



**CONTROL DESIGN AND HARDWARE  
IMPLEMENTATION OF A MULTI-ROTOR SYSTEMS**

**BY**

**OBOH EDWARD OSARETIN**

**ENG1503587**

**SUPERVISED BY**

**ENGR. J.A. IGIMOH**

**DEPARTMENT OF COMPUTER ENGINEERING**

**FACULTY OF ENGINEERING**

**UNIVERSITY OF BENIN**

**A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE  
REQUIREMENTS FOR THE AWARD OF THE DEGREE OF  
BACHELOR OF ENGINEERING (B.ENG) IN COMPUTER  
ENGINEERING, UNIVERSITY OF BENIN, EDO STATE,  
NIGERIA.**

**JULY 2021**

# CERTIFICATION

This is to certify that **OBOH EDWARD OSARETIN** an undergraduate student in the Department of Computer Engineering, Faculty of Engineering, University of Benin, Edo state, with Matriculation number **ENG1503587** satisfactorily completed this work on his own as a partial fulfillment of the requirement for the award of **Bachelor Degree in Engineering (B.Eng) in Computer Engineering**.

---

*Engr. J.A Igimoh*  
(Supervisor)

---

*Engr. U. Iruansi*  
(Head of Department)

# **DEDICATION**

This project is dedicated to God for His mercy and protection and for the knowledge he has enabled me to acquire.

# ACKNOWLEDGEMENT

My sincere gratitude goes to God Almighty for giving me the moral, courage and enthusiasm to embark on this project. Project which has opened my eyes to the different technology advancements under the scope of study of the work.

I appreciate the efforts of my parents **Mr. and Mrs. Oboh** for bringing me up morally and academically. I must register my profound gratitude to my parent for their guide, moral and financial supports.

I wish to appreciate the University of Benin for giving me this great opportunity. I also wish to appreciate the Head of department, Computer Engineering, **Engr. Dr. U. Iruansi** and my course adviser Engr A. Obayawana.

It is pertinent at this juncture to appreciate the efforts of my project supervisor **Engr. J.A Igimoh** for his support and encouragement.

I also wish to appreciate all my lecturers whose efforts have contributed to my knowledge base in Engineering. I want to say a big **Thank you**.

# **ABSTRACT**

Unmanned aerial vehicles (UAVs) are being increasingly used today than ever before in both military and civil applications. The rapid advancement in miniature sensors, actuators and processors has led to the development of powerful autopilot systems which play a major role in UAVs control by making the flight safer and more efficient. These gave a rise to small sized, interestingly featured commercial UAVs, one of which is the quadcopter.

Firstly, the project started by developing a reference frame and a mathematical model for a Quadcopter system. Next, a flight orientation estimation was determined through an assortment of MEMS sensors such as an accelerometer and a gyroscope. Each sensor was individually addressed as to its strengths and weaknesses with regards to orientation estimation. Key constructs of the system include hardware and software specifications for a flight controller, a radio system, and “sensorless” brushless motor controllers. An algorithm was then used for the data fusion of these various sensors. The fused data was then fed into a control system that will efficiently stabilize the quadcopter.

At the end, the project work overviewed crucial concepts involved in achieving quadcopter flight such as orientation estimation and control system implementation. The project would present researchers with comprehensive hardware and software specifications for a quadcopter system. The primary application for the system would be for research with regards to the implementation of advance control techniques as well as data acquisition.

# TABLE OF CONTENT

CONTROL DESIGN AND HARDWARE .....	0
IMPLEMENTATION OF A MULTI-ROTOR SYSTEMS .....	0
CERTIFICATION .....	1
DEDICATION .....	2
ACKNOWLEDGEMENT .....	3
ABSTRACT .....	4
TABLE OF CONTENT .....	5
CHAPTER 1 .....	7
INTRODUCTION .....	7
1.1 History of Quadcopters:.....	7
1.2 Problem Statement: .....	8
1.3 Motivation and Objective: .....	9
1.4 Scope of Studies: .....	9
CHAPTER 2 .....	10
RELATED WORKS .....	10
2.1 Previous Work .....	10
2.2 Contributions .....	11
2.3 Common Quadcopter Motor Types .....	12
2.4 Quadcopter Configurations and Frame Design.....	13
2.5 Quadcopter System Architecture.....	15
CHAPTER 3 .....	17
METHODOLOGY .....	17
3.1 Reference Frame and Control Objective.....	18
3.1.1 Quadcopter Reference Frame .....	18
3.1.2 Control Objective .....	18
3.1.3 Reason for Mathematical Derivation.....	19
3.2 Hardware Components .....	20
3.3 IMU Sensor Implementation .....	20
3.3.1 3 Axis Accelerometer .....	21
3.3.2 3 Axis Gyroscope .....	22

3.3.3 Sensor Fusion .....	23
3.4 Control System Implementation .....	24
3.4.1 PID Basics.....	25
3.4.2 PID Control Software.....	27
3.5 Software Architecture .....	28
3.4.1 Software Flowchart Diagram .....	28
3.6 Hardware Implementation .....	29
3.4.1 Flight Controller Hardware .....	29
3.4.2 Remote Controller Hardware.....	31
3.4.3 Electronic Speed Controller Hardware.....	33
3.4.4 Brushless Motor .....	34
CHAPTER 4 .....	36
EXPECTED RESULTS .....	36
4.1 Result and Outcome .....	36
4.2 Bill of Engineering Measurement and Evaluation .....	36
CHAPTER 5 .....	38
CONCLUSION .....	38
5.1 Overview.....	38
5.2 Future Work .....	38
REFERENCES .....	39
APPENDIX .....	40
Flight Controller Arduino Code.....	40
Radio Transmitter Arduino Code.....	57
Radio Receiver Arduino Code.....	60
ESC Calibration Arduino Code .....	65

# CHAPTER 1

## INTRODUCTION

A quadcopter also known as a quadrotor is a multi-rotor unmanned aerial vehicle (UAV). Quadcopters falls in the category of vertical take-off and landing (VTOL) UAVs. A quadcopter has four rotors in square formation at the equal distance from the center of mass of the vehicle. Speed of the rotors is manipulated to perform different maneuvers, hovering, take-off and landing. Before GPS and internet, drones were only available for military use, but with continuous development in the UAV technology and exceptional growth rate in the previous ten years the drones have become very popular among the civil sector. With growth in popularity drones' market was valued at 18.14 billion USD and it is expected to reach 52.30 billion USD by 2025 (Markets 2018.)

Advancement and introduction of an impressive technology in drones has developed new fields of applications for it. Today drones are being used in several areas for various purposes. Stated below are some of common areas of drones' applications. (Fiaz and Mukarram 2018a; VBROADCAST LIMITED 2019.)

- Aerial photography
- Search and rescue
- Agriculture
- Shipping and delivery
- Engineering applications
- 3-D mapping
- Research and science
- Aerial surveillance
- Mineral exploration
- Military use, etc.

### 1.1 History of Quadcopters:

The history of Quadcopters starts in the beginning of the 20<sup>th</sup> century. The first ever quadcopter built was "Gyroplane n: 01" in 1907. This quadcopter had many limitations. Its stabilization was achieved by control of people on the ground. During the 1920's, other quadcopters with much improved performance were



built by engineers who targeted the vertical flight. In later years, the concept of Vertical Take-off and landing (VTOL) (add abbreviation) has become of a major interest to aerial researches and thus the study of quadcopter structure has evolved. For various applications, the need for Unmanned aerial vehicles (UAV) (add abbreviation) has arose and thus requirements for small sized and efficient vehicles are considered. Recently, small sized quadcopters have become a subject of UAVs due to their agile maneuverability and indoor and outdoor flight capabilities. Quadcopter nowadays are cooperated for the purpose of achieving many consumer tasks and applications. Open-source systems are now available to allow people developing their own quadcopters with light and inexpensive electronic materials.

## 1.2 Problem Statement:

The project entails quadcopter control theory, inertial measurement unit (IMU) orientation fusion, and then provides a comprehensive software/hardware platform. To establish a reference frame, the project specifies a mathematical coordinate system and then uses this system to determine control goals. The remainder of the project focuses on how to meet the proposed control goals. An overview of existing IMU sensors is presented as well as their trade-offs with respect to quadcopter flight. An algorithm is then used to overcome existing sensor limitations by fusing IMU data to obtain orientation. Once known orientation is established, the project then focuses on specifying multi-rotor kinematics, software, and hardware involved in flight control.

The primary problem this project attempts to solve is in reducing barriers to entry for advance control techniques. When building a multi-rotor vehicle (drone), designers are faced with the choice of paying for a custom designed aerial vehicle, building their own vehicle from scratch, or sacrificing controllability for an inexpensive off-the-shelf system. While numerous inexpensive off-the-shelf multi-rotor platforms are available, they often consist of proprietary modules even when advertised as open-source. Common examples of these black-box modules are sensor-less brushless motor controllers, flight controllers, and radios. These modules are often proprietary and have limited hardware specifications (Clean Flight, “Clean Flight”, 2016), (Open Pilot, “Open Pilot”, 2016). Consequently, for a researcher, the control and the modifiability of these modules is limited.

### **1.3 Motivation and Objective:**

A main objective of this work is to provide researchers with a functional, fully specified, and stabilized quadcopter. This system will be specified from scratch hardware and software with the intent of eliminating as many black box components as possible. In addition, this flight system will have an emphasis on theoretical control as well as IMU data collection making it a prime candidate for future research.

### **1.4 Scope of Studies:**

The objective of this project is to utilize the existing material to understand dynamic equations and behavior of quadcopters. Depending on the dynamic equations of the quadcopter a Proportional, Integral and Derivative (PID) based control system will be designed and implemented to achieve control of the quadcopter. The designed controller will be able to control attitude of the vehicle (Roll, Pitch and Yaw). The final project work will explain the PID controllers tuning process and integration of the designed controller with real hardware in detail. The project is primarily focused on the PID controller, other control strategies would not be explained in this project. Altitude control and autonomous navigation are not part of this project, altitude and position of the vehicle in an inertial frame will be controlled by the pilot commands. Hardware board, Sensors, Electrical and Mechanical components will be selected, and their general working and compatibility to each other will be discussed for implementation of the controller on the actual system. However, components manufacturing process, materials and detailed working are not concerned with the purpose of the project.

# CHAPTER 2

## RELATED WORKS

### 2.1 Previous Work

Admittedly, much work has been done in the area of advance control of multi-rotor systems and these works are too numerous for a comprehensive listing. Consequently, key examples will be provided that were be used as a reference for the development of this project. For example, Robert Mahony presents a comprehensive method for modeling, orientation, and control of a quadcopter with state space methods (R. Mahony, V. Kumar, and P. Corke, 2012). Another example of advance control is where researchers at University of Zurich implemented a quadcopter with a model predictive control were able to perform extreme acrobatic maneuvers (M. Mueller and D. Raffaello, 2013), (S. Lupashin, A. Schoellig, M. Sherback, and R. D’Andrea, 2010). Numerous other control techniques have been applied to quadcopters as well such as PID, LQR, LQR-PID, and  $H_\infty$  (L. Argentim, W. Contrimas, P. Santos, and R. Aguiar, 2013), (G. Raffo, M. Ortega, and F. Rubio, 2013). With regards to open-source multi-rotor systems, Open Pilot and Clean Flight are perhaps two of the most popular open software flight controller systems specifications (Clean Flight, “Clean Flight”, 2016), (Open Pilot, “Open Pilot”, 2016). These frameworks support a broad range of multi-rotor vehicles from tri-copters to octo-copters. With regards to open-source software and hardware systems, the Pixhawk and Sparky systems feature an open-source flight controller (Pixhawk, 2016), (Tau Labs, “Sparky 2”, 2016). While these systems feature some open hardware and software, they integrate with systems that are proprietary. In regards to indoor autonomous control, this is an active area of research for all types of remote vehicles. Mapping of unknown environments has been conducted with mutli-rotor vehicles utilizing lidar and employing the iterative closest point (ICP) algorithm (S. Winkvist, 2013), (Z. Zhang, 1994), (J. Zhang and S. Singh, 2014). However, in these cases the multi-rotor vehicle was operated by a human. A team at MIT achieved autonomous indoor control of an aerial vehicle by combining lidar data and IMU data with an extended Kalman filter as well as a Gaussian particle filter (A. Bry, A. Bachrach, and N. Roy, 2012). However, in this case the environment had been pre-mapped and pre-determined trajectories were used. Comprehensive simultaneous mapping and control is still an ongoing area of research.

In order to provide context for future chapters, this chapter introduces the basic inputs and outputs common to a quadcopter system. As the name quadcopter implies, a quadcopter is a multi-rotor aircraft with four propellers. Beyond the similarity of four propellers, there is significant design diversity. This design diversity includes but is not limited to motor type and frame design.



Figure 2.1: Examples of quadcopter implementations

The quadcopters in Fig. 2.1 are referenced throughout this chapter as demonstrations of different types of design. A defining aspect of these quadcopters is the motor type that they utilize. Consequently, the next section will overview the common types of quadcopter motors.

## 2.2 Contributions

This project develops a functional quadcopter system with a higher level of integration than most other open-source options. The key contributions of this flight system are as follows:

- Provides open-source software / hardware files for a functional stabilized auto-leveling flight controller
- Provides open-source software / hardware files for a remote-control system

- Offers practical design insights for other researchers attempting to construct their own aerial vehicles

## 2.3 Common Quadcopter Motor Types

Low-cost commercial quadcopters generally use electric DC motors such as brushed and brushless permanent magnet motors. In contrast to gas motors, there exist small electric motors that are light-weight, low-cost, and of simple construction. These features make small electric motors ideal for low-cost commercial quadcopters. Among these electric motors, two common types of DC electric motors exist which are brushed and brushless motors. As the name implies, brushed DC motors are mechanically commutated with a brush, are powered by DC, and are explained in detail in (C. Hubert, 2002). The control simplification and cost of brushed DC motors makes them a popular choice for micro-size quadcopters such as the Turnigy Micro-X shown in Fig. 2.1(d). However, the brush which mechanically commutates the motor results in friction losses as well as limited motor life span. Consequently, larger quadcopters often use brushless permanent magnet DC motors which are electronically commutated. Brushless motors are also split into two common types which are sensored and sensor-less motors. Examples of these two motor types are illustrated in Fig. 2.2.

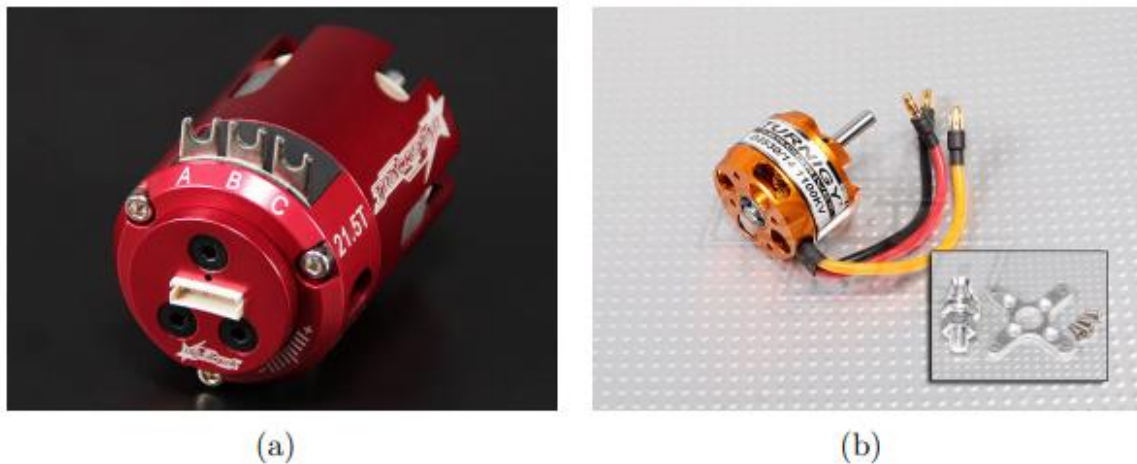


Figure 2.2: Examples of sensored(a) and sensor-less(b) brush-less permanent magnet motors

In order to electronically commutate a permanent magnet motor, the rotor's position must be known. This can be achieved by using Hall effect sensors or by using sensor-less driving techniques. Sensored motor

operation simplifies driving complexity but results in heavier and more expensive motor as seen in Fig. 2.2. Instead of using sensors, sensor-less techniques such as back electro-motive force (BEMF) zero-cross detection and field-oriented control can be implemented (NXP, 2016). Sensor-less operation is desired since sensor-less motors have reduced weight, cost, and complexity. As a result of this, sensor-less brushless motors are common in quadcopters. However, in order to drive these motors, a DC to AC 3 phase sensor-less motor driver is needed. In terms of popular multi-rotor vernacular, these are commonly referred to as electronic speed controllers (ESC).

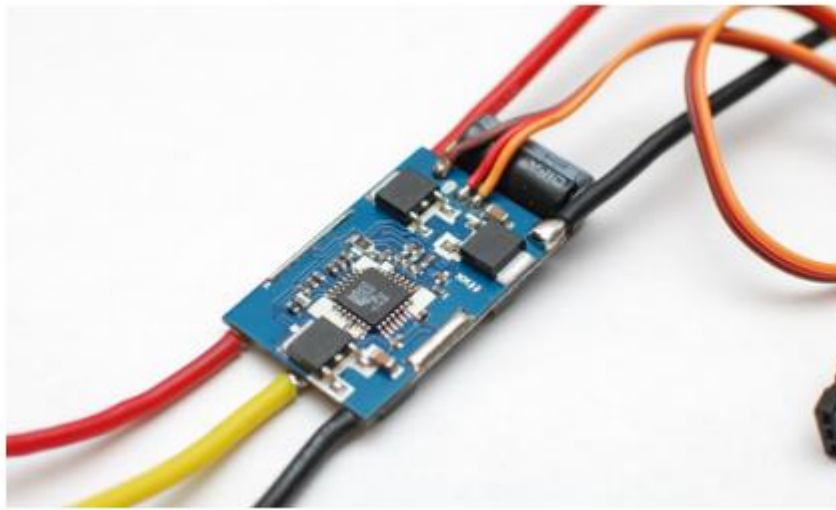


Figure 2.3: Example of ESC

These ESCs represent an integral part of the quadcopter system architecture since their output controls the orientation of the quadcopter by varying the speed of the propellers.

## 2.4 Quadcopter Configurations and Frame Design

Quadcopters have various configurations though the most common types are the X configuration, the H configuration, and the + configuration. These various configurations are illustrated in Fig. 2.4.

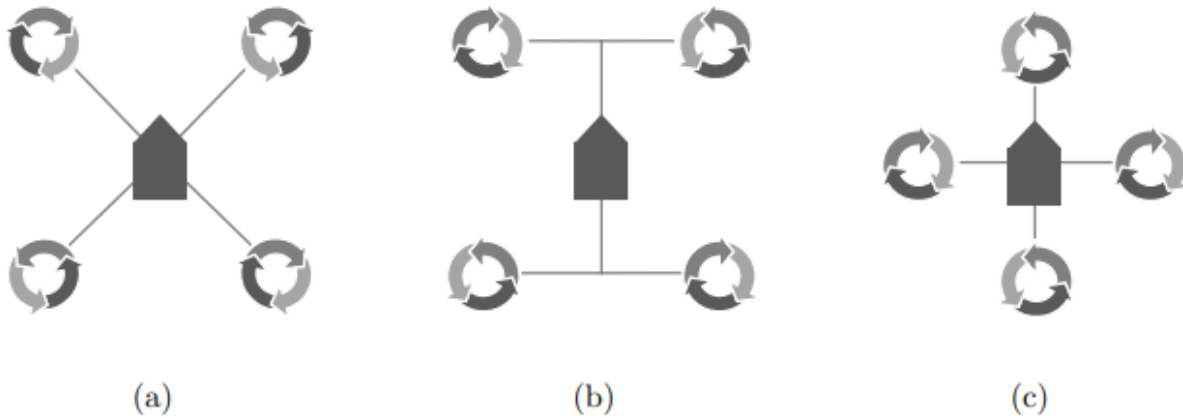


Figure 2.4: (a): X, (b): H, (c): + configuration quadcopters. In this figure, arrows represent propeller direction, wires represent rigid frame, and the center represents the quad's frame body

Each type of configuration offers advantages and disadvantages. The X configuration is the most commonly utilized motor configuration since it is simple to construct, ideal for a forward-facing camera, and is symmetrical. Quadcopters in Fig. 2.1(a,b,c) are using this type of configuration. A disadvantage of this configuration is an increase in control complexity. In contrast to the X configuration, the + configuration is the simplest to mathematically model and control. However, this configuration is least ideal for a forward-facing camera. Consequently, very few commercial drones are sold in this configuration and they generally only appear in research or DIY projects. Finally, the H configuration is sometimes built for mechanical convenience as seen in Fig. 2.1(d). In contrast, the DJI Inspire in Fig. 2.1(c) was designed as a H configuration quadcopter to achieve improved camera perspective (DJI, 2015). While this configuration can be ideal for forward facing cameras, it is also not symmetrical about its center. This lack of symmetry should be taken into account and should be considered when choosing which configuration to with. These three configurations including minor deviations represent the majority of quadcopter configurations in common use today.

## 2.5 Quadcopter System Architecture

A step in presenting necessary background information for the quadcopter project is to briefly overview the electrical architecture. An example block diagram of the architecture unique to the most quadcopters is shown in Fig. 2.5.

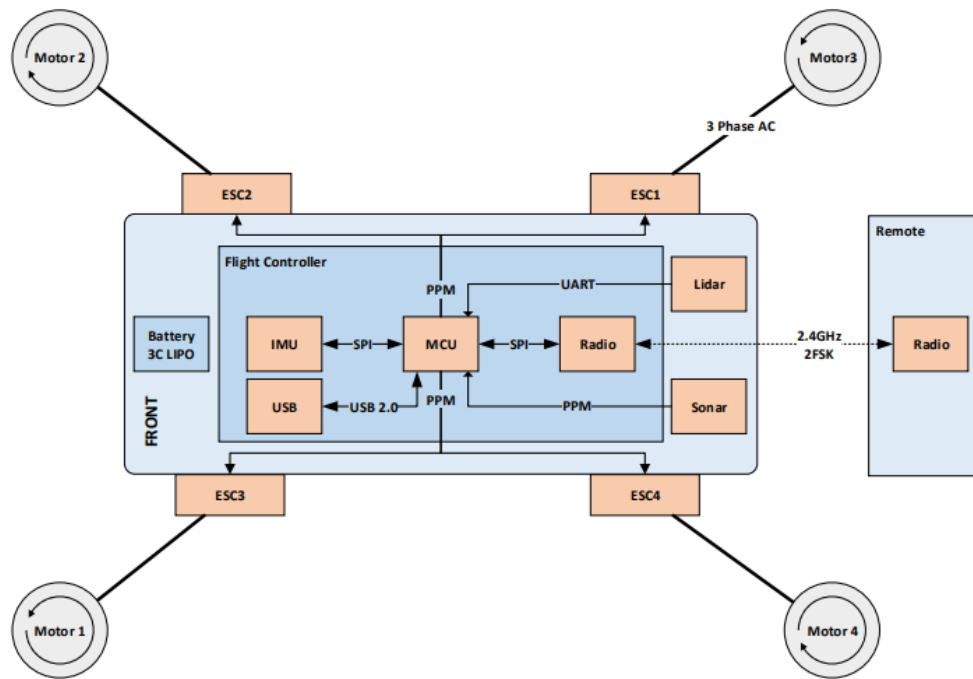


Figure 2.5: Drone electrical architecture.

The underlying system for the quadcopter shown above is named the “Marq Drone” system. This system consists of a flight controller, a radio, and ESCs. Core components of the flight controller include a microcontroller unit (MCU), a radio, and an inertial measurement unit (IMU). The flight controller communicates with the ESCs through pulse position modulation (PPM). More information about PPM can be found here (G Lazaridis, 2011).

In addition to communicating with ESCs, the flight controller shown above communicates with a lidar sensor through a universal asynchronous receive and transmit (UART) interface. The build of the quadcopter did not include a lidar sensor and UART interface because I decided to make use of a cheap low-end microcontroller as the MCU in my build. A lidar sensor could be seamlessly interfaced with our choice of microcontroller but as the number of sensors to be interfaced increases, the response time of the



system will increase, hence the exemption of the lidar sensor. To communicate with devices external to the quadcopter, a USB interface can be used for data acquisition or programming. Another communication method is through a wireless 2.4GHz frequency shift keyed (FSK) interface for receiving flight commands from the radio controller and transceiving flight data. Now that all the components involved have been introduced, the I/O of the system can be identified. For the flight controller itself, feedback inputs are from the IMU sensor. The outputs of the system are the four individual PPM signals that are sent to the ESCs. In the project work, a control scheme was derived that uses these I/O to achieve stable flight.

## CHAPTER 3

# METHODOLOGY

This chapter overviews the electronic hardware design of the quadcopter. These include the radio system, flight controller, and motor controllers. An example picture of a quadcopter is shown below

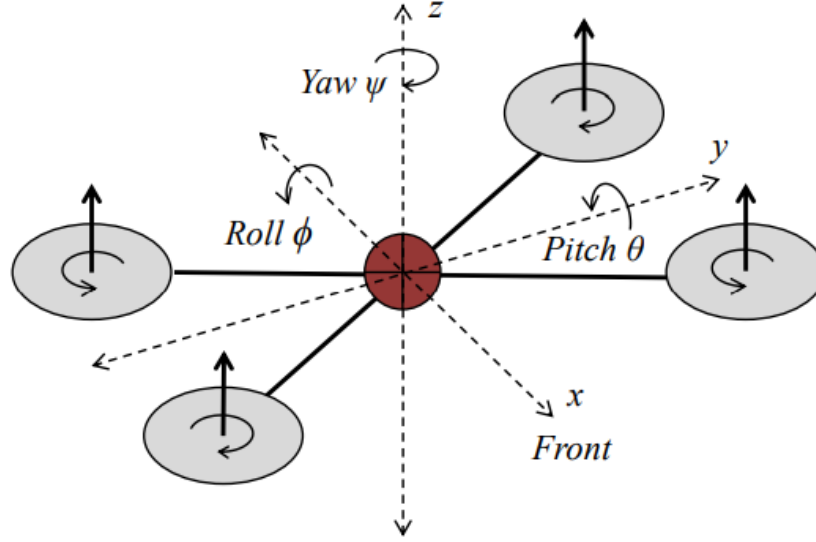


These pieces of hardware in the diagram above were chosen in order to help understand the nature of the underlying system of a quadcopter as well to be able to implement a more flexible architecture.

### 3.1 Reference Frame and Control Objective

A reference coordinate system for the quadcopter will be presented in addition to control goals. Like other multi-rotor vehicles, the primary control objective is to keep the quadcopter's orientation controlled and its altitude above ground non zero. With this in mind, a commonly used reference coordinate system will be presented to explicitly define this control objective.

#### 3.1.1 Quadcopter Reference Frame



The figure above demonstrates a reference frame for quadcopters. The reference frame uses Cartesian x,y,z system of coordinates for position and uses Euler angles pitch ( $\theta$ ), roll ( $\phi$ ), and yaw ( $\psi$ ) to denote orientation. The origin the axes represents the center of the quadcopter as well as the center of mass.

#### 3.1.2 Control Objective

A primary control objective for the quadcopter is to be able to control orientation. With this in mind, the first set of control goals can be summarized in equation 3.1 where  $\theta_c$ ,  $\phi_c$ ,  $\psi_c$  represent a user commanded rotation, where  $\dot{\theta}$ ,  $\dot{\phi}$ ,  $\dot{\psi}$  represent the rotational velocities of the quadcopter, and where  $T_c$  represents the user commanded throttle.

$$\begin{aligned} \dot{\theta} &= 0 \quad \dot{\phi} = 0 \quad \dot{\psi} = 0 \\ \theta &\rightarrow \theta_c \quad \phi \rightarrow \phi_c \quad \psi \rightarrow \psi_c, \quad T \rightarrow T_c \end{aligned} \quad (3.1)$$

These control parameters,  $\theta_c$ ,  $\phi_c$ ,  $\psi_c$ ,  $T_c$ , allow a user to maneuver the quadcopter to anywhere in three-dimensional space. Consequently, many commercial systems give users these four degrees of freedom to operate a quadcopter. However, the user must also act as a control system in order to regulate the quadcopter's height to keep it above the ground. To achieve this, the user must observe the quadcopter's height and constantly adjust the throttle,  $T_c$ , such that the height with respect to ground,  $z$ , is roughly constant. Also, if a user desires a quadcopter to stay at a fixed position in space, they must also observe the  $x$ ,  $y$  position of the quadcopter and adjust  $\theta$ ,  $\phi$  accordingly. While these parameters allow a user to control a quadcopter, these four degrees of freedom are not enough for stable autonomous flight. Without a user to observe and control the quadcopter's  $x$ ,  $y$ ,  $z$  position with respect to the room, stable flight is not possible. Therefore, as a secondary control objective, equation 3.2 demonstrates the control needed for autonomous flight where  $x_c$ ,  $y_c$ ,  $z_c$  represents a user desired position.

$$\begin{aligned} \dot{x} &= 0, \dot{y} = 0, \dot{z} = 0 \\ x &\rightarrow x_c, y \rightarrow y_c, z \rightarrow z_c \end{aligned} \quad (3.2)$$

To meet the autonomous control objectives in equation 3.2, additional sensors are needed such as lidar, GPS, and sonar. These methods will be introduced later in this thesis.

### 3.1.3 Reason for Mathematical Derivation

A mathematical derivation of a multi-rotor frame provides a basis for update laws and helps a designer gain intuition regarding the system. In simple cases, these update laws can be derived from visual inspection of the airframe.

However, this method becomes less effective as motor count is increased, as asymmetries are introduced to the air frame, and as motor angles are changed. Consequently, the incentive of the following sections is to translate input of motor speed to angular velocities ( $\dot{\theta}$ ,  $\dot{\phi}$ ,  $\dot{\psi}$ ) such that the control system can drive pitch ( $\theta$ ), roll ( $\phi$ ), and yaw ( $\psi$ ) to their desired values.

### 3.2 Hardware Components

The components were selected considering the performance and compatibility with other selected components. The most suitable components were selected considering the application and budget of the project to build the actual model.

Frame	DJI F450 Quadcopter Frame
Propellers/Rotors	10x4.5 Propellers (4)
Motors	A2212 BLDC motors
ESCs	Hobby King 30A Brushless ESC
Battery	11.1V Li-po Battery
Transmitter	NRF24L10 PA (power amplifier) LNA (low noise amplifier)
Receiver	NRF24L10
Inertial Measurement Unit (IMU)	MPU 6050 6dof IMU
Arduino Boards	Arduino UNO (1), Arduino Nano (2)

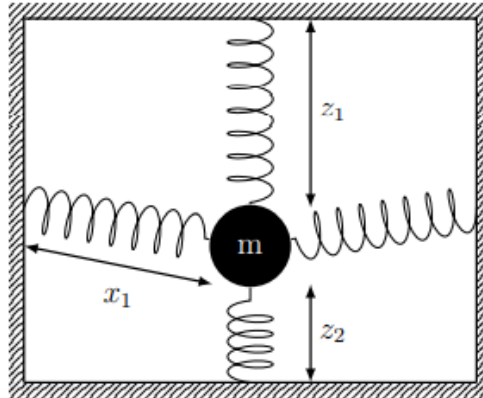
### 3.3 IMU Sensor Implementation

In order to meet the control goals, estimation of the quadcopter's orientation is required. With modern advancements in electronics, determining orientation can be done inexpensively, efficiently, and quickly with micro electrical mechanical system (MEMS) based sensors. Unfortunately, currently, there is no single affordable MEMS sensor that directly measures  $\theta$ ,  $\phi$ ,  $\psi$ . Consequently, the combination of multiple MEMS sensors is required in order to accurately estimate orientation. This section overviews the operation

of relevant MEMS sensors as well as their strengths and weaknesses with respect to multi-rotor aircraft. The sensor used for Inertial Measurement Unit (IMU) is the MPU 6050 6dof IMU.

### 3.3.1 3 Axis Accelerometer

3 axis accelerometers are devices that are designed to measure Cartesian  $\hat{x}$ ,  $\hat{y}$ ,  $\hat{z}$  accelerations. It follows that the output of the device is an acceleration vector denoted as  $A = [A_x, A_y, A_z]$ .



Spring Accelerometer

```

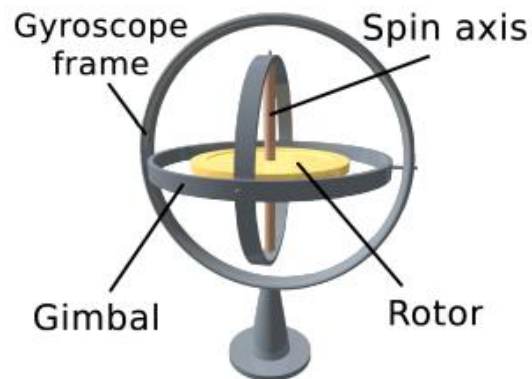
229
230 ////////////////////////////////////////////////////Acc read//////////////////////////////////////
231 Wire.beginTransmission(0x68); //begin, Send the slave address (in this case 68)
232 Wire.write(0x3B); //Ask for the 0x3B register- correspond to AcX
233 Wire.endTransmission(false); //keep the transmission and next
234 Wire.requestFrom(0x68,6,true); //We ask for next 6 registers starting with the 3B
235 /*We have asked for the 0x3B register. The IMU will send a burst of register.
236 * The amount of register to read is specify in the requestFrom function.
237 * In this case we request 6 registers. Each value of acceleration is made out of
238 * two 8bits registers, low values and high values. For that we request the 6 of them
239 * and just make then sum of each pair. For that we shift to the left the high values
240 * register (<<) and make an or (|) operation to add the low values.
241 If we read the datasheet, for a range of +-8g, we have to divide the raw values by 4096*/
242 Acc_rawX=(Wire.read()<<8|Wire.read())/4096.0 ; //each value needs two registres
243 Acc_rawY=(Wire.read()<<8|Wire.read())/4096.0 ;
244 Acc_rawZ=(Wire.read()<<8|Wire.read())/4096.0 ;
245 /*Now in order to obtain the Acc angles we use euler formula with acceleration values
246 after that we substract the error value found before*/
247 /*---X---*/
248 Acc_angle_x = (atan((Acc_rawY)/sqrt(pow((Acc_rawX),2) + pow((Acc_rawZ),2)))*rad_to_deg) - Acc_angle_error_x;
249 /*---Y---*/
250 Acc_angle_y = (atan(-1*(Acc_rawX)/sqrt(pow((Acc_rawY),2) + pow((Acc_rawZ),2)))*rad_to_deg) - Acc_angle_error_y;
251

```

Screenshot of code showing Accelerometer sensor reading and calculation

### 3.3.2 3 Axis Gyroscope

In contrast to the accelerometers, gyroscopes measure rotation. For decades, gyroscopes have been used as an integral part in naval and aerospace navigational systems. The first gyroscopes were generally spinning masses held in a gimbal frame. These frames allowed the mass to spin freely and maintain its axis of rotation regardless of the orientation of the external frame. Due to conservation of angular momentum, the spinning mass would resist any changes to its rotation. The structure is illustrated below



Classical Gyroscope or Gyrostat

To electronically measure changes in rotation using a classical gyroscope, electrical potentiometers are connected to the gimbal frames. While these devices are capable of a large degree of accuracy, they are large, heavy, and expensive.

```

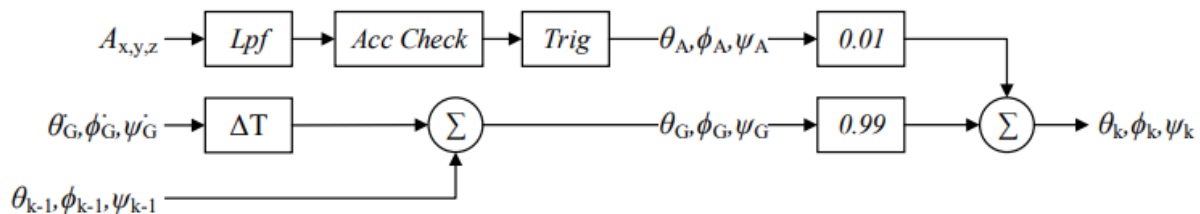
206
207 //////////////////////////////////////////////////Gyro read//////////////////////////////////////
208 Wire.beginTransaction(0x68);           //begin, Send the slave address (in this case 68)
209 Wire.write(0x43);                       //First address of the Gyro data
210 Wire.endTransmission(false);
211 Wire.requestFrom(0x68,4,true);          //We ask for just 4 registers
212 Gyr_rawX=Wire.read()<<8|Wire.read();    //Once again we shif and sum
213 Gyr_rawY=Wire.read()<<8|Wire.read();
214 /*Now in order to obtain the gyro data in degrees/seconds we have to divide first
215 the raw value by 32.8 because that's the value that the datasheet gives us for a 1000dps range*/
216 /*---X---*/
217 Gyr_rawX = (Gyr_rawX/32.8) - Gyro_raw_error_x;
218 /*---Y---*/
219 Gyr_rawY = (Gyr_rawY/32.8) - Gyro_raw_error_y;
220 /*Now we integrate the raw value in degrees per seconds in order to obtain the angle
221 * If you multiply degrees/seconds by seconds you obtain degrees */
222 /*---X---*/
223 Gyro_angle_x = Gyr_rawX*elapsedTime;
224 /*---X---*/
225 Gyro_angle_y = Gyr_rawY*elapsedTime;
226
227

```

Screenshot of code showing Gyroscope sensor reading and calculation

### 3.3.3 Sensor Fusion

The process of combining data from multiple sensors and coming up with a collective estimate is commonly called sensor fusion. For orientation, there are multiple different sensor fusion algorithms such as gradient descent and Kalman filtering. However, due to the high frequency vibrations, large system accelerations, low computational power available, and fast control loop update requirements, not all algorithms are ideal for quadcopter flight use. With this in mind, a simple complementary filter was selected. This work is shown in the figure below where subscript k represents the sample number.



Block Diagram showing Sensor Fusion

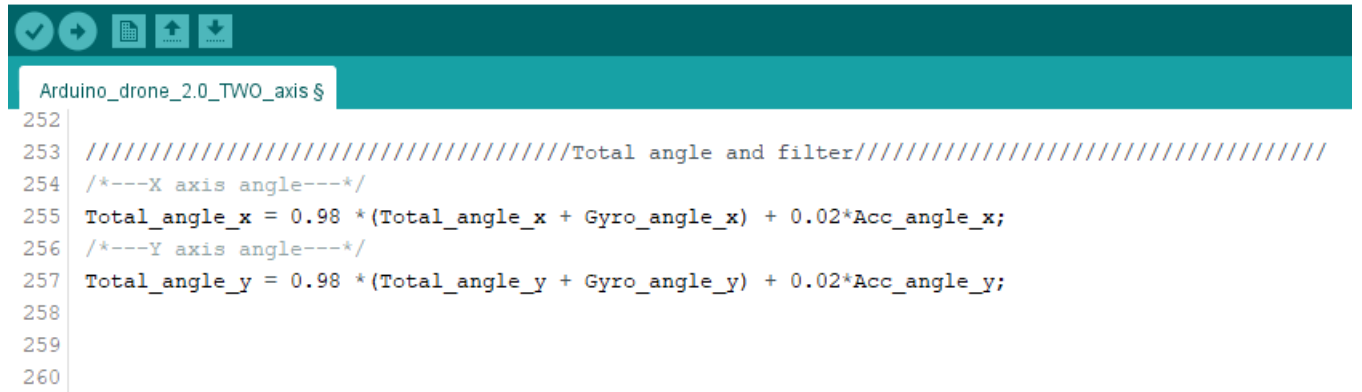


A block diagram representation of a basic complementary filter. Where  $A_{x,y,z}$  is the raw accelerometer data and  $G \cdot \theta, \phi, \psi$  is the raw gyroscope data. The weakness addressed by this filter are; while gyroscopes have good dynamic response and noise immunity, they have long term drift. In contrast, accelerometers have poor dynamic response but are not susceptible to drift in the same manner. Consequently, a high pass filter is used on the gyroscope data and a low pass filter is used on the accelerometer data. However, before the data is combined, it is important to verify validity of the accelerometer data. Also, before combining the gyroscope and accelerometer data, the accelerometer's data must be converted to orientation angles using equation 3.3 below

$$\theta = \text{atan}\left(\frac{A_x}{\sqrt{A_y^2 + A_z^2}}\right) \cdot \frac{180}{\pi}$$

$$\phi = \text{atan2}\left(\frac{A_y}{A_z}\right) \cdot \frac{180}{\pi}$$

(3.3)



```

252
253 //////////////////////////////////////////////////Total angle and filter////////////////////////////////////
254 /*---X axis angle---*/
255 Total_angle_x = 0.98 *(Total_angle_x + Gyro_angle_x) + 0.02*Acc_angle_x;
256 /*---Y axis angle---*/
257 Total_angle_y = 0.98 *(Total_angle_y + Gyro_angle_y) + 0.02*Acc_angle_y;
258
259
260

```

Screenshot of code showing Sensor fusion between Accelerometer and Gyroscope

### 3.4 Control System Implementation

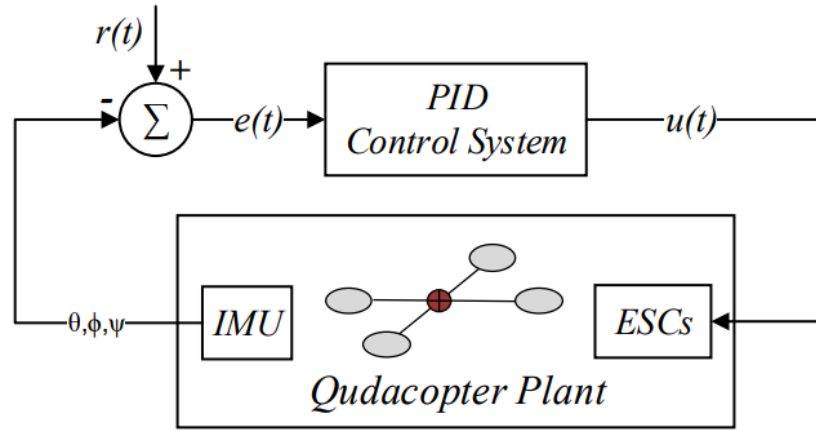
This section will overview the control system used to achieve stable flight and meet the control objectives. This control system was needed in order to handle real world disturbances and to account for unknown offsets. In order to drive the quadcopter's orientation ( $\theta, \phi, \psi$ ) to desired values, a series of proportional integral derivative (PID) controllers were implemented.

### 3.4.1 PID Basics

A PID controller is a control loop that updates based upon the error observed between a desired output and the measured output. This control loop's response is characterized by coefficients  $K_p$ ,  $K_i$ ,  $K_d$  and is shown in equation 3.3 where  $e$  represents error and  $u(t)$  represents the PID output.

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \quad (3.3)$$

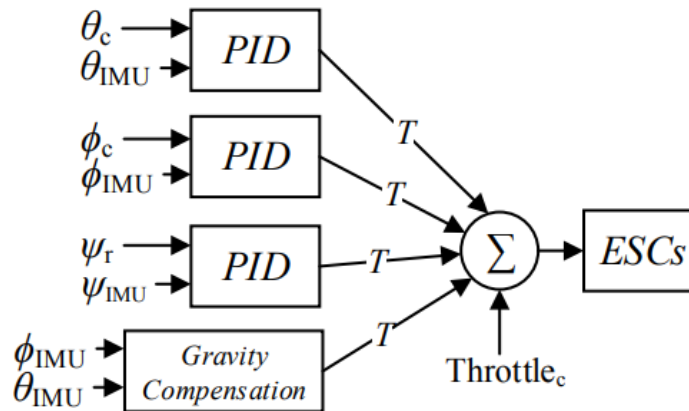
The magnitude of coefficients  $K_p$ ,  $K_i$ ,  $K_d$  determines how responsive the controller is to proportional, integral, and derivative errors respectively. Control loops such as PID act on systems to stabilize them if needed. In addition, a PID loop can improve the transient response of a system. Systems themselves in controls context are often referred to as a “plant”. In the case of this project, the plant is the quadcopter system. This plant along with control is visualized in the figure below, where  $u(t)$  is the input to the plant, where  $r(t)$  is the desired set point of the plant, and where  $e(t)$  is the error between the output of the plant and the desired set point.



Block Diagram of PID in Control Loop

The input to the plant,  $u(t)$ , represents the speed commanded to each of the quadcopter's motors. For simplicity,  $u(t)$  will be redefined to vector  $T$  where  $T = [T_0, T_1, T_2, T_3]$ .  $T_0, T_1, T_2, T_3$  represent the four individual motor thrust vectors.  $r(t)$ , the desired quadcopter outputs, is the user commanded orientation angles  $\theta_c, \phi_c, \psi_c$  as well as throttle  $T_c$ . Recall that these outputs were previously defined in equation 3.1. Note that this system is linearized at a current point in time and consequently assumes that superposition is upheld. It follows that the input to the system,  $u(t)$ , is calculated as a summation of the individual control

components. This control concept is shown in the figure below and the calculations for each block are demonstrated in section 3.4.2.



Block Diagram of PID Error Fusion

With the input and output relationship established, the following section overviews the PID code developed for the system as well as how T is updated such that the control goals are achieved.

### 3.4.2 PID Control Software



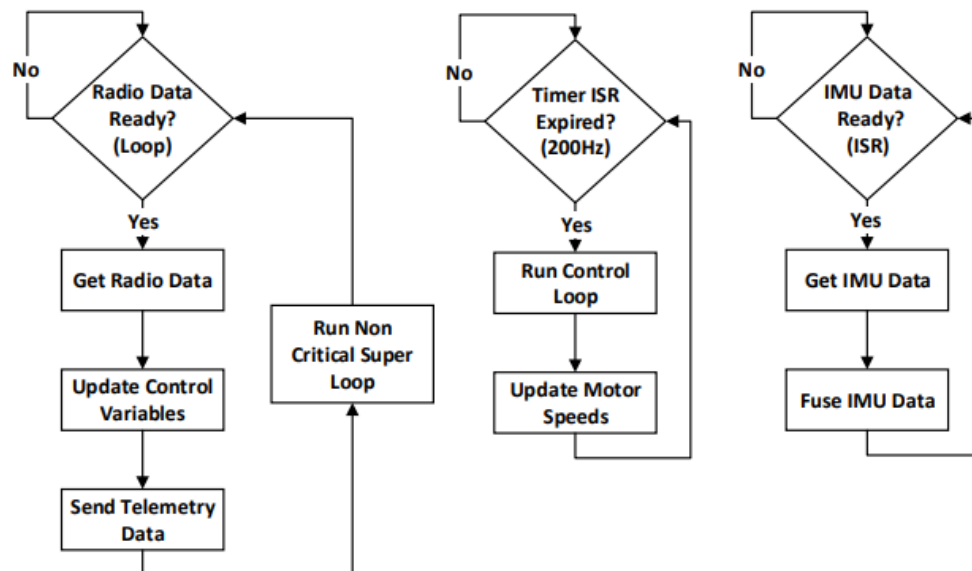
```
Arduino_drone_2.0_TWO_axis $
266
267 /*////////////////////////////////////P I D////////////////////////////////////*/
268 roll_desired_angle = map(input_ROLL,1000,2000,-10,10);
269 pitch_desired_angle = map(input_PITCH,1000,2000,-10,10);
270
271 /*First calculate the error between the desired angle and *the real measured angle*/
272 roll_error = Total_angle_y - roll_desired_angle;
273 pitch_error = Total_angle_x - pitch_desired_angle;
274 /*Next the proportional value of the PID is just a proportional constant *multiplied by the error*/
275 roll_pid_p = roll_kp*roll_error;
276 pitch_pid_p = pitch_kp*pitch_error;
277 /*The integral part should only act if we are close to the desired position but we want to fine
278 tune the error. That's why I've made a if operation for an error between -2 and 2 degree.
279 To integrate we just sum the previous integral value with the error multiplied by the integral
280 constant. This will integrate (increase) the value each loop till we reach the 0 point*/
281 if(-3 < roll_error <3)
282 {
283   roll_pid_i = roll_pid_i+(roll_ki*roll_error);
284 }
285 if(-3 < pitch_error <3)
286 {
287   pitch_pid_i = pitch_pid_i+(pitch_ki*pitch_error);
288 }
289
290 /*The last part is the derivate. The derivate acts upon the speed of the error. As we know the speed is the amount
291 of error that produced in a certain amount of time divided by that time. For taht we will use a variable called
292 previous_error. We substract that value from the actual error and divide all by the elapsed time. Finnaly we multiply
293 the result by the derivate constant*/
294 roll_pid_d = roll_kd*((roll_error - roll_previous_error)/elapsedTime);
295 pitch_pid_d = pitch_kd*((pitch_error - pitch_previous_error)/elapsedTime);
296 /*The final PID values is the sum of each of this 3 parts*/
297 roll_PID = roll_pid_p + roll_pid_i + roll_pid_d;
298 pitch_PID = pitch_pid_p + pitch_pid_i + pitch_pid_d;
299 /*We know taht the min value of PWM signal is 1000us and the max is 2000. So that tells us that the PID value can/s
300 oscilate more than -1000 and 1000 because when we have a value of 2000us the maximum value taht we could substract
301 is 1000 and when we have a value of 1000us for the PWM signal, the maximum value that we could add is 1000 to reach
302 the maximum 2000us. But we don't want to act over the entire range so +-400 should be enough*/
303 if(roll_PID < -400){roll_PID=-400;}
304 if(roll_PID > 400) {roll_PID=400; }
305 if(pitch_PID < -4000){pitch_PID=-4000;}
306 if(pitch_PID > 400) {pitch_PID=400;}
307
308 /*Finnaly we calculate the PWM width. We sum the desired throttle and the PID value*/
309 pwm_R_F = 115 + input_THROTTLE - roll_PID - pitch_PID;
310 pwm_R_B = 115 + input_THROTTLE - roll_PID + pitch_PID;
311 pwm_L_B = 115 + input_THROTTLE + roll_PID + pitch_PID;
312 pwm_L_F = 115 + input_THROTTLE + roll_PID - pitch_PID;
```

Screenshot showing PID Controller implementation in code

### 3.5 Software Architecture

The control loop along with the IMU code shown in the flowchart diagram below account for the majority of flight critical code running on the quadcopter. To keep control timing and IMU collection constant, the software was implemented in three separate threads. These threads primarily manage IMU filter updates, control system updates, and radio link updates. For this quadcopter software implementation, IMU data was received on a consistent interrupt basis. Gyro and accelerometer readings were updated and filtered at 500 Hz. The magnetometer was updated at 100 Hz which is the maximum for the selected device. These updates were performed based on an interrupt to give the IMU filter constant timing characteristics. Likewise, the control system was also triggered through interrupts. The control system was executed every 200 Hz and used the most up to date data from both the controller and the IMU filter to perform its calculations. Less timing critical tasks were then executed in a super loop such as checking for new radio data and other non-flight related features.

#### 3.4.1 Software Flowchart Diagram



Flowchart Diagram of Software Loop

### 3.6 Hardware Implementation

This section overviews the electronic hardware design of the quadcopter. Electronic hardware components of this system include the radio system, flight controller, and motor controllers. A picture of the quadcopter is shown below

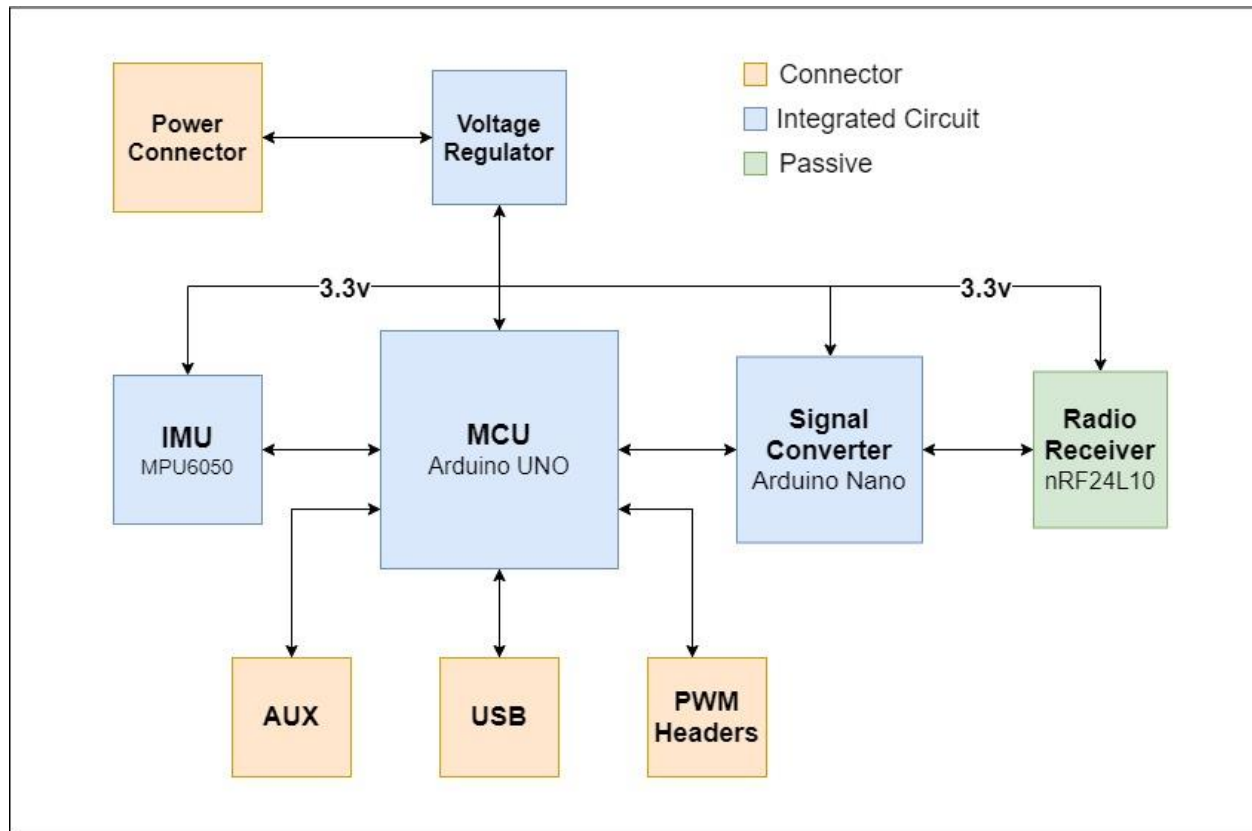


Image of Quadcopter

These pieces of hardware were designed to help understand the nature of the underlying system as well to be able to implement a more flexible architecture. Hardware involved in this project were selected separately from different manufacturers and are compatible with one another. The following sections overview the hardware design for the quadcopter.

#### 3.4.1 Flight Controller Hardware

The flight controller consists of a MCU (an Arduino UNO is used here), a IMU and Radio receiver. The flight controller is responsible for stabilizing the quadcopter and executing user control. To enable ease of use and easy implementation of the control software on cheap, easy to find hardware, the Arduino UNO was selected as the MCU. Below is a block diagram showing the connection between all elements which make up the flight control



Block Diagram Showing Components in Flight Control Hardware

The Arduino Uno is a microcontroller board based on the ATmega328P. It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz quartz crystal, a USB connection, a power jack, an ICSP header and a reset button. It contains everything needed to support the microcontroller; We simply connect it to a computer with a USB cable or power it with an AC-to-DC adapter or battery to get it started.

With regards to the IMU, the MPU-6050 by Adafruit was selected since it is a compact integrated solution with a 3-axis accelerometer and a gyroscope. Another advantage of the MPU-6050 is that it had already been flight tested in another multi-rotor platform.

The radio receiver used here is the nRF24L10 from Nordic Semiconductor. The nRF24L10 is a wireless transceiver module, meaning each module can both send as well as receive data. They operate in the frequency of 2.4GHz, which falls under the ISM band and hence it is legal to use in almost all countries for engineering applications. The modules when operated efficiently can cover a distance of 100 meters (200 feet) which makes it a great choice for all wireless remote-controlled projects. The module operates at 3.3V hence can be easily used with 3.2V systems or 5V systems. Each module has an address range of



125 and each module can communicate with 6 other modules hence it is possible to have multiple wireless units communicating with each other in a particular area. Hence mesh networks or other types of networks are possible using this module. Below is a photo showing the flight controller implementation

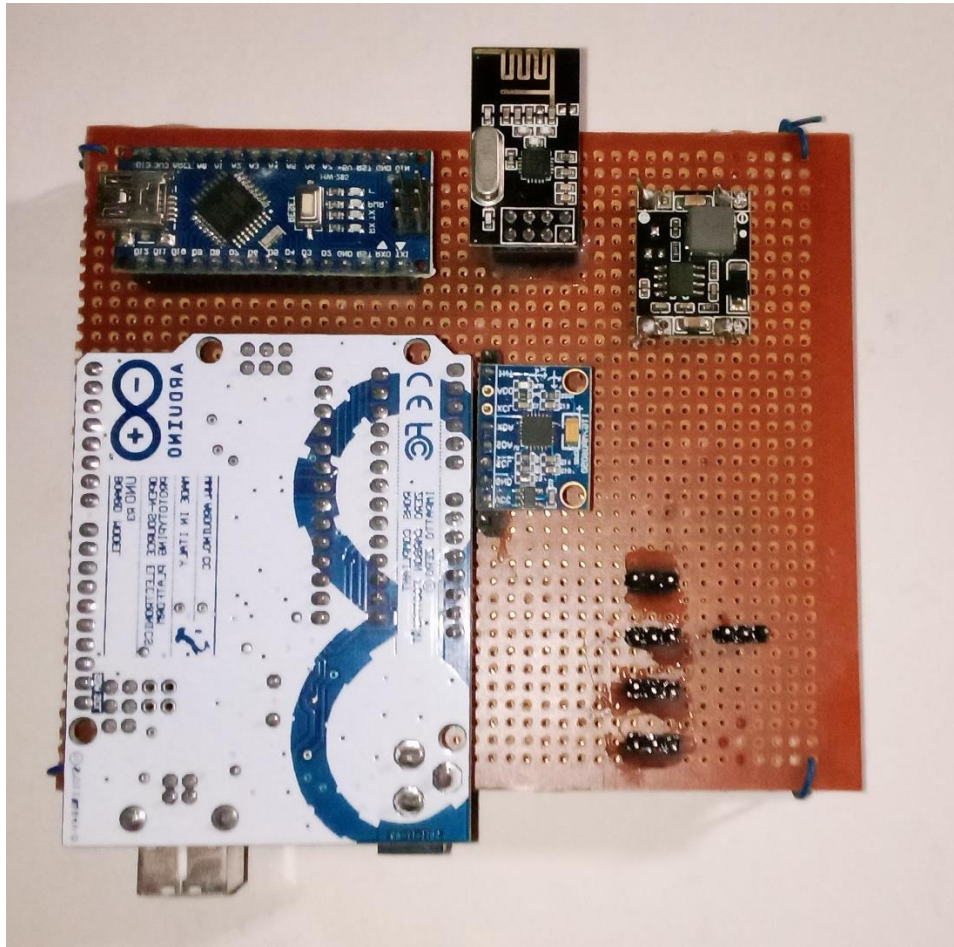


Image of Flight Controller Hardware

### 3.4.2 Remote Controller Hardware

The remote controller hardware consists of two joysticks, two toggle switches, two potentiometer switches, a micro-controller (an Arduino Nano in this case) with a USB interface, a radio receiver (nRF24L10 low noise – power amplifier), 3.3v voltage regulator and a 9v battery. A system level block diagram for the remote is shown in the figure below. This diagram illustrates all the major hardware



blocks utilized by the controller. Further information can be derived from the source code as well as the schematic.

For the joysticks, two Gimbals were selected. These joysticks were selected since they were primarily designed for remote flight control and allow for easy integration with an Arduino. To simplify the mechanical design, these joysticks mount directly to the Veroboard which avoids requiring an injection molded or 3D printed enclosure. With regards to the microcontroller, an Arduino Nano was selected due to its low cost, availability, and USB interface. The radio transmitter used here is the same as that used in the flight controller which is the nRF24L10 except its variant is built for low noise and power amplification. Toggle switches were added for flight commands such as emergency cutoff and for extra commands to be added later. Two potentiometer switches were also added for alternative throttle control, LEDs were also added to indicate power supply to the board. Finally, for remote portability, a 1 cell 9V battery was used such that the remote could be operated independently of a computer. In addition, a 3.3v voltage regulator is also present in order to step down the 9v from the battery to 3.3v, suitable for the radio receiver. A picture of the assembled controller is illustrated below.

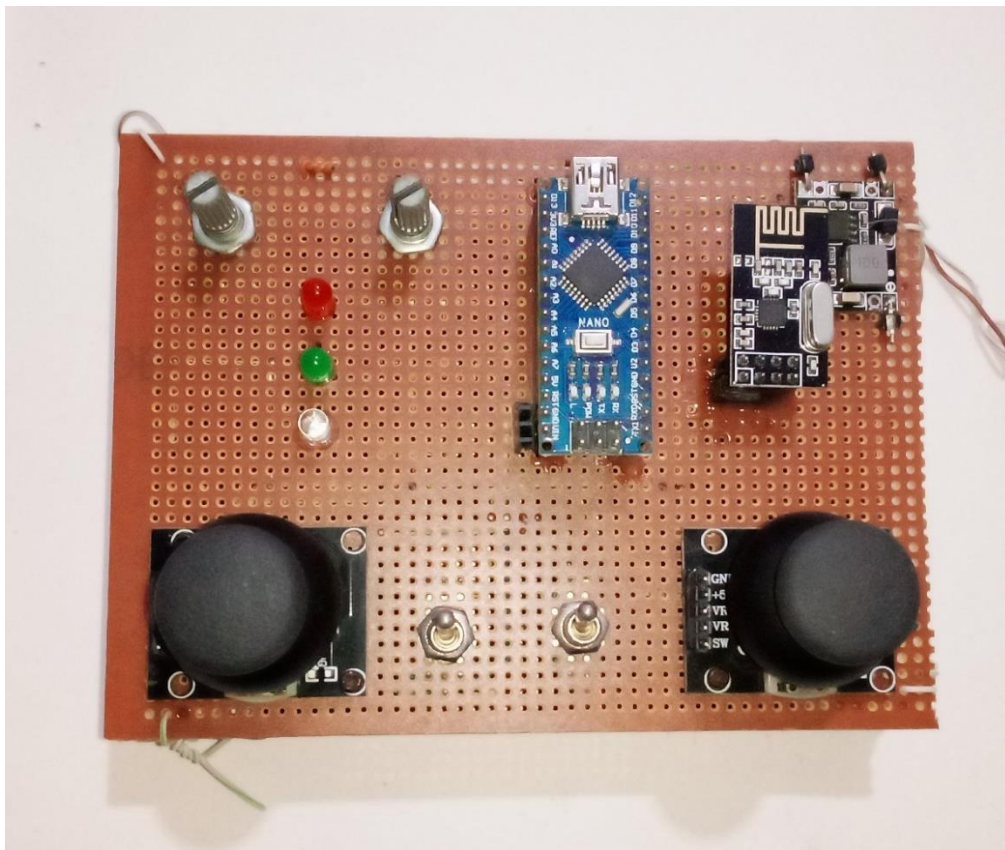
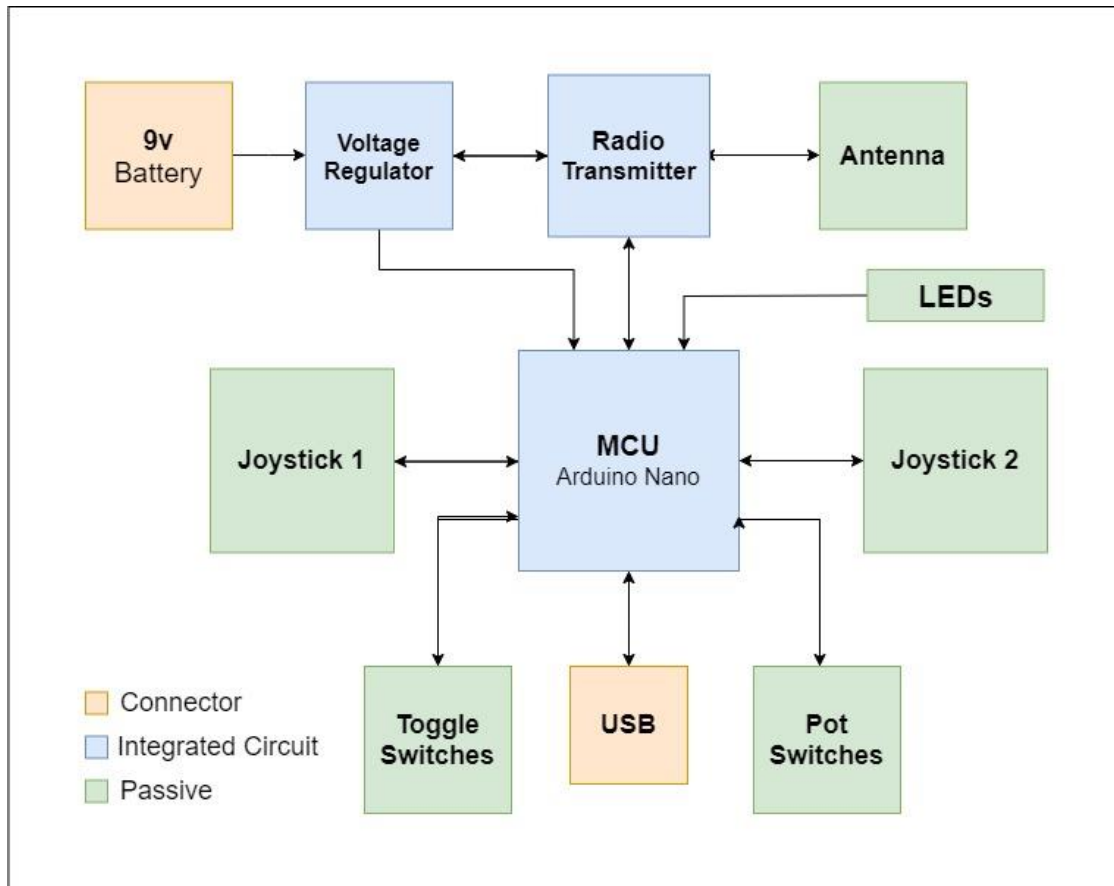


Image of Drone Remote Controller



Block Diagram of Remote Controller

### 3.4.3 Electronic Speed Controller Hardware

The brushless sensorless motor controllers, or ESCs, are made up primarily of a microcontroller, gate drivers, and high-power FETs. The MCU selected was the Cortex M0+ KL25Z128 Kinetis by Freescale. The gate driver selected was the ADP3110 by ON Semiconductor since it supported dual high side low side NFET driving, supported 3.3V logic signals, is low cost, and has an easy to solder package. The primary switching FET selected was the PSMN011 by NXP since it supports a max drain current of 61A, has an RDS(on) of 10 mΩ, and since it utilizes a small QFN package. This ESC was designed to be compact and have capacity to drive at least 10A of current. To achieve this, double sided population was required.

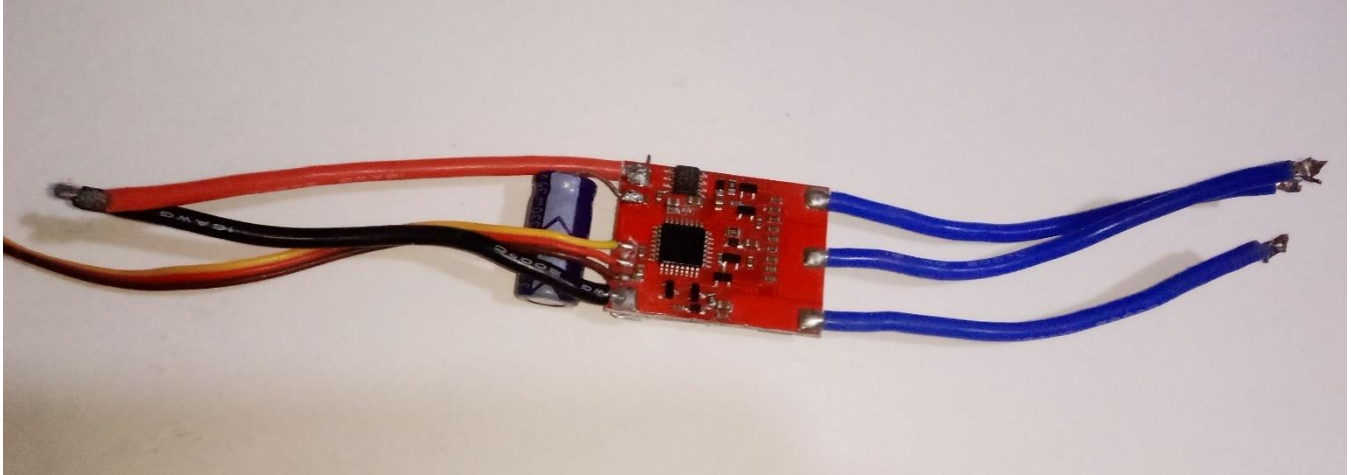


Image of Electronic Speed Controller

#### **3.4.4 Brushless Motor**

Brushless motors are synchronous 3 phase permanent magnet motors. Like a DC brushed motor, brushless motor consists of a stator which contains coils and a rotor which contains permanent magnets. If the coils are energized correctly, the coils create a rotational torque that acts on the magnets on the rotor. Since the magnets are adhered to rotor, this rotational torque causes the motor shaft to spin. However, if the coils on the stator are not energized correctly with respect to the polarity and position of the magnets, rotational torque is not generated. Consequently, knowledge of the rotors position is required in order to properly drive a DC motor. For brushed motors, a mechanical device called a brush is used such that stator coils are always properly energized with respect to the rotor. However, with a brushless motor, this mechanical linkage from stator to the rotor is removed in order to reduce friction and extend motor lifetime. Without the mechanical linkage between the rotor and the stator, external sensing methods of the rotor are required. Hall effect sensor can be used in order to detect the rotors position. However, hall effect sensors are expensive and require additional wires as well as mechanical protection. Another method of driving brushless motors is with sensor-less techniques such as back electromotive force (BEMF) zero cross detection.



Image of Brushless DC Motor

# CHAPTER 4

## EXPECTED RESULTS

### 4.1 Result and Outcome

For safety and verification purposes, experimental setup is designed to examine the controller response before applying it to a real flight test. An experimental testbench was used. On the test bench, the quadcopter is held firmly from the top and the bottom allowing only rotational motion, i.e., three degrees of freedom. Various reference signals are sent via RC to explore the system reference tracking.

At the end of testing, as expected, the multirotor system (Quadcopter in this case) was controlled by the electronic Remote controller using RF signal. The flight controller was able to control roll, pitch, yaw, altitude and motion in x or y direction. As expected, the multirotor system balanced itself using commands in the control software written to its microcontroller, when there were no changes to inputs sent from the remote controller.

### 4.2 Bill of Engineering Measurement and Evaluation

Component	Specification	Number	Cost
Frame	DJI F450 Quadcopter Frame	1	8000
Propellers/Rotors	10x4.5 Propellers	4	4 x 1500
Motors	A2212 BLDC motors	4	4 x 3500
ESCs	Hobby King 30A Brushless ESC	4	4 x 3000
Battery	11.1 V Li-po Battery	1	8000

Transmitter	NRF24L10 PA(power amplifier) LNA(low noise amplifier)	1	3500
Receiver	NRF24L10	1	3000
Inertial Measurement Unit (IMU)	MPU 6050 6dof IMU	1	2400
Arduino UNO	Arduino UNO Artmega MCU	1	8000
Arduino Nano	Arduino Nano, Atmega MCU	2	4200
Vero Board	Dotted Copper Vero Board	2	2 x 400
Joystick	2 Axis Joystick Potentiometer	2	2 x 3500
Other Components for soldering, connections, testing and binding	Switches, LEDs, Potentiometers, header connectors, connecting wires, Glue, solder, Battery cap	-	3500
Shipping and Delivery fee	-	-	7000
<b>TOTAL</b>			87,400

# CHAPTER 5

## CONCLUSION

### 5.1 Overview

The main purpose of this project was to develop both the hardware and software for a quadcopter system. We developed a control update laws as well as control objectives, we developed a 6 degrees of freedom fusion filter to provide accurate estimations of orientation, we demonstrated a control system to achieve stable auto leveling flight using a series of PID controllers. We then overviewed the RF hardware involved in creating the transceiving link between the remote and the quadcopter. We specified other hardware involved in the system such as the flight controller and electronic speed controllers. Finally, the conclusion demonstrated flight results and performance of the overall system. With regards to contributions, this project provides reference source files and multi-rotor design insights. Specifically, this project provides open-source software / hardware files for a functional stabilized auto-leveling flight controller. It also provides open-source software / hardware files for a remote controller. It offers design insights into achieving multi-rotor flight with a 6 degree of freedom orientation fusion algorithm as well as a PID control system. Finally,

### 5.2 Future Work

This section describes potential follow-up work to the current Drone Design. Currently, the IMU filter operates on Euler angles when ideally it should be based upon quaternions to avoid singularities. In addition, quaternions also allow for true linear spherical interpolation where as low pass filtering Euler angles does not produce a linear response. With regards to the control system, the first thing that would be improved is the PID control system performance. The current balancing action is somewhat noisy and the yaw control is lacking. In addition to PID control, other control techniques could be implemented such as linear quadratic regulator (LQR) as well as model predictive control (MPC). Another improvement would be to integrate the designed electronic speed controllers into quadcopter system. A final improvement would be the addition of an onboard GPS sensor and a Lidar sensor to facilitate autonomous navigation in both indoor and outdoor environments; addition of a camera to enable the system to be used for aerial surveillance or for gathering data.

# REFERENCES

- A. Bry, A. Bachrach, and N. Roy (2012) “State estimation for aggressive flight in gps-denied environments using onboard sensing”, in IEEE International Conference on Robotics and Automation, pp. 19–25
- C. Hubert (2002), Electric Machines, 2nd edition.
- Clean Flight, “Clean Flight” (2016), [Online]. Available at:  
<http://cleanflight.com>
- DJI (2015), “Inspire 1”, [Online]. Available at:  
<http://www.dji.com/product/inspire-1>
- G Lazaridis (2011), “Pulse Position Modulation and Differential PPM”, [Online]. Available at:  
[http://www.pcbheaven.com/wikipages/Pulse\\_Position\\_Modulation/](http://www.pcbheaven.com/wikipages/Pulse_Position_Modulation/)
- G. Raffo, M. Ortega, and F. Rubio (2013) “An integral predictive/nonlinear h infinity control structure for a quadrotor helicopter”, in Automatica, pp. 1–7.
- J. Zhang and S. Singh (2014), “Loam: Lidar odometry and mapping in real-time”, in Robotics: Science and Systems Conference.
- L. Argentim, W. Contrimas, P. Santos, and R. Aguiar (2013) “Lqr and lqr-pid on a quadcopter platform”,  
in IEEE Int.Conference on Electronics, Informatics and Vision.
- M. Mueller and D. Raffaello (2013) “A model predictive controller for quadrocopter state interception”, in ControlConference (ECC), European. IEEE, pp. 1383–1389.
- NXP (2016), “3-Phase BLDC Motor Control with Sensorless Back EMF Zero Crossing Detection Using 56F80x”, [Online]. Available: <https://cache.freescale.com/files/product/doc/AN1914.pdf>
- NXP (2016), “Sensorless PMSM Field-Oriented Control”, [Online]. Available at:  
<http://www.nxp.com/doc/DRM148>.
- Open Pilot, “Open Pilot” (2016), [Online]. Available at:  
<https://www.openpilot.org>
- Pixhawk (2016), “PIXHAWK is the all-in-one unit, combining FMU and IO into a single package”, [Online]. Available at: <https://pixhawk.org/>.
- R. Mahony, V. Kumar, and P. Corke (2012) “Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor”, IEEE Robotics Automation Magazine, vol. 19, no. 3, pp. 20–32.
- S. Lupashin, A. Schoellig, M. Sherback, and R. D’Andrea (2010) “A simple learning strategy for high speed quadrocopter multi-flips”, in Robotics and Automation (ICRA), IEEE International Conference on, pp. 1642–1648.
- S. Winkvist (2013), “Low computational SLAM for an autonomous indoor aerial inspection vehicle”, Master’s thesis, University of Warwick, the Netherlands
- Tau Labs, “Sparky 2” (2016), [Online]. Available at:  
<https://github.com/TauLabs/TauLabs/wiki/Sparky2>
- Z. Zhang (1994), “Iterative point matching for registration of free-form curves and surfaces”, in Int. Journal of Computer Vision. pp. 19–25, IEEE.



# APPENDIX

## Flight Controller Arduino Code

```
/*
 * Arduino pin | MPU6050
 * 5V          | Vcc
 * GND         | GND
 * A4          | SDA
 * A5          | SCL
 *
 * F_Left__motor | D4
 * F_Right__motor| D7
 * B_Left__motor | D5
 * B_Right__motor | D6
 */
#include <Wire.h>
#include <Servo.h>

Servo L_F_prop;
Servo L_B_prop;
Servo R_F_prop;
Servo R_B_prop;

//We create variables for the time width values of each PWM input signal
unsigned long counter_1, counter_2, counter_3, counter_4, current_count;

//We create 4 variables to store the previous value of the input signal (if LOW or HIGH)
```

```

byte last_CH1_state, last_CH2_state, last_CH3_state, last_CH4_state;

//To store the 1000us to 2000us value we create variables and store each channel
int input_YAW;    //In my case channel 4 of the receiver and pin D12 of arduino
int input_PITCH;  //In my case channel 2 of the receiver and pin D9 of arduino
int input_ROLL;   //In my case channel 1 of the receiver and pin D8 of arduino
int input_THROTTLE; //In my case channel 3 of the receiver and pin D10 of arduino

/*MPU-6050 gives you 16 bits data so you have to create some float constants
*to store the data for accelerations and gyro*/

//Gyro Variables
float elapsedTime, time, timePrev;    //Variables for time control
int gyro_error=0;                     //We use this variable to only calculate once the gyro data error
float Gyr_rawX, Gyr_rawY, Gyr_rawZ;   //Here we store the raw data read
float Gyro_angle_x, Gyro_angle_y;     //Here we store the angle value obtained with Gyro data
float Gyro_raw_error_x, Gyro_raw_error_y; //Here we store the initial gyro data error

//Acc Variables
int acc_error=0;                      //We use this variable to only calculate once the Acc data error
float rad_to_deg = 180/3.141592654;   //This value is for passing from radians to degrees values
float Acc_rawX, Acc_rawY, Acc_rawZ;   //Here we store the raw data read
float Acc_angle_x, Acc_angle_y;       //Here we store the angle value obtained with Acc data
float Acc_angle_error_x, Acc_angle_error_y; //Here we store the initial Acc data error

float Total_angle_x, Total_angle_y;

//More variables for the code

```

```

int i;

int mot_activated=0;

long activate_count=0;

long des_activate_count=0;

//////////////////////////////////PID FOR ROLL//////////////////////////////////

float roll_PID, pwm_L_F, pwm_L_B, pwm_R_F, pwm_R_B, roll_error, roll_previous_error;

float roll_pid_p=0;

float roll_pid_i=0;

float roll_pid_d=0;

//////////////////////////////////ROLL PID CONSTANTS//////////////////////////////////

double roll_kp=0.7;//3.55

double roll_ki=0.006;//0.003

double roll_kd=1.2;//2.05

float roll_desired_angle = 0;  //This is the angle in which we want the

//////////////////////////////////PID FOR PITCH//////////////////////////////////

float pitch_PID, pitch_error, pitch_previous_error;

float pitch_pid_p=0;

float pitch_pid_i=0;

float pitch_pid_d=0;

//////////////////////////////////PITCH PID CONSTANTS//////////////////////////////////

double pitch_kp=0.72;//3.55

double pitch_ki=0.006;//0.003

double pitch_kd=1.22;//2.05

float pitch_desired_angle = 0;  //This is the angle in which we want the

```

```

void setup() {

    PCICR |= (1 << PCIE0); //enable PCMSK0 scan
    PCMSK0 |= (1 << PCINT0); //Set pin D8 trigger an interrupt on state change.
    PCMSK0 |= (1 << PCINT1); //Set pin D9 trigger an interrupt on state change.
    PCMSK0 |= (1 << PCINT2); //Set pin D10 trigger an interrupt on state change.
    PCMSK0 |= (1 << PCINT4); //Set pin D12 trigger an interrupt on state change.


    DDRB |= B00100000; //D13 as output
    PORTB &= B11011111; //D13 set to LOW


    L_F_prop.attach(4); //left front motor
    L_B_prop.attach(5); //left back motor
    R_F_prop.attach(7); //right front motor
    R_B_prop.attach(6); //right back motor

    /*in order to make sure that the ESCs won't enter into config mode
    *I send a 1000us pulse to each ESC.*/
    L_F_prop.writeMicroseconds(1000);
    L_B_prop.writeMicroseconds(1000);
    R_F_prop.writeMicroseconds(1000);
    R_B_prop.writeMicroseconds(1000);


    Wire.begin(); //begin the wire comunication
    Wire.beginTransmission(0x68); //begin, Send the slave adress (in this case 68)
    Wire.write(0x6B); //make the reset (place a 0 into the 6B register)
    Wire.write(0x00);
    Wire.endTransmission(true); //end the transmission

```

```

Wire.beginTransmission(0x68);    //begin, Send the slave adress (in this case 68)
Wire.write(0x1B);                //We want to write to the GYRO_CONFIG register (1B hex)
Wire.write(0x10);                //Set the register bits as 00010000 (100dps full scale)
Wire.endTransmission(true);      //End the transmission with the gyro


Wire.beginTransmission(0x68);    //Start communication with the address found during search.
Wire.write(0x1C);                //We want to write to the ACCEL_CONFIG register (1A hex)
Wire.write(0x10);                //Set the register bits as 00010000 (+/- 8g full scale range)
Wire.endTransmission(true);


Serial.begin(9600);
delay(1000);
time = millis();                //Start counting time in milliseconds


/*Here we calculate the gyro data error before we start the loop
* I make the mean of 200 values, that should be enough*/
if(gyro_error==0)
{
  for(int i=0; i<200; i++)
  {
    Wire.beginTransmission(0x68);    //begin, Send the slave adress (in this case 68)
    Wire.write(0x43);                //First adress of the Gyro data
    Wire.endTransmission(false);
    Wire.requestFrom(0x68,4,true);    //We ask for just 4 registers


    Gyr_rawX=Wire.read()<<8|Wire.read(); //Once again we shif and sum
    Gyr_rawY=Wire.read()<<8|Wire.read();
  }
}

```



```

/*---X---*/

Acc_angle_error_x = Acc_angle_error_x + ((atan((Acc_rawY)/sqrt(pow((Acc_rawX),2) +
pow((Acc_rawZ),2))))*rad_to_deg));

/*---Y---*/

Acc_angle_error_y = Acc_angle_error_y + ((atan(-1*(Acc_rawX)/sqrt(pow((Acc_rawY),2) +
pow((Acc_rawZ),2))))*rad_to_deg));

if(a==199)
{
    Acc_angle_error_x = Acc_angle_error_x/200;
    Acc_angle_error_y = Acc_angle_error_y/200;
    acc_error=1;
}
}

} //end of acc error calculation
} //end of setup loop

void loop() {

//////////////////////////////////IMU//////////////////////////////////

timePrev = time; // the previous time is stored before the actual time read
time = millis(); // actual time read
elapsedTime = (time - timePrev) / 1000;

/*The timeStep is the time that elapsed since the previous loop.
*This is the value that we will use in the formulas as "elapsedTime"
*in seconds. We work in ms so we have to divide the value by 1000

```

```

to obtain seconds*/

/*Reed the values that the accelerometer gives.

* We know that the slave adress for this IMU is 0x68 in
* hexadecimal. For that in the RequestFrom and the
* begin functions we have to put this value.*/

//////////////////////////////////Gyro read//////////////////////////////////
Wire.beginTransmission(0x68);      //begin, Send the slave adress (in this case 68)
Wire.write(0x43);                  //First adress of the Gyro data
Wire.endTransmission(false);
Wire.requestFrom(0x68,4,true);     //We ask for just 4 registers
Gyr_rawX=Wire.read()<<8|Wire.read(); //Once again we shif and sum
Gyr_rawY=Wire.read()<<8|Wire.read();

/*Now in order to obtain the gyro data in degrees/seconds we have to divide first
the raw value by 32.8 because that's the value that the datasheet gives us for a 1000dps range*/

/*---X---*/
Gyr_rawX = (Gyr_rawX/32.8) - Gyro_raw_error_x;

/*---Y---*/
Gyr_rawY = (Gyr_rawY/32.8) - Gyro_raw_error_y;

/*Now we integrate the raw value in degrees per seconds in order to obtain the angle
* If you multiply degrees/seconds by seconds you obtain degrees */

/*---X---*/
Gyro_angle_x = Gyr_rawX*elapsedTime;

/*---X---*/
Gyro_angle_y = Gyr_rawY*elapsedTime;

```



```

////////////////////////////////////////Acc read////////////////////////////////////////
Wire.beginTransmission(0x68); //begin, Send the slave address (in this case 68)
Wire.write(0x3B); //Ask for the 0x3B register- correspond to AcX
Wire.endTransmission(false); //keep the transmission and next
Wire.requestFrom(0x68,6,true); //We ask for next 6 registers starting withj the 3B
/*We have asked for the 0x3B register. The IMU will send a burst of register.
* The amount of register to read is specify in the requestFrom function.
* In this case we request 6 registers. Each value of acceleration is made out of
* two 8bits registers, low values and high values. For that we request the 6 of them
* and just make then sum of each pair. For that we shift to the left the high values
* register (<<) and make an or (|) operation to add the low values.
If we read the datasheet, for a range of+-8g, we have to divide the raw values by 4096*/
Acc_rawX=(Wire.read()<<8|Wire.read())/4096.0 ; //each value needs two registres
Acc_rawY=(Wire.read()<<8|Wire.read())/4096.0 ;
Acc_rawZ=(Wire.read()<<8|Wire.read())/4096.0 ;
/*Now in order to obtain the Acc angles we use euler formula with acceleration values
after that we subtract the error value found before*/
/*---X---*/
Acc_angle_x = (atan((Acc_rawY)/sqrt(pow((Acc_rawX),2) + pow((Acc_rawZ),2)))*rad_to_deg) -
Acc_angle_error_x;
/*---Y---*/
Acc_angle_y = (atan(-1*(Acc_rawX)/sqrt(pow((Acc_rawY),2) + pow((Acc_rawZ),2)))*rad_to_deg) -
Acc_angle_error_y;

////////////////////////////////////////Total angle and filter////////////////////////////////////////
/*---X axis angle---*/
Total_angle_x = 0.98 *(Total_angle_x + Gyro_angle_x) + 0.02*Acc_angle_x;
/*---Y axis angle---*/

```

```
Total_angle_y = 0.98 *(Total_angle_y + Gyro_angle_y) + 0.02*Acc_angle_y;
```

```
/*////////////////////////////////P I D////////////////////////////////*/
```

```
roll_desired_angle = map(input_ROLL,1000,2000,-10,10);
```

```
pitch_desired_angle = map(input_PITCH,1000,2000,-10,10);
```

```
/*First calculate the error between the desired angle and  
*the real measured angle*/
```

```
roll_error = Total_angle_y - roll_desired_angle;
```

```
pitch_error = Total_angle_x - pitch_desired_angle;
```

```
/*Next the proportional value of the PID is just a proportional constant  
*multiplied by the error*/
```

```
roll_pid_p = roll_kp*roll_error;
```

```
pitch_pid_p = pitch_kp*pitch_error;
```

```
/*The integral part should only act if we are close to the  
desired position but we want to fine tune the error. That's  
why I've made a if operation for an error between -2 and 2 degree.
```

```
To integrate we just sum the previous integral value with the  
error multiplied by the integral constant. This will integrate (increase)  
the value each loop till we reach the 0 point*/
```

```
if(-3 < roll_error <3)
```

```
{
```

```
roll_pid_i = roll_pid_i+(roll_ki*roll_error);
```

```

}

if(-3 < pitch_error <3)
{
    pitch_pid_i = pitch_pid_i+(pitch_ki*pitch_error);
}

/*The last part is the derivate. The derivate acts upon the speed of the error.
As we know the speed is the amount of error that produced in a certain amount of
time divided by that time. For taht we will use a variable called previous_error.
We substract that value from the actual error and divide all by the elapsed time.
Finnaly we multiply the result by the derivate constant*/
roll_pid_d = roll_kd*((roll_error - roll_previous_error)/elapsedTime);
pitch_pid_d = pitch_kd*((pitch_error - pitch_previous_error)/elapsedTime);
/*The final PID values is the sum of each of this 3 parts*/
roll_PID = roll_pid_p + roll_pid_i + roll_pid_d;
pitch_PID = pitch_pid_p + pitch_pid_i + pitch_pid_d;

/*We know taht the min value of PWM signal is 1000us and the max is 2000. So that
tells us that the PID value can/s oscilate more than -1000 and 1000 because when we
have a value of 2000us the maximum value taht we could substract is 1000 and when
we have a value of 1000us for the PWM signal, the maximum value that we could add is 1000
to reach the maximum 2000us. But we don't want to act over the entire range so -+400 should be enough*/
if(roll_PID < -400){roll_PID=-400;}
if(roll_PID > 400) {roll_PID=400; }
if(pitch_PID < -4000){pitch_PID=-400;}
if(pitch_PID > 400) {pitch_PID=400;}

/*Finnaly we calculate the PWM width. We sum the desired throttle and the PID value*/
pwm_R_F = 115 + input_THROTTLE - roll_PID - pitch_PID;
pwm_R_B = 115 + input_THROTTLE - roll_PID + pitch_PID;
pwm_L_B = 115 + input_THROTTLE + roll_PID + pitch_PID;

```

```
pwm_L_F = 115 + input_THROTTLE + roll_PID - pitch_PID;
```

```
/*Once again we map the PWM values to be sure that we won't pass the min  
and max values. Yes, we've already mapped the PID values. But for example, for  
throttle value of 1300, if we sum the max PID value we would have 2300us and  
that will mess up the ESC.*/
```

```
//Right front
```

```
if(pwm_R_F < 1100)
```

```
{  
    pwm_R_F= 1100;  
}
```

```
if(pwm_R_F > 2000)
```

```
{  
    pwm_R_F=2000;  
}
```

```
//Left front
```

```
if(pwm_L_F < 1100)
```

```
{  
    pwm_L_F= 1100;  
}
```

```
if(pwm_L_F > 2000)
```

```
{  
    pwm_L_F=2000;  
}
```

```
//Right back
```

```
if(pwm_R_B < 1100)
{
    pwm_R_B= 1100;
}
```

```
if(pwm_R_B > 2000)
{
    pwm_R_B=2000;
}
```

```
//Left back
```

```
if(pwm_L_B < 1100)
{
    pwm_L_B= 1100;
}
```

```
if(pwm_L_B > 2000)
{
    pwm_L_B=2000;
}
```

```
roll_previous_error = roll_error; //Remember to store the previous error.
```

```
pitch_previous_error = pitch_error; //Remember to store the previous error.
```

```
/*
```

```
Serial.print("RF: ");
```

```
Serial.print(pwm_R_F);
```

```
Serial.print(" | ");
```

```
Serial.print("RB: ");
```

```
Serial.print(pwm_R_B);
```

```
Serial.print(" | ");
```

```

Serial.print("LB: ");
Serial.print(pwm_L_B);
Serial.print(" | ");
Serial.print("LF: ");
Serial.print(pwm_L_F);

```

```

Serial.print(" | ");
Serial.print("X°: ");
Serial.print(Total_angle_x);
Serial.print(" | ");
Serial.print("Y°: ");
Serial.print(Total_angle_y);
Serial.println(" ");
*/

```

```

/*now we can write the values PWM to the ESCs only if the motor is activated

```

```

*/
if(mot_activated)
{
L_F_prop.writeMicroseconds(pwm_L_F);
L_B_prop.writeMicroseconds(pwm_L_B);
R_F_prop.writeMicroseconds(pwm_R_F);
R_B_prop.writeMicroseconds(pwm_R_B);
}
if(!mot_activated)
{
L_F_prop.writeMicroseconds(1000);
L_B_prop.writeMicroseconds(1000);
R_F_prop.writeMicroseconds(1000);

```

```

    R_B_prop.writeMicroseconds(1000);
}
if(input_THROTTLE < 1100 && input_YAW > 1800 && !mot_activated)
{
    if(activate_count==200)
    {
        mot_activated=1;
        PORTB |= B00100000; //D13 LOW
    }
    activate_count=activate_count+1;
}
if(!(input_THROTTLE < 1100 && input_YAW > 1800) && !mot_activated)
{
    activate_count=0;
}
if(input_THROTTLE < 1100 && input_YAW < 1100 && mot_activated)
{
    if(des_activate_count==300)
    {
        mot_activated=0;
        PORTB &= B11011111; //D13 LOW
    }
    des_activate_count=des_activate_count+1;
}
if(!(input_THROTTLE < 1100 && input_YAW < 1100) && mot_activated)
{
    des_activate_count=0;
}
}

```

```

ISR(PCINT0_vect){
//First we take the current count value in micro seconds using the micros() function
current_count = micros();

////////////////////////Channel 1

if(PINB & B00000001){          //We make an AND with the pin state register, We verify if pin 8 is
HIGH???

    if(last_CH1_state == 0){      //If the last state was 0, then we have a state change...
        last_CH1_state = 1;      //Store the current state into the last state for the next loop
        counter_1 = current_count; //Set counter_1 to current value.
    }
}

else if(last_CH1_state == 1){    //If pin 8 is LOW and the last state was HIGH then we have a state
change
    last_CH1_state = 0;          //Store the current state into the last state for the next loop
    input_ROLL = current_count - counter_1; //We make the time difference. Channel 1 is current_time -
timer_1.
}

////////////////////////Channel 2

if(PINB & B00000010 ){          //pin D9 -- B00000010
    if(last_CH2_state == 0){
        last_CH2_state = 1;
        counter_2 = current_count;
    }
}

else if(last_CH2_state == 1){
    last_CH2_state = 0;
    input_PITCH = current_count - counter_2;
}

////////////////////////Channel 3

```



```

if(PINB & B00000100 ){                //pin D10 - B00000100
    if(last_CH3_state == 0){
        last_CH3_state = 1;
        counter_3 = current_count;
    }
}
else if(last_CH3_state == 1){
    last_CH3_state = 0;
    input_THROTTLE = current_count - counter_3;
}

////////////////////////////////////////Channel 4
if(PINB & B00010000 ){                //pin D12 -- B00010000
    if(last_CH4_state == 0){
        last_CH4_state = 1;
        counter_4 = current_count;
    }
}
else if(last_CH4_state == 1){
    last_CH4_state = 0;
    input_YAW = current_count - counter_4;
}
}

```

## Radio Transmitter Arduino Code

```
/*A basic 4 channel transmitter using the nRF24L01 module.
*
*/
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

/*Create a unique pipe out. The receiver has to wear the same unique code*/
const uint64_t pipeOut = 0xE8E8F0F0E1LL; //IMPORTANT: The same as in the receiver
RF24 radio(9, 10); // select CSN pin

// The sizeof this struct should not exceed 32 bytes
// This gives us up to 32 8 bits channels
struct MyData {
    byte throttle;
    byte yaw;
    byte pitch;
    byte roll;
    byte AUX1;
    byte AUX2;
};

MyData data;
void resetData()
{
    //This are the start values of each channel
    // Throttle is 0 in order to stop the motors
```

```

//127 is the middle value of the 10ADC.
data.throttle = 0;
data.yaw = 127;
data.pitch = 127;
data.roll = 127;
data.AUX1 = 0;
data.AUX2 = 0;
}

void setup()
{
  //Start everything up
  radio.begin();
  radio.setAutoAck(false);
  radio.setDataRate(RF24_250KBPS);
  radio.openWritingPipe(pipeOut);
  resetData();
}

/*****/
// Returns a corrected value for a joystick position that takes into account
// the values of the outer extents and the middle of the joystick range.
int mapJoystickValues(int val, int lower, int middle, int upper, bool reverse)
{
  val = constrain(val, lower, upper);
  if ( val < middle )
    val = map(val, lower, middle, 0, 128);
  else
    val = map(val, middle, upper, 128, 255);
}

```

```

    return ( reverse ? 255 - val : val );
}

void loop()
{
    // The calibration numbers used here should be measured
    // for your joysticks till they send the correct values.
    data.throttle = mapJoystickValues( analogRead(A0), 13, 524, 1015, true );
    data.yaw      = mapJoystickValues( analogRead(A1), 1, 505, 1020, true );
    data.pitch    = mapJoystickValues( analogRead(A2), 12, 544, 1021, true );
    data.roll     = mapJoystickValues( analogRead(A3), 34, 522, 1020, true );
    data.AUX1     = digitalRead(4); //The 2 toggle switches
    data.AUX2     = digitalRead(5);
    radio.write(&data, sizeof(MyData));
}

```

## Radio Receiver Arduino Code

```
/*
 * A basic receiver test for the nRF24L01 module to receive 6 channels send a ppm sum
 * with all of them on digital pin D2.
 */

#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

//////////////////// PPM CONFIGURATION////////////////////
#define channel_number 6 //set the number of channels
#define sigPin 2 //set PPM signal output pin on the arduino
#define PPM_FrLen 27000 //set the PPM frame length in microseconds (1ms = 1000µs)
#define PPM_PulseLen 400 //set the pulse length
////////////////////

int ppm[channel_number];
const uint64_t pipeIn = 0xE8E8F0F0E1LL;
RF24 radio(9, 10);
// The sizeof this struct should not exceed 32 bytes
struct MyData {
    byte throttle;
    byte yaw;
    byte pitch;
    byte roll;
    byte AUX1;
    byte AUX2;
};
```

```

MyData data;

void resetData()
{
    // 'safe' values to use when no radio input is detected
    data.throttle = 0;
    data.yaw = 127;
    data.pitch = 127;
    data.roll = 127;
    data.AUX1 = 0;
    data.AUX2 = 0;
    setPPMValuesFromData();
}

void setPPMValuesFromData()
{
    ppm[0] = map(data.throttle, 0, 255, 1000, 2000);
    ppm[1] = map(data.yaw, 0, 255, 1000, 2000);
    ppm[2] = map(data.pitch, 0, 255, 1000, 2000);
    ppm[3] = map(data.roll, 0, 255, 1000, 2000);
    ppm[4] = map(data.AUX1, 0, 1, 1000, 2000);
    ppm[5] = map(data.AUX2, 0, 1, 1000, 2000);
}

/*****/

void setupPPM() {
    pinMode(sigPin, OUTPUT);
    digitalWrite(sigPin, 0); //set the PPM signal pin to the default state (off)
}

```

```

cli();
TCCR1A = 0; // set entire TCCR1 register to 0
TCCR1B = 0;
OCR1A = 100; // compare match register (not very important, sets the timeout for the first interrupt)
TCCR1B |= (1 << WGM12); // turn on CTC mode
TCCR1B |= (1 << CS11); // 8 prescaler: 0,5 microseconds at 16mhz
TIMSK1 |= (1 << OCIE1A); // enable timer compare interrupt
sei();
}

```

```

void setup()
{
  resetData();
  setupPPM();
  // Set up radio module
  radio.begin();
  radio.setDataRate(RF24_250KBPS); // Both endpoints must have this set the same
  radio.setAutoAck(false);
  radio.openReadingPipe(1,pipeIn);
  radio.startListening();
}

```

```

/*****

```

```

unsigned long lastRecvTime = 0;
void recvData()
{
  while ( radio.available() ) {
    radio.read(&data, sizeof(MyData));
    lastRecvTime = millis();
  }
}

```

```

}
}

/*****/
void loop()
{
  recvData();
  unsigned long now = millis();
  if ( now - lastRecvTime > 1000 ) {
    // signal lost?
    resetData();
  }
  setPPMValuesFromData();
}

/*****/
#define clockMultiplier 2 // set this to 2 if you are using a 16MHz arduino, leave as 1 for an 8MHz arduino
ISR(TIMER1_COMPA_vect){
  static boolean state = true;
  TCNT1 = 0;

  if ( state ) {
    //end pulse
    PORTD = PORTD & ~B00000100; // turn pin 2 off. Could also use: digitalWrite(sigPin,0)
    OCR1A = PPM_PulseLen * clockMultiplier;
    state = false;
  }
  else {
    //start pulse

```



```

static byte cur_chan_numb;

static unsigned int calc_rest;


PORTD = PORTD | B00000100; // turn pin 2 on. Could also use: digitalWrite(sigPin,1)

state = true;


if(cur_chan_numb >= channel_number) {
    cur_chan_numb = 0;
    calc_rest += PPM_PulseLen;
    OCR1A = (PPM_FrLen - calc_rest) * clockMultiplier;
    calc_rest = 0;
}
else {
    OCR1A = (ppm[cur_chan_numb] - PPM_PulseLen) * clockMultiplier;
    calc_rest += ppm[cur_chan_numb];
    cur_chan_numb++;
}
}
}

```

## ESC Calibration Arduino Code

```
/*ESC calibration sketch; author: EDWARD OSARETIN OBOH */
```

```
#include <Servo.h>
```

```
#define MAX_SIGNAL 2000
```

```
#define MIN_SIGNAL 1000
```

```
#define MOTOR_PIN 9
```

```
int DELAY = 1000;
```

```
Servo motor;
```

```
void setup() {
```

```
  Serial.begin(9600);
```

```
  Serial.println("Don't forget to subscribe!");
```

```
  Serial.println("ESC calibration...");
```

```
  Serial.println(" ");
```

```
  delay(1500);
```

```
  Serial.println("Program begin...");
```

```
  delay(1000);
```

```
  Serial.println("This program will start the ESC.");
```

```
  motor.attach(MOTOR_PIN);
```

```
  Serial.print("Now writing maximum output: (");Serial.print(MAX_SIGNAL);Serial.print(" us in this case");Serial.print("\n");
```

```
  Serial.println("Turn on power source, then wait 2 seconds and press any key.");
```

```
  motor.writeMicroseconds(MAX_SIGNAL);
```

```
  // Wait for input
```

```
  while (!Serial.available());
```

```
  Serial.read();
```

```

// Send min output
Serial.println("\n");
Serial.println("\n");

Serial.print("Sending minimum output: (");Serial.print(MIN_SIGNAL);Serial.print(" us in this
case)");Serial.print("\n");

motor.writeMicroseconds(MIN_SIGNAL);

Serial.println("The ESC is calibrated");

Serial.println("----");

Serial.println("Now, type a values between 1000 and 2000 and press enter");

Serial.println("and the motor will start rotating.");

Serial.println("Send 1000 to stop the motor and 2000 for full throttle");
}

void loop() {
  if (Serial.available() > 0)
  {
    int DELAY = Serial.parseInt();

    if (DELAY > 999)
    {
      motor.writeMicroseconds(DELAY);

      float SPEED = (DELAY-1000)/10;

      Serial.print("\n");

      Serial.println("Motor speed:"); Serial.print(" "); Serial.print(SPEED); Serial.print("%");
    }
  }
}

```