2024

Senior Project Report: Collision Avoidance Drone

Submitted to: Ganghee Jang

Submitted by: Cameron Robinson

June 13, 2024

Version 1.6

Submitted in fulfillment of requirements for Embedded Senior Project (CST 473), Oregon Institute of Technology.

Revision History

Version	Date	Editor	Changes
1.0	5/26/23	Cameron Robinson	Initial submission for Spring of 2023
1.1	6/14/23	Cameron Robinson	Revised submission for Spring of 2023
1.2	11/12/23	Cameron Robinson	Updated report for middle of Fall 2023
1.3	12/12/23	Cameron Robinson	Updated report for end of Fall 2023
1.4	2/28/24	Cameron Robinson	Updated report for middle of Winter 2024
1.5	3/17/24	Cameron Robinson	Updated report for end of Winter 2024
1.6	6/13/24	Cameron Robinson	Added final sections. Revised all other sections.

Legal Notice

The following parties have reviewed and agreed to the terms presented in this proposal.

Student:

____

Name Date

Professor:

____

Name Date

Abstract

The following paper is a report on the previously proposed project to design and build a collision avoidance quadcopter drone. Development for this project began in fall 2023 and will end in spring 2024. The estimated non-recurring costs for one engineer to work on this project 15-20 hours a week for 33 weeks is about $26,717.14 ($45.00/hour and $729.64for parts). In addition to skills such as time management and researching, this project will require the engineer to be able to design an embedded system; code in C; be able to design and print 3D models; and be able to design and build electric circuits.

The purpose of this project is to research LiDAR, drone, and autonomous vehicle technologies by creating a project that incorporates all three of them. This project is a quadcopter drone that uses a LiDAR sensor to detect obstacles in the surrounding environment. If the drone senses it is going to collide with any obstacles, it will automatically avoid it. The project will also include a remote control to manually control the drone. This project is not intended to solve a particular problem or become a sellable product.

The following paper will include details related to the major components of the project. These details include system design, hardware, software, project requirements, testing results, and encountered problems.

Overall, the project turned out well, and nearly all the requirements were completed and tested. Unfortunately, the drone is still unable to fly due to destabilization in the flight controller. Despite this, a lot was learned about collision avoidance algorithms, quadcopters, LiDARs, and flight control software throughout the development of this project.

Table of Contents

List of Figures

List of Tables

Project Management

Project Timeline

Figure 1, pictured below, is the timeline of the collision avoidance quadcopter drone project. The blue tasks will be done by the end of winter 2024. The purple tasks represent project stretch goals; they may or may not be implemented during spring of 2024. There were a couple of delays throughout the development of this project. The first delay was due to the IMU data not being filtered enough, which caused the accelerometer readings to spike to very large values. The second, and most significant delay, was due to the drone's flight controller destabilizing and causing the drone to oscillate out of control. Due to these delays, it is unlikely that the stretch goals will be implemented during spring term.

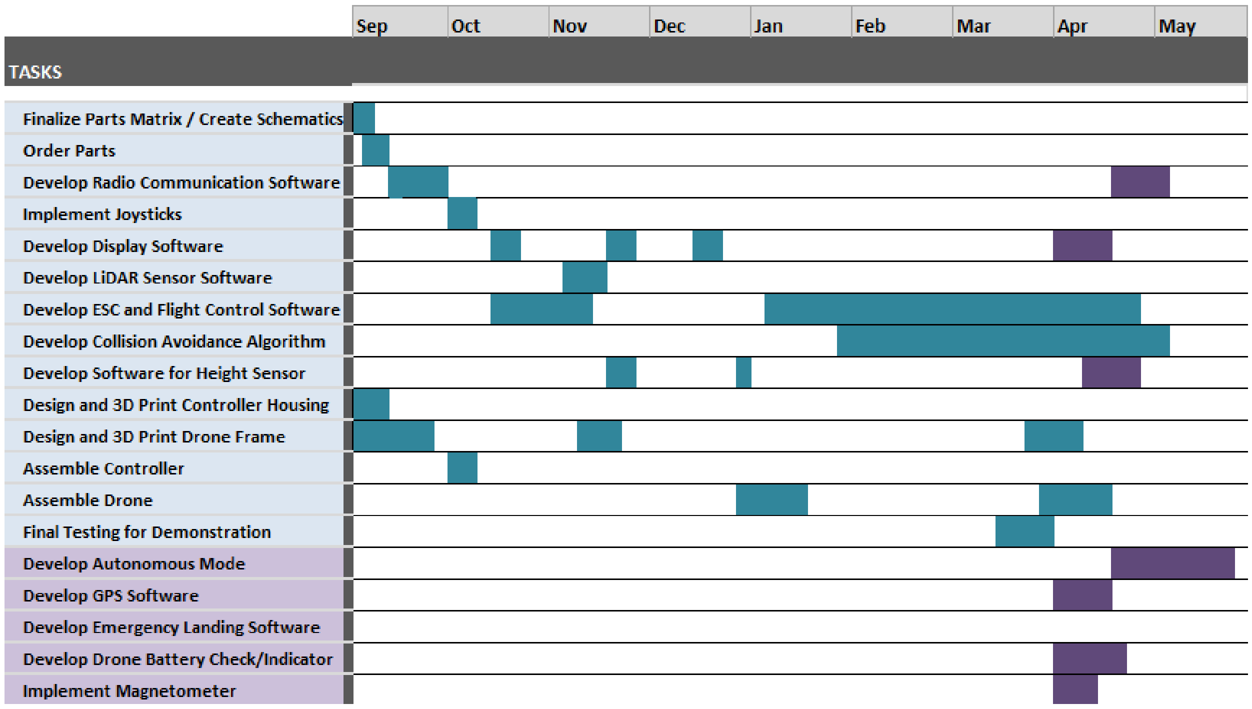


Figure 1: Project Timeline

Non-Recurring Costs

The cost of one engineer working on this project for 15 – 20 hours a week at $45.00 per hour is about $787.5 dollars per week. The total amount of time for initial development is about 22 weeks (fall 2023 and winter 2024), which results in a cost of $17,325.00. The total time for testing and developing additional features is about 11 weeks (Spring 2024) which results in an additional cost of $8,662.50. Table 1 shows the estimated engineering costs for fall, winter, and spring, as well as the total estimated engineering cost. Table 2 lists all the parts and materials used in the project and how much they cost. The total estimated cost of all parts is $729.64. Overall, the total estimated non-recurring cost for research and development over a 33-week period is $25,987.50 in engineering costs and $729.64 in materials, which results in a total cost of $26,717.14.

Reason	Cost ($)
Fall Term Research and Development (11 Weeks)	8,662.50
Winter Term Development (11 Weeks)	8,662.50
Spring Term Testing and Development (11 Weeks)	8,662.50
Total:	**25,987.50**

Table 1: Estimated Engineering Costs

Part Name	Quantity	Unit	Total Part Cost ($)
NUCLEO-H755ZI-Q	2	pcs	58.00
Grove 6-Axis IMU (BMI088)	1	pc	29.04
ICP-10111 Pressure Sensor	1	pc	7.90
F405 V3 50A 4in1 ESC	1	pc	49.99
NRF24L01+PA+LNA w/ Breakout Adapter	3	pcs	13.99
ILI9341 2.8" SPI TFT LCD	1	pc	16.39
FHL-LD19	1	pc	84.34
2212 920KV BLDC Motor	4	pcs	39.99
HRB 4S LiPo	1	pc	37.49
LIDAR07	1	pc	19.99

1045 Propellers	4	pcs	15.99
NylonG	1000	g	128.00
NylonX	500	g	58.00
Rosin Core Lead-Free Solder	100	g	19.99
URGENEX 7.4V LiPo	2	pcs	19.99
SCIGRIP Acrylic Plastic Cement	5	fl. oz.	14.35
14 AWG Silicone Wire	50	ft	19.98
Dupont Wires	120	pcs	6.98
LM7805	2	pcs	8.29
Rocker Switch	1	pc	6.99
Joystick	2	pcs	31.98
LEDs	2	pcs	0.05
330-Ohm Resistors	2	pcs	0.06
LM2596 Buck Converter	4	pcs	13.99
1045 Carbon Fiber Propellers	8	pcs	27.88
Total Estimated Parts Cost			**729.64**

Table 2: Estimated Cost of Parts

Project Tools

GitHub, Word, and Excel will be the primary project management tools used in this project. GitHub will be used for source control and storing documentation, and diagrams. Word will be used for documentation development and storing research notes. Excel will be used to create graphs, tables, and charts (such as Gantt charts) that will be used for project management.

Onshape, a CAD program, will be used for designing the 3D parts for both the drone and controller. As a student, Onshape is free to use, so using it does not add to the cost of the project.

Other tools that will be used throughout this project include Dia and KiCad. Dia will be used to create project diagrams (such as system block diagrams), and KiCad will be used to make wiring schematics. Other software tools, such as IDEs, are discussed under the 'Software' heading in the 'System Description' section.

Change Management Procedure

Any changes to this project after the beginning of fall 2023 will require the engineer to submit a request for that change in the form of an Engineering Change Order (ECO). The ECO form should contain information about what change is being made, why it is being made, and what will be affected by the change. The engineer must submit the ECO and get it approved by their supervisor. Changes to the project may require additional forms to be filled out or previously submitted work to be revised or recreated.

Status Report

The engineer in charge of the project will be required to submit a weekly memo to their supervisor. This memo will contain:

- Project progress
- Updates to the timeline
- Problems encountered
- Goals for the following week
- Any major problems or setbacks

Should the engineer fail to submit the weekly memo, there will be repercussions.

Skills

This project will require the following skills:

- Programming in C
- Designing system diagrams and wiring schematics
- Soldering
- Designing and printing 3D models

- Researching and understanding complex topics (drone flight, PID control, collision avoidance algorithms, etc.)
- Managing time and workload
- Ability to solve complex problems
- Ability to test hardware and software on embedded systems

Conceptual Overview

Introduction

Technology tends to advance very quickly, with new technologies constantly being released and the applications of existing technologies expanding. This is also the case with technologies such as LiDAR, drones, and autonomous vehicles. Fields that LiDAR is used in include agriculture, construction, geographical mapping, and autonomous vehicles. Drones, in addition to being used in agriculture and geographical mapping, are also used in weather and storm monitoring, cinematography, and much more. Autonomous vehicle technologies like autonomous delivery drones and cars are also constantly advancing.

The project outlined in this paper is a research project that combines these three technologies: drones, LiDAR, and autonomous vehicle algorithms. The purpose of this project is to learn about these technologies and gain a better understanding of them. Although this project is not going to solve a particular solution in a completely unique way, it is still a valuable project because drones, LiDAR and autonomous vehicles are advancing technologies with a multitude of applications.

Intended Audience

Since my project is a research project, my primary audience is academics. Particularly, any person wanting to learn more about LiDAR, drones, and collision avoidance algorithms.

Proposed Project

This document is a proposal to create a collision avoidance quadcopter drone. The main components of this project are a quadcopter, remote control, and collision avoidance algorithm. For collision avoidance the drone will use a 2D 360-degree rotating LiDAR. This module will sit on the drone and constantly measure the distance between the drone and obstacles in the surrounding environment. These distance measurements will be used by collision avoidance algorithms to adjust the drone's flight to avoid any obstacles it will collide with. The remote control will be used to manually control the flight of the drone with two joysticks.

Existing Solutions

Quadcopters

There are multiple collision avoidance drones currently on the market. The cost of these drones tends to range from around $300.00 to well over $1000.00. Two example drones are the DJI Air 2S [1] and the EXO X7 Ranger PLUS [2]. Although neither of the drones' product pages specify what kind of sensor (LiDAR, ultrasonic, etc.) each drone uses, neither one appears to use a 2D 360-degree rotating LiDAR sensor. Additionally, although the EXO X7 Ranger PLUS does have 360-degree obstacle detection, it avoids obstacles by simply stopping the drone, rather than flying around the obstacle.

Collision Avoidance Algorithm

The collision avoidance algorithm developed throughout this project shares some similarities to the dynamic window approach (DWA) [21]. The DWA is a collision avoidance algorithm that analyzes a small area around a robot and determines the path the robot should take by considering its velocity and the location of surrounding obstacles. The area around the robot that is analyzed, the dynamic window, is determined by how far the robot can travel given its restrictions and velocity. This algorithm is an example of a dynamic collision avoidance algorithm because it reacts to obstacles without prior knowledge of the environment. In this project, the collision avoidance algorithm will be a dynamic algorithm, however, it will not implement any form of path planning like the DWA.

Flight Controller

There are many methods of implementing flight controllers for quadcopters, however, one of the most common methods uses a 6-axis inertial measurement unit (IMU) and PID controllers. Often, the PID controllers are implemented to control the angle and angular rate of the quadcopter.

The IMU data is used as measurement inputs into the PID controllers. Measurement inputs represent the current state of the drone. For example, a flight controller could have three angular rate controllers (one for roll rate, pitch rate, and yaw rate), where each controller receives its respective angular rate measurement from the IMU’s 3-axis gyroscope that represents how fast the drone is currently rotating on that specific axis.

The PID controllers also receive a setpoint value, which represents the state they are controlling to. In most quadcopters, the setpoint for each axis is determined by the position of the joysticks on the remote control. PID controllers calculate output values based on an error term, which is the difference between the setpoint and the measurement (the difference between the current state and the desired state). These outputs are used in motor-mixing equations to calculate throttle values for each of the quadcopter’s motors that will get the drone from its current state to its desired state.

Deliverables

The final deliverables of this project will be a mostly functional remote control and quadcopter drone as specified in the ‘Requirements’ section. The controller will control the flight of the drone with two joysticks. The drone will incorporate a LiDAR module which it will use to detect obstacles in the surrounding environment. The drone will attempt to avoid obstacles it is going to collide with. The project should be mostly functional by the end of spring 2024.

Requirements

The following is a list of requirements for the collision-avoidance quadcopter drone. These requirements define what the final project should be able to do by the end of Spring 2024. Under each requirement is a list of the identification numbers of codependent requirements. The listed codependent requirements are requirements that are essential to the completion of the requirement they are listed under.

1. The project will have a quadcopter drone with four propellers, each rotating on a brushless DC motor.
    a. The motors and propellers will be used by the drone to maneuver in the air.
    b. The drone will be custom-built for this project.

    Codependent requirements: 3, 5.

2. The project shall have a remote control to operate the drone from a distance.
    a. The controller will be custom-built for this project.

    Codependent requirements: 1, 3, 4, 7.

3. The drone and controller will both have their own microcontrollers.
    a. Each microcontroller shall be fast enough to process all needed data and respond to external events in near real-time.

    Codependent requirements: 1, 2, 5.

Justification: This project will use microcontrollers for both the controller and drone since most commercial quadcopters use them. Near real-time responses will be essential for avoiding obstacles.

4. The drone and controller will communicate via radio.
    a. May be able to communicate from 50 feet away.

    b. Shall be able to communicate from 25 feet away.

Codependent requirements: 1, 2, 3, 5.

5. The drone and controller will be powered by batteries.
    a. The batteries will be rechargeable.
    b. The drone may be able to fly for 15 minutes or more.

    Codependent requirements: 1, 2.

6. The controller will have an LCD screen to display relevant information related to the drone.
    a. The LCD screen shall display indications of obstacles around the drone.
    b. The LCD screen may display the drone's GPS location data.
    c. The LCD screen shall refresh at least 15 times per second.
    d. The LCD screen may indicate quadcopter battery level.

    Codependent requirements: 2, 4, 8.

7. The controller will have two joysticks.
    a. One joystick will control the throttle and yaw of the drone.
    b. One joystick will control the pitch and roll of the drone.
    c. The values of each joystick will be read through an ADC on the controller's MCU.

Codependent requirements: 1, 2, 3, 4.

8. The drone shall use a rotating LiDAR module for obstacle detection.
    a. LiDAR module will be active while the drone is in flight.
    b. The module shall detect objects at least three meters away.
    c. The module may detect objects up to twelve meters away.

Codependent requirements: 1, 3, 5.

Justification: One of the main goals of this project is to learn about LiDAR, so while something like an ultrasonic sensor may fulfill a similar purpose, a LiDAR is more in line with the goals of this research project. The LiDAR should be able to detect obstacles at least three meters away since that will hopefully give the drone enough time to stop or avoid it, however, due to restrictions on LiDAR resolution, some small objects may be missed even at distances closer than three meters.

9. The drone shall use an inertial measurement unit (IMU) to increase flight stability.
    a. The IMU data shall be used to adjust the speed of the drone's motors.
    b. The IMU will have at least a 3-axis gyroscope and 3-axis accelerometer.

Codependent requirements: 1, 3, 5.

10. The drone shall attempt to avoid detected obstacles if it senses it is going to collide with any.
    a. Results from object detection algorithms shall be used along with drone movement data (i.e., from IMU and joysticks) to determine if the drone will collide with an obstacle.
    b. The movement of the drone shall be adjusted to prevent collision with obstacles.

Codependent requirements: 1, 2, 3, 4, 5, 7, 8, 9.

11. The drone may have an autonomous mode.
    a. In autonomous mode, the drone may take off, fly to a specified location, avoid obstacles, and land on its own.
    b. Autonomous mode may be turned on and off with a switch on the controller.

Codependent requirements: 1, 3, 5, 8, 9, 10, 13, 14.

Justification: As discussed later in the 'Part Matrices' section, the quadcopter will use an H7-series STM32 microcontroller. Many commercial quadcopter drones, even those with GPS return-home functions, use F4 or F7-series STM32 MCUs. The F-series MCUs have less than half the maximum clock speed of the H7-series, which makes it seem feasible that an autonomous mode could be implemented in this project.

12. The drone shall have a semi-autonomous mode.
    a. In semi-autonomous mode, the drone's flight shall be controlled with the controller.
    b. In semi-autonomous mode, the drone shall automatically detect and attempt to avoid obstacles it senses it will collide with.
    c. Semi-autonomous mode may be turned on and off with a switch.

    Codependent requirements: 1, 2, 3, 4, 5, 7, 8, 9.

13. The drone will use a distance measuring module to measure its distance from the ground.
    a. The module will either be a LiDAR or ultrasonic ranging module.
    b. The module's distance measurements may be used to make the drone hover some distance above the ground.
    c. The module's distance measurements may be sent to the controller to display on the LCD.

    Codependent requirements: 1, 3, 5.

Justification: Knowing how far the drone is from the ground will allow for better obstacle avoidance since the drone may need to know if it can fly up or down to avoid an obstacle. Either a LiDAR or ultrasonic ranging module could fulfill this purpose.

14. The drone will use a pressure sensor to estimate its distance from the ground.
    a. The pressure sensor may be used in conjunction with the distance sensor to get a more accurate measurement of how far the drone is from the ground.

Codependent requirements: 1, 3, 5.

Justification: A pressure sensor fulfills the same role as the distance sensor mentioned in requirement 13, however, it can measure much further distances from the ground than the distance measuring module. If used alongside the distance measuring module, the estimated distance from the ground should be more accurate than just using one or the other.

15. The drone may use a GPS module to receive data related to its own location.
    a. GPS data may be used to allow the drone to fly to a specific location.

    Codependent requirements: 1, 3, 5.

16. The drone may have an emergency landing feature where the drone will automatically land on the ground.
    a. An emergency landing may initiate when the battery reaches a certain level.
    b. An emergency landing may initiate if the drone loses connection with the controller.

    Codependent requirements: 1, 3, 5.

17. Most parts of the drone and controller’s housing shall be 3D printed.

    Codependent requirements: 1, 2.

System Description

The following section is an overview of the system designs for remote control and drone, which are designed to meet the above requirements.

Remote Control

Figure 2, pictured below, illustrates the system block diagram for the remote control. The main components in the controller are the analog to digital converter (ADC), radio transmitter (Radio TX), radio receiver (Radio RX), display, and the two joysticks.

ADC and Radio TX:

The ADC is used to read the positions of the two connected joysticks. In total, the ADC will read four values from the joysticks; throttle and yaw from one joystick, and pitch and roll from the other. These values are used to control the angular velocity of the drone's four motors. Once the data is collected from the joysticks, it is sent to the radio transmitter which will transmit it to the drone's radio receiver.

Radio RX and LCD

The radio receiver receives LiDAR data from the drone and sends it to the LCD, which will display detected obstacles within a certain distance from the drone.

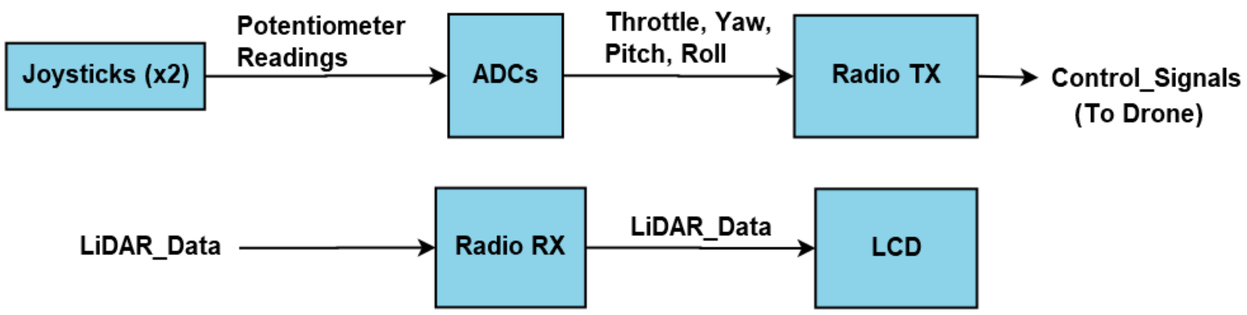


Figure 2: Remote Control System Block Diagram

Quadcopter Drone

Figure 3, shown below, illustrates the system block diagram for the drone. The major components include the radio transmitter and receiver, 2D LiDAR, rangefinder, inertial measurement unit, extended

Kalman filter, PID controllers, brushless DC (BLDC) motors, electronic speed controller (ESC), pressure sensor, and collision avoidance algorithm.

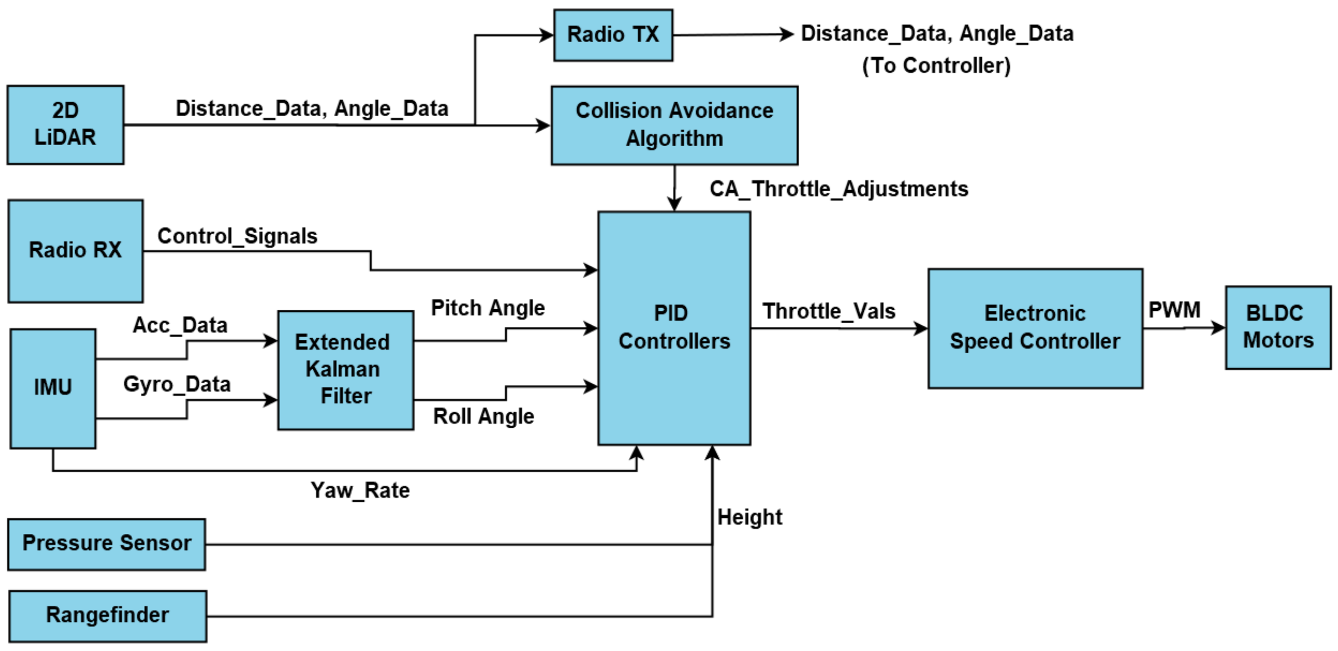


Figure 3: Drone System Block Diagram

2D LiDAR

The 2D LiDAR module is a 360-degree rotating distance sensor that measures distances between the drone and obstacles in the surrounding environment. This distance data is sent to the Radio TX and Collision Avoidance Algorithm blocks.

Rangefinder and Pressure Sensor

Both the rangefinder and pressure sensor perform the same function: they measure the distance between the drone and the ground. This data will potentially be used to allow the drone to hover a certain height above the ground. Not only does measuring the distance to the ground with two different sensors help verify that the measurements are correct, but each sensor also helps account for the other's weaknesses. The rangefinder is only accurate when measuring certain types of materials and can only measure up to a maximum distance. On the other hand, the pressure sensor is not restricted by distance from the ground, however, its measurements are based on the air pressure at sea level, so it does not necessarily know how far away the ground is.

Radio TX

The radio transmitter sends the LiDAR angles and distance measurements to the remote control.

Collision Avoidance Algorithm

The collision avoidance algorithm for this project uses the 2D LiDAR data to detect obstacles in the drone’s surroundings. This algorithm is divided into three steps: obstacle detection, hazard assessment, and obstacle avoidance.

Obstacle detection analyzes the LiDAR data to determine the distance and direction of obstacles around the drone. The hazard assessment step determines which obstacles the drone is most likely to collide with and outputs values that prioritizes avoiding those obstacles. Lastly, obstacle avoidance uses the results from the hazard assessment and obstacle detection stages to determine adjustment values for the drone’s motors that will allow it to avoid the obstacles.

Radio RX

The radio receiver receives joystick position data from the remote control. This data is sent directly to the PID Controllers block.

Inertial Measurement Unit

The inertial measurement unit (IMU) is a sensor that detects linear acceleration and angular velocity. This module is essential to the flight controller since it provides angle and angular rate measurements to the PID controllers. Angular velocity is measured with a 3-axis gyroscope and linear acceleration is measured with a 3-axis accelerometer. The gyroscope and accelerometer readings are sent to the

Extended Kalman Filter block. The gyroscope yaw reading is also sent directly to the PID Controllers block.

Extended Kalman Filter

The extended Kalman filter is implemented in software and uses the gyroscope and accelerometer data to estimate the quadcopter's roll and pitch angles. The roll and pitch angle estimations are sent to the PID Controllers block and are used for controlling and the drone's flight.

PID Controllers

The PID Controllers block represents software algorithms for controlling drone flight and stabilization. This block takes in the joystick data, collision avoidance results, IMU data, and the drone's distance from the ground. This data is used to calculate how fast each of the drone's four motors should be spinning to avoid obstacles or move in the direction specified by the joystick values. Once the throttle for each motor is calculated, the values are sent to the ESC through a protocol called DSHOT.

In general, the data will be used as follows:

- The joystick data will control the drone's orientation and throttle (PID setpoint).
- The IMU data will provide the measured orientation of the drone (PID measurement).
- The results from the collision avoidance algorithm will adjust the drone's flight to avoid obstacles (also a PID setpoint).
- The drone's measured height may be used to hover the drone a set distance above the ground (not currently implemented).

Electronic Speed Controller

The ESC receives data through a protocol called DSHOT. There is one communication line for each of the four motors, which allows each of the motors' throttles to be changed independently.

BLDC Motors

The BLDC motors are what enable the quadcopter to fly. These motors are controlled through the ESC. The angular velocity of each motor determines the net thrust on the drone which controls all movements of the drones.

Project Housing

Figure 4 shows the 3D model for the remote control. The face of the controller shell has cut-outs for the joysticks, LCD, two LEDs, and a power switch. The front of the controller has cut-outs for the battery and radio antenna.

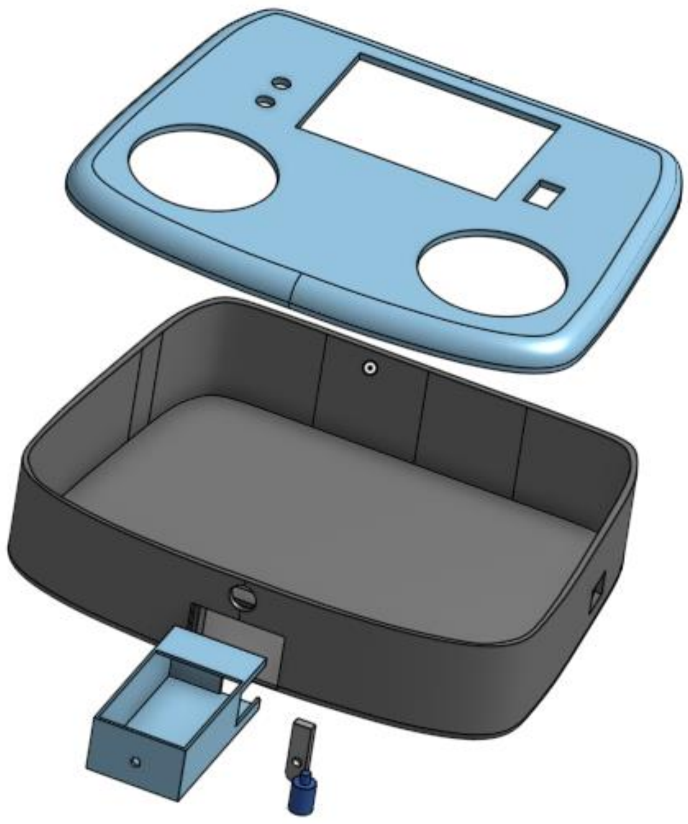


Figure 4: Controller Shell

Figure 5 is a 3D model of the quadcopter frame. The frame measures about 445mm on the diagonal (not including the bumpers) and weighs about 500g. All the components, aside from the BLDC motors, are designed to fit on or within the central container. The rectangular hole in the top of the frame is for the 2D rotating LiDAR. The LiDAR is designed to stick out above the rest of the frame so that it can measure 360-degrees around the drone. Each BLDC motor sits on one of the four corners of the frame. The circular bumpers around the frame are intended to minimize the damage caused by collisions that may occur during testing.

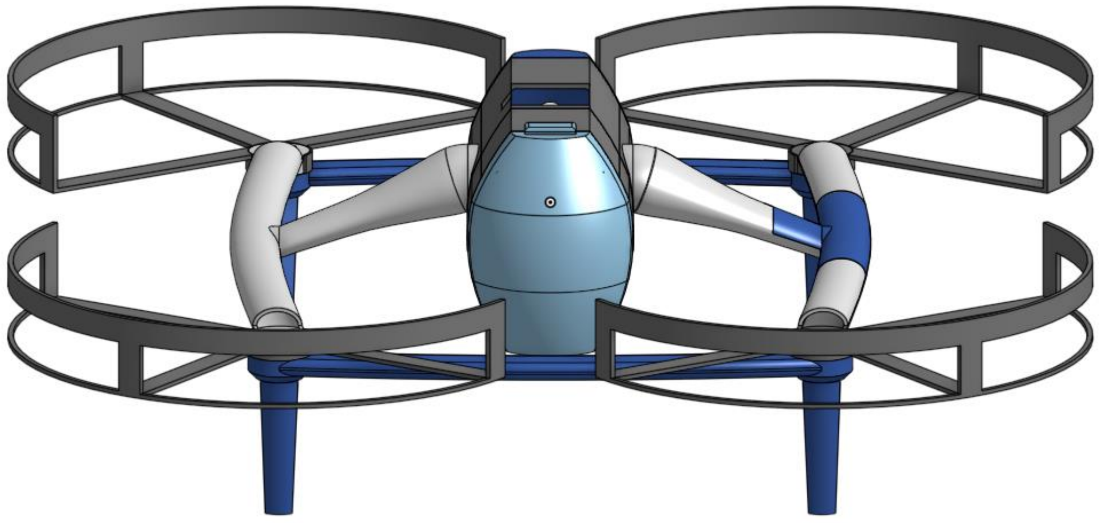


Figure 5: Quadcopter Frame

The housing for both the drone and controller were made using Onshape.

Hardware

The controller will have six hardware components. These components include an MCU, radio transceiver, LCD display, two joysticks, and a rechargeable battery. The MCU will either need multiple ADCs or one ADC with multiple channels to read the positions of each joystick. Additionally, the MCU will need pins for some serial protocols in order to communicate with the radio and LCD display.

The hardware components for the drone include:

- An MCU
- Four BLDC motors

- One 6-axis IMU
- A radio transceiver
- LiDAR module
- Distance measuring unit (ultrasonic or LiDAR)
- Pressure sensor
- Rechargeable battery (LiPo or Li-ion)
- Electronic speed controller (ESC)

The MCU will need enough serial protocol pins to communicate with each of the peripheral devices. The MCU will also need four pins capable of outputting DSHOT or PWM signals for sending data to the ESC. The battery needs to be able to supply enough current to run all the motors at full power and it must be able to power the normal functions of the drone for 15 minutes.

Part Matrices

The following tables are part matrices for the core components of the project described above. Each table specifies at least three potential components that could be used. The components highlighted in green are the ones chosen for this project.

Table 3 illustrates potential MCUs that would work for both the quadcopter and controller. The board chosen for this project was the NUCLEO-H755ZI-Q. Although this board has a lower clock speed than the Pi boards, it has far more GPIO pins and serial communication pins; both of which will be extremely useful for this project since there are multiple peripheral sensors and motors. The NUCLEO board also takes far less power than the Pi boards which will help extend the battery life of the quadcopter. The need for an MCU is directly connected to the third requirement in the ‘Requirements’ section.

MCUs				
Module Name	NUCLEO-H755ZI-Q	Raspberry Pi 4	Raspberry Pi Zero W	NanoPi M4B
Cost ($)	29.00	82.00	15.00	119.99
CPU Speed	480MHz	1.5GHz	1GHz	2.0GHz
RAM	1MB	4GB	512MB	2GB
Flash	1MB	Micro SD	Micro SD	Micro SD
Relevant Serial Protocols (I2C, SPI, UART, etc.)	4x I2C, 4x UART, 1x LPUART, 6x SPI	1x I2C, 1x UART, 1x SPI, 2x USB 2.0, 2x USB 3.0	1x I2C, 1x UART, 1x SPI	2x USB 2.0, 2x USB 3.0, 3x I2C, 1x UART, 1x SPI
GPIO	144 pins	40 Pins	40 Pins	64 pins
Other	3.3V/0.5A (max)	5V/3A	5V/2.5A	5V/3A

Table 3: Potential MCUs

Table 4 shows several options for the quadcopter's inertial measurement unit (IMU). The IMU chosen for this project was the BMI088. The primary reason for this decision was that the BMI088 was designed specifically for quadcopters and other drones, so although it is not the cheapest option, it will hopefully be the best performing one. The IMU is necessary to fulfill requirement nine.

Inertial Measurement Unit				
Module Name	Grove 6-Axis IMU (BMI088)	GY-87 10DOF Module	BNO055+BMP280 10DOF Module	Fermion: 10DOF IMU Sensor
Cost ($)	29.04	16.99	25.90	39.50
Accelerometer	3-Axis	3-Axis	3-Axis	3-Axis
Gyroscope	3-Axis	3-Axis	3-Axis	3-Axis
Magnetometer	No	3-Axis	3-Axis	3-Axis
Pressure Sensor	No	Yes	Yes	Yes
Communication Protocol	I2C	I2C	I2C	I2C
Other Notes	Designed specifically for drones.		Also has temperature sensor	Also has temperature sensor

Table 4: Potential IMUs

Table 5 shows options for pressure sensors, which are needed because the BMI088 does not have one. Any of the listed devices would work and none of them are very expensive. The device chosen for this project was the Ferimon ICP-10111 pressure sensor since it was designed specifically for drones, and it has the highest relative accuracy. A pressure sensor is needed to fulfill requirement 14.

Pressure Sensors			
Module Name	Ferimon ICP-10111	BME280 Altimeter	BMP390
Cost ($)	7.90	9.98	14.99
Communication Protocol	I2C	I2C/SPI	I2C/SPI
Relative Accuracy	+/- 1 Pa	+/- 12 Pa	+/- 3 Pa
Other Notes	Intended for drones.	Specifies “flying toys” as a target device. Also has a humidity sensor.	

Table 5: Potential Pressure Sensors

Table 6 illustrates various electronic speed controllers (ESC) to control the quadcopter’s BLDC motors. The ESC chosen for this project was the SpeedyBee F405 V3 50A 4-in-1 ESC since it has a good balance between firmware and cost. Although the Hobbypower and QWinOut ESCs are both cheaper, they use SimonK firmware, which is outdated. These ESCs are also bulkier than the other three options since they are not 4-in-1 ESCs. The HolyBro and T-Motor boards have the newest firmware, however, they are nearly double the cost of the SpeedyBee ESC and the increase in performance does not justify the extra cost. The need for an ESC is directly connected to requirement one.

Electronic Speed Controllers					
Module Name	Hobbypower SimonK 30A ESC Brushless Speed Controller BEC 2A	QWinOut 2-4S 30A RC Brushless ESC Simonk Firmware Electric Speed Controller	SpeedyBee F405 V3 50A 4in1 ESC Board	HolyBro Tekko32 F4	T-Motor F55A Pro
Cost ($)	35.99	38.00	49.99	106.91	90.90
Firmware	SimonK	SimonK	BLHeli_S	BLHeli32	BLHeli32
Amp Rating (A)	30	30	50	60	55
Amount	4	4	1	1	1
Other	Controlled through PWM		4-in-1. Uses DSHOT300/600 protocol.	4-in-1. Supports nearly all protocols (PWM, DSHOT, etc.).	4-in-1. Supports nearly all protocols. (PWM, DSHOT, etc.).

Table 6: Potential ESCs

Table 7 shows three options for radio transceivers that would work for communication between the controller and quadcopter. This project will use the NRF24L01+PA+LNA simply because there are already some on hand, which makes the cost of the part effectively zero. Additionally, when compared to the other NRF24L01+, the NRFL01+PA+LNA has a much larger external antenna which should allow it to transmit further distances. A radio transceiver is necessary to fulfill requirement number four.

Radio Transceivers			
Module Name	NRF24L01+PA+LNA Wireless Transceiver RF Transceiver Module + Breakout Adapter	NRF24L01+ (Integrated Antenna)	ExpressLRS ELRS EP1 TCXO Long Range Low Latency Receiver
Cost ($)	13.99	11.99	22.99
Amount	3	10	1
Frequency (GHz)	2.4	2.4	2.4
Communication Protocol	SPI	SPI	UART

Table 7: Potential Radio Transceivers

Table 8 shows three options for the controller's LCD display. All the options are fairly cheap and all of them use SPI to communicate, so the primary deciding factor was the size of the LCD since it needs to be able to clearly display obstacles surrounding the quadcopter. For this reason, the HiLetgo ILI9341 2.8" TFT LCD was chosen since it has a 0.8" larger screen than the Waveshare IPS LCD. An LCD is needed for requirement six.

LCD Displays			
Module Name	Waveshare General 2inch IPS LCD Display	1.8" SPI TFT LCD Display	HiLetgo ILI9341 2.8" SPI TFT LCD
Cost ($)	14.65	11.37	16.39
Resolution	240x320	128x160	240x320
Driver	ST7789	ST7789	ILI9341
Serial Protocol	SPI	SPI	SPI
Other	IPS Display		

Table 8: Potential LCDs

Table 9 illustrates the various LiDAR modules for obstacle detection. The main considerations were the cost, samples per second, mass, and ability to function in sunlight (lux rating). In the end, the LiDAR chosen for this project was the FHL-LD19. Although this LiDAR has the lowest sample rate, it also has the lowest cost, and lowest mass. Even with the low sample rate, the LiDAR should still be fast enough to detect most obstacles, however, if the object is too small, the LiDAR may miss it from further distances, which means the quadcopter will have less time to react. Overall, it was difficult to justify using an expensive LiDAR on a quadcopter that will likely collide with multiple walls throughout the development of the collision avoidance and flight controller software. A 2D LiDAR module is needed to fulfill requirement eight.

2D LiDAR Modules				
Module Name	RPLIDAR A2M12	RPLIDAR A1M8-R6	youyeetoo FHL-LD19 Lidar Sensor	RPLiDAR S2L
Cost ($)	229.00	99.00	84.34	299.00
Range (m)	0.2 - 12	0.2 - 12	0.2 - 12	0.05 - 18
Communication Protocol	UART	UART	UART	UART
Samples Per Second	16k	8k	4.5k	32k
Rotation Frequency (Hz)	5 - 15	5.5	5 - 13	10
Lux Rating	Doesn't Specify	Doesn't Specify	30k	80k
Mass (g)	170	170	50	190
Other	Needs PWM. Triangulation.	Needs PWM. Triangulation.	Needs PWM. Time of Flight.	Doesn't appear to need PWM. Time of Flight.

Table 9: Potential Rotating LiDARs

Table 10 shows three of the options for BLDC motors. The main consideration for the motors was the KV rating. The quadcopter in this project will be relatively large, so a motor with a lower KV rating is most preferable since it will work better with larger propellors. The higher KV motors are often catered towards FPV racing drones which are designed to be small and light. For this reason, with the added benefit of also being the cheapest, the 2212 920KV BLDC motors were chosen for this project. Although the peak current of these motors is not specified, it is very unlikely to be over 50 amps which is the rating of the SpeedyBee ESC. BLDC motors are needed to fulfill requirement one.

Brushless DC Motors			
Module Name	2212 920KV Brushless Motors	XING-E Pro 2207 1800KV Brushless Motor	XING-E Pro 2207 2450KV
Cost	39.99	62.99	63.99
Amount	4	4	4
Voltage (V)	7-12	7.4 – 22.2	7.4 – 14.8
Peak Current (A)	Doesn't Specify	35	43
KV Rating	920	1800	2450
Mass (g)	53	33.8	33.8
Other	~750-800g lift. Intended for DJI Phantom.	Intended to be used with a 6S battery. For FPV.	Intended to be used with 4S batteries. For FPV.

Table 10: Potential BLDC Motors

Table 11 indicates multiple options for LiPo batteries that would be used to power the quadcopter and remote control. Most of the batteries are 14.8V (4S) since that is the most common voltage for quadcopters and will work with the ESC and the motors. The battery chosen for the drone was the HRB 4S 4200mAh LiPo. The main considerations when choosing this battery were the mAh rating, C rating, weight, and cost. These values were compared to the needs of the BLDC motors, which have the highest power consumption in this project. The mAh rating was based on what was recommended for a 450mm quadcopter with 920 KV motors. The HRB battery also has a 60C rating which means it should be able to supply 50A to all four motors at the same time, though that much current should not be necessary. The URGENEX 7.4V battery was chosen to power the remote control because it is small and there are already some on hand. LiPo batteries are needed to fulfill requirement five.

LiPo Batteries					
Module Name	URGENEX 7.4V Lipo Battery	Zeee 14.8V 4S Lipo Battery 50C 3300mAh	Zeee 4S Lipo Battery 5200mAh 14.8V 120C	Zeee 4S Lipo Battery 6200mAh 14.8V 80C	HRB 4S Lipo Battery 14.8V 4200mAh 60C
Cost ($)	19.99	62.09	89.09	92.79	37.49
Amount	2	2	2	2	1
Voltage (V)	7.4	14.8	14.8	14.8	14.8
C Rating	35	50	120	80	60
mAh	1000	3300	5200	6200	4200

Max Cont. Current (A)	35	165	624	496	252
Mass (g)	51	348	470	625	410

Table 11: Potential LiPo Batteries

Table 12 illustrates three distance measuring sensors that could be used by the quadcopter to measure its distance from the ground. The module chosen for this project is the LIDAR07 since it can measure much further than the HC-SR04, and is half as expensive as the TFmini-S. Although the TFmini-S is more accurate than the LIDAR07, it is not by much, so the extra cost does not seem justified for this project. The distance sensor will be used in conjunction with the pressure sensor to determine the height of the quadcopter, so an extra 0.5% accuracy is irrelevant. A distance measuring module is needed to fulfill requirement 13.

Distance Measuring Units			
Module Name	LIDAR07 (SEN0413)	HC-SR04	TFmini-S Lidar Sensor
Cost ($)	19.99	8.99	43.90
Type	LiDAR	Ultrasonic	LiDAR
Range (m)	0.2 - 12	0.2 - 4	0.1 - 12
Communication Protocol	I2C/UART	Trigger-Pulse	UART/I2C
Accuracy	+/- 5cm @ 20 – 350 cm +/- 1.5% @ 351 – 1200 cm	Up to 3mm resolution	+/- 6cm @ 10 – 600 cm +/- 1% @ 601 – 1200 cm

Table 12: Potential Distance Measuring Units

Hardware Components

Figure 34 and Figure 35, shown in the appendix, are the wiring schematics for the controller and drone, respectively. The schematics illustrate how all the components are connected to the MCUs and how they are powered by the batteries. In general, everything but the ESC can be powered with 5V. The ESC is powered directly from the drone’s battery which is about 14.8V. It is important to note that the radios should only be powered with 5V if using their breakout adapters, otherwise, they should be powered with 3.3V.

NUCLEO-H755ZI-Q

The NUCLEO-H755SZI-Q is a development board used in both the drone and controller. The boards use the STM32H755ZI microcontroller. Both NUCLEO boards are powered with LiPo batteries through their E5V pins. The voltage supplied to each board is stepped-down to 5V with buck-converters. The clock frequencies on the boards are set to 300MHz; the peripheral clocks are set to 75MHz; and the timer clocks are set to 150MHz. Table 13 and Table 14 show the peripheral settings for the controller and drone respectively. The tables contain the peripheral names, important settings, and what it is used for. The GPIOs are not listed because they are shown in the wiring schematics found in the appendix. For more details on the MCU, see the STM32H755xI datasheet [17].

Remote Control STM32CubeIDE Peripheral Settings		
Peripheral	**Settings**	**Use**
DMA1 Stream 0	Mem-to-peripheral; Request: SPI1 Transmit; Data Width: 8-bits; Priority: High	LCD Communication
NVIC1	ADC1; ADC2; EXTI Line [9:5]; TIM1; TIM3; SPI1	Interrupt Controller
ADC1 / ADC2	Single Conversion Mode; Independent; 16-bit resolution; Regular Conversions	Read Joysticks
TIM1	FreeRTOS Defaults (1000Hz Tick Rate)	FreeRTOS Clock
TIM3	Prescaler: 149; Counter Period: 29999	Timer to Begin ADC Conv
SPI 1	Motorola; 8-bit data; CPOL: Low; CPHA: 1 Edge; Prescaler: 2 (25MBits/s); MSB First	LCD Communication
SPI 2	Motorola; 8-bit data; CPOL: Low; CPHA: 1 Edge; Prescaler: 8 (6.25MBits/s); MSB First	Radio Communication
FREERTOS_M7	Software pack for FreeRTOS	System OS

Table 13: Remote Control Peripheral Settings

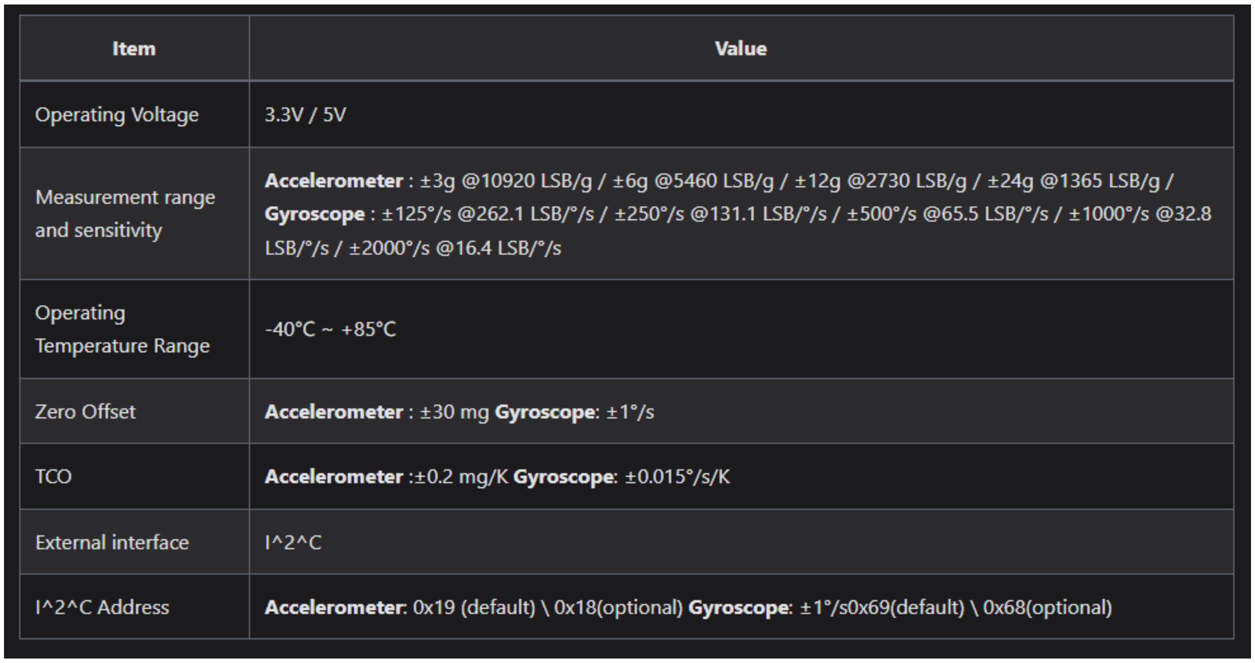
Drone STM32CubeIDE Peripheral Settings		
Peripheral	**Settings**	**Use**
DMA1 Stream 1	Mem-to-peripheral; Request: TIM3_CH1; Data Width: Half-Word; Priority: High	ESC DShot Comm. M1
DMA1 Stream 2	Mem-to-peripheral; Request: TIM4_CH2; Data Width: Half-Word; Priority: High	ESC DShot Comm. M2

DMA1 Stream 3	Mem-to-peripheral; Request: TIM4_CH3; Data Width: Half-Word; Priority: High	ESC DShot Comm. M3
DMA1 Stream 4	Mem-to-peripheral; Request: TIM3_CH4; Data Width: Half-Word; Priority: High	ESC DShot Comm. M4
DMA1 Stream 0	Mem-to-peripheral; Request: UART5_RX; Data Width: 8-bits; Priority: Med; Circular FIFO	Rangefinder
DMA 2 Stream 0	Mem-to-peripheral; Request: USART1_RX; Data Width: 8-bits; Priority: Med; Circular FIFO	2D LiDAR
NVIC1	EXTI Line [9:5]; DMA (all above)	Interrupt Controller
TIM1	FreeRTOS Defaults (1000Hz Tick Rate)	FreeRTOS Clock
TIM3	PWM Mode; Prescaler: 9; Counter Period: 24	DShot Timer
TIM4	PWM Mode; Prescaler: 9; Counter Period: 24	DShot Timer
TIM7	Prescaler: 14999; Counter Period: 65535	Radio Recovery Timer
I2C 1	Fast Mode (400KHz); 7-bit Address	Pressure Sensor
I2C 2	Fast Mode (400KHz); 7-bit Address	IMU
SPI 2	Motorola; 8-bit data; CPOL: Low; CPHA: 1 Edge; Prescaler: 8 (6.25MBits/s); MSB First	Radio
UART 5	115200 baud; 1 start bit; 8 data bits; 1 stop bit; 16x Oversample; RX and TX	Rangefinder
USART 1	230400 baud; 1 start bit; 8 data bits; 1 stop bit; 16x Oversample; RX Only	2D LiDAR
FREERTOS_M7	Software pack for FreeRTOS	System OS

Table 14: Drone Peripheral Settings

Grove 6-Axis IMU

The Grove 6-Axis IMU uses the BMI088, which is a 6-axis IMU (3-axis accelerometer and 3-axis gyroscope) designed for arial vehicle applications. The module communicates with the drone's MCU with I2C running at 400KHz. In this project, the data is read from the IMU in polling mode, though the module does have pins for interrupts. Further specifications of the IMU can be found in Figure 6, the product wiki page [15], and the BMI088 datasheet [19].



Item	Value
Operating Voltage	3.3V / 5V
Measurement range and sensitivity	**Accelerometer** : ±3g @10920 LSB/g / ±6g @5460 LSB/g / ±12g @2730 LSB/g / ±24g @1365 LSB/g / **Gyroscope** : ±125°/s @262.1 LSB/°/s / ±250°/s @131.1 LSB/°/s / ±500°/s @65.5 LSB/°/s / ±1000°/s @32.8 LSB/°/s / ±2000°/s @16.4 LSB/°/s
Operating Temperature Range	-40°C ~ +85°C
Zero Offset	**Accelerometer** : ±30 mg **Gyroscope**: ±1°/s
TCO	**Accelerometer** :±0.2 mg/K **Gyroscope**: ±0.015°/s/K
External interface	I^2^C
I^2^C Address	**Accelerometer**: 0x19 (default) \ 0x18(optional) **Gyroscope**: ±1°/s0x69(default) \ 0x68(optional)

Figure 6: IMU Specifications Adapted from [15]

Table 15 contains the initialization values for both the accelerometer and gyroscope on the BMI088 IC. These are not necessarily the only values that will work. For example, the gyroscope may not need a measurement range of +/- 250 deg/s, and the accelerometer may not need a range of +/- 24g. Similarly, the filtering of both sensors' data should be adjusted based on the application; the values chosen in this project are a balance between filtering to eliminate noise caused by vibrations and having a fast-enough output data rate.

BMI088 Initialization					
Chip	Register	Address	Default Value	Value Needed	Description
ACC	ACC_PWR_CTRL	0x7D	0x00	0x04	Power on
ACC	ACC_PWR_CONF	0x7C	0x03	0x00	Set to active mode
ACC	ACC_CONF	0x40	0xA8	0x8A	4-fold oversample; 400Hz data rate
ACC	ACC_RANGE	0x41	0x01	0x03	+/- 24g range
GYRO	GYRO_LPM1	0x11	0x00	0x00	Normal power mode
GYRO	GYRO_BANDWIDTH	0x10	0x80	0x03	47Hz Filter BW
GYRO	GYRO_RANGE	0x0F	0x00	0x03	+/- 250 deg/s range

Table 15: BMI088 Register Initialization Values

Ferimon ICP-10111

The Ferimon ICP-10111 pressure sensor communicates with the MCU using the I2C protocol at 400KHz. Like the IMU, the data is received through polling. A data packet from the pressure sensor contains both temperature and pressure measurements, however, they are not in a usable format. The temperature data can be converted to Celsius, and the pressure data can be converted to Pascals through the equations shown in Figure 7. For the pressure data conversion A, B, and C are values calculated in software with functions provided by the chip manufacturer. More specifications on the ICP-10111 and software functions to use the chip can be found in the datasheet [13].

$$T = -45^{\circ}C + \frac{175^{\circ}C}{2^{16}} \times t\_dout.$$

$$P = A + \frac{B}{C + p_{\mathrm{dout}}},$$

Figure 7: Temperature and Pressure Conversion Equations Adapted from [13]

SpeedyBee F405 ESC

The SpeedyBee F405 ESC communicates with the MCU through the DShot600 protocol, which operates at 600,000 bits per second. There is one wire connected to the MCU for each motor. The ESC accepts values ranging from 0 – 2047; values from 0 – 47 are commands, while values from 48 – 2047 are throttle values. On startup, the ESC needs to receive zero throttle for each of the motors before it will start running. In this project, DMA is used to send DSHOT data packets to the ESC with one DMA channel assigned to each motor. Four timers are used to measure the pulse widths of the packet transmission since bit-values are determined by the pulse widths on the data transmit line. More details on the SpeedyBee F405 ESC can be found in the datasheet [20].

NRF24L01+PA+LNA

The NRF24 radio is implemented on both the drone and the remote control. Since the NRF24 cannot handle receiving and transmitting at the same time, acknowledgement packets are used to synchronize sending and receiving between the two systems. The drone's radio is initialized as the primary transmitter while the controller is the primary receiver. This means that the drone initiates all data transactions by sending a packet. When the controller's radio receives the packet, it will immediately send an acknowledgement-payload back to the drone. This payload indicates to the drone that the controller received the data, and it also contains joystick data from the controller. The datasheet for the NRF24 [16] contains more information on the registers and functionality of the module.

Table 16 contains the initialization-values of the radio's register for both the drone and controller. Not all the values are needed to make the radios function; registers like data rate and channel frequency can be changed, but they need to match on both the drone and controller. The only difference between the radio initializations is that the drone is set as the transmitter while the controller is set as the receiver.

NRF Initialization				
Register	**Address**	**Default Value**	**Value Needed**	**Description**
CONFIG	0x00	0x08	0x0A	Power on; TX mode (DRONE)
CONFIG	0x00	0x08	0x0B	Power on; RX mode (CONTROLLER)
DYNPD	0x1C	0x00	0x01	Enable dynamic payload for pipeline 0
EN_AA	0x01	0x3F	0x3F	Enable auto-ack
FEATURE	0x1D	0x00	0x06	Enable dynamic payload and ack-payload
RF_SETUP	0x06	0x0E	0x0E	2Mbps data rate; 0dBm
RF_CH	0x06	0x02	0x64	2500MHz channel frequency
SETUP_RETR	0x04	0x03	0xFF	4ms delay b/w re-TX; Re-TX 15 times

Table 16: NRF24 Initialization for Drone and Controller

HiLetgo ILI9341 2.8" LCD

The LCD is part of the controller and is communicated to through SPI. The SPI transmit and receive are handled through DMA on the MCU. Although the LCD has touch-screen capabilities, these are not utilized in this project. Table 17 shows the command sequence to initialize the LCD in this project. These commands are from the ili9341_initialize(ili9341_t *lcd) function, which is part of the LCD library [9] used in this project. The commands in the table are sent to the LCD from top to bottom. The ILI9341 IC datasheet [18] contains more information on the registers and how to use the LCD.

ILI9341 Initialization Sequence		
Command	Command Data	Purpose
0x01	None	SW RST
0xCB	0x39, 0x2C, 0x00, 0x34, 0x02	Power Control A
0xCF	0x00, 0xC1, 0x30	Power Control B
0xE8	0x85, 0x00, 0x78	Driver Timing Control A
0xEA	0x00, 0x00	Driver Timing Control B
0xED	0x64, 0x03, 0x12, 0x81	Power On Sequence Control
0xF7	0x20	Pump Ratio Control
0xC0	0x23	Power Control VRH [5:0]
0xC1	0x10	Power Control SAP[2:0], BT[3:0]
0xC5	0x3E, 0x28	VCM Control
0xC7	0x86	VCM Control 2
0x36	0x48	Memory Access Control
0x3A	0x55	Pixel Format
0xB1	0x00, 0x18	Set Frame Radio Control to Standard RGB
0xB6	0x08, 0x82, 0x27	Display Function Control
0xF2	0x00	3Gamma Function Disable
0x26	0x01	Gamma Curve Select
0xE0	0x0F, 0x31, 0x2B, 0x0C, 0x0E, 0x08, 0x4E, 0xF1, 0x37, 0x07, 0x10, 0x03, 0x0E, 0x09, 0x00	Positive Gamma Correction
0xE1	0x00, 0x0E, 0x14, 0x03, 0x11, 0x07, 0x31, 0xC1, 0x48, 0x08, 0x0F, 0x0C, 0x31, 0x36, 0x0F	Negative Gamma Correction
0x11	None	Exit Sleep
0x29	None	Turn Display On
0x36	0x28	Set Screen Orientation to Left

Table 17: Init Sequence for LCD from ili9341.c in [9]

FHL-LD19 LiDAR

The 2D LiDAR is extremely simple to set up since it does not have an RX pin. Once the LiDAR is powered, it will begin transmitting packets through UART at 230400 baud. One UART transmission contains a start bit, eight data bits, and a stop bit. Since there is no way to synchronize the MCU's data reception with the LiDAR, the LiDAR's ground is connected to a TIP120 transistor. The base of the transistor is connected to a pin on the MCU which allows the MCU to turn on/off the LiDAR. This allows the MCU to boot and initialize everything before powering-on the LiDAR, so it does not become desynched from the start of the LiDAR data reception. Additionally, to make sure that the data reception is not desynched after boot-up, the data is received through DMA so that the CPU does not need to switch from other processes just to receive the LiDAR data.

The LD19 has a PWM pin which is used to control the rotation of the LiDAR's motor. If the PWM pin is connected to ground, as it is in this project, the motor rotates ten times per second. Since the LiDAR measures at 4500Hz and rotates at 10Hz, it measures about 450 times per second, which results in a 0.8-degree resolution. This means that there is a difference of about 0.8-degrees between any two LiDAR distance measurements. More information on the LD19 can be found on the manufacturer's wiki page [3].

2212 920KV BLDC Motors

Four BLDC motors are needed for this project. Each motor is connected to the ESC which drives each of them individually. It is important to note that on a quadcopter, the motors diagonal from each other on the frame must spin in the same directions. That is to say, the front left and back right motors should spin the same direction, but opposite the direction of the front right and back left motors. This approach is used to balance the net torque from the motors' rotations. If not implemented correctly, the torques from the motors will cause the drone to rotate on its z-axis (yaw).

Joysticks

The controller has two joysticks for controlling the drone: a left joystick for throttle and yaw, and a right joystick for pitch and roll. Although not essential to the project, the left joystick's y-axis (throttle) is not connected to a spring, so it will not automatically center itself. The yaw, pitch, and roll axis will automatically center themselves. Each axis on the joysticks is connected to an ADC channel on the MCU. The ADCs periodically read the values of the joysticks which are translated into control signals for the drone in software. How often the joystick values are sampled is not very important. Currently they are being sampled every 30ms, which is fast enough that control of the drone feels responsive, but also not so fast that the other controller tasks are being interrupted constantly.

Software

All the code for this project is written in C. Since the project uses an STM32 MCU, STM32CubeIDE 1.12.1 is used to program and debug the drone and controller. Both the drone and controller use FreeRTOS to handle the various tasks on each device. Aside from the Blink tasks, each task has a UML diagram associated with it. These UML diagrams contain brief descriptions of the tasks, essential variables each task uses, and essential functions each task uses.

Remote Control

Figure 8 illustrates the software block diagram for the controller. There are only two tasks running on the controller, the LCD Display Task, and the Blink Task; everything else is handled through interrupt service routines (ISRs).

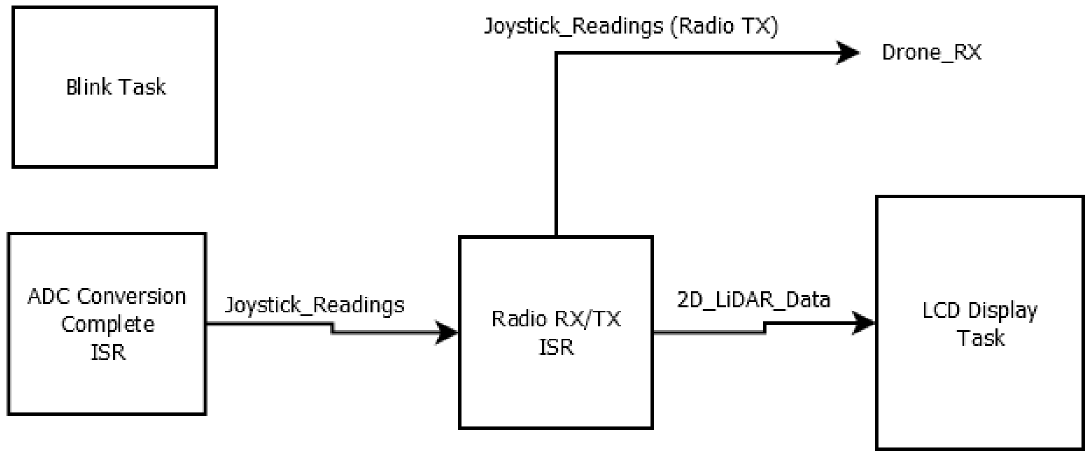
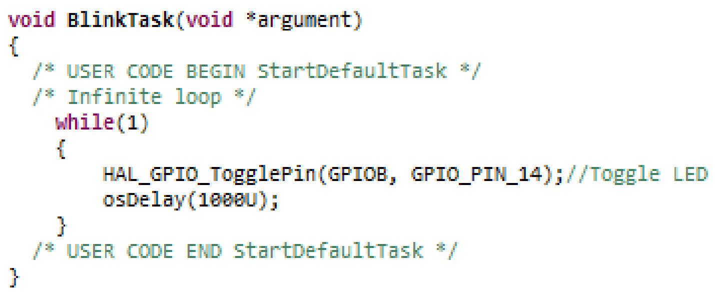


Figure 8: Controller Software Diagram

Blink Task

The Blink task toggles an LED on the NUCLEO board every second. The purpose of this task is to give visual confirmation that the system tasks are running, and that no major errors have occurred. Code for the Blink task is shown in Figure 9.



```
void BlinkTask(void *argument)
{
  /* USER CODE BEGIN StartDefaultTask */
  /* Infinite loop */
    while(1)
    {
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_14);//Toggle LED
        osDelay(1000U);
    }
  /* USER CODE END StartDefaultTask */
}
```

Figure 9: Blink Task Code

ADC Conversion Complete ISR

The ADC Conversion Complete ISR runs every time the MCU's ADCs finish reading the positions of the joysticks. This task first processes the joystick readings and then stores them in an array which is sent to the drone via the radio. Before getting stored in the radio buffer, the raw joystick readings are processed in a moving-average filter to eliminate some high-frequency noise and then scaled to range from 0 – 2047. The scaling is primarily due to the DSHOT protocol on the ESC requiring throttle values between 0 – 2047. Although this means only the throttle readings need to be scaled, all the values are scaled to keep them consistent. Code for this ISR is shown in Figure 37 in the appendix.

Figure 10 illustrates a UML diagram for the ADC Conversion Complete ISR. The *sums_index* variable and signed integer arrays are all used for the moving average filter, which averages the last five joystick readings for throttle, yaw, pitch, and roll. The *adc_val* variables temporarily stores the most recent ADC conversion before it is processed and stored in the *RadioTX* array.

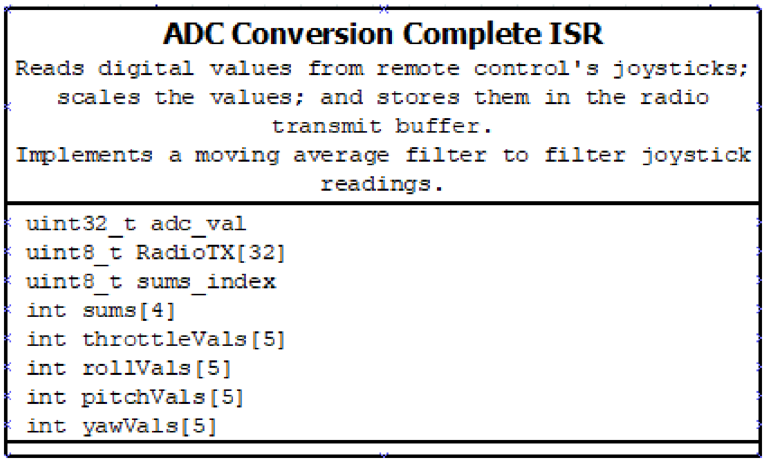
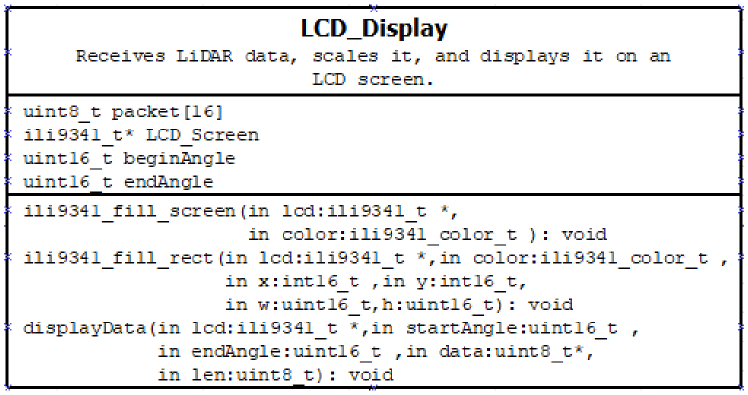

ADC Conversion Complete ISR
Reads digital values from remote control's joysticks; scales the values; and stores them in the radio transmit buffer.
Implements a moving average filter to filter joystick readings.
uint32_t adc_val
uint8_t RadioTX[32]
uint8_t sums_index
int sums[4]
int throttleVals[5]
int rollVals[5]
int pitchVals[5]
int yawVals[5]

Figure 10: ADC Conv. Complete ISR UML

LCD Display Task

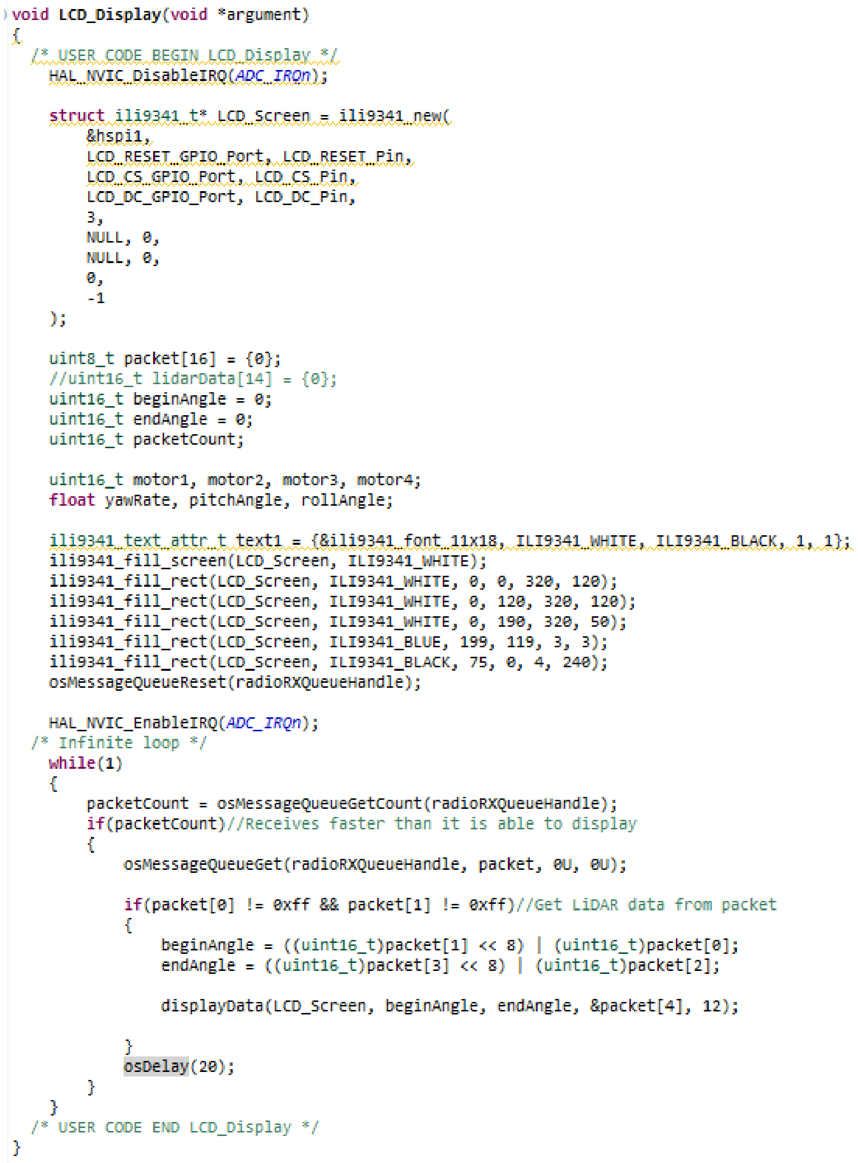
The LCD Display Task receives the 2D LiDAR data from the drone via the radio. Once it has the data, this task parses it into packets and transmits the data to the LCD to display any detected obstacles around the drone. The LCD display task will update the display for every LiDAR packet it receives. These LiDAR packets contain a start angle, end angle, and twelve distance measurements taken between the start and end angles. The display task will use this data to first clear any drawn pixels in the same direction as the new distance data and then draw new pixels to represent the distances to the most recently measured objects.

Figure 11 shows the UML for the LCD Display task. The integer array, *packet*, contains LiDAR data from the quadcopter. The *beginAngle* and *endAngle* variables contain the start and end angles of the LiDAR measurements and are passed into the *displayData* function along with the LiDAR data packet. The *diaplayData* function handles updating the LCD with the most recent LiDAR measurements. The *ili9341_t* struct, *ili9341_fill_rect*, and *ili9341_fill_screen* are part of a C library for the ILI9341 LCD driver [9]. Code for the LCD Display task is shown in Figure 12.



```
LCD_Display
Receives LiDAR data, scales it, and displays it on an
LCD screen.
-----------------------------------------------------
uint8_t packet[16]
ili9341_t* LCD_Screen
uint16_t beginAngle
uint16_t endAngle
-----------------------------------------------------
ili9341_fill_screen(in lcd:ili9341_t *,
                    in color:ili9341_color_t ): void
ili9341_fill_rect(in lcd:ili9341_t *,in color:ili9341_color_t ,
                  in x:int16_t ,in y:int16_t,
                  in w:uint16_t,h:uint16_t): void
displayData(in lcd:ili9341_t *,in startAngle:uint16_t ,
            in endAngle:uint16_t ,in data:uint8_t*,
            in len:uint8_t): void
```

Figure 11: LCD Display Task UML



```
void LCD_Display(void *argument)
{
  /* USER CODE BEGIN LCD_Display */
    HAL_NVIC_DisableIRQ(ADC_IRQn);

    struct ili9341_t* LCD_Screen = ili9341_new(
        &hspi1,
        LCD_RESET_GPIO_Port, LCD_RESET_Pin,
        LCD_CS_GPIO_Port, LCD_CS_Pin,
        LCD_DC_GPIO_Port, LCD_DC_Pin,
        3,
        NULL, 0,
        NULL, 0,
        0,
        -1
    );

    uint8_t packet[16] = {0};
    //uint16_t lidarData[14] = {0};
    uint16_t beginAngle = 0;
    uint16_t endAngle = 0;
    uint16_t packetCount;

    uint16_t motor1, motor2, motor3, motor4;
    float yawRate, pitchAngle, rollAngle;

    ili9341_text_attr_t text1 = {&ili9341_font_11x18, ILI9341_WHITE, ILI9341_BLACK, 1, 1};
    ili9341_fill_screen(LCD_Screen, ILI9341_WHITE);
    ili9341_fill_rect(LCD_Screen, ILI9341_WHITE, 0, 0, 320, 120);
    ili9341_fill_rect(LCD_Screen, ILI9341_WHITE, 0, 120, 320, 120);
    ili9341_fill_rect(LCD_Screen, ILI9341_WHITE, 0, 190, 320, 50);
    ili9341_fill_rect(LCD_Screen, ILI9341_BLUE, 199, 119, 3, 3);
    ili9341_fill_rect(LCD_Screen, ILI9341_BLACK, 75, 0, 4, 240);
    osMessageQueueReset(radioRXQueueHandle);

    HAL_NVIC_EnableIRQ(ADC_IRQn);
  /* Infinite loop */
    while(1)
    {
        packetCount = osMessageQueueGetCount(radioRXQueueHandle);
        if(packetCount)//Receives faster than it is able to display
        {
            osMessageQueueGet(radioRXQueueHandle, packet, 0U, 0U);

            if(packet[0] != 0xff && packet[1] != 0xff)//Get LiDAR data from packet
            {
                beginAngle = ((uint16_t)packet[1] << 8) | (uint16_t)packet[0];
                endAngle = ((uint16_t)packet[3] << 8) | (uint16_t)packet[2];

                displayData(LCD_Screen, beginAngle, endAngle, &packet[4], 12);

            }
            osDelay(20);
        }
    }
  /* USER CODE END LCD_Display */
}
```

Figure 12: LCD Task Code

Radio RX/TX ISR

The Radio RX/TX ISR is triggered whenever the controller's radio receives a packet from the drone. Once a packet containing 2D LiDAR data is received, this task sends the data to the LCD Display task and then transmits and acknowledgement packet, containing the joystick data, to the drone.

Figure 13 is the UML for the Radio RX/TX ISR. *RadioRX* and *RadioTX* are integer buffers for receiving and transmitting data through the radio. The *radioCount* variable is used to count the number of radio transmissions. An LED on the controller is toggled every 300 transmissions. The *status* variable is used to check the status of the NRF24L01P which indicates events and errors. The *nrf24l01p_write_ack_payload* is used to write to the radio's acknowledgement payload which is sent back to the quadcopter along with an acknowledgement signal after each successful reception. Lastly, the *nrf24l01p_rx_receive* is a library function from [12] and is used to receive data transmitted by the quadcopter. Code for the radio ISR is shown in Figure 14, while Figure 15 illustrates code for *nrf24l01p_write_ack_payload*.

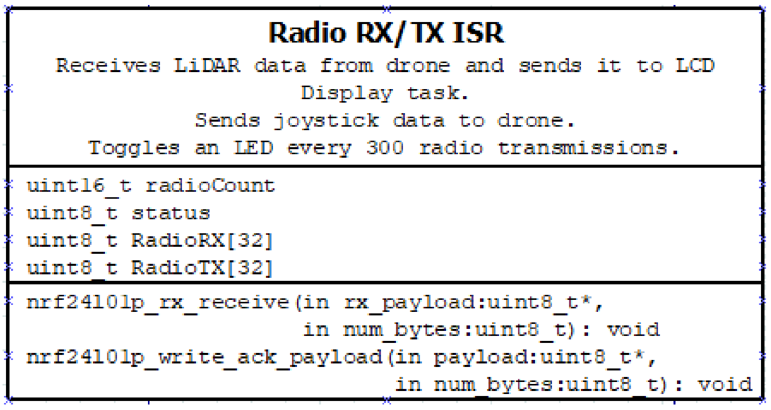
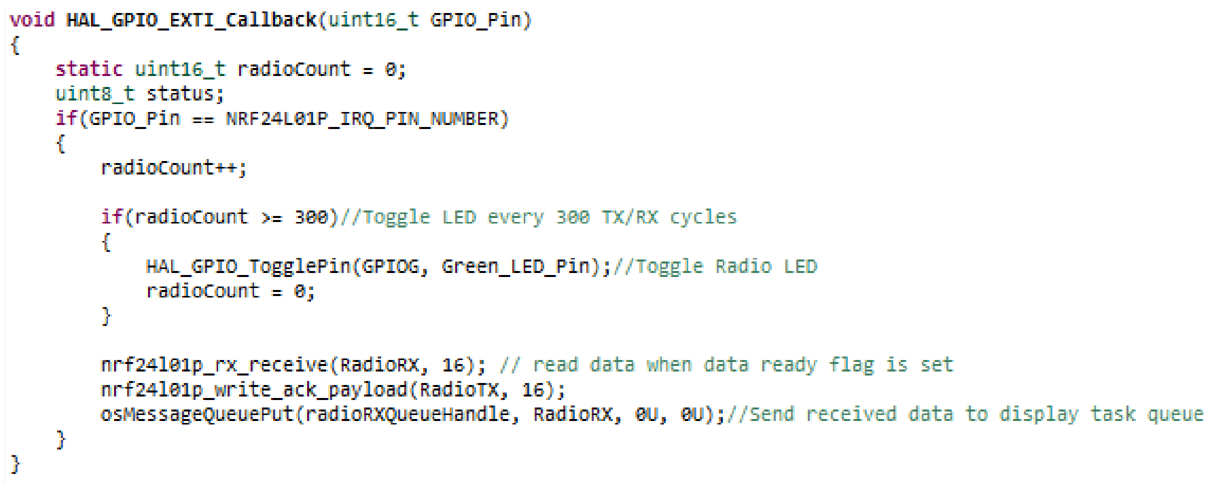
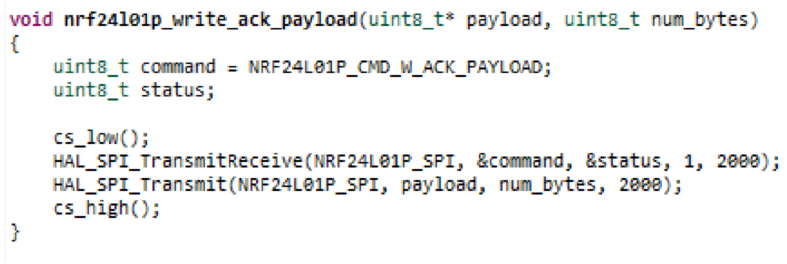

Radio RX/TX ISR
Receives LiDAR data from drone and sends it to LCD Display task.
Sends joystick data to drone.
Toggles an LED every 300 radio transmissions.
uint16_t radioCount
uint8_t status
uint8_t RadioRX[32]
uint8_t RadioTX[32]
nrf24l01p_rx_receive(in rx_payload:uint8_t*, in num_bytes:uint8_t): void
nrf24l01p_write_ack_payload(in payload:uint8_t*, in num_bytes:uint8_t): void

Figure 13: Radio RX/TX ISR UML



```c
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    static uint16_t radioCount = 0;
    uint8_t status;
    if(GPIO_Pin == NRF24L01P_IRQ_PIN_NUMBER)
    {
        radioCount++;

        if(radioCount >= 300)//Toggle LED every 300 TX/RX cycles
        {
            HAL_GPIO_TogglePin(GPIOG, Green_LED_Pin);//Toggle Radio LED
            radioCount = 0;
        }

        nrf24l01p_rx_receive(RadioRX, 16); // read data when data ready flag is set
        nrf24l01p_write_ack_payload(RadioTX, 16);
        osMessageQueuePut(radioRXQueueHandle, RadioRX, 0U, 0U);//Send received data to display task queue
    }
}
```

Figure 14: Controller Radio ISR Code



```c
void nrf24l01p_write_ack_payload(uint8_t* payload, uint8_t num_bytes)
{
    uint8_t command = NRF24L01P_CMD_W_ACK_PAYLOAD;
    uint8_t status;

    cs_low();
    HAL_SPI_TransmitReceive(NRF24L01P_SPI, &command, &status, 1, 2000);
    HAL_SPI_Transmit(NRF24L01P_SPI, payload, num_bytes, 2000);
    cs_high();
}
```

Figure 15: Radio Write Ack Payload Function

Quadcopter

Figure 16 illustrates the software block diagram for the drone. The drone has eight tasks and one ISR.

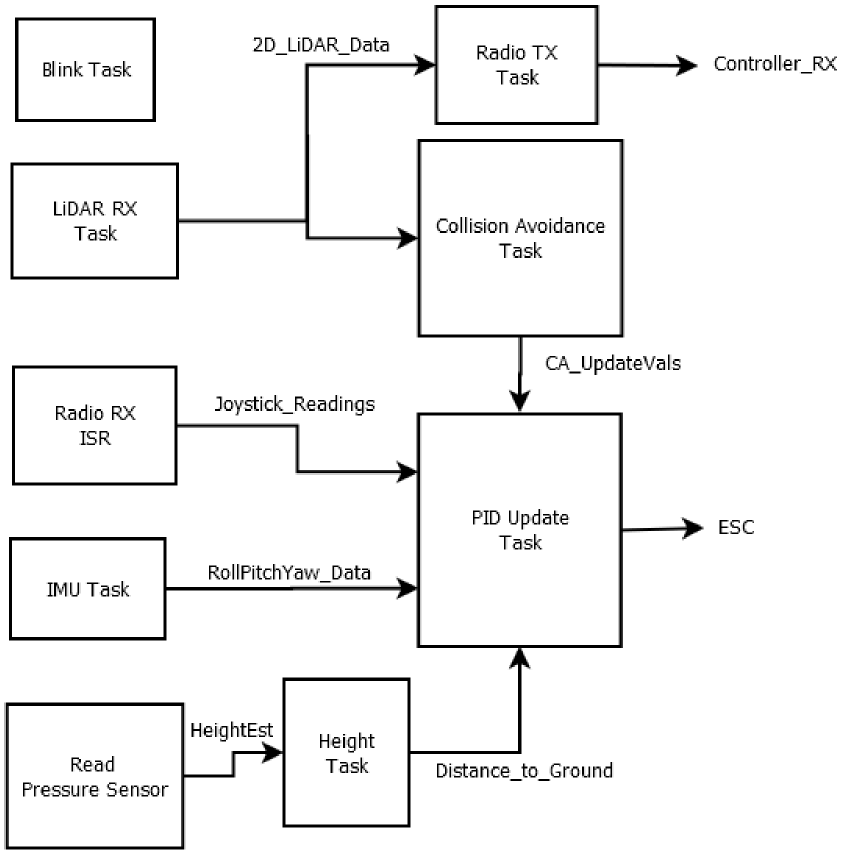


Figure 16: Drone Software Diagram

Blink Task

The drone's Blink task functions exactly like the one in the controller. It simply toggles an onboard LED to indicate that no major errors have occurred.

LiDAR RX Task

The LiDAR RX task receives and analyzes data from the 2D LiDAR. The raw LiDAR data consists of forty-seven bytes containing distance measurements (twelve per packet), angle measurements, rotation rates, light intensity, a CRC, packet header, and packet version. The only data needed from the packet are the distance measurements, angle measurements, and CRC. When raw data is first received, the LiDAR RX task checks to make sure the data is valid by sending it through a CRC function. If the data turns out to be invalid, it is discarded. Otherwise, it is passed into a parsing function to get an array containing only the angle data and distance measurements. The parsed data is then sent to the radio transmitter, and the Collision Avoidance task. Code for the CRC function and parsing function is shown in Figure 36 and Figure 35, respectively. Both figures are in the appendix.

Figure 17 is the UML diagram for the LiDAR RX task. The array, *packet*, stores the raw LiDAR data that is transferred to the task through DMA. The *CalCRC8* function calculates the CRC for the received data packet. Only if the LiDAR data is valid does the *getLiDARMeasurements* function process the LiDAR data into two packets, one containing the data that is sent over the radio to the controller, and another containing data used in the *obstacleDetection* function. The reason for having two packets is that the data sent over the radio is compressed to make the radio transmissions faster, while the full LiDAR data packet is used for obstacle detection. Figure 18 contains the code for *getLiDARMeasurements*. The *obstacleDetection* function performs the first step of the collision avoidance algorithm by updating an array containing all measured distances around the quadcopter and determining the closest obstacles in each sector; this process is explained further in the 'Collision Avoidance Task' section. Figure 42 contains code for the *obstacleDetection* function.

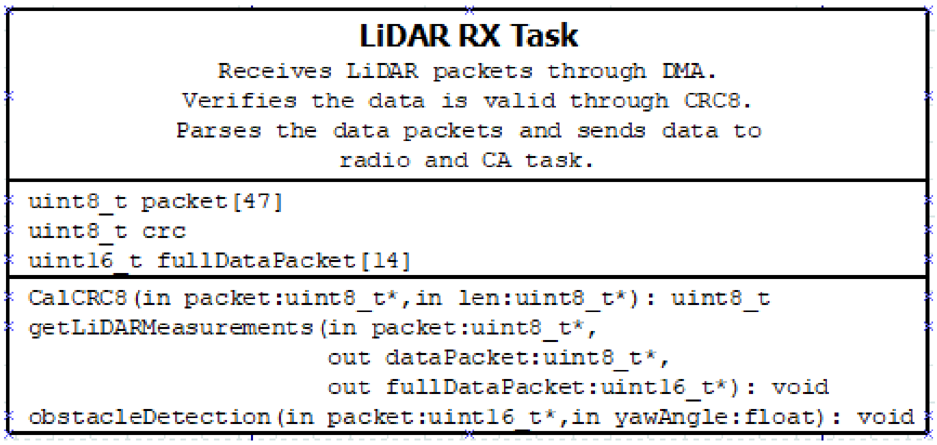
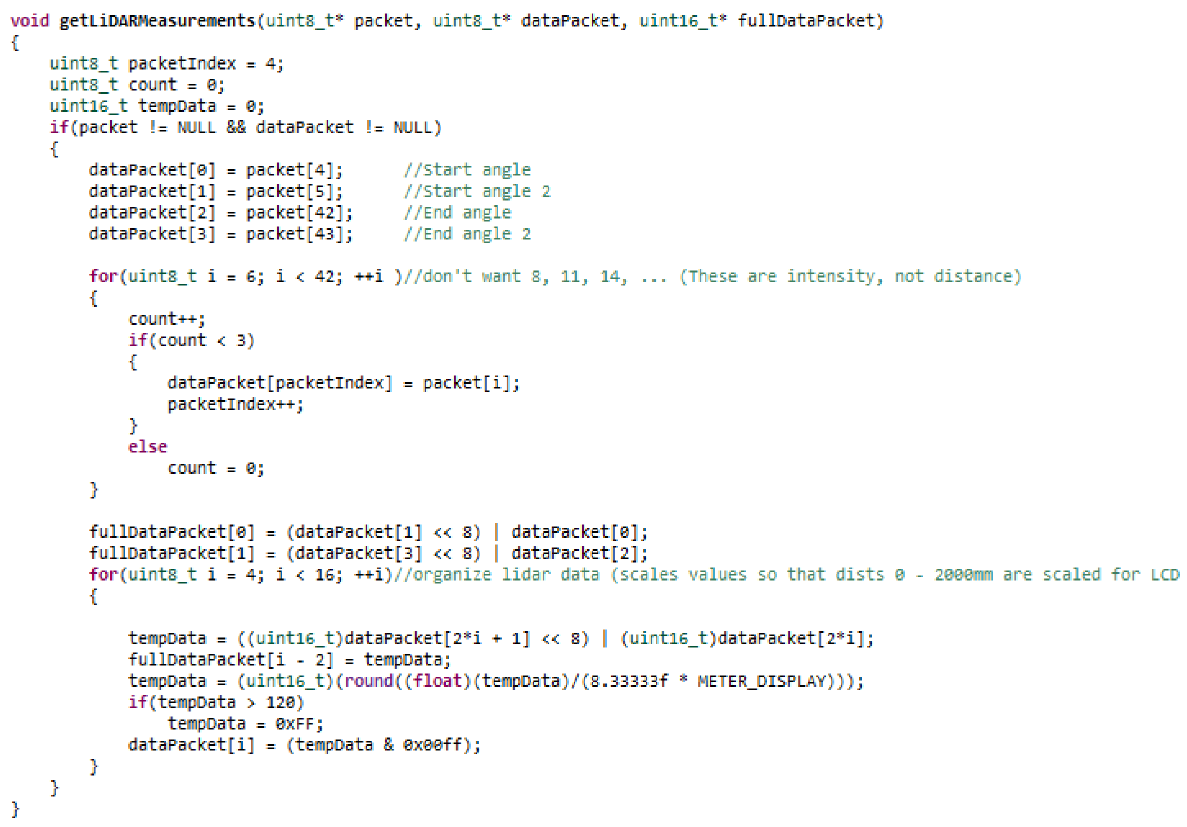

LiDAR RX Task
Receives LiDAR packets through DMA.
Verifies the data is valid through CRC8.
Parses the data packets and sends data to
radio and CA task.
uint8_t packet[47]
uint8_t crc
uint16_t fullDataPacket[14]
CalCRC8(in packet:uint8_t*,in len:uint8_t*): uint8_t
getLiDARMeasurements(in packet:uint8_t*,
out dataPacket:uint8_t*,
out fullDataPacket:uint16_t*): void
obstacleDetection(in packet:uint16_t*,in yawAngle:float): void

Figure 17: 2D LiDAR Task UML



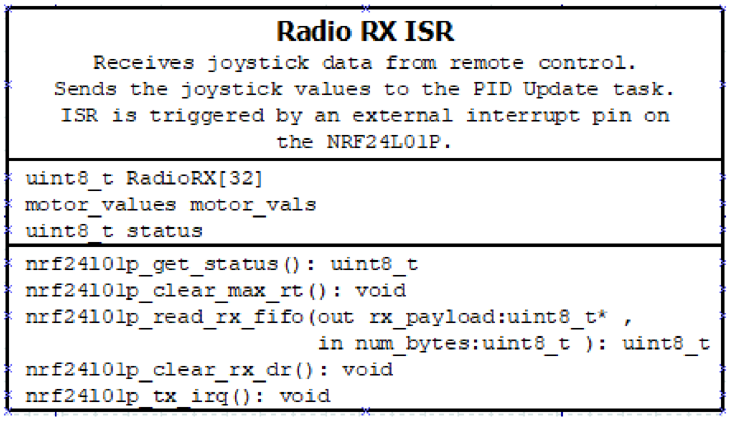
```
void getLiDARMeasurements(uint8_t* packet, uint8_t* dataPacket, uint16_t* fullDataPacket)
{
    uint8_t packetIndex = 4;
    uint8_t count = 0;
    uint16_t tempData = 0;
    if(packet != NULL && dataPacket != NULL)
    {
        dataPacket[0] = packet[4];      //Start angle
        dataPacket[1] = packet[5];      //Start angle 2
        dataPacket[2] = packet[42];     //End angle
        dataPacket[3] = packet[43];     //End angle 2

        for(uint8_t i = 6; i < 42; ++i )//don't want 8, 11, 14, ... (These are intensity, not distance)
        {
            count++;
            if(count < 3)
            {
                dataPacket[packetIndex] = packet[i];
                packetIndex++;
            }
            else
                count = 0;
        }

        fullDataPacket[0] = (dataPacket[1] << 8) | dataPacket[0];
        fullDataPacket[1] = (dataPacket[3] << 8) | dataPacket[2];
        for(uint8_t i = 4; i < 16; ++i)//organize lidar data (scales values so that dists 0 - 2000mm are scaled for LCD
        {

            tempData = ((uint16_t)dataPacket[2*i + 1] << 8) | (uint16_t)dataPacket[2*i];
            fullDataPacket[i - 2] = tempData;
            tempData = (uint16_t)(round((float)(tempData)/(8.33333f * METER_DISPLAY)));
            if(tempData > 120)
                tempData = 0xFF;
            dataPacket[i] = (tempData & 0x00ff);
        }
    }
}
```

Figure 18: LiDAR Get Measurements Function

Radio RX ISR

The Radio RX ISR handles the reception of acknowledgement packets from the controller, which contain joystick data. When a packet is received, it is parsed for the joystick readings, which are then sent to the PID Update task through a queue.

Figure 19 shows the UML diagram for the Radio RX ISR. The *nrf24l01p* functions are from a C library [12] and handle event flags and the data reception from the controller. Like in the controller, the *RadioRX* array is a buffer for storing received data, while the *status* variable stores data read from the radio's status register. The *motor_vals* variable is a struct storing four 16-bit integers; the name does not make sense in this context, but the joystick values are later used in calculations for the motors' throttle values. Code for this ISR is shown in Figure 40.

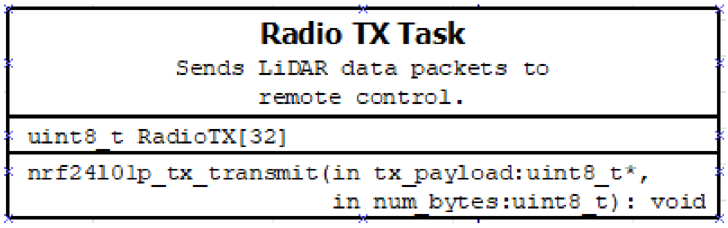

Radio RX ISR
Receives joystick data from remote control.
Sends the joystick values to the PID Update task.
ISR is triggered by an external interrupt pin on
the NRF24L01P.

```
uint8_t RadioRX[32]
motor_values motor_vals
uint8_t status

nrf24l01p_get_status(): uint8_t
nrf24l01p_clear_max_rt(): void
nrf24l01p_read_rx_fifo(out rx_payload:uint8_t* ,
                       in num_bytes:uint8_t ): uint8_t
nrf24l01p_clear_rx_dr(): void
nrf24l01p_tx_irq(): void
```

Figure 19: Radio RX ISR

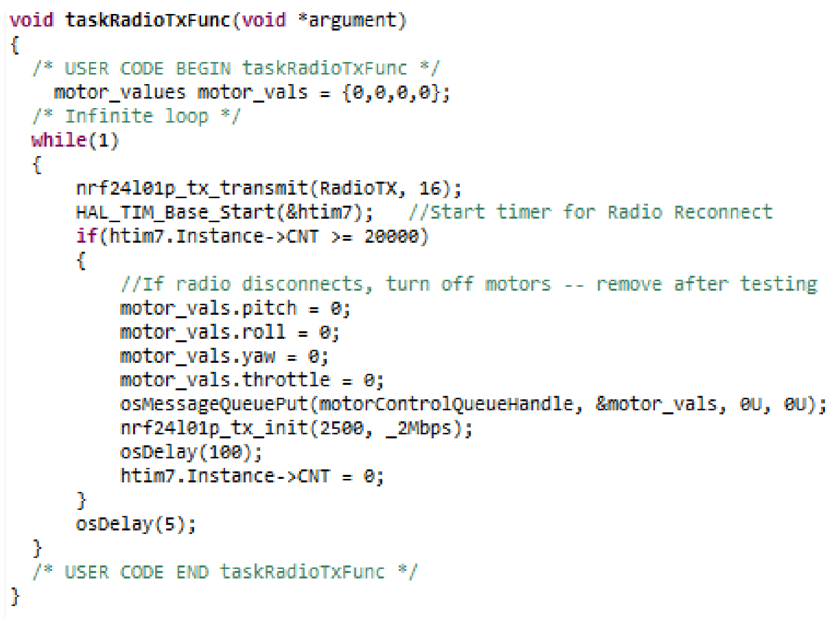
Radio TX Task

The Radio TX Task runs every few milliseconds and transmits whatever LiDAR data is stored in its buffer over to the controller. Figure 20 shows the UML for this task. The *nrf24l01p_tx_trasnmit* function is also from the C library for the NRF24L01P [12], and handles transmitting the data in the *RadioTX* buffer. The code for this task is shown in Figure 21.



```
uint8_t RadioTX[32]

nrf24l01p_tx_transmit(in tx_payload:uint8_t*,
                      in num_bytes:uint8_t): void
```

Figure 20: Radio TX Task UML



```
void taskRadioTxFunc(void *argument)
{
  /* USER CODE BEGIN taskRadioTxFunc */
    motor_values motor_vals = {0,0,0,0};
  /* Infinite loop */
  while(1)
  {
      nrf24l01p_tx_transmit(RadioTX, 16);
      HAL_TIM_Base_Start(&htim7);   //Start timer for Radio Reconnect
      if(htim7.Instance->CNT >= 20000)
      {
          //If radio disconnects, turn off motors -- remove after testing
          motor_vals.pitch = 0;
          motor_vals.roll = 0;
          motor_vals.yaw = 0;
          motor_vals.throttle = 0;
          osMessageQueuePut(motorControlQueueHandle, &motor_vals, 0U, 0U);
          nrf24l01p_tx_init(2500, _2Mbps);
          osDelay(100);
          htim7.Instance->CNT = 0;
      }
      osDelay(5);
  }
  /* USER CODE END taskRadioTxFunc */
}
```

Figure 21: Drone Radio TX Task Code

Collision Avoidance Task

The Collision Avoidance task implements two of the three stages in the collision avoidance algorithm. As mentioned previously, the three stages are obstacle detection, hazard assessment, and obstacle avoidance. The obstacle detection step is performed in the LiDAR RX task, while hazard assessment and obstacle avoidance are performed in the Collision Avoidance task.

In this algorithm, the area around the drone is divided into sectors and each sector is assigned one PID controller that controls the distance between the drone and the closest obstacle its sector. The setpoint for each controller is the minimum distance obstacles can be from the drone before it tries to avoid them, while the current distance (the PID measurement) is the distance measured by the 2D LiDAR.

A diagram illustrating the features of the collision avoidance algorithm is shown in Figure 22. In this diagram, the area around the drone is divided into eight sectors. The hazard assessment stage determines which of the obstacles the drone is most likely to collide with based on the drone's distance to the obstacles, while the obstacle avoidance stage calculates update values for the drone's motors using motor-mixing equations.

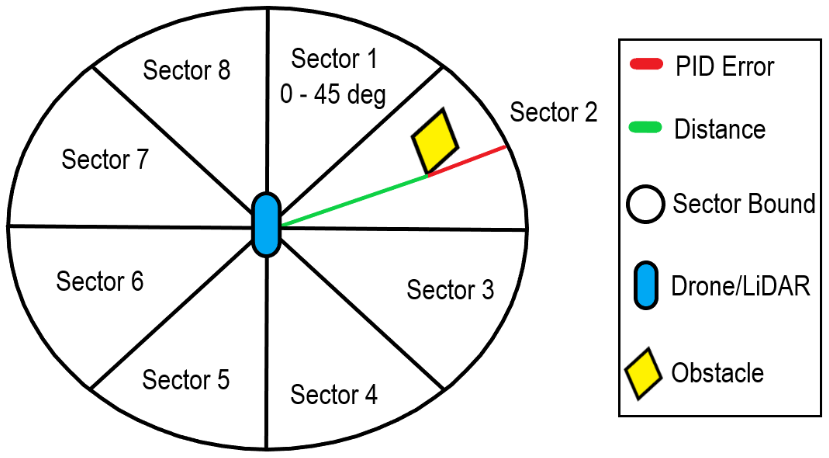
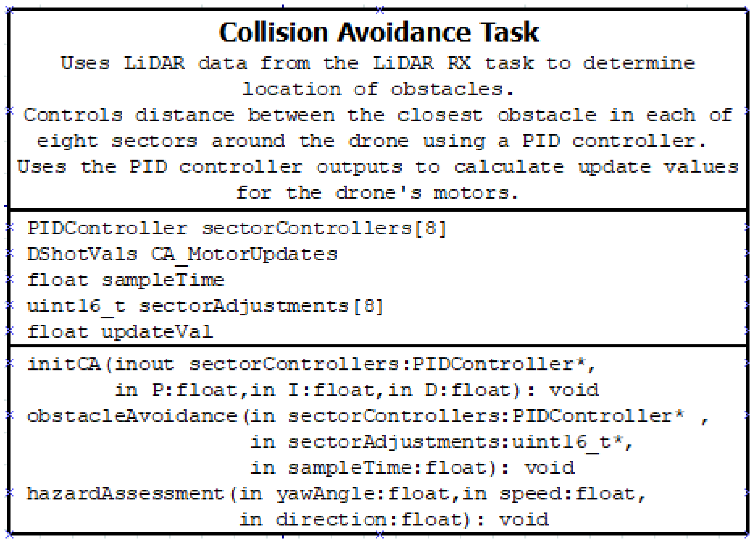


Figure 22: Collision Avoidance Algorithm Diagram

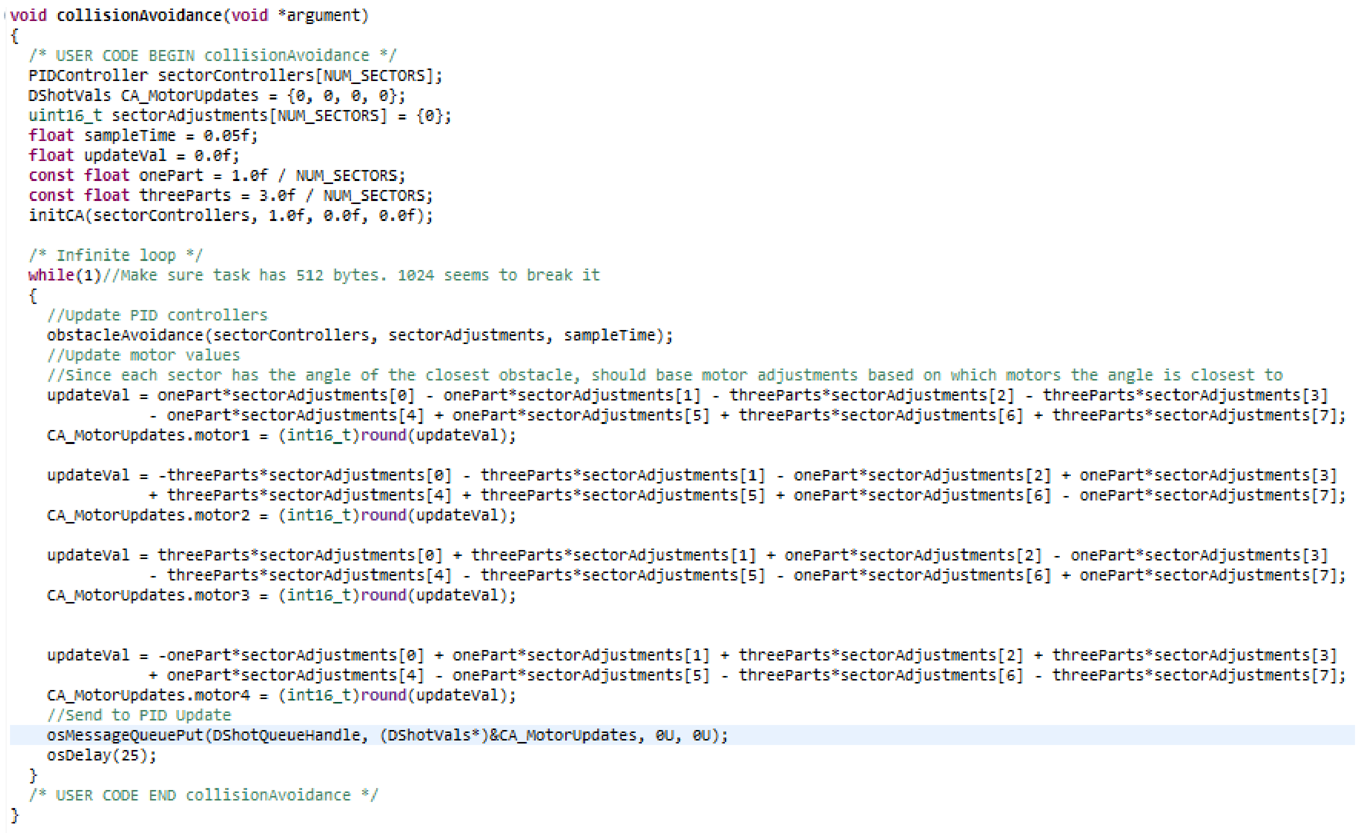
Figure 23 is the Collision Avoidance task UML diagram. The *initCA* function initializes the algorithm's distance-measurement buffer, which contains all measured points around the drone. It also initializes the PID controllers for each sector (*sectorControllers[8]*), and structures containing the closest obstacle in each sector. The closest obstacle for each sector is recorded as a polar-coordinate pair. The *obstacleAvoidance* function calculates update values for each sector by sending the closest distance in each sector into PID update functions and storing the output in the *sectorAdjustments* array. The values in this array are summed together to find the final adjustments for each of the four BLDC motors. Code for *initCA* is shown in Figure 42, and code for *obstacleAvoidance* is shown in Figure 43.



```
Collision Avoidance Task
Uses LiDAR data from the LiDAR RX task to determine
location of obstacles.
Controls distance between the closest obstacle in each of
eight sectors around the drone using a PID controller.
Uses the PID controller outputs to calculate update values
for the drone's motors.

PIDController sectorControllers[8]
DShotVals CA_MotorUpdates
float sampleTime
uint16_t sectorAdjustments[8]
float updateVal

initCA(inout sectorControllers:PIDController*,
       in P:float,in I:float,in D:float): void
obstacleAvoidance(in sectorControllers:PIDController* ,
                  in sectorAdjustments:uint16_t*,
                  in sampleTime:float): void
hazardAssessment(in yawAngle:float,in speed:float,
                 in direction:float): void
```

Figure 23: Collision Avoidance Task UML

Figure 24 shows the code for the Collision Avoidance task. After *obstacleAvoidance* is called, the task creates update values for the four motors through motor-mixing equations. These equations add portions of the updates from each of the PID controllers to the motor throttles. Motors that are closer to a sector get a larger portion of the update-values for that sector. Like the flight controller throttle updates, complimentary motors are updated with values that are the same magnitude, but opposite signs. For example, if there is an obstacle in the first sector, motor three would get the largest update value added to its throttle since it is the closest motor, while motor two, the one furthest away, gets that same update value subtracted from its throttle.



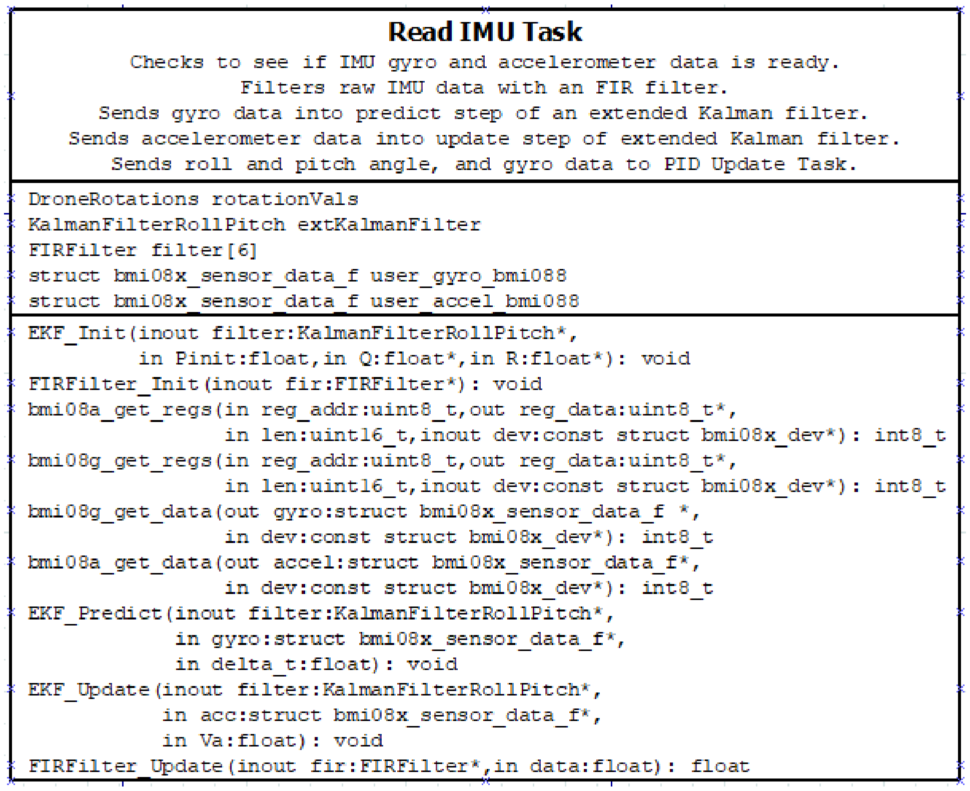
```
void collisionAvoidance(void *argument)
{
  /* USER CODE BEGIN collisionAvoidance */
  PIDController sectorControllers[NUM_SECTORS];
  DShotVals CA_MotorUpdates = {0, 0, 0, 0};
  uint16_t sectorAdjustments[NUM_SECTORS] = {0};
  float sampleTime = 0.05f;
  float updateVal = 0.0f;
  const float onePart = 1.0f / NUM_SECTORS;
  const float threeParts = 3.0f / NUM_SECTORS;
  initCA(sectorControllers, 1.0f, 0.0f, 0.0f);

  /* Infinite loop */
  while(1)//Make sure task has 512 bytes. 1024 seems to break it
  {
    //Update PID controllers
    obstacleAvoidance(sectorControllers, sectorAdjustments, sampleTime);
    //Update motor values
    //Since each sector has the angle of the closest obstacle, should base motor adjustments based on which motors the angle is closest to
    updateVal = onePart*sectorAdjustments[0] - onePart*sectorAdjustments[1] - threeParts*sectorAdjustments[2] - threeParts*sectorAdjustments[3]
              - onePart*sectorAdjustments[4] + onePart*sectorAdjustments[5] + threeParts*sectorAdjustments[6] + threeParts*sectorAdjustments[7];
    CA_MotorUpdates.motor1 = (int16_t)round(updateVal);

    updateVal = -threeParts*sectorAdjustments[0] - threeParts*sectorAdjustments[1] - onePart*sectorAdjustments[2] + onePart*sectorAdjustments[3]
              + threeParts*sectorAdjustments[4] + threeParts*sectorAdjustments[5] + onePart*sectorAdjustments[6] - onePart*sectorAdjustments[7];
    CA_MotorUpdates.motor2 = (int16_t)round(updateVal);

    updateVal = threeParts*sectorAdjustments[0] + threeParts*sectorAdjustments[1] + onePart*sectorAdjustments[2] - onePart*sectorAdjustments[3]
              - threeParts*sectorAdjustments[4] - threeParts*sectorAdjustments[5] - onePart*sectorAdjustments[6] + onePart*sectorAdjustments[7];
    CA_MotorUpdates.motor3 = (int16_t)round(updateVal);


    updateVal = -onePart*sectorAdjustments[0] + onePart*sectorAdjustments[1] + threeParts*sectorAdjustments[2] + threeParts*sectorAdjustments[3]
              + onePart*sectorAdjustments[4] - onePart*sectorAdjustments[5] - threeParts*sectorAdjustments[6] - threeParts*sectorAdjustments[7];
    CA_MotorUpdates.motor4 = (int16_t)round(updateVal);
    //Send to PID Update
    osMessageQueuePut(DShotQueueHandle, (DShotVals*)&CA_MotorUpdates, 0U, 0U);
    osDelay(25);
  }
  /* USER CODE END collisionAvoidance */
}
```

Figure 24: Collision Avoidance Task Code

Read IMU Task

The Read IMU task handles sending commands and receiving data from the BMI088 IMU. Once the data is collected, it is filtered and then sent to the PID Update task to be used in the PID control equations. The raw IMU data goes through two filters, an FIR filter and an extended Kalman filter. The FIR filter is a low-pass filter that eliminates some of the high-frequency noise from the data. The Kalman filter is used to estimate the current roll and pitch angle of the IMU using the rotation rates from the gyroscope and the linear acceleration from the accelerometer. Once the data is done being processed by the Kalman filter, the roll angle, pitch angle, and yaw rotation rate are sent to the PID Update task. The IMU task code is shown in Figure 39 in the appendix.

Figure 25 is the UML diagram for the Read IMU task. The *bmi08* functions are from a C library for the BMI088 [10]. These functions are for reading and writing to the BMI088's registers, including the data registers for the accelerometer and gyroscope data. *FIRFilter_Init, FIRFilter_Update,* and the *FIRFilter* struct are all used in the low-pass FIR filter which is used to filter the raw IMU data before sending it into the extended Kalman filter. The *KalmanFilterRollPitch* struct stores variables needed for the extended Kalman filter, including the predicted roll and pitch angles. *EKF_Predict* and *EKF_Update* perform the core operations of the extended Kalman filter and output the estimated roll and pitch angles of the quadcopter. These values, along with the yaw rotation rate, are stored in the *rotationVals* struct (which is just three floats) and sent to the PID Update task. The FIR filter code is from [8], while the extended Kalman filter code is adapted from [7] and [14].

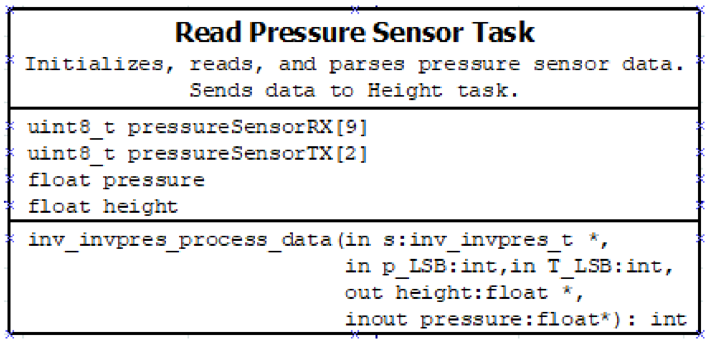

Read IMU Task

```
Checks to see if IMU gyro and accelerometer data is ready.
Filters raw IMU data with an FIR filter.
Sends gyro data into predict step of an extended Kalman filter.
Sends accelerometer data into update step of extended Kalman filter.
Sends roll and pitch angle, and gyro data to PID Update Task.

DroneRotations rotationVals
KalmanFilterRollPitch extKalmanFilter
FIRFilter filter[6]
struct bmi08x_sensor_data_f user_gyro_bmi088
struct bmi08x_sensor_data_f user_accel_bmi088

EKF_Init(inout filter:KalmanFilterRollPitch*,
         in Pinit:float,in Q:float*,in R:float*): void
FIRFilter_Init(inout fir:FIRFilter*): void
bmi08a_get_regs(in reg_addr:uint8_t,out reg_data:uint8_t*,
                in len:uint16_t,inout dev:const struct bmi08x_dev*): int8_t
bmi08g_get_regs(in reg_addr:uint8_t,out reg_data:uint8_t*,
                in len:uint16_t,inout dev:const struct bmi08x_dev*): int8_t
bmi08g_get_data(out gyro:struct bmi08x_sensor_data_f *,
                in dev:const struct bmi08x_dev*): int8_t
bmi08a_get_data(out accel:struct bmi08x_sensor_data_f*,
                in dev:const struct bmi08x_dev*): int8_t
EKF_Predict(inout filter:KalmanFilterRollPitch*,
            in gyro:struct bmi08x_sensor_data_f*,
            in delta_t:float): void
EKF_Update(inout filter:KalmanFilterRollPitch*,
           in acc:struct bmi08x_sensor_data_f*,
           in Va:float): void
FIRFilter_Update(inout fir:FIRFilter*,in data:float): float
```

Figure 25: Read IMU Task

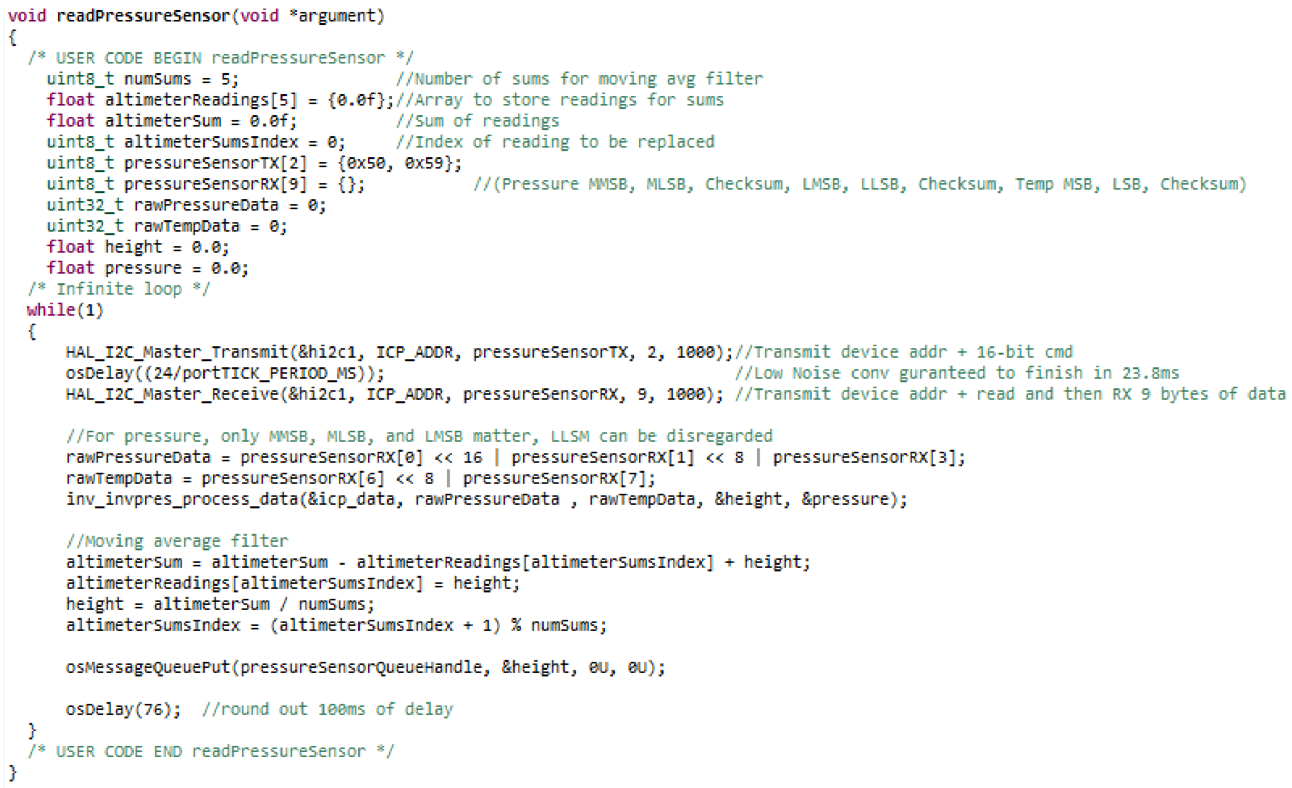
Read Pressure Sensor Task

The Read Pressure Sensor task reads pressure data from the drone's pressure sensor and converts the data into height estimations. The estimated height is sent to the Height task through a queue. Figure 26 shows the UML for this task. The *inv_invpres_process_data* function is from the ICP-10111 datasheet [13], and uses the data read from the pressure sensor to estimate the atmospheric pressure and height, which are stored in the *pressure* and *height* float variables. The estimated height is stored in meters. Figure 27 shows the code for the Read Pressure Sensor task.



```
Read Pressure Sensor Task
Initializes, reads, and parses pressure sensor data.
Sends data to Height task.

uint8_t pressureSensorRX[9]
uint8_t pressureSensorTX[2]
float pressure
float height

inv_invpres_process_data(in s:inv_invpres_t *,
                         in p_LSB:int,in T_LSB:int,
                         out height:float *,
                         inout pressure:float*): int
```

Figure 26: Read Pressure Sensor Task



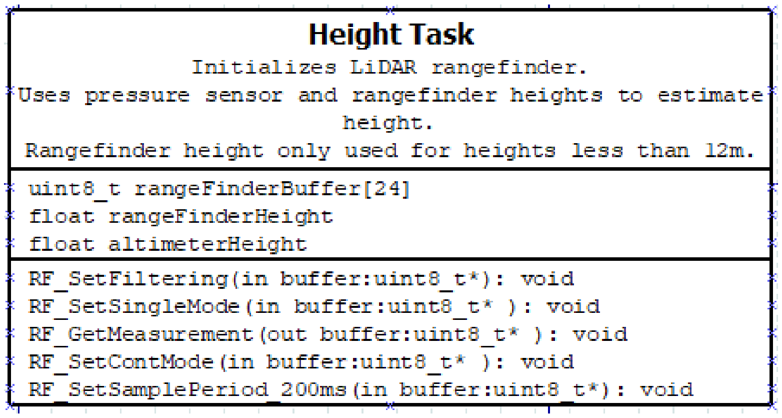
```
void readPressureSensor(void *argument)
{
  /* USER CODE BEGIN readPressureSensor */
    uint8_t numSums = 5;                //Number of sums for moving avg filter
    float altimeterReadings[5] = {0.0f};//Array to store readings for sums
    float altimeterSum = 0.0f;          //Sum of readings
    uint8_t altimeterSumsIndex = 0;     //Index of reading to be replaced
    uint8_t pressureSensorTX[2] = {0x50, 0x59};
    uint8_t pressureSensorRX[9] = {};           //(Pressure MMSB, MLSB, Checksum, LMSB, LLSB, Checksum, Temp MSB, LSB, Checksum)
    uint32_t rawPressureData = 0;
    uint32_t rawTempData = 0;
    float height = 0.0;
    float pressure = 0.0;
  /* Infinite loop */
  while(1)
  {
      HAL_I2C_Master_Transmit(&hi2c1, ICP_ADDR, pressureSensorTX, 2, 1000);//Transmit device addr + 16-bit cmd
      osDelay((24/portTICK_PERIOD_MS));                                    //Low Noise conv guranteed to finish in 23.8ms
      HAL_I2C_Master_Receive(&hi2c1, ICP_ADDR, pressureSensorRX, 9, 1000); //Transmit device addr + read and then RX 9 bytes of data

      //For pressure, only MMSB, MLSB, and LMSB matter, LLSM can be disregarded
      rawPressureData = pressureSensorRX[0] << 16 | pressureSensorRX[1] << 8 | pressureSensorRX[3];
      rawTempData = pressureSensorRX[6] << 8 | pressureSensorRX[7];
      inv_invpres_process_data(&icp_data, rawPressureData , rawTempData, &height, &pressure);

      //Moving average filter
      altimeterSum = altimeterSum - altimeterReadings[altimeterSumsIndex] + height;
      altimeterReadings[altimeterSumsIndex] = height;
      height = altimeterSum / numSums;
      altimeterSumsIndex = (altimeterSumsIndex + 1) % numSums;

      osMessageQueuePut(pressureSensorQueueHandle, &height, 0U, 0U);

      osDelay(76);  //round out 100ms of delay
  }
  /* USER CODE END readPressureSensor */
}
```

Figure 27: Read Pressure Sensor Task Code

Height Task

The Height task receives height estimates from the Read Pressure Sensor task and the LiDAR rangefinder, and then combines the measurements to calculate an estimate for the drone's distance from the ground. The first ten measurements of the rangefinder are used to zero the pressure sensors height readings so that the pressure sensor readings are relative to the drone's starting height rather than sea level. The height estimate favors the rangefinder within six meters of the ground but begins to favor the pressure sensor until 12 meters, where the rangefinder is ignored entirely. The estimated height of the drone is sent to the PID Update task, though it is not used for anything yet.

Figure 28 illustrates the Height task UML diagram. The 'RF' functions are used to initialize the rangefinder so that it measures every 200ms. The *rangefinderBuffer* contains the raw data read from the rangefinder; this data is used to calculate the measured distance in meters, which is then stored in *rangeFinderHeight*. The *altimeterHeight* variable contains the pressure sensor height estimate received from the Read Pressure Sensor task. Code for the rangefinder functions are illustrated in Figure 41 and code for the task itself is shown in Figure 44; both figures are in the appendix.



Height Task
Initializes LiDAR rangefinder.
Uses pressure sensor and rangefinder heights to estimate height.
Rangefinder height only used for heights less than 12m.
uint8_t rangeFinderBuffer[24]
float rangeFinderHeight
float altimeterHeight
RF_SetFiltering(in buffer:uint8_t*): void
RF_SetSingleMode(in buffer:uint8_t*): void
RF_GetMeasurement(out buffer:uint8_t*): void
RF_SetContMode(in buffer:uint8_t*): void
RF_SetSamplePeriod_200ms(in buffer:uint8_t*): void

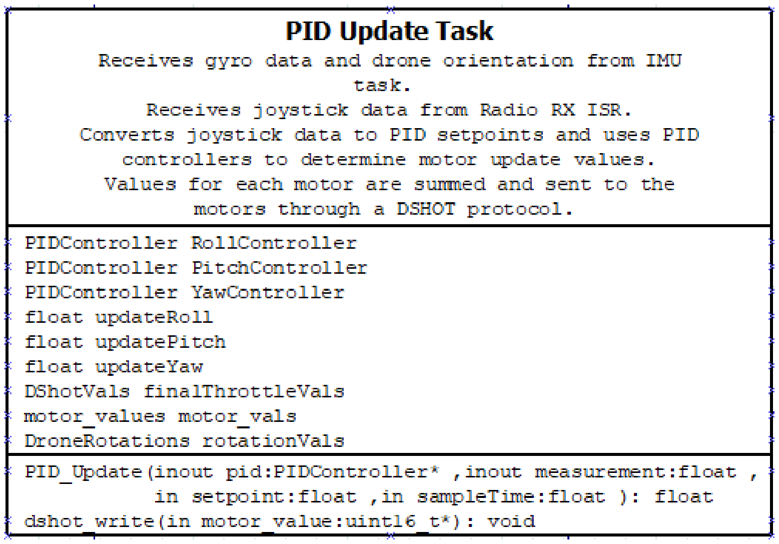
Figure 28: Height Task UML

PID Update Task

The PID Update task receives data from the other tasks and uses that data to adjust the throttle values of each BLDC motor. The IMU and joystick data are used in PID control equations to control the flight and balance of the drone; the data from the Collision Avoidance task is used along with the IMU data to determine if the drone is going to collide with an obstacle; and the height of the drone may eventually be used for hovering and avoiding collisions with the ground. Once all the data is received and motor update values are calculated, the PID Update task sends four DSHOT signals to the ESC which then drives the four motors to adjust the movement of the drone.

The drone's flight controller consists of three PID controllers, one for the roll angle, another for the pitch angle, and the third for yaw rotation rate. The PID controllers take in the IMU data as the drone's current state and control to setpoints calculated from the joystick control values. Code for the PID controllers is shown in Figure 38 in the appendix; these equations are adapted from [5].

Figure 29 shows the UML diagram for the PID Update task. The *PIDController* structs are used to store the gain-values of each controller along with the previous measurement, previous error, and other terms used in the *PID_Update* function. *PID_Update* calculates adjustments to the four BLDC motors using a setpoint (joystick data) and a measurement (IMU data). These adjustments are stored in the update variables for roll, pitch, and yaw, which are later summed with the base throttle taken directly from the throttle joystick value. The final throttle values are written to the ESC through the *dshot_write* function, which is from a DSHOT protocol C library [11].



PID Update Task

```
Receives gyro data and drone orientation from IMU
task.
Receives joystick data from Radio RX ISR.
Converts joystick data to PID setpoints and uses PID
controllers to determine motor update values.
Values for each motor are summed and sent to the
motors through a DSHOT protocol.
```

```
PIDController RollController
PIDController PitchController
PIDController YawController
float updateRoll
float updatePitch
float updateYaw
DShotVals finalThrottleVals
motor_values motor_vals
DroneRotations rotationVals
```

```
PID_Update(inout pid:PIDController* ,inout measurement:float ,
         in setpoint:float ,in sampleTime:float ): float
dshot_write(in motor_value:uint16_t*): void
```

Figure 29: PID Update Task UML

Encountered Problems

One problem encountered throughout this project was difficulty communicating with the IMU, which caused multiple tasks to be delayed by a couple of weeks. This problem was solved by switching from SPI communication to I2C. Another issue was that the vibrations from the quadcopter's motors caused too much noise in the IMU accelerometer readings. This problem was resolved after adjusting the parameters of the IMU's built-in low-pass filter. The IMU data rate is slower, but it blocks out much more high-frequency noise. A third issue was that the 2D LiDAR would toggle on and off during operation. The issue seems to have been lack of power since it stopped after moving the LiDAR to its own buck converter.

The most significant problem encountered throughout this project was the instability of the drone's flight controller observed during test flights and PID tuning. At high-enough throttle values (200+), the quadcopter oscillates on the x and y-axis. These oscillations grow until the quadcopter nearly flips upside down. This issue occurs even when the PID controllers are turned off, which indicates it may be an issue with the motors or the design of the drone frame. This problem has not been solved even after redesigning and rebuilding the drone frame, debugging the IMU and flight-control software, and implementing angular rate PID controllers instead of angle controllers (also tried angular rate and angle controllers combined). Regardless of what was changed, the oscillations seemed to remain the same.

Assembled Project

Figure 30 shows the assembled quadcopter drone. Overall, the drone weighs approximately three pounds. All the components are wired together inside the central compartment and the wires connecting the motors to the ESC run through the drone's hollow limbs. Although not visible in the picture, the rangefinder is mounted on the underside of the drone, pointing downwards.

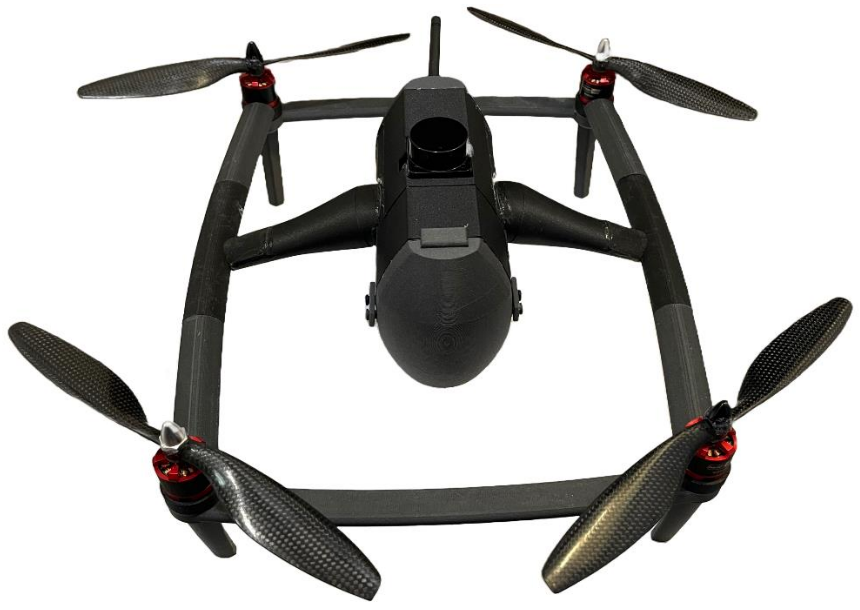


Figure 30: Assembled Quadcopter Drone

Figure 31 and Figure 32 show the assembled controller. Like the quadcopter, all the components are wired together on the inside. The face of the controller shows the LCD, joysticks, a power switch, and two LEDs (red for power, green for radio connection). The front of the controller has a hole for the controller's battery and antenna.

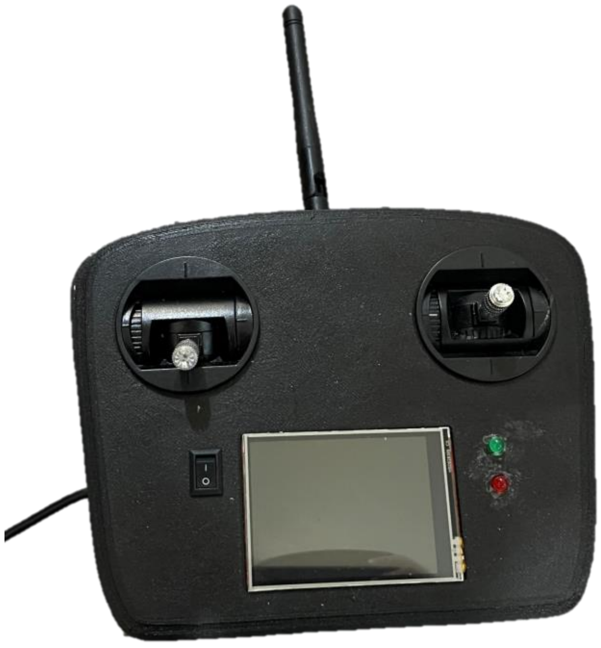


Figure 31: Assembled Controller - Top-Down

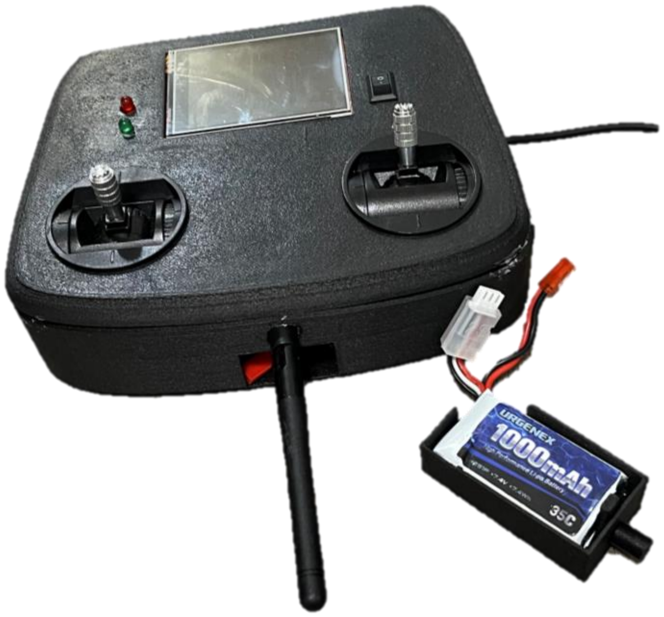


Figure 32: Assembled Controller – Front

Testing

To prove that the project met the proposed requirements, a series of tests were performed to test the related functions of the drone and controller. Many of the requirements had overlap in their functionality. For most requirements, videos were used as proof. Requirements 11, 15, and 16 were stretch goals, and were not implemented. Table 18 shows the requirement, test method, and results from the tests performed. Overall, nearly every requirement was successfully tested and proven through data, images, or videos. The only requirement test that did not completely pass was requirement five, because the radio communication between the drone and controller was heavily delayed when the controller was powered by battery.

Req. #	***Description***	***Test methodology***	***Results / Notes***
1	Drone has 4 propellers and BLDC motors used for flight. Drone is custom-built for the project.	Images of the drone will be used to show that it is built from custom parts, and that it has four motors and propellers.	**Test Passed.** Included images of constructed drone as proof.
2	Remote control is custom-built and used to operate the drone from a distance.	Use the controller to change the throttle, yaw, pitch, and roll controls on the drone.	**Test Passed.** Used images and videos from requirement seven as proof since they overlap.
3	Drone and controller have separate microcontrollers. The MCUs are fast-enough to process necessary data and respond in near real-time.	The MCUs can be verified by looking at the internals of the drone and controller. The display speed on the controller, and the drone's responsiveness to joystick controls and collision avoidance can be used to verify that the MCUs are fast-enough. Images/Videos will be used as proof.	**Test Passed.** All videos and images were taken with everything running on both the drone and controller. They showcase the PID controllers, joystick reading, IMU data, collision avoidance, LiDAR data display, and radio communication.
4	The drone and controller communicate over radio at least 25 feet away.	Transmit data between the drone and controller while they are at least 25 feet away and check if the data	**Test Passed.** A video was taken of the controller operating the drone at about 35 feet.

		received on each end is correct.	
5	The drone and controller can be powered by rechargeable batteries.	Power-on and operate the drone while both the drone and controller are powered by battery. Images/Videos will be used as proof.	**Test partially passed.** For some reason radio communication runs very slowly when the controller is powered by battery.
6	The controller has an LCD screen that displays obstacles around the drone and updates at least 15 times a second.	Place obstacles around the drone and verify that the LCD updates to correctly show the new objects. The refresh rate of the LCD will be tested by timing how often the LCD task runs (# times ran / time) and comparing the measured value with the required value.	**Test Passed.** Timer and counter in LCD task were used to measure the LCD update rate. The CSV file has all data sent to serial monitor and the average refresh rate. The average was over 15 updates per second. Video shows LiDAR data display.
7	The controller has two joysticks read by the MCU's ADCs: one for throttle and yaw, and one for pitch and roll.	Each joystick will be tested individually to control the drone to prove that one controls throttle and yaw, and the other controls pitch and roll.	**Test Passed.** Included image of constructed controller and videos for each test.
8	The drone uses a rotating LiDAR to detect obstacles. The LiDAR is active while drone is in flight. The LiDAR can detect obstacles three meters away.	Show that the rotating LiDAR data is being used to determine the location of obstacles. Verify that the LiDAR continues to run when the drone motors are active. Set obstacles three meters away from the drone and verify that it can detect them.	**Test Passed.** One video demonstrates the motor updates from the collision avoidance algorithm for obstacles around the drone. Another video demonstrates that the drone can detect obstacles over three meters away. The rotating LiDAR is visible in both videos.
9	The drone's flight controller uses a 3-axis gyro and 3-axis accelerometer IMU to calculate adjustments to the motors' throttles.	Record and display data from the IMU and verify that the drone's PID controllers use that data to balance the drone.	**Test Passed.** Added images of code for EKF using IMU data and PID controllers. Videos are of pure rate controller. Serial data is from angle controller.
10	The drone attempts to avoid obstacles it senses it will collide with.	Place obstacles around the drone and verify that the motor adjustments calculated by the collision avoidance (CA) algorithm will move the drone away from the objects.	**Test Passed.** Video demonstrates the motor adjustments from the collision avoidance algorithm. The motors closest to the obstacle speed up when it gets too close (this would tilt the drone away from the object). The image shows serial data with the PID updates, setpoints, CA motor updates,

			current angle, and angular rate of the drone.
12	The drone is controlled through the controller, but automatically attempts to avoid obstacles. (Semi-autonomous mode)	Show that the collision-avoidance algorithm outputs motor adjustments to avoid obstacles while the user is controlling the drone through the controller.	**Test Passed.** Video demos CA motor adjustments to avoid obstacles while the drone is being controlled through the controller. The collision avoidance speeds up the motors closest to the object, which are normally stopped because the controller is telling the drone to pitch forward. The images show serial data for collision avoidance with a front obstacle while the drone is being told to pitch and roll.
13	Drone estimates its distance from the ground with a range finder.	Use the rangefinder to measure the drone's height at known distances and verify that the measurements are correct.	**Test Passed.** Included images of setup and data. Raw data was averaged. Needed multiple trials to adjust pressure sensor offset
14	Drone estimates its distance from the ground with a pressure sensor. Pressure sensor and rangefinder data may be combined into a single height estimate.	Use the pressure sensor to measure the drone's height at known distances and verify that the measurements are correct.	**Test Passed.** Tested alongside rangefinder (requirement 13). Rangefinder tended to overestimate height, while pressure sensor tended to underestimate it. The combined measurements became more accurate due to this.
17	Most of the drone and controller housings are 3D printed.	The drone and controller housing can be verified visually. Images/videos of printing and the 3D models will be included as proof. STL files for the parts will be included too.	**Test Passed.** Included STL files for drone and controller. Included images of the 3D models. Included image and video of drone leg printing.

Table 18: Requirement Testing Data

Conclusion

Overall, the project was fairly successful. Despite the continued issues with the destabilization of the drone's flight controller, most of the requirements were implemented and successfully tested. As a research project, this project was especially successful, since a lot was learned about designing and building quadcopters, developing collision avoidance algorithms, and using a 2D LiDAR. If this project were to be done again, more time would be spent in the early stages of development to create a model for the drone that can be controlled by the PID controllers. This would have potentially made tuning the PID controllers much easier since more would be known about the system and more efficient methods could be used to determine the PID gains.

Glossary

IMU – Acronym for inertial measurement unit. This module is capable of sensing rotational and linear accelerations.

GPS – Acronym for Global Positioning System. This module receives data from GPS satellites and uses this data to calculate its position in relation to the surface of the planet using trilateration.

LiDAR – Acronym for Light Detection and Ranging. LiDAR modules send out pulses of light that reflect off surfaces in the surrounding environment and return to the module. The LiDAR module uses time-of-flight or triangulation to calculate how far away the obstacle is from the module.

PID – Acronym for proportional, integral, derivative. This term is generally associated with PID controllers which are equations that use measurements and some kind of control system to reach a certain set of parameters (i.e., reading a thermometer and using a heater to make a room reach a certain temperature).

Appendix

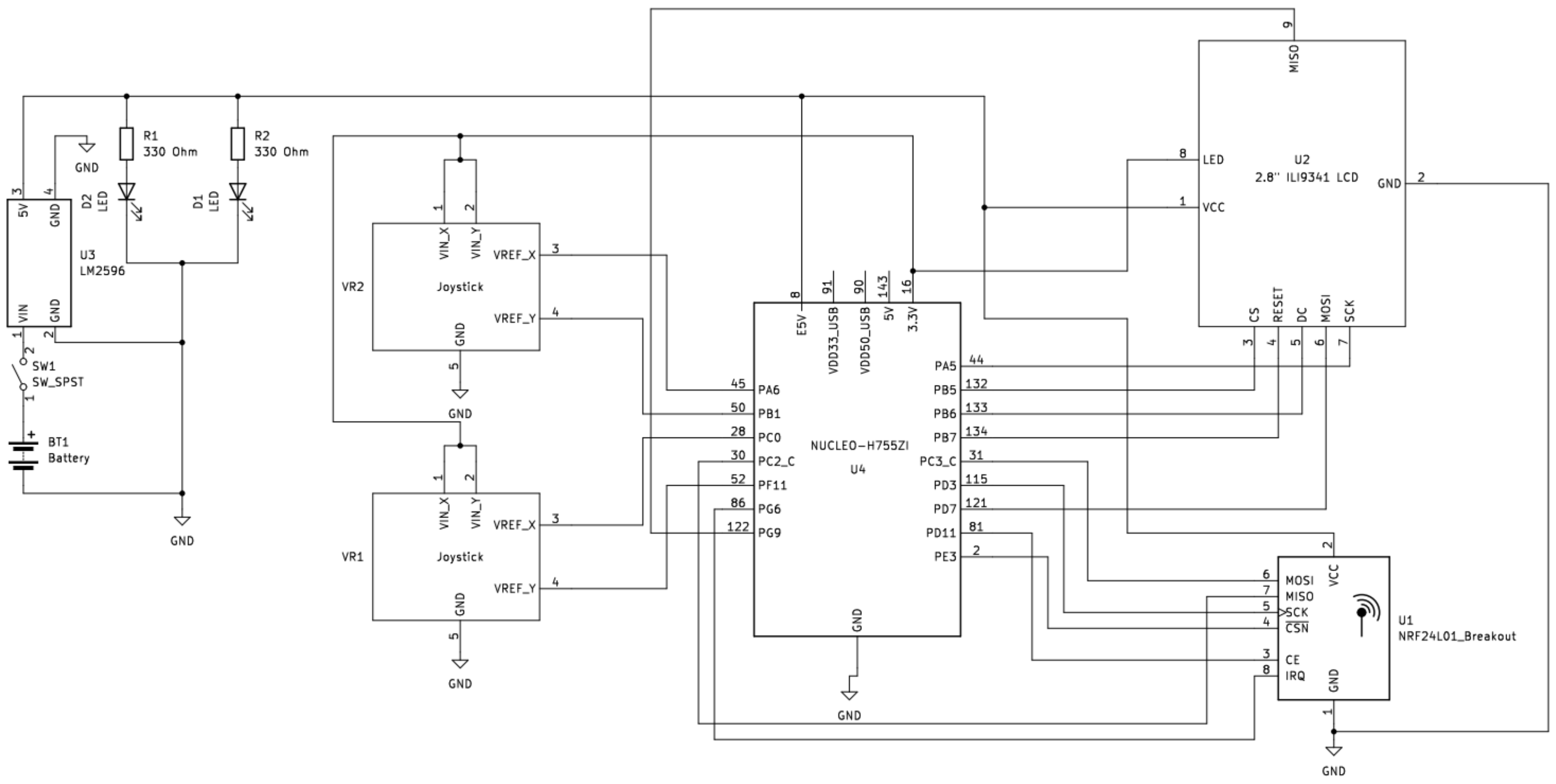


Figure 33: Controller Wiring Schematic

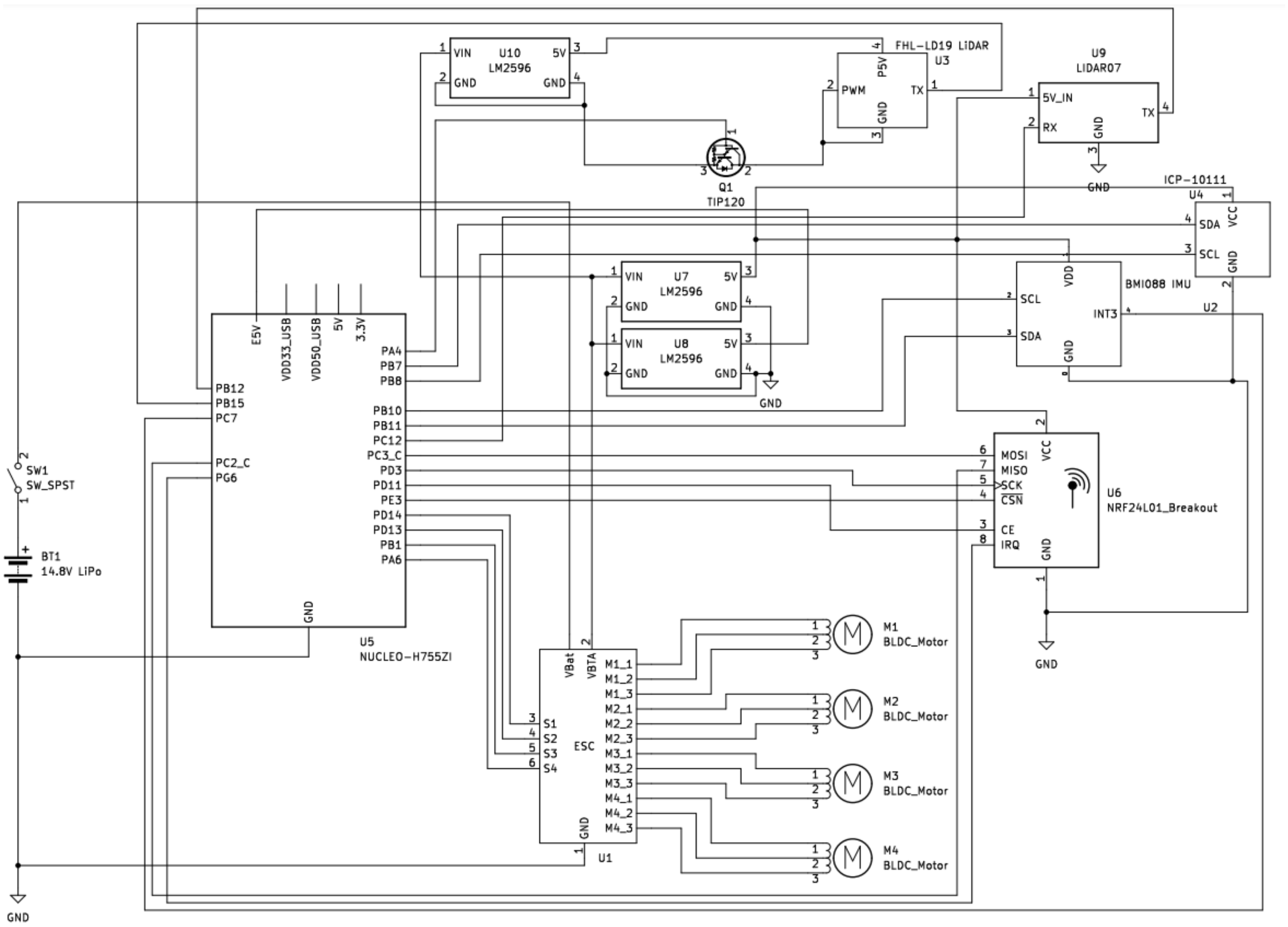
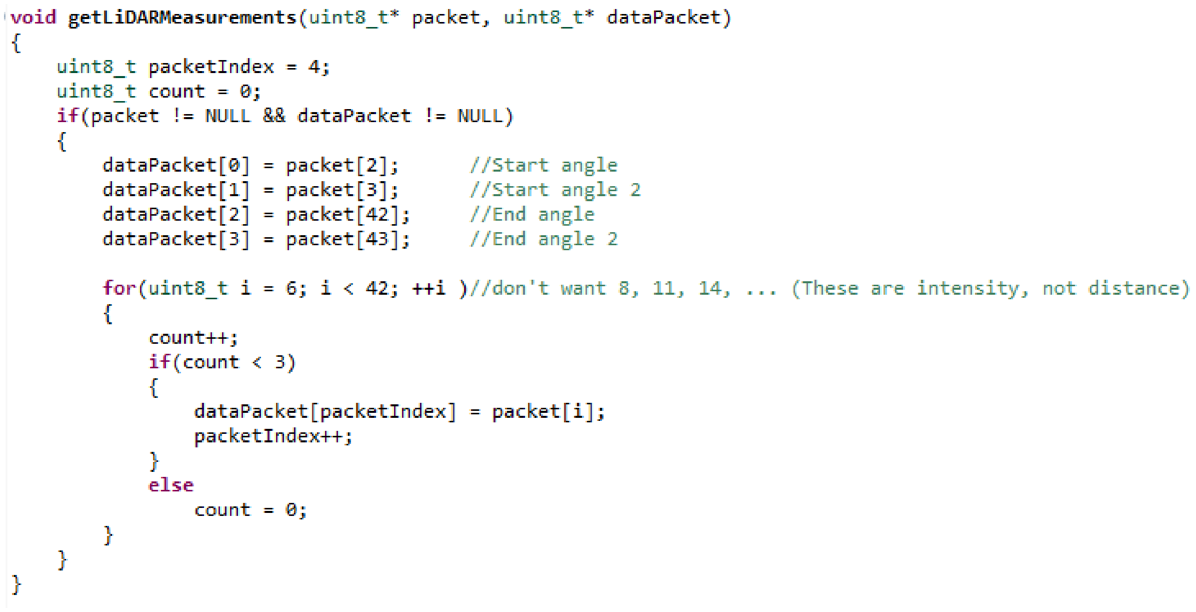
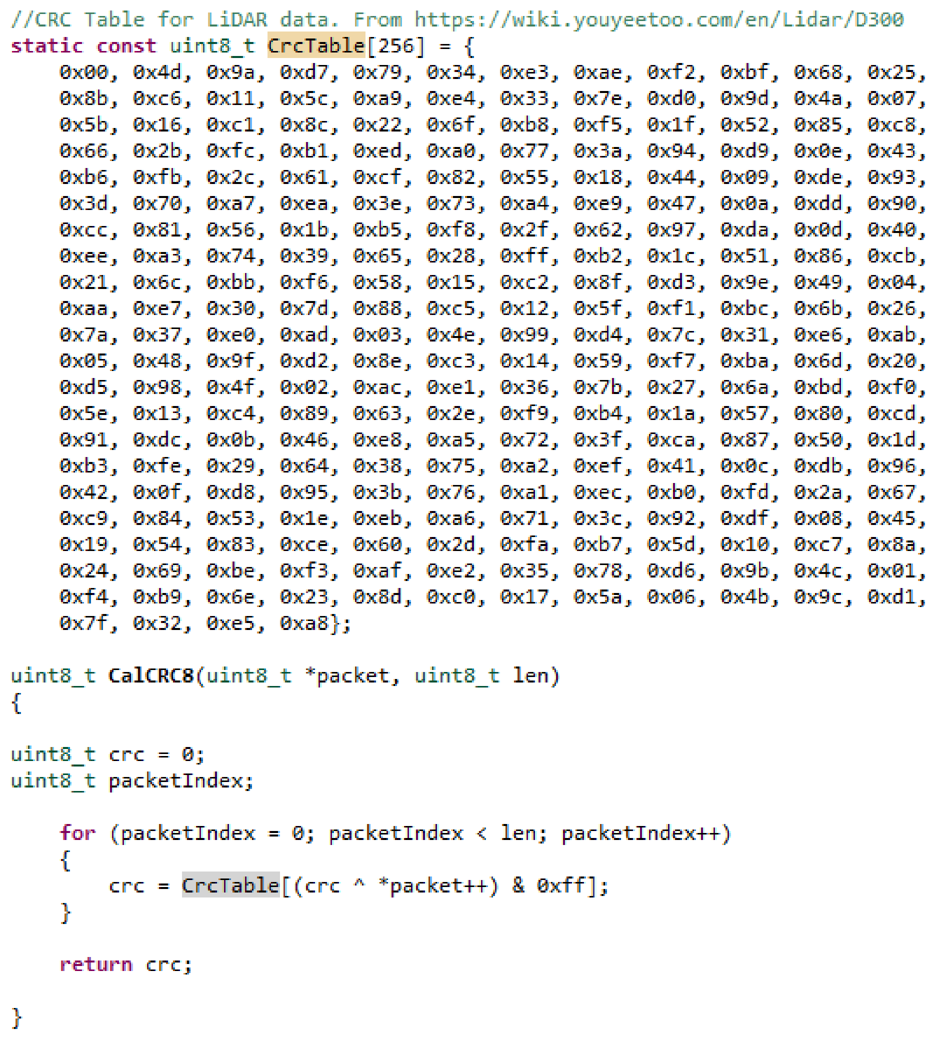


Figure 34: Drone Wiring Schematic



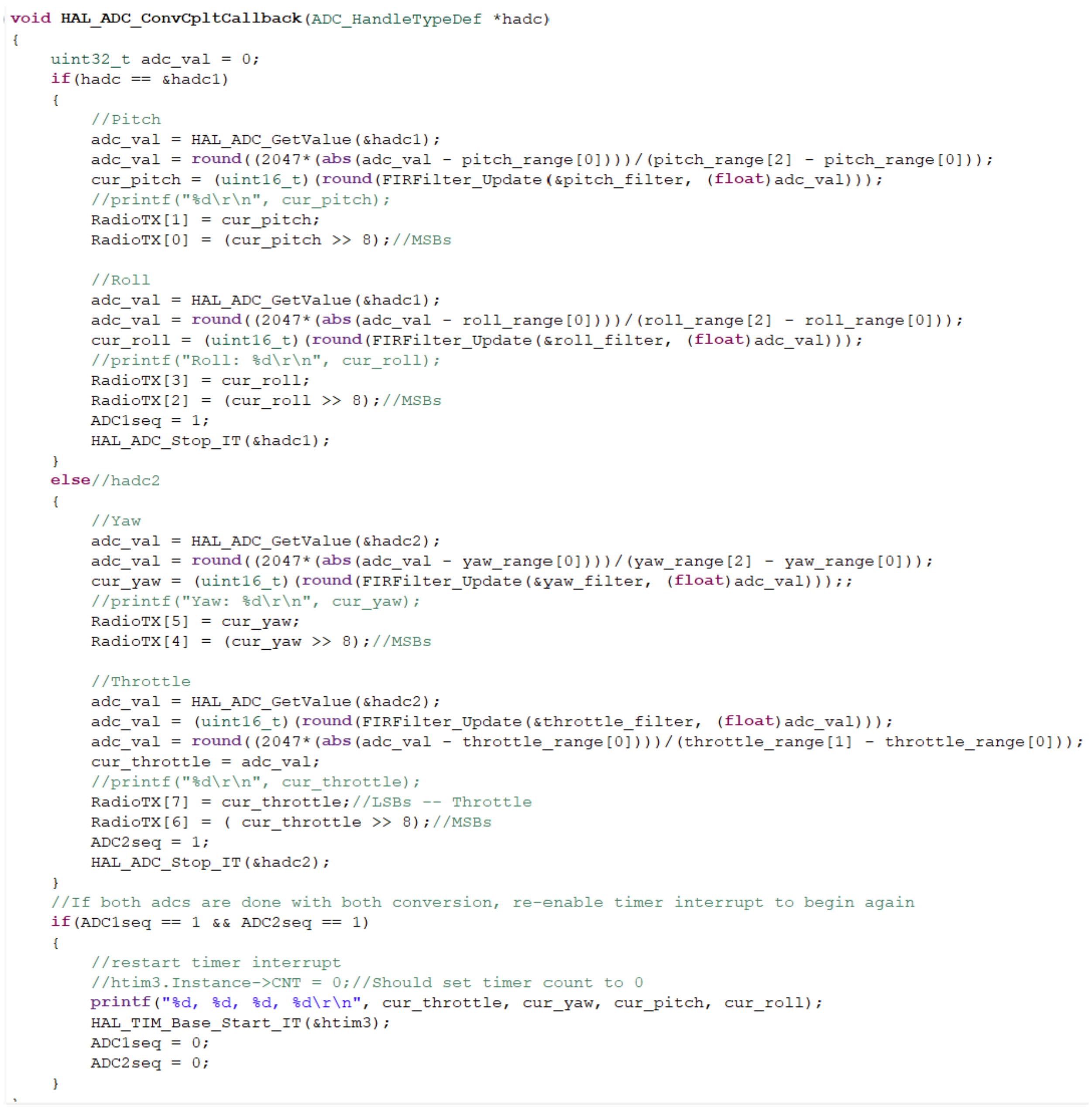
```c
void getLiDARMeasurements(uint8_t* packet, uint8_t* dataPacket)
{
    uint8_t packetIndex = 4;
    uint8_t count = 0;
    if(packet != NULL && dataPacket != NULL)
    {
        dataPacket[0] = packet[2];      //Start angle
        dataPacket[1] = packet[3];      //Start angle 2
        dataPacket[2] = packet[42];     //End angle
        dataPacket[3] = packet[43];     //End angle 2

        for(uint8_t i = 6; i < 42; ++i )//don't want 8, 11, 14, ... (These are intensity, not distance)
        {
            count++;
            if(count < 3)
            {
                dataPacket[packetIndex] = packet[i];
                packetIndex++;
            }
            else
                count = 0;
        }
    }
}
```

Figure 35: LiDAR Data Parsing Function



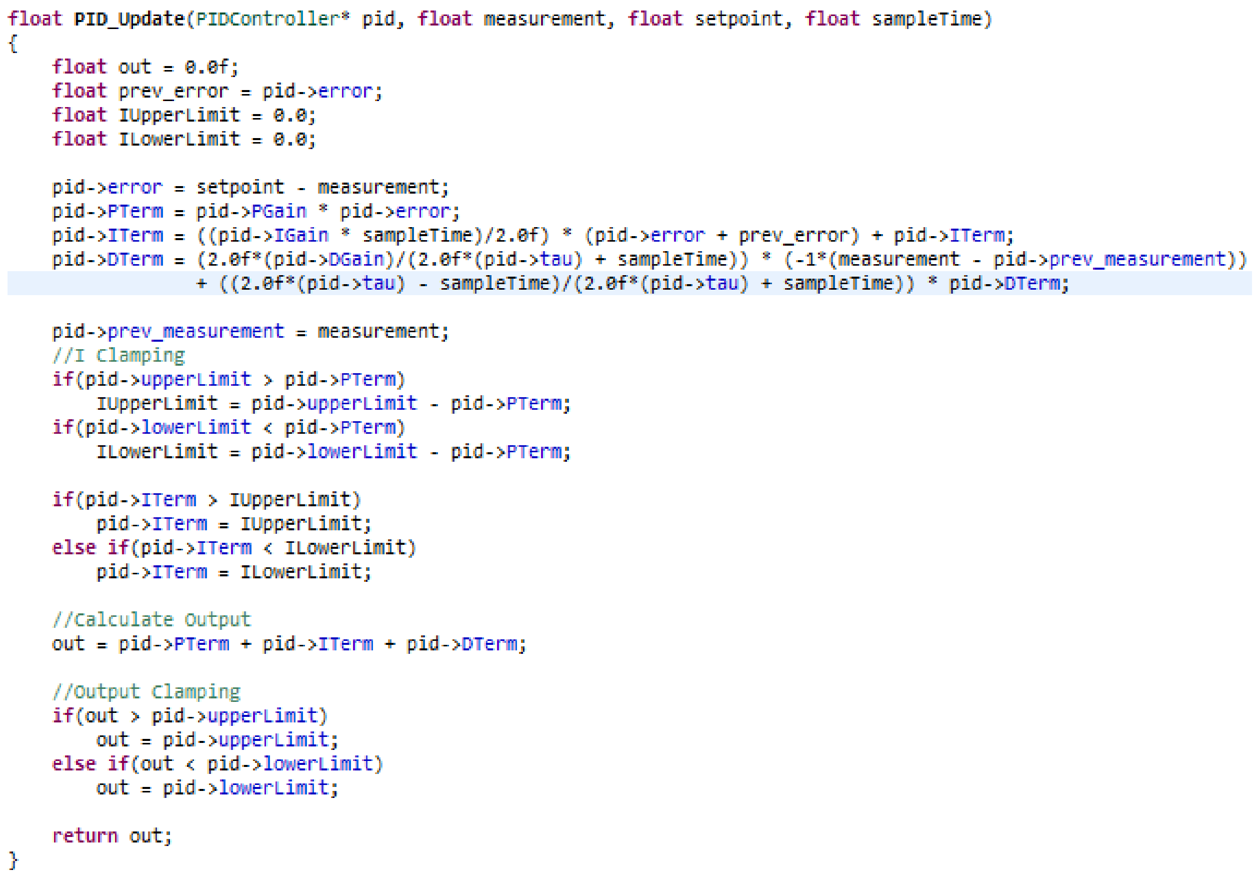
```
//CRC Table for LiDAR data. From https://wiki.youyeetoo.com/en/Lidar/D300
static const uint8_t CrcTable[256] = {
    0x00, 0x4d, 0x9a, 0xd7, 0x79, 0x34, 0xe3, 0xae, 0xf2, 0xbf, 0x68, 0x25,
    0x8b, 0xc6, 0x11, 0x5c, 0xa9, 0xe4, 0x33, 0x7e, 0xd0, 0x9d, 0x4a, 0x07,
    0x5b, 0x16, 0xc1, 0x8c, 0x22, 0x6f, 0xb8, 0xf5, 0x1f, 0x52, 0x85, 0xc8,
    0x66, 0x2b, 0xfc, 0xb1, 0xed, 0xa0, 0x77, 0x3a, 0x94, 0xd9, 0x0e, 0x43,
    0xb6, 0xfb, 0x2c, 0x61, 0xcf, 0x82, 0x55, 0x18, 0x44, 0x09, 0xde, 0x93,
    0x3d, 0x70, 0xa7, 0xea, 0x3e, 0x73, 0xa4, 0xe9, 0x47, 0x0a, 0xdd, 0x90,
    0xcc, 0x81, 0x56, 0x1b, 0xb5, 0xf8, 0x2f, 0x62, 0x97, 0xda, 0x0d, 0x40,
    0xee, 0xa3, 0x74, 0x39, 0x65, 0x28, 0xff, 0xb2, 0x1c, 0x51, 0x86, 0xcb,
    0x21, 0x6c, 0xbb, 0xf6, 0x58, 0x15, 0xc2, 0x8f, 0xd3, 0x9e, 0x49, 0x04,
    0xaa, 0xe7, 0x30, 0x7d, 0x88, 0xc5, 0x12, 0x5f, 0xf1, 0xbc, 0x6b, 0x26,
    0x7a, 0x37, 0xe0, 0xad, 0x03, 0x4e, 0x99, 0xd4, 0x7c, 0x31, 0xe6, 0xab,
    0x05, 0x48, 0x9f, 0xd2, 0x8e, 0xc3, 0x14, 0x59, 0xf7, 0xba, 0x6d, 0x20,
    0xd5, 0x98, 0x4f, 0x02, 0xac, 0xe1, 0x36, 0x7b, 0x27, 0x6a, 0xbd, 0xf0,
    0x5e, 0x13, 0xc4, 0x89, 0x63, 0x2e, 0xf9, 0xb4, 0x1a, 0x57, 0x80, 0xcd,
    0x91, 0xdc, 0x0b, 0x46, 0xe8, 0xa5, 0x72, 0x3f, 0xca, 0x87, 0x50, 0x1d,
    0xb3, 0xfe, 0x29, 0x64, 0x38, 0x75, 0xa2, 0xef, 0x41, 0x0c, 0xdb, 0x96,
    0x42, 0x0f, 0xd8, 0x95, 0x3b, 0x76, 0xa1, 0xec, 0xb0, 0xfd, 0x2a, 0x67,
    0xc9, 0x84, 0x53, 0x1e, 0xeb, 0xa6, 0x71, 0x3c, 0x92, 0xdf, 0x08, 0x45,
    0x19, 0x54, 0x83, 0xce, 0x60, 0x2d, 0xfa, 0xb7, 0x5d, 0x10, 0xc7, 0x8a,
    0x24, 0x69, 0xbe, 0xf3, 0xaf, 0xe2, 0x35, 0x78, 0xd6, 0x9b, 0x4c, 0x01,
    0xf4, 0xb9, 0x6e, 0x23, 0x8d, 0xc0, 0x17, 0x5a, 0x06, 0x4b, 0x9c, 0xd1,
    0x7f, 0x32, 0xe5, 0xa8};

uint8_t CalCRC8(uint8_t *packet, uint8_t len)
{

uint8_t crc = 0;
uint8_t packetIndex;

    for (packetIndex = 0; packetIndex < len; packetIndex++)
    {
        crc = CrcTable[(crc ^ *packet++) & 0xff];
    }

    return crc;

}
```

Figure 36: LiDAR CRC Function



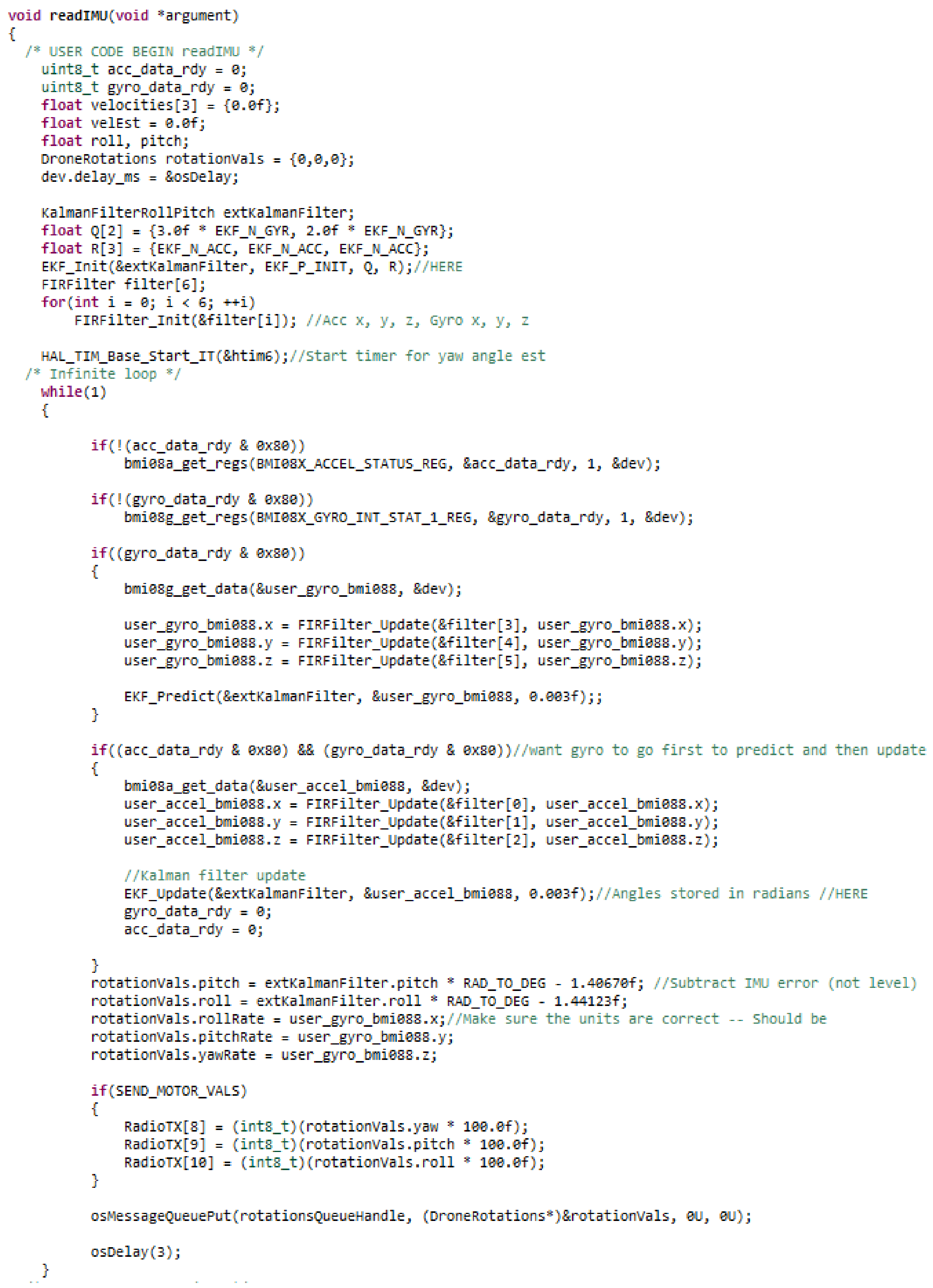
```c
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
    uint32_t adc_val = 0;
    if(hadc == &hadc1)
    {
        //Pitch
        adc_val = HAL_ADC_GetValue(&hadc1);
        adc_val = round((2047*(abs(adc_val - pitch_range[0])))/(pitch_range[2] - pitch_range[0]));
        cur_pitch = (uint16_t)(round(FIRFilter_Update(&pitch_filter, (float)adc_val)));
        //printf("%d\r\n", cur_pitch);
        RadioTX[1] = cur_pitch;
        RadioTX[0] = (cur_pitch >> 8);//MSBs

        //Roll
        adc_val = HAL_ADC_GetValue(&hadc1);
        adc_val = round((2047*(abs(adc_val - roll_range[0])))/(roll_range[2] - roll_range[0]));
        cur_roll = (uint16_t)(round(FIRFilter_Update(&roll_filter, (float)adc_val)));
        //printf("Roll: %d\r\n", cur_roll);
        RadioTX[3] = cur_roll;
        RadioTX[2] = (cur_roll >> 8);//MSBs
        ADC1seq = 1;
        HAL_ADC_Stop_IT(&hadc1);
    }
    else//hadc2
    {
        //Yaw
        adc_val = HAL_ADC_GetValue(&hadc2);
        adc_val = round((2047*(abs(adc_val - yaw_range[0])))/(yaw_range[2] - yaw_range[0]));
        cur_yaw = (uint16_t)(round(FIRFilter_Update(&yaw_filter, (float)adc_val)));;
        //printf("Yaw: %d\r\n", cur_yaw);
        RadioTX[5] = cur_yaw;
        RadioTX[4] = (cur_yaw >> 8);//MSBs

        //Throttle
        adc_val = HAL_ADC_GetValue(&hadc2);
        adc_val = (uint16_t)(round(FIRFilter_Update(&throttle_filter, (float)adc_val)));
        adc_val = round((2047*(abs(adc_val - throttle_range[0])))/(throttle_range[1] - throttle_range[0]));
        cur_throttle = adc_val;
        //printf("%d\r\n", cur_throttle);
        RadioTX[7] = cur_throttle;//LSBs -- Throttle
        RadioTX[6] = ( cur_throttle >> 8);//MSBs
        ADC2seq = 1;
        HAL_ADC_Stop_IT(&hadc2);
    }
    //If both adcs are done with both conversion, re-enable timer interrupt to begin again
    if(ADC1seq == 1 && ADC2seq == 1)
    {
        //restart timer interrupt
        //htim3.Instance->CNT = 0;//Should set timer count to 0
        printf("%d, %d, %d, %d\r\n", cur_throttle, cur_yaw, cur_pitch, cur_roll);
        HAL_TIM_Base_Start_IT(&htim3);
        ADC1seq = 0;
        ADC2seq = 0;
    }
}
```

Figure 37: ADC Conversion Complete ISR



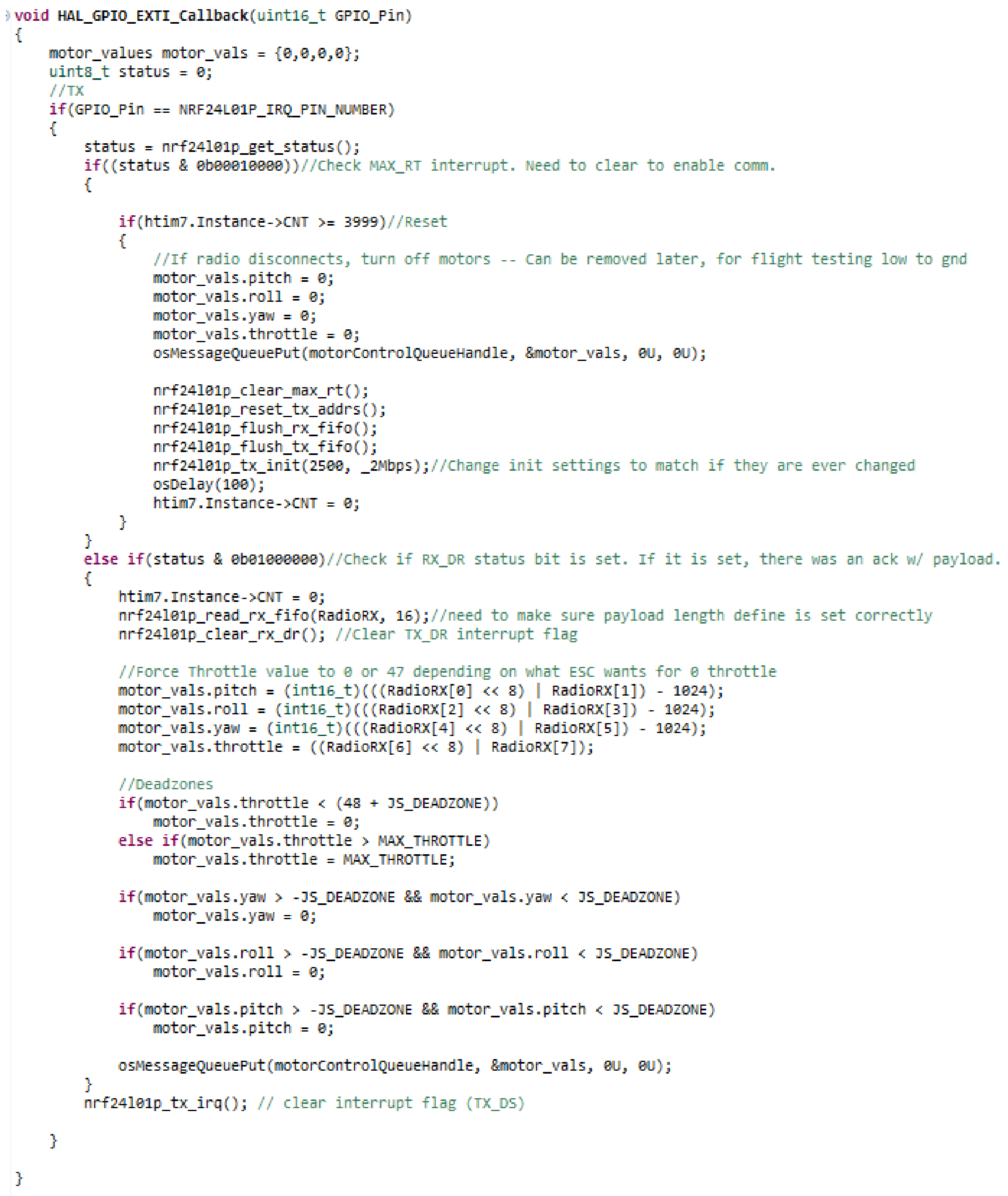
```c
float PID_Update(PIDController* pid, float measurement, float setpoint, float sampleTime)
{
    float out = 0.0f;
    float prev_error = pid->error;
    float IUpperLimit = 0.0;
    float ILowerLimit = 0.0;

    pid->error = setpoint - measurement;
    pid->PTerm = pid->PGain * pid->error;
    pid->ITerm = ((pid->IGain * sampleTime)/2.0f) * (pid->error + prev_error) + pid->ITerm;
    pid->DTerm = (2.0f*(pid->DGain)/(2.0f*(pid->tau) + sampleTime)) * (-1*(measurement - pid->prev_measurement))
                 + ((2.0f*(pid->tau) - sampleTime)/(2.0f*(pid->tau) + sampleTime)) * pid->DTerm;

    pid->prev_measurement = measurement;
    //I Clamping
    if(pid->upperLimit > pid->PTerm)
        IUpperLimit = pid->upperLimit - pid->PTerm;
    if(pid->lowerLimit < pid->PTerm)
        ILowerLimit = pid->lowerLimit - pid->PTerm;

    if(pid->ITerm > IUpperLimit)
        pid->ITerm = IUpperLimit;
    else if(pid->ITerm < ILowerLimit)
        pid->ITerm = ILowerLimit;

    //Calculate Output
    out = pid->PTerm + pid->ITerm + pid->DTerm;

    //Output Clamping
    if(out > pid->upperLimit)
        out = pid->upperLimit;
    else if(out < pid->lowerLimit)
        out = pid->lowerLimit;

    return out;
}
```

Figure 38: PID Control Equations. Adapted from [5]



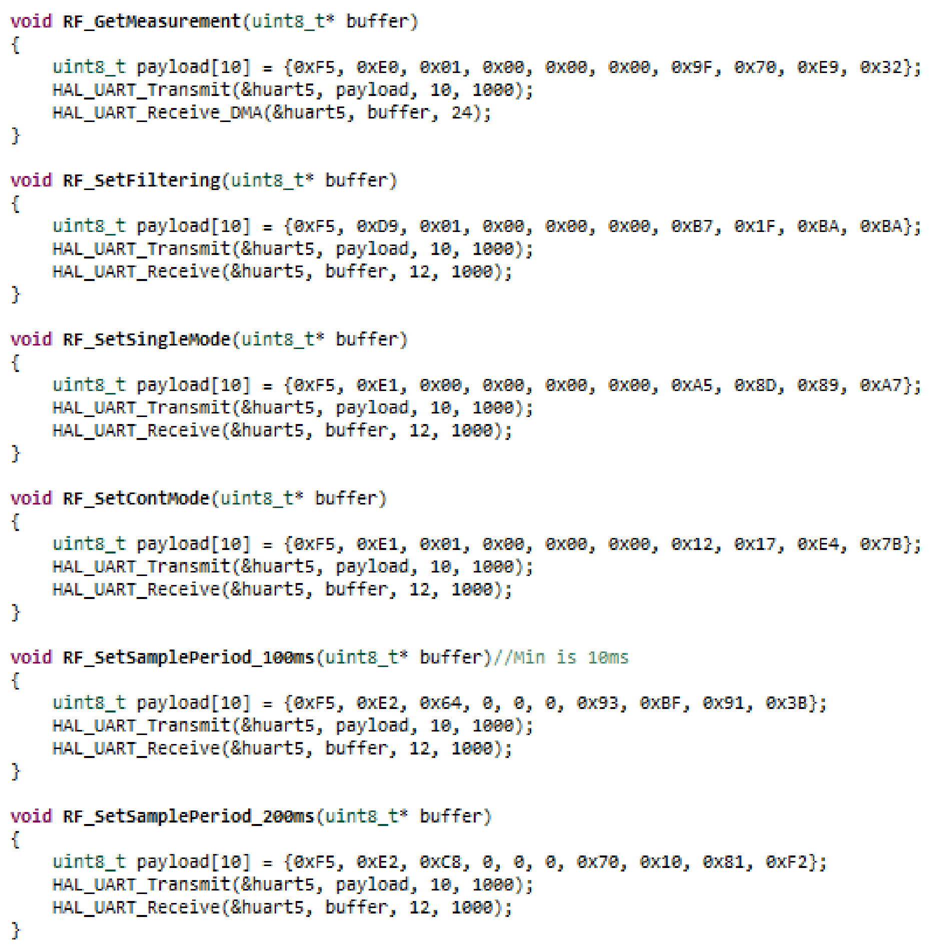
```c
void readIMU(void *argument)
{
  /* USER CODE BEGIN readIMU */
    uint8_t acc_data_rdy = 0;
    uint8_t gyro_data_rdy = 0;
    float velocities[3] = {0.0f};
    float velEst = 0.0f;
    float roll, pitch;
    DroneRotations rotationVals = {0,0,0};
    dev.delay_ms = &osDelay;

    KalmanFilterRollPitch extKalmanFilter;
    float Q[2] = {3.0f * EKF_N_GYR, 2.0f * EKF_N_GYR};
    float R[3] = {EKF_N_ACC, EKF_N_ACC, EKF_N_ACC};
    EKF_Init(&extKalmanFilter, EKF_P_INIT, Q, R);//HERE
    FIRFilter filter[6];
    for(int i = 0; i < 6; ++i)
        FIRFilter_Init(&filter[i]); //Acc x, y, z, Gyro x, y, z

    HAL_TIM_Base_Start_IT(&htim6);//Start timer for yaw angle est
  /* Infinite loop */
    while(1)
    {

          if(!(acc_data_rdy & 0x80))
              bmi08a_get_regs(BMI08X_ACCEL_STATUS_REG, &acc_data_rdy, 1, &dev);

          if(!(gyro_data_rdy & 0x80))
              bmi08g_get_regs(BMI08X_GYRO_INT_STAT_1_REG, &gyro_data_rdy, 1, &dev);

          if((gyro_data_rdy & 0x80))
          {
              bmi08g_get_data(&user_gyro_bmi088, &dev);

              user_gyro_bmi088.x = FIRFilter_Update(&filter[3], user_gyro_bmi088.x);
              user_gyro_bmi088.y = FIRFilter_Update(&filter[4], user_gyro_bmi088.y);
              user_gyro_bmi088.z = FIRFilter_Update(&filter[5], user_gyro_bmi088.z);

              EKF_Predict(&extKalmanFilter, &user_gyro_bmi088, 0.003f);;
          }

          if((acc_data_rdy & 0x80) && (gyro_data_rdy & 0x80))//want gyro to go first to predict and then update
          {
              bmi08a_get_data(&user_accel_bmi088, &dev);
              user_accel_bmi088.x = FIRFilter_Update(&filter[0], user_accel_bmi088.x);
              user_accel_bmi088.y = FIRFilter_Update(&filter[1], user_accel_bmi088.y);
              user_accel_bmi088.z = FIRFilter_Update(&filter[2], user_accel_bmi088.z);

              //Kalman filter update
              EKF_Update(&extKalmanFilter, &user_accel_bmi088, 0.003f);//Angles stored in radians //HERE
              gyro_data_rdy = 0;
              acc_data_rdy = 0;

          }
          rotationVals.pitch = extKalmanFilter.pitch * RAD_TO_DEG - 1.40670f; //Subtract IMU error (not level)
          rotationVals.roll = extKalmanFilter.roll * RAD_TO_DEG - 1.44123f;
          rotationVals.rollRate = user_gyro_bmi088.x;//Make sure the units are correct -- Should be
          rotationVals.pitchRate = user_gyro_bmi088.y;
          rotationVals.yawRate = user_gyro_bmi088.z;

          if(SEND_MOTOR_VALS)
          {
              RadioTX[8] = (int8_t)(rotationVals.yaw * 100.0f);
              RadioTX[9] = (int8_t)(rotationVals.pitch * 100.0f);
              RadioTX[10] = (int8_t)(rotationVals.roll * 100.0f);
          }

          osMessageQueuePut(rotationsQueueHandle, (DroneRotations*)&rotationVals, 0U, 0U);

          osDelay(3);
    }
```

Figure 39: IMU Task



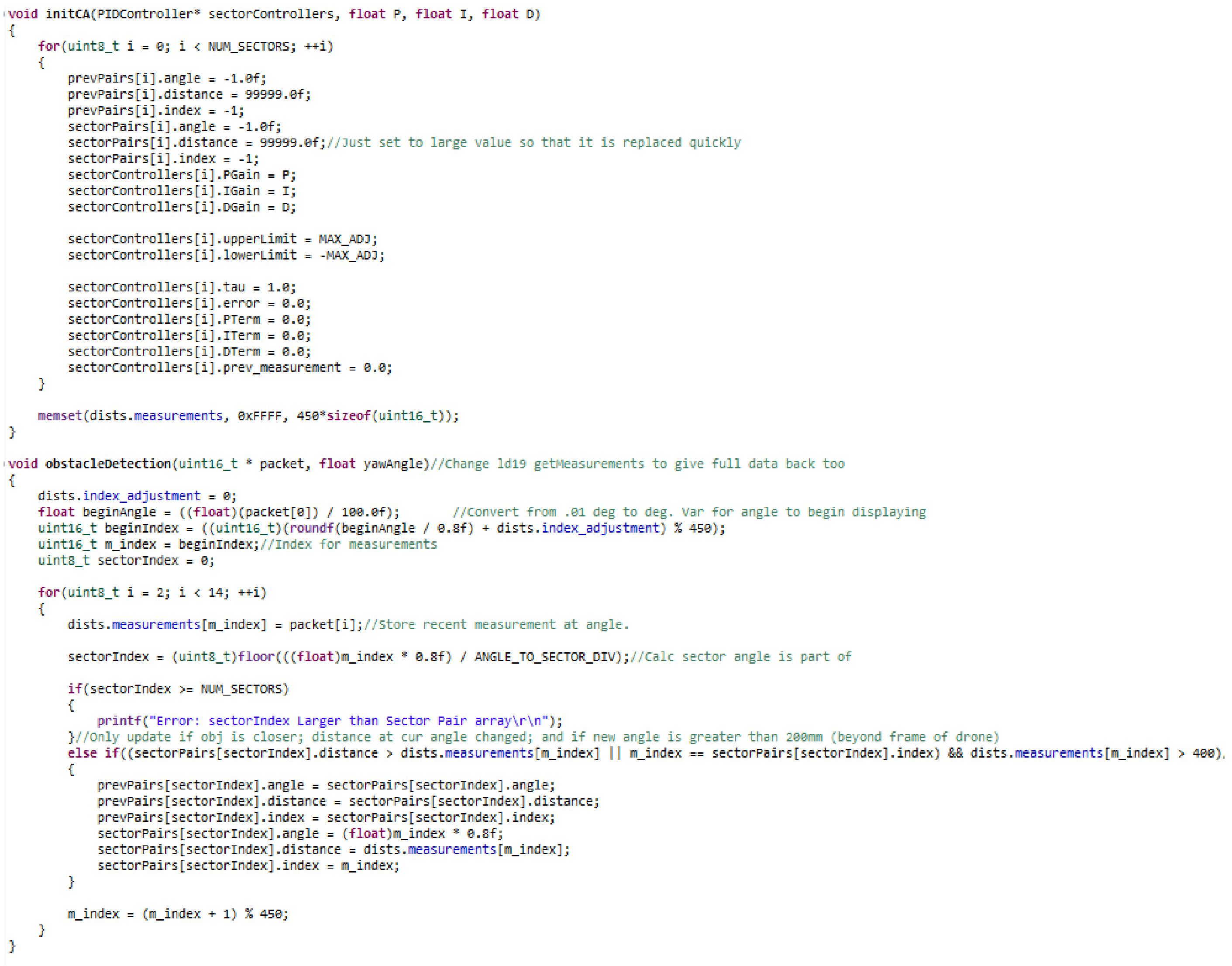
```c
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    motor_values motor_vals = {0,0,0,0};
    uint8_t status = 0;
    //TX
    if(GPIO_Pin == NRF24L01P_IRQ_PIN_NUMBER)
    {
        status = nrf24l01p_get_status();
        if((status & 0b00010000))//Check MAX_RT interrupt. Need to clear to enable comm.
        {

            if(htim7.Instance->CNT >= 3999)//Reset
            {
                //If radio disconnects, turn off motors -- Can be removed later, for flight testing low to gnd
                motor_vals.pitch = 0;
                motor_vals.roll = 0;
                motor_vals.yaw = 0;
                motor_vals.throttle = 0;
                osMessageQueuePut(motorControlQueueHandle, &motor_vals, 0U, 0U);

                nrf24l01p_clear_max_rt();
                nrf24l01p_reset_tx_addrs();
                nrf24l01p_flush_rx_fifo();
                nrf24l01p_flush_tx_fifo();
                nrf24l01p_tx_init(2500, _2Mbps);//Change init settings to match if they are ever changed
                osDelay(100);
                htim7.Instance->CNT = 0;
            }
        }
        else if(status & 0b01000000)//Check if RX_DR status bit is set. If it is set, there was an ack w/ payload.
        {
            htim7.Instance->CNT = 0;
            nrf24l01p_read_rx_fifo(RadioRX, 16);//need to make sure payload length define is set correctly
            nrf24l01p_clear_rx_dr(); //Clear TX_DR interrupt flag

            //Force Throttle value to 0 or 47 depending on what ESC wants for 0 throttle
            motor_vals.pitch = (int16_t)(((RadioRX[0] << 8) | RadioRX[1]) - 1024);
            motor_vals.roll = (int16_t)(((RadioRX[2] << 8) | RadioRX[3]) - 1024);
            motor_vals.yaw = (int16_t)(((RadioRX[4] << 8) | RadioRX[5]) - 1024);
            motor_vals.throttle = ((RadioRX[6] << 8) | RadioRX[7]);

            //Deadzones
            if(motor_vals.throttle < (48 + JS_DEADZONE))
                motor_vals.throttle = 0;
            else if(motor_vals.throttle > MAX_THROTTLE)
                motor_vals.throttle = MAX_THROTTLE;

            if(motor_vals.yaw > -JS_DEADZONE && motor_vals.yaw < JS_DEADZONE)
                motor_vals.yaw = 0;

            if(motor_vals.roll > -JS_DEADZONE && motor_vals.roll < JS_DEADZONE)
                motor_vals.roll = 0;

            if(motor_vals.pitch > -JS_DEADZONE && motor_vals.pitch < JS_DEADZONE)
                motor_vals.pitch = 0;

            osMessageQueuePut(motorControlQueueHandle, &motor_vals, 0U, 0U);
        }
        nrf24l01p_tx_irq(); // clear interrupt flag (TX_DS)

    }

}
```

Figure 40: Drone Radio RX ISR



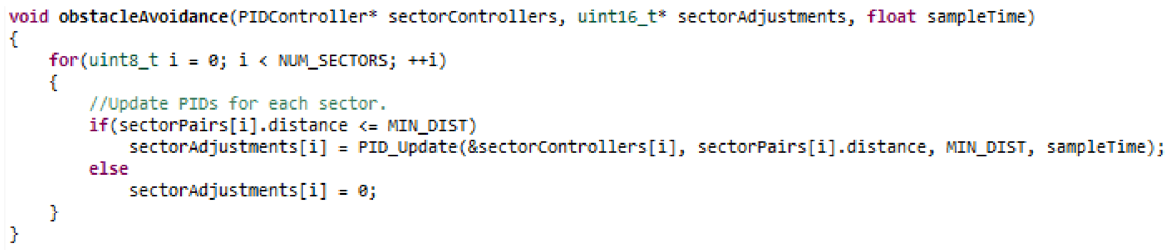
```
void RF_GetMeasurement(uint8_t* buffer)
{
    uint8_t payload[10] = {0xF5, 0xE0, 0x01, 0x00, 0x00, 0x00, 0x9F, 0x70, 0xE9, 0x32};
    HAL_UART_Transmit(&huart5, payload, 10, 1000);
    HAL_UART_Receive_DMA(&huart5, buffer, 24);
}

void RF_SetFiltering(uint8_t* buffer)
{
    uint8_t payload[10] = {0xF5, 0xD9, 0x01, 0x00, 0x00, 0x00, 0xB7, 0x1F, 0xBA, 0xBA};
    HAL_UART_Transmit(&huart5, payload, 10, 1000);
    HAL_UART_Receive(&huart5, buffer, 12, 1000);
}

void RF_SetSingleMode(uint8_t* buffer)
{
    uint8_t payload[10] = {0xF5, 0xE1, 0x00, 0x00, 0x00, 0x00, 0xA5, 0x8D, 0x89, 0xA7};
    HAL_UART_Transmit(&huart5, payload, 10, 1000);
    HAL_UART_Receive(&huart5, buffer, 12, 1000);
}

void RF_SetContMode(uint8_t* buffer)
{
    uint8_t payload[10] = {0xF5, 0xE1, 0x01, 0x00, 0x00, 0x00, 0x12, 0x17, 0xE4, 0x7B};
    HAL_UART_Transmit(&huart5, payload, 10, 1000);
    HAL_UART_Receive(&huart5, buffer, 12, 1000);
}

void RF_SetSamplePeriod_100ms(uint8_t* buffer)//Min is 10ms
{
    uint8_t payload[10] = {0xF5, 0xE2, 0x64, 0, 0, 0, 0x93, 0xBF, 0x91, 0x3B};
    HAL_UART_Transmit(&huart5, payload, 10, 1000);
    HAL_UART_Receive(&huart5, buffer, 12, 1000);
}

void RF_SetSamplePeriod_200ms(uint8_t* buffer)
{
    uint8_t payload[10] = {0xF5, 0xE2, 0xC8, 0, 0, 0, 0x70, 0x10, 0x81, 0xF2};
    HAL_UART_Transmit(&huart5, payload, 10, 1000);
    HAL_UART_Receive(&huart5, buffer, 12, 1000);
}
```

Figure 41: Rangefinder Functions



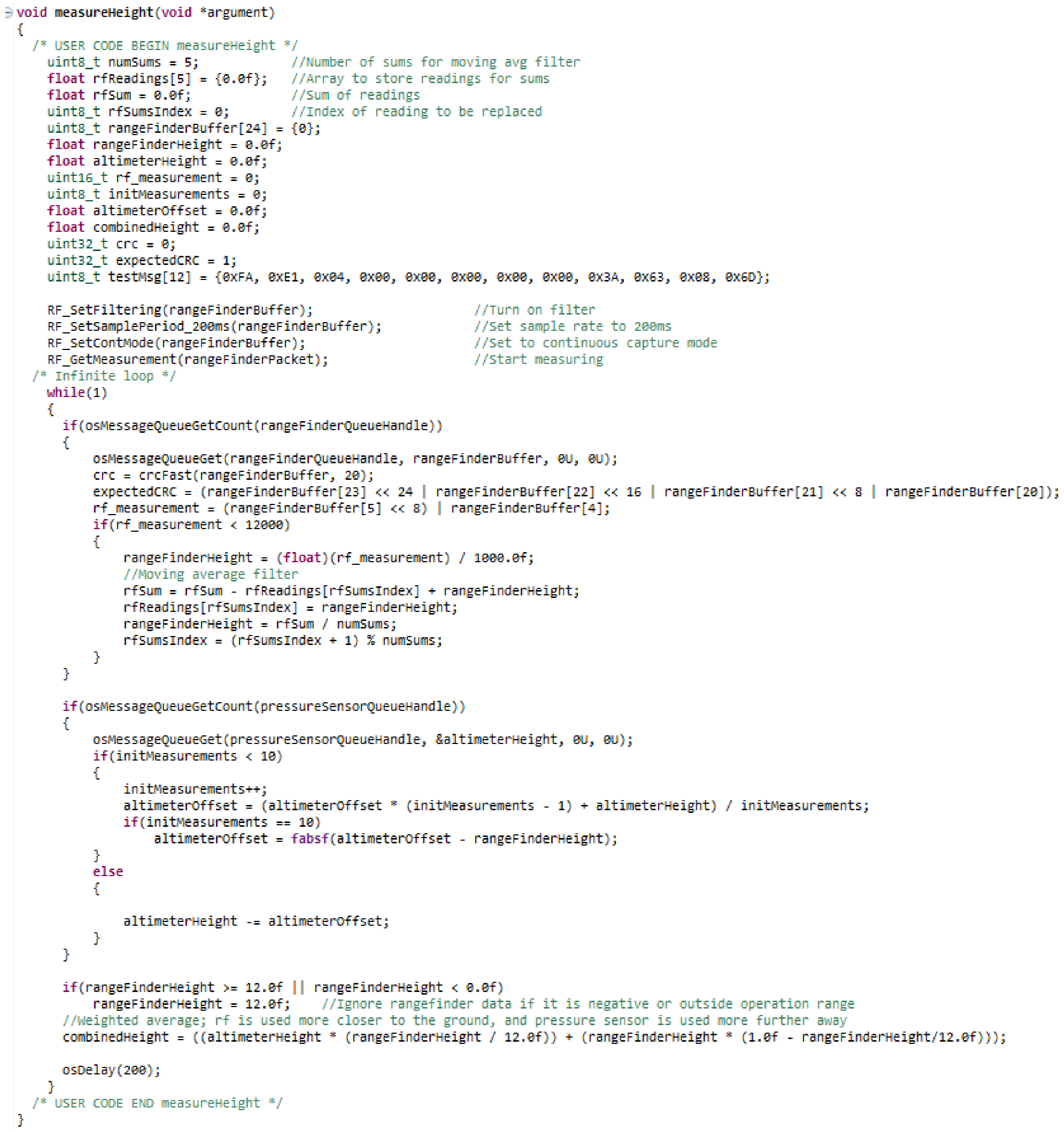
```c
void initCA(PIDController* sectorControllers, float P, float I, float D)
{
    for(uint8_t i = 0; i < NUM_SECTORS; ++i)
    {
        prevPairs[i].angle = -1.0f;
        prevPairs[i].distance = 99999.0f;
        prevPairs[i].index = -1;
        sectorPairs[i].angle = -1.0f;
        sectorPairs[i].distance = 99999.0f;//Just set to large value so that it is replaced quickly
        sectorPairs[i].index = -1;
        sectorControllers[i].PGain = P;
        sectorControllers[i].IGain = I;
        sectorControllers[i].DGain = D;

        sectorControllers[i].upperLimit = MAX_ADJ;
        sectorControllers[i].lowerLimit = -MAX_ADJ;

        sectorControllers[i].tau = 1.0;
        sectorControllers[i].error = 0.0;
        sectorControllers[i].PTerm = 0.0;
        sectorControllers[i].ITerm = 0.0;
        sectorControllers[i].DTerm = 0.0;
        sectorControllers[i].prev_measurement = 0.0;
    }

    memset(dists.measurements, 0xFFFF, 450*sizeof(uint16_t));
}

void obstacleDetection(uint16_t * packet, float yawAngle)//Change ld19 getMeasurements to give full data back too
{
    dists.index_adjustment = 0;
    float beginAngle = ((float)(packet[0]) / 100.0f);        //Convert from .01 deg to deg. Var for angle to begin displaying
    uint16_t beginIndex = ((uint16_t)(roundf(beginAngle / 0.8f) + dists.index_adjustment) % 450);
    uint16_t m_index = beginIndex;//Index for measurements
    uint8_t sectorIndex = 0;

    for(uint8_t i = 2; i < 14; ++i)
    {
        dists.measurements[m_index] = packet[i];//Store recent measurement at angle.

        sectorIndex = (uint8_t)floor(((float)m_index * 0.8f) / ANGLE_TO_SECTOR_DIV);//Calc sector angle is part of

        if(sectorIndex >= NUM_SECTORS)
        {
            printf("Error: sectorIndex Larger than Sector Pair array\r\n");
        }//Only update if obj is closer; distance at cur angle changed; and if new angle is greater than 200mm (beyond frame of drone)
        else if((sectorPairs[sectorIndex].distance > dists.measurements[m_index] || m_index == sectorPairs[sectorIndex].index) && dists.measurements[m_index] > 400),
        {
            prevPairs[sectorIndex].angle = sectorPairs[sectorIndex].angle;
            prevPairs[sectorIndex].distance = sectorPairs[sectorIndex].distance;
            prevPairs[sectorIndex].index = sectorPairs[sectorIndex].index;
            sectorPairs[sectorIndex].angle = (float)m_index * 0.8f;
            sectorPairs[sectorIndex].distance = dists.measurements[m_index];
            sectorPairs[sectorIndex].index = m_index;
        }

        m_index = (m_index + 1) % 450;
    }
}
```

Figure 42: Init CA and Obstacle Detection Functions



```c
void obstacleAvoidance(PIDController* sectorControllers, uint16_t* sectorAdjustments, float sampleTime)
{
    for(uint8_t i = 0; i < NUM_SECTORS; ++i)
    {
        //Update PIDs for each sector.
        if(sectorPairs[i].distance <= MIN_DIST)
            sectorAdjustments[i] = PID_Update(&sectorControllers[i], sectorPairs[i].distance, MIN_DIST, sampleTime);
        else
            sectorAdjustments[i] = 0;
    }
}
```

Figure 43: Obstacle Avoidance Function



```c
void measureHeight(void *argument)
{
  /* USER CODE BEGIN measureHeight */
    uint8_t numSums = 5;            //Number of sums for moving avg filter
    float rfReadings[5] = {0.0f};   //Array to store readings for sums
    float rfSum = 0.0f;             //Sum of readings
    uint8_t rfSumsIndex = 0;        //Index of reading to be replaced
    uint8_t rangeFinderBuffer[24] = {0};
    float rangeFinderHeight = 0.0f;
    float altimeterHeight = 0.0f;
    uint16_t rf_measurement = 0;
    uint8_t initMeasurements = 0;
    float altimeterOffset = 0.0f;
    float combinedHeight = 0.0f;
    uint32_t crc = 0;
    uint32_t expectedCRC = 1;
    uint8_t testMsg[12] = {0xFA, 0xE1, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x3A, 0x63, 0x08, 0x6D};

    RF_SetFiltering(rangeFinderBuffer);                     //Turn on filter
    RF_SetSamplePeriod_200ms(rangeFinderBuffer);            //Set sample rate to 200ms
    RF_SetContMode(rangeFinderBuffer);                      //Set to continuous capture mode
    RF_GetMeasurement(rangeFinderPacket);                   //Start measuring
  /* Infinite loop */
    while(1)
    {
      if(osMessageQueueGetCount(rangeFinderQueueHandle))
      {
          osMessageQueueGet(rangeFinderQueueHandle, rangeFinderBuffer, 0U, 0U);
          crc = crcFast(rangeFinderBuffer, 20);
          expectedCRC = (rangeFinderBuffer[23] << 24 | rangeFinderBuffer[22] << 16 | rangeFinderBuffer[21] << 8 | rangeFinderBuffer[20]);
          rf_measurement = (rangeFinderBuffer[5] << 8) | rangeFinderBuffer[4];
          if(rf_measurement < 12000)
          {
              rangeFinderHeight = (float)(rf_measurement) / 1000.0f;
              //Moving average filter
              rfSum = rfSum - rfReadings[rfSumsIndex] + rangeFinderHeight;
              rfReadings[rfSumsIndex] = rangeFinderHeight;
              rangeFinderHeight = rfSum / numSums;
              rfSumsIndex = (rfSumsIndex + 1) % numSums;
          }
      }

      if(osMessageQueueGetCount(pressureSensorQueueHandle))
      {
          osMessageQueueGet(pressureSensorQueueHandle, &altimeterHeight, 0U, 0U);
          if(initMeasurements < 10)
          {
              initMeasurements++;
              altimeterOffset = (altimeterOffset * (initMeasurements - 1) + altimeterHeight) / initMeasurements;
              if(initMeasurements == 10)
                  altimeterOffset = fabsf(altimeterOffset - rangeFinderHeight);
          }
          else
          {

              altimeterHeight -= altimeterOffset;
          }
      }

      if(rangeFinderHeight >= 12.0f || rangeFinderHeight < 0.0f)
          rangeFinderHeight = 12.0f;    //Ignore rangefinder data if it is negative or outside operation range
      //Weighted average; rf is used more closer to the ground, and pressure sensor is used more further away
      combinedHeight = ((altimeterHeight * (rangeFinderHeight / 12.0f)) + (rangeFinderHeight * (1.0f - rangeFinderHeight/12.0f)));

      osDelay(200);
    }
  /* USER CODE END measureHeight */
}
```

Figure 44: Measure Height Task Code

References

[1] “DJI Air 2S - all in one,” DJI, https://www.dji.com/de/air-2s (accessed May 26, 2023).

[2] “Exo X7 Ranger Plus,” EXO Drones, https://exodrones.com/products/exo-x7-ranger-plus (accessed May 26, 2023).

[3] “FHL-LD19 12meter lidar - 30k lux,” youyeetoo wiki, https://wiki.youyeetoo.com/en/Lidar/D300 (accessed Dec. 11, 2023).

[4] J. Strickland, “What is a gimbal -- and what does it have to do with NASA?,” HowStuffWorks, https://science.howstuffworks.com/gimbal.htm (accessed Dec. 11, 2023).

[5] Phil's Lab. *PID Controller Implementation in Software - Phil's Lab #6*. (May 22, 2020). Accessed: Dec. 12, 2023. [Online Video]. Available: https://www.youtube.com/watch?v=zOByx3Izf5U

[6] Phil's Lab. *Extended Kalman Filter Software Implementation – Sensor Fusion #4 - Phil's Lab #3*. (Aug 22, 2022). Accessed: Dec. 12, 2023. [Online Video]. Available: https://www.youtube.com/watch?v=7HVPjkWOrLE

[7] Phil's Lab. *Extended Kalman Filter Software Implementation - Sensor Fusion #4 - Phil's Lab #73* (Aug. 22, 2022). Accessed: Feb. 27, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=7HVPjkWOrLE

[8] Phil's Lab. *FIR Filter Design and Software Implementation - Phil's Lab #17*. (Dec. 20, 2020). Accessed: Feb. 27, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=uNNNj9AZisM

[9] ardnew, “Ili9341-STM32-hal,” GitHub, https://github.com/ardnew/ILI9341-STM32-HAL (accessed Feb. 27, 2024).

[10] bao-eng, “BMI088-STM32,” GitHub, https://github.com/bao-eng/BMI088-STM32 (accessed Feb. 27, 2024).

[11] E. Seok, “stm32_hal_dshot,” GitHub, https://github.com/mokhwasomssi/stm32_hal_dshot (accessed Feb. 27, 2024).

[12] E. Seok, “stm32_hal_nrf24l01p,” GitHub, https://github.com/mokhwasomssi/stm32_hal_nrf24l01p (accessed Feb. 27, 2024).

[13] “ICP-10111 Datasheet.” InvenSense, San Jose, 2021.

[14] P. Salmony, “HADES Flight Control System,” GitHub, https://github.com/pms67/HadesFCS/tree/master (accessed Feb. 27, 2024).

[15] “Grove - 6-axis accelerometer&gyroscope(bmi088): Seeed studio wiki,” Seeed Studio Wiki RSS, https://wiki.seeedstudio.com/Grove-6-Axis_Accelerometer&Gyroscope_BMI088/ (accessed Mar. 16, 2024).

[16] “nRF24L01+ Single Chip 2.4GHz Transceiver Product Specification v1.0.” Nordic Semiconductor, Trondheim, 2008.

[17] “STM32H755xI.” STMicroelectronics, Santa Clara, 2023.

[18] “a-Si TFT LCD Single Chip Driver 240RGBx320 Resolution and 262K color Specification.” ILI Technology Corp., Zhubei, 2011.

[19] “BMI088: Data Sheet.” Bosch, Sunnyvale, 2022.

[20] “SpeedyBee BLS 50A 30×30 Stack User Manual.” Zhuhai KuaiFeng Technology Co., Zhuhai, 2022.

[21] D. Fox, W. Burgard, and S. Thrun, “The dynamic window approach to collision avoidance,” Carnegie Mellon University: Robotics Institute, https://www.ri.cmu.edu/pub_files/pub1/fox_dieter_1997_1/fox_dieter_1997_1.pdf (accessed Mar. 14, 2024).