass * linear\ acceleration$$

For rotational motion:

$$Torque = inertia * angular\ acceleration$$

For a quadcopter to be stable and hover in the air it must be in equilibrium state. All the forces must be balanced and the total thrust produced by F1, F2, F3, and F4 must be equal to the weight of the quadcopter, thus all the moments are equal to zero. The equation that governs the condition of quadcopter suspension is given as follows:

$$mass * gravity = F1 + F2 + F3 + F4$$

$$All\ Moments = 0$$

The equation of motion denoted as $m\ddot{r}$ for hovering condition is:

$$m\ddot{r} = F1 + F2 + F3 + F4 - (mass * gravity)$$

$$m\ddot{r} = 0$$

In case of upward motion, the net thrust produced by all propellers must be greater than the weight of the quadcopter.

$$mass * gravity < F1 + F2 + F3 + F4$$

$$All\ Moments = 0$$

$$m\ddot{r} = F1 + F2 + F3 + F4 - (mass * gravity)$$

$$m\ddot{r} > 0$$

In case of downward motion, the net thrust produced by all propellers must be less than the weight of the quadcopter.

$$mass * gravity > F1 + F2 + F3 + F4$$

$$All\ Moments = 0$$

$$m\ddot{r} = F1 + F2 + F3 + F4 - (mass * gravity)$$

$$m\ddot{r} < 0$$

Pitch and roll motion are rotation of quadcopter about horizontal axis. In this condition the opposite pair of propellers produce unequal force causes net none zero moment. For linear motion on the horizontal plane the pitch and roll angles must be none zero. This causes none zero component of thrust in the horizontal direction, these forces in turn causes result in moment in the horizontal plain. For roll motion the following equations are given:

$$mass * gravity < F1 + F2 + F3 + F4$$

$$All\ Moments \neq 0$$

$$m\ddot{r} = F1 + F2 + F3 + F4 - (mass * gravity)$$

$$I_{xx} \times \ddot{\varphi} = (F1 - F3) \times L$$

Where

$$I_{xx} = Moment\ of\ inertia\ about\ the\ x\ axis$$

The yaw motion is the rotation of quadcopter in the horizontal plane. If the reaction moment produced by one pair of propellers is more than the moment produced by other pair of propellers then there is net result in moment produced which causes a yaw rotational motion; otherwise, no yaw motion is produced. For a quadcopter to yaw while hovering the following conditions should be achieved:

$$mass * gravity = F1 + F2 + F3 + F4$$

$$All\ Moments \neq 0$$

$$m\ddot{r} = F1 + F2 + F3 + F4 - (mass * gravity)$$

$$I_{zz}\ddot{\psi} = M1 + M2 + M3 + M4$$

2.2 Controller

Many different types of controllers can be used to achieve the required stability process with the drone itself. Begin by understanding the main approaches when designing a controller. Many paths leads to a good controller and this choice usually depends on what kind of data is available. The simplest case is when the plant of the processes already has its mathematical model available so one can directly design the controller based on that plant. In most cases this model is not available though. In simple process with simple plant it is normally easy to make trial and extract the necessary parameters or estimation model so the controller design can be done with that estimated model; however, with other plants it may not be as easy and straight forward to realize a trial which can hinder the model estimation. In the case of this project, it is not simple to realize an open loop trials since it can damage the structure with collisions. Therefore, one must prepare specific environment test and test strategies to avoid any damage or problem to the system or the individuals themselves. The use of simulation programs is often recommended, especially in those cases where trials may become expensive or too complex. The most used structure with a generic controller is depicted in Figure 2.6. The main idea is to read the outputs of the plant through the sensors and use them as the feedback signal. The difference between this feedback signal and a given reference signal (usually called error signal) is used to feed the controller which will generate the next input to the plant. Usually one tries to reduce the error signal as much as possible at any given moment. Thus, one must determine which parameters of highest priority are to be optimized in the controller. Choices may vary from a fast response controller, prioritizing minimum steady state error, stability of the closed-loop system, and maximum overshoot.

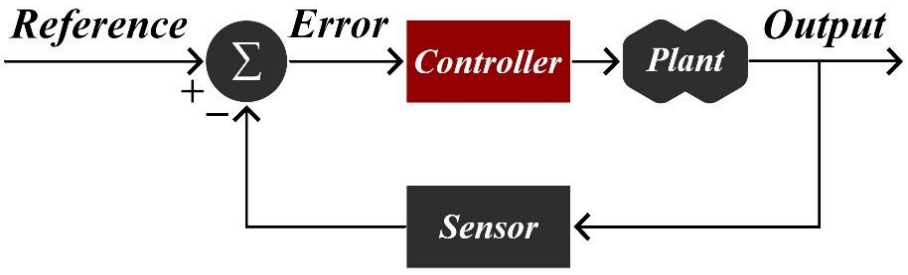

Figure 2.6 – Common Closed-Loop Configuration.

2.2.1 PID Controller

Proportional-integral-derivative controller is a control loop feedback mechanism widely used in industrial control systems. A PID controller calculates an error value as the difference between a measured process variable and a desired set-point. The controller attempts to minimize the error by adjusting the process through use of a manipulated variable. The PID controller algorithm involves three separate constant parameters and is accordingly: the proportional, the integral and the derivative values, denoted by P, I, D [4]. Figure 2.7 shows the block diagram of a PID controller.

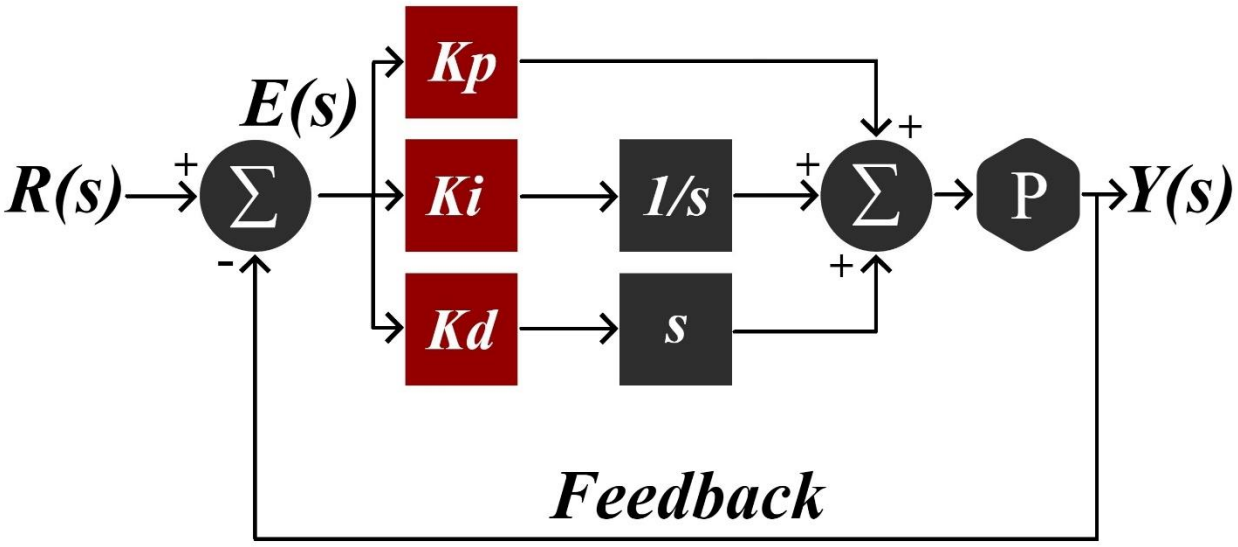

Figure 2.7 – PID Controller Block Diagram.

The PID controller is widely used due to its simplicity along with high efficiency in many cases. The intuitive idea behind each of the three components (P, I, and D) is rather simple to understand. From Figure 2.7, the proportional (P) block is represented as a constant 'Kp', the integral (I) block is a series combination of constant 'Ki' and an integration block, and the derivative (D) block is similarly a series combination of constant 'Kd' and a derivative block. The proportional (P) controller generates a control signal which is proportional to the present error. That is, as the error arises, higher control signals will be produced by the P block to try to minimize this error. The integral (I) term will generate a signal which is proportional to the sum

of past error values. This block is very useful in cases where a single P controller cannot reach zero error in steady state. When that occurs, a constant error will remain in the input of the controller and a simple I block can integrate that constant producing an increasing control signal to lead the error to zero. Finally, the derivative (D) controller will generate a control signal which is proportional to the rate of change of the error signal. Thus, it has a predictive effect. Usually this term contains a built-in low-pass filter to avoid reacting to noisy data. This block has no effect to steady state. In many cases a simple P controller is enough to run a stable system. Although, sometimes it would be better to use a complete PID structure when considering minimum overshoot, rising time, and accurate steady state results such as the case when designing a quadcopter.

SISO Approach

A single-input and single-output (SISO) system is a simple single variable control system with one input and one output. As the linear model of the quadcopter shows, it is possible to use SISO approach for controlling attitude components. SISO systems are typically less complex compared to MIMO systems [5]. Hence a SISO approach is advised for designing a PID controller for the system.

Control parameters of PID controller are usually tuned so that the closed- loop system meets the following three objectives:

1 – Stability and stability robustness, usually measured in frequency domain.

2 – Transient response, including rise time, overshoot, and settling time.

3 – Steady state accuracy.

A widely used strategy to control a quadcopter with a PID controller is to create an individual closed-loop system for each independent axis of each sensor. That is, the same structure depicted in Figure 2.7 for each sensor axis where Y(s) would be the sensor output (angular velocity or acceleration of each axis) and R(s) is the desired set point or reference point value to that axis. Using this method, three different PID controllers for the gyroscope (one for each axis) and the input to each control loop would be an angular velocity reference. That is, making all references equal to zero would make the quadcopter preserve all its rotation angles. Other three distinct

PIDs would then be used to control the accelerometer axes. Therefore, setting zero to X and Y axes would make the quadcopter hover (all weight force would be concentrated in Z axis).
The method that will be used here will implement **sensor fusion** technique which will rely on mixing both the accelerometer and gyroscope data signals to find a single angle value for each axis. That is, combining the gyroscope information with the accelerometer information to find a unique angle value for each axis that can represent an absolute angle of rotation or variation with respect to the previous time step. Therefore, a total of only three closed loop systems will be considered instead of six.

From Figure 2.7 a control block diagram can be used for each one of φ, θ, ψ components. As shown in the figure, one controller should be designed for each one of φ, θ, ψ. The P symbol on the hexagon block represents the plant/process. Quadcopter is the plant for which the controller has to be designed. Here it is the linearized state space model. From the cascaded blocks that represents the PID block system a generalized transfer function of the PID controller is given by,

$$K_p + \frac{K_i}{s} + K_d s$$

$$K_p + \frac{K_i}{s} + K_d s = \frac{K_d s^2 + K_p s + K_i}{s}$$

Where

$K_p = proportional\ gain$

$K_i = integral\ gain$

$K_d = derivative\ gain$

To have any kind of control over the quadcopter or multicopters, it is necessary to be able to measure the quadcopter sensor output (for example the roll angle), so as to estimate the error

(how far we are from the desired roll angle, e.g. horizontal, 0 degree). We can then apply the 3 control algorithms to the error, to get the next outputs for the motors aiming to correct the error. First, we begin to examine how the PID controller works in a closed- loop system using the schematic shown in Figure 2.7. The variable (E) represent the tracking error, the difference between the desired input value (R) and the actual output (Y). This error signal (E) will be sent to the PID controller, and the controller computes both the derivative and the integral of this error signal. The output signal from the PID controller is a combination of the proportional gain (K_p) times the magnitude of the error plus the integral gain (K_i) times the integral of the error plus the derivative gain (K_d) times the derivative of the error.

$$u = K_p e + K_i \int e\,dt + K_d \frac{de}{dt}$$

This signal (u) will be sent to the plant, and the new output (Y) will be obtained. This new output (Y) will be sent back to the sensor again to find the new error signal (E). The controller takes this new error signal and compute the proportional, derivative and integral again. This process goes on and on indefinitely.

The characteristics of P, I and D controllers:

To understand how each parameter of the PID controller effect the stability response of the quadcopter system, a detailed explanation of how each of these 3 PID coefficient (K_p, K_i, K_d) can vary and alter the effectiveness of stabilization should be provided. As mentioned before, there are 3 PID loops with their own "PID" coefficients, one per axis, so you will have to set P, I and D values for each axis (Roll φ, Pitch θ, Yaw ψ).

A proportional controller (K_p) will have the effect of reducing the rise time and will reduce but never eliminate the steady-state error. An integral controller (K_i) will have the effect of eliminating the steady-state error, but it may make the transient response worse. A derivative controller (K_d) will have the effect of increasing the stability of the system, reducing the

overshoot, and improving the transient response. Effects of each of controller on a closed-loop system for the Proportional Integral and Derivative gain are summarized in the Table 1.
The performance chart for the K_p, K_i, K_d are taken from the [4] reference mentioned. Note that these correlations may not be exactly the same, because they are dependent on each other. In fact, changing one of these variables can change the effect of the other two. [6]

Controller	*Rise time*	*Overshoot*	*Settling time*	*Steady-state error*	*Stability*
$\mathbf{K_p}$	Decreases	Increases	Small Change	Decreases	Degrade
$\mathbf{K_i}$	Decreases	Increases	Increases	Eliminate	Degrade
$\mathbf{K_d}$	Small Change	Decreases	Decreases	No Effect in Theory	Improve if K_d is Small

Table 1 – Response of Different Parameters of Proportional Integral and Derivative Gains.

Consider an unstable system with its associated unit step response shown in Figures 2.8 and 2.9:

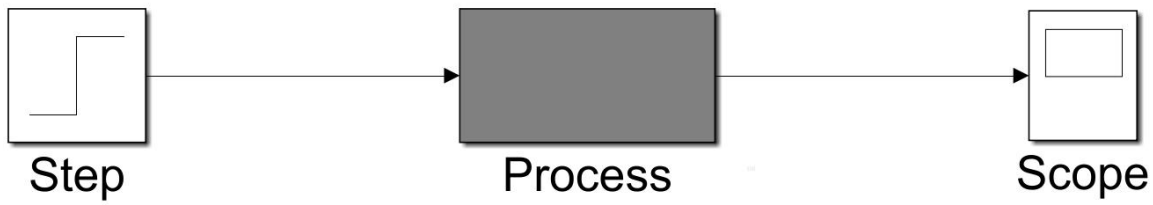

Figure 2.8 – Unstable System Open-Loop Block Diagram.

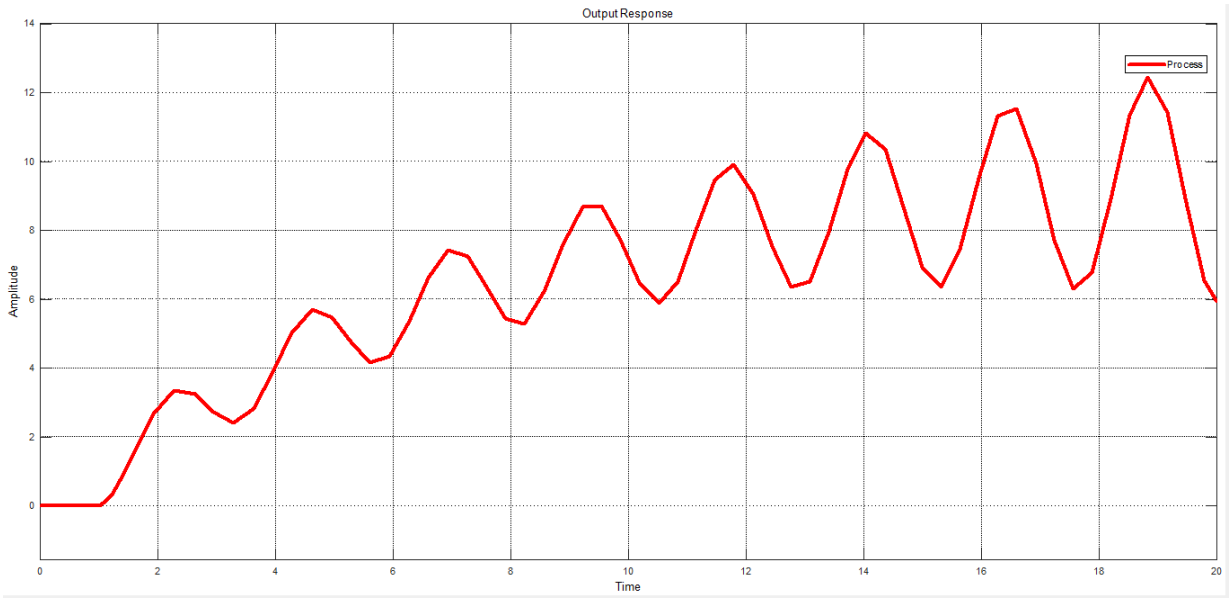

Figure 2.9 – Unstable System Open-Loop Output Response.

Using a unity gain negative feedback loop, the block diagram of system and the output response are shown in Figures 2.10 and 2.11.

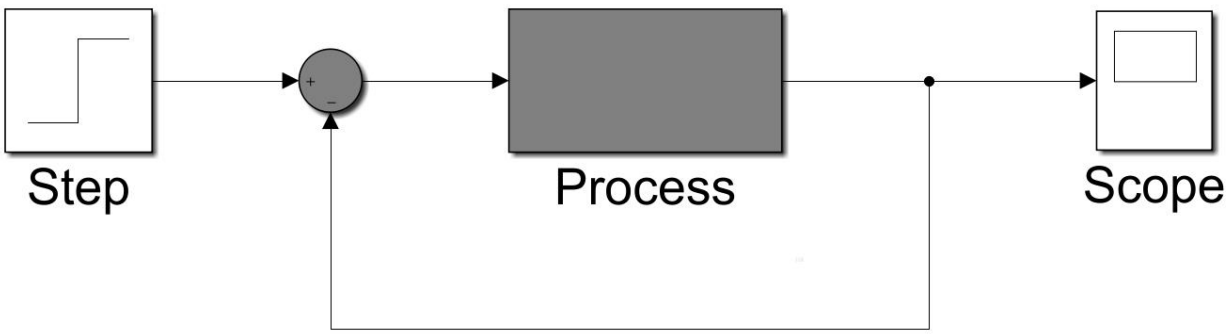

Figure 2.10 – Block Diagram of a System with Unity Gain Feedback.

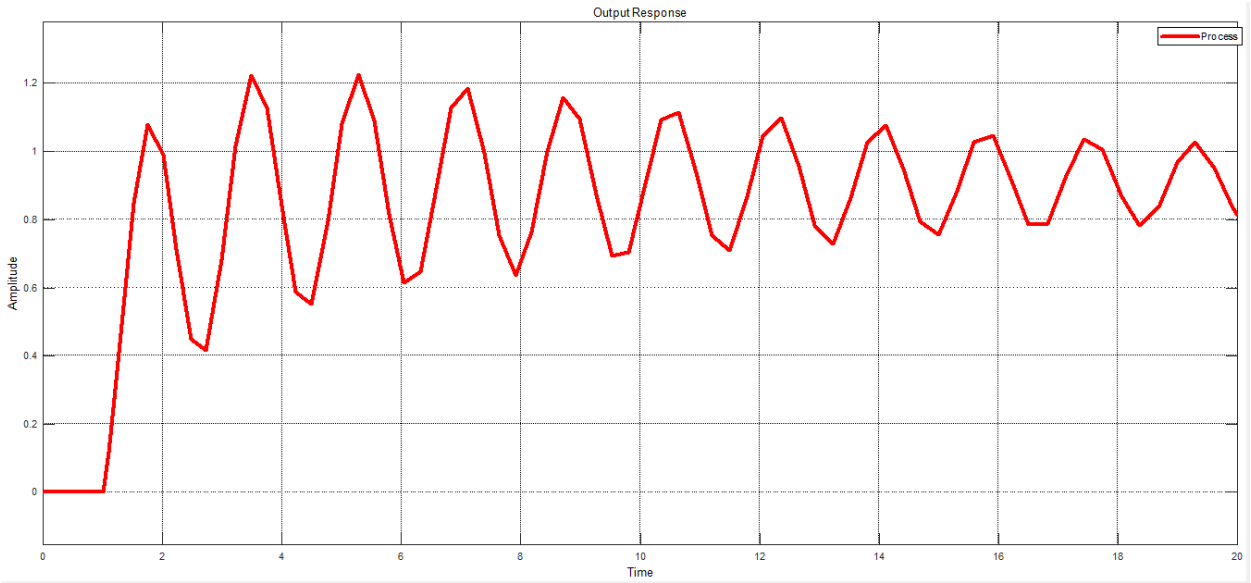

Figure 2.11 – Output Response of a System with Unity Gain Feedback.

A PID controller is introduced into the process to obtain a system with precision results. These effects can be seen from Figures 2.12 and 2.13 which shows a system block diagram with the PID controller along with the output response of the system.

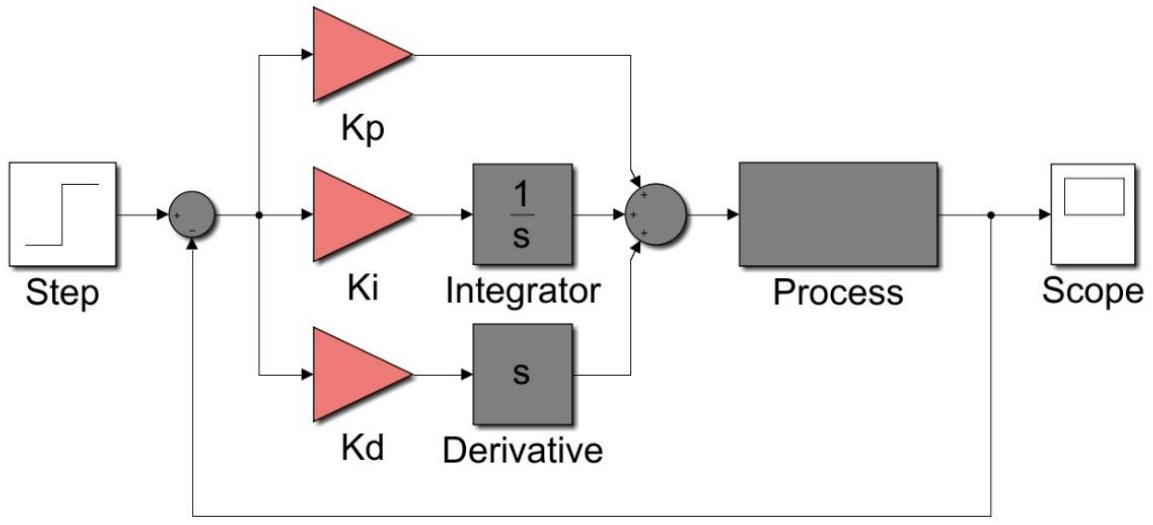

Figure 2.12 – Block Diagram of a System with PID Controller.

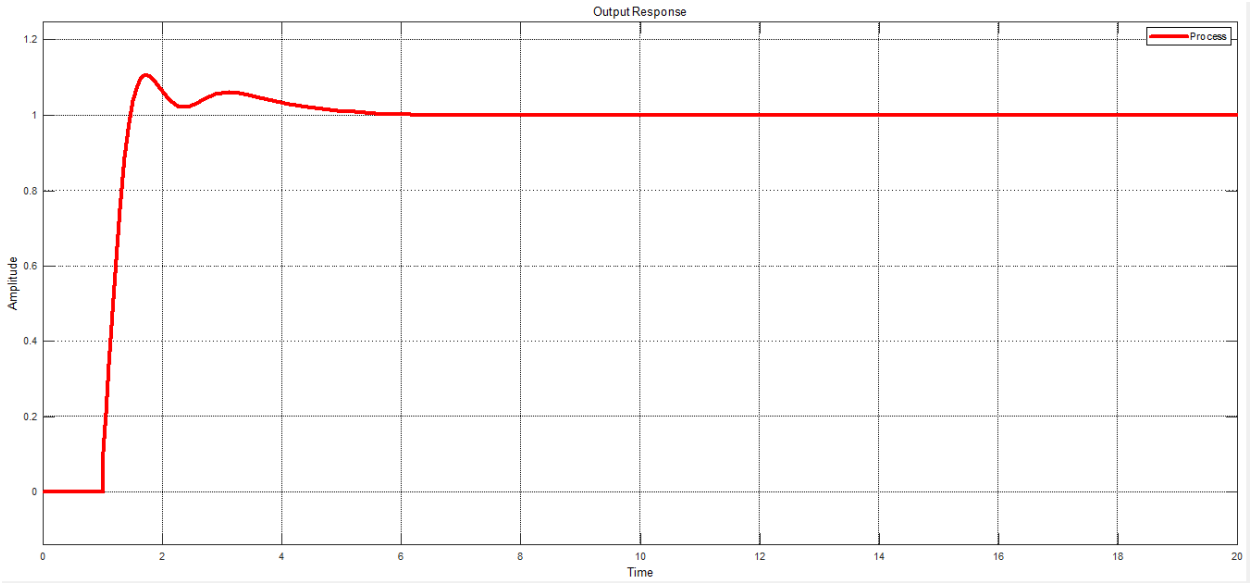

Figure 2.13 – Output Response of a System with PID Controller.

The proportional block of the PID controller will drive the system into stability; however, the overshoot problem will not be eliminated. The derivative block is used to minimize the overshoot but increase the steady-state error. The integral block of the PID will guarantee a zero steady-state error; however, with an increase in the integral gain, the system begins to oscillate with low frequency. It should be noted that when applying an integral gain into the system the overshoot increase. The gains of the PID may be tuned further to obtain a desire output response depending on the application. It can be seen that a PID controller helps deliver a precise, sturdy, and responsive steady-state system. By giving a set point, the controller tracks the desired value to reach an almost zero error output, all depending on the PID coefficients applied and tuned towards the optimal results. Tuning the PID coefficients depends on the availability of a transfer function; however, many applications may require trial-and-error where transfer function of a given system is not available or can be difficult and too complex to realize. There are several methods for tuning a PID loop. The most effective methods generally involve the development of some form of process model, then choosing P, I, and D based on the dynamic model parameters. Manual tuning methods can be relatively time consuming, particularly for systems with long loop times.

Table 2 summarizes the different methods used in order to tune the PID controller. The PID tuning methods are taken from the [4] reference mentioned.

Method	*Advantages*	*Disadvantages*
Manual Tuning	No math required; online.	Requires experienced personnel
Ziegler-Nichols	Proven method; online.	Process upset, some trial-and-error, very aggressive tuning.
Tyreus Luyben	Proven method; online.	Process upset, some trial-and-error, very aggressive tuning.
Software Tools	Consistent tuning; online or offline – can employ computer-automated control system design (CAutoD) techniques; may include valve and sensor analysis; allows simulation before downloading; can support non-steady-state (NSS) tuning.	Some cost or training involved.
Cohen-Coon	Good process models	Some math; offline; only good for first-order processes.
Åström-Hägglund	Can be used for auto tuning; amplitude is minimum so this method has lowest process upset.	The process itself is inherently oscillatory.

Table 2 – Different PID Tuning Methods.

The choice of method will depend largely on whether or not the loop can be taken "offline" for tuning, availability of transfer function, and on the response time of the system. If the system can be taken offline, the best tuning method often involves subjecting the system to a step change in input, measuring the output as a function of time, and using this response to determine the control parameters. Figure 2.14 shows the PID controllers for each roll, pitch, and yaw axis.

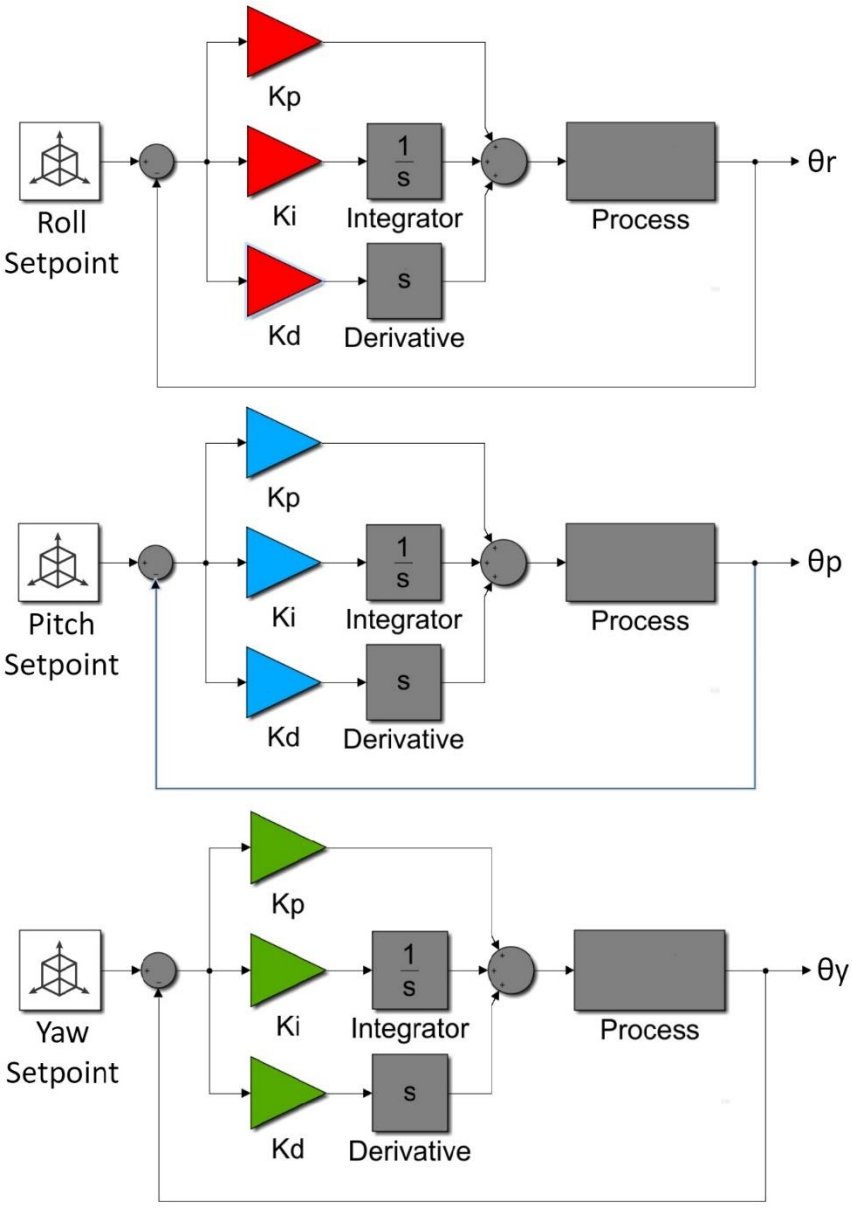

Figure 2.14 – Roll, Pitch, and Yaw PID Controller Angles.

2.3 Power Conditioning

In order to power up the quadcopter's electronic subsystems (Microcontroller, Sensors, Bluetooth, and GPS) a power conditioning circuit should be considered to provide constant, efficient power to all electronic circuitry within the quadcopter. The quadcopter subsystems are explained in the followings:

a) Microcontroller – The main brain of the quadcopter that computes and handles most of the tasks from stabilizing the drone to decoding GPS and Bluetooth data. The microcontroller that will be used for this project has an already built-in voltage regulator that supplies constant 3.3V to the controller. The voltage regulator takes voltage inputs that ranges from 7V and up to 12V.

b) Bluetooth 4.0 Adapter – In order to provide means of communication between the quadcopter and the smartphone a Bluetooth adapter is used to allow data exchange between the two devices. The Bluetooth chip has an already built-in voltage regulator that supplies constant 3.3V to the chip itself. The voltage regulator takes voltage inputs that ranges from 5V and up to 6V.

c) GPS – The global positioning system (GPS) breakout board provides GPS data in the form of NMEA (National Marine Electronics Association) sentences. This data can be used to provide multiple informations such as the quadcopter's location, the current date and time, acceleration speed, etc. The module has a built-in voltage regulator that supplies constant 3.3V to the chip itself. The voltage regulator takes voltage inputs that ranges from 3.3V and up to 5V.

d) IMU – The inertial measurement unit (IMU) is an electronic device that measures and reports a body's specific force, angular rate, and sometimes the magnetic field surrounding the body, using a combination of accelerometers and gyroscopes. IMU's are used to maneuver aircrafts and allow them to stabilize themselves autonomously. The module has a built-in voltage regulator that supplies constant 3.3V to the chip itself. The voltage regulator takes voltage inputs that ranges from 3.3V and up to 5V.

e) Auxiliary Sensors – The quadcopter is provided with other sensors that measure the ambient data and convert it to voltage levels where the microcontroller converts these voltages to digital format using analog-to-digital (ADC) circuit. The input voltage to the ADC pins of the microcontroller takes voltage ranges from 0V and up to 3.3V. The input

voltage of the ADC depends on the output voltage of the sensors; thus, the sensors' output voltage should be considered and their input voltage depends on their circuit schematics. A simple voltage divider can be used to lower the voltage to ranges where the microcontroller can read. It should be noted that one of the ADC input pins is tasked to reading the quadcopter battery; therefore, the unconditioned battery voltage is used as an input to the voltage divider circuit to read the variations of the battery as means to compensate the motors speed.

Table 3 summarizes the quadcopter's subsystems along with their minimum and maximum input voltages.

Subsystem	*Min Input Voltage*	*Max Input Voltage*	*Target Input Voltage*
Microcontoller	7V	12V	7.5V
Bluetooth	5V	6V	5V
GPS	3.3V	5V	5V
IMU	3V	5V	3.3V
Auxiliary Sensors	3.3V	Depends on Circuit Design	3.3V/5V/12V

Table 3 – Quadcopter's Subsystems and Input Voltage Ratings.

From Table 3 it can be observed that the target input voltages to each subsystem are categorized into 4 sections: 12V, 7.5V, 5V, and 3.3V. One important note to take into consideration is that the microcontroller can provide a constant, regulated 3.3V output. With this information taken into account and the fact that one subsystem requires input voltage directly from the battery then a schematic diagram of the power conditioning circuit can be realized with SIMO (single input, multiple output) block where the input is the battery voltage and the regulated voltage outputs are both 5V and 7.5V respectively. Figure 2.15 shows the general SIMO block diagram for the power conditioning system.

Figure 2.15 – Power Conditioning System SIMO Block Diagram.

Before designing the power conditioning circuit, a few key points are to be considered first:

1 The dimensions of the power circuit should be small and compact.
2 The power conditioning circuit should be lightweight to minimize weight capacity.
3 The power conditioning circuit should provide constant uninterruptable power with high efficiency
4 The circuit board should be rigid and strong to withstand external forces that may break the circuit.
5 The circuit board should be fixed and held in place firmly; otherwise, the circuit may break free during flight and interrupt the power supplied to the system.

There are three approaches to design a circuit that provides constant voltage to the load, they are:

1 Voltage Divider
2 Voltage Regulator
3 Buck Converter

2.3.1 Power Circuit Approaches

Voltage Divider:

The first approach to step down DC to a desired voltage level is by using a voltage divider.
In electronics, a voltage divider (also known as a potential divider) is a passive linear circuit that produces an output voltage (V_{out}) that is a fraction of its input voltage (V_{in}). Voltage division is the result of distributing the input voltage among the components of the divider. A simple example of a voltage divider is two resistors connected in series, with the input voltage applied across the resistor pair and the output voltage emerging from the connection between them. [7]
Figure 2.16 shows a simple voltage divider circuit using two resistors in series.

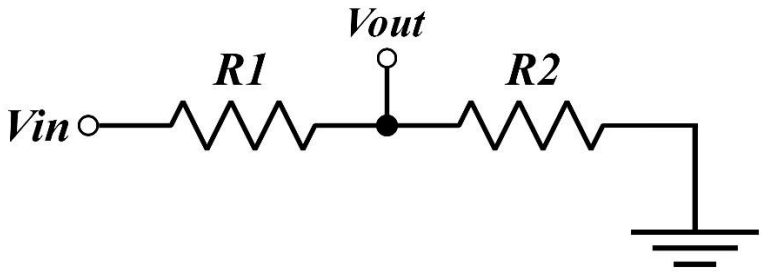

Figure 2.16 – Simple Resistive Voltage Divider.

From figure 2.16 the output voltage (V_{out}) equation is giving as:

$$V_{out} = V_{in} \cdot \frac{R_2}{R_1 + R_2}$$

As can be seen from the above equation, the output voltage depends on the factor ($\frac{R_2}{R_1 + R_2}$).
However, the output voltage of a voltage divider will vary according to the electric current it is supplying to its external electrical load. To obtain a sufficiently stable output voltage, the output current must either be stable (and so be made part of the calculation of the potential divider values) or limited to an appropriately small percentage of the divider's input current. Load sensitivity can be decreased by reducing the impedance of both halves of the divider, though this

increases the divider's quiescent input current and results in higher power consumption (and wasted heat) in the divider. Therefore, a voltage divider suffers from low efficiency performance and the lack of a feedback circuit to provide controllability for the output voltage with variation of the load. Despite the disadvantages of a voltage divider, yet this circuit can be used in applications where it can allow a microcontroller to measure the resistance of a sensor. The microcontroller's ADC converter is connected to the center tap of the divider so that it can measure the tap voltage and, by using the measured voltage and the known resistance and voltage, compute the sensor resistance.

Voltage Regulator:

An alternative solution for voltage divider circuits is by using a voltage regulator. A voltage regulator is a system designed to automatically maintain a constant voltage level. A voltage regulator may use a simple feed-forward design or may include negative feedback. Feedback voltage regulators operate by comparing the actual output voltage to some fixed reference voltage. Any difference is amplified and used to control the regulation element in such a way as to reduce the voltage error. This forms a negative feedback control loop; increasing the open-loop gain tends to increase regulation accuracy but reduce stability. Figure 2.17 shows an example of linear voltage regulator using transistor.

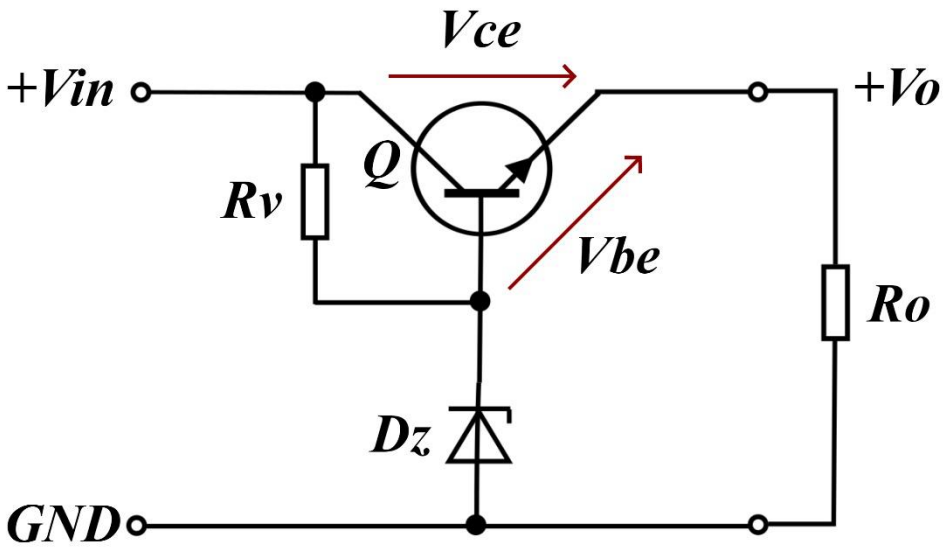

Figure 2.17 – Linear Voltage Regulator Using Transistor (Common Collector Configuration).

A simple transistor regulator will provide a relatively constant output voltage (V_{out}) for changes in the voltage of the power source (V_{in}) and for changes in load (R_o) provided that V_{in} exceeds V_{out} by a sufficient margin, and that the power handling capacity of the transistor is not exceeded. The stability of the output voltage can be significantly increased by using an operational amplifier. Although, the voltage regulator may seem to be a practical solution to be used for providing constant, regulated voltage it suffers from extremely low efficiency due to heat dissipation. Usually a heat sink maybe used in combination with voltage regulators to minimize over-heating of the module. Voltage regulators can be used where efficiency is not the main concern or when there's no other alternative solution available. Unlike other forms of voltage conditioning, voltage regulators are not affected by external electromagnetic waves nor do they produce electromagnetic noises making them the ideal approach when considering a system where electromagnetic interference should be at minimum.

Buck Converter:

Buck converter is a DC-to-DC step-down power converter that combines high efficiency with controllability. It is a class of switched-mode power supply (SMPS) typically containing at least two semiconductors (a diode and a transistor) and at least one energy storage element, a capacitor, inductor, or the two in combination. To reduce voltage ripple, filters made of capacitors (sometimes in combination with inductors) are normally added to such a converter's output (load-side filter) and input (supply-side filter). Switching converters provide much greater power efficiency as DC-to-DC converters than linear regulators by stepping the output current. Buck converters can be highly efficient (often higher than 90%).

Buck converters uses the switching operation to switch the input voltage of the converter from off to on state repeatedly. A simplified circuit of the buck converter is shown in Figure 2.18. Switch S represents an electrical switch that turns on and off at a specific frequency. The diode is used to allow the current to flow in just one direction during the ON/OFF state which is enough to guarantee the proper functioning of the device. The conceptual model of the buck converter is best understood in terms of the relation between current and voltage of the inductor. Beginning with the switch open (off-state), the current in the circuit is zero. When the switch is first closed (on-state), the diode is open because the cathode voltage is higher than the anode and the current will begin to increase, and the inductor will produce an opposing voltage across its terminals in

response to the changing current. This voltage drop counteracts the voltage of the source and therefore reduces the net voltage across the load. Over time, the rate of change of current decreases, and the voltage across the inductor also then decreases, increasing the voltage at the load. During this time, the inductor stores energy in the form of a magnetic field. If the switch is opened while the current is still changing, then there will always be a voltage drop across the inductor, so the net voltage at the load will always be less than the input voltage source. When the switch is opened again (off-state), the voltage source will be removed from the circuit due to the fact that the diode's cathode voltage is lower than the anode which it can be represented as a short circuit ideally, and the current will decrease. The decreasing current will produce a voltage drop across the inductor (opposite to the drop at on-state), and now the inductor becomes a Current Source. The stored energy in the inductor's magnetic field supports the current flow through the load. This current, flowing while the input voltage source is disconnected, when concatenated with the current flowing during on-state, totals to current greater than the average input current (being zero during off-state). The "increase" in average current makes up for the reduction in voltage, and ideally preserves the power provided to the load. During the off-state, the inductor is discharging its stored energy into the rest of the circuit. If the switch is closed again before the inductor fully discharges (on-state), the voltage at the load will always be greater than zero. [8]

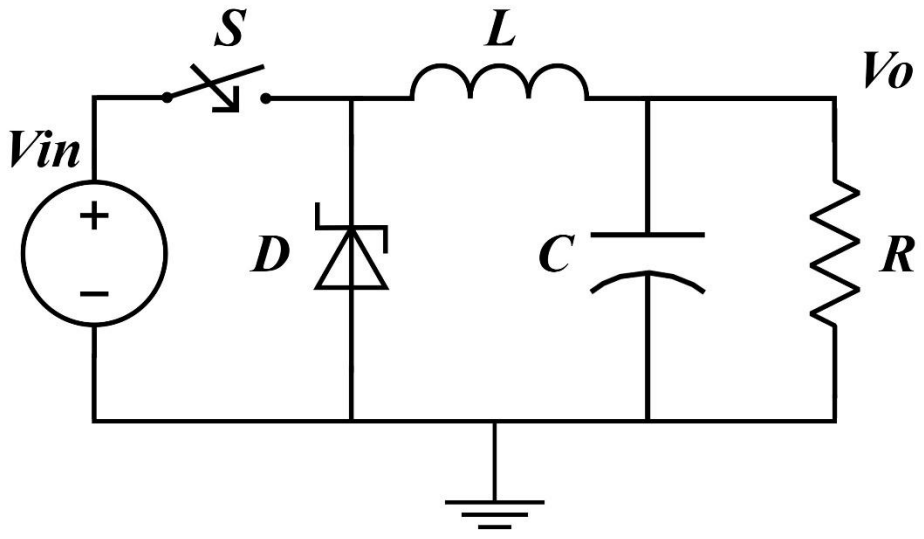

Figure 2.18 – Simple Buck Converter Circuit Diagram

The switch can be substituted for a transistor, usually for high switching speed a MOSFET transistor is used in combination with a schottky diode. The signal in the gate of the MOSFET is a pulse-width modulation (PWM) signal. A PWM signal is a form of digitally modulated binary signal that can be controlled using frequency and duty cycle. The term duty cycle describes the proportion of 'on' time to the regular interval or 'period' of time; a low duty cycle corresponds to low power, because the power is off for most of the time. Duty cycle is expressed in percent, 100% being fully on. When a digital signal is on half of the time and off the other half of the time, the digital signal has a duty cycle of 50% and resembles a "square" wave. Figure 2.19 shows an example of a pulse-width modulation signal shape.

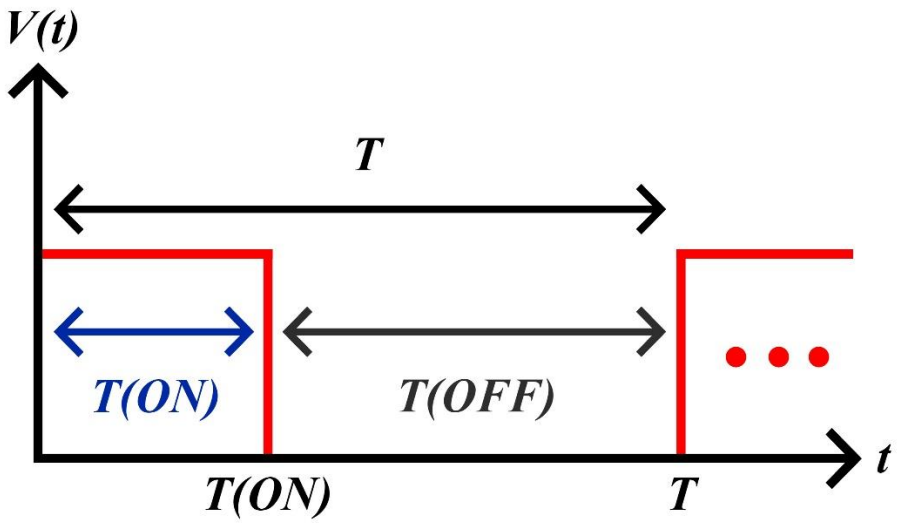

Figure 2.19 – Pulse-Width Modulation Signal Shape

From the above figure, the duty cycle of the signal can be found as:

$$Duty\ (\%) = \frac{T_{(ON)}}{T} \cdot 100\%$$

Where $T_{(ON)}$ is the pulse width (pulse active time) and T is the total period of the signal which is equal to $T_{(ON)} + T_{(OFF)}$. Therefore, a 40% duty cycle implies that the signal is on 40% of the time but off 60% of the time. The PWM signal will be the factor that decides the value of the

output voltage of the buck converter circuit. In order to derive an equation for the output voltage the buck converter circuit will be divided into two stages, ON and OFF stage. When the switch is on (switch is closed), the diode is replaced with an open circuit and the inductor current can be found as follows:

$$V_{in} - V_o = L\,\frac{di_L(t)}{dt} \quad , \quad 0 < t < T_{OFF}$$

$$i_L(t) = \frac{V_{in} - V_o}{L}\,t + C$$

$$\Delta i_L(ON) = \frac{V_{in} - V_o}{L} \cdot (Duty)\,T \quad \rightarrow \quad (1)$$

Similarly when the switch is OFF (switch is open), the diode is replaced with a short circuit and the inductor current can be found as follows:

$$-V_o = L\,\frac{di_L(t)}{dt} \quad , \quad T_{ON} < t < T$$

$$i_L(t) = \frac{-V_o}{L}\,t + C$$

$$\Delta i_L(OFF) = \frac{-V_o}{L} \cdot (1 - Duty)\,T \quad \rightarrow \quad (2)$$

Using superposition, add equation 1 with 2 and equate them to zero which yields the following:

$$\Delta i_L(ON) + \Delta i_L(OFF) = Zero$$

$$\frac{V_{in} - V_o}{L} \cdot (Duty)\,T + \frac{-V_o}{L} \cdot (1 - Duty)\,T = Zero$$

$$\therefore V_o = Duty_{normalized} \cdot V_{in}$$

Therefore, the output voltage is the input voltage multiplied by the duty cycle. The duty cycle of the PWM can have values between 0 and 1 (normalized). The only possible output will be equal or lower than the input. Thus, this configuration is called step down converter. The inductance value for the converter can be found using the following equation:

$$L = \frac{(V_{in} - V_{sw} - V_o) \cdot (V_D + V_o)}{(V_{in} - V_{sw} + V_D) \cdot f_{sw} \cdot \gamma \cdot I_{out}}$$

Where V_{sw} is the MOSFET on-state voltage drop, f_{sw} is the switching frequency, I_{out} is the average inductor current value, and γ is the ripple ration given as:

$$\gamma = \frac{\Delta I_L}{I_{out}}$$

Figure 2.20 shows a buck converter circuit with a PWM signal on the gate of the MOSFET transistor.

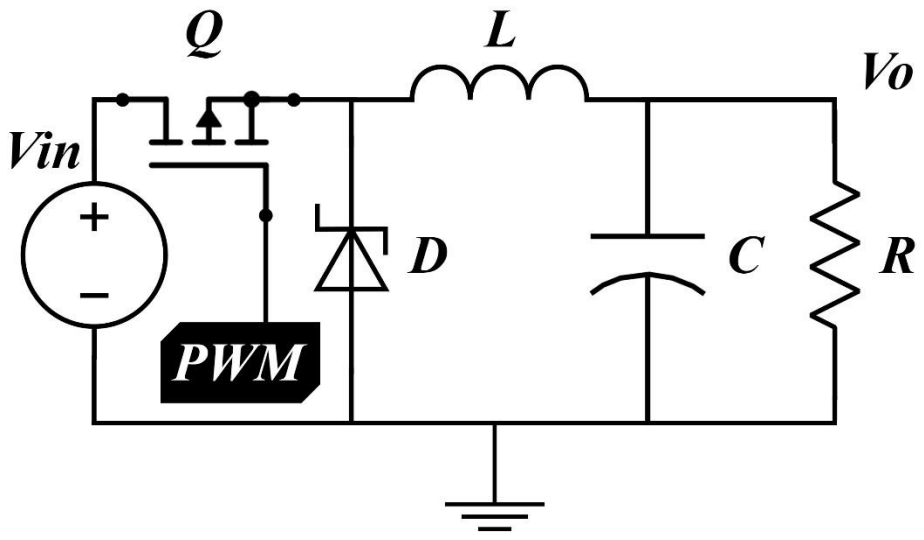

Figure 2.20 – Buck Converter Using PWM Gate Signal

The above buck converter suffers from the lack of a feedback control, and the output voltage will change if the output load changes. In order to provide a feedback a series resistance combination is added in parallel at the output side. This series combination represents a voltage divider circuit where the center tap of the divider is fed to a microcontroller allowing it to vary the PWM signal applied to the gate of the transistor. With this configuration, the efficiency and controllability of the buck converters gives them huge advantage over the counter parts. It should be noted that buck converters have a disadvantage of producing electromagnetic noise from the inductor as mentioned earlier; therefore, using them in applications where noise interference should be minimum is not preferable and as a replacement the voltage regulators are considered to perform better since they do not produce any noise interference. Figure 2.21 shows a buck converter circuit with a feedback loop taken from the output of the resistance divider.

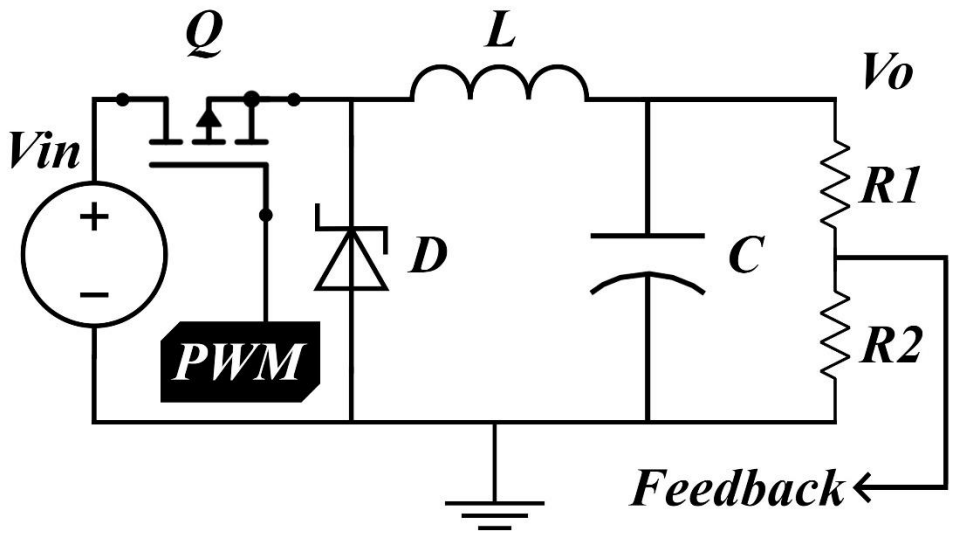

Figure 2.21 – Buck Converter with Feedback Loop

2.3.2 Power Circuit Selection

In order to design a power conditioning circuit, a SIMO approach has been used and, using the three design approaches, the solution is to use either a buck converter circuit or a voltage divider circuit. The target output voltages of the SIMO power conditioning block are 5V and 7.5V respectively. Using a buck converter for both output is reasonable and more preferable since the approach is to minimize power loss in our system. Generally, a monolithic integrated circuits that

provide all the active functions for a step-down (buck) conversion using control topology can be used as a replacement for the feedback microcontroller. These integrated circuits normally are available for stepping down voltage to 3.3V, 5V, 12V, 15V, and an adjustable output version; however, a standard value of 7.5V is not available in the market; similarly, an adjustable version was not found either, and since the goal is to minimize space and design our own buck converters from the basic levels, using a predesigned adjustable buck converters has been excluded. To output a 7.5V an adjustable voltage regulator has been used instead. The power loss can be tolerated in this design by using multiple software techniques to assure that other subsystems only function when needed. A 5V output voltage can therefore be designed using 5V buck converter controller. Both **LM2576T** and **LM317** IC chips by **Texas Instruments** will be used in this project. A functional block diagram of the **LM2576T** buck converter controller chip can be seen in Figure 2.22.

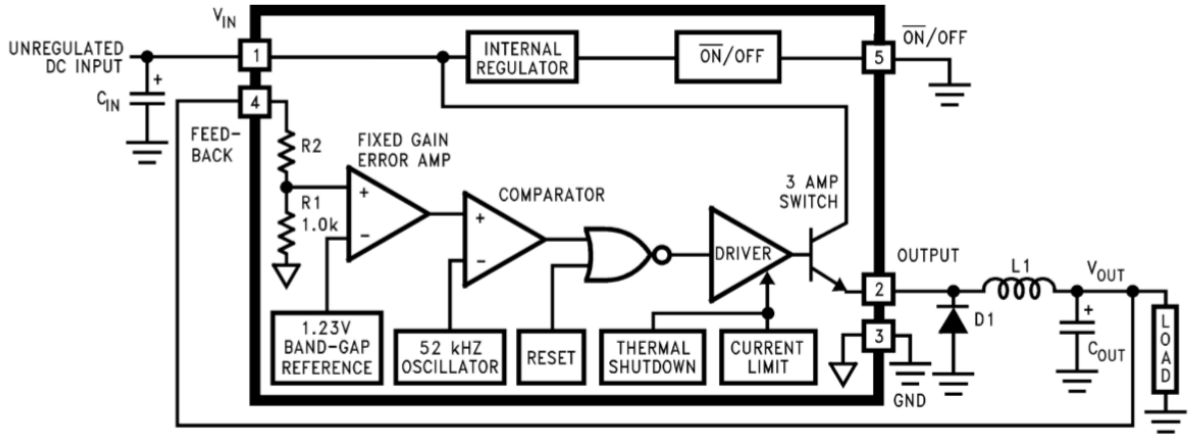

Figure 2.22 – LM2576T Functional Block Diagram (Copyright © 2016 Texas Instruments)

Via using the datasheet for a 5V LM2576T chip, values for C_{in}, C_{out}, and L_1 are 100uF, 1000uF, and 100uH respectively. On the other hand, a simplified block diagram of LM317 adjustable voltage regulator can be seen in Figure 2.23:

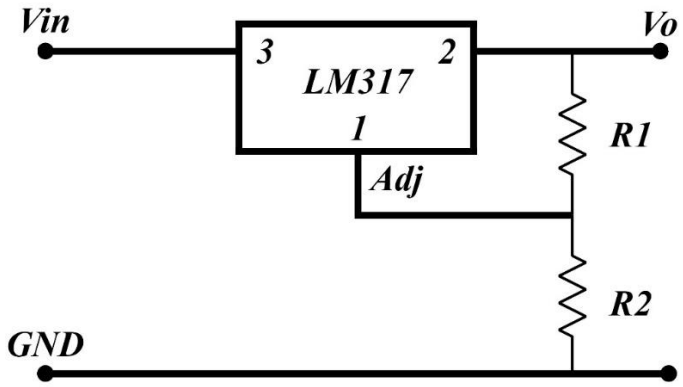

Figure 2.23 – LM317 Circuit Block Diagram

The output voltage can be found using the following equation:

$$Vo = 1.25 \cdot (1 + \frac{R_2}{R_1})$$

To obtain an output voltage of 7.5V, the factor $(\frac{R_2}{R_1})$ should be equal to 5. A standard resistance values for R_2 and R_1 are chosen to be 1.2kΩ and 240Ω respectively.

Now that a standard circuits have been selected, the complete power conditioning block is represented in Figure 2.24

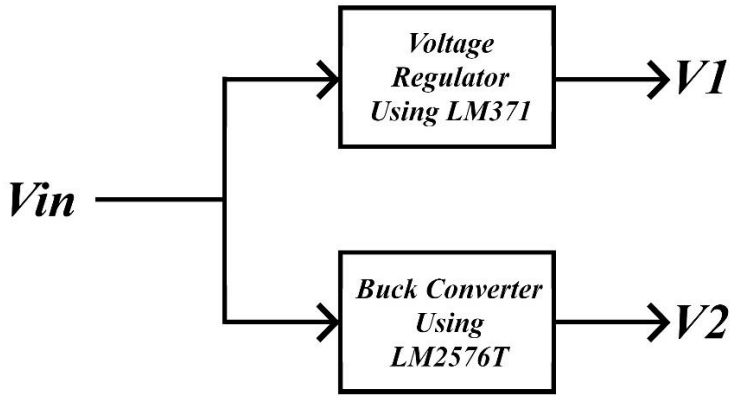

Figure 2.24 – Power Conditioning Block Diagram Using Buck Converter & Voltage Regulator

Figure 2.25 shows the testing stages for the power conditioning circuit. Figure 2.26 shows the PCB design of the power conditioning circuit.

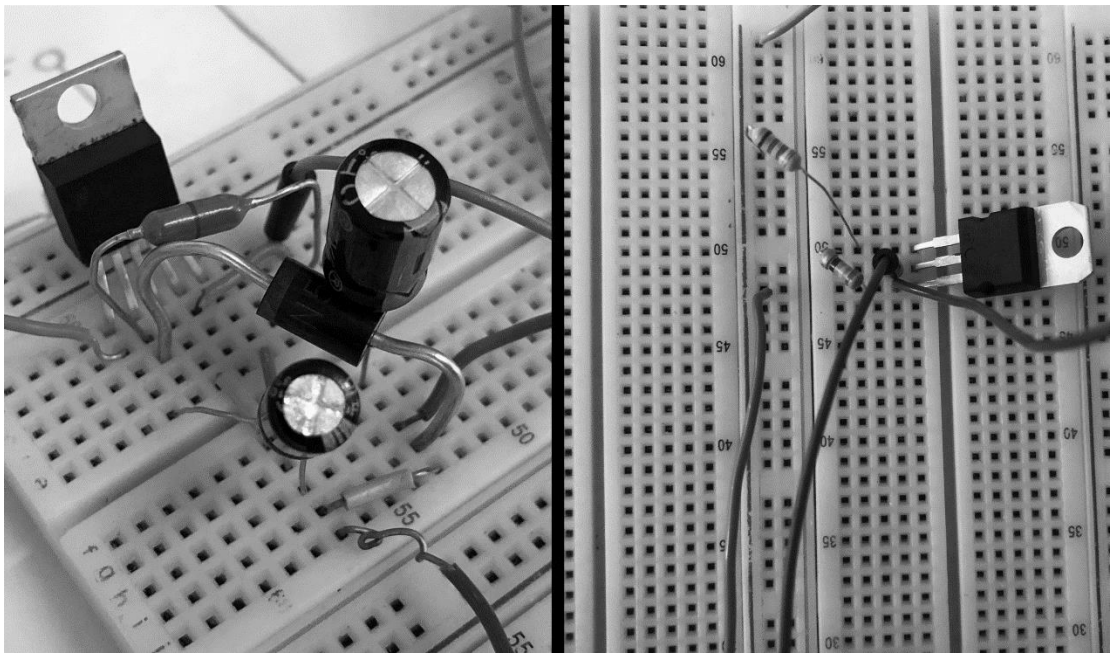

Figure 2.25 – Power Conditioning Circuit Testing Phase

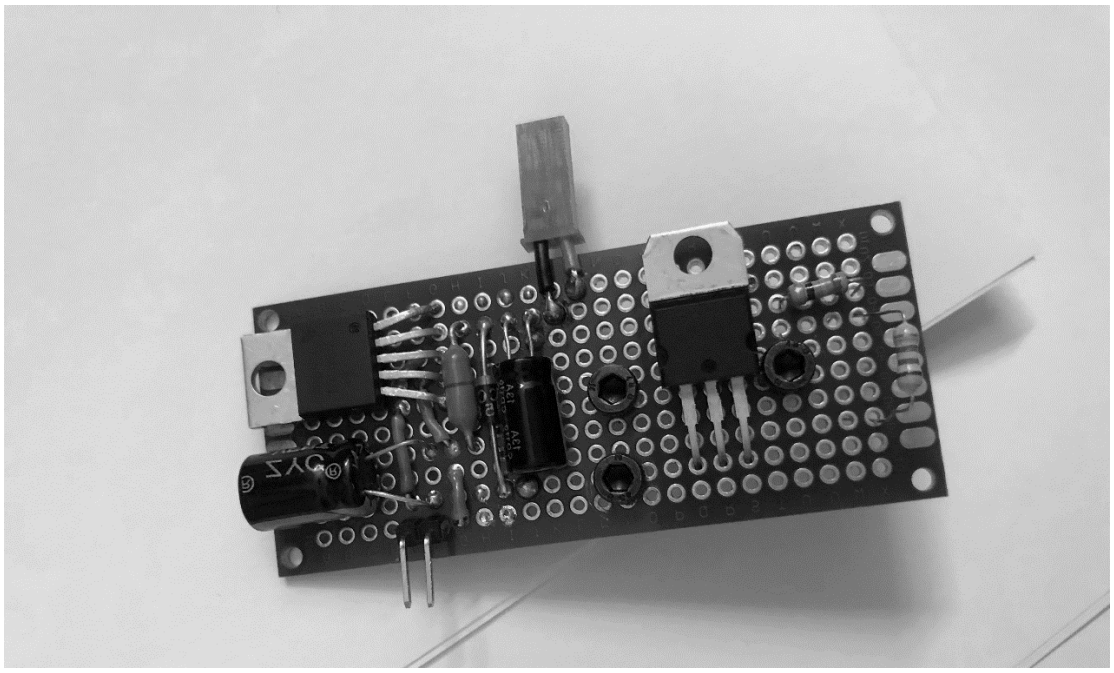

Figure 2.26 – Power Conditioning Circuit PCB Design

2.4 Main Board Design

To obtain flight stability and process multiple incoming data a microcontroller is used to handle multiple tasks at once. The microcontroller, as explained in **Section 2.3**, is considered a subsystem of a complete system namely the quadcopter, other subsystems including the Bluetooth adapter, GPS, IMU, sensors, power, and other auxiliary modules should be able to communicate with the microcontroller and other electronic components by means of circuit connections. To provide connections between each electronic subsystem a printed circuit board (PCB) is used to mechanically hold and electrically connects each electronic component using conductive tracks. A basic PCB consists of a flat sheet of insulating material and a layer of copper foil, laminated to the substrate. To design the main board of the quadcopter a single-sided PCB with a 33 by 33 matrix of mounting holes is used. The choice of this PCB is selected due to the fact that the project is considered as a prototype design and thus a prototype PCB is used in the first place. To mount each component into the circuit board a soldering material is used to allow a reasonably permanent but reversible connections between the electronic modules and the board itself. Figure 2.27 shows the construction of single-sided PCB board. Figure 2.28 shows the method of soldering discrete electronic components onto a PCB sheet.

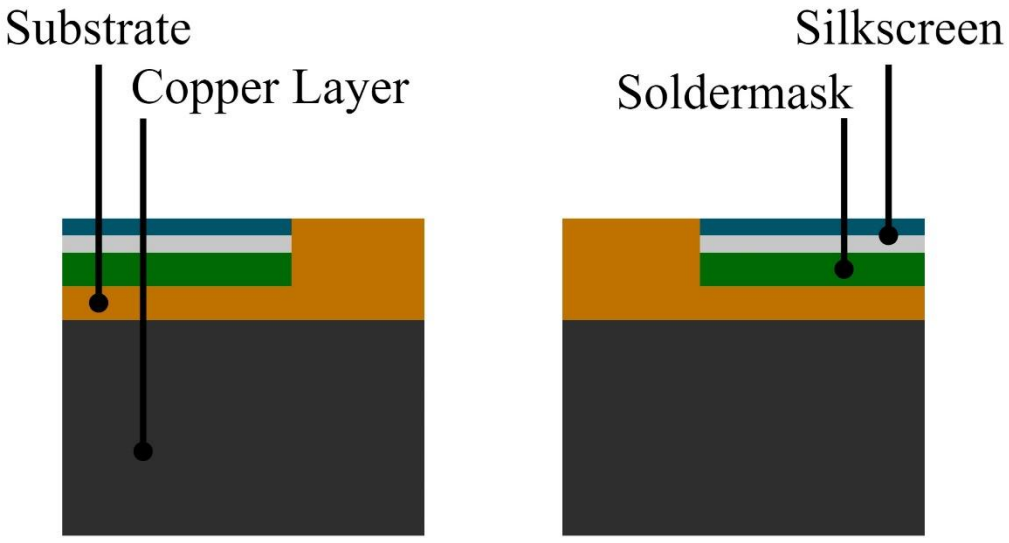

Figure 2.27 – Construction of Single-Sided PCB Board

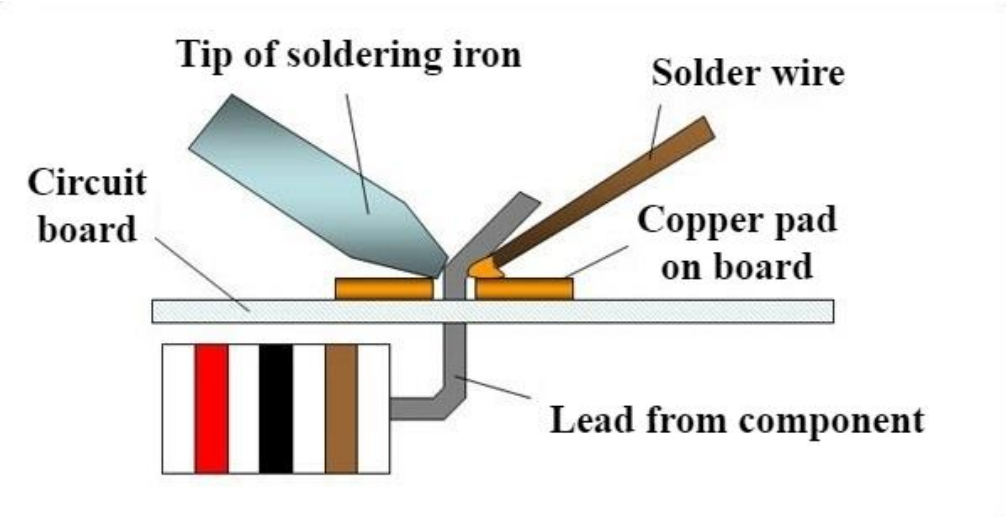

Figure 2.28 – Basic Method of Soldering Electronic Components

To begin the design and plan procedure; firstly, the main board's electronic components and modules should be listed in order to simplify the process and hence give a starting point. Below is the complete list of each module and component to be soldered into the board itself:

1 – Microcontroller.
2 – Inertial Measurement Unit (IMU).
3 – Global Positioning System unit (GPS).
4 – Bluetooth and/or RF unit.
5 – Electronic Speed Controllers (ESCs) connectors.
6 – Sensors (On-board ambient light sensor, battery sensor, and connectors for external sensors – if needed).
7 – Input power connectors

The prototype board can be modified later for additional connections and/or components to be added such as flash light, servo motor to title a mounted camera, and so forth. Figure 2.29 shows the block diagram design and connections for each electronic part to be used within the main board.

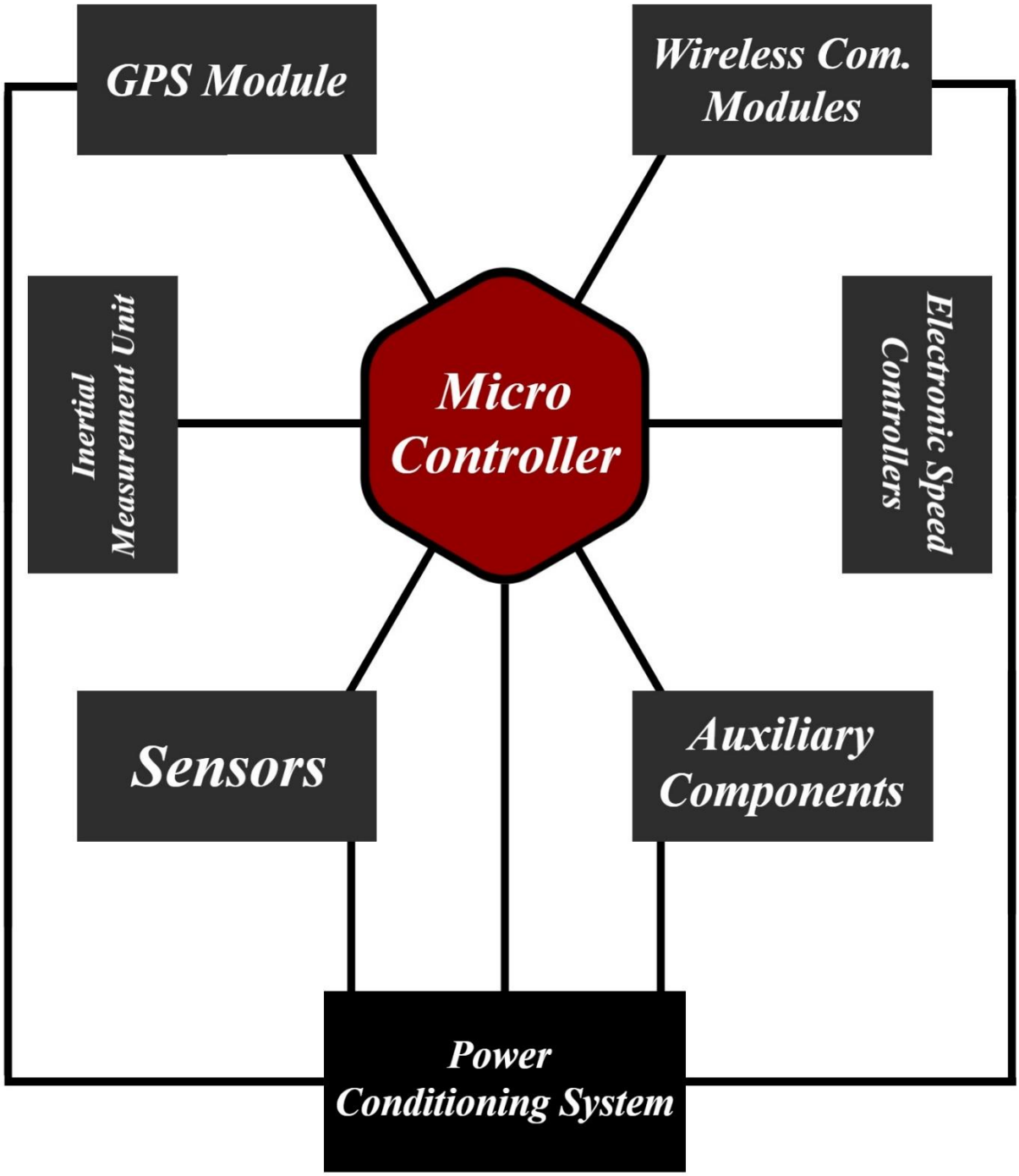

Figure 2.29 – Block Diagram of Main Board Components & Modules

It should be noted as before that the microcontroller can provide regulated 3.3V to any electronic device that operates on that region. Other microcontrollers can provide both regulated 3.3V and 5V output; however, these output voltages suffers from inefficiency due to the problem in the nature of voltage regulators which step-down voltages by dissipating heat as discussed previously in **Section 2.3**. Therefore, choosing which microcontroller and power circuit approach depends on the hardware designer him/her-self. In the upcoming pages, a brief summary on each main board's subsystem will be discussed in details to understand how a microcontroller communicates with each module and component separately via each port. Before discussing each subsystem, a list of each electronic unit used in this project is given. These units are available commercially in the markets.

1 – ESP32 Microcontroller by Espressif Systems
2 – MPU6050 IMU by InvenSense Inc.
3 – HC-05 Bluetooth Module / nRF24L01 RF Module by Nordic Semiconductor
4 – Ultimate GPS Module by Adafruit Industries and MediaTek Inc.

The wireless communication modules block as seen in Figure 2.29 consists of two modules both of which are the Bluetooth and the RF module. For the purpose of prototype and testing a Bluetooth module is sufficient to act as a channel for both data exchange and movement control of the quadcopter. One should be aware that for long distance control, Bluetooth modules are not applicable; therefore, to allow the drone to fly over long distances a radio frequency modules should be used instead. Nonetheless, the basic idea of movement control is that a PWM signal is used as movement control signal for both the roll, pitch, and yaw angles. This signal can be sent either in a raw format from the RF unit or in an encoded format from the Bluetooth unit. The HC-05 module can have ranges up to 9 meters (30 ft.), on the other hand the nRF24L01 module can achieve a wireless link up to 1 kilometers (3,380 feet) at 250 kbps. For this obvious reason it is necessarily and important to use RF modules for the final design. One important detail to be noted down that is despite the fact it is impractical to use Bluetooth as a means of controlling the quadcopter; however, they provide a backup controller in case the main hand-held controller is out of power or faulty. For this reason a backup virtual analog controller feature is implemented into the smartphone companion app as will be seen in the next chapter.

Details about each electronic unit used in the project including number of pins, function of each pin, operating power, features of the unit, and so forth is given as follows:

ESP32 Microcontroller

The ESP32 microcontroller will be used to control and operate the main functions of the quadcopter. ESP32 is a series of low-cost, low-power system on a chip microcontrollers with integrated Wi-Fi and dual-mode Bluetooth. The ESP32 series employs a Tensilica Xtensa LX6 microprocessor in both dual-core and single-core variations and includes in-built antenna switches, RF balun, power amplifier, low-noise receive amplifier, filters, and power-management modules. ESP32 is created and developed by Espressif Systems, a Shanghai-based Chinese company, and is manufactured by TSMC using their 40 nm process. It is a successor to the ESP8266 microcontroller. [9]

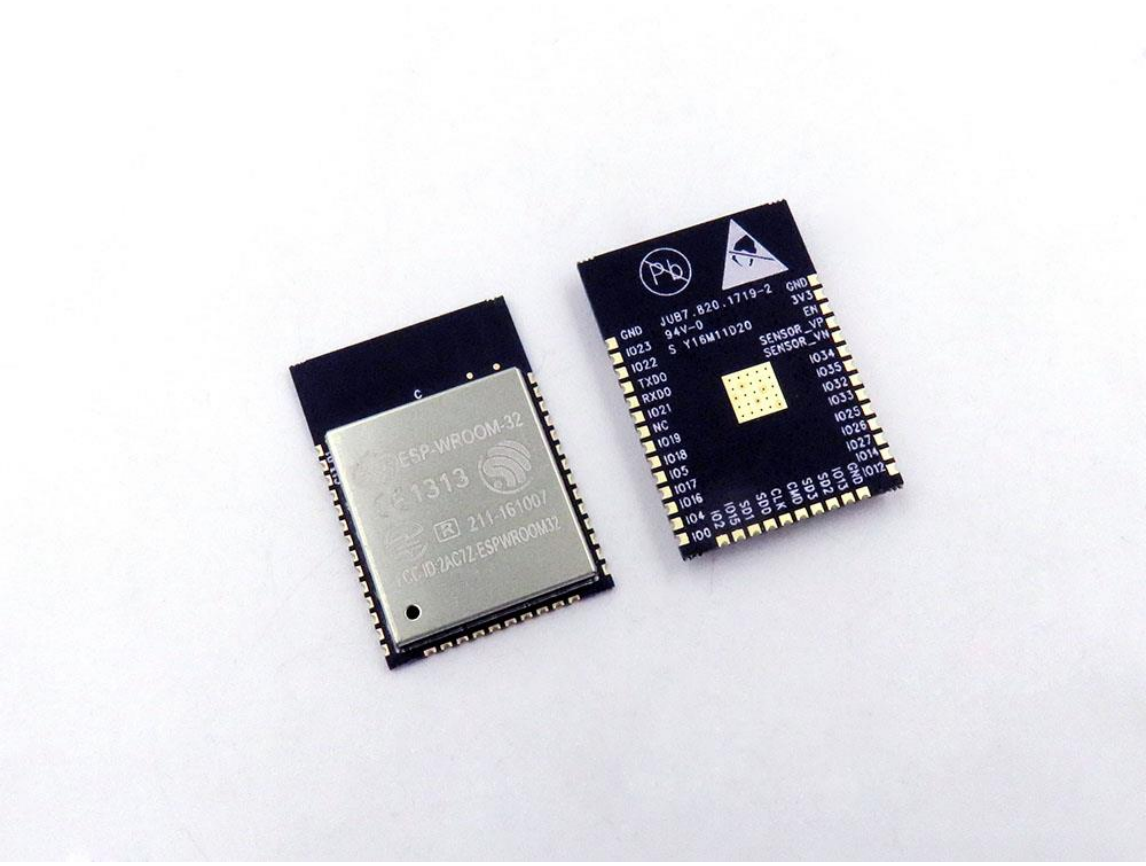

Figure 2.30 – ESP32 Microcontroller with Built-in Shielding

A functional block diagram of the ESP 32 chip is shown in Figure 2.31, this includes the core and memory, radio, cryptographic hardware acceleration, RTC and low power management subsystems, peripheral interfaces, embedded flash memory, Bluetooth and Wi-Fi basebands, Bluetooth link controller, and Wi-Fi MAC.

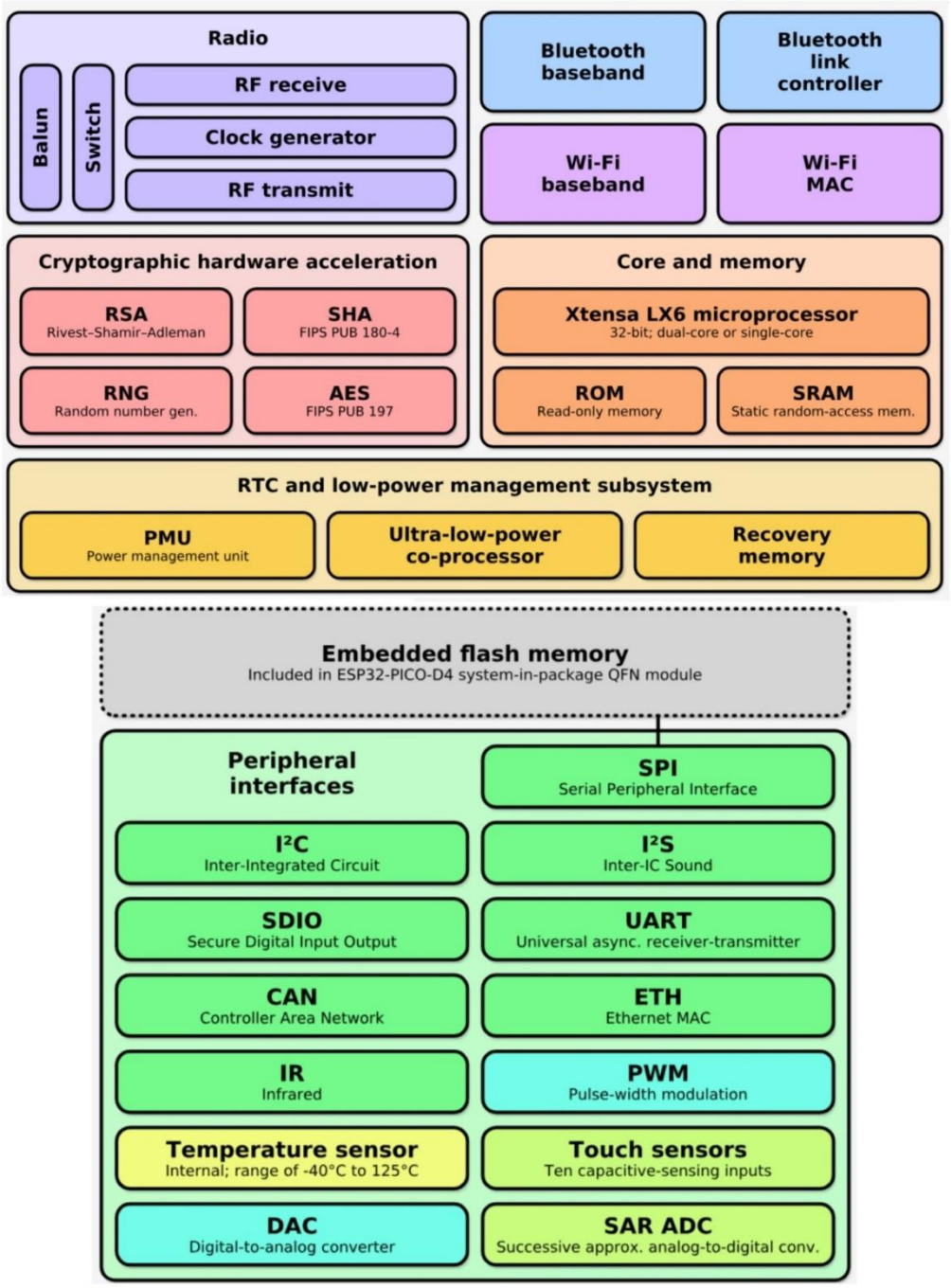

Figure 2.31 – Espressif ESP32 Chip Functional Block Diagram

***Notice** that the ESP32 has an already in-built antenna band with integrated Wi-Fi and dual-mode Bluetooth which can be used as a replacement for the Bluetooth module; however, this mode has not been implemented fully with the **Arduino** platform. The Arduino Bluetooth sketch code for the ESP32 microcontroller development board uses almost 95% of program storage space leaving only 5% of program space available for the user. Additionally, using radio communication with ESP32 restricts the usage of the dual core CPUs allowing only a single core to handle all the tasks alone which, therefore, adds more pressure for the CPU to run multiple tasks simultaneously. It should be noted that the ESP32 chip is a non-unified flash-based microcontroller, meaning it usually requires an external SPI flash memory. This memory is upgradable depending on the manufacturer development board. For the sake of this project an ESP32 DoIT DevKit development breakout board will be used with maximum of 1310720 bytes of external flash storage. Development & break-out boards extend wiring and may add functionality, often building upon ESP32 module boards and making them easier to use for development purposes. The ESP32 DoIT DevKit development board is compatible with Arduino IDE software allowing an easier and simpler code development to be implemented. It also adds pin mapping, USB-to-TTL programming chip, and 3.3V voltage regulator all together in a single board.

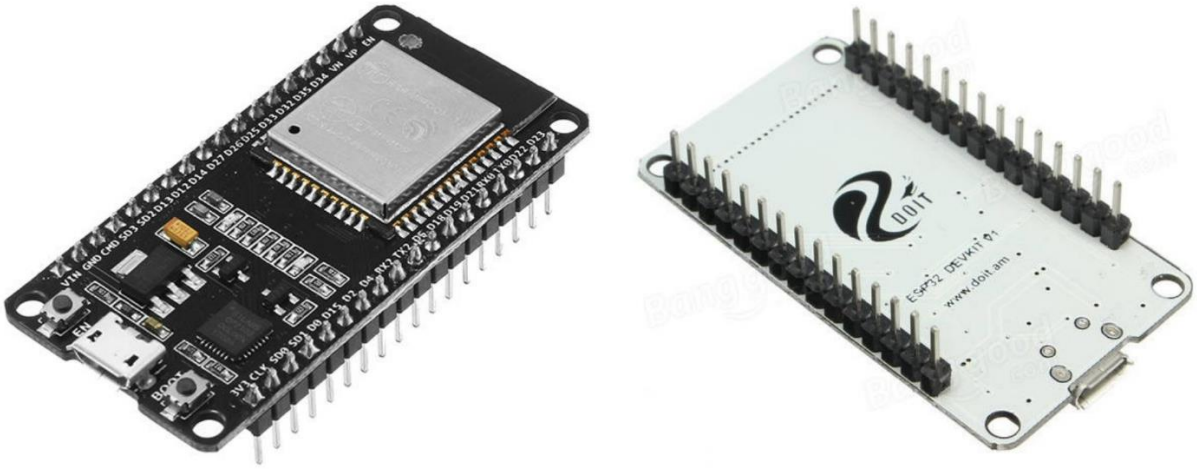

Figure 2.32 –ESP32 DoIT DevKit V1 Development Board

Table 4 summarizes the basic features and specs of the ESP32 DoIT DevKit development board.

Specs	*Information*	*Side Note*
Microcontroller	Tensilica 32-bit Single-/Dual-core CPU Xtensa LX6. Ultra low power (ULP) co-processor.	Dual core CPU operates at 160 or 240 MHz and performing up to 600 DMIPS. 5uA deep sleep current consumption.
Operating Voltage	3.3V	Has built it voltage regulator with input voltages from 7V and up to 12V
Wireless Connectivity	Built-in antenna band. Supports Wi-Fi and dual mode Bluetooth.	Wi-Fi: 802.11 b/g/n Bluetooth: v4.2 BR/EDR and BLE.
Peripheral Interfaces	30 Pins	25 digital I/O pins, 6 ADC pins, 2 DAC pins, 3 UARTS, 2 SPIs, and 3 I²C s.
Sensors	Temperature, hall effect, and touch sensors.	10 pins can be used as capacitive touch detection.
Programming	USB-to-TLL chip.	Supports USB Micro B connector for programing and power.
Security	IEEE 802.11 standard security, secure boot, flash encryption, cryptographic hardware acceleration, and 1042-but OTP.	IEEE 802.11 standard security features all supported, including WFA, WPA/WPA2 and WAPI.

Table 4 – ESP32 DoIT DevKit Development Board Specs and Features

MPU6050 Inertial Measurement Unit

The MPU-6050 is an integrated 6-axis motion tracking device that combines a 3-axis gyroscope, 3-axis accelerometer, and a digital motion processor (DMP). MPU6050 features three 16-bit analog-to-digital converters for digitizing the gyroscope outputs and three 16-bit ADCs for digitizing the accelerometer outputs. For precision tracking of both fast and slow motions, the parts feature a user-programmable gyroscope full-scale range of ±250, ±500, ±1000, and ±2000°/sec (dps) and a user-programmable accelerometer full-scale range of ±2g, ±4g, ±8g, and ±16g. An on-chip 1024 Byte FIFO buffer helps lower system power consumption by allowing the system processor to read the sensor data in bursts and then enter a low-power mode as the MPU collects more data. With all the necessary on-chip processing and sensor components required to support many motion-based use cases, the MPU-6050 uniquely enables low-power motion interface applications in portable applications with reduced processing requirements for the system processor. By providing an integrated MotionFusion™ output, the DMP in the MPU-6050 offloads the intensive motion processing computation requirements from the system processor, minimizing the need for frequent polling of the motion sensor output.[10] For this reason the MPU6050 is considered an attractive choice when designing a quadcopter system. An I²C communication protocol is used between the microcontroller and the module itself to fetch the gyroscope and accelerometer data at x, y, and z axis. The microcontroller, with the help of software filtering, can then process the incoming data using sensor fusion to produce roll, pitch, and yaw angles to be used for the PID controller to stabilize the quadcopter system. For this project, the MPU6050 GY-521 module with built-in 3.3V voltage regulator will be used.

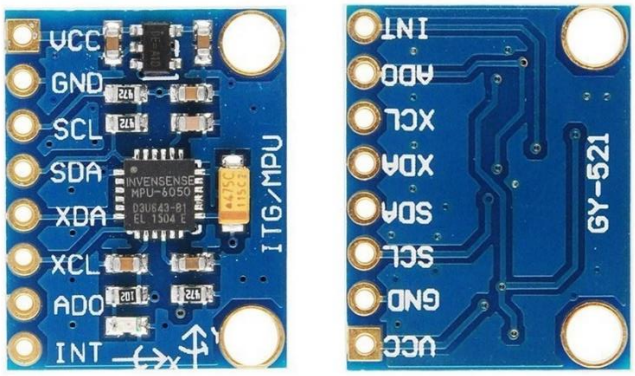

Figure 2.33 – MPU6050 GY-521 IMU Module

Figure 2.34 shows the block diagram of the MPU60X0 IMU. Note from the figure that pin names in round brackets apply only to MPU6000 module; on the other hand, pin names in square brackets apply only to MPU6050. This figure is taken from reference [10].

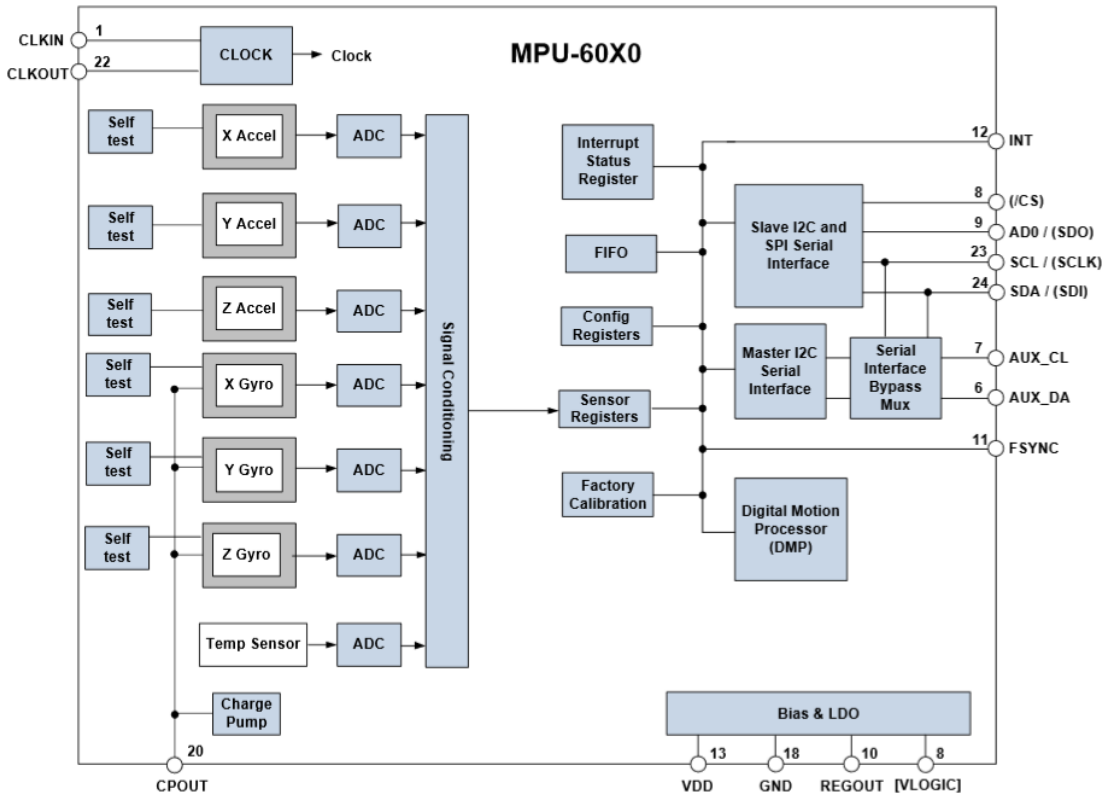

Figure 2.34 – MPU60X0 Block Diagram

Figure 2.35 shows the orientation of axes of sensitivity and polarity of rotation for MPU60X0

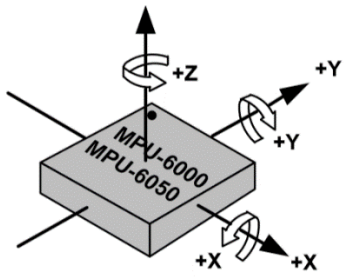

Figure 2.35 – MPU60X0 Orientation of Axes of Sensitivity and Polarity of Rotation

Table 5 summarizes the basic features and specs of MPU6050 GY-521 IMU Module.

Specs	*Information*	*Side Note*
Chip	MPU6050	Chip built-in 16-bit ADC, 16-bit data output.
Operating Voltage	2.375V-3.46V	Has onboard regulator with input voltages from 3V and up to 5V
Communication Mode	Standard I²C communication protocol	I²C communication protocol at 400KHz.
Peripheral Interfaces	8 Pins	VCC pin, GND pin, 2 I²C pins, 2 external I²C pins, I²C address select pin, and an interrupt pin.
Function	Provides tri-axis accelerometer and gyroscope data (6 DOF).	Directly accepts inputs from an external 3-axis compass to provide a complete 9-axis MotionFusion™ output.
Sensors	Digital-output temperature sensor.	Temperature in degrees C = (TEMP_OUT Register Value as a signed quantity)/340 + 36.53

Table 5 – MPU6050 GY-521 IMU Module Specs and Features

Adafruit Ultimate GPS Module

The Adafruit Ultimate GPS module is a board built around the MTK3339 chipset, a high-quality GPS module by MediaTek that can track up to 22 satellites on 66 channels, has a high-sensitivity receiver (-165 dB tracking), and a built in antenna. It can do up to 10 location updates a second for high speed, high sensitivity logging or tracking. Power usage is at low ranges, only 20 mA during navigation. The MT3339 includes on-chip CMOS RF, digital baseband, and ARM7 CPU. It is able to achieve the industry's highest level of sensitivity, accuracy and Time-to-First-Fix (TTFF) with the lowest power consumption in a small-footprint lead-free package. Its small footprint and minimal BOM requirement provide significant reductions in the design, manufacturing and testing resource required for portable applications. [11] The version of Adafruit Ultimate GPS used during this project is the third version of the series. Two features that really stand out about version 3 MTK3339-based module is the external antenna functionality and the built-in data-logging capability. The module has a standard ceramic patch antenna; however, a user can have a bigger antenna by snapping on any 3V active GPS antenna via the uFL connector. The module will automatically detect the active antenna and switch over. The average time of warm/cold start is about 34 seconds, operating current during satellite tracking is about 25 mA respectively.

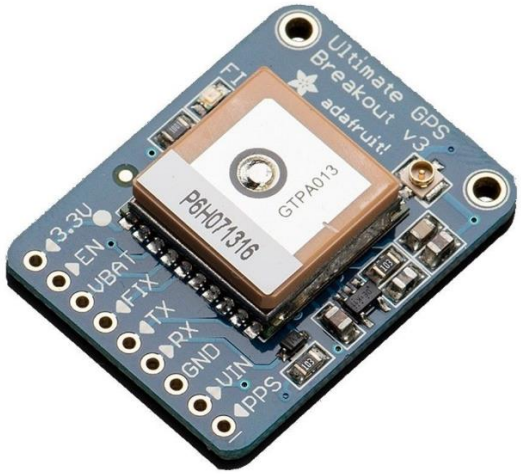

Figure 2.36 – Adafruit Ultimate GPS V3 Module

Figure 2.37 shows the block diagram of the MTK3339 chipset used in the Adafruit Ultimate GPS module. This figure is taken from reference [11].

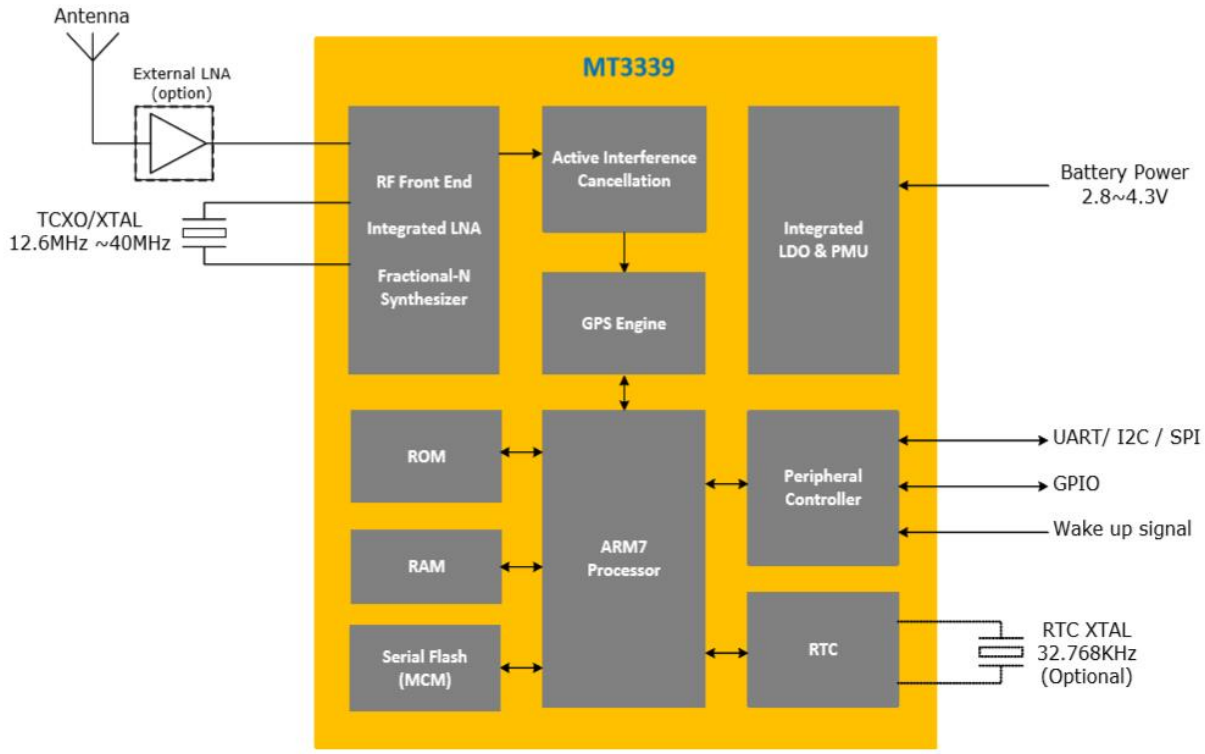

Figure 2.37 – MTK333 System Block Diagram

During operation the GPS module will always send data even without a satellites fix. In order to get 'valid' (not-blank) data, the GPS module should be placed directly outside, with the square ceramic antenna pointing up with a clear sky view. In ideal conditions, the module can get a fix in under 45 seconds. However, depending on your location, satellite configuration, solar flares, tall buildings nearby, RF noise, etc. it may take up to half an hour (or more) to get a fix. Data received from the GPS module are in the NMEA (National Marines Electronic Association) format. This data includes the complete PVT (position, velocity, time) solution computed by the GPS receiver. The idea of NMEA is to send a line of data called a sentence that is totally self-contained and independent from other sentences.

Table 6 summarizes the basic features and specs of Adafruit Ultimate GPS V3 Module.

Specs	*Information*	*Side Note*
Chip	MTK3339 chipset	Chip built-in low noise amplifier, image-rejection mixer, automatic center frequency calibration BPF, and power management system.
Operating Voltage	2.8V-4.3V	Has onboard regulator with input voltages from 3V and up to 5V
Communication Mode	Standard UART communication protocol.	Fixed 9600 baud rate communication speed.
Peripheral Interfaces	9 Pins	VCC pin, GND pin, 2 UART pins, 3.3V output pin, enable pin, 3.3V coin cell battery input pin, interrupt pin, GPS fix output pin, and a pulse per second output pin.
Function	Provides GPS data in the form of NMEA format.	In order to get valid NMEA data, the GPS module should be placed directly outside, with the square ceramic antenna pointing up with a clear sky view.

Table 6 – Adafruit Ultimate GPS V3 Module Specs and Features

HC–05 Bluetooth Module

To provide a means of data transfer between the smartphone and the quadcopter a Bluetooth communication channel is used. HC-05 Bluetooth Module is a Bluetooth SPP (Serial Port Protocol) module, designed for transparent wireless serial connection setup. Its communication is via UART communication which makes an easy way to interface with controller. This serial port Bluetooth module is fully qualified Bluetooth v2.0+EDR (Enhanced Data Rate) 3Mbps Modulation with complete 2.4GHz radio transceiver and baseband. It uses CSR Bluecore 04-External single chip Bluetooth system with CMOS technology and with AFH (Adaptive Frequency Hopping Feature). The HC-05 embedded Bluetooth module has two working modes: order-response work mode and automatic connection work mode. And there are three work roles (Master, Slave and Loopback) at the automatic connection work mode. When the module is at the automatic connection work mode, it will follow the default way set lastly to transmit the data automatically. When the module is at the order-response work mode, user can send the AT command to the module to set the control parameters and sent control order. The work mode of module can be switched by controlling the module PIN (PIO11) input level.

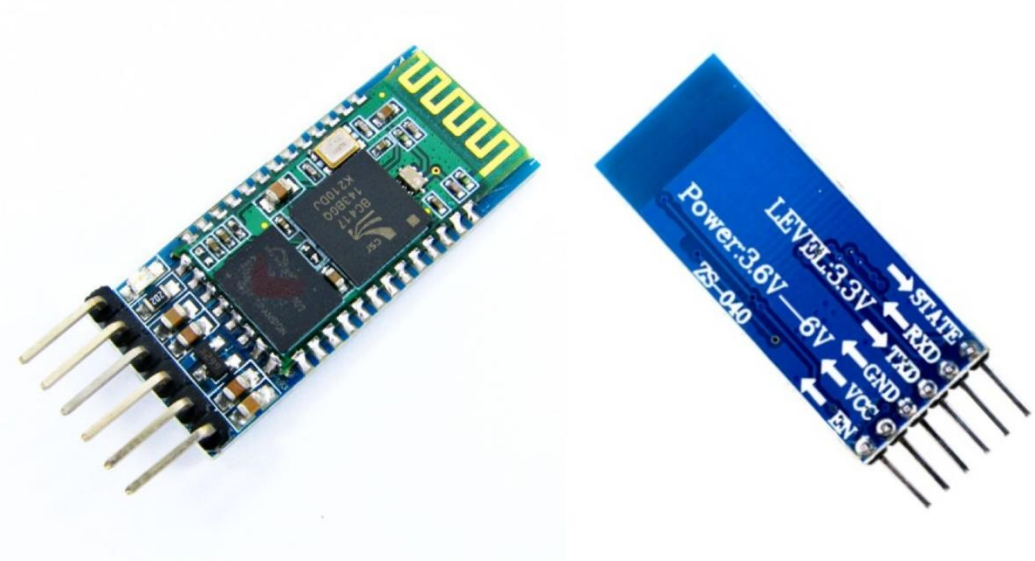

Figure 2.38 – HC-05 Bluetooth Module

Table 7 summarizes the basic features and specs of HC-05 Module.

Specs	*Information*	*Side Note*
Microcontroller	CSR-BC417 chip	Interfaces to 8Mbit of external Flash memory. When used with the CSR Bluetooth software stack, it provides a fully compliant Bluetooth system to v2.0 of the specification for data and voice communications.
Operating Voltage	3.15V	Has onboard regulator with input voltages from 3.6V and up to 6V
Communication Mode	Standard UART communication protocol.	AT Command mode requires a fixed 38400 baud rate. Normal operation mode of baud rate up to 115200.
Peripheral Interfaces	6 Pins	VCC pin, GND pin, 2 UART pins, EN pin, and STATE pin.
Function	Provide Bluetooth v2.0 full duplex communication channel.	The module can be used as either master or slave mode.
Programming	AT command Mode	AT command mode can be switched by controlling the module PIN (PIO11) input level. AT command mode operates at 38400 baud rate

Table 7 – HC-05 Bluetooth Module Specs and Features

Now that each component has been summarized, the main board's block diagram can; therefore, include the connections between each component's peripheral pins as seen in Figure 2.39.

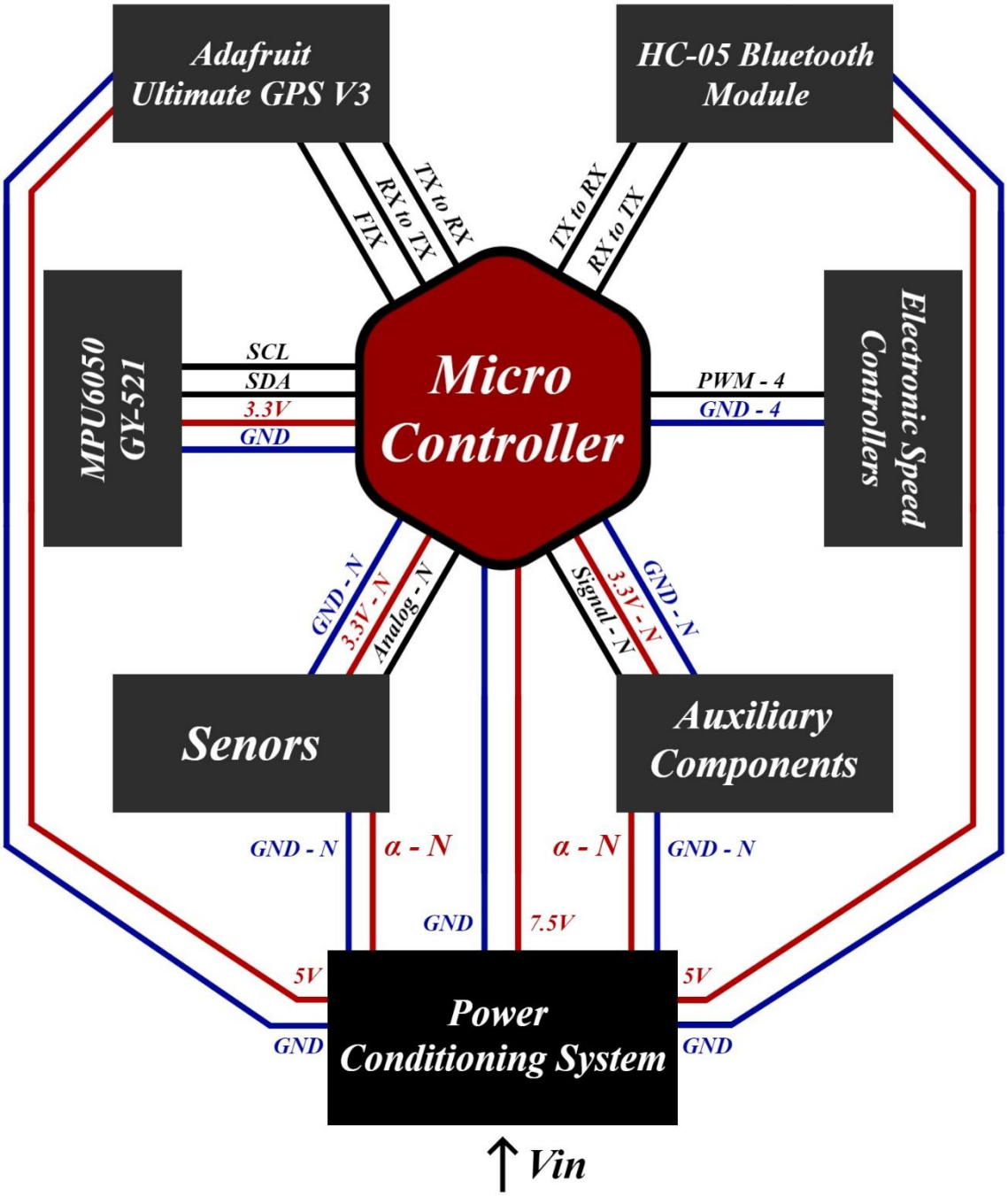

Figure 2.39 – Block Diagram of Prototype Main Board Components & Modules Including their Connections.

From Figure 2.39, the symbol α represents voltages of 5V, 7.5V, and the raw Vin voltage from the main power supply. N is an integer number of connections greater than zero. 'Signal' connection represents an output signal to an auxiliary component such as PWM, logic, UART, I²C, and so forth, whereas 'Analog' connection represents the analog read input to the microcontroller respectively. The electronic speed controllers take an input signal in the form of PWM, and since four electronic speed controllers will be used with their corresponding four DC brushless motors; therefore, a four PWM output signals are taken as control signal to vary the speed of the motors. Both the sensors and auxiliary components blocks takes input voltages of 3.3V from the microcontroller, 5V, 7.5V, and raw Vin voltage. The raw voltage is useful to read the value of the power supply using a basic voltage divider sensor. This value can then be used to compensate the speed of the motors in correspondence with the input voltage from the main power supply. Notice an RF Com channel is not represented in this diagram since the prototype model will only rely on Bluetooth communication channel for movement control; however, an RF Com channel will be considered as part of auxiliary components block. ***Keep in mind** for long range communication an RF channel must be used! Only for prototype and testing purpose we can rely on Bluetooth communication for now. Figure 2.40 shows the achieved design of the main board prototype.

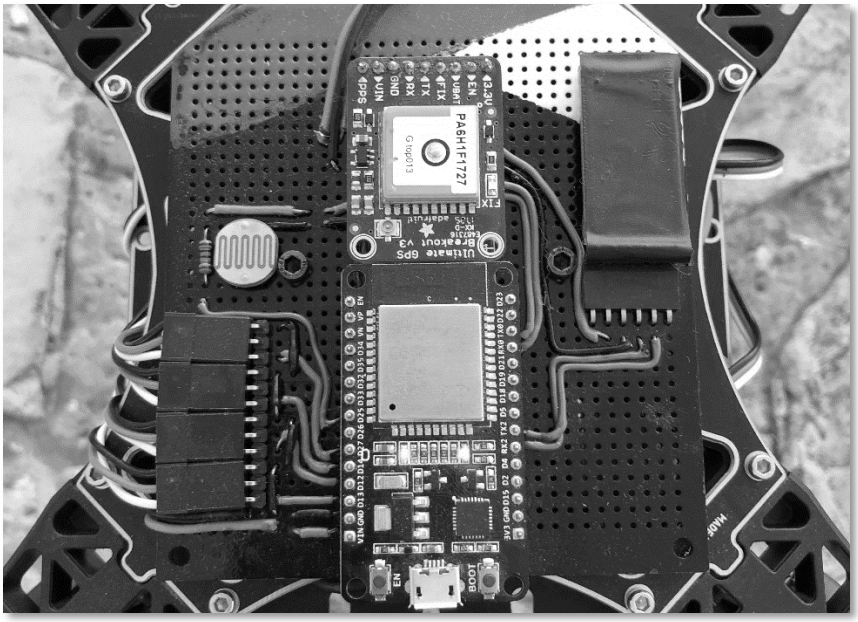

Figure 2.40 – Quadcopter Main Board Prototype

2.5 Body Frame

For the quadcopter to hold all the components together, a rigid, tough, and lightweight chassis should be considered. The chassis should be capable of withstanding vibrations from the motors and external forces acting upon it such as wind and shock. A symmetrical design is used to distribute the mass equally across the body. Mounting screws should be kept in mind when designing or choosing a frame to be able to hold the brushless motors fixed and tight in place. Quadcopter's arms also play a vital role in the fight against vibrations, which can cause a number of different issues. Flight controllers, with their sensitive gyroscopes, do not generally react well to incessant shaking. Jostle them too much through a poor setup and you could see erratic behavior, sometimes bad enough to cause crashes. Vibrations are also the dread of anyone hoping to use a camera on a quadcopter. The frame of the quadcopter that will be used during this project will be an integrated PCB S500 quadcopter frame. Figure 2.41 shows a 3D model of the S500 frame (propellers, motors and ESCs are included).

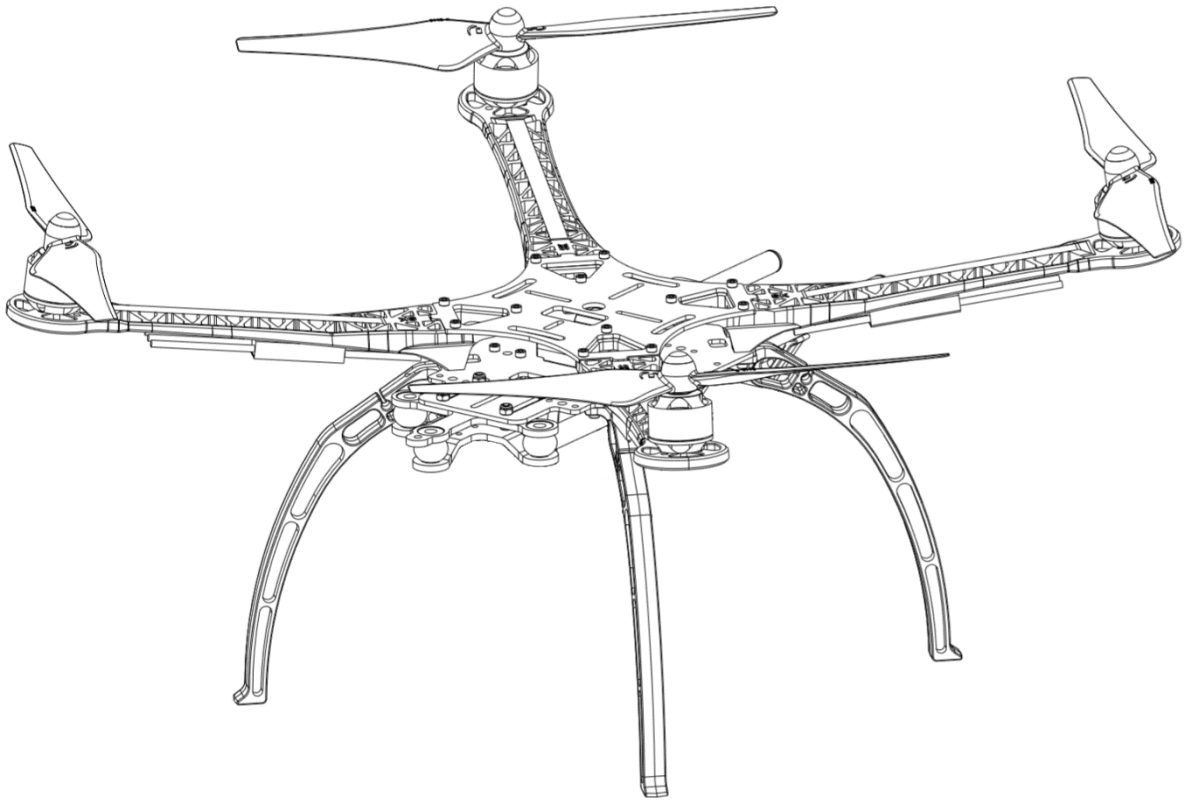

Figure 2.41 – S500 Quadcopter Frame with Integrated PCB

Common is the integration of power distribution PCB circuits into the frame plates. Battery connections are soldered directly to pads built into the material, and power can be tapped at other solder points without the need for extensive, messy wiring. Although by no means essential, a clean setup is satisfying and less likely to fail, making power-distributing frames a worthy decision. The frame adds an extra feature by using a specially designed angled arms that increase flight stability while minimizing power loss. Additionally, the frame is mainly made of durable PCB with lightweight carbon fiber material. The scale of a multi-rotor is often denoted by the horizontal width of the frame assembly, including its arms. The standard measurement is taken in millimeters from motor to motor through the center of the frame. It should be noted that carbon fiber is known to block radio signals, which is obviously not ideal for a system that depends on multiple transmissions. It can be used though, and is often. Therefore, caution should be taken when choosing a carbon fiber frame since blocked signals are a possibility. Figure 2.42 shows a diagram of the S500 power distribution PCB (ESCs are included).

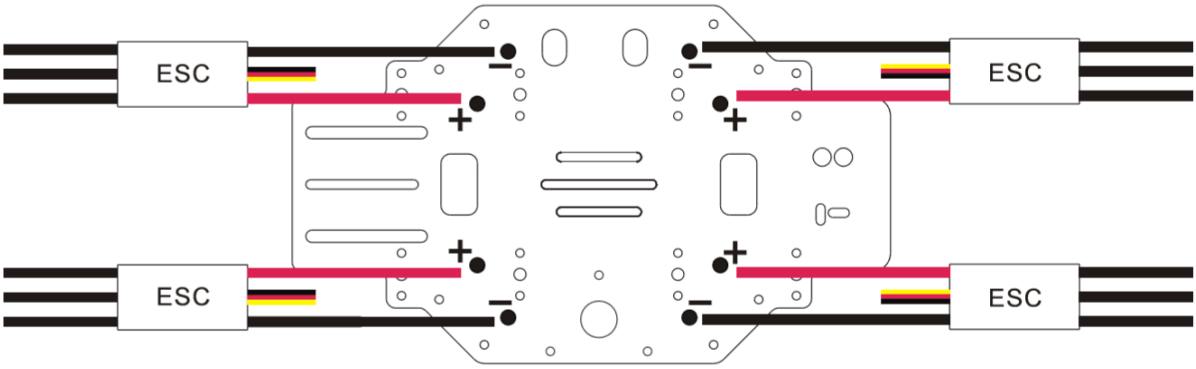

Figure 2.42 – S500 Power Distribution PCB Including ESCs

The S500 quadcopter frame consists of two PCB boards. The lower PCB board holds the power conditioning circuit as well as handles power distribution to ESCs. On the other hand, the main board can be mounted on the upper PCB board to provide clear, line-of-sight signals propagation for GPS and other communication modules.

Table 8 shows the specifications and requirements of the S500 quadcopter frame.

Specs	*Information*
Built	The frame is built from glass fiber and polyamide nylon. Pre-threaded brass sleeves for all the frame bolts. Integrated PCB connection for direct soldering of battery and ESCs.
Height	From the frame's legs to the tip of the arms is approximately 172mm. Arms elevation increase the height to 190mm. Legs height are 150mm.
Weight	Total frame weight is 405g. This excludes any motors, ESCs, battery, and board components.
Motors Centers	Motor centers is 480mm
Motors Ratings	4 x 28mm 800KV ~ 1000KV DC brushless motors
ESCs Rating	4 x 18A ~ 30A speed controllers
Propellers	4 x 9x4.7 ~ 10x5 props
Battery	1 x 2200mAh 3S 11.1V LiPo battery or more

Figure 8 – S500 Quadcopter Specs & Requirements

2.5.1 ESCs and Motors

The selection of ESCs and motors depends on the frame of the quadcopter as well as the maximum weight carrying capacity. Motors used for drone application normally are DC brushless motors. Brushless DC electric motor (BLDC motors, BL motors) also known as electronically commutated motors (ECMs, EC motors), or synchronous DC motors, are synchronous motors powered by DC electricity via an inverter or switching power supply produces an AC electric current to drive each phase of the motor via a closed loop controller. The controller provides pulses of current to the motor windings that control the speed and of the motor. With the rapid development of semiconductor electronics, it allowed the commutator and brushes to be eliminated in DC motors. In brushless DC motors, an electronic servo system replaces the mechanical commutator contacts. An electronic sensor detects the angle of the rotor, and controls semiconductor switches such as transistors which switch current through the windings, either reversing the direction of the current, or in some motors turning it off, at the correct time each 180° shaft rotation so the electromagnets create a torque in one direction. The elimination of the sliding contact allows brushless motors to have less friction and longer life; their working life is only limited by the lifetime of their bearings. [12] The speed control mechanism is used in drones are the electronic speed controllers (ESC) circuit. An electronic speed control follows a speed reference signal (derived from a throttle lever, joystick, or other manual input) and varies the switching rate of a network of field effect transistors (FETs). By adjusting the duty cycle or switching frequency of the transistors, the speed of the motor is changed. The rapid switching of the transistors is what causes the motor itself to emit its characteristic high-pitched whine, especially noticeable at lower speeds. [13] Electronic Speed Controllers are an essential component of modern quadcopters (and all multirotors) that offer high power, high frequency, and high resolution 3-phase AC power to the motors in an extremely compact miniature package. These craft depend entirely on the variable speed of the motors driving the propellers. This wide variation and fine RPM control in motor/prop speed gives all of the control necessary for a quadcopter (and all multirotors) to fly. For simplicity sake further details on how ESCs and DC brushless motors work will not be covered entirely in this documentation report and it's left for the reader to study more about them in other references. The main idea here is to understand the basics of ESCs and DC brushless motors in that the ESCs take the DC input voltage from the battery source and convert it to a 3-phase AC signal to

drive the motors and vary their speeds. Input control signal of the ESC is usually in the form of PWM signal that vary from maximum to minimum throttle speed (Typically 1000us to 2000us pulse). The list of ESCs, motors, and propellers used in this project are as follows:

1 – Readytosky RS 2212 – 920KV brushless DC motor.
2 – Hobbywing Skywalker 40A Electronic speed controller.
3 – Gemfan 10x4.5 propeller.

Images of each of the above three components are shown in Figure 2.43 to Figure 2.45, respectively.

Figure 2.43 – Readytosky RS 2212 920KV Brushless DC Motors (CW & CCW)

It should be noted that DC brushless motors comes in two pairs – CW (clockwise) and CCW (counter clockwise) formation. Basically CW and CCW motors are still the same motors, apart from the prop shaft threads which have opposite direction. That means for a CCW motor, the prop nut is secured by turning clockwise. And for CW motors, the prop nuts needs to be rotated

anti-clockwise in order to fasten. By using threads that are the opposite to motor spin direction, the prop nuts automatically lock themselves down and won't come loose.

Figure 2.44 – Hobbywing Skywalker 40A Electronic Speed Controllers

Figure 2.45 – Gemfan 10x4.5 Propellers (CW & CCW)

Figure 2.46 and 2.47 shows images of the quadcopter system

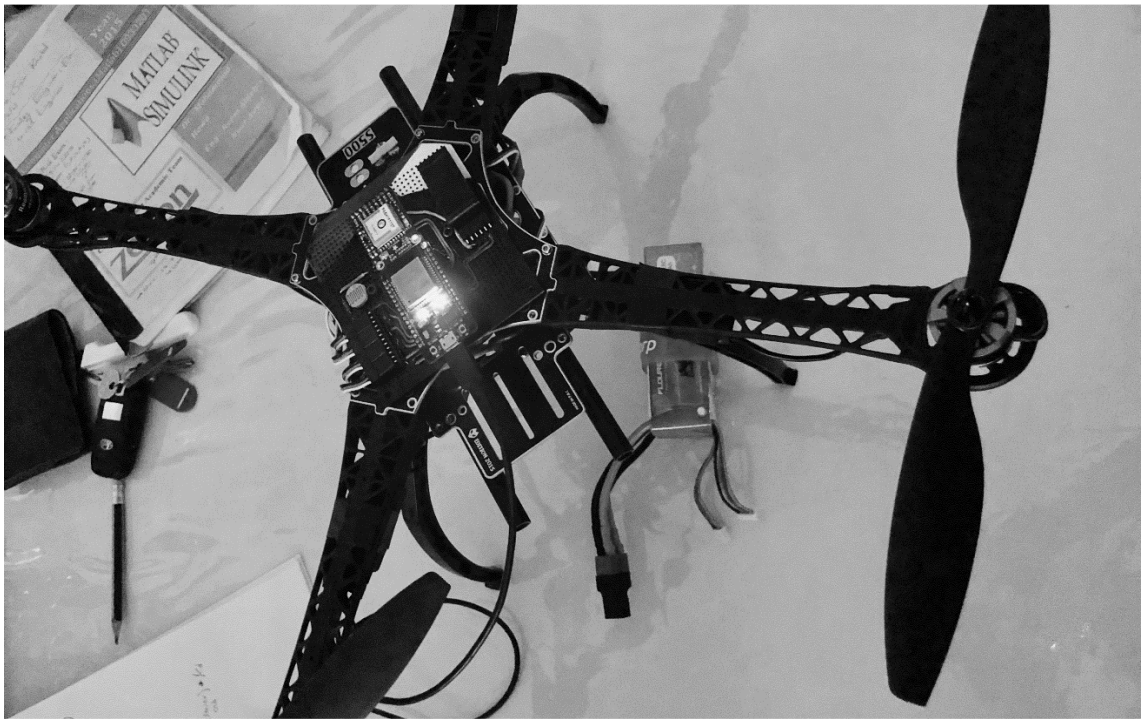

Figure 2.46 – Quadcopter System Far View

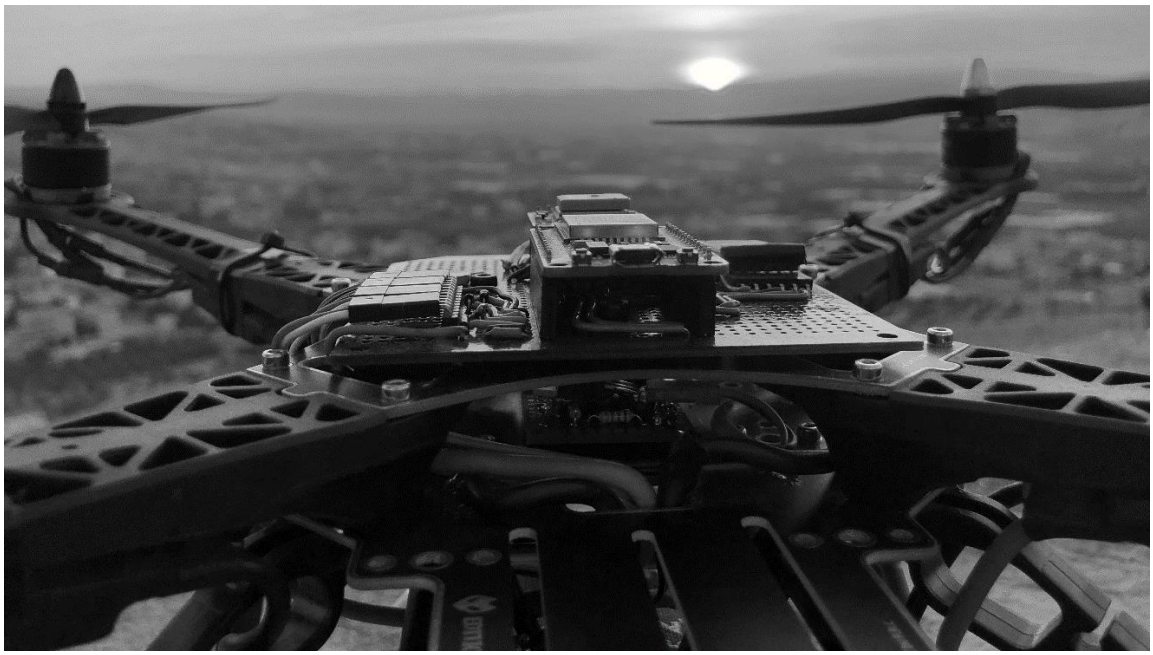

Figure 2.47 – Quadcopter System Near View

CHAPTER 3

SOFTWARE DEVELOPEMNT

3.1 Arduino Code Development

In Chapter 2 a brief discussion on the modeling and design of a quadcopter has been covered. Using the modeling equations given previously we can apply them into the microcontroller to achieve stability and controllability of the system. The ESP32 microcontroller program can be compiled and uploaded using multiple software languages such as C, C++, JavaScript, and MicroPython. One of many software platforms used to write program codes for the ESP32 microcontroller is the Arduino IDE. Arduino is an open-source hardware and software company, project and user community that designs and manufactures single-board microcontrollers and microcontroller kits for building digital devices and interactive objects that can sense and control objects in the physical and digital world. The Arduino IDE uses a set of C++/C code languages to compile and upload a program onto a microcontroller, the IDE is written in Java, however. Despite the fact that the ESP32 chip is not manufactured by Arduino; however, the chip can be fully programmed within the Arduino platform. Therefore, libraries used for Arduino hardware can thus be used and implemented as well within the ESP32 code structure.
The Arduino code structure is divided into three main parts:

1 – The setup method
2 – The main method
3 – The user-defined methods

Both the setup and main method blocks should always be included in the program. A user-defined method can be any function where the setup and main methods use it as a callback, this include a section of a program code, a method with input parameters, a method with a return statement, an interrupt service routines and so forth. The setup code block specifies the beginning of program execution within the microcontroller. This code segment is only executed once; therefore, it is understandable that this portion of the program is used to initialize the settings of the microcontroller such as input/output mapping of pins, UART baudrate setup, I²C

communication setup, external modules setup and wakeup protocols, and so on. Whereas the main code is used to execute a block of codes indefinitely for as long as the microcontroller is powered on and is not in the deep sleep state. A simple Arduino code structure can be seen in Figure 3.1

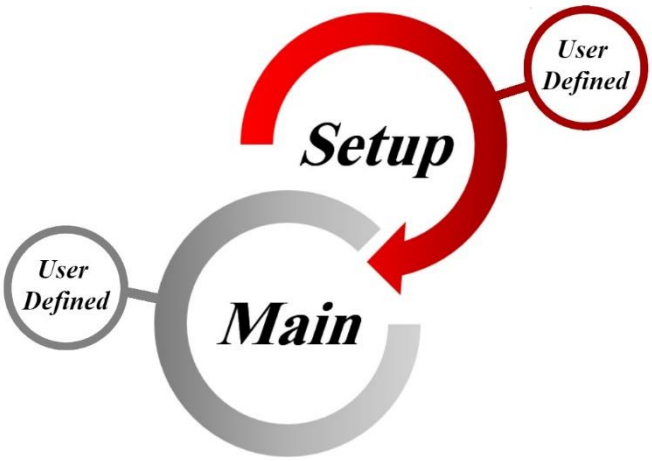

Figure 3.1 – Arduino Code Structure

We begin discussing each section of the code separately starting from the setup code towards the main loop code including all user-defined methods used. A flow chart diagram of the code will be introduced afterwards.

Setup Code

The setup code is the first thing the microcontroller executes and it is done only once. Defining the function of each peripheral interface within the controller is done in this part. Additionally, any hardware setup procedure should be included at the start of the code. To configure the specified pin to behave either as input or output the following line of code is used:

```
pinMode(A, B);                // Pin-mode assignment
```

Where A is the pin number and B is the function of the pin either as: INPUT, OUTPUT or INPUT-PULLUP. Many Arduino libraries automatically assign the pin function by simply

plugging in the specified pin number. For the ESP32 microcontroller, the output logic levels are 0 and 3.3V; whereas, the input voltage ranges from 0 to 3.3V. It should be noted that not all pins within the ESP32 microcontroller are bi-directional, thus it is recommended to use the pin mapping diagram as a reference before pin assignments. Figure 3.2 shows the pin mapping for the ESP32 DoIT DevKit V1 board.

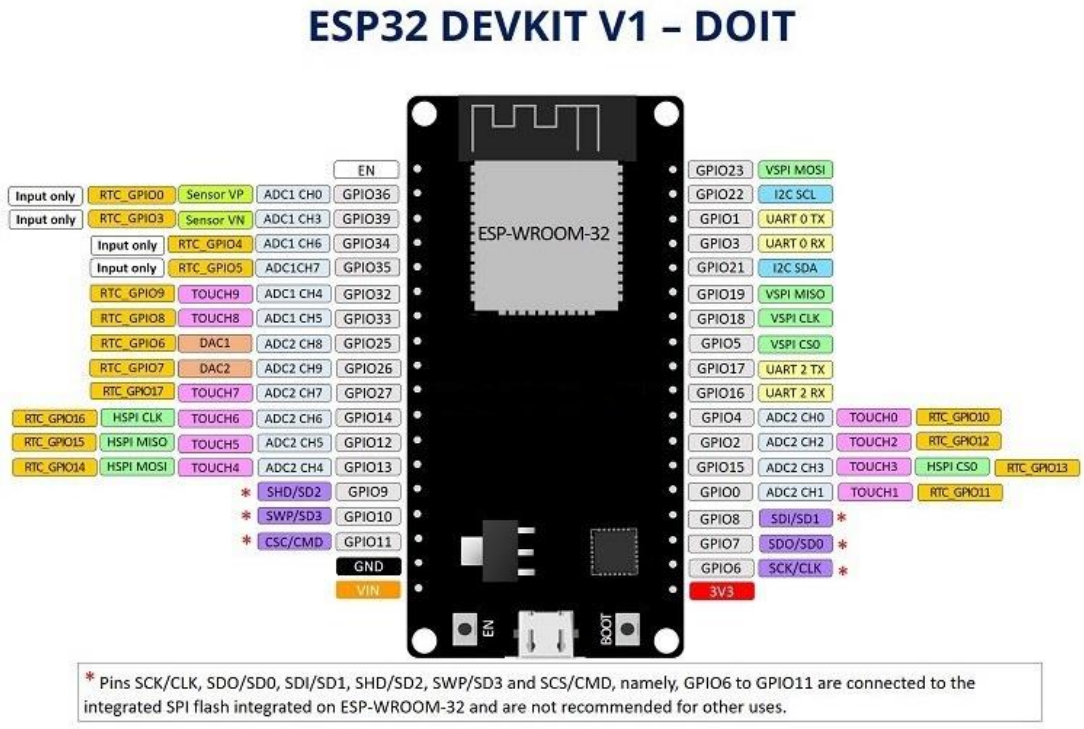

Figure 3.2 – ESP32 DoIT DevKit V1 Pin Mapping

If the pin has been configured as an output, its voltage will be set as either logic high or logic low, the following line of code is used to set the digital voltage level of an output pin:

```
digitalWrite(A, B);          // Digital pin output logic configuration
```

Where A is the output pin number and B is the output logic level either HIGH or LOW. On the other hand, if the pin has been assigned as an input, then it can be used to read analog/digital data

or as an interrupt pin. To read the analog/digital data from an input pin the following line of code is used (assume the pin has already been configured):

```
int A;                    // Assign variable A as an integer
A = digitalRead (B);      // Read input digital data from an input assigned pin
                                        OR
A = analogRead (B);       // Read input analog data from an input assigned pin
```

Where A is the stored variable and B is the input pin number. The ESP32 ADC input channels have a 12 bit resolution; therefore, variable A has ranges from 0 to 4095 (0 to $2^{12} - 1$), in which 0 corresponds to 0V and 4095 corresponds to 3.3V, respectively. To assign an input pin as an interrupt in the ESP32 microcontroller, the following lines of code are used:

```
portMUX_TYPE Mux = portMUX_INITIALIZER_UNLOCKED;  // Interrupt synchronization
…
pinMode(A, INPUT);                                // Configure pin
attachInterrupt(digitalPinToInterrupt(A), B, C);  // Attach interrupt to pin
```

Where A is the pin number, B is the interrupt service routine callback function (this function must take no parameters and return nothing), and C defines when the interrupt should be triggered (RISING, FALLING, CHANGE, and HIGH). The first line of code is declared outside the scope of the setup block. Two important notes to mention for interrupt pins, one is whenever the interrupt service routine function is called the following lines of code should be included at the start and end of interrupt routine for the ESP32 microcontroller:

```
portENTER_CRITICAL_ISR(&Mux);     // ISR port enter
…                                 // ISR code
portEXIT_CRITICAL_ISR(&Mux);      // ISR port exit
```

The second note is that a variable should be declared as ‘volatile’ whenever its value can be changed by something beyond the control of the code section in which it appears, such as the

case for interrupt service routines. Pins configuration does not strictly apply for the above conditions. For example, some output pins can be configured to provide PWM signal or DAC signal; moreover, other output pins can be configured for different communication protocols. It can be similarly said for the input pins as well. The ESP32 is provided with 3 hardware UART ports, two of which (serial 0 and serial 2) will be used to communicate with the Bluetooth and GPS modules. To setup and initialize the UART ports, the following lines are used:

```
HardwareSerial Serial_0(0);          // Define serial 0
HardwareSerial Serial_2(2);          // Define serial 2
…
Serial_0.begin(A);                   // Initialize the baudrate of serial 0
Serial_2.begin(B);                   // Initialize the baudrate of serial 2
```

Where A and B are the serial baudrates for serial 0 and serial 2, respectively. The first two lines of code are declared outside the scope of the setup block. To unlock the second core of the ESP32 microcontroller the following lines of code are used:

```
TaskHandle_t  A;                                        // Define task name
…
 xTaskCreatePinnedToCore(B, “A”, C, D, E, F, G);        // Initialize task to core
```

Where A is the task name, B is the function to implement the task, C is the stack size in words, D is the task input parameter, E is the priority of the task (higher number leads to higher priority), F is the task handle, and G is the core where the task should run. The first line of code is declared outside the scope of the setup block. It should be noted that whenever a task is created it will be executed once; therefore, in order to run a task indefinitely the code within the task should be inside an endless loop using either “for” or “while” loop. ***Keep in mind** unlocking the second core in the ESP32 microcontroller disables the ability to use the in-built antenna band with integrated Wi-Fi and dual-mode Bluetooth.

In order for the microcontroller to communicate with the MPU6050 module, an I^2C communication protocol is need. Pins 21 and 22 in the ESP32 chip corresponds to I^2C SDA

and I²C SCL. I²C (Inter-Integrated Circuit), pronounced I-squared-C, is a synchronous, multi-master, multi-slave, packet switched, single-ended, serial computer bus. I²C uses only two bi-directional open collector or open drain lines, Serial Data Line (SDA) and Serial Clock Line (SCL), pulled up with resistors. The I²C reference design has a 7-bit address space, with a rarely-used 10-bit extension. Common I²C bus speeds are the 100 Kbits per seconds for standard mode and the 400 Kbits per second for fast mode. When using the I²C communication protocol, each module has a specific address that a microcontroller can communicate with. Multiple modules can be connected on the same I²C bus each having their unique address; therefore, this procedure maintains low pin/signal count even with numerous devices on the bus. I²C incorporates ACK/NACK bits functionality for improved error handling. We can see that I²C is especially suitable when you have a complicated, diverse, or extensive network of communicating devices. UART interfaces are generally used for point-to-point connections because there is no standard way of addressing different devices or sharing pins. Figure 3.3 shows the I²C communication interface diagram.

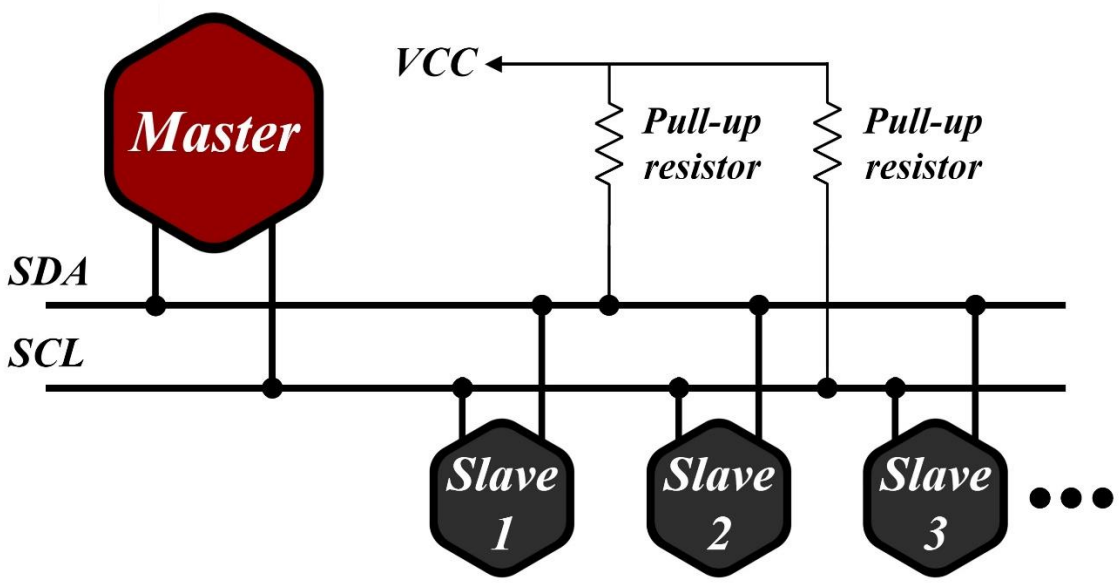

Figure 3.3 – I²C Communication Interface

The master device can be considered as the microcontroller, other slave devices can be connected in parallel with the I²C bus. Each I²C line has a pull up resistor. Typical I²C speed for the ESP32 microcontroller is 400 KHz (400 Kbps). The Arduino provides a ready-to-use I²C communication library, implementation of this library uses a 32 byte buffer; therefore, any communication should be within this limit. Exceeding bytes in a single transmission will just be dropped. To initialize and setup the MPU6050 module, the device address needs to be known and the registers where the microcontroller request to read from the module. The MPU6050 default address is 0x68; however, this address can be changed to 0x69 by applying logic high to AD0 pin of the MPU6050 module. The MPU6050 has a total of 108 registers 18 of which will be used during this project. For example to activate the module a value of 0x00 is sent to PWR_MGMT_1 register, this can be done using the following lines of code (assume device address is 0x68):

```
#include "Wire.h"                       // Include I²C library
…
Wire.begin();                           // Begin I²C communication
Wire.beginTransmission(0x68);           // Start communicating with MPU6050
Wire.write(0x6B);                       // Request a write to the PWR_MGMT_1 register
Wire.write(0x00);                       // Set the register bits as 0x00 to activate MPU6050
Wire.endTransmission(true);             // End I²C communication
```

The same procedure can be done when writing to other registers. The settings applied to the MPU6050 module are as follows:

- Configure gyroscope to output data at 500dps full-scale range by sending 0x08 value to GYRO_CONFIG register.
- Configure accelerometer to output data at +/– 8g full-scale range by sending 0x10 value to ACCEL_CONFIG register.
- Set Digital Low Pass Filter to ~43Hz by sending 0x03 value to CONFIG register

These settings along with activating the MPU6050 are called using the user-defined setupIMU() function. To read the gyro, accelerometer, and temperature data (registers 0x3B to 0x48) from the MPU6050 module the following lines of code are used:

```
#include "Wire.h"                    // Include I²C library
…
Wire.begin();                        // Begin I²C communication
Wire.beginTransmission(0x68);        // Start communicating with MPU6050
Wire.write(0x3B);                    // Start reading from register 0x3B (ACCEL_XOUT_H)
Wire.endTransmission(false);         // Keep I²C communication running
Wire.requestFrom(0x68,14,true);      // Request a total of 14 registers (0x3B to 0x48) and end TX
AcX=Wire.read()<<8|Wire.read();  // 0x3B (ACCEL_XOUT_H) & 0x3C (ACCEL_XOUT_L)
AcY=Wire.read()<<8|Wire.read();  // 0x3D (ACCEL_YOUT_H) & 0x3E (ACCEL_YOUT_L)
AcZ=Wire.read()<<8|Wire.read();  // 0x3F (ACCEL_ZOUT_H) & 0x40 (ACCEL_ZOUT_L)
Tmp=Wire.read()<<8|Wire.read();  // 0x41 (TEMP_OUT_H) & 0x42 (TEMP_OUT_L)
GyX=Wire.read()<<8|Wire.read();  // 0x43 (GYRO_XOUT_H) & 0x44 (GYRO_XOUT_L)
GyY=Wire.read()<<8|Wire.read();  // 0x45 (GYRO_YOUT_H) & 0x46 (GYRO_YOUT_L)
GyZ=Wire.read()<<8|Wire.read();  // 0x47 (GYRO_ZOUT_H) & 0x48 (GYRO_ZOUT_L)
```

The Wire.requestFrom method in the above code is used to request a total of 14 bytes from the MPU6050 (address 0x68), the Boolean true will send a stop message after the request is complete and releases the bus. If false is replaced instead of true then it will continually send a restart after the request, keeping the connection active. The Wire.requestFrom method is usually called once the user specifies the beginning register by using Wire.write and applying Boolean false for Wire.endTransmission to keep the I²C communication running. Notice that for each register in the MPU6050 module the data is stored in a 16-bit format and since each register holds only 8-bits, then the 16-bit binary data is stored in the upper and lower registers respectively. Therefore, in order to merge both registers the following line was applied by using shifting technique:

```
Data=Wire.read()<<8|Wire.read();    // Read and merge both registers to form a 16-bit format
```

The developer can decide the amount of data shifting from the line "<<N|" where N is an integer number referring to the amount of bits to be shifted. From our code it is clear that the data variable should be a 16-bit format or above. The readIMU() function will be called to read the data within the MPU6050 module.

For safety reasons, a register check is performed along the setup procedure to ensure that the module is connected correctly and functional. The following lines of code are used to perform register check at address 0x1B for the MPU6050 module:

```
#include "Wire.h"                          // Include I²C library
…
Wire.begin();                              // Begin I²C communication
Wire.beginTransmission(0x68);              // Start communication with MPU6050
Wire.write(0x1B);                          // Start reading at register 0x1B
Wire.endTransmission();                    // End I²C communication
Wire.requestFrom(0x68, 1);                 // Request 1 byte from the gyroscope
while(Wire.available() < 1);               // Wait until the byte is received
if(Wire.read() != 0x08){                   // Check if the value is 0x08
        while(true) {
                // If true then loop forever while blinking the LED to indicate setup error
                digitalWrite(SM_LED, ! digitalRead(SM_LED));
                delay(100);
        }
}
```

Where SM_LED is the surface mount LED. It can be seen from the above code that if the condition where the module is malfunctioned then the LED mounted on the ESP32 microcontroller will blink at a delay about 100 milliseconds to indicate an error message during setup procedure. However, any other mean of error indication can be implemented to warn the user of setup failure. These lines of code are incorporated within the setupIMU() function.

To finish setting up the MPU6050 module a calibration and adjustment procedure should take place in order to remove any offset errors during the reading of incoming data. A measurement error is the difference between a measured value of quantity and its true value. Naturally, any electronic sensor module requires calibration due to the none-ideal behavior of the device, construction of the device, and frequent use of the device in operation. Calibration ensures reliability and consistent readings of such device. The technique used to calibrate the module is by averaging the data while placing the drone in the spirit level. The microcontroller will average a total of N readings and subtract that error from the actual data. N is any integer greater than zero. The higher the value of N the greater the accuracy of calibration. The following lines of code are used to calibrate the gyroscope data automatically by the microcontroller.

```
#include "Wire.h"                // Include I²C library
…
for(calMPU = 0; calMPU <= 2000 ; calMPU++){
        if(calMPU % 15 == 0) {
                // Change the surface mount LED status to indicate calibration.
                digitalWrite(SM_LED, ! digitalRead(SM_LED));
        }
        readIMU();               // Read the raw data from the MPU6050
        GyX_Cal += GyX;          // Add the gyro x-axis offset to the gyro-x calibrated variable
        GyY_Cal += GyY;          // Add the gyro y-axis offset to the gyro-y calibrated variable
        GyZ_Cal += GyZ;          // Add the gyro z-axis offset to the gyro-z calibrated variable
        delay(3);                // Delay to simulate the 250Hz program loop
}
// Divide the gyro-x/y/z calibrated variable by total readings to get the average offset
GyX_Cal /= 2000;                 // Subtract this value from the actual reading when computing
GyY_Cal /= 2000;                 // Subtract this value from the actual readings when computing
GyZ_Cal /= 2000;                 // Subtract this value from the actual readings when computing
// Check if IMU calibration is accomplished or not
if(GyX_Cal == 0 || GyY_Cal == 0 || GyZ_Cal == 0) {
                                                        …
```

```
...
        while(true) {
            // If true then loop forever while blinking the LED to indicate setup error
            digitalWrite(SM_LED, ! digitalRead(SM_LED));
            delay(100);
        }
}
```

In addition to checking the safety of the module using register check process, another method is performed to see whether the average offset error is zero or not. This procedure is done due to the fact that sometimes the MPU6050 module can communicate with the master device; however, the data is read as zero due to software/hardware errors. The probability of any of the data calibrated to be zero is extremely low and can be ignored. In fact even if the calibration process yields to zero offset errors then it is safer for the personal to not fly the drone than hovering it with the assumption of a correct calibration. After all, it is better to be safe than sorry. The calibration code is called using the caliIMU() function.

To control the speed of the motors a varying PWM signal must be applied to the ESC from the microcontroller. The ESP32 can provide up to 16 PWM channels using hardware timers. These timers can be modified to allow timing precision up to 16-bits. Since there are four motors to control and four ESCs to signal them, then a total of four channels from the 16 PWM channels are used (channel 1 – 4, channel 0 is reserved for surface mount LED). The timing precision will be set to 16 bit and the PWM signal frequency will be set to 250Hz; therefore, the period at 100% duty is 4 milliseconds. As a standard, the ESCs will take minimum and maximum PWM of 1 to 2 milliseconds, respectively. For 16-bit precision timing, the maximum duty output value is $(2^{12} - 1)$ or 65535. Similarly, at 250Hz the 1 and 2 milliseconds pulses correspond to 25% and 50% duty cycles which take values of 16384 and 32768. It should be noted that the ESCs must be calibrated first to take minimum and maximum of 1 and 2 milliseconds pulses. This can be done by applying maximum pulse at **power on** and wait for a series of beep sounds. Once the sounds are off, apply the minimum pulse and wait for the confirmation sounds in the form of

beeps again. This process can be programmed automatically by the microcontroller to handle ESCs calibration.
To setup one of the pins of the ESP32 to output PWM signal, the following code is used:

```
ledcSetup(A, B, C);                    // Setup the PWM channel
ledcAttachPin(D, A);                   // Attach the PWM channel to a corresponding pin
```

Where A is the PWM channel number (starts from 0 to 15), B is the PWM frequency, C is the timer precision, and D is the pin where the PWM channel will be attached to. The code can be repeated for all four pins. Once the channel is attached to the pin, the corresponding pin will output PWM signal as long as the microcontroller is powered on. To modify the duty cycle the following line is used:

```
ledcWrite(A, B);                       // Change duty cycle of PWM channel
```

Where A is the channel number (0 to 15) and B is the duty in numerical format. At power on together the ESCs and motors will begin to output a constant, repetitive beeping sound. To disable beeping, a pulse less than the minimum adjusted pulse is applied to all four ESCs typically a value of 16000 is sent. This is done for safety and precautions.

A timer interrupt is used to send sensors data from the Bluetooth module to the smartphone at a constant time interval; therefore, constant transmission of sensor's data becomes independent from the delay time of the main running code. This can be also applied to GPS data transmission. The ESP32 has four hardware timer interrupts that can be used to our advantage. To setup timer interrupt the following lines of code are used:

```
portMUX_TYPE Mux = portMUX_INITIALIZER_UNLOCKED; // Interrupt synchronization
hw_timer_t * A = NULL;                 // Define timer
…
A = timerBegin(B, C, D);               // Begin timer
                                       …
```

```
                                                    …
timerAttachInterrupt(A, &E, F);          // Attach a callback for the timer interrupt
timerAlarmWrite(A, G, H);                // Setup timer interrupt
timerAlarmEnable(A);                     // Enable timer interrupt
```

Where A is the timer name, B is the hardware timer-interrupt number (0 - 3), C is the timer base frequency (for ESP32 DoIT DevKit V1 the default value is 80 for 80MHz), D is a Boolean value indicating a counting up (true) or down (false) timer interrupt, E is the timer ISR callback function (the '&' symbol must be included together with the name of the function), F is a Boolean value indicating the type of interrupt as edge (true) or level (false), G is the interrupt time (for ESP32 DoIT DevKit V1 the max value is 1000,000us for 80MHz), H is a Boolean value indicating a the periodical interrupt generation (true) or not (false). The first two lines of code are declared outside the scope of the setup block.

To jump into the main code the user is required to place the controller stick in the position shown in Figure 3.4

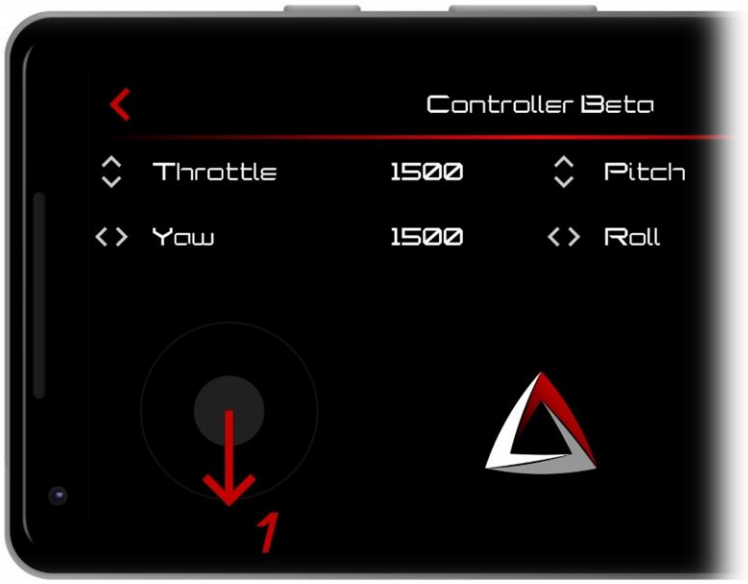

Figure 3.4 – Ending Setup Process by Moving the Throttle Stick Downward

As it can be seen from Figure 3.4, in order to break from the setup code and jump into the main code the user is required to turn the throttle analog stick towards the minimum pulse. The average between the minimum and maximum pulse for 1-2 milliseconds is 1.5 milliseconds. This middle value pulse will serve as the base pulse for roll, pitch, and yaw angles, any value above or below the average pulse will cause the quadcopter to rotate along these angles. It should be noted that once the function xTaskCreatePinnedToCore() is called, the core related to that task will immediately begin operating. For the quadcopter, core 0 will handle any communication tasks; therefore, once the task is pinned to core 0 during setup then the core will begin to handle received data from the user. This procedure allows the user to transmit data to the drone during the setup waiting time. In conclusion, the first step to begin flying the quadcopter is to place the throttle at 1 millisecond pulse. This can be done using the following lines of code:

```
while(Received_Throttle < 990 || Received_Throttle > 1020){
        // Blink LED to indicate setup on hold
        digitalWrite(SM_LED, !digitalRead(SM_LED));
        delay(500);
        readIMU();                    // Read IMU data on hold
}
Start = 0;                            // Set start flag to 0
```

Where ‘Start’ variable is the flag used to indicate whether the motors should start or stop. ‘Received_Throttle’ variable is the received throttle pulse. *Note at setup the Received_Throttle variable is 1500 as a default. A dead band of 30 microseconds is applied for throttle readings to get a better result. During the waiting process the microcontroller reads the MPU6050 gyroscope and accelerometer data. It should be noted that the above code should be placed after declaring a task to core 0 using xTaskCreatePinnedToCore() function. The developer can use any method to start the quadcopter’s motors and begin flying; however, using the method introduced in this project can be easily implemented to both RF controller and Bluetooth smartphone controller. This method is a set of three methods applied in order to begin the motors, the other two methods are introduced in the main code section that follow up from this method.

One last step before heading to the main code is to set the LED pin to HIGH to indicate the completion of setup code and initialize a variable to store the time in microseconds.

```
digitalWrite(SM_LED, HIGH);                    // Turn on LED again (Setup is completed)
Loop_Timer = micros();                         // Reset the loop timer
```

The 'Loop_Timer' will be used in the main code to assimilate a 250Hz program. The micros () function returns the number of microseconds since the Arduino board began running the current program. This is used to ensure the refresh rate of the ESCs every 4 milliseconds.

Core 1 Main Code

Core 1 of the microcontroller is responsible to handle the process of stability for the entire quadcopter system. Using the quadcopter dynamics and control theories introduced in **Chapter 2** (**Section 2.1** and **2.2**) will help the system to maintain and allow itself to be capable of self-sustained flight. Before applying the equations, we will continue to explain how to start the motors of the quadcopter. After the first step shown in Figure 3.4 where the user breaks from the setup code and jumps into the main code, the second step is to move the yaw stick towards the minimum pulse as seen in Figure 3.5.

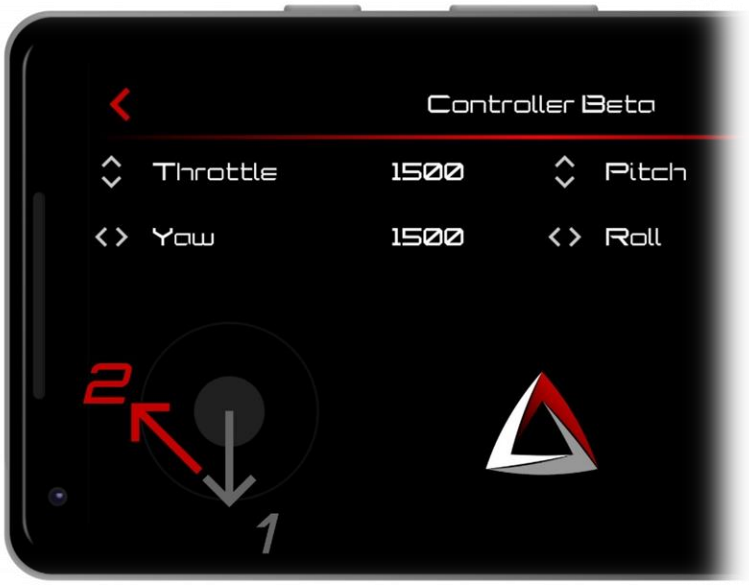

Figure 3.5 – First Step to Startup Motors

This can be read by the microcontroller to change the ‘Start’ flag from 0 to 1. The following code shows how this is done:

```
if(Received_Throttle < 1200 && Received_Yaw < 1200) Start = 1;     // Set start flag to 1
```

Where ‘Received_Yaw’ variable is the received yaw pulse. The third and final step in starting the motors is to return the yaw stick to the original position that is to return the yaw stick to the average pulse of 1.5 milliseconds. This can be seen in Figure 3.6

Figure 3.6 – Second Step to Startup Motors

This procedure changes the ‘Start’ flag from 1 to 2. The code corresponding to this step is shown below:

```
if(Start == 1 && Received_Throttle < 1050 && Received_Yaw > 1450){
        Start = 2;                  // Set start flag to 2
        // Set the gyro pitch and roll angle equal to the accelerometer pitch and roll angle …
        // … when the quadcopter is started.
        Angle_Pitch = Angle_Pitch_Acc;
                                                    …
```

```
…
        Angle_Roll = Angle_Roll_Acc;
        // Reset the PID controllers for a bump-less start.
        PID_I_Mem_Roll = 0;
        PID_Last_Roll_D_Error = 0;
        PID_I_Mem_Pitch = 0;
        PID_Last_Pitch_D_Error = 0;
        PID_I_Mem_Yaw = 0;
        PID_Last_Yaw_D_Error = 0;
}
```

In addition to changing the ‘Start’ flag, the PID controller variables are reset to zero and the gyroscope pitch and roll angles are set to accelerometer pitch and roll angles, respectively. With the ‘Start ’flag set to 2 the user can begin to fly the quadcopter and the microcontroller begins to stabilize it. To stop the motors from spinning, move the throttle stick to the lower position and send a stop flag to the microcontroller to confirm the process. Figure 3.7 shows the steps for stopping the motors from spinning.

Figure 3.7 –Steps to Stop Motors from Spinning

The following code shows how this microcontroller stops the motors:

```
// Reset start flag to zero
if(Start == 2 && Received_Throttle < 1050 && Motors_Flag == true) Start = 0;
```

Where ‘Motors_Flag’ is the flag that indicates whether to start or stop the motors. The ‘Motors_Flag’ is set to true once the button is pushed and immediately set to false once the user releases his/her figure from the button. This ensures no error occurs during the second attempt to start and stop the motors. Once the user stops the motors he/she can start up the motors again by following the steps from Figure 3.4 to 3.5. When using an RF communication channel where data are only represented in pulses then the process of stopping the motors can be done by moving the throttle and yaw stick to minimum pulse. For diversity either include two code statements for the RF and Bluetooth channels or using the RF communication technique to Bluetooth as well to stop the motors. However, it is proven safer to use the ‘Motors_Flag’ instead.

Now that the steps of starting and stopping the quadcopter motors have been covered, our next discussion will focus on how to convert the raw data of the MPU6050 into roll, pitch, and yaw angles using **sensor fusion** technique. Begin by subtracting the gyroscope raw data from the offset calibrated value and store the result in a variable. This is shown as follows:

```
Gyro_Roll = GyX - GyX_Cal;              // Subtract the actual data from the calibrated offset
Gyro_Pitch = GyY - GyY_Cal;             // Subtract the actual data from the calibrated offset
Gyro_Yaw = GyZ - GyZ_Cal;               // Subtract the actual data from the calibrated offset
```

The MPU6050 gyroscope was configured to output 65.5 per axis at angular velocity of 1°/s. Given the refresh rate of the ESCs at 250Hz; to convert the gyroscope raw data into degrees per second divide the data by 250/65.5 or multiply it by a factor of 0.0000611 and compute the numerical integration to a corresponding new variable as shown:

```
Angle_Pitch += Gyro_Pitch * 0.0000611;          // Calculate the traveled pitch angle
```

```
Angle_Roll += Gyro_Roll * 0.0000611;                    // Calculate the traveled roll angle
```

*Notice that only the pitch and roll angles have been calculated since they are the main angles used for stability. It may seem that these angles are sufficient; however, these angles suffer from yaw drifting problem. To overcome this obstacle the yaw axis needs to be coupled with the pitch and roll axis, in other words, if the IMU has yawed transfer the roll angle to the pitch angle and the pitch angle to the roll angle. The mathematical formula can be derived by noticing that the change in the pitch angle verse the rotation in the yaw angle resembles the sine function; therefore, multiply the transferred angles with the sine of yaw axis. The sine function in Arduino takes values in radians so we multiply the angles with a factor of $\pi/180$ or 0.000001066. Below is the code that corrects the yaw drift.

```
// If the IMU has yawed transfer the roll angle to the pitch angle and vice versa.
Angle_Pitch -= Angle_Roll * sin(Gyro_Yaw * 0.000001066);
Angle_Roll += Angle_Pitch * sin(Gyro_Yaw * 0.000001066);
```

One of the main problems the gyroscope based IMU module encounters is the drifting of the angles during slow moving conditions and the moment the module is started at an angled surface since the IMU has no reference to what level is. These two problems can be solved with the use of an accelerometer. To understand how an accelerometer sensor works, a two axis model is shown in Figure 3.8 where an object is attached by 4 strings.

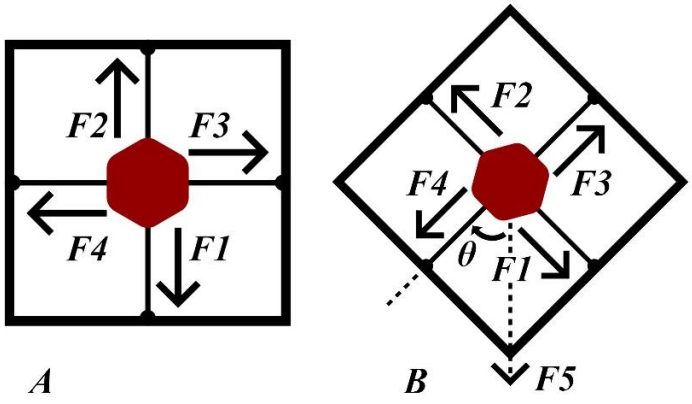

Figure 3.8 –Two Axis Accelerometer Sensor Model

From the figure, model A is place perpendicular to the surface where F1 to F4 are the forces acting on each string, respectively. Assume four force measuring sensors are placed at each string, then force F1 will experience a gravitational force at 9.8 m/s^2; therefore, the sensor at F2 will have a maximum reading whereas sensors at F3 and F4 will have minimum readings. The MPU6050 was configured to output 4096 when an acceleration of 9.8 m/s^2 is applied to the sensor. Using this information the sensor at F2 from model A will output a value of 4096. When the model is tilted 45° along its axis as can be seen from model B, then F4 and F1 will both experience a gravitational force; therefore, sensors attached at F3 and F2 will have an equal readings of 2920. From the figure of model B, the acceleration total vector is represented as F5. Using the Pythagoras theorem, F5 can be obtained simply as follows:

$$F5 = \sqrt{2920^2 + 2920^2} = 4129$$

Using the value obtained from F5, the angle θ between F1 and F4 can be calculated using the inverse cosine as follows:

$$\theta = \cos^{-1}\frac{2920}{4129} \cong 45°$$

For a three axis model, the acceleration total vector is given as:

$$ATV = \sqrt{(Acc.X)^2 + (Acc.Y)^2 + (Acc.Z)^2}$$

Using the acceleration total vector formula, the roll and pitch angles from the accelerometer can be measured by dividing the acceleration along x and y axis by the acceleration total vector and taking the inverse sine of the result. These steps for computing the roll and pitch angles are represented in the following code. Note that the inverse sine and cosine functions in Arduino IDE output the results in radians. To convert radians to degrees multiply the results by 57.296.

```
// Calculate the pitch and roll angles form the total accelerometer vector.
AccTotalVector = sqrt((AcX*AcX) + (AcY*AcY) + (AcZ*AcZ));
if(abs(AcY) < AccTotalVector){
        Angle_Pitch_Acc = asin((float)AcY/AccTotalVector)* 57.296;
}
if(abs(AcX) < AccTotalVector){
        Angle_Roll_Acc = asin((float)AcX/AccTotalVector)* -57.296;
}
```

A conditional 'if' statement is introduced to ensure the inverse sine does not output a NaN when the acceleration total vector is less than the acceleration axis. One final step to obtain the roll and pitch angles from the accelerometer is to calibrate the results of **Angle_Pitch_Acc** and **Angle_Roll_Acc**. Unlike gyroscope calibration, accelerometer angles calibration requires the user to manually calibrate and adjust the angles by inspection. This procedure is done only once by placing the MPU6050 at spirit level and note the values of **Angle_Pitch_Acc** and **Angle_Roll_Acc**. Figure 3.9 shows the non-calibrated accelerometer angles with respect to the calibrated ones.

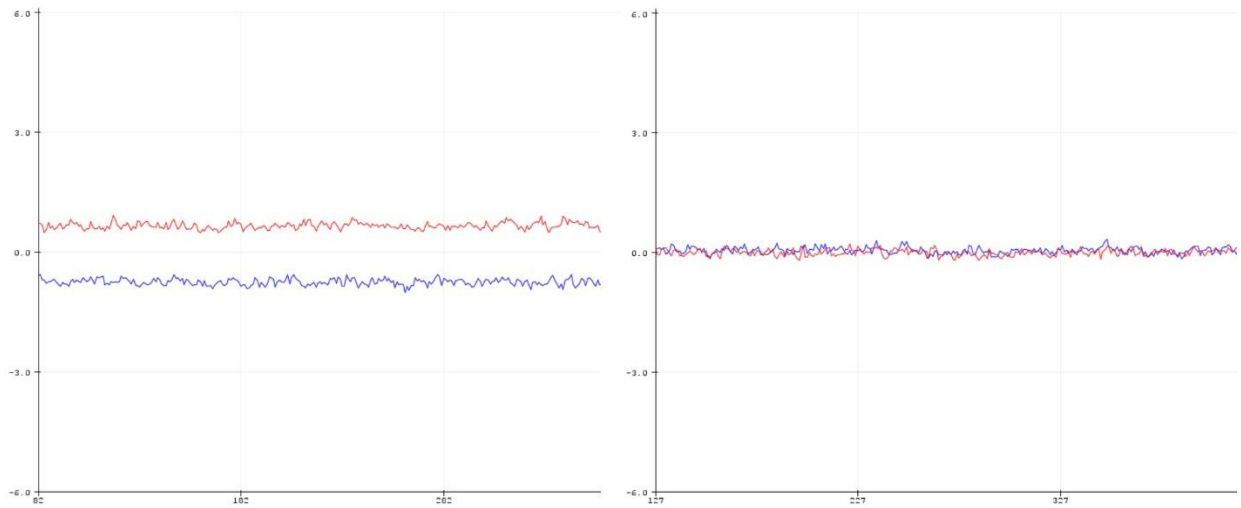

Figure 3.9 –Accelerometer Angles with and without Calibration

From Figure 3.8 the angles have been calibrated by adding an offset values of +0.8 and -0.6 for the pitch and roll angles, respectively. This is done using the following two lines of code:

```
Angle_Pitch_Acc += 0.8;          // Accelerometer calibration value for pitch.
Angle_Roll_Acc -= 0.6;           // Accelerometer calibration value for roll.
```

It is not sufficient to use the accelerometer angles alone since they suffer drastically from vibrations due to motors rendering them completely useless. To obtain the desired pitch and roll angles a technique of combining both accelerometer and gyroscope data known as **sensor fusion** is used. Combining of sensory data or data derived from disparate sources results in information with less uncertainty than would be possible when these sources were used individually. Different sensor fusion methods can be utilized here; however, since timing is important when updating the ESCs pulse a robust and fast method should be considered. The **complementary filter** is one of the sensor fusion methods that is sufficient enough to provide less delay for the microcontroller to compute alongside of it to being a powerful and solid method. A complementary filter applies correction by using a low-pass and a high-pass filter to the incoming data. Data with superimposed noise is directed towards the low-pass filter, whereas data with drifting problem is directed towards the high-pass filter. Figure 3.10 shows a block diagram of the complementary filter using the data from the gyroscope and accelerometer.

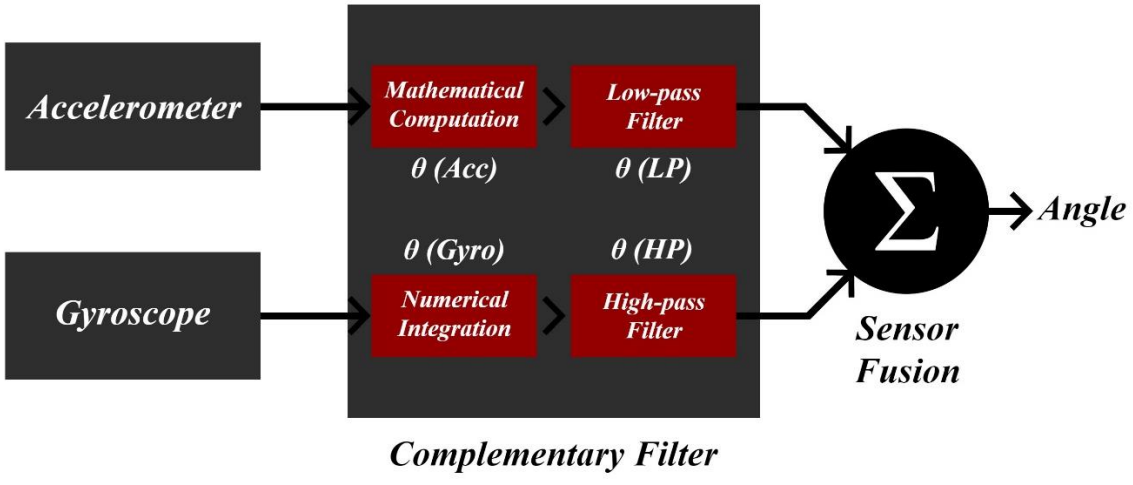

Figure 3.10 –Complementary Filter Block Diagram for Accelerometer and Gyroscope

For the quadcopter system, the gyroscope angles suffers from drifting complications over time due to numerical integration of the data; on the other hand, the accelerometer angles suffers from vibrations and jitteriness of the system resulting in noisy data. Therefore, the accelerometer angles are applied to the low-pass filter and the gyroscope angles are applied to the high-pass filter as can be seen in Figure 3.10. The mathematical form of this complementary filter can be written as:

$$Angle = (\alpha_{cf}) * (Angle\ Gyro) + (1 - \alpha_{cf}) * (Angle\ Acc)$$

Where α_{cf} is the filter coefficient of the complementary filter. This equation can be used for both pitch and roll angles using software code as follows:

```
// Apply complementary filter when the motors have started.
if (Start == 2) {
        // Complementary filter
        Angle_Pitch = Angle_Pitch * 0.9996 + Angle_Pitch_Acc * 0.0004;
        Angle_Roll = Angle_Roll * 0.9996 + Angle_Roll_Acc * 0.0004;
}
```

From inspection, the value of α_{cf} is chosen to be 0.9996. Notice that the complementary filter is only applied when the motors have started (i.e. when the 'Start' flag is 2). This is done since at the starting condition of the motors the accelerometer angles are passed to the gyroscope angles.

Now that the IMU angles are available the next step is to apply the PID controller algorithms. Before calling the PID function, the set points for pitch, roll, and yaw angles must be set first. The angles obtained must be scaled to the configured pulse values that are between 1000 and 2000 for angle correction. This is done using the following two lines of code:

```
Pitch_Level_Adjust = Angle_Pitch * 15;          // Calculate the pitch angle correction
Roll_Level_Adjust = Angle_Roll * 15;            // Calculate the roll angle correction
```

The PID set points are dependent on the received angles from the user. A zero set point corresponds to a 1500us pulse (1.5 milliseconds pulse), any value above or below a 1500us pulse will cause the quadcopter to maneuver itself by subtracting the received value from 1500us pules. A dead-band of 16us is used for better accuracy and results. The set points are then subtracted from the angles correction. Once the set points have been calculated the results is then divided by 3 to get the angles in degrees. The factor 3 is determined from the max pitch, roll, and yaw rate is which is approximately 164 degrees per second ((500-8)/3 = 164d/s). This process can be interpreted in the following code for the roll angle:

```
// Calculate the set point for roll angle
PID_Roll_Setpoint = 0;
// A dead band of 16us is applied for better results.
if(Received_Roll > 1508) PID_Roll_Setpoint = Received_Roll - 1508;
else if(Received_Roll < 1492) PID_Roll_Setpoint = Received_Roll - 1492;
PID_Roll_Setpoint -= Roll_Level_Adjust;    // Subtract the set point from the angle correction.
PID_Roll_Setpoint /= 3.0;
```

The same procedure is applied for both pitch and yaw set points, however, since the yaw angle has not been calculated an angle correction is not applied. The yaw axis is still used to yaw the quadcopter, but mainly the pitch and roll angles play the role in stability. After computing the set points, the PID algorithm is then applied. The PID controller introduced in **Chapter 2**, **Section 2.2** is used for continuous time signals; however, digital electronic systems are discrete in time. To represent an approximate PID controller in discrete format the following equation is introduced.

$$u = K_p e + K_i \sum e_n + K_d \frac{e_n - e_{n-1}}{\Delta t}$$

In discrete mathematics the role of derivatives is played by finite differences and the role of integrals by partial sums of series. The accuracy of this approximate model depends on the

sampling period of the data. The lower the sampling period the higher the accuracy. Here the sampling speed is 4ms. The discrete PID block of the quadcopter is shown in Figure 3.11.

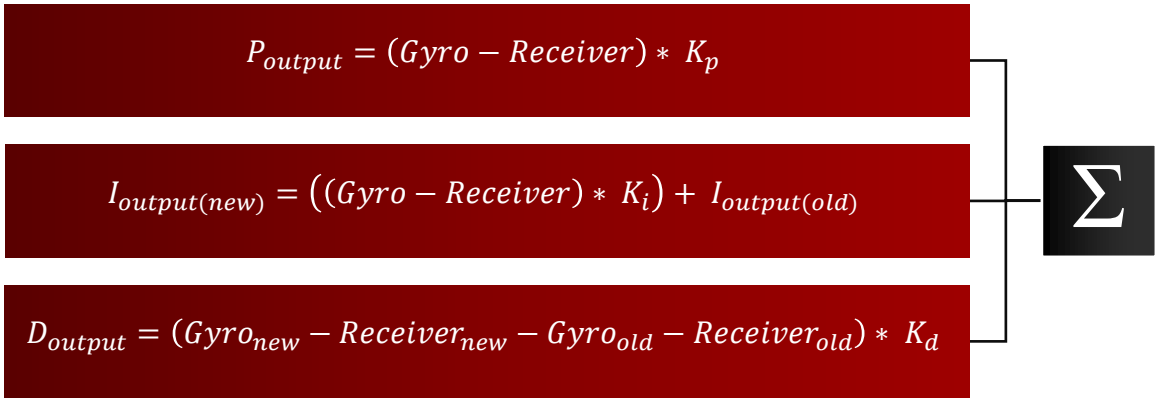

Figure 3.11 – Quadcopter PID Controller Block Diagram

Where the error is defined as:

$$e = Gyro - Receiver$$

In other words, the error is the input gyroscope angle subtracted from the calculated set point corresponding to that angle. For convenience a complementary filter is applied once more for the gyroscope angles before they are passed into the PID controller. This can be seen in the following code where the filter coefficient is chosen to be 0.7:

```
Gyro_Roll_Input = (Gyro_Roll_Input * 0.7) + ((Gyro_Roll / 65.5) * 0.3);
Gyro_Pitch_Input = (Gyro_Pitch_Input * 0.7) + ((Gyro_Pitch / 65.5) * 0.3);
Gyro_Yaw_Input = (Gyro_Yaw_Input * 0.7) + ((Gyro_Yaw / 65.5) * 0.3);
```

The gyroscope angles are divided by 65.5 to convert them into degrees per second as per the MPU6050 datasheet. Together, the gyroscope filtered angles and the angles used to calculate set

points are used to compute the error for the PID controller. For the roll angle The PID block diagram shown in Figure 3.11 can be represented in the following code using the PID_Control() user defined function:

```
// PID roll calculations
PID_Error_Temp = Gyro_Roll_Input - PID_Roll_Setpoint;
PID_I_Mem_Roll += PID_I_Gain_Roll * PID_Error_Temp;
if(PID_I_Mem_Roll > PID_Max_Roll) PID_I_Mem_Roll = PID_Max_Roll;
else if(PID_I_Mem_Roll < PID_Max_Roll * -1) PID_I_Mem_Roll = PID_Max_Roll * -1;
PID_Output_Roll = PID_P_Gain_Roll * PID_Error_Temp + PID_I_Mem_Roll + …
… PID_D_Gain_Roll * (PID_Error_Temp - PID_Last_Roll_D_Error);
if(PID_Output_Roll > PID_Max_Roll) PID_Output_Roll = PID_Max_Roll;
else if(PID_Output_Roll < PID_Max_Roll * -1) PID_Output_Roll = PID_Max_Roll * -1;
PID_Last_Roll_D_Error = PID_Error_Temp;
```

The ‘PID_Max_Roll’ variable holds the maximum and minimum allowable PID roll output value. This ensures that the PID controller does not output values beyond the safe levels. ‘PID_P_Gain_Roll’, ‘PID_I_Gain_Roll’, ‘PID_D_Gain_Roll’ are the PID roll gains, respectively. For the pitch and yaw PID controllers the procedure is the same with the exception of the maximum and minimum allowable PID output value and the PID gains values.

Once the PID algorithm is applied, the outputs of the PID controller are then used to compute the PWM signal of the ESCs. A throttle signal will serve as the base signal for the ESCs. The signal will be manually controlled by the user to increase or decrease the uplifting force of the quadcopter. A safety margin is used to ensure that the microcontroller does not output a PWM signal greater than 2 milliseconds; additionally, for PWM signal less than 1.1 milliseconds the microcontroller outputs minimally a 1.1 milliseconds pulse. The 1 millisecond pulse is reserved for stopping the motors of the quadcopter as it was seen Figure 3.7. It should be noted that each ESC of the quadcopter should receive a different PWM signal depending on the placement of the motors. Figure 3.12 shows the corresponding PWM output signal calculation for each motor for the ‘x’ formation quadcopter.

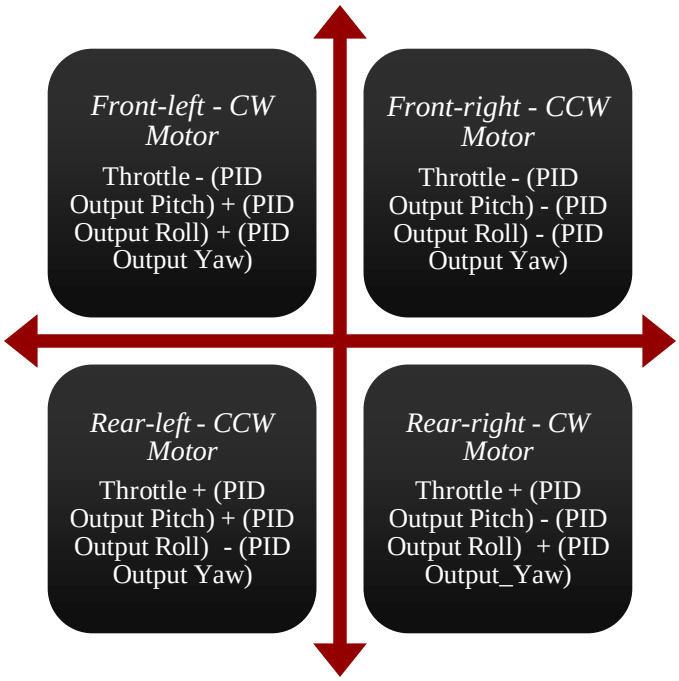

Figure 3.12 – PWM Signal for the X-Formation Quadcopter

Using these information, the code to calculate the PWM signals for the ESCs is shown below:

```
Throttle = Received_Throttle;                    // Make throttle signal as a base signal.
if(Start == 2){                                  // The motors are started.
        ESC_1 = Throttle - PID_Output_Pitch - PID_Output_Roll;
        ESC_2 = Throttle + PID_Output_Pitch - PID_Output_Roll;
        ESC_3 = Throttle + PID_Output_Pitch + PID_Output_Roll;
        ESC_4 = Throttle - PID_Output_Pitch + PID_Output_Roll;
        if(ESC_1 < 1100) ESC_1 = 1100;           // Keep the motors running.
        if(ESC_2 < 1100) ESC_2 = 1100;           // Keep the motors running.
        if(ESC_3 < 1100) ESC_3 = 1100;           // Keep the motors running.
        if(ESC_4 < 1100) ESC_4 = 1100;           // Keep the motors running.
        if(ESC_1 > 2000) ESC_1 = 2000;           // Limit the ESC_1 pulse to 2000us.
        if(ESC_2 > 2000) ESC_2 = 2000;           // Limit the ESC_2 pulse to 2000us.
                                                 …
```

```
                                                    …
        if(ESC_3 > 2000) ESC_3 = 2000;              // Limit the ESC_3 pulse to 2000us.
        if(ESC_4 > 2000) ESC_4 = 2000;              // Limit the ESC_4 pulse to 2000us.
}
else {
        ESC_1 = 1000;               // If start is not 2 keep a 1000us pulse for ESC_1.
        ESC_2 = 1000;               // If start is not 2 keep a 1000us pulse for ESC_2.
        ESC_3 = 1000;               // If start is not 2 keep a 1000us pulse for ESC_3.
        ESC_4 = 1000;               // If start is not 2 keep a 1000us pulse for ESC_4.
}
```

Once the ESCs pulses have been calculated the microcontroller awaits until a 4 milliseconds time have passed to refresh the ESCs pulse. The process of computing the IMU angles, applying PID algorithm, calculating the pulses and outputting the pulses to the four ESCs are then repeated. *If, however, the process takes longer time , more than 4 milliseconds, then the user needs to re-evaluate the code once again to insure that the all the processing done within the microcontroller does not reach 4 milliseconds or above. Below is the final part of the code where the timing process is handled and the PWM signals are applied to the ESCs.

```
// Warns the developer if the code's processing time is greater than 4ms
if(micros() - Loop_Timer > 4000) {
        …                                   // Code to handle warning.
}
while(micros() - Loop_Timer < 4000);    // Wait until 4000us are passed - Refresh rate of ESC.
Loop_Timer = micros();                  // Set the timer for the next loop.
  // Write PWM to ESCs
  PWM_Write(PWM_CHANNEL_2, ESC_1);
  PWM_Write(PWM_CHANNEL_1, ESC_2);
  PWM_Write(PWM_CHANNEL_3, ESC_3);
  PWM_Write(PWM_CHANNEL_4, ESC_4);
```

Where PWM_Write() is the user defined function that converts the PWM signal from 1000 - 2000 pulse to 16384 – 32768 using linear interpolation as follows:

ledcWrite(channel, 16384 + ((duty-1000) * 16.384));

Core 0 Main Code

Core 0 of the microcontroller is responsible to handle the process of data communication between the smartphone and the microcontroller and/or an external control device and the microcontroller. Additionally, core 0 handles any miscellaneous process that is not responsible in the stability of the system such as communicating with a GPS module, reading data from external sensors, and controlling external hardware units such as an LED flash or a servo motor to title a mounted camera. As discussed previously from the setup part of the Arduino code, a task will be attached to core 0 and in doing so the task will be assigned with an infinite conditional loop statement in order to keep the task running.

Before discussing how Bluetooth communication code is executed, a basic understanding of the protocol used to handle incoming data is introduced first. The data received from the Bluetooth module will have the following format:

$$\alpha XXXX\beta$$

Where α is a character that distinguishes the form of the data type, β marks the end of the data sequence, and the data between α and β is the actual value. The value received can be either a float or an integer number; therefore, for a float data type β will be replaced with a semicolon and for an integer data type β will be replaced with a comma.

To read the received data from the Bluetooth module, first check if the Bluetooth serial buffer is full. Once the buffer is full compare the data from the buffer with a look-up table. For example assume the incoming integer data is in the form of ‘A1220,’, the first letter points to the category of the data and will be stored in a variable to be used later, the next four numbers are the actual data and are stored in a string format, and the last letter is a comma which marks the ending of the data sequence. The next step is to convert the numbers from a string format to an integer

value. The first letter that was stored will then be used as a pointer to let the microcontroller decide where the data can be used. This process can be summarized in the following code:

```
if(Serial_2.available()) {                    // Check if serial buffer is full
        BL_Data = Serial_2.read();
        // Test if incoming data is a character
        if(!isDigit(BL_Data)) {
                switch (BL_Data) {
                        case ',':             // Integer data type
                                if(BL_RX_Pointer = 'A') {
                                        …     // Data handler
                                }
                                else if …
                                …             // Other data categories
                                else {
                                        …     // Clear all variables if character is undefined
                                }
                                …             // Clear all variables at the end
                                break;
                        case ';':             // Float data type
                                if(BL_RX_Pointer = 'J') {
                                        …     // Data handler
                                }
                                else if …
                                …             // Other data categories
                                else {
                                        …     // Clear all variables if character is undefined
                                }
                                …             // Clear all variables at the end
                                break;
                                        …
```

```
                                                    …
                        default:
                                BL_RX_Pointer = BL_Data;
                }
        }
        // Test if incoming data is a number
        if(isDigit(BL_Data)) BL_RX_String += (char)BL_Data;
}
```

To convert a string data type to an integer or float data type the following code is utilized:

```
ReceivedString.toInt();          // Convert data from string to integer
ReceivedString.toFloat();        // Convert data from string to float
```

The user defined Bluetooth () function will handle the Bluetooth communication protocol from serial 2 ports at 115200 baudrate. To read the incoming data from the GPS module, a library made by Mika Hart [14] called TinyGPS++ will be used to parse and handle the GPS NMEA characters. Using this library will ease the code development for encoding GPS data. To begin using the library first include the TinyGPS++ file and initialize a GPS method outside the scope of the main code as follows:

```
#include "TinyGPS++.h"           // Include TinyGPS++ library
TinyGPSPlus GPS;                 // Initialize TinyGPS++ method
```

The ‘GPS’ initializer can then be used to encode the incoming data from the GPS module. To start encoding the GPS data, check if the serial 0 buffer is full. Once the buffer is full, encode the incoming data stream using the ‘GPS’ initializer by testing if the specified GPS data objects have been updated or not. The following code summarizes the procedure to encode GPS data using TinyGPS++ library:

```
#include "TinyGPS++.h"           // Include TinyGPS++ library
```

```
TinyGPSPlus GPS;                    // Initialize TinyGPS++ method
…
if(Serial_0.available()) {          // Check if GPS serial buffer is full
        GPS.encode(Serial_0.read());        // Encode the GPS data
        if(GPS.location.isUpdated()) {      // Store location if it is updated
                Latitude = GPS.location.lat();
                Longitude = GPS.location.lng();
        }
        // Store altitude if it is updated
        if(GPS.altitude.isUpdated()) Altitude = GPS.altitude.meters();
        // Store satellites count if it is updated
        if(GPS.satellites.isUpdated()) Satellites = GPS.satellites.value();
        …
}
```

The default baudrate used for GPS is 9600. This code is called using the user defined function GPS_Data(). The GPS module offers a FIX pin that indicates whether the module has a satellites fix or not. The FIX pin outputs a pulse once every second in case there is no satellites fix; on the other hand, when there is satellites fix the pin will output a 200 milliseconds pulse every 15 seconds. The FIX pin can; therefore, be used as an interrupt pin to inform the user whether there's satellites fix or not. This is done by measuring the period of each pulse then transmitting the GPS state to the smartphone application. The following code is used to indicate the status of the GPS module:

```
// Configuration is done in the setup function
pinMode(GPS_Interrupt, INPUT);    // Configure GPS FIX pin as input
// Attach pin to interrupt
attachInterrupt(digitalPinToInterrupt(GPS_Interrupt), ExtInterrupt, CHANGE);
                                                    …
```

```
...
// Interrupt service routine for GPS FIX pin
portENTER_CRITICAL_ISR(&Mux);
if(!GPS_Int_LastCh && digitalRead(GPS_Interrupt) == HIGH) {
        GPS_Int_LastCh = true;
        GPS_Int_Timer = millis();
}
else if(GPS_Int_LastCh && digitalRead(GPS_Interrupt) == LOW) {
        GPS_Int_LastCh = false;
        GPS_Int_RX = millis() - GPS_Int_Timer;
        // Send GPS no fix code if pulse is greater than 0.7s
        if(GPS_Int_RX >= 700) {
                GPSFix = false;
                Serial_2.print("G404,");        // Send no-fix data
        }
        // Send GPS fix code if pulse is less than 0.3s
        else if (GPS_Int_RX < 700) {
                GPSFix = true;
                Serial_2.print("G101,");        // Send fix data
        }
}
portEXIT_CRITICAL_ISR(&Mux);
```

The next part to be handled by the core 0 processor is to transmit data from the microcontroller to the smartphone application. It was observed that data cannot be transmitted instantly to the Bluetooth module. A small delay should be used in order to transmit data packets without malfunctioning the module. However, using delays within the code structure is not feasible and will cause a longer and slower code execution. This is solved by using a timer interrupt to transmit data packets at a certain delay period. The timer will interrupt every 20 milliseconds. Each time the timer interrupts a single data packet is sent to the smartphone application. The data packets are multiplexed over a specific time period. During this period a total of 12 data packets

from the sensors are transmitted, and every 12th transmission a single data packet from the GPS is then sent. The developer can modify the multiplexing process depending on the application. Here the ratio of sensors data to GPS data is 12:1. This ratio is chosen since the sensors data received in the smartphone is plotted in real time; therefore, little delay should be applied for sensors data transmission. It should be noted that when transmitting float data with n-decimal places via the UART ports the following code line should be used:

```
Serial.print(A, B);
```

Where A is the floating point number and B is an optional second parameter specifies the base (format) to use. For floating point numbers, this parameter specifies the number of decimal places. Below shows the code used for data transmission, this code is called using the user defined function Data_Transmission():

```
// Configuration is done in the setup function
timer = timerBegin(0, 80, true);                  // Use timer 0 from the ESP32
timerAttachInterrupt(timer, &onTimer, true);  // Attach a timer interrupt callback
timerAlarmWrite(timer, 25000, true);          // Configure timer to interrupt every 25ms
timerAlarmEnable(timer);                      // Enable timer
                                                  …
// Data_Transmission() function
SensorsCounter += 1;
if(SensorsCounter <= 12) {
        …                                     // Send sensors data
}
else {
        SensorsCounter = 0;
        …                                     // Send GPS data
}
```

The timer interrupt callback function sets the transmission flag to true. This flag is set to false from the main core 0 task function every time a data packet is sent. The core 0 task function is shown below:

```
while(true) {
        Bluetooth();                    // Check Bluetooth data
        GPS_Data();                     // Check GPS data
        if(TimerIntterupt) {            // Transmit data every time the timer 0 interrupts
                TimerIntterupt = false; // Set transmission flag to false
                Data_Transmission();    // Transmit data to smartphone
        }
}
```

Now that the code structure has been covered, we can summarize it in the following flow chart:

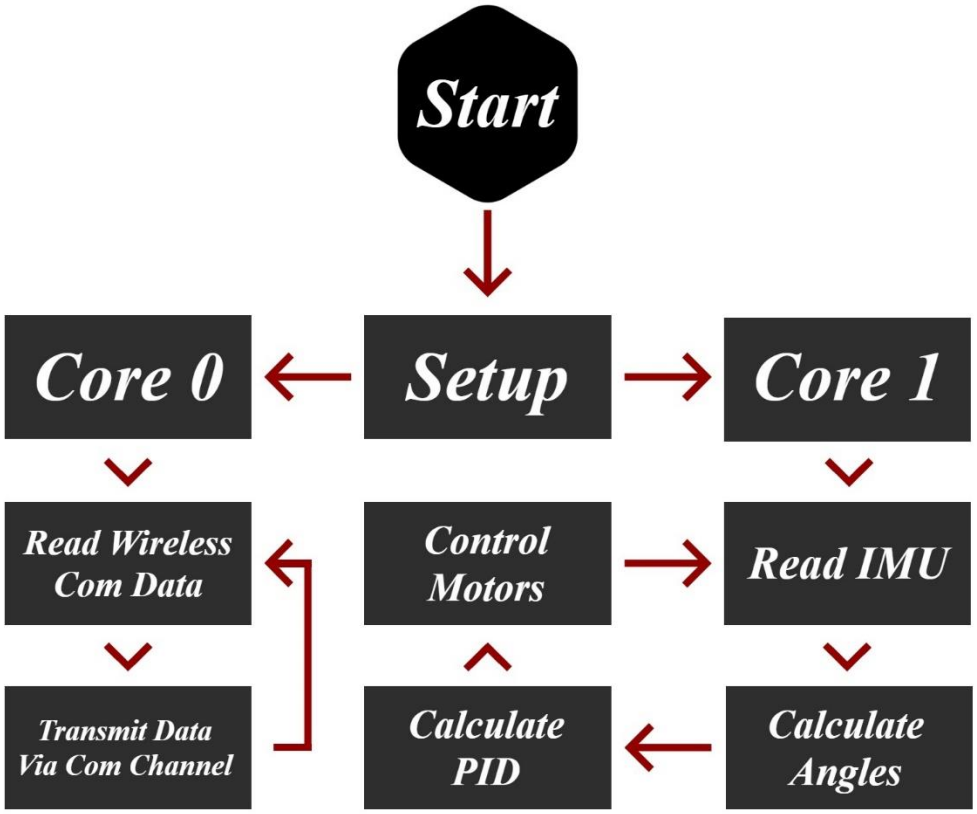

Figure 3.13 – ESP32 Microcontroller Flowchart

3.2 Android Code Development

Android is a mobile operating system developed by Google, based on a modified version of the Linux kernel and other open source software and designed primarily for touchscreen mobile devices such as smartphones and tablets. The core Android source code is known as Android Open Source Project (AOSP), and is primarily licensed under the Apache License. Android's default user interface is mainly based on direct manipulation, using touch inputs that loosely correspond to real-world actions, like swiping, tapping, pinching, and reverse pinching to manipulate on-screen objects, along with a virtual keyboard. Android homescreens are typically made up of app icons and widgets; app icons launch the associated app, whereas widgets display live, auto-updating content, such as a weather forecast, the user's email inbox, or a news ticker directly on the homescreen. Applications ("apps"), which extend the functionality of devices, are written using the Android software development kit (SDK) and, often, the Java programming language. Java may be combined with C/C++, together with a choice of non-default runtimes that allow better C++ support. [15] To develop Android applications the Android Studio software tool is used. **Android Studio** is the official integrated development environment (IDE) for Google's Android operating system, built on JetBrains' IntelliJ IDEA software and designed specifically for Android development.

Figure 3.14 – Android (Left) and Android Studio (Right) Logos

Unlike Arduino, Android software development is more complicated and requires deeper understanding and explanation. Discussing the Android code structure in details will not be covered in this report, only the main information and key points will be listed. Further explanations are left by the reader to investigate if desired. The Android code structure is summarized as follows:

1 – Manifest
2 – Java Classes
3 – Resources
4 – Assets
5 – Gradle Scripts

The manifest presents essential information about the application to the Android system, information the system must have before it can run any of the application's code. Permissions to access the hardware within the smartphone is done in the Manifest file. Every application must have Manifest file in its root directory. Java classes are templates that are used to create objects, and to define object data types and methods. Core properties include the data types and methods that may be used by the object. All class objects should have the basic class properties. Classes are categories, and objects are items within each category. Resources are the additional files and static content that the code uses, such as bitmaps, layout definitions, user interface strings, animation instructions, and more. Resources such as user interface layout and data values are stored in an XML file format (Manifest file is an xml file format as well). Any resource file name should be in lowercase letters. Assets provides a way to include arbitrary files like text, fonts, music, and video in the application. If the developer tries to include these files as resources, Android will process them into its resource system and he/she will not be able to get the raw data. If you want to access data untouched, Assets are one way to do it. Gradle allows the developer to automatize their application build following multiple steps. All the configuration is handled in the build.gradle file. The build.gradle file provides repositories and dependencies to allow the developer to simplify the implementation of external libraries. When a project is created, Android Studio already generate a working template with a default configuration. Figure 3.15 shows the structure of the Android code from the Android Studio software tool.

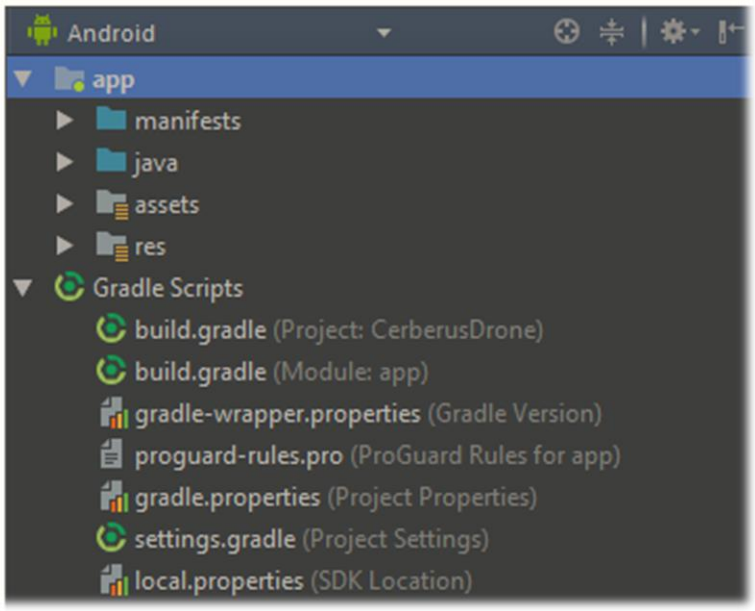

Figure 3.15 –Android Studio Code Structure

The code name of the project will be ***Cerberus Drone***. This name along with its custom designed logo will be used throughout the entire application's user interface. Figure 3.16 shows the design of Cerberus logo.

Figure 3.16 – Cerberus Logo

The layout of the application is chosen to be in the form of a tabbed navigation. The navigation style is an action bar tabs with view pager. The design consists of three tabs, they are:

1 – Home Tab
2 – Data Tab
3 – Settings Tab

This tabs layout design can be seen in the following figure:

Figure 3.17 – Application Tabs Layout Design

The home tab will provide features and information the user needs to know and be aware of. The process of controlling the quadcopter's movement, configuring the quadcopter parameters, and controlling the hardware features within the quadcopter is done in the home tab. The data tab displays any information the quadcopter transmits. This information could be the sensors readings, GPS data, and so on. The settings tab will provide the user to tweak and toggle the application's settings. The permission to switch on or off the smartphone's wireless communication hardware is done in the settings tab. Each tab will have its own Java class and layout design. In order to allow each tab to communicate with one another, a main Java class known as the main activity (MainActivity) will be used to provide a means of data exchange between each tab and handle most of the application's tasks. Therefore, each tab represents a fragment of the main activity. In Android application a **Fragment** represents a behavior or a

portion of user interface. Multiple fragments can be used in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. A fragment must always be hosted in an activity and the fragment's lifecycle is directly affected by the host activity's lifecycle. For example, when the main activity is paused, so are all the tab fragments in it, and when the main activity is destroyed, so are all the tab fragments. Below shows the hierarchical information between the main activity class and fragment classes:

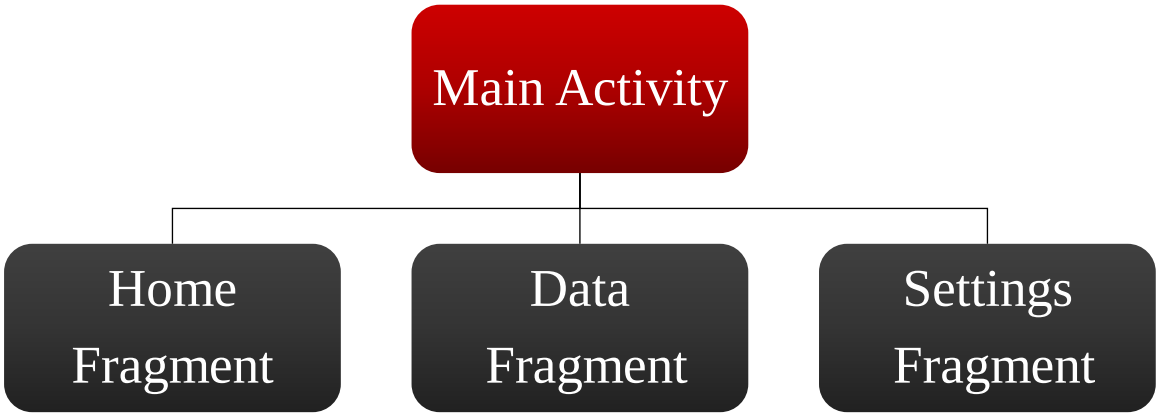

Figure 3.18 – Hierarchy Chart of Main Activity and Fragments Classes

To setup the view pager for each tab the SectionsPageAdapter class is used. An instance of the SectionsPageAdapter will be created in the MainActivity and each tab is initialized using the following method:

```
/* ******************************** Tabs ViewPager ******************************** */
public void setupViewPager(ViewPager viewPager) {
    SectionsPageAdapter adapter = new
                                  SectionsPageAdapter(getSupportFragmentManager());
    adapter.addFragment(new FragmentHome(), "Home");
    adapter.addFragment(new FragmentData(), "Data");
    adapter.addFragment(new FragmentSettings(), "Settings");
    viewPager.setAdapter(adapter);
}
```

The FragmentHome, FragmentData, and FragmentSettings are the Java classes for each tab. The layouts associated for home, data, and settings tabs are fragment_home.xml, fragment_data.xml, and fragment_settings.xml, respectively. The associated layout file for the MainActivity class is

main_activity.xml. Once the tabs have been created, the next step is to decide what should be displayed for the user into these tabs. If the smartphone is not connected to the quadcopter, both the home and data tabs should display a message indicating the user to connect to the quadcopter first as shown below:

Figure 3.19 – Indication Message if the Quadcopter is not Connected

The connection procedure is done in the settings tab where the user should toggle on the Bluetooth switch and begin searching for nearby devices. If Bluetooth is already active then the switch will automatically be on and vice versa. A dialog message will appear if the user toggles the switch on requesting permission to enable Bluetooth. In order to use Bluetooth features in the application, two permissions should be declared in the Manifest file. The first of these is **BLUETOOTH**. This permission is required to perform any Bluetooth communication, such as requesting a connection, accepting a connection, and transferring data. The other permission to be declared is either **ACCESS_COARSE_LOCATION** or **ACCESS_FINE_LOCATION**. A

location permission is required because Bluetooth scans can be used to gather information about the location of the user. This information may come from the user's own devices, as well as Bluetooth beacons in use at locations such as shops and transit facilities.

Figure 3.20 – Bluetooth Switch Toggled ON

The “ENABLE DISCOVERY” and “FIND DEVICES” buttons are only visible once the user successfully enables Bluetooth. The BluetoothAdapter class is used to check and enable/disable Bluetooth. The process of enabling or disabling Bluetooth is done by passing BluetoothAdapter action to an intent filter. **Intents** are asynchronous messages which allow application components to request functionality from other Android components. Intents allow you to interact with components from the same applications as well as with components contributed by other applications. The intent filter is then passed to a broadcast receiver using registerReceiver() method as follows:

```
if (!mBluetoothAdapter.isEnabled()) {      // Check if Bluetooth is not enabled
    Intent enableBTIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivity(enableBTIntent);
    IntentFilter BTIntent = new IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED);
    registerReceiver(mBroadcastReceiver1, BTIntent);
    return true;}
```

```
if (mBluetoothAdapter.isEnabled()) {       // Check if Bluetooth is enabled
    mBluetoothAdapter.disable();
    IntentFilter BTIntent = new IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED);
    registerReceiver(mBroadcastReceiver1, BTIntent);
    return false;}
```

A broadcast receiver (receiver) is an Android component which allows the developer to register for system or application events. All registered receivers for an event are notified by the Android runtime once this event happens. Aside from registering receivers for Bluetooth events, the system can be as well programmed to get notified each time the battery percentage is decreased or increased, to respond if the mobile network or Wi-Fi connection is changed, to broadcast if the system has finished booting, and so forth. For this project broadcast receivers are only used for Bluetooth action events. The following is the broadcast receiver that is triggered each time the state of the Bluetooth is changed:

```
/* ************ Broadcast receiver for Bluetooth ACTION_STATE_CHANGED ************ */
public final BroadcastReceiver mBroadcastReceiver1 = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (action.equals(mBluetoothAdapter.ACTION_STATE_CHANGED)) {
            final int state = intent.getIntExtra(BluetoothAdapter.EXTRA_STATE,
                                mBluetoothAdapter.ERROR);
            switch(state){
                case BluetoothAdapter.STATE_OFF:
                    FragmentSettings.Bluetooth_Switch.setChecked(false);
                    FragmentSettings.Bluetooth_SubLayout.setVisibility(View.GONE);
                    break;
                case BluetoothAdapter.STATE_TURNING_OFF:
                    // Do nothing
                    break;
                case BluetoothAdapter.STATE_ON:
                    FragmentSettings.Bluetooth_Switch.setChecked(true);
                    FragmentSettings.Bluetooth_SubLayout.setVisibility(View.VISIBLE);
                    break;
                case BluetoothAdapter.STATE_TURNING_ON:
                    // Do nothing
                    break;
            }
        }
    }
};
```

The “ENABLE DISCOVERY” button from the settings tab is used to place the Bluetooth into discovery mode for a specific period of time allowing other devices to detect your smartphone. This method is not required when connecting to a device such as the quadcopter; however, this can be used for debugging purposes. The “FIND DEVICES” button is used to scan for nearby devices and return the device name, address, and RSSI (Received Signal Strength Indicator). The

scan can be terminated by the user from the dialog box or until a certain scanning period has passed. To achieve this both the DeviceListAdapter and BluetoothAdapter classes are used in combination with intent filter and broadcast receiver. The layout used for displaying each scanned device information is the device_adapter_view.xml. This layout is filled by the DeviceListAdapter class and displayed in the fragment_settings.xml layout using FragmentSettings class. Once scan is complete each detected nearby devices are listed and the user is asked to tap on one of the devices to connect to.

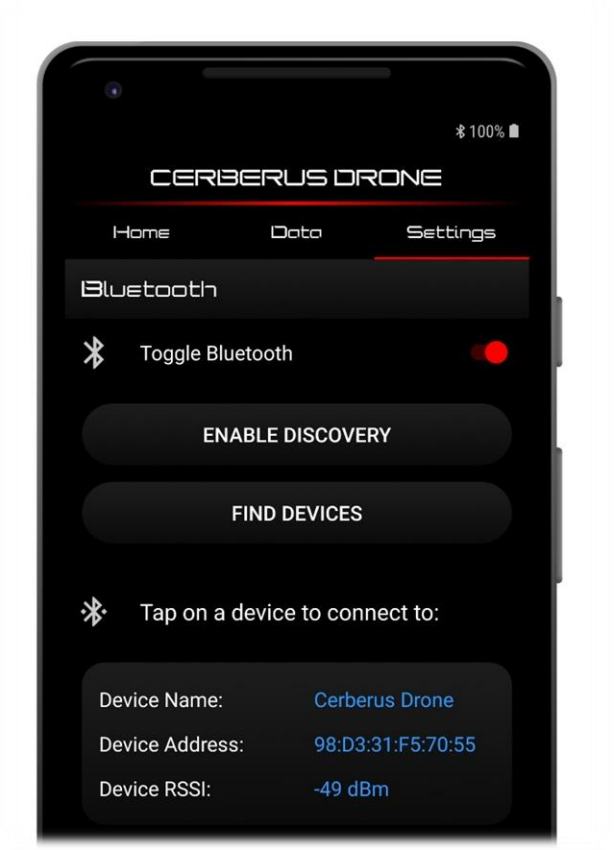

Figure 3.21 – Bluetooth Scan Results

When a user selects a device, if the device is not yet paired then the application will automatically request the user to insert the pin code for the device in order to pair with it; otherwise, if the device is already paired then the process of securing a Bluetooth connection channel is then applied. Once the device is successfully connected then the application will start listening to incoming data stream and begin data transmission; additionally, the home and data

tabs displayed message is remove to reveal their contents. The Java class used for handling the connect/disconnect process of Bluetooth as well as reading and writing to Bluetooth buffer is the BluetoothConnectionService. An intent filters are passed to broadcast receivers to respond when a connection is established or dismissed and when an incoming data is received into the Bluetooth buffer. A strip bar below the application title is used to indicate whether a device has been successfully connected or not. In the condition where a device has been connected the strip bar turns green; otherwise, the bar turns red. This can be seen in the following figure:

Figure 3.22 – Strip Bar Indicator for Successful (Right) or Unsuccessful (Left) Bluetooth Connection.

It should be noted that each Bluetooth device has its own 128-bit digit UUID code. The UUID is used for uniquely identifying information. It identifies a particular service provided by a Bluetooth device. For the HC-05 Bluetooth module the UUID is:

"00001101-0000-1000-8000-00805F9B34FB"

In **Section 3.1** the protocol used to handle incoming data was introduced for the Arduino code platform. The data format was as follows:

$$\alpha XXXX\beta$$

This protocol is carried into the Android code platform as well. To send integer or float data from the smartphone to the quadcopter device the following code methods are used:

```
/* ***************************** Send integer data ***************************** */
public void SendInteger(String DataKey, int value, String breakpoint) {
    String RXDataString = DataKey + Integer.toString(value) + breakpoint;
    byte[] bytes = RXDataString.getBytes(Charset.defaultCharset());
    mBluetoothConnection.write(bytes);}


/* ***************************** Send float data ***************************** */
public void SendFloat(String DataKey, float value, String breakpoint) {
    String RXDataString = DataKey + Float.toString(value) + breakpoint;
    byte[] bytes = RXDataString.getBytes(Charset.defaultCharset());
    mBluetoothConnection.write(bytes);}
```

When an incoming data is received, a broadcast receiver is used to respond and encode the message. An intent filter that corresponds to this broadcast receiver is declared inside the BluetoothConnectionService class as follows:

```
// Read Bluetooth buffer
bytes = mmInStream.read(buffer);
String incomingMessage = new String(buffer, 0, bytes);
Intent incomingMessageIntent = new Intent("incomingMessage");
incomingMessageIntent.putExtra("theMessage", incomingMessage);
LocalBroadcastManager.getInstance(mContext).sendBroadcast(incomingMessageIntent);
```

In MainActivity the broadcast receiver is declared as follows:

```
/* *********** Broadcast receiver for handling incoming Bluetooth data *********** */
public final BroadcastReceiver mBroadcastReceiver5 = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String text = intent.getStringExtra("theMessage");
        messages.append(text);  // Increment the message variable
        try {
            inString = messages.toString();
            // Sensors data
            if(inString.contains("D") && inString.contains(",")) {
                messages.setLength(0);  // Empty the data inside the message variable
                // Transmit acknowledge every nth readings for debugging
                DataCount++;
                if(DataCount >= 60) {
                    SendInteger("*", 1, ",");
                    DataCount = 0;}
                inString = inString.substring(inString.indexOf("D")+1,
                                    inString.indexOf(","));
                if(isInteger(inString)) inNumb = Integer.parseInt(inString));
                 // ....... Decide where the data should be used
            }
             // ....... Encode other data types
        }
        catch (Exception e) {
            messages.setLength(0);  // Empty the data inside the message variable
        }
    }
};
```

To terminate the connection between the smartphone and quadcopter an “END CONNECTION” button is used. This button becomes visible as soon as the connection between devices is accomplished.

Now that the basics of Bluetooth connection and data exchange has been covered, the developer can then decide how the received data is used and what data should be sent to the quadcopter. Data from the quadcopter’s sensors can be displayed in a graphical format using **GraphView** library. The received GPS data such as altitude, location, and satellites count can be displayed in a text view format for the user to read; additionally, the latitude and longitude data can be passed to Google Maps API to pin the location of the quadcopter on the map. It should be noted that before implementing Google Maps into the application, a unique API key should be obtained from the Google Developers website; additionally, permissions to use internet services should be included inside the Manifest file. The data tab displays the GPS and sensors data information as seen in the following figure:

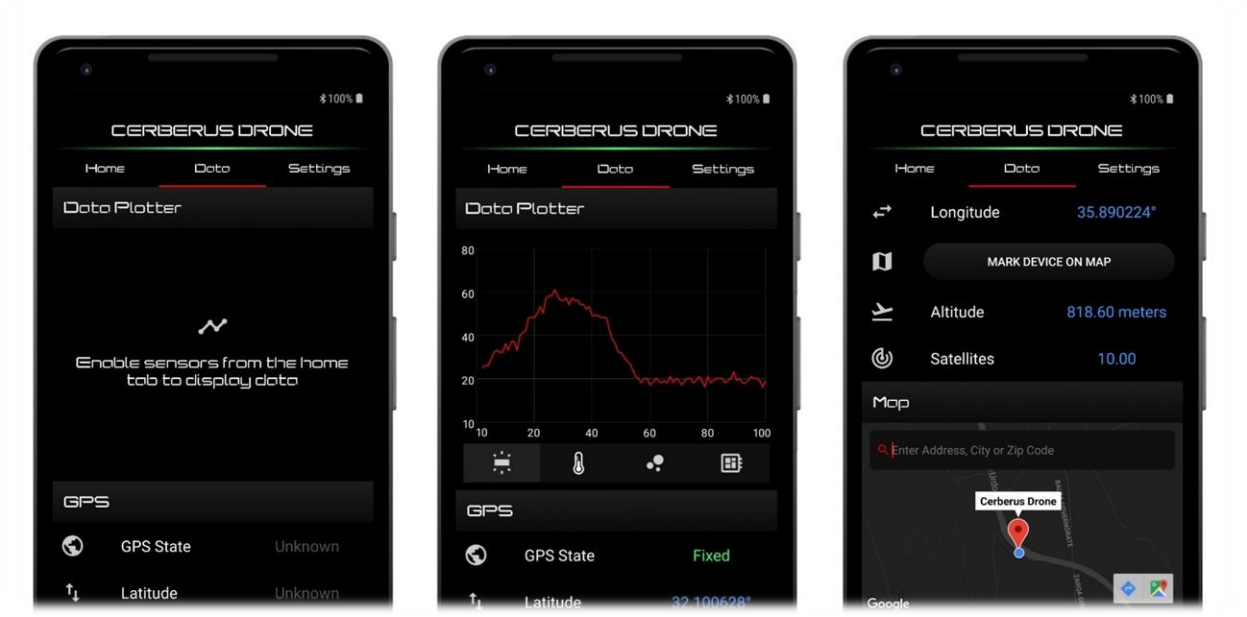

Figure 3.23 – Application Data Tab Content

Controlling the hardware and software features of the quadcopter is done in the home tab; additionally, weather forecast can be displayed to allow the user to know if the weather

conditions is suitable for flying the quadcopter or not. The home tab contents is show in the following figure:

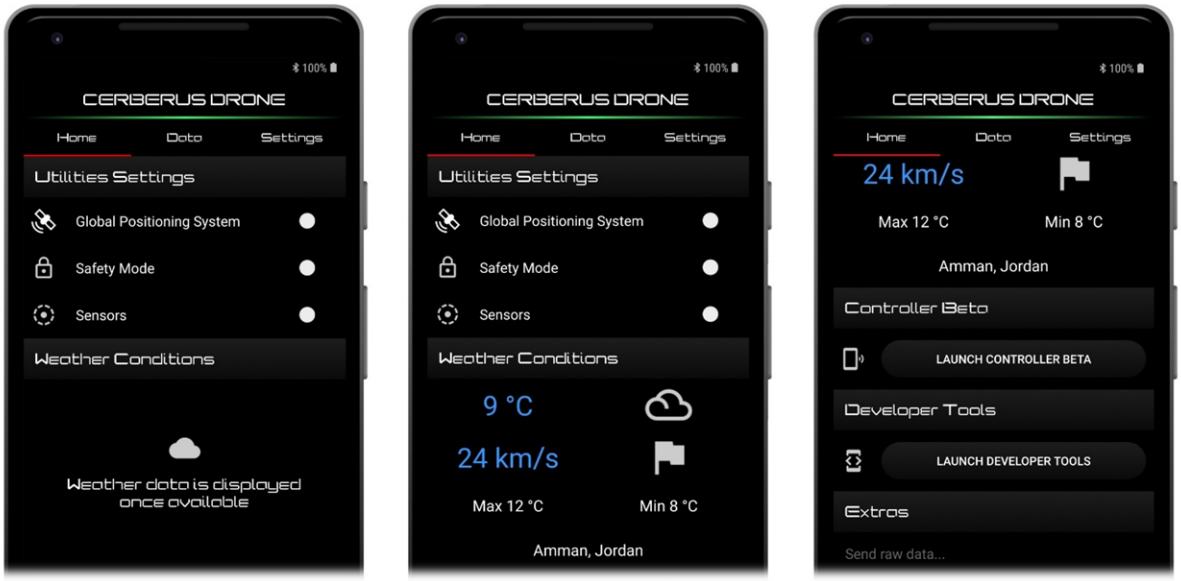

Figure 3.24 – Application Home Tab Content

To control the movements of the quadcopter a ControlActivity class is used. This Java class is responsible for sending the throttle, roll, pitch, and yaw signals to the quadcopter. The layout associated with the ControlActivity class is the activity_control.xml. Launching the ControlActivity is done from the home tab. The following figure shows the user interface design for the virtual analog controller:

Figure 3.25 – Application Control Activity Layout

To emulate an analog sticks, the **Virtual-Joystick-Android** library made by Damien Brun [16] is used. The angle of the virtual analog stick follows the rules of a simple counter-clock protractor, while the strength is percentage of how far the button is from the center to the border. This representation can be seen in the following figure:

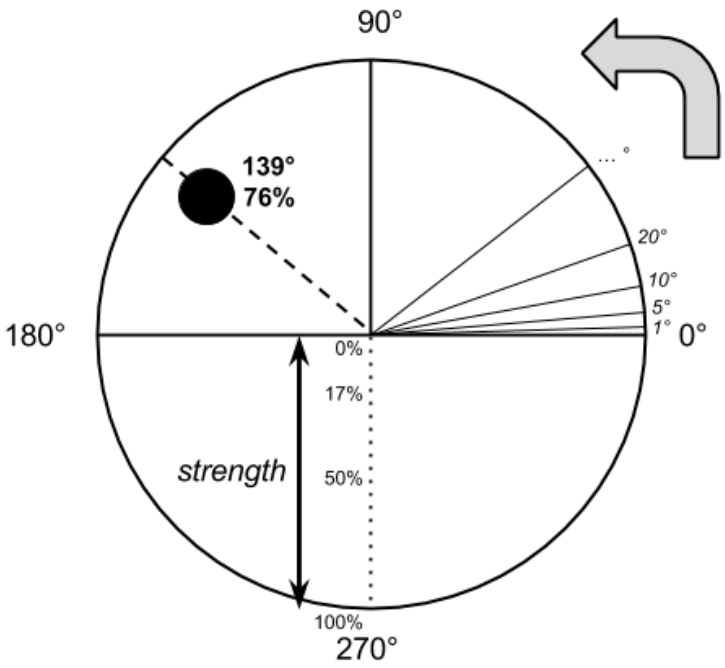

Figure 3.26 – Representation of Virtual Analog Stick Strength and Angle using Unit Circle

The code for converting, displaying, and sending the throttle and yaw values from the analog sticks to their corresponding pulses is shown below:

```
/* ********************* Analog stick for throttle and yaw ********************* */
Joystick1.setOnMoveListener(new JoystickView.OnMoveListener() {
    @Override
    public void onMove(int angle, int strength) {

        Throttle = Double.parseDouble(decimalFormat.format(strength *
                (Math.sin(Math.toRadians(angle)))
        ));
        Throttle = ((1000*Throttle)/200) + BaseValue;
        Yaw = Double.parseDouble(decimalFormat.format(strength *
                (Math.cos(Math.toRadians(angle)))
        ));
        Yaw = ((1000*Yaw)/200) + BaseValue;
        JoysticksThrottle.setText(Integer.toString((int)Throttle));
        JoysticksYaw.setText(Integer.toString((int)Yaw));
        MainActivity.getInstance().SendInteger("T", (int)Throttle, ",");
        MainActivity.getInstance().SendInteger("Y", (int)Yaw, ",");
        if(Throttle < 1080 && AnimFlag == false) {
            fadeAnimationLayout(ControllerCont, true, 100);
```

```
            fadeAnimationLayout(ControllerFlags, false, 100);
            AnimFlag = true;
        }
        else if(Throttle >= 1080 && AnimFlag == true) {
            fadeAnimationLayout(ControllerCont, false, 100);
            fadeAnimationLayout(ControllerFlags, true, 100);
            AnimFlag = false;
        }
    }
});
```

The same procedure is applied for roll and pitch analog stick values. The DeveloperActivity class is used to configure the quadcopter parameters such the PID coefficients. Like the ControlActivity, the DeveloperActivity is launched from the home tab. The layout associated with the DeveloperActivity class is the activity_developer.xml. In this activity, the user can request the PID coefficients from the quadcopter or update them. Other quadcopter parameters can be modified by the user from the smartphone depending on whether the developer decides that the user should access them or not. The DeveloperActivity layout design is shown below:

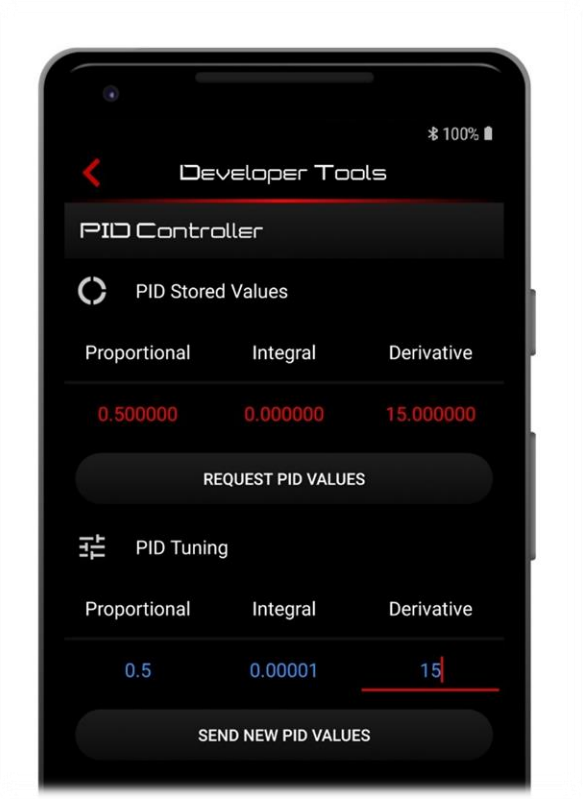

Figure 3.27 – Application Developer Activity Layout

An Android application takes some time to start up, especially during the first time the app is run on a device (sometimes this is referred to as a cold start). A splash screen is used to display start up progress to the user, or it may display branding information to identify and promote the application. For this application a splash screen activity is used to promote the logo and display guidelines before using the application. Once the user agrees and accept all guidelines the application then translates to launching the MainActivity class and layout. The SplashActivity class will handle all the startup procedures and the associated layout for it is activity_splash.xml. The following figure shows the SplashActivity layout design:

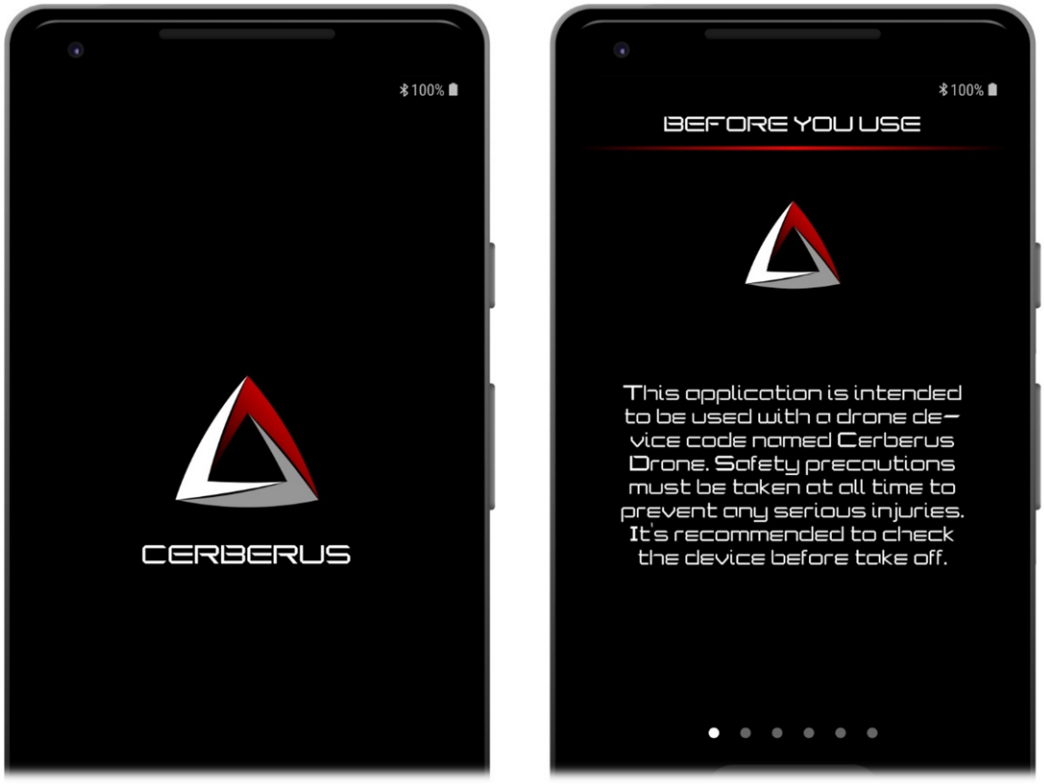

Figure 3.28 – Application Splash Activity Layout

Extra features can be added and further improvements to the application can be performed. This is left for the developer to decide how the user should interact with the application and what features are accessible to the public and which ones are not.

CHAPTER 4

RESULTS AND CONCLUSION

4.1 Project Results

For the Arduino code the delay time of each process handled by ESP32 core 1 is documented in the following table:

Process	*Time*
Reading IMU Data	~1750us
Calculating Angles	~40us
Calculating PID Outputs	~5us
Calculating ESCs Output	~1us
Output PWM Signal	~35us
Starting and Stopping Motors	~1us
Computing Other Miscellaneous Processes	~3us

Table 9 – ESP32 Core 1 Processes Time

The total processing time is then approximately 1835 microseconds or about 1.835 milliseconds which is less than the maximum time of 4 milliseconds. A period of 2.165 milliseconds is available to perform other tasks if needed.

For the gyroscope and accelerometer data obtained from the IMU, the roll and pitch angles were calculated using sensor fusion technique by applying complementary filter, the results of tilting the quadcopter along the roll and pitch axis are shown in Figure 4.1 using the graph view from the smartphone application. The Arduino code has been modified twice to display once the pitch angle and then the roll angle:

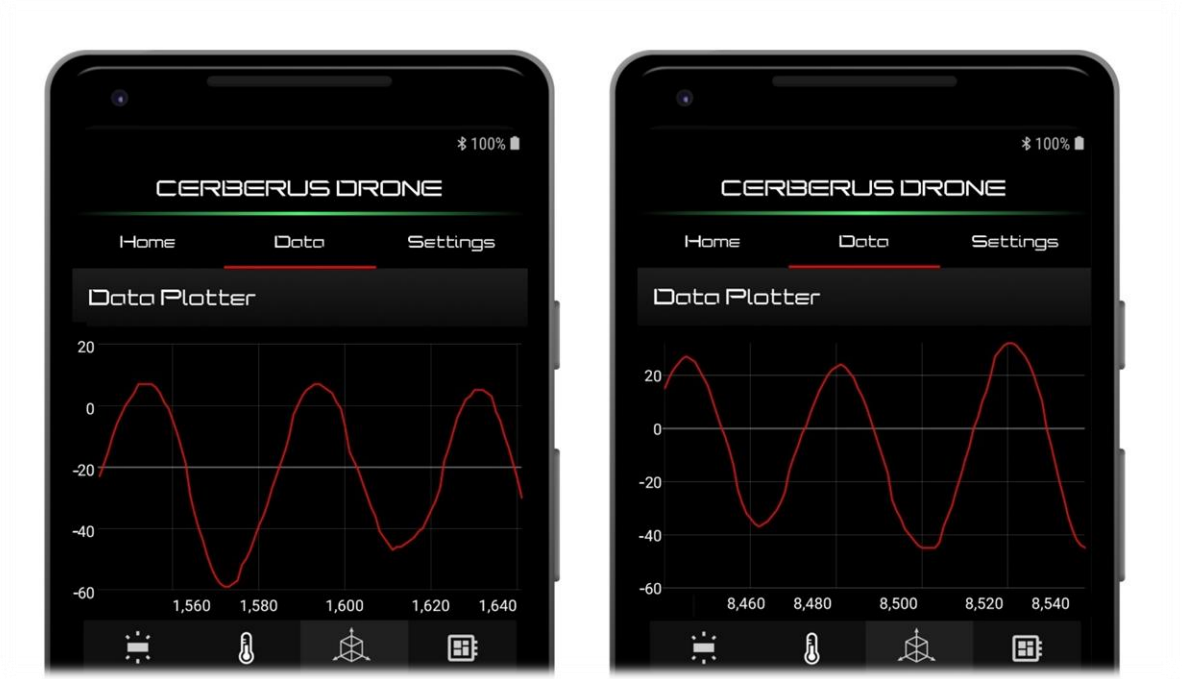

Figure 4.1 – Pitch (Left) and Roll (Right) Angles as Seen from the Application

Therefore, from Figure 4.1 the angles calculated by the microcontroller were as expected.

The roll, pitch, and yaw PID outputs are shown in Figure 4.2 1 using the graph view from the smartphone application.

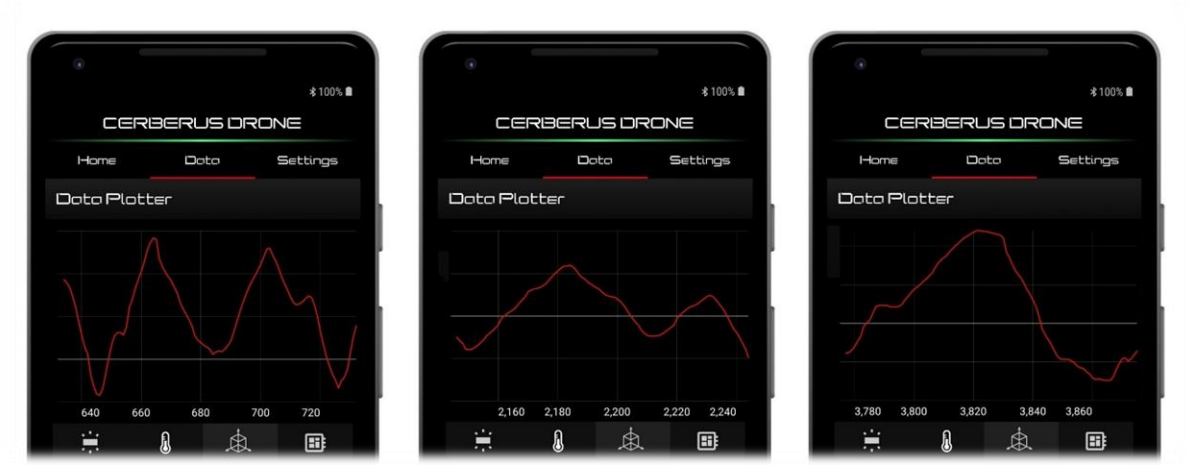

Figure 4.2 – PID Roll (Left), Pitch (Center) and Yaw (Right) Outputs as Seen from the Application

The quadcopter was tilted along the roll, pitch, and yaw angles with a small delay time between each tilt. The longer the quadcopter is away from the reference point the more the PID controller will correct the error. This response was observed by allowing only one motor to respond to changes in roll, pitch, and yaw angles. With increasing the roll or pitch tilt angle, the motor's angular speed increases as a result. Increasing the title angles suddenly resulting in the PID controller taking action and respond to the changes by compensating the error in a linear fashion.

Data obtained from the GPS module is sent to the smartphone application. The quadcopter was placed directly outside, with the square ceramic antenna pointing up with a clear sky view. The following figure shows the data received from the application

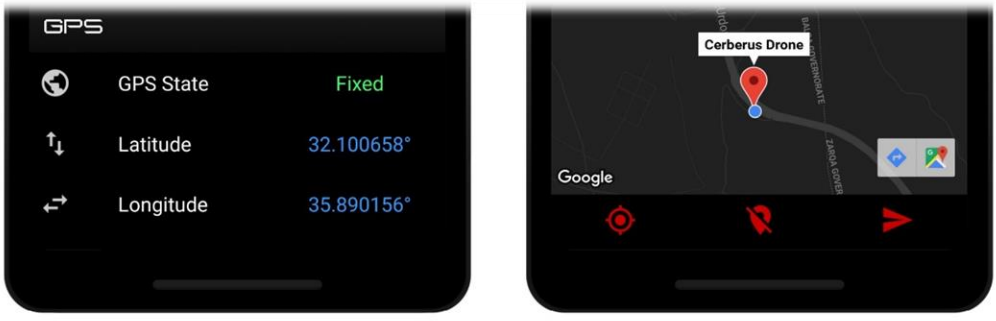

Figure 4.3 – GPS Data as Seen from the Application

4.2 Conclusion

Quadcopters in general are unstable and requires a deep understanding of the control process used to maintain the system to be stable and be capable of self-sustained flight. Different factors such as the control techniques applied and the selection of the body frame, ESCs, motors, microcontroller, power conditioning circuit and other electronic modules affect the overall system performance and flight time. The approach selected to stabilize the quadcopter system varies and is dependent upon the knowledge of the individuals and the processing speed of the electronic chip used. Efficiency of the overall system must be carefully understood. Learning and applying programming languages becomes a necessity as it is the most important part and requires most of the group effort. During the project period, the group got the opportunity to learn a substantial amount about quadcopter dynamics, control theories, circuit board design, and both Arduino and Android programming. In this period we succeeded in developing a fully functional and professionally made companion application to communicate with the quadcopter and exchange data as well as developing a ready to use code to control the quadcopter. Hovering the quadcopter was partially successful, minor setbacks such as LiPo battery problems and determining the optimum PID coefficients due to the unavailability of a system's transfer function made the project more challenging. Overall we are proud of our hard work and accomplishments and are willing to learn more on how quadcopters and UAVs work. Future plans for this project are available, these include optimizing the overall system further, understanding more about other techniques used to achieve flight stability, begin using other powerful and robust chips to handle complex computation such as decoding image and video data, design an RF analog controller to be able to control the quadcopter over long distances, start designing and developing custom made PCB boards rather than using a PCB matrix, add more sensors for the quadcopter to detect obstacles and battery monitoring for compensating the motors speed, and developing mobile applications on different platforms.

REFERENCES

[1] – "The History of Drones and Quadcopters" – www.quadcopterarena.com

[2] – "Quadcopter" – www.wikipedia.com/wiki/Quadcopter

[3] – "History of Quadcopters and Multirotors" – www.krossblade.com

[4] – "PID Controller" – www.wikipedia.org/wiki/PID_controller

[5] – "SISO System" – www.wikipedia.org/wiki/Single-input_single-output_system

[6] – "Quadcopter flight mechanics model and control algorithms", Eswarmurthi Gopalakrishnan, May 2016.

[7] – "Voltage Divider" – www.wikipedia.org/wiki/Voltage_divider

[8] – "Buck Converter" – www.wikipedia.org/wiki/Buck_converter

[9] – "ESP32 Microcontroller" – www.wikipedia.org/wiki/ESP32, "ESP32 Overview". Espressif Systems. Retrieved 1/9/2016.

[10] – "MPU-6000 and MPU-6050 Product Specification Revision 3.4", InvenSense Inc., release date: 08/19/2013.

[11] – "MT3339 All-in-One GPS Datasheet", MediaTek Inc., release date: 13/1/2017.

[12] – "Brushless DC Motors" – www.wikipedia.org/Brushless_DC_electric_motor

[13] – "Electronic Speed Control" – www.wikipedia.org/Electronic_speed_control

[14] – "TinyGPS++ Arduino library", Mikal Hart, last modified: 4/2/2018.

[15] – "Android (operating system)" – www.en.wikipedia.org/wiki/Android_(operating_system)

[16] – "Virtual-Joystick-Android Android library", Damien Brun, version 1.9.2, release date: 18/5/2018.