Dizzy n-copter

Controlled ‘Chaos’ in the Air

Divide and Conquer Document

EEL4914 - Senior Design I

Group 29

Authors:

Corey Barbera, Computer Engineer

Carlos Alfonso Vargas, Computer Engineer

Alex Bayda, Electrical Engineer

James Thompson, Electrical Engineer

Sponsored by Mike Borowczak in the DRACO Lab

Reviewer Committee:

Dr. Mike Borowczak, Dr. Aman Behal , Dr. Truong Nghiem, Dr. Chinwendu Enyioha

Mentor: Sreeram Sundaresh

Table of Contents

Chapter 1 - Executive Summary

The Dizzy n-copter project is a sub-250-gram drone that is designed to achieve continuous yaw rotation while maintaining stable flight while reliably collecting data. The primary goal of this design is to collect data while spinning at approximately 60 RPM. This motion enables the drone to perform panoramic sensing and telemetry logging using onboard sensors without the need for a complex mechanical stabilization system.

To meet the strict weight and performance requirements, the drone utilizes a three-motor configuration with independent servo-controlled tilting mechanism that will provide both lift and directional control. Using an ESP-32 for the onboard flight controller that will serve as the core processing unit for the flight stabilization and data management. Power is supplied by a 3S 850 mAh Li-Po battery, providing efficient and sustained operation for a lasting flight time of 5 minutes. This will support Wi-Fi and Bluetooth telemetry with the base station for real-time monitoring and data transfer.

A custom-designed 3D-printed frame ensures both strength and minimal weight when using materials like LW-PLA or LW-ASA. This will help keep the total system mass below the FAA's 250-gram threshold. The project's software stack integrates Betaflight for flight control and custom UART-based communications for sensor data and servo coordination.

The Dizzy n-Copter represents a multidisciplinary engineering effort combining control systems, embedded software, PCB design, and additive manufacturing. The end goal is to produce a fully functional and lightweight drone capable of continuous rotation and environmental telemetry. Serving as proof of concept for future 360 degree mapping or scanning applications.

Chapter 2 - Project Description

2.1 Motivation & Background

In today's world, unmanned aerial systems (UAS) have become essential across research, industry, and defense. While most drones are designed for steady, stabilized flight, our project takes a fundamentally different approach. Unlike other conventional drones, our drone is engineered to continuously spin around its vertical axis. Now this unique feature of the drone isn't something that is a challenge to overcome, but the mechanism that makes it special.

High-end drone systems on the market today use specialized rotating sensors to gather data, track movement, and map out large areas. Some of them include sensors that are powerful, but financially out of reach since they can cost tens of thousands of dollars. One of the most common examples is LiDAR. Rather than employing conventional photo cameras, LiDAR sensors send out rapid laser pulses and capture the responses. Using those data points to map out an area with a great deal of both precision and accuracy [1]. The base drone that includes LiDAR can exceed $10,000. To address this challenge and make data gathering more cost-efficient, our team was given an idea for a drone that spins to capture sensor data. Effectively serving as its own

low-cost LIDAR system. Allowing for easier and more affordable environmental mapping, but also giving us the ability to add a unique “spin” on drone technology.

We have the privilege of working under Dr. Mike Borowczak in the Design of Resilient Architectures for Computing (DRACO) Lab at the University of Central Florida. A research group that is focused on robotics, embedded systems, and aerospace innovation. This sponsorship aligned perfectly with project ideas that we had already been brainstorming as a team, but with a unique "twist". It has given us an incredible opportunity to pursue a project idea we envisioned while benefiting from the guidance of an experienced mentor.

2.2 Review of Related Ideas

2.2.1 Related Project: Continuously Rotating LiDAR MAV

One directly related project that we could find was the continuously rotating perception system developed by Droeschel, Behnke, and Holz (2013) for micro aerial vehicles. Their design mounts a lightweight 3D laser scanner onto a platform that rotates at a constant speed. Enabling full 360-degree spatial mapping. By intentionally spinning the sensor rather than the aircraft, the system achieved high-resolution omnidirectional awareness suitable for navigation and obstacle avoidance. Although highly capable, this approach required specialized LiDAR hardware and precise synchronization between the spinning sensor and the flight controller. Such complexity and cost make the system impractical for small, lightweight platforms, but the underlying concept. Using rotation to expand sensor coverage, which kind of fits into the goals for our project.

2.2.2 Another Related Project: The Quadichopter Rotational Multirotor

Another project closely aligned with the goals of our drone is the Quadichopter, an experimental quadcopter platform designed to operate while continuously spinning around its vertical axis [DrehmFlight, 2024]. Unlike traditional multirotors that avoid unintentional yaw rotation, the Quadichopter intentionally incorporates constant body rotation as part of its flight behavior. Its design uses a symmetrical four-motor layout where the drone spins by default, and thrust vectoring is achieved through dynamic motor balancing. This rotational motion allows the drone to achieve a simplified form of control while producing visually stable video using a lightweight onboard camera.

The purpose of the Quadichopter project was to explore the feasibility of actively spinning multirotors as a means of reducing mechanical complexity and enabling unique forms of stabilization and perception. Because the drone rotates continuously, the platform can capture panoramic or spiraling imagery from a simple fixed camera without requiring a gimbal. The project also demonstrated that conventional flight controllers and PID loops can be tuned to maintain stability even when the airframe is undergoing rapid yaw rotation, proving that rotational dynamics can be predictable and manageable.

Although the Quadichopter does not incorporate sensors for environmental mapping, its intentional spinning motion and motor-driven control approach directly relate to the Dizzy

n-Copter. It validates the concept that a drone can maintain controlled flight while continuously yawing, and it highlights how rotation can be used not only for propulsion efficiency but also for visual perception. The project serves as real-world evidence that continuous rotation and stable flight are compatible, reinforcing the feasibility of our design, which extends the idea further by integrating a 360-degree sensing payload for environmental scanning

2.2.3 Summary

Recent advancements in drone technology have demonstrated that controlled rotation around the Z-axis, or yaw, can be achieved through the coordinated use of propellers, as seen in standard quadcopter designs where opposing propellers spin in opposite directions. Beyond its traditional use for navigation, researchers have explored intentional self-rotation of drones to expand sensor coverage. For example, Droeschel et al. (2013) developed a continuously rotating lightweight 3D laser scanner for micro aerial vehicles, enabling omnidirectional perception and mapping [2]. Similarly, a more recent design introduced a self-rotating, single-actuated UAV that increased sensor field of view while maintaining controllability [3]. These works illustrate that spinning platforms can be leveraged for perception rather than movement alone.

Our project builds on these concepts by integrating a sensor system designed to function in a manner similar to LIDAR technology. By enabling the drone itself to rotate while collecting distance measurements, the system can serve as a full 360-degree scanning tool for mapping and environmental sensing. While rotating 3D LiDAR systems have been successfully implemented on drones for high-resolution mapping [4], their high cost limits broader accessibility. Research on ultrasonic-based scanning and collision avoidance provides evidence that lower-cost alternatives can still support environmental awareness, though at reduced fidelity [5][6]. This highlights the opportunity for more affordable solutions like ours, which seek to balance functionality with cost-effectiveness.

Such an approach not only connects to existing drone applications but also suggests expanded use in areas such as infrastructure inspection, environmental monitoring, and search-and-rescue operations. As demonstrated in prior work, spinning drones and rotating sensor systems provide a strong foundation for achieving reliable 360-degree scanning, and our design aims to extend these benefits in a more accessible form.

2.3 Overview of Project Goals and Objectives

The idea behind this project is to build an affordable alternative to LIDAR drones. Creating a 3D printed carbon fiber micro-drone where the main body spins around its Z-Axis while being fully maneuverable indoors.

2.3.1 Goals

Basic Goals:

- Develop a fully functional drone that maintains a constant rotational frame speed of 60 RPM.

- Communicate key telemetry such as spin rate, battery life, and flight status to a base station in real time.
- Create an Indoor-safe drone that has enclosed propellers to protect the propellers themselves as well as others around.
- Achieve a minimum 5-minute flight time through optimized battery usage.
- Keep the complete build weight of the drone under 250 g to remain compliant with sub-250 g safety classifications
- Implement basic autonomous safety features such as automatic landing on low battery or lost signal.

Advanced Goals:

- Develop a ground application capable of controlling the drone and displaying telemetry logs in real time.
- Use hardware PID control to stabilize spin rate and correct disturbances.
- Integrate a distance sensor payload for altitude awareness and collision prevention.
- Enhance onboard processing to support low-latency feedback control loops.
- Preserving full translational movement (forward, backward, left, right).

Stretch Goals:

- Extend flight control software to support obstacle avoidance and semi-autonomous navigation.
- Incorporate wireless charging capabilities for convenient recharging without physical connectors.
- Design a modular frame that allows rapid replacement of propellers, guards, or battery compartments.
- Enable local control as an alternative to base station commands.

2.3.2 Objectives

Basic Objectives:

- Implement a spin-stabilization system that maintains 60 RPM ±5% during sustained hover.
- Develop translational flight control algorithms that achieve commanded speeds while maintaining spin stability.
- Stream telemetry data (spin rate, battery voltage, flight mode) to a base station at a minimum of 10 Hz.
- Integrate full propeller enclosures and verify protection effectiveness during light collisions.
- Demonstrate at least 5 minutes of continuous flight time under normal operating conditions.
- Confirm that the drone’s all-up weight does not exceed 250 g by final assembly.

- Implement fail-safe features including automatic landing during low-battery events or loss of communication with the base station.

Advanced Objectives:

- Develop a mobile application to send flight commands and display live telemetry.
- Offload spin-control loops to dedicated hardware PID controllers running at ≥500 Hz.
- Integrate a distance sensor capable of measuring from 0.2–2.0 m with an accuracy of ±3 cm.
- Implement simple altitude-holding logic using the distance sensor to maintain a stable hover range.
- Enhance telemetry protocols to minimize packet loss (<1%) and latency (<200 ms).

Stretch Objectives:

- Implement forward obstacle detection within a 90° arc and trigger automatic avoidance maneuvers.
- Demonstrate wireless inductive charging with at least 70% charging efficiency and successful drone docking.
- Design and validate a modular frame that allows component swaps in under 3 minutes with minimal tools.
- Integrate local manual controls with a safety lockout to prevent conflict with automated modes.

2.4 Project Functionalities

The Dizzy n-copter system is designed not only to demonstrate stable flight while spinning, but also serve as proof of a low cost LiDAR. The functionalities below show the objective of the project to ensure both safe operation and data collection. These functions will highlight how the drone will interact with its environment and transmit telemetry back to the base station.

- Continuous Spinning Flight: Maintain stable indoor flight while spinning around its Z-axis at approximately 60 RPM.

- Real-Time Sensing: Collect distance measurements using a mounted ultrasonic sensor during continuous rotations.

- Telemetry Transmission: Communicate sensor reading and spin-rate data wirelessly to a base station

- Stability Control: Utilize hardware-based PID control loops to ensure consistent spin stability (Hardware preferred)

- Safety: Enclosed propellers

- Modularity: Lightweight, modular frame design that allows for replacement or reconfiguration of components.

2.5 Requirement Specifications

Specifications	
Communication rate:	3600 Samples per minute (1 sample per 10 degrees)
Drone	
Performance:	Maintain ~60 RPM rotation about the Z-axis
Weight:	Less than 250 g total system weight
Production:	Modular frame for ease of maintenance and component replacement
Battery:	11.4V 3S 850mAh LiHV rechargeable battery
Minimum Lifespan:	Minimum 5 minutes of continuous operation
Flight Requirements:	Maintain stable hover with minimal drift
Safety:	Enclosed propellers for safe indoor operation
Base Station/BLE	
Display:	Display rotation speed and sensor data via BLE interface
Sensor Integration	Continuous communication with onboard ToF sensor

Figure 2.1. *Specifications diagram*

Notes: These design specifications were based on what the sponsor wanted us to fulfill. These specifications might change due to the different actions taken to make the product as desired.

We must have a flying drone that meets the specifications mentioned above. Smaller details will be added into the final report, addressing any issues or changes made later on in the process.

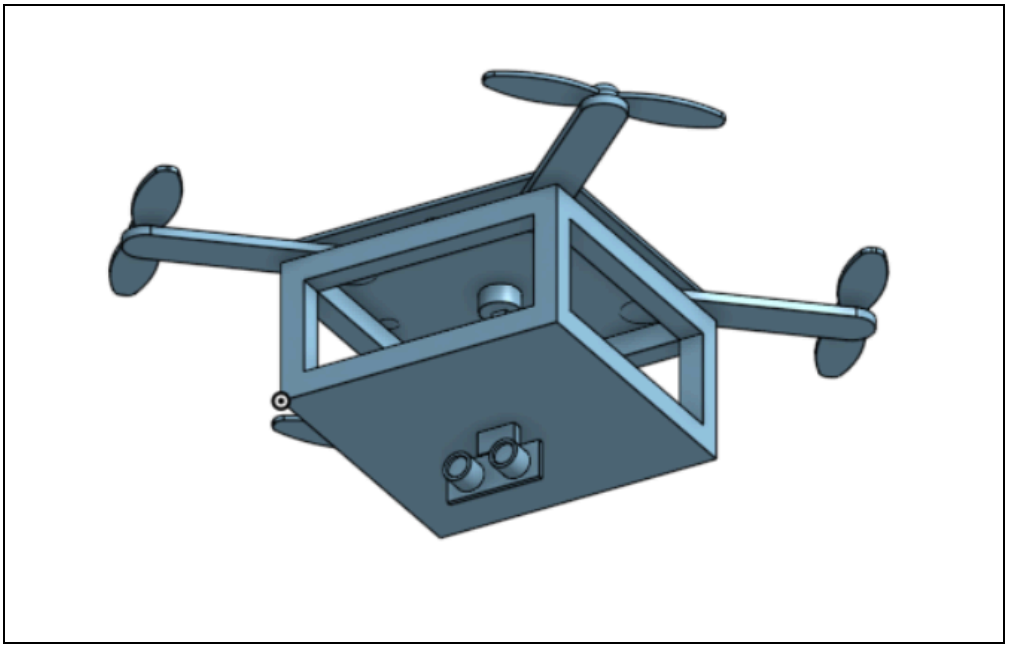


Figure 2.1. *Drone Prototype*

Rotational Scanning Requirements for Ultrasonic Sensor

Objective: Ensure reliable detection of targets during sensor rotation by defining constraints on allowable rotation speed and maximum measurable distance.

1. Critical Rotation Speed

The ultrasonic sensor may miss a target if its rotational sweep rate exceeds the time window in which the echo returns.

The critical angular speed is defined as:

$$w = \theta \cdot c / 2R$$

Where:

θ = sensor beam width (deg)

c = speed of sound in air (~343 m/s)

R = target distance (m)

Converted to rotations per minute (rpm):

$$Rpm = w / 6$$

Requirement:

The rotation speed of the ultrasonic sensor **shall not exceed rpm for the maximum intended detection range R**.

2. Maximum Detection Distance at Fixed Rotation Speed

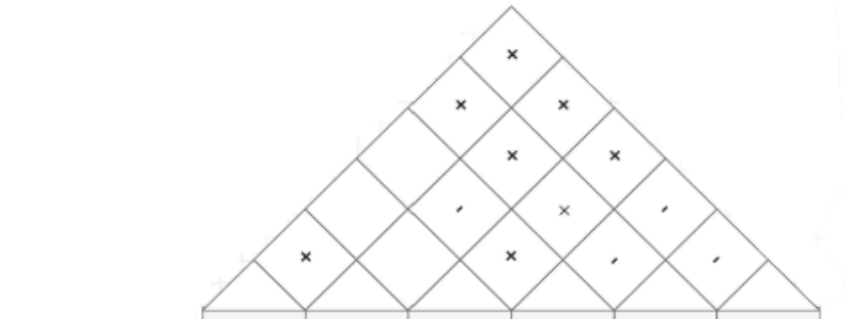
For a given rotation speed w, the maximum distance before missed detections is:
$R_{max} = (\theta / w) \cdot (c / 2)$

Requirement:

At a rotation speed of **60 rpm (360°/s)** with a beam width of **15°**, the sensor shall reliably detect targets up to:

$$R_{max} \approx 7.1 \text{ m}$$

2.6 House of Quality (HoQ)



	(+)Communications	(+)Battery Life	(+)Stability	(-)Amount of Motors	(-)Cost	(-)Weight
Cost(-)	↓	↓	↓	↑	⇈	⇈
Weight(-)	↓		↓	⇈	⇈	⇈
Maneuverability(+)			⇈	⇊	⇊	⇊
Power Usage(-)	↓	⇊		⇈	⇈	⇈
Targets For Engineering Requirements	3600 samples / minute	>5 min	Spinning	<5	<$200	<250 g

Figure 2.3. *House of Quality*

2.7 Hardware (includes block diagram)

The hardware for this project will be developed by the Electrical Engineering team, with support from the Computer Engineering partners. To control the height, spin, and direction of the drone, we divided the design into three main subsections. This structure allows us to adjust the drone's motion and rotation speed based on data collected during testing.

The first subsection consists of the three primary motors and propellers that provide lift. Each propeller spins opposite to the one across from it, enabling smooth takeoff and descent, managed by the motor controller. The second subsection includes four additional motors, which are used to rotate the primary lifting propellers. This system gives us control over the drone's spin, allowing for more stable and precise flight. The third subsection is a single motor with its own controller, responsible for moving the drone across the x- and y-plane. Unlike a traditional drone, this setup eliminates the need to tilt the body for horizontal movement, enabling us to scan the ground below without compensating for tilt during motion or when coming to a stop.

In total, the system will use four PCBs: two main PCBs for communication and power distribution, one dedicated PCB to control the single central propeller, and one PCB in the base station for communication. Power will be distributed through the two main PCBs, while the separate propeller PCB operates as an independent control unit. The central propeller is mounted on a bearing, preventing it from spinning with the rest of the drone and ensuring stable operation during rotation. The drone body will be built from a lightweight, low-cost 3D-printed frame designed to hold the components and support the bearing assembly. Similarly, the base station will use a 3D-printed body and will also include an LCD screen for user interaction.

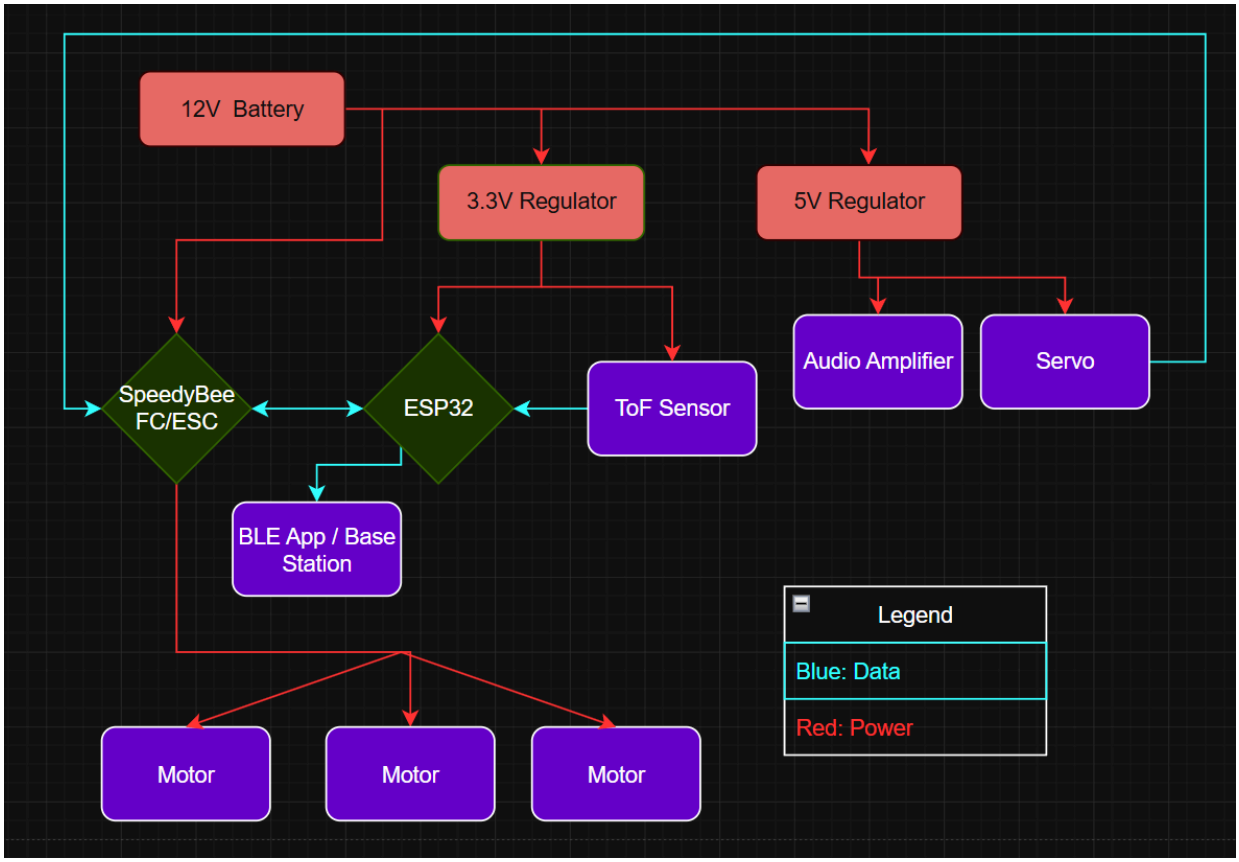


Figure 2.4. *Hardware Diagram*

2.8 Flow Charts (Communication)

The communication for this device will primarily occur between the drone and the base station. The drone will include multiple sensors connected to a dedicated PCB designed for both sensor integration and communication with the base station. Since the drone operates at 90 RPM, we need to collect at least one data point every 10 degrees of rotation, resulting in 36 samples per full turn. This sampling rate will determine the minimum communication speed required. In addition to these rotational data points, other sensors will provide information such as orientation, battery life, and rotational speed. Sensor communication within the drone will be handled using I2C due to its simplicity and compatibility. Transmission of the collected data to the base station will use a different protocol, which is still being determined.

Beyond the sensor communication, there is also a direct communication link between the base station and the drone's single fan, which is responsible for movement. This fan operates as a separate subsystem with its own PCB and battery, functioning as an independent entity within the overall design.

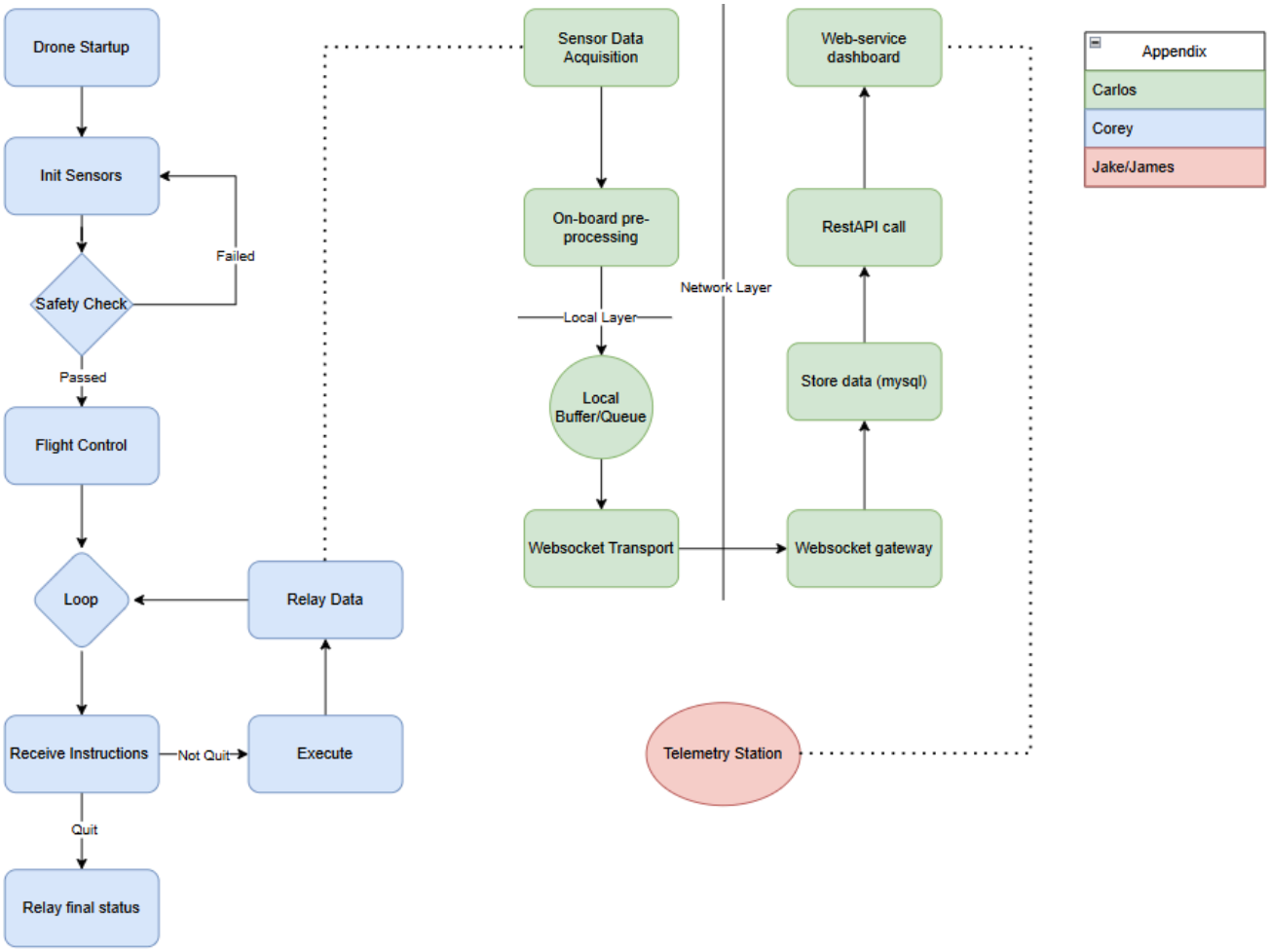


Figure 2.6. *Software-Oriented Flow Chart*

Chapter 3 - Research

3.1 Technologies

3.1.1 Microcontroller Unit (MCU)

3.1.1.1 Overview

For the embedded systems of our project we are going with a microcontroller unit over a field-programmable gate array. The FPGA requires much more power and is much too expensive, while MCUs are versatile and low power. MCUs are built with integrated components like analog-to-digital converters, timers, and communication interfaces like UART, I2C, and SPI, which helps with development.

This makes a microcontroller development board extremely user friendly and highly approachable for developers. They use common programming languages like C, C++, and Python. MCUs come with established development environments and extensive documentation from easy-to-use documentation and online resources, from hobbyist communities and manufacturers. This helps with facilitating debugging and implementation when working on projects where an MCU is required.

In terms of power and stability, microcontrollers are optimized for low-power operation and usually include multiple power-saving modes. They will often have built in error handling, watchdog timers, and other safeguards that improve reliability. MCUs generally can operate under stable conditions with varying temperatures and conditions without the need for additional projections.

3.1.1.2 Why MCUs Are Ideal for Our Drone

To support the control and sensing requirements of the rotating-frame drone, several microcontroller (MCU) and flight-controller options were evaluated. The system requires reliable real-time control of five rotors (four for lift, one for thrust), continuous 60 RPM rotation, and acquisition of ultrasonic sensor data for 360° spatial mapping. Key design criteria include computational performance, power efficiency, physical size, and integration capability with sensors and motor controllers.

3.1.1.3 STM32F4 Series (e.g., STM32F405)

The STM32F4 series from STMicroelectronics is based on the ARM Cortex-M4 core and operates at a maximum clock speed of 168 MHz. These microcontrollers include an integrated floating-point unit (FPU), which allows them to perform complex mathematical operations efficiently. This capability is particularly beneficial for drones, as it enables accurate real-time flight control, precise PID loops, and sensor fusion calculations without the need for additional hardware acceleration.

One of the major advantages of the F4 series is its extensive set of built-in peripherals. These include multiple timers for PWM motor control, analog-to-digital converters (ADC) for reading sensor inputs, and communication interfaces such as UART, I2C, and SPI for connecting to external modules. The F4 series also includes direct memory access (DMA), which allows data to be moved between peripherals and memory with minimal CPU intervention, freeing up processing resources for critical flight tasks.

Programming the F4 series is done primarily in C or C++, using established development environments. Extensive documentation, tutorials, and an active online community make development accessible even to teams new to microcontroller programming. The F4 MCUs are also designed for low-power operation, supporting multiple sleep and low-power modes to extend battery life in lightweight drones. Overall, the STM32F4 series represents a strong balance of computational power, peripheral support, and energy efficiency, making it an ideal choice

3.1.1.4 STM32F7 Series (e.g., STM32F722)

The STM32F7 series builds on the capabilities of the F4 series by incorporating the more advanced ARM Cortex-M7 core and increasing the clock speed to 216 MHz. This higher-performance core, combined with improved cache and memory bandwidth, allows the F7 series to handle more demanding computational tasks such as processing data from multiple high-speed sensors simultaneously or performing advanced filtering algorithms in real time.

Like the F4, the F7 includes integrated peripherals such as PWM timers, ADCs, UART, I2C, SPI, and DMA. However, its higher clock speed and enhanced memory architecture make it capable of supporting more complex operations or handling larger datasets without bottlenecks. This makes the F7 particularly well suited for drones that need to process higher volumes of sensor data or require faster control loop updates to maintain stability under demanding flight conditions.

The trade-offs of the F7 series are its slightly higher power consumption and larger physical size compared to the F4 series. While still relatively compact, these factors must be considered in micro drone applications where weight and energy efficiency are critical. Nevertheless, the F7 provides excellent headroom for future system upgrades or more advanced functionality, such as integrating additional navigation sensors or performing onboard preprocessing for more autonomous control. Development remains accessible using standard STM32 IDEs, libraries, and community-supported firmware, making the F7 a compelling choice for mid- to high-performance drones.

3.1.1.5 STM32H7 Series (e.g., STM32H743)

The STM32H7 series represents the high-end option within the STM32 family, featuring an ARM Cortex-M7 core that can run at speeds up to 480 MHz. These microcontrollers include expanded memory, larger peripheral sets, and higher throughput, which makes them capable of handling the most compute-intensive tasks a drone may require, including onboard SLAM, dense point-cloud mapping, or even simple neural network inference for autonomous navigation.

While extremely powerful, the H7 series comes with notable trade-offs. Its higher clock speed and increased peripheral complexity result in greater power consumption, which may reduce flight time in battery-constrained drones. The larger physical footprint also limits suitability for micro drones under 250 g unless the design can accommodate additional weight and space.

Like the F4 and F7, the H7 provides integrated PWM timers, ADCs, UART, I2C, SPI, and DMA for sensor and actuator connectivity. However, designing for the H7 often requires more careful consideration of memory management, thermal performance, and overall system architecture due to its higher complexity. Programming is still performed in C or C++ using STM32 IDEs, but developers may need more experience to fully exploit the advanced features. Overall, the STM32H7 series is best suited for larger drones or applications requiring heavy onboard computation where weight and power are less critical concerns.

3.1.1.6 ESP32 Series

The ESP32 series from Espressif Systems is designed primarily for networking and IoT applications, integrating Wi-Fi and Bluetooth connectivity into a single compact platform. Each module includes an onboard antenna and at least 4 MB of flash memory, providing both communication and storage in a lightweight design.

Powered by a dual-core processor running at up to 240 MHz with 520 KB of SRAM, the ESP32 delivers strong computational performance for its class. The clock speed can be lowered to 80 MHz for reduced power consumption, making it well suited for energy-efficient systems. It operates on a supply voltage as low as 3.0 V, with a current draw of around 28 mA in normal operation and up to 379 mA during wireless transmission.

The ESP32 provides a wide range of integrated peripherals, including UART, SPI, I2C, PWM, and ADC distributed across 26 GPIO pins. These features enable direct connection to sensors, actuators, and other devices without additional interface hardware. Despite its broad capability set, the ESP32 remains one of the most affordable development boards available, typically priced around $10 USD.

Its combination of low cost, low power usage, and comprehensive connectivity makes the ESP32 series one of the most versatile and accessible options for embedded systems and wireless control applications

3.1.1.7 MCU Comparison Table

Table 3.1. *MCU Comparison*

Feature	**STM32F4**	**STM32F7**	**STM32H7**
Core	ARM Cortex-M4	ARM Cortex-M7	ARM Cortex-M7
Clock Speed	168 MHz	216 MHz	Up to 480 MHz
Floating-Point Unit (FPU)	Yes	Yes	Yes
Flash Memory	1 MB	1-2 MB	2-4 MB
RAM	192 KB	512 KB	Up to 1 MB
DMA / Timers / PWM	Yes	Yes (enhanced)	Yes (advanced)
Power Consumption	Low	Moderate	High
Board Size	Compact (20x20 mm)	Medium (20x30 mm)	Larger (35x35 mm)
Use Case	Standard flight	Multi-sensor drones,	SLAM, onboard AI,

	control, sensor fusion	higher-rate filtering	mapping
Voltages	1.8V to 3.6V	1.7V to 3.6V	1.62V to 3.6V
Advantages	Low cost, low power, strong community	Higher performance, better memory	Extreme processing power
Disadvantages	Limited headroom for upgrades	Higher power draw	Heavy, complex, expensive

3.1.2 Flight Controllers

A flight controller (FC) serves as the central processing of a drone. It integrates sensors, software, and communication modules to control the drone's flight and stability. It essentially acts as the "brain" of the drone, allowing it to fly smoothly and respond to the pilot's inputs. Some of the core responsibilities of the fight controller is to manage all real-time control operations, including stabilization, attitude correction, sensor fusion, and motor output. Flight controllers sometimes use other firmwares like Betaflight, PX4, or ArduPilot.

FCs typically consist of a microcontroller (MCU), gyroscope, and accelerometer, sometimes it will also include a magnetometer and barometer for altitude and orientation sensing. Many boards include stuff like integrated power management, onboard data logging (black box storage), and communication interfaces like UART, I2C, and SPI for external modules like GPS, ESCs, and telemetry.

3.1.2.1 SpeedyBee F405 Mini (STM32F405)

The SpeedyBee F405 Mini is a compact and efficient flight controller built around the STM32F405 microcontroller, part of the STM32F4 series renowned for reliable real-time performance and energy efficiency. The MCU operates at 168 MHz with an integrated floating-point unit, offering ample processing power for PID control, sensor fusion, and ultrasonic ranging tasks. Its 20×20 mm mounting size makes it ideal for lightweight UAVs, including sub-250 g drones where compactness is essential.

One of its main advantages is the integrated 4-in-1 ESC connector, which reduces wiring complexity and improves overall balance on a rotating frame. The board also includes onboard Bluetooth connectivity, allowing wireless tuning and configuration through the SpeedyBee app. The F405 Mini supports popular firmware options such as Betaflight and INAV, ensuring flexibility for both manual and semi-autonomous flight modes. With its combination of small size, dependable MCU performance, and broad community support, the SpeedyBee F405 Mini offers an optimal balance of capability and simplicity for this drone design.

3.1.2.2 RDQ Bardwell F7 V2 (STM32F7)

The RDQ Bardwell F7 V2 utilizes the STM32F7 microcontroller, offering higher computational power and enhanced memory performance compared to F4-based flight controllers. Operating at 216 MHz, the Cortex-M7 core enables faster control loop execution, improved signal filtering, and smoother flight dynamics. This makes the board well-suited for drones requiring high sensor data rates or advanced stabilization algorithms.

The board's larger 30x30 mm mounting pattern provides space for additional input/output ports, including multiple UART interfaces for GPS, telemetry, and external sensors. Its layout also includes a built-in black box logger for flight data analysis, enabling precise performance tuning. The RDQ Bardwell F7 V2 supports Betaflight and INAV firmware, ensuring a familiar software ecosystem for configuration and testing. While it consumes slightly more power and occupies more space than smaller F4 controllers, the improved processing headroom and flexibility make it ideal for higher-performance drone applications where size and weight constraints are less restrictive.

3.1.2.3 Lumenier Lux Mini F7 V2 (STM32F722)

The Lumenier Lux Mini F7 V2 flight controller integrates the STM32F722 MCU, part of the high-speed F7 family optimized for low-latency control and multi-sensor data processing. Operating at 216 MHz, the controller efficiently handles advanced filtering and high-frequency PID loops, resulting in smoother and more stable flight performance. Its small 20x20 mm format is particularly advantageous for compact drones where internal space is limited but processing power cannot be compromised.

Hardware features include multiple UART ports, an onboard OSD (On-Screen Display) chip, and support for both analog and digital video systems. The board's vibration-resistant mounting and clean power regulation further enhance signal integrity for sensor inputs. Firmware compatibility with Betaflight and INAV allows flexible setup for manual and autonomous flight profiles. The Lumenier Lux Mini F7 V2 strikes a balance between the raw processing strength of larger F7 boards and the minimal footprint of F4 designs, making it a strong choice for lightweight UAVs requiring advanced control without increasing system mass

3.1.2.4 Matek H743-WLITE (STM32H7)

The Matek H743-WLITE represents one of the most powerful flight controllers in this evaluation, built around the STM32H743 MCU from the STM32H7 series. The ARM Cortex-M7 core operates at 480 MHz and provides exceptional computational performance for complex tasks such as SLAM, advanced filtering, and high-resolution sensor integration. This board is targeted at larger or research-grade UAVs where onboard computation is essential.

In addition to its processing power, the H743-WLITE includes an extensive suite of peripherals, such as dual camera inputs, CAN bus support, dual gyroscopes, and onboard barometer sensors.

Its design emphasizes modularity and expandability, accommodating diverse payload and navigation systems. Despite its high performance, the H743-WLITE is physically larger and consumes more power than F4 or F7 boards, making it less ideal for small, compact drones. However, it offers a robust upgrade path for future drone iterations requiring enhanced autonomy, mapping, or AI processing capabilities.

3.1.2.5 TBS Lucid Pro (AT32F435)

The TBS Lucid Pro is a versatile and durable flight controller that uses the AT32F435 microcontroller, based on the ARM Cortex-M4 architecture. This MCU operates at 288 MHz and provides performance comparable to the STM32F4 series while maintaining cost efficiency and low power consumption. The board's design focuses on reliability, compactness, and easy integration with modern ESCs and receiver systems.

It includes multiple UART, SPI, and I2C interfaces, allowing flexible communication with GPS, telemetry, and sensor modules. The Lucid Pro supports both analog and digital signal processing, enabling compatibility with a wide range of peripherals. Its sturdy layout and reinforced mounting holes make it well-suited for drones operating under high vibration conditions. Firmware compatibility is somewhat more limited than STM32-based boards, but it remains functional within standard open-source flight stacks. Overall, the TBS Lucid Pro offers a balance of performance, durability, and affordability, making it an attractive option for mid-range drones and experimental UAV builds.

3.1.2.6 Flywoo Goku F722 Pro Mini (STM32F7)

The Flywoo Goku F722 Pro Mini is a compact, high-performance flight controller powered by the STM32F7 series MCU. Running at 216 MHz, it provides fast processing for advanced flight algorithms, ensuring stable and responsive control during high-speed maneuvers. With a 20x20 mm mounting pattern, it is optimized for micro-drones where compactness and minimal weight are critical.

The board includes integrated features such as a barometer for altitude control, onboard black box storage for data logging, and multiple UART ports for peripheral expansion. It also offers soft-mounted vibration isolation, which improves IMU accuracy by reducing mechanical noise from motors. The Flywoo Goku F722 Pro Mini supports Betaflight and INAV firmware, giving developers flexible control over tuning and setup. Combining a small form factor with F7-level performance, this board is ideal for high-efficiency small, compact drones that demand fast response and reliable operation.

3.1.2.7 Diatone Mamba F405 MK2 (STM32F405)

The Diatone Mamba F405 MK2 is a robust and widely used flight controller that employs the STM32F405 MCU, known for stable performance and low latency. Operating at 168 MHz, it provides sufficient power for PID control, sensor fusion, and reliable signal processing across

multiple sensors. Its 30×30 mm mounting size offers flexibility for integration into a range of drone sizes, from compact builds to larger aerial platforms.

The board features dual gyroscopes for improved stability, integrated OSD functionality, and convenient solder pads for VTX and camera modules. Its modular layout simplifies assembly and repair, making it popular among drone builders. Firmware compatibility with Betaflight ensures easy configuration, while built-in voltage regulation maintains consistent performance even under variable load. Though slightly heavier than mini-format options, the Mamba F405 MK2 remains an affordable and dependable controller that delivers proven results for general UAV applications.

3.1.2.8 Mini PIX (STM32F405 Pixhawk Variant)

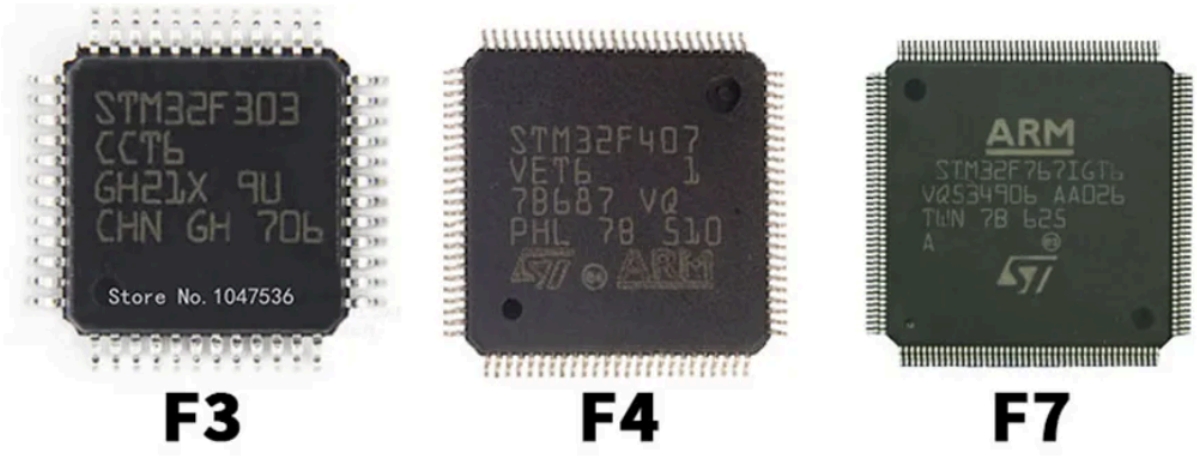
The Mini PIX flight controller is a compact version of the Pixhawk platform built around the STM32F405 MCU. It supports advanced autopilot firmware such as ArduPilot and PX4, enabling waypoint navigation, autonomous flight, and precise mission planning. The 32-bit ARM Cortex-M4 processor running at 168 MHz provides sufficient computational resources for real-time stabilization, GPS navigation, and multi-sensor fusion.

Physically smaller than a standard Pixhawk, the Mini PIX retains many professional-grade features, including redundant IMUs, barometer sensors, and multiple UART and I2C ports for telemetry and GPS. It also supports logging and power monitoring for system diagnostics. While its feature set exceeds what is required for a small, compact drone, its comprehensive functionality makes it well-suited for research, mapping, or semi-autonomous UAV projects. The Mini PIX bridges the gap between hobbyist and professional flight controllers, offering a strong foundation for future system expansion or integration into more complex aerial platforms.

3.1.2.9 Flight Controller Comparison Tables

Feature	F4	F7
Clock Speed	168 MHz	216 MHz
Max Cycle Time	~32 kHz (limited by CPU)	Faster, higher real-time performance
UARTs	3–5	5+ (with built-in inversion)
Advanced Features	Limited	Superscalar pipeline, DSP, more advanced PID algorithms
IMU Performance	Standard	Often paired with higher-end IMUs
Pros	Good for racing/freestyle	Better for advanced flight algorithms, object avoidance, autonomous flight, higher response speed

Figure 3.13. *Comparison between F4 and F7 (FC)*



User Type	Recommended MCU	Reason
Beginner / Standard FPV	F4	Affordable, stable, supports most FC features
Advanced Racer / Freestyle	F7	Faster CPU, more UARTs, built-in inversion, better future-proofing
Professional / Cutting-edge FPV	H7	Maximum processing, handles complex tasks, ideal for research/autonomous drones
Vintage Builds / Lightweight	F3	Faster than F1, native UART inversion, decent for 4K loops

Figure 3.13. *Comparison between F3, F4, and F7 (FC)*

Feature	F7	H7
Clock Speed	216 MHz	400 MHz
RAM / Flash	512 KB / 2 MB	1 MB / 2 MB (plus optional external memory)
UARTs	5+ (with inversion)	Similar or more, built-in inversion
Cycle Time	High (up to gyro limit)	Very high, can handle extremely complex loops
Firmware Support	Latest Betaflight, Cleanflight	Latest Betaflight, Cleanflight; optimized for H7
Advanced Applications	Acro flights, racing, freestyle	Acro flights, professional FPV, autonomous drones, complex sensor setups
Pros	High-speed, handles advanced algorithms	Extremely fast, best for cutting-edge drones and experimental features

Figure 3.14. *Comparison between F7 and H7 (FC)*

Comparison Summary

The F7 and H7 boards provide higher computational performance and more peripheral interfaces but come with increased size, cost, and power draw. For a 6×6 inch drone, these trade-offs reduce available space for batteries and sensors. In contrast, the SpeedyBee (F4)05 Mini offers a compact 20×20 mm stack format, integrated 4-in-1 ESC compatibility, and onboard Bluetooth connectivity for wireless tuning. These features are critical for rapid development, simplified wiring, and reduced mechanical stress on the rotating assembly.

3.1.3 Sensors

Sensors are an essential component in modern drone systems. They help with enabling flight stabilization, navigation, and environmental awareness. While also collecting real-time data on motion, orientation, distance, and environmental conditions that are used by onboard controllers. This helps with the ability to maintain stability and execute control algorithms. The most common categories of sensors used in aerial vehicles include inertial measurement units (IMUs), distance or range sensors, and environmental sensors such as barometers and magnetometers. When selecting sensors some of the critical factors to look at include accuracy, update rate, weight, power consumption, and communication interface compatibility with the main processing unit (I2C, SPI, or UART).

3.1.3.1 Inertial Measurement Unit (IMU) - Fundamentals

An inertial measurement unit (IMU) is a multi-axis sensing module that is able to measure angular velocity and linear acceleration. This allows a flight controller to estimate orientation,

angular position, and motion dynamics. Most IMUs include an integrated gyroscope and accelerometer, while some also include a magnetometer for heading correction. A gyroscope is able to detect rotational motion in degrees per second (°/s), which is done using the Coriolis effect. The accelerometer measures linear acceleration, which is in meters per second squared (m/s^2). When used in combination, these signals provide the basis for attitude estimation through sensor fusion algorithms such as complementary or Kalman filtering.

Some of the key performance parameters include noise density, bias stability, output data rate (ODR), and full-scale range. A low noise density ensures that small angular changes are detectable, while good bias stability minimizes drift during long-term operations. The ODR defines how quickly new data can be produced. Values above 8 kHz are preferred for high-speed control loops to minimize phase lag in feedback systems.

Typical digital interfaces include I2C and SPI. SPI is favored for low-latency applications because of its deterministic timing and higher throughput. Some modern IMUs integrate digital filters, temperature compensation, and FIFO buffers to smooth out vibrations and reduce microcontroller processing overhead. Physical design can also be crucial. Soft-mounting the IMU on foam or rubber standoffs isolates it from motor-induced vibration, reducing measurement artifacts.

- **MPU-6000/6050:** A widely used MEMS IMU offering 3-axis gyro and 3-axis accelerometer with 16-bit resolution. It supports up to 8 kHz gyro bandwidth and is compatible with both SPI and I2C interfaces. Its popularity stems from mature driver libraries and low cost, though it has moderate temperature drift.

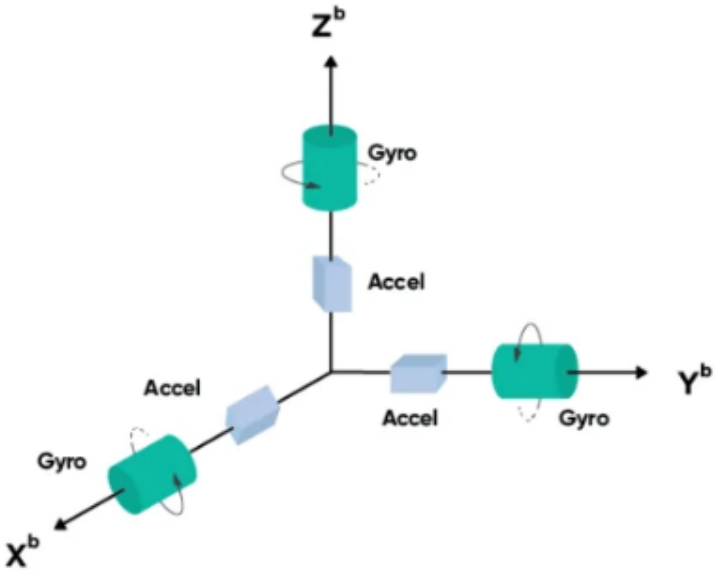
- **BMI270:** A low-power, 6-axis IMU featuring built-in motion detection, 6.4 kHz ODR, and advanced digital filtering. It is optimized for portable systems and can offload simple motion computations through its internal DSP core.

- **ICM-42688-P:** A high-precision, 6-axis IMU offering exceptionally low noise, programmable ODR up to 32 kHz, and a digital low-pass filter with selectable cutoff frequencies. It supports both SPI and I2C and includes an integrated temperature sensor for bias correction.

Table 3.1.3.1: *IMU Devices Comparison*

Device	Axes	Max ODR	Interface	Noise Density	Typical Current	Notes
MPU-6000	6	8 kHz	SPI / I2C	0.01	3.6 mA	Proven, low cost
BMI270	6	6.4 kHz	SPI / I2C	0.007	2 mA	Low-power MEMS
ICM-42688-P	6	32 kHz	SPI / I2C	0.005	2.5 mA	High precision

Each IMU family presents a trade-off between power consumption, precision, and computational load. Older sensors like the MPU-6000 remain attractive for cost-sensitive designs. While newer devices such as the ICM-42688-P and BMI270 provide improved noise performance and higher data throughput that support more responsive and stable control systems.



3.1.3.2 Distance and Range Sensors

Distance or range sensors enable a drone to measure the proximity to surrounding objects or the ground. This allows it to maintain altitude, avoid obstacles, or generate spatial maps. These sensors operate on different physical principles (acoustic, optical, or laser-based) and differ significantly in range, resolution, sampling rate, and environmental robustness. Selecting the proper technology depends on required accuracy, environmental conditions, and power constraints.

Ultrasonic Sensors:

Ultrasonic sensors, such as the HC-SR04 and US-100, operate by transmitting a short burst of high-frequency sound (typically around 40 kHz) and measuring the time delay between transmission and echo reception. The measured delay is converted into distance using the speed of sound in air (~343 m/s at 20°C). These sensors are inexpensive, simple to interface using GPIO trigger/echo pins, and effective for short distances (20 cm–4 m).
However, ultrasonic sensors suffer from several limitations: readings can be distorted by air turbulence from propellers, temperature gradients, or soft surfaces that absorb sound. Their update rate (≈10–20 Hz) is slower than optical alternatives, and angular sensitivity can produce inaccurate readings if the sensor is not orthogonal to the target. Despite these drawbacks, they

remain attractive for low-cost or short-range applications where precision and speed are not critical.

Time-of-Flight (ToF) Sensors:

Time-of-Flight sensors determine distance by emitting modulated light (usually near-infrared) and measuring the phase shift or travel time of reflected photons. Devices such as the VL53L0X and VL53L1X from STMicroelectronics can measure distances from a few centimeters to approximately 4 meters with ± 3% accuracy and update rates of up to 50 Hz.

ToF sensors are solid-state, lightweight (< 5 g), and communicate digitally via I²C, making them easy to integrate into compact systems. They also perform well in low-light environments and are largely unaffected by air movement. However, ToF sensors can exhibit reduced accuracy on highly reflective or transparent surfaces and are sensitive to direct sunlight or strong ambient infrared interference.

Some modern ToF arrays, such as the VL53L8CX, provide multi-zone ranging (8×8 pixel grids) that allow for depth mapping or rudimentary obstacle detection without mechanical scanning. These capabilities come at the expense of higher computational demand and power consumption.

Lidar-Based Systems:

Lidar (Light Detection and Ranging) systems extend ToF principles using higher-energy laser diodes and precision timing circuits to measure distances up to several tens of meters. Compact examples include Garmin's Lidar-Lite v3 and Benewake TFmini. They offer superior range and accuracy (±2.5 cm typical) and can operate at higher sampling rates (100–1000 Hz), which is advantageous for autonomous navigation.

The trade-off is size, cost, and current draw—typical units weigh 20–35 g and require 200–400 mA during operation. Additionally, Lidar systems often need optical isolation and dedicated voltage regulation to mitigate noise injection into sensitive electronics. These factors make them better suited for mid- to large-scale drones or mapping-oriented applications rather than micro aerial vehicles.

Infrared Proximity Sensors:

Simpler infrared (IR) reflective sensors such as the Sharp GP2Y0A21 use triangulation rather than ToF measurement. They are compact and low-cost but have nonlinear output curves and perform poorly under varying ambient light. While useful for simple obstacle avoidance, they are less reliable for precise distance measurement.

Table 3.1.3.2: Range and Distance Sensor Comparison

Technology	Example Model	Range	Accuracy	Update Rate	Interface	Pros	Cons
Ultrasonic	HC-SR04	2-400 cm	±10%	10-20 Hz	GPIO	Low cost, simple	Airflow noise, slow

Optical ToF	Vl53L1X	3-400 cm	±3%	30-50 Hz	I2C	Compact, accurate	IR interference
Multi-zone ToF	VL53L5CX	2-400 cm	±3%	15-30 Hz	I2C	Depth mapping	Higher power
LiDAR	LiDAR-Litev3 / TFmini	10-40 m	±2.5 cm	100-1000 Hz	UART /PWN	Long range	Heavy, high current
IR Triangulation	Sharp GP2Y0A21	10-80 cm	±15%	20-50 Hz	Analog	Tiny, cheap	Nonlinear output

3.1.3.3 Environmental Sensors

Environmental sensors give a system awareness of the world around it, allowing it to sense conditions like air pressure, magnetic fields, and ambient temperature. While these sensors may not directly control flight, they play a key role in improving accuracy, stability, and environmental adaptation. In drone and robotics applications, they are often used to refine altitude estimation, orientation, or environmental calibration.

One of the most common environmental sensors is the barometer, which measures air pressure to estimate altitude. By tracking how atmospheric pressure changes with height, a barometer can provide smooth, continuous altitude readings that complement data from other sensors such as accelerometers or range finders. Modern barometric sensors like the BMP280 or BMP388 are extremely precise, detecting height changes of less than ten centimeters while using almost no power. These sensors are lightweight and connect easily over I2C or SPI interfaces, making them well-suited for small, battery-powered systems. To ensure accuracy, they are usually mounted in an area shielded from propeller airflow, since turbulent pressure changes can distort readings.

Another key environmental sensor is the magnetometer, which measures the strength and direction of the Earth's magnetic field. When combined with gyroscope data, a magnetometer can determine heading or yaw orientation over time. Devices like the QMC5883L and HMC5883L are popular because they are compact, energy efficient, and easy to interface with microcontrollers. However, magnetometers are sensitive to electromagnetic interference from motors, wires, and metal components. To minimize distortion, they are often placed away from high-current areas and calibrated through software routines that correct for nearby magnetic disturbances.

Optical flow sensors add another layer of environmental awareness. They use small cameras to take continuous images of the ground or nearby surfaces and calculate movement by comparing how patterns shift from frame to frame. This lets a system determine horizontal velocity and

detect drift, even in areas where GPS signals are unavailable. Modules like the PMW3901 and PX4Flow include built-in processing that handles the heavy image calculations, outputting easy-to-read motion data to the controller. These sensors perform best when flying over textured surfaces at low to medium altitudes and under good lighting conditions.

Finally, temperature and humidity sensors like the BME280 or SHT31 provide additional environmental information that helps calibrate other sensors. For example, temperature data can be used to correct bias drift in IMUs or barometers, ensuring measurements remain accurate as conditions change. While these sensors don't directly impact flight control, they improve system calibration and can also be useful for environmental monitoring during testing.

Together, these environmental sensors enhance the precision and reliability of an aerial system. By providing consistent altitude, orientation, and environmental data, they help maintain stability and ensure that the drone or embedded platform can adapt to changes in pressure, temperature, and magnetic interference. Their small size, low power use, and digital interfaces make them a practical addition to modern control systems that rely on dependable, high-quality sensor feedback.

3.1.3.5 Integration Considerations

Bringing multiple sensors together in an aerial system requires attention to the electrical, mechanical, and software aspects of design. Each of these areas affects how well the sensors work together and how accurate their data will be once the system is operating.

On the electrical side, most sensors communicate over shared buses like I2C or SPI. If too many devices share the same lines without proper configuration, it can cause interference or timing delays that lead to unreliable readings. High-speed sensors, such as IMUs or optical flow modules, often need their own dedicated connections to make sure data is transferred quickly and consistently. Maintaining clean and stable signals is also important. Using decoupling capacitors, proper grounding, and small filters can help reduce electrical noise. Another key factor is voltage compatibility. Many digital sensors run at 3.3 volts, while other components might use 5 volts. Without proper level shifting or conversion, this mismatch can lead to corrupted data or even damaged parts.

Mechanical placement also plays a major role in sensor performance. The IMU, which measures motion and rotation, should be mounted as close to the drone's center of mass as possible to reduce measurement error. Distance sensors, such as ultrasonic or ToF modules, need a clear and unobstructed view of their target to return accurate readings. Barometers should be protected from direct airflow caused by the propellers but still exposed to ambient air pressure. Even small details like using vibration-damping foam or rubber mounts can make a noticeable difference by preventing mechanical vibrations from affecting sensitive MEMS sensors.

The software side ties everything together. Sensor data often needs to be filtered and combined so that the flight controller can interpret it correctly. Algorithms such as complementary filters or Kalman filters are used to merge readings from multiple sensors into a single, stable estimate of position, speed, or orientation. Calibration routines correct for small offsets or biases that naturally develop over time, ensuring each sensor stays accurate. Filtering techniques, like

low-pass or notch filters, help remove electrical or mechanical noise without introducing too much delay in control response. Finally, keeping all sensor readings synchronized with the control loop ensures that the system responds quickly and accurately to changes during flight.

When electrical, mechanical, and software considerations are all handled carefully, the sensors work together as a cohesive system. The result is more accurate data, smoother flight control, and a more reliable platform overall. These are qualities that are essential for any embedded or autonomous drone design.

3.1.4 Communication

Since our drone rotates at 60 RPM (1 rotation per second) and continuously transmits LiDAR data to a base station equipped with hardware and a web interface, it is critical to make the communication system as efficient and reliable as possible. To ensure consistent real-time operation, we carefully evaluated multiple design options for every aspect of the communication pipeline. This section outlines the key design decisions and rationale behind them.

3.1.4.1 Data Rate and Bandwidth

The LiDAR on our drone generates a substantial volume of data—millions of points per second. Managing this data efficiently is essential to maintaining real-time streaming without overloading the communication link or processing hardware.

Point cloud compression: We compress LiDAR data before transmission to reduce bandwidth requirements while preserving data fidelity.
Pros: Minimizes required bandwidth, compatible with standard protocols, reduces network congestion.
Cons: Adds processing load on both the drone and the base station, potentially impacting real-time performance if computational resources are limited.

Voxel-based downsampling: Reduces the number of points transmitted by lowering point density.
Pros: Significantly reduces data size, lowers bandwidth requirements.
Cons: Loss of detail could negatively affect mapping accuracy and environmental representation.

We have chosen to implement point cloud compression as our primary data reduction method. This approach balances the need for high-fidelity LiDAR data with the practical constraints of bandwidth and processing. We remain open to reconsidering this decision if computational load proves to interfere with real-time performance.

3.1.4.2 Wireless Communication

We require a wireless link capable of transmitting high-volume LiDAR data from the spinning drone to the base station in real time.

Wi-Fi (802.11ac/ax):
Pros: High bandwidth, easy to integrate, low cost, suitable for controlled environments.
Cons: Limited range, potentially susceptible to interference from other devices.

5G/LTE:
Pros: High throughput, low latency, wide coverage if cellular network is available.
Cons: Requires cellular access, potentially costly, additional regulatory considerations.

Proprietary RF/UWB:
Pros: Can be optimized for robustness, low interference in customized frequency bands.
Cons: Development complexity, requires specialized hardware, longer implementation time.

For our controlled testing environment and high data throughput requirements, we have selected Wi-Fi. It offers sufficient bandwidth for LiDAR transmission and is straightforward to implement with existing hardware.

3.1.4.3 Antenna Design

The continuous rotation of the drone presents challenges for maintaining a stable wireless link, as the orientation of directional antennas would constantly change.

Omnidirectional antennas:
Pros: Maintains a consistent signal regardless of drone orientation, simple to implement.
Cons: Slightly lower gain compared to directional antennas, which may limit range in some scenarios.

Beamforming or antenna diversity:
Pros: Can improve reliability and compensate for orientation changes or interference.
Cons: Requires additional hardware and signal processing, increasing system complexity.

We have selected omnidirectional antennas for their simplicity and predictable performance. Given the drone's rotation, this option provides the most reliable solution without introducing unnecessary hardware complexity.

3.1.4.4 Latency and Transmission Protocol

Minimizing latency is critical for real-time LiDAR streaming.

TCP (Transmission Control Protocol):
Pros: Reliable delivery, ensures all packets arrive in order.
Cons: Retransmissions introduce latency, which can interfere with real-time streaming.

UDP (User Datagram Protocol):
Pros: Minimal latency, suitable for high-throughput streaming applications.
Cons: Packet loss may occur, requiring additional error-handling strategies.

We have chosen UDP combined with error detection mechanisms. This combination allows us to maintain low latency while ensuring data integrity.

3.1.4.5 Base Station Data Handling

The base station is responsible for receiving, processing, and visualizing LiDAR data.

- **Data pipeline:** We are using WebSockets to stream data to the web interface in real time, enabling responsive visualization.
- **Processing hardware:** The base station hardware must handle high-throughput data efficiently, reconstructing point clouds without introducing delays.
- **Synchronization:** Incoming LiDAR data is carefully synchronized with the web interface to ensure smooth and accurate visualization of the environment.

3.1.4.6 Error Handling

Wireless links are subject to interference and occasional packet loss. Two primary strategies can address these challenges:

CRC (Cyclic Redundancy Check):
Pros: Detects transmission errors efficiently, lightweight to implement.
Cons: Cannot recover lost data.

Limited packet retransmission:
Pros: Recovers lost data without introducing significant latency.
Cons: May slightly increase transmission time if errors occur frequently.

We have decided to implement CRC with limited packet retransmission. This approach provides a good balance between robustness and real-time performance, ensuring that LiDAR data remains accurate without excessive latency.

3.1.4.7 Synchronization

To reconstruct coherent and accurate point clouds, each LiDAR scan must be timestamped relative to the drone's rotation.

Rotation-synchronized timestamps:
Pros: Ensures alignment between LiDAR points and drone orientation, enabling precise 3D reconstruction.
Cons: Requires precise timing and coordination between drone and base station.

We are implementing **rotation-synchronized timestamps** to maintain data accuracy while minimizing system complexity.

3.1.4.8 Final Design Decisions

Data Compression	Point Cloud Compression
Wireless Link	Wi-Fi
Antenna	Omnidirectional
Protocol	UDP with CRC
Data Pipeline	Websocket for real-time web display
Error Handling	CRC with limited retransmission
Synchronization	Rotation-synchronized timestamps

By implementing this communication design, our drone can reliably transmit high-volume LiDAR data in real time while maintaining robustness, low latency, and predictable operation. Each design choice prioritizes system reliability and efficiency, ensuring that our drone can function effectively in a controlled test environment while remaining flexible for future upgrades or adjustments.

3.1.5 Motors

Motors are the central propulsion components of a drone, responsible for converting electrical energy into mechanical force to drive the propellers. The overall performance, efficiency, and stability of a drone depend greatly on the type of motor used, as it determines thrust generation, flight responsiveness, and power consumption. In modern drone technology, **Brushless DC (BLDC)** motors have become the industry standard, replacing older brushed designs due to their higher efficiency, torque output, and reliability. Among these, **outrunner BLDC** motors are particularly common because they produce high torque at relatively low rotational speeds, allowing direct propeller drive without additional gearing. The selection of the appropriate motor influences nearly every aspect of flight—from maneuverability and responsiveness to overall endurance—making it one of the most critical considerations in drone design.

3.1.5.1 Brushed DC Motors (BDC)

Early drones and toy-grade quadcopters often used **Brushed DC (BDC)** motors, one of the oldest and simplest electric motor types. These operate through mechanical commutation, using brushes and a commutator to switch current across the motor's windings. While their simplicity allows for easy control using only a DC voltage, their drawbacks—such as mechanical wear, low efficiency, and short lifespan—make them unsuitable for performance drones. The friction of the brushes limits rotational speed and leads to continuous power loss. Furthermore, their torque-to-weight ratio is poor, meaning they struggle to deliver the sustained lift and stability necessary for modern multirotors. Although they are cost-effective and simple to integrate, brushed motors are generally reserved for low-cost toys or prototypes rather than drones requiring reliable and efficient propulsion.

3.1.5.2 Brushless DC Motors (BLDC)

In contrast, **Brushless DC (BLDC)** motors eliminate mechanical commutation by electronically controlling current flow through the windings. This is achieved using an **Electronic Speed Controller (ESC)**, which precisely times the electrical switching to generate a rotating magnetic field. The absence of brushes eliminates frictional wear, significantly improving efficiency, longevity, and responsiveness. BLDC motors provide excellent torque at high RPM, making them ideal for driving propellers efficiently and with minimal vibration. They are lightweight, durable, and deliver excellent throttle response—qualities crucial for stable flight control. The **XING2 1404 3800KV** motor, for example, represents an optimized BLDC motor designed for 2–3-inch propellers. It balances high power output and efficiency while maintaining low weight, making it an excellent choice for sub-250 g drones. Although BLDC motors require separate ESCs, the advantages in performance, control precision, and energy efficiency far outweigh this minor complexity.

3.1.5.3 Outrunner Motors

Within BLDC designs, there are two main configurations: **outrunner** and **inrunner** motors. **Outrunner brushless motors** have their rotor (the outer shell containing permanent magnets) spinning around a stationary stator with fixed windings. This configuration produces high torque at lower RPMs, ideal for directly driving propellers without gear reduction. The larger effective torque arm and improved heat dissipation—due to the spinning outer can—result in higher efficiency and superior cooling. When properly matched to propellers, outrunners typically achieve efficiencies between 80–90%, providing the smooth throttle response and agility required for stable drone flight. For your 250 g drone, an outrunner like the **XING2 1404 3800KV** is ideal, offering the right balance between torque, weight, and responsiveness. Outrunners remain the gold standard in lightweight drone propulsion systems, used in everything from compact FPV drones to professional UAVs.

3.1.5.4 Inrunner Brushless Motors

By contrast, **inrunner brushless motors** feature a rotor located inside the stator, producing high rotational speeds but limited torque. This makes them more suitable for applications such as RC

cars, boats, or ducted fan jets, where high RPM is desirable. For propeller-driven drones, torque is more important than raw speed, as maintaining steady lift and stability requires slower but stronger rotational motion. Using an inrunner would necessitate a gear reduction system to match a propeller's optimal speed range, introducing additional weight, friction, and mechanical loss. Therefore, inrunners are generally inefficient for direct-drive drone propulsion and impractical for lightweight designs like yours.

3.1.5.5 Coreless Brushless Motors

A variation of the brushless motor is the **coreless brushless design**, which removes the iron core from the stator or rotor. This reduces weight and inertia, resulting in exceptionally fast acceleration, smooth motion, and minimal cogging torque. Coreless motors excel in applications where smooth low-speed control is more critical than torque output, such as camera gimbals or micro drones under 100 g. However, the thin coil structure has limited thermal dissipation and torque capacity, making them unsuitable for larger drones. For a 250 g platform using 2–3-inch propellers, a coreless motor would lack sufficient thrust and cooling capability, though it remains effective in extremely lightweight micro quadcopters.

3.1.5.6 Axial Flux (Pancake) Brushless Motors

Another emerging configuration is the **axial flux (pancake) motor**, which employs a flat, disc-shaped design where the magnetic flux travels parallel to the shaft. This geometry provides a large torque arm, yielding very high torque density and excellent cooling performance. Axial flux motors can reach efficiencies exceeding 90%, making them attractive for compact, high-performance propulsion systems. However, they are typically custom-built and costly, with limited availability in small, lightweight versions. Although promising for future drones, outrunner BLDCs currently offer similar performance at a fraction of the cost, making them the more practical choice for small-scale systems.

3.1.5.7 Gimbal Brushless Motors

Gimbal brushless motors represent a special subset of low-KV outrunner motors designed for camera stabilization systems rather than propulsion. These motors produce smooth, precise torque at low RPM but lack the speed and thermal handling required for continuous rotation. Their winding configuration prioritizes holding torque and precision rather than power output, meaning they would overheat or stall if used to drive propellers. While they share structural similarities with standard outrunners, gimbal motors are strictly unsuitable for drone flight applications.

3.1.5.8 Motor Design Variations

In addition to these main types, BLDC motors can also be categorized by **stator design**—either slotted or slotless. **Slotted BLDC motors** contain iron teeth around which the windings are wrapped, providing high torque density but introducing minor cogging effects. **Slotless motors** eliminate these slots, offering smoother rotation and slightly higher efficiency at low speeds,

though at the expense of torque output. For drone propulsion, slotted designs are preferred because they deliver higher torque per weight ratio and better heat dissipation, while slotless motors are more common in robotics or high-precision control systems.

Another distinction is between **sensorless** and **sensored** BLDC motors. **Sensorless motors** estimate rotor position using back electromotive force (back-EMF), simplifying wiring, reducing weight, and lowering cost. These are the standard for drones, where lightweight efficiency is essential. **Sensored motors**, which use Hall effect sensors for rotor position detection, are better suited for robotics or applications requiring fine low-speed control. Therefore, sensorless BLDC motors remain the optimal choice for high-speed, propeller-driven drones.

3.1.5.10 Other Motor Types

There are also other motor types occasionally considered for UAV use, though generally unsuitable. **Stepper motors**, which move in fixed angular increments, are highly precise but too heavy, slow, and inefficient for drone propulsion. **AC induction motors**, while robust and common in industrial settings, are too heavy and require complex AC control systems that add unnecessary bulk. Even advanced **axial flux motors**, though compact and efficient, remain largely impractical for small drones due to manufacturing complexity and cost. For drone applications—particularly lightweight systems—BLDC outrunners remain unmatched in terms of torque, efficiency, and weight balance.

3.1.5.11 Motor Size and Selection

Motor selection also depends heavily on stator size and configuration. Smaller motors like **1103 and 1104** (11 mm diameter, 3–4 mm height) weigh only about 3–5 g and are designed for micro drones with 2-inch propellers. While highly responsive, they lack the torque required for heavier drones and are best suited for indoor or sub-100 g builds. Slightly larger **1204 and 1206** motors (12 mm diameter, 4–6 mm height) offer more torque and can efficiently spin 2.5–3-inch props using 2S–3S batteries, making them transitional options for lightweight outdoor drones. For a 250 g design, these motors could work under minimal payload conditions but may struggle with efficiency during sustained hover.

1303 and 1306 motors, popular in racing-style micro drones, balance efficiency and thrust. The 1303 variant focuses on longer flight time and control precision, whereas the 1306 is tuned for higher thrust at the cost of efficiency. While effective, neither provides the ideal torque curve for endurance-focused builds. The **1404 motor**, however, represents the sweet spot for sub-250 g drones. With a 14 mm stator diameter and 4 mm height, it can drive 2.5–3.5-inch propellers on 3S LiPo batteries while maintaining an excellent balance of thrust, efficiency, and responsiveness. The **iFlight XING2 1404 3800KV** motor, when paired with a 3S 850 mAh 80C battery, can produce around 80–100 g of thrust per motor, enabling smooth, stable flight with minimal power loss.

Larger motors such as **1505**, **1804**, and **2004** increase torque significantly but at the cost of weight and power consumption. The 1505 motor, for example, offers powerful thrust for

3–4-inch props but weighs up to 15 g per unit, reducing flight efficiency. The 1804 and 2004 series are optimized for 4–5-inch props in heavier drones (300–400 g or more) and are considered excessive for lightweight designs, as they require larger batteries and result in shorter flight times. For a 250 g platform emphasizing endurance, these would be unnecessarily powerful and inefficient.

3.1.5.16 Summary Table

Motor Type	Torque	Efficiency	Weight	Drone Suitability	Notes
Brushed DC	Low	Low	Medium	❌	Easy to use but inefficient
Brushless DC (Outrunner)	High	High	Low	✅✅	Standard for drones
Coreless	Low	Medium	Very Low	⚠️	Good for <100 g micro drones
Inrunner	Low	Medium	Medium	❌	Too high RPM, low torque
Stepper	High (static)	Low	Heavy	❌	Poor for continuous rotation
AC Induction	Medium	Medium	Heavy	❌	Industrial, not drone-suitable
Axial Flux	Very High	Very High	Medium	⚠️	Great concept, not practical yet

3.1.6 LCD

Liquid Crystal Displays (LCDs) are widely used in embedded and control systems to provide a visual interface for real-time data display and user interaction. They operate by controlling the orientation of liquid crystal molecules through electric fields, modulating light transmission to produce characters or images. LCDs are lightweight, compact, and energy efficient, making them ideal for portable and low-power applications.

These displays are generally categorized into two main types: character displays, which show alphanumeric information using fixed segments, and graphical displays, which offer pixel-level control for rendering text and images. More advanced versions, such as TFT color LCDs, expand these capabilities with vibrant visual output and higher refresh rates. Their broad compatibility with microcontrollers and communication interfaces has made LCDs a standard component in modern electronic systems where visual feedback is required.

3.1.6.1 Character LCDs

Character LCDs are among the simplest and most common display modules used in embedded designs. They are based on a controller standard such as the Hitachi HD44780, typically configured in formats such as 16x2, 20x2, or 20x4 characters. These modules are designed to display letters, numbers, and basic symbols arranged in a grid layout.

Character LCDs can be interfaced using either parallel communication or via an I²C expander module, which reduces the number of microcontroller pins required. They consume very little power and have minimal software overhead, making them ideal for displaying static or low-refresh-rate information. While they are limited in terms of graphical capability, their reliability and ease of integration have made them a staple in instrumentation, measurement, and diagnostic devices.

3.1.6.2 Graphical LCDs

Graphical LCDs (GLCDs) provide greater versatility by allowing pixel-addressable control, enabling the display of both text and images. Common resolutions include 128x64, 192x64, and 240x128 pixels, often driven by controllers such as the ST7920, KS0108, or RA6963. These displays support a wide range of communication protocols including parallel, SPI, and I2C, depending on the application.

GLCDs are useful in systems that require the presentation of charts, icons, or variable data in a structured layout. They provide a balance between simplicity and flexibility, offering monochrome graphical output with reasonable refresh rates and moderate memory requirements. Their broad adoption in industrial equipment, laboratory devices, and small-scale embedded systems demonstrates their effectiveness for intermediate-level visual applications.

3.1.6.3 TFT LCDs (Color Displays)

Thin Film Transistor (TFT) LCDs represent an advanced form of liquid crystal display technology that enables high-resolution, full-color visualization. These displays use an active

matrix of transistors to control each pixel individually, providing high contrast ratios, faster response times, and smooth color gradients.

TFT displays are commonly available in resolutions ranging from 160x128 up to 480x320 pixels, and are typically driven by controllers such as the ILI9341, ST7735, or SSD1963. They interface with microcontrollers using SPI, parallel, or RGB communication buses. Although TFT modules consume more power compared to character or monochrome graphic LCDs, they allow for complex graphical interfaces with multiple layers of information and color coding. This makes them well suited for control panels, handheld instruments, and visualization systems that require clarity and dynamic feedback.

3.1.6.4 LCD Interface and Communication

LCD modules communicate with host controllers through standard serial or parallel protocols. I2C and SPI are preferred for compact designs because they reduce wiring complexity while maintaining reliable data transmission. Parallel interfaces offer higher data throughput, which benefits large displays or applications requiring fast updates.

Most LCDs operate within a voltage range of 3.3 V to 5 V and are compatible with common microcontroller logic levels. Data is transmitted to the display as commands and pixel information, handled by an onboard driver integrated circuit that interprets the signals and controls the liquid crystal matrix accordingly. Display refresh rates and update frequencies vary depending on resolution and interface type, but typically range from tens to hundreds of hertz.

3.1.7 Propellors

The number of blades significantly affects thrust, efficiency, and overall motor performance. Two-blade propellers tend to be more efficient because they incur less aerodynamic drag and reduced interference between blades. This makes them ideal for maximizing flight time, which is especially important in a tricopter where each motor contributes a larger portion of total thrust.

Three-blade propellers, on the other hand, provide smoother flight and better control, particularly during yaw adjustments and under varying flight conditions. This added stability is beneficial in a tricopter configuration, which has inherently less symmetry than a quadcopter. The tradeoff is a higher power draw and increased load on the motors.

For this design, a **combination of two-blade and three-blade propellers** was selected. Two-blade propellers are used to improve efficiency and reduce power consumption, while a three-blade propeller is incorporated to enhance stability and control authority.

3.1.7.1 Number of Blades

The number of blades significantly affects thrust, efficiency, and overall motor performance. Two-blade propellers tend to be more efficient because they incur less aerodynamic drag and reduced interference between blades. However, they may produce more vibration and slightly less stability in hover conditions.

Three-blade propellers, on the other hand, provide smoother flight and better control, especially under variable wind or during agile maneuvers. The tradeoff is a slightly higher power draw and reduced efficiency. For sub-250 g drones, a two- or three-blade configuration offers the best balance between thrust and efficiency [31][34].

For instance, the Gemfan Hurricane 2009-3 (2-inch tri-blade) propeller is popular for micro drones due to its stability and low weight (0.4 g) [32]. Meanwhile, the Emax Avan Blur 2×1.9×3 is a lightweight, efficient tri-blade prop designed for smooth performance and optimized for small motors [33]. These examples illustrate how blade count can be tailored to achieve specific design goals—efficiency for longer flight or stability for tighter control.

3.1.7.2 Propeller Shape

The propeller's geometry, especially diameter, pitch, and tip shape helps determine how efficiently it converts motor torque into thrust.

For drones under 250 g, 2-inch to 3-inch propellers are ideal. Smaller propellers (< 2″) often fail to generate sufficient lift, while larger ones (> 3″) can overload the motors. A moderate pitch (around 2.5–3.0 inches) provides a good balance between thrust and endurance.

For example, the HQProp T2×2.5×3 has a 2-inch diameter and a 2.5-inch pitch, delivering higher thrust with controlled load [4]. In contrast, the Emax Avan Blur features a lower pitch (1.9″), emphasizing smooth and stable hovering [33]. The blade tip design, such as the slightly swept tips found on newer Gemfan series which helps reduce drag and noise, increasing efficiency [32].

3.1.7.3 Propeller Material

Material choice affects the tradeoff between weight, rigidity, and impact durability. For a micro drone, the goal is to minimize mass while maintaining stability and strength.

Polycarbonate (PC) propellers are common for lightweight builds because they are durable, flexible, and inexpensive. The Emax Avan Blur uses a high-strength PC blend to achieve a 1.1 g weight while remaining rigid enough for consistent thrust [33]. The Gemfan Hurricane 2009-3 is also made from polycarbonate, weighing only 0.4 g, which minimizes rotational inertia and improves throttle response [32].

For higher-end applications, nylon composites and carbon fiber propellers offer greater stiffness and performance at high RPMs, but they add mass and can cause higher vibration or motor strain in crashes [35][36].

3.1.7.4 Final Recommendation

Considering the drone's 250-gram weight limit, efficiency goals, and motor capabilities, the optimal configuration would be:

- **Propeller size:** 3″–3.5″ (around 2″ is ideal for small brushless motors)
- **Blade count:** 3 blades for maximum efficiency; 3.5 blades for improved stability and control
- **Pitch:** 2.5–3″ (e.g., 3×2.5×3 or 3.5×3×3)
- **Material:** Polycarbonate or nylon composite
- **Examples:**
  • *Gemfan Hurricane 2009-3* — lightweight, stable, polycarbonate [32]
  • *Emax Avan Blur 2×1.9×3* — smooth hover performance, durable PC [33]
  • *HQProp T2×2.5×3* — moderate pitch, balanced thrust [34]

This configuration ensures a good balance between thrust, efficiency, and stability while keeping the total weight under the 250 g restriction.

Prop diameter	Typical pitch range (common)	RPM behavior	Relative efficiency (hover g/W)	Thrust potential (small motors)	Rotational inertia & response	Motor KV range	Best uses / notes
2.0"	1.6 – 2.5	Very High RPM	**Low**	Low	Very low inertia → very responsive	4500–9000 KV	Tiny toothpicks, indoor micros, racing — poor endurance.
2.5"	1.4 – 3.0	High RPM	Low–Medium	Medium (for micro class)	Low inertia, quick response	3000–6000 KV	Balanced micro builds — good compromise between control & efficiency.

3.0"	1.8 – 3.5	Moderate RPM	**Medium–High**	Good	Moderate inertia, still fairly responsive	2500–4200 KV	Ideal for efficient sub-250 g drones — often the best endurance choice.
3.5"	2.0 – 4.0	Lower RPM	High	Better than 3" for same power if motor torque supports it	Slightly higher inertia, smoother	2000–3500 KV	When frame allows, gives extra g/W and payload margin for micro long-range builds.
4.0"	2.0 – 4.5	Low RPM	High (good for larger motors)	High	Noticeable inertia; slower response	1500–3000 KV	Small frames with 1806+ motors; common on small cinewhoops and efficient micro quads.
5.0"	2.5 – 5.0	Low RPM	High for larger stators	High	High inertia; not for quick agility	1200–2200 KV	Standard in 5" FPV & efficient cruising with bigger motors.

6.0"	3.0 – 6.0	Very Low RPM	Very high with correct motor	Very high	High inertia; smooth but slow response	800–1800 KV	Long-range builds and heavy-lift (larger frames & motors required).

3.1.8 Frame Materials

The frame material plays a crucial role in determining the strength, durability, and overall performance of a drone. Since modern fabrication often involves additive manufacturing, 3D-printed materials are becoming increasingly popular for lightweight drones. These materials allow for complex, customizable geometries while maintaining low production cost and weight. Both being essential factors for a drone under 250 g.

Different filament types offer unique mechanical and thermal properties that affect how well a drone can handle impacts, vibration, and heat from its motors or electronics. The following sections discuss several common 3D-printed materials used in drone design and their general characteristics.

3.1.8.1 Carbon-Fiber-Reinforced Nylon (PA-CF)

Carbon-fiber-reinforced nylon is one of the strongest and most durable 3D-printing materials used for drone construction. It combines the flexibility and toughness of nylon with the stiffness and rigidity of chopped carbon fibers, creating a high-strength composite with excellent resistance to deformation. This material can withstand significant mechanical stress, making it suitable for load-bearing parts such as arms, base plates, or motor mounts.

PA-CF also offers high temperature resistance and good vibration damping, which helps reduce noise in sensor readings and improves stability during flight. However, it requires careful printing since the filament must be kept dry, and a hardened steel nozzle is needed to handle the abrasive carbon fibers. Overall, carbon-fiber-reinforced nylon provides the best balance of strength, stiffness, and weight for high-performance drone frames.

3.1.8.2 PETG (Polyethylene Terephthalate Glycol)

PETG is a durable and versatile filament that offers a good balance between flexibility and rigidity. It is less brittle than PLA and can absorb impacts more effectively, making it suitable for drone components that may experience mechanical stress or vibration. PETG also has better temperature resistance than PLA, allowing it to maintain structural integrity under heat from motors or electronics.

This material provides excellent layer adhesion, making prints less prone to delamination during crashes or high-frequency vibration. Although PETG is slightly heavier and more flexible than PLA, it remains easy to print and is widely used for parts such as mounting brackets, protective covers, and internal supports.

3.1.8.3 PLA (Polylactic Acid)

PLA is one of the most common and beginner-friendly 3D-printing materials. It offers high rigidity and dimensional accuracy, which makes it excellent for prototyping and testing drone parts. However, it is relatively brittle and can crack or deform when exposed to heat or mechanical stress. PLA softens around 55 °C, so it may warp in hot environments or when mounted near heat-producing components.

Because of its ease of use and low cost, PLA is typically used for early design iterations or non-structural components where high strength is not required. While not ideal for final flight frames, it remains valuable for rapid prototyping and proof-of-concept testing during drone development.

3.1.8.4 TPU (Thermoplastic Polyurethane)

TPU is a flexible, rubber-like filament known for its high impact resistance and elasticity. It is often used for protective or vibration-damping parts on drones, such as propeller guards, landing pads, camera mounts, or bumpers. TPU's flexibility allows it to absorb shocks and vibrations effectively, which helps protect delicate electronics and sensors from mechanical damage.

Although TPU requires slower print speeds and careful temperature control, it produces lightweight and resilient components that complement rigid frame materials like carbon-fiber nylon or PETG. Its ability to deform under impact and return to shape makes it ideal for absorbing energy during crashes or rough landings.

3.1.9 Voltage Regulation

3.1.9.1 Overview

Voltage regulation is the process of maintaining a constant output voltage level regardless of variations in input voltage or load current. It plays a crucial role in all electronic systems by ensuring that sensitive components receive stable and reliable power for consistent operation. Without proper regulation, circuits may experience fluctuations that lead to instability, signal distortion, or component damage.

Regulators are typically categorized into two primary types: linear regulators and switching regulators. Linear regulators operate by dissipating excess energy as heat through a pass transistor, providing clean, low-noise power suitable for analog and low-power circuits. Switching regulators, in contrast, convert energy efficiently using high-frequency switching and energy storage elements, making them ideal for higher-power applications. Each method presents

distinct advantages and limitations in terms of efficiency, complexity, and electromagnetic performance.

Modern regulation circuits often integrate advanced control methods, protection mechanisms, and compact designs to meet the increasing power density demands of embedded and portable systems. These technologies are widely employed in microcontrollers, communication modules, sensor networks, and aerospace-grade electronics where precision voltage control is essential for stable operation.

3.1.9.2 Linear Voltage Regulators

Linear regulators are among the simplest and most widely used voltage control devices. They maintain a stable output by adjusting a pass transistor's resistance to drop the excess voltage between input and output. The output is compared to an internal reference, and feedback continuously corrects deviations to ensure stability.

Their primary advantage lies in producing very low output noise and excellent transient response. Because linear regulators operate without high-frequency switching, they avoid generating electromagnetic interference (EMI), making them ideal for analog, RF, or precision sensor systems. However, their efficiency decreases as the difference between input and output voltage grows, since the voltage drop across the pass element directly translates into power loss as heat.

Two major subtypes of linear regulators exist: standard linear regulators, such as the LM7805 and LM317, and low-dropout regulators (LDOs), such as the LD1117. LDOs improve upon conventional designs by minimizing the voltage difference required between input and output, thus enhancing efficiency and making them suitable for battery-powered applications.

Linear regulation remains a preferred technology for systems requiring clean, stable voltage outputs with minimal ripple. Although less efficient than switching methods, linear regulators are valued for their simplicity, stability, and low noise characteristics.

3.1.9.3 Switching Voltage Regulators

Switching regulators, also known as switch-mode power supplies (SMPS), achieve voltage conversion through high-frequency energy transfer using inductors, capacitors, and semiconductor switches. Instead of dissipating excess energy as heat, switching regulators store and release energy in controlled pulses, resulting in much higher efficiencies compared to linear regulators.

Common topologies include buck (step-down), boost (step-up), and buck-boost converters. Buck converters reduce voltage to power lower-voltage devices, boost converters increase voltage when supply levels drop below required thresholds, and buck-boost converters can both increase and decrease output depending on input conditions.

Switching regulators typically operate at frequencies between 20 kHz and several MHz, depending on the design. Higher switching frequencies allow smaller inductors and capacitors

but increase switching losses and potential EMI. Despite these challenges, switching regulators are preferred for high-efficiency designs and systems that must deliver substantial power within compact form factors.

The technology has evolved to include synchronous designs—where diodes are replaced with low-resistance MOSFETs to further increase efficiency—and advanced control techniques such as current-mode and hysteretic regulation for improved load response. These characteristics make switching regulation an essential technology in nearly all modern embedded and power-electronic systems.

3.1.9.4 Common Voltage Regulator Devices

LM7805 (Fixed 5 V Linear Regulator)

The LM7805 is a widely used fixed-output linear regulator that provides a stable 5-volt output from higher input voltages, typically up to 35 volts. Known for its simplicity and robustness, it is one of the most common voltage regulators in both educational and industrial circuits. The LM7805 features built-in thermal shutdown and short-circuit protection, ensuring safe operation under varying load conditions. It produces minimal output ripple, making it ideal for low-noise applications. However, its efficiency is limited by its voltage dropout of approximately 2 volts, which results in substantial heat dissipation at higher currents. The LM7805 remains an excellent choice for circuits requiring reliable, noise-free power where efficiency is not the primary concern.

LM317 (Adjustable Linear Regulator)

The LM317 is an adjustable three-terminal linear regulator that provides an output voltage range between 1.25 and 37 volts, set by two external resistors. It is capable of delivering up to 1.5 amperes of current with built-in overload and thermal protection. The flexibility of the LM317 makes it suitable for custom voltage applications, laboratory setups, and circuit prototyping. While its performance is reliable and stable, it suffers from the same efficiency limitations as other linear regulators, as excess voltage is dissipated as heat. The LM317 remains a preferred option for variable supply designs and regulated laboratory circuits.

LD1117 (Low-Dropout Linear Regulator)

The LD1117 is a low-dropout (LDO) linear regulator that improves upon traditional linear designs by allowing regulation with a smaller difference between input and output voltages—typically around 1.1 volts. It provides clean, low-ripple output and is commonly available in fixed 3.3-volt and 5-volt versions, as well as an adjustable configuration. The LD1117 is widely used in digital and analog circuits to power microcontrollers, sensors, and communication modules. Its compact packaging and stability make it ideal for integrated systems requiring efficient, low-noise voltage conversion at moderate current levels.

LM2576 (Step-Down Switching Regulator)

The LM2576 is a monolithic step-down switching regulator capable of delivering up to 3 amperes of output current with efficiencies around 85%. It simplifies design implementation by integrating an internal power transistor, oscillator, and control circuitry. The regulator operates at a fixed frequency of 52 kHz and supports fixed output versions (3.3 V, 5 V, 12 V) as well as adjustable models. Because it efficiently converts higher input voltages to stable lower outputs, the LM2576 is frequently used in embedded systems, robotics, and other battery-powered electronics requiring moderate power delivery and thermal efficiency.

TPS5430 (Synchronous Buck Converter)

The TPS5430 is a high-efficiency synchronous buck converter that integrates both high-side and low-side MOSFETs, allowing it to achieve conversion efficiencies of up to 95%. Operating at a switching frequency of 500 kHz, it allows the use of smaller inductors and capacitors, enabling compact circuit layouts. The device supports input voltages up to 36 volts and can deliver continuous currents of up to 3 amperes. The TPS5430 includes protection features such as soft-start, overcurrent, and thermal shutdown, making it suitable for modern, compact power supply designs. Its synchronous design reduces conduction losses and enhances efficiency compared to traditional diode-based switching converters.

3.1.9.5 Hybrid Regulation and Integration Techniques

In many modern systems, designers employ hybrid power architectures that combine the efficiency of switching regulators with the noise suppression of linear regulators. A common configuration involves using a buck converter to step down a high input voltage efficiently, followed by an LDO regulator to provide low-noise power for sensitive analog or digital circuits. This cascaded approach balances performance, efficiency, and signal integrity.

Advancements in semiconductor technology have also led to the development of power management integrated circuits (PMICs) that consolidate multiple regulation stages into a single chip. These ICs often include both buck and LDO regulators, protection circuits, and enable controls, simplifying board design while improving overall efficiency and reliability. Such integrated solutions have become standard in compact embedded systems, portable electronics, and aerospace technologies that demand lightweight yet stable power delivery.

3.1.10 Batteries

Batteries serve as the primary energy source for drones, supplying electrical power to the motors, flight controller, and onboard electronics. The choice of battery chemistry directly impacts flight time, performance, and safety. Modern drones require high discharge rates to power brushless motors efficiently, along with lightweight and rechargeable energy storage. While early drone prototypes relied on NiMH or primary cells, advancements in lithium-based technologies. Particularly Lithium Polymer (LiPo) and Lithium-Ion (Li-ion) which have been the industry standard for both recreational and professional UAVs.

3.1.10.1 Primary (Non-Rechargeable) Batteries

Primary batteries, such as alkaline and lithium primary cells, are designed for single use and cannot be recharged once depleted. They store chemical energy that is converted directly to electrical energy through irreversible reactions. Their typical voltages range from 1.5 V (alkaline) to 3 V (lithium cells), and they have high energy density relative to their cost and size. However, they have low current output and are not built to handle high discharge rates required by drone motors. Additionally, their single-use nature makes them unsustainable and impractical for devices demanding frequent re-energizing. Because drones require batteries that can discharge rapidly and recharge efficiently, primary batteries are almost never used in drones except for very small fixed-wing prototypes or remote-control transmitters.
Pros: Inexpensive, long shelf life, readily available.
Cons: Non-rechargeable, low current capacity, environmentally wasteful.

3.1.10.2 Secondary (Rechargeable) Batteries

Secondary batteries are the foundation of drone power systems. They can be recharged hundreds of times and are capable of delivering large bursts of current to drive motors and onboard electronics. Their internal chemistry allows reversible electrochemical reactions, which makes them cost-effective over the long term. Within this class, lithium-based batteries dominate the drone market because they provide a high energy-to-weight ratio—an essential factor for flight efficiency. The main subtypes used in drones are Lithium-ion (Li-ion), Lithium Polymer (LiPo), and Lithium Iron Phosphate ($LiFePO_4$). Older chemistries like Nickel-Metal Hydride (NiMH) appear only in educational kits or older UAV systems.
Pros: Rechargeable, powerful, adaptable in shape and configuration.
Cons: Require careful charging, can be hazardous if overcharged or punctured.

3.1.10.3 Lithium-Ion (Li-ion) Batteries

Lithium-ion batteries are the most common type used in commercial and photography drones such as DJI's Mavic or Phantom series. Each cell typically has a nominal voltage of 3.6 – 3.7 V and a fully charged voltage of 4.2 V. Li-ion batteries use cylindrical or prismatic cells containing a liquid electrolyte and a protective metal casing, which gives them durability and long cycle life. Their energy density is among the highest (150–250 Wh/kg), enabling long flight durations, but their discharge rate (10–20 C) is moderate, making them less suitable for racing or acrobatic drones.
Pros: High energy density, stable voltage, long lifespan (500–1000 cycles), low self-discharge.
Cons: Heavier and bulkier than LiPo, lower instantaneous current output, rigid construction limits design flexibility.
Used for: Aerial photography drones, mapping UAVs, and endurance-focused fixed-wing drones where efficiency outweighs agility.
Example: DJI Mavic 3's 4S 5000 mAh Li-ion battery offering up to 46 minutes of flight time.

3.1.10.4 Lithium Polymer (LiPo) Batteries

Lithium Polymer batteries are the standard power source for racing, freestyle, and custom-built drones. Instead of a metal canister, they use a soft polymer pouch that contains a gel-like electrolyte and layered electrodes. This makes them lightweight and shape-flexible, allowing

manufacturers to fit them into various drone frames. LiPos have nominal voltages around 3.7 V per cell (4.2 V fully charged) and C-ratings from 20 C up to 150 C, meaning they can discharge extremely high currents to handle aggressive maneuvers and instant throttle changes. However, their chemistry is volatile and requires careful charging and storage at safe voltages (around 3.8 V per cell).

Pros: Excellent power-to-weight ratio, high discharge capability, lightweight design, customizable form factors.

Cons: Sensitive to overcharge and deep discharge, short lifespan (200–300 cycles), prone to swelling and fire risk if misused.

Used for: FPV racing drones, freestyle quadcopters, micro drones, and any UAV requiring rapid bursts of power.

Example: Tattu R-Line 1300 mAh 4S 100C LiPo—common in 5-inch FPV drones due to its balance of weight (~150 g) and punch.

3.1.10.5 Lithium Iron Phosphate ($LiFePO_4$ or LFP) Batteries

$LiFePO_4$ batteries are built for safety and longevity rather than maximum performance. Each cell has a nominal voltage of 3.2 V, which is lower than standard Li-ion or LiPo cells, but they can sustain 2000+ charge cycles and have exceptional thermal and chemical stability. They can handle moderate discharge rates (10–30 C) but at the cost of lower energy density (90–120 Wh/kg). The voltage profile of $LiFePO_4$ cells remains flat during discharge, providing consistent power output. Because of their robustness, they are used in industrial drones, heavy-lift UAVs, and training systems where battery safety and lifespan are critical.

Pros: Extremely stable chemistry, non-flammable, very long life cycle, tolerant of over-discharge.

Cons: Lower energy density (heavier for same capacity), lower voltage requires more cells in series to achieve equivalent output.

Used for: Agricultural drones, surveillance UAVs, and beginner-training multirotors prioritizing safety over speed.

Example: A123 26650 $LiFePO_4$ 4S 2300 mAh pack used in research or teaching UAVs.

3.1.10.6 Nickel-Metal Hydride (NiMH) Batteries

NiMH batteries are a legacy power option that predates the lithium era. Each cell delivers around 1.2 V, with energy densities between 60 and 120 Wh/kg—roughly half of what Li-ion offers. NiMH uses a hydrogen-absorbing alloy for the negative electrode and nickel oxyhydroxide for the positive. While safer and more environmentally friendly than older nickel-cadmium (NiCd) cells, they are heavy, suffer from self-discharge, and cannot supply high continuous currents. In modern drones, NiMH packs are sometimes used for ground control units, transmitters, or small fixed-wing trainers but are unsuited for multirotor propulsion.

Pros: Inexpensive, durable, safer than lithium types, recyclable.

Cons: Heavy for their capacity, poor power delivery, high self-discharge rate, voltage sag under load.

Used for: Educational drones, toy-grade UAVs, RC transmitters, or non-flight electronics.

3.1.11 Development Environment: Languages and Repositories

3.1.11.1 C / C++

C is a very straightforward programming language that is known for its simplicity. C/C++ is primarily used for low-level programming and does not include many object-oriented features. C is widely used in hardware development and is incredibly popular for embedded systems and semiconductor programming. C programming is so widely used that it has many libraries that allow developers to solve problems without having to write all the functions from scratch. C is very well suited for hardware interaction since it provides direct access to registers, memory, and flags, which enable precise control over attached hardware. Programs that are written with C/C++ must be compiled, which translates the high-level code into assembly and then into machine code that the computer can execute. C/C++ lacks the modern programming features such as object oriented design since it is older. Despite this, it still remains an excellent choice for low-level coding and embedded software development and projects involving microcontrollers.

3.1.11.2 Java

Java is a very versatile programming language that emphasizes object-oriented and class-based programming. It is able to serve as both a programming language and a software platform, making it highly portable and easy to run on different devices. When Java code is compiled, it is turned into an intermediate form called bytecode. Bytecode is then executed on a machine through the Java Development Kit (JDK). Java is designed around object-oriented programming (OOP), which simplifies coding by organizing programs into classes and objects, unlike C. The key OOP concepts that are supported by Java include encapsulation, abstraction, inheritance, and polymorphism. These features help provide several benefits for team-based projects, which will help improve code organization, reusability, collaboration, and scalability. Overall, Java is an excellent choice for projects due to its integrated platform support and robust object-oriented design.

3.1.11.3 Python

Python is a high-level, interpreted programming language that is known for its simplicity and readability. It supports multiple programming paradigms, including object-oriented, procedural, and functional programming. This makes it highly flexible for a wide range of applications. Python code is executed directly by the Python interpreter, which eliminates the need for compilation and allows for rapid development and testing. The object-oriented features that python offers, such as classes, objects, inheritance, and polymorphism helps organize code efficiently and supports reuseability, scalability, and collaboration in team projects. Its extensive standard library and range of third-party modules makes it easy for implementation of complex functions without writing code from scratch. On top of that, the readability and clean syntax makes it easy for maintenance, debugging, and expanding projects for teams. Overall, Python is an excellent choice for prototyping and full-scale development.

3.1.11.4 GitHub

GitHub is a web-based platform that helps with version control and collaborative software development. It is built around Git, which is a distributed version control system that allows for multiple developers to work on the same codebase. It will help track changes, manage revisions, and maintain project history. GitHub enables features such as branching, merging, pull request, and issue tracking, which help teams coordinate development, review code, and manage tasks efficiently. GitHub allows for the developers to share their code either publicly or privately, contribute to open-source projects, and integrate with various tools for continuous integration. It has collaborative features to make it easier to coordinate team efforts, maintain organized code, and ensure consistent project documentation. Overall, GitHub is an essential tool for modern software development and is efficient project management for both small teams and large organizations.

3.1.11.5 Betaflight

Betaflight is an open-source flight control firmware designed for multirotor drones and other unmanned aerial systems. It provides the software environment that governs flight behavior, stabilization, and motor output through advanced PID-based control algorithms. Betaflight is widely used across both research and hobbyist applications due to its extensive configurability, community support, and compatibility with a wide range of flight controllers.

The firmware is developed primarily in C and C++, making it ideal for real-time embedded systems that require high-speed control loops. Betaflight includes features such as gyro filtering, DShot and RPM telemetry support, and configurable PID tuning, all of which help optimize flight stability and responsiveness. It also includes a graphical configurator tool that allows developers to modify parameters, test sensor inputs, and tune flight performance without altering source code directly.

For this project, Betaflight serves as the primary development environment for configuring the drone's flight controller, monitoring telemetry, and fine-tuning control parameters to maintain stable operation during continuous 60 RPM rotation. Its open-source nature and modular firmware structure make it an excellent platform for experimentation, testing, and iterative improvement during development.

3.1.12 Circuits

3.1.12.1 Overview

Electronic circuits form the backbone of all modern electrical systems. They serve as the medium through which electrical energy is directed, controlled, and transformed into useful operations. A circuit is composed of interconnected components—such as resistors, capacitors, inductors, diodes, and transistors—that work together to manipulate voltage and current in predictable ways. Circuits can generally be classified as analog, digital, or mixed-signal, depending on the nature of the signals they process. Analog circuits deal with continuously varying signals, while digital circuits handle discrete binary logic levels. Most modern systems

combine both types to achieve complex functionality, where analog subsystems manage signal acquisition and conditioning, and digital circuits perform computation and control. Supporting circuits such as power regulation, protection, and signal interface networks ensure that these operations remain stable, accurate, and efficient. The continuous evolution of semiconductor and integrated circuit technology has made it possible to perform increasingly complex tasks within smaller, more power-efficient designs.

3.1.12.2 Analog Circuits

Analog circuits operate using continuous electrical signals that represent variations in voltage or current over time. They are essential in systems that process or interpret real-world phenomena such as sound, temperature, motion, or light. The design and behavior of analog circuits are rooted in linear systems theory and frequency-domain analysis, which govern their amplification, filtering, and oscillation characteristics. Amplifier circuits are a central component of analog electronics, using transistors or operational amplifiers to increase the amplitude of input signals without significantly altering their form. They are employed in audio equipment, sensor conditioning, and instrumentation systems where signal strength must be boosted for further processing.

Filtering is another vital function in analog circuit design. Filter circuits made of resistors, capacitors, and inductors are designed to control the frequency content of signals by allowing certain frequency ranges to pass while attenuating others. Low-pass filters are used to remove high-frequency noise, high-pass filters block low-frequency drift, and band-pass filters isolate specific frequency bands of interest. Oscillator circuits, by contrast, generate periodic waveforms such as sine, square, or triangular signals and are widely used for clock generation, communication carriers, and waveform synthesis. Collectively, these analog building blocks allow systems to acquire, shape, and stabilize signals before digital processing occurs.

3.1.12.3 Digital Circuits

Digital circuits form the foundation of modern computation, communication, and control systems. They operate by processing discrete voltage levels that represent binary logic states, typically “1” for a high level and “0” for a low level. At the heart of all digital designs are logic gates—devices that perform basic Boolean operations such as AND, OR, NOT, and XOR. By interconnecting these gates, engineers construct complex functional units capable of arithmetic, memory storage, and decision-making.

Digital circuits can be categorized into combinational and sequential systems. Combinational circuits determine outputs purely based on current input conditions, while sequential circuits include memory elements that store previous states, allowing them to perform time-dependent operations. Components such as flip-flops, latches, counters, and shift registers make sequential logic possible and enable more sophisticated timing and control mechanisms. Modern digital circuits are realized within microcontrollers, microprocessors, and programmable devices such as field-programmable gate arrays (FPGAs). The advancement of very-large-scale integration (VLSI) technology allows billions of transistors to coexist on a single chip, significantly enhancing computational power and efficiency while minimizing physical size and energy

consumption. These digital subsystems handle everything from basic data processing to advanced automation and embedded control.

3.1.12.4 Power and Regulation Circuits

Power and regulation circuits are responsible for distributing and controlling electrical energy throughout an electronic system. Their primary function is to maintain consistent voltage and current levels to ensure that every component operates within its specified range. A basic power supply circuit typically converts alternating current (AC) from an external source into direct current (DC) using a rectifier composed of diodes. Capacitors are often added to smooth out the pulsating DC waveform that follows rectification, reducing ripple voltage and providing a more stable supply.

Voltage regulation is achieved using either linear or switching regulator circuits. Linear regulators maintain a fixed output voltage by dissipating excess energy as heat through a pass element, while switching regulators achieve higher efficiency by rapidly toggling a transistor to transfer energy through inductors and capacitors. The choice between the two depends on the desired balance between efficiency, size, and noise performance. Many systems also incorporate DC–DC converters to adapt one voltage level to another, enabling different subsystems to operate independently from a shared power source. Proper regulation and filtering prevent unwanted voltage fluctuations that could disrupt the performance of sensitive electronics. These circuits form the electrical foundation of any system, ensuring power integrity and long-term operational stability.

3.1.12.5 Signal Conditioning and Interface Circuits

Signal conditioning circuits are used to prepare raw input data from sensors or other sources for accurate measurement or processing. These circuits may amplify weak signals, remove noise, shift voltage levels, or convert signal formats. Instrumentation amplifiers are a key element of signal conditioning, providing precise amplification with high input impedance and excellent common-mode rejection. They are commonly used to measure small differential signals from sensors such as strain gauges, thermocouples, and accelerometers.

Another important aspect of signal conditioning involves adapting voltage levels between subsystems that operate at different logic thresholds. Level-shifting circuits and voltage dividers allow safe interfacing between components that use different voltage standards. In mixed-signal systems, analog-to-digital converters (ADCs) and digital-to-analog converters (DACs) serve as bridges between continuous and discrete domains, enabling sensors and controllers to communicate effectively. Interface circuits extend beyond conversion by incorporating communication transceivers, such as those used for UART, I²C, SPI, and CAN protocols. These hardware interfaces facilitate reliable data exchange between integrated circuits, sensors, and actuators, forming the backbone of coordinated electronic communication.

3.1.12.6 Protection Circuits

Protection circuits are designed to safeguard electronic components from electrical faults and environmental hazards. They protect against overvoltage, overcurrent, electrostatic discharge

(ESD), and incorrect polarity connections. One of the simplest and most common protection mechanisms is the diode, which can be placed in series or parallel with the load to prevent reverse current flow. Metal-oxide varistors (MOVs) and transient voltage suppression (TVS) diodes are used to absorb high-energy transients caused by switching operations or lightning strikes, clamping voltage spikes to safe levels.

Overcurrent protection is often implemented with fuses or resettable polymer fuses (PTCs), which interrupt current flow when it exceeds a predetermined limit. More advanced systems employ electronic current limiting circuits that dynamically restrict output current using feedback control. Protection circuits also help maintain long-term reliability by preventing cumulative stress on sensitive components. These measures are essential in modern electronics, where dense circuit integration increases the vulnerability to electrical and thermal damage. By incorporating multiple layers of protection, engineers enhance system durability and operational safety.

3.1.12.7 Control and Drive Circuits

Control and drive circuits are responsible for managing and delivering power to electromechanical devices such as motors, relays, and actuators. These circuits often translate low-power control signals into higher-power outputs that can drive physical motion or perform mechanical work. Transistors, MOSFETs, and integrated driver ICs are the key active components in such systems. In many designs, transistors are arranged in configurations like H-bridges to enable bidirectional current flow, which allows a motor to rotate in either direction.

Pulse-width modulation (PWM) is a common technique used in control circuits to adjust the effective voltage or current delivered to a load by varying the duty cycle of a periodic signal. This approach allows precise power control with high efficiency since the switching elements dissipate minimal power when fully on or off. Drive circuits may also incorporate feedback mechanisms to monitor current, position, or speed, facilitating closed-loop control for improved performance and stability. Additionally, flyback diodes or snubber networks are often integrated to manage inductive kickback when driving coils or motors, preventing voltage spikes from damaging the switching elements. Control and drive circuits form the critical link between electronic logic systems and mechanical output devices.

3.1.12.8 Communication and Telemetry Circuits

Communication circuits enable data exchange between electronic devices, while telemetry circuits extend this concept by allowing information to be transmitted over distances for monitoring or control. These systems utilize various methods of signal transmission, including electrical, optical, and wireless communication. Typical implementations employ transceiver modules that perform both modulation and demodulation of signals, converting baseband data into carrier waveforms suitable for transmission and back into digital form at the receiver end.

Different communication protocols and physical layers have been developed to support a wide range of data rates and distances. Serial interfaces such as UART, SPI, and I²C are commonly used for short-distance communication between integrated circuits, while wireless technologies such as Bluetooth, Wi-Fi, and radio frequency (RF) systems support longer-range data

transmission. Proper design of communication circuits requires careful impedance matching, filtering, and shielding to minimize noise and ensure data integrity. Telemetry circuits are frequently employed in systems that require remote observation or real-time performance tracking, enabling continuous communication between mobile or distributed electronic platforms and central control units.

3.2 Part Selection

3.2.1 Microcontroller Unit - ESP32

After comparing several microcontroller families, the **ESP32** from Espressif Systems was selected as the most practical and versatile choice for our project. It provides an excellent balance of processing power, connectivity, and energy efficiency, all within a compact and cost-effective platform. The ESP32 features a dual-core Tensilica LX6 processor running at up to 240 MHz, along with 520 KB of SRAM and at least 4 MB of onboard flash memory, offering ample performance for handling real-time data processing, wireless communication, and system control simultaneously.

A major advantage of the ESP32 is its integrated Wi-Fi and Bluetooth connectivity, which allows for seamless telemetry, configuration, and communication between the drone and the base station without additional wireless modules. Its wide range of built-in peripherals that includes UART, I²C, SPI, PWM, ADC, etc makes it highly adaptable for interfacing with sensors, motor controllers, and other external hardware components used in the design.

The ESP32 also supports multiple low-power operating modes, making it suitable for lightweight drone applications where efficiency directly affects flight time. It is supported by several development frameworks such as the ESP-IDF and Arduino IDE, both of which have extensive documentation and community support, simplifying firmware development and debugging.

Overall, the ESP32 provides a strong combination of computational capability, integrated connectivity, and flexibility, making it an ideal choice for both the drone's embedded control systems and the wireless communication interface of the base station.

3.2.2 Flight Controller - SpeedyBee F405 Mini

After reviewing several flight controller options, the **SpeedyBee F405 Mini** was selected as the most suitable choice for our design. It provides a strong combination of performance, compact size, and compatibility with common drone firmware. Built around the STM32F405 microcontroller, it operates at 168 MHz and includes a floating-point unit that allows for fast, precise flight calculations. This makes it ideal for real-time control tasks such as stabilization, sensor fusion, and motor signal processing. The 20x20 mm mounting pattern fits perfectly within a small drone frame, helping to minimize weight and maintain a clean internal layout.

The SpeedyBee F405 Mini also includes integrated features that simplify the overall system design. It supports a 4-in-1 ESC connector, reducing wiring complexity and making assembly much easier. Built-in Bluetooth connectivity allows for wireless tuning and configuration

through the SpeedyBee mobile app, which is especially convenient during testing. Its compatibility with widely used firmware such as Betaflight and INAV gives the flexibility to experiment with different control modes while maintaining a reliable flight foundation. The board's strong community support, low cost, and proven stability make it an efficient and dependable choice for our project's control system.

3.2.3 Sensor - VL53L8CX

For this project, a single payload sensor was chosen to serve as the system's primary measurement device: the **VL53L8CX Time-of-Flight (ToF)** distance sensor. This sensor was selected because it offers a strong balance between accuracy, size, and reliability, all of which are essential for a scanning drone operating under tight weight constraints. The VL53L8CX uses modulated infrared light to determine distance with millimeter-level precision and can detect objects up to several meters away. Its small form factor and low power consumption make it ideal for integration into a lightweight platform, while its digital I2C interface allows for easy communication with the flight controller.

The ToF sensor is mounted on the underside of the drone and collects distance readings as the system continuously rotates. This spinning motion allows the sensor to gather a full set of environmental measurements in all directions, effectively creating a 360-degree map without the need for multiple sensors. Compared to alternatives such as ultrasonic modules or full lidar systems, the VL53L8CX provides similar accuracy in a smaller and more energy-efficient package. Its fast response time and immunity to propeller noise make it particularly well suited for rotating applications. By using a single, high-performance ToF sensor as the payload, the system achieves the precision and coverage required for environmental scanning while maintaining a simple, lightweight, and efficient design.

3.2.4 Motor XING2 1404 (2650KV or 3800KV)

For this project, we are using XING2 1404 brushless motors rated at **2650KV**, selected to provide a balance between torque and RPM suitable for slightly larger propellers. Each motor has a 14 mm stator diameter and 4 mm stator height, weighing roughly 9–10 grams. Electrically, these motors draw an average of approximately 5 A in hover with a maximum continuous current around 12 A, making them suitable for 3S LiPo operation. The 1404's compact size offers an excellent power-to-weight ratio and is capable of producing up to 400–500 g of thrust per motor when paired with appropriately sized propellers (such as 3–3.5 inch props), which is more than sufficient for a 250–400 g drone. Mechanically, they use M2 mounting screws and a standard micro prop adapter compatible with small-format propellers.

These motors were chosen because they provide the right balance of thrust and efficiency for a lightweight **tri-motor (tricopter) setup**, allowing stable hover, responsive control, and safe current margins when paired with 20 A ESCs. Using three motors results in a combined hover current near 15 A total, which ties directly into the battery and flight time calculations below.

3.2.5 Propellers - 3.5-inch Tri-Blade (HQProp T3.5×2×3

After evaluating multiple propeller options and conducting real-world flight testing, we selected the **HQProp T3.5×2×3 tri-blade propellers** as the primary prop choice for our tricopter design. Compared to earlier 3-inch configurations, the 3.5-inch props provide improved lift efficiency and lower required throttle for hover, which is critical for achieving stable flight and longer flight times.

Each propeller features a **3.5-inch diameter with a 2-inch pitch** and is constructed from durable polycarbonate, offering a strong balance between thrust generation and efficiency. The tri-blade design increases control authority, particularly for stabilization during hover and while performing controlled spin maneuvers (~60 RPM), which is a key requirement of our project.

In testing, these propellers allowed the drone to hover at approximately **~1600–1650 throttle**, demonstrating improved efficiency compared to smaller or higher-pitch alternatives. The increased diameter produces more thrust at lower RPMs, reducing strain on the motors and improving overall flight stability. While tri-blade props introduce slightly higher drag and current draw than bi-blade options, the added control responsiveness and smoother flight characteristics outweigh this drawback for our application.

Mechanically, the HQProp T3.5×2×3 propellers are fully compatible with our motor setup and mount securely without modification. Their relatively low mass helps minimize rotational inertia, improving response time and reducing oscillations during rapid adjustments.

Overall, this propeller choice provides the best balance of **thrust, efficiency, and control**, enabling stable hover performance and consistent spin behavior while maintaining acceptable power consumption.

3.2.6 Batteries — 3S 850 mAh 60C LiHV Battery

After reviewing multiple battery options and conducting real-world flight testing on our 3-motor, 3″ prop drone, we selected the **Gaoneng (GNB) 850 mAh 3S 60C LiHV battery** as our primary power source. Unlike earlier estimates, this selection is based on measured performance rather than theoretical calculations.

This battery provides a balanced combination of **low weight (~53 g)** and sufficient discharge capability for stable hover and controlled spinning flight. During testing, the drone achieved approximately **5–5.5 minutes of hover time** and about **~4 minutes during continuous spin operation**, which aligns with our project requirements. The significantly lower weight compared to larger batteries improves thrust-to-weight ratio, reduces drift, and enhances overall flight stability—critical for a tricopter design.

Although higher-capacity batteries (e.g., 1500 mAh–2200 mAh) can theoretically increase flight time, their added weight negatively impacts efficiency and control. Our testing showed that maintaining a lighter system yields more reliable and predictable flight behavior, especially for hover stabilization and spin consistency.

As a fallback strategy, higher-capacity batteries may still be considered if longer endurance is required and the drone can maintain stable flight under increased weight. However, the 850 mAh configuration currently provides the best real-world balance between **flight time, efficiency, and stability**.

Battery	Capacity	Weight	Estimated Hover Time (min)	Notes / Strategy
GNB 850 mAh 3S 60C LiHV	850 mAh	~53 g	~5.2	**Primary choice**: longest hover time per gram of weight. Will be used if total drone weight and stability are acceptable.
Zeee 1500 mAh 3S 120C	1500 mAh	~134 g	~7.4	**Fallback 1**: good mid-range option. Used if Ovonic is too heavy or hover current becomes too high.
Tattu 1300 mAh 3S 75C	1300 mAh	~120 g	~6.5	**Fallback 2**: lightest option. Used if weight savings is critical or motors need better thrust margin.

3.2.7 Communication Technologies

For the communication system of the Dizzy n-Copter, the priority was selecting a wireless technology that is lightweight, low-power, and capable of reliably transmitting telemetry data from the drone to a base station. Since the drone integrates an ESP32 microcontroller, the most practical option was to leverage its built-in wireless capabilities rather than adding external radio modules. This reduces weight, minimizes power consumption, and simplifies the overall system architecture.

The ESP32 supports both Wi-Fi and Bluetooth Low Energy (BLE), which provides two viable approaches for communication with a smartphone. After evaluating range, bandwidth, reliability, and ease of integration with mobile apps, Wi-Fi was selected as the primary communication technology. Wi-Fi provides significantly higher throughput compared to BLE, making it more

suitable for sending rapid telemetry updates, sensor readings, and potential future data streams such as rotation-based scanning output. It also allows the drone to operate in one of two modes:

1. **ESP32 Soft-AP mode**: Where the ESP32 creates its own local Wi-Fi network the phone connects to

2. **Station mode**: Where both the ESP32 and the mobile device join the same network.

Both modes allow a direct, low-latency link between the drone and the app without requiring additional hardware.

Bluetooth Low Energy was also considered due to its simplicity and ultra-low power draw. However, its limited data rate and shorter range make it less suitable for continuous telemetry during flight. While BLE could still be used for configuration or initial pairing, Wi-Fi remains the better fit for live data transmission.

Overall, selecting the ESP32's built-in Wi-Fi system streamlined the communication design by removing the need for external radios, reducing cost, and allowing direct integration with the mobile application. This ensures a reliable link between the drone and the user while keeping the system lightweight and easy to implement.

3.2.8 Development Environment

For the development environment, the most important factor for our group was familiarity and simplicity. Since this is a collaborative project, it was crucial to use tools and programming languages that everyone on the team already had experience with to minimize setup time and reduce unnecessary complexity. Our goal was to avoid learning entirely new systems and instead focus on developing, testing, and integrating our hardware and software as efficiently as possible.

For version control and collaboration, we chose GitHub as our primary repository. Every member of our group has experience using GitHub, which allows for easy file sharing, revision tracking, and collaborative coding. Using GitHub also ensures that our firmware and base station code remain organized and accessible, with version history for debugging and development milestones. Additionally, GitHub provides access to open-source libraries and example code that can accelerate development, especially for embedded systems and wireless communication tasks.

Our primary programming language for the project is C, which is ideal for embedded systems programming and offers direct control over microcontroller peripherals. We will also use C++ when object-oriented functionality is needed, and Python for higher-level testing or data visualization. For the ESP32 microcontroller, development will be done through ESP-IDF or the Arduino IDE, depending on the subsystem being developed.

For flight control, we are utilizing Betaflight, an open-source firmware platform commonly used in multirotor drones. Betaflight provides reliable stabilization, sensor integration, and motor control algorithms that have been thoroughly tested by the drone community. By using Betaflight

on the SpeedyBee F405 Mini flight controller, we can focus our development efforts on higher-level features such as telemetry, base station communication, and custom control logic rather than rewriting low-level flight control code from scratch.

Overall, we will use GitHub for collaboration, C/C++ for firmware, ESP-IDF for development, and Betaflight for flight control. Which provides a familiar, efficient, and stable environment for our team. It allows each member to contribute effectively while ensuring smooth integration between the drone's firmware, hardware, and base station systems.

Chapter 4 - Standards and Design Constraints

4.1 Industrial Standards

4.1.1 PCB Design Standards

4.1.1.1 Overview

All printed circuit boards (PCBs) for the Dizzy n-Copter are designed to meet industry-recognized standards to ensure electrical reliability, manufacturability, and long-term durability. Because the system integrates multiple low-voltage logic and power domains on lightweight, high-density boards, adherence to these standards is critical to achieving both performance and safety objectives. The following guidelines establish the design rules, fabrication constraints, and documentation requirements used throughout the project.

4.1.1.2 Applicable Standards

The PCB design process follows specifications derived primarily from **IPC** and **ISO** standards commonly used in aerospace and consumer-electronics applications:

Standard	Description
IPC-2221 B	Generic Standard on Printed Board Design; governs trace width, clearance, via design, and mechanical spacing.
IPC-7351	Land Pattern Design Standard for surface-mount components; used to ensure manufacturable pad geometry.

IPC-6012 D	Qualification and Performance Specification for Rigid Printed Boards; establishes minimum copper thickness, dielectric spacing, and hole plating quality.
IPC-A-600 G	Acceptability of Printed Boards; used as the inspection reference for board fabrication and assembly quality.
ISO 9001	Ensures that board fabrication and assembly partners follow quality-management processes for traceability and repeatability.
IPC-A-610 H	Acceptability of Electronic Assemblies; defines workmanship criteria for solder joints and component alignment.

4.1.1.3 Design Rules and Manufacturing Constraints

To achieve compact and lightweight layouts without sacrificing reliability, the following quantitative design rules were applied in EasyEDA:

- **Minimum Trace Width:** 0.25 mm (10 mil) for signal lines; 0.5 mm (20 mil) or greater for high-current motor and power traces.
- **Minimum Clearance:** 0.25 mm (10 mil) between conductive features to prevent arcing or solder bridging.
- **Via Hole Diameter:** 0.6 mm drill / 1.0 mm pad for standard vias; 0.8 mm drill for current-carrying vias.
- **Copper Thickness:** 1 oz/ft² for signal layers; 2 oz/ft² for the power-distribution layer on the motor board.
- **Board Thickness:** 1.6 mm (standard FR-4).
- **Solder Mask Clearance:** 0.05 mm to ensure proper insulation and solderability.
- **Silkscreen Text Height:** ≥ 1.0 mm for legibility during inspection.

All footprints were selected from verified component libraries in accordance with IPC-7351. Component placement was performed to minimize high-frequency coupling and noise between switching regulators, analog sensors, and the ESP32 RF section. Return-path integrity and decoupling capacitor placement follow standard high-speed design practices.

4.1.1.4 Layer Stack-Up and Grounding

A **two-layer stack-up** was chosen for all boards to minimize weight and cost while maintaining signal quality:

- **Top Layer:** High-speed and power routing; critical analog and digital signal traces.
- **Bottom Layer:** Dedicated ground plane with localized copper pours for +5 V and +3.3 V rails.

Ground returns from sensors, regulators, and servos converge at a single star-point reference near the battery connector to reduce common-mode noise and ground bounce. The ESP32's RF ground region is isolated from switching converter grounds to comply with FCC EMI considerations.

4.1.1.5 Thermal and Reliability Considerations

Power components such as the LM2576 regulator, MOSFETs, and linear regulators are placed with adequate copper pour for heat dissipation. Thermal vias are used under devices with exposed pads. The design ensures:

- **Temperature rise < 40 °C** under maximum continuous load.
- Adequate separation between hot and temperature-sensitive components (sensors, MCU).
- Conformal-coating clearance zones for environmental protection during testing.

4.1.1.6 Assembly and Inspection Practices

All PCBs are designed for **surface-mount assembly (SMD)** except for essential connectors (battery input, motor/servo headers). Component orientation follows a consistent reference system for automated placement. During inspection, boards will be evaluated under IPC-A-600 and IPC-A-610 standards for solder joint quality, solder-mask coverage, and silkscreen accuracy.

Test pads are included for continuity verification, and fiducial marks are provided to ensure alignment during pick-and-place manufacturing.

4.1.1.7 Documentation and Version Control

Each schematic and PCB layout is maintained under version control using EasyEDA's integrated document history system. Every design revision includes:

- Updated **BOM** with verified LCSC part numbers.
- **Gerber files** reviewed for DRC/DRC compliance.
- Annotated **assembly drawings** identifying reference designators and orientation marks.
- **Revision log** summarizing engineering changes.

4.1.1.8 Summary

By conforming to IPC and ISO standards, the Dizzy n-Copter's PCB design process ensures repeatable quality, mechanical integrity, and electrical performance. The resulting layouts are optimized for manufacturability, reliability, and minimal electromagnetic interference—meeting the dual requirements of a research-grade embedded platform and a safe, lightweight aerial prototype.

4.1.2 I2C Protocol Standards

The Inter-Integrated Circuit (I2C) protocol is a widely used synchronous, multi-master, multi-slave serial communication standard developed by Philips (now NXP). It allows multiple devices to communicate over just two wires: a serial data line (SDA) and a serial clock line (SCL). For this project, I2C is employed primarily to interface onboard sensors, including inertial measurement units (IMUs), barometers, and magnetometers, with the SpeedyBee F4 flight controller. It may also be used to interface with auxiliary microcontrollers (such as an ESP32) for telemetry and wireless data transmission.

I2C was selected due to its simplicity, low pin count, and ability to handle multiple devices over a single bus, which is critical in small UAVs where space and weight are limited.

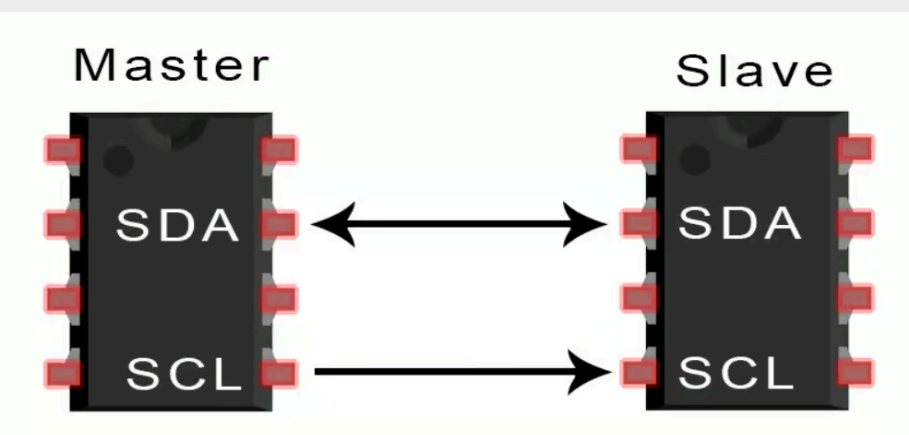


Figure 4.3. *Basics of the I2C Communication Block Diagram by Image Source:* *https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/*

4.1.2.1 Electrical Design Standards

1. **Bus Voltage Levels**:
    - All devices on the I²C bus must operate at compatible voltage levels.
    - In this project, the bus is standardized to 3.3 V logic, as both the SpeedyBee F4 and ESP32 operate at 3.3 V. Level shifters are required if any 5 V sensors are connected.
2. **Pull-up Resistors**:
    - I2C requires pull-up resistors on both SDA and SCL lines to ensure proper logic at high levels.
    - For our 3.3 V bus and expected bus capacitance (<200 pF), 4.7 kΩ resistors are used on both lines, in accordance with NXP recommended values for 400 kHz operation.
    - Resistors are placed close to the microcontroller to minimize ringing and noise.
3. **Bus Capacitance and Trace Lengths**:
    - I2C bus capacitance must not exceed 400 pF to maintain signal integrity.
    - PCB traces are kept as short as possible (<10 cm), and shielded or twisted-pair wiring is used for longer sensor cables to minimize electromagnetic interference (EMI) from motors and ESCs.

4. **Clock Frequency**:
    - Standard I2C speed is 100 kHz, while fast mode is 400 kHz.
    - For this drone project, the bus will operate at 400 kHz to achieve rapid sensor polling without compromising flight controller responsiveness.
5. **Bus Termination and Filtering**:
    - Optional series resistors (22–47 Ω) can be added to SDA and SCL lines to reduce ringing.
    - Decoupling capacitors (0.1 μF) are placed near each sensor VCC to reduce voltage fluctuations caused by PWM-driven motors.

4.1.2.2 Protocol Standards

1. **Addressing**:
    - I²C devices must have unique 7-bit addresses on the bus to avoid collisions.
    - Sensors and peripherals are verified against the bus to prevent address conflicts.
    - Where conflicts arise, address pins (AD0/AD1) are configured per device datasheets.
2. **Data Integrity**:
    - Multi-byte transmissions are acknowledged by the receiving device (ACK bit) to ensure data correctness.
    - Software-level timeouts are implemented on the master (SpeedyBee F4 or ESP32) to recover from bus hangs.
3. **Clock Stretching**:
    - Slave devices may hold the SCL line low to delay the master if they need extra time to process data.
    - All slave devices in this project support clock stretching, and the master firmware handles stretched clocks according to I²C standard timing diagrams.
4. **Error Handling**:

- Any NACK received triggers a retry mechanism in firmware.
- Bus recovery procedures include toggling SCL up to 9 times to release a stuck slave, followed by a restart condition.

4.1.2.3 Mechanical and Layout Considerations

1. **Separation from High-Current Lines**:
    - SDA and SCL traces are routed away from high-current ESC and motor traces to reduce EMI.
    - Ground planes are used extensively under the I²C bus to provide shielding.
2. **Connector and Cable Standards**:
    - On modular connections (e.g., between the flight controller and external ESP32 telemetry module), 4-pin JST-SH or Dupont cables are used: SDA, SCL, VCC, GND.
    - Cables are kept short (<15 cm) and twisted when necessary to reduce noise pickup.
3. **Mechanical Vibration Mitigation**:
    - Small foam dampeners are placed under sensitive sensors to reduce vibration-induced I²C errors.
    - Flexible wires with strain relief prevent mechanical fatigue during drone maneuvers.

4.1.2.4 Summary of I2C Design Guidelines for This Project

Parameter	Recommended Value / Standard	Notes
Logic voltage	3.3 V	Matches SpeedyBee F4 and ESP32
Pull-up resistors	4.7 kΩ	Close to master

Maximum bus capacitance	400 pF	Ensure short trace lengths
Clock frequency	400 kHz	Fast mode for responsive sensor polling
Bus wiring	Short, shielded, twisted where needed	Minimize EMI
Error handling	ACK/NACK + retry + bus recovery	Software implemented on master
Addressing	Unique 7-bit addresses	Configure AD pins if needed
Decoupling	0.1 µF ceramic per sensor	Stabilize supply voltage
Mechanical	Foam dampening + strain relief	Protect against vibration and mechanical fatigue

4.1.2.5 Conclusion

Adhering to these I²C design standards ensures reliable and robust communication between the flight controller, sensors, and telemetry modules in the UAV. Proper bus termination, voltage matching, EMI mitigation, and error handling are critical in the high-vibration, high-EMI environment of a quadcopter. By following these guidelines, the system achieves low-latency, accurate sensor readings while maintaining fault tolerance, enabling safe and efficient drone operation.

4.1.3 UART Protocol Standards

The Universal Asynchronous Receiver-Transmitter (UART) protocol is one of the most widely used serial communication standards for embedded systems. It enables full-duplex data transfer between two devices using just two lines: TX (Transmit) and RX (Receive). Unlike synchronous protocols such as I2C or SPI, UART does not use a shared clock line; instead, both devices must agree on timing parameters like baud rate and frame format.

In this project, UART communication is primarily used for telemetry and data exchange between the SpeedyBee F4 flight controller and auxiliary systems such as the ESP32 Wi-Fi module or a ground control station. It may also be used for serial debugging and sensor data output during development. The simplicity, reliability, and long-distance capability of UART make it ideal for asynchronous communication in UAV systems.

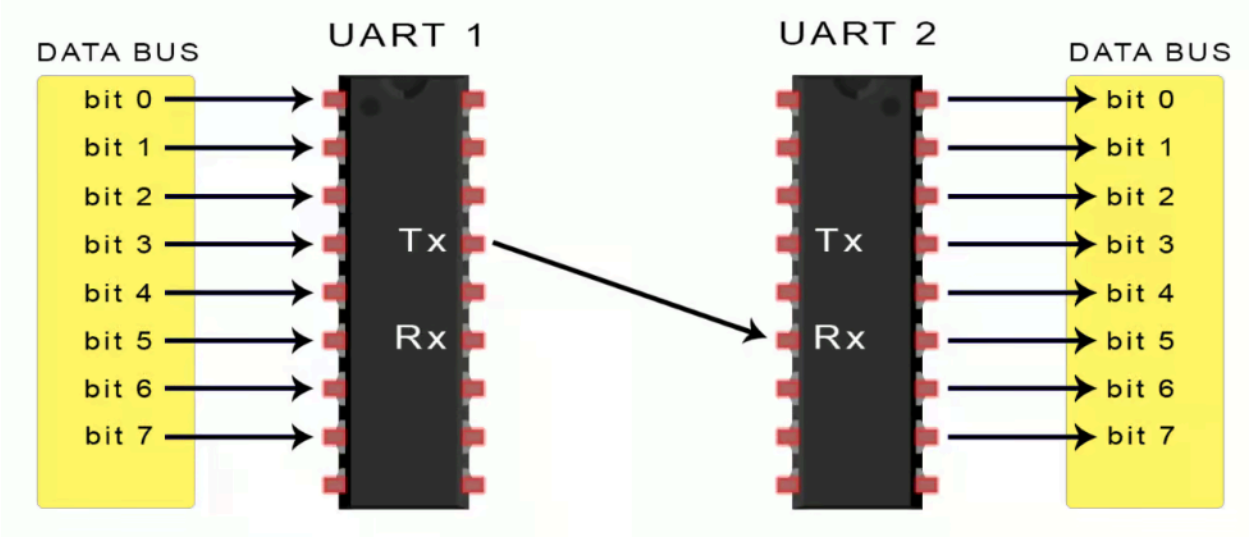


Figure 4.4. *Basics of UART Block Diagram by Image Source:* *https://www.circuitbasics.com/basics-uart-communication/*

4.1.3.1 Electrical Design Standards

1. **Voltage Levels:**
    - The UART lines operate at 3.3 V logic, matching both the *SpeedyBee F4* and *ESP32*.
    - If a 5 V UART device is connected, a bi-directional logic level shifter must be used to prevent damage.
2. **Wiring and Connections:**
    - UART requires crossed connections between devices (TX → RX and RX → TX).
    - A common ground (GND) must be shared between both communicating devices.
    - For modular components, 4-pin JST-SH or Dupont connectors are used (TX, RX, VCC, GND).
3. **Noise Reduction:**

- TX and RX traces are kept short (<15 cm) and routed away from high-current ESC and motor lines.
- Where longer runs are needed, twisted-pair wiring or shielded cables are used to minimize EMI pickup.
- Decoupling capacitors (0.1 μF) are placed near UART power pins for stability.

4.1.3.2 Protocol Standards

1. **Frame Format:**
    - Standard UART frame: 1 start bit, 8 data bits, no parity, 1 stop bit (8N1).
    - Optional parity (even/odd) can be configured if data integrity issues arise.
2. **Baud Rate:**
    - Common rates: 9600, 57600, 115200, and 921600 bps.
    - For this project, a 115200 baud rate is used for sensor data and command transmission.
    - The baud rate tolerance must be within 2% between devices to prevent bit drift.
3. **Flow Control:**
    - Hardware flow control (RTS/CTS) is disabled for simplicity; software handles buffering.
    - If high-speed communication (>500 kbps) or continuous telemetry is used, enabling RTS/CTS is recommended.
4. **Error Detection:**
    - UART includes no built-in CRC, but parity bits or software-level checksums can be added.
    - The firmware implements timeout detection and retry routines if data framing errors (FE) or overruns (OE) occur.
5. **Synchronization:**

- Because UART is asynchronous, both ends must share the same configuration (baud rate, parity, stop bits).
- Any mismatch results in corrupted or unreadable data.

4.1.3.3 Mechanical and Layout Considerations

1. **Trace Routing:**
    - TX/RX traces are routed away from switching components and PWM lines to minimize noise.
    - Ground planes are maintained beneath signal traces for shielding.
2. **Connector Standards:**
    - 4-pin JST-SH or Dupont connectors are used for UART interfaces: TX, RX, VCC, GND.
    - Strain relief or hot glue is applied to prevent connector fatigue during vibration.
3. **Cable Lengths:**
    - For 3.3 V TTL UART, cable lengths should not exceed 30 cm without shielding.
    - For longer runs, differential line drivers (RS-485) can be used to extend range.
4. **Vibration and Strain Mitigation:**
    - Flexible silicone-insulated wires are used to reduce fatigue during drone movement.
    - UART modules mounted with small foam dampers minimize stress on solder joints.

4.1.3.4 Summary of UART Design Guidelines for This Project

Parameter	Recommended Value / Standard	Notes

Logic voltage	3.3 V	Matches SpeedyBee F4 and ESP32
Frame format	8N1	8 data bits, no parity, 1 stop bit
Baud rate	115200 bps	Standard for telemetry/debugging
Flow control	None (manual/software)	RTS/CTS optional for higher speeds
Cable length	<30 cm (shielded if longer)	Minimize EMI
Pull-ups	Not required	UART uses push-pull lines
Ground reference	Common GND required	Prevent signal offset
Error handling	Timeout + retry + checksum	Implemented in firmware
Connectors	4-pin JST-SH/Dupont	TX, RX, VCC, GND
Vibration mitigation	Strain relief + flexible wire	Increases reliability

4.1.3.5 Conclusion

By adhering to these UART design and communication standards, reliable asynchronous serial data transfer is ensured between the flight controller, telemetry modules, and auxiliary microcontrollers. Proper voltage matching, cable management, and EMI reduction are crucial to maintaining stable communication in the electrically noisy environment of a multirotor UAV.

These design practices conform to the TIA/EIA-232-F and TTL-level UART electrical guidelines, ensuring compatibility, low latency, and fault tolerance in drone operations.

4.1.4 Betaflight

4.1.4.1 Overview

Betaflight is an open-source flight control firmware optimized for multirotor and fixed-wing aircraft, focusing on high-performance, real-time attitude control. It runs on a wide range of STM32-based flight controllers, including the SpeedyBee F4 used in this project. Betaflight's flexibility, modular configuration system, and active developer community make it the de facto standard for drone flight stabilization and tuning in both hobbyist and research-grade UAVs.

In this project, Betaflight is used to handle core flight stabilization, sensor fusion, ESC and motor control, and telemetry management, while the ESP32 serves as an auxiliary module for data logging and wireless communication. The design adheres to Betaflight configuration, wiring, and tuning standards to ensure stable, repeatable, and safe flight performance.

4.1.4.2 Firmware and Configuration Standards

Firmware Versioning
The project standardizes on Betaflight 4.5 or newer, ensuring compatibility with the SpeedyBee F4 and access to modern features such as improved PID filtering and bidirectional DShot support. Firmware version and configuration dump (diff all) are archived for version control and reproducibility.

Flight Controller Target
The firmware is compiled for the SPEEDYBEEF4 target to ensure proper pin mapping, UART configuration, and hardware compatibility. No custom firmware modifications are made outside of configurable settings to maintain firmware integrity.

Configuration Tool
Betaflight Configurator (desktop application) is used for all setup and tuning. Configuration backups are stored as JSON files under version control, ensuring recovery in case of data corruption or firmware re-flash.

Parameter Consistency
All tuning parameters (PID gains, filter settings, rates, and mixer settings) are documented and version-tagged.Configuration changes are tracked through Betaflight's CLI to maintain transparency in performance testing.

4.1.4.3 Hardware and Wiring Standards

UART and Peripheral Assignments
Betaflight supports multiple UART ports for peripherals such as GPS, smart audio, telemetry, and serial RX.

For this project:

1. **UART1** → Receiver (SBUS/CRSF)
2. **UART2** → ESC Telemetry (if supported)
3. **UART3** → ESP32 module for data transmission
4. **I²C Bus** → Sensors (Barometer, Magnetometer, etc.)

All serial peripherals use 3.3 V logic and share a common ground to ensure reliable communication.

Power Distribution

The SpeedyBee F4 receives regulated 5 V power from the power distribution board (PDB). All peripherals (ESP32, sensors) are powered from 3.3 V regulators with adequate decoupling capacitors (≥0.1 µF local, 10 µF bulk). Power lines are twisted or shielded to reduce electromagnetic interference from ESC switching transients.

ESC and Motor Standards
ESCs are configured to use DShot600 digital protocol for precise, noise-tolerant motor control. Bidirectional DShot is enabled for RPM filtering, which provides real-time feedback to Betaflight for vibration reduction. Motor numbering follows Betaflight's quadcopter X configuration standard for clockwise/counterclockwise rotation patterns.

Sensor Integration
IMU (gyro and accelerometer) sampling rate is configured to 8 kHz with a control loop rate of 4 kHz for stable flight response. Sensors are mounted on vibration-damping foam to minimize aliasing in gyro readings. I2C sensors follow the design standards described in Section 3.X (I²C Communication Standards).

4.1.4.4 Software and Control Loop Standards

PID Controller
The firmware uses the PID 2.0 controller (default) with active notch and low-pass filters. Initial PID tuning is derived from Betaflight presets for 5-inch quads, followed by manual tuning during flight testing. PID coefficients are adjusted to achieve minimal oscillation while maintaining agile response under load.

Filter Configuration
Dynamic Notch Filter enabled for motor noise adaptation. Gyro low-pass filters are set between 90–120 Hz to balance latency and noise rejection. RPM-based filtering is utilized when bidirectional DShot is enabled.

Failsafe and Arming Standards

1. **Stage 1 Failsafe**: Throttle hold for 0.4 s.

2. **Stage 2 Failsafe**: Motor stop and disarm to prevent runaway drone events.

Arming is only permitted when accelerometer calibration is complete and no sensors report fault states.

Telemetry and Logging
Betaflight's SmartPort telemetry transmits flight data to the ESP32 for real-time wireless logging. Data includes voltage, current draw, altitude, and flight mode information. Blackbox logging is enabled on the onboard flash memory for debugging control loop performance.

4.1.4.5 Safety and Testing Standards

Pre-Flight Safety Checks

Before arming, Betaflight verifies that all sensors are initialized, receiver input is valid, and no failsafe conditions exist. Propellers remain removed during all bench testing and configuration changes.

Power-On Self-Test (POST)
The flight controller performs automatic gyro calibration and sanity checks on startup. LED indicators confirm successful boot, arming readiness, and battery voltage status.

Firmware Integrity and Recovery
Bootloader access is preserved for emergency reflashing. A stable backup configuration is stored in both local and cloud repositories.

Signal and Noise Testing
All signal lines (PWM, UART, I2C) are verified using an oscilloscope for clean transitions and minimal jitter. EMI mitigation is tested by powering the drone with all motors active and verifying data integrity across the telemetry link.

4.1.4.6 Summary of Betaflight Design Guidelines

Category	Standard / Configuration	Purpose
Firmware	Betaflight 4.5+	Stable and supported firmware version
Control Loop	8 kHz gyro / 4 kHz PID	Optimal for 5-inch drone responsiveness

ESC Protocol	DShot600 with bidirectional telemetry	Fast, noise-resistant motor control
UART3	ESP32 data link	Wireless telemetry and command
Sensor Mounting	Soft foam isolation	Minimize vibration interference
Failsafe	Stage 1 (hold), Stage 2 (disarm)	Prevent flyaway and hardware damage
Power	5 V FC, 3.3 V peripherals	Prevent voltage mismatch and brownouts
Filter Setup	Dynamic notch + RPM filtering	Stable and noise-free flight
Logging	SmartPort + Blackbox	In-flight diagnostics and performance evaluation

4.1.4.7 Conclusion

By following these Betaflight design standards, the system achieves consistent, safe, and high-performance flight behavior. The standardized configuration and wiring conventions ensure maintainability and reproducibility across all drone iterations. Integration with the ESP32 provides a flexible platform for telemetry expansion and real-time performance monitoring. These standards ensure the drone can be tuned, tested, and flown with both stability and adaptability for future development and research tasks.

4.1.5 Flight Controllers

In the field of unmanned aerial vehicles (UAVs), the flight controller (FC) serves as the primary control unit responsible for interpreting sensor data and generating motor commands to maintain flight stability. While there are no single global physical-dimension standards for flight controllers as there are for electric motors, their electrical, communication, and safety interfaces are guided by several international standards and design conventions recognized across the drone industry.

Flight controllers are typically designed according to electronics and data communication standards such as the International Electrotechnical Commission (IEC), Institute of Electrical and Electronics Engineers (IEEE), and Radio Technical Commission for Aeronautics (RTCA). These organizations provide framework guidelines ensuring electromagnetic compatibility, environmental robustness, and communication integrity among onboard avionics.

Key standards often referenced in UAV flight controller design include IEC 60068 for environmental testing (vibration, shock, temperature), IEC 61000 for electromagnetic compatibility (EMC), and RTCA DO-160G for airborne electronic equipment environmental conditions. Compliance with these standards ensures that the flight controller maintains reliable operation under high-vibration and high-EMI conditions caused by brushless motors and ESCs.

In the United States, Federal Aviation Administration (FAA) UAS regulations indirectly guide the control system's design reliability and fail-safe behavior. Although not a hardware specification, these operational standards influence design elements such as redundant sensor inputs, failsafe landing modes, and loss-of-signal protocols, all of which must be supported in the flight controller's firmware and architecture.

Most consumer and research UAVs follow open-source flight control firmware frameworks such as Betaflight, ArduPilot, or PX4, which conform to the MAVLink communication standard (Micro Air Vehicle Link) for telemetry and control data exchange. MAVLink is an open protocol defined and maintained by the Dronecode Foundation and is recognized globally as the standard for autopilot-to-ground station communication.

The SpeedyBee F4 flight controller used in this project conforms to the STM32 microcontroller standard architecture, supporting serial communication protocols such as I2C, SPI, and UART, each governed by their respective electrical standards. It also follows common layout and pin conventions defined by the Drone Manufacturers Alliance (DMA) and OpenPilot board format, allowing for interoperability with standard ESCs, GPS modules, and sensors.

The board typically operates at 3.3 V logic levels and provides a standard PWM output interface for motor control, conforming to the timing requirements outlined by the RC PWM Standard (1–2 ms pulse width, 50–500 Hz frequency range). For digital speed control, the DShot protocol (a modern digital PWM standard) is implemented, ensuring error-free communication between the flight controller and ESCs through CRC-based validation.

Similar to IEC and NEMA classifications for motors, flight controllers can be categorized by their intended duty class and environmental rating. According to IEC 60068, controllers are tested under continuous operation with intermittent vibration exposure (equivalent to Duty Class S6 under IEC 60034-1 for motors). Controllers designed for extended outdoor use are expected to meet IP54 or higher ingress protection ratings, providing dust and splash resistance.

In terms of mechanical and dimensional coordination, most flight controllers follow standardized mounting hole patterns:

- 30.5 × 30.5 mm (standard 5-inch racing drones)

- 25.5 × 25.5 mm (compact builds)
- 20 × 20 mm (micro drones)
- 16 × 16 mm (ultralight frames)

These mounting standards are defined by open-source hardware guidelines and widely adopted across the drone industry to ensure component interchangeability.

In conclusion, the design and integration of flight controllers follow internationally recognized standards that ensure electrical compatibility, environmental reliability, and communication interoperability. Adherence to these conventions allows UAV systems to maintain stable flight performance, ensure user safety, and achieve modular compatibility with various sensors, ESCs, and telemetry systems across manufacturers.

4.2 Practical Constraints

4.2.1 Time

The primary constraint of this project is how limited we are on time. Completing the Dizzy Copter as part of the Senior Design project gives us only a short window to research, design, build, and test a functioning prototype. This time restricted schedule limits how much we can accomplish and requires that certain design decisions prioritize efficiency and feasibility over achieving the most advanced product possible.

One of the first major challenges associated with the time constraint is that we only have a single academic year, two semesters, to produce a fully functional drone. Since extensions are not permitted, this creates a strict and immovable deadline that we must meet to successfully pass the course. Fortunately, our group took the initiative and showed real interest in this project and its design. Our approach should have us developing a working prototype by the end of the first semester.

Another challenge brought on by the time constraint is the need to complete a significant amount of testing, debugging, and design work within a limited timeframe. Throughout the course of this project, there will inevitably be setbacks, errors, and adjustments that require additional time to address. This makes time management critical, as every delay in hardware integration, flight testing, or software development reduces the margin we have to meet our final goals.

A common issue in senior design projects related to time is the ordering and delivery of components. Essential parts such as the flight controller, brushless motors, or custom PCBs could experience manufacturing or shipping delays. If even one crucial component arrives late, the group could lose valuable time waiting, which would slow progress on assembly, calibration, and testing.

Another aspect affected by time is the fabrication of the drone’s frame and structure. These components must be designed, cut, and assembled before the start of Senior Design II to

maintain a consistent build schedule. This process can be challenging, as the drone must stay under the 250-gram weight limit while still maintaining stability and strength. Completing this portion early ensures the remainder of the project, such as motor tuning and control software, can proceed without delay.

The result of these time constraints could lead to a rushed development cycle, potentially causing reduced performance or missed goals. This could force the team to make compromises near the end of the semester, impacting the overall quality of the final prototype. To prevent this, our team has committed to key values such as organization, accountability, communication, and time management. We aim to remain consistent and dependable. With proper planning and teamwork, we will be able to overcome the time constraint and complete a successful project within the allotted schedule.

4.2.2 Economic

The project operates under a sponsorship limit of $250, which defines the maximum allowable budget for all hardware, materials, and supporting components. This constraint strongly influences system design choices, requiring a balance between performance, cost, and functionality. Every hardware selection, from the flight controller to communication modules, was made with the goal of achieving reliable drone operation and data transmission without exceeding the funding cap.

Because the drone must support both flight control and wireless telemetry, the design approach emphasizes cost-efficiency through modularity and open-source integration. Expensive proprietary systems such as DJI flight controllers or closed telemetry hardware were excluded in favor of lower-cost, open standards that maintain compatibility with widely supported tools like Betaflight and the ESP32 ecosystem.

4.2.2.1 Cost Optimization Strategies

To remain within the $250 limit, several cost optimization strategies were employed throughout the design process:

1. **Reuse and Modularity**
   Components such as the flight controller, ESP32, and power system are modular and reusable. Should a later phase require different sensors or higher-level processing, these components can be retained, reducing the cost of future iterations. The modular structure also simplifies maintenance and replacement in case of damage.

2. **Open-Source Integration**
   The use of Betaflight, BLHeli_32, and Arduino-based ESP32 firmware avoids licensing costs associated with proprietary flight software. These platforms have extensive community support, allowing free access to tuning guides, configuration tools, and performance updates.

3. **Balanced Performance Scaling**
   Instead of investing in a more expensive STM32F7 or H7 controller, computational tasks such as telemetry processing and sensor data handling are distributed to the ESP32. This approach achieves comparable performance to higher-end systems at a fraction of the cost.

4. **Targeted Procurement**
   Components were sourced through reputable low-cost vendors and suppliers that offer educational discounts or bundle deals. Purchases were prioritized in stages, beginning with the flight-critical subsystems before investing in optional features like telemetry expansion.

5. **Avoidance of Proprietary Interfaces**
   All subsystems communicate via standardized protocols (UART, I2C, and DShot). This ensures compatibility across brands and allows inexpensive off-the-shelf replacements if any part fails, instead of locking the design to a single vendor's ecosystem.

4.2.2.2 Trade-Offs and Economic Implications

The $250 funding cap introduces important trade-offs that influenced design decisions.

- **Processing Power vs. Cost:** A higher-end controller (F7/H7) could offer more UARTs and higher loop frequencies, but its price would exceed the budget. The SpeedyBee F4 meets the control requirements with slight firmware tuning adjustments.

- **Sensor Complexity vs. Reliability:** Advanced sensors such as optical flow, LiDAR, or GPS with RTK positioning were excluded. Instead, the design relies on the onboard IMU and barometer for stable flight and basic altitude estimation.

- **Flight Time vs. Cost:** Larger LiPo batteries were avoided to reduce both cost and added mass, leading to a moderate but acceptable flight duration.

- **Frame Durability vs. Expense:** A carbon fiber frame was chosen over plastic alternatives due to its superior strength-to-weight ratio, even though it consumes a larger portion of the budget.

These trade-offs ensure that the drone remains both functional and financially feasible while maintaining safety and performance standards suitable for field testing and data collection.

4.2.2.3 Long-Term Cost Efficiency

Although the project is initially limited to $250, its architecture supports incremental expansion with minimal redesign. Future upgrades can include such as GPS modules, higher-capacity batteries, or camera payloads. This can be added without replacing core components. The

decision to base the system on open-source firmware also ensures long-term sustainability, as updates and community support remain freely available.

4.2.2.4 Summary

In summary, the project's economic design emphasizes maximized functionality within a fixed financial limit. Through open-source tools, modular hardware selection, and strategic cost optimization, a fully capable 5-inch drone system with wireless telemetry was achieved under a total cost of $250. The economic planning process balanced performance and reliability with affordability, ensuring essential capabilities of flight stability, data transmission, and system safety. To make sure none were compromised.

This constraint-driven design process not only demonstrates fiscal responsibility but also highlights the team's ability to deliver a technically advanced, expandable, and cost-efficient UAV system within a limited budget framework.

Chapter 5 -Application of ChatGPT and Other Similar Platforms

ChatGPT was used as a research and resource-finding tool throughout the development of our drone project. It helped us identify relevant components, technologies, and academic references by providing quick overviews, summaries, and lists of potential options to investigate further. This greatly accelerated the research process, especially when comparing sensors, microcontrollers, motor controllers, and other drone subsystems.

While ChatGPT's responses provided useful direction, we treated all of its outputs as a starting point rather than a verified source. Its knowledge base can sometimes be outdated or incomplete, particularly regarding pricing, availability, and newly released hardware. For that reason, every piece of technical information obtained through ChatGPT was cross-checked using manufacturer datasheets, vendor product pages, and peer-reviewed papers before inclusion in our report.

In summary, ChatGPT was an effective research aid that allowed our team to quickly organize topics, find relevant resources, and develop an initial understanding of key technologies used in our drone system. However, all critical data, which includes electrical specifications, part ratings, and design standards, were verified through official documentation to ensure the accuracy and reliability of the final design.

5.1 Case Study 1

When tasked with selecting a flight controller for our drone, ChatGPT was a helpful tool when trying to identify compact, lightweight, and performance-optimized options that could support multiple UARTs, IMUs, and communication interfaces. Since our project is limited in weight we need to make sure we are

Question: *"What flight controllers are best suited for a sub-250 g drone?"*

“Here’s a quick breakdown:

- Lumenier Lux Mini F7 V2: Very compact (20×20 mm) and lightweight; F7 processor means future-proof. Excellent for high performance micro builds where every gram counts.

- TBS Lucid Pro F4 Mini: Also 20×20 footprint, F4 processor, designed with plug-and-play in mind (good wiring simplicity).

- Flywoo Goku GN405 Nano V3 F4/ELRS: Nano size, built-in ELRS 2.4GHz receiver option – helpful if you want minimal extra modules.

- iFlight Borg F7 Mini: Slightly larger/more capable micro board; good if you plan to use 3″ props or more aggressive setup while staying under weight.

- Sub250 Redfox A3 F722 35A/45A 4‑in‑1 AIO: A more specialized board labelled “Sub250” – an AIO (flight controller + ESC) which simplifies wiring and saves space/weight — ideal if you’re focused on build simplicity.

- Holybro Kakute F4 V2.4: A trusted “classic” board, slightly larger footprint (30×30 mm maybe) but still feasible for a 3″ build that stays under 250 g with careful selection of other components.

- GEPRC FETtec Alpha G4: More high-end board; may be more capability than needed for ultra-light builds, but useful if you want headroom for more features, telemetry, etc.

- SpeedyBee F405 Mini: Entry level/affordable choice for a micro build; 20×20 footprint, good for beginners staying under 250 g.

My recommendation based on build class

Since you said “sub-250 g drone” (no further specification), here are tailored picks:

- If you are building a very lightweight cinewhoop / 2.5″ or micro FPV (< ~150 g) for fun/travel → choose the ultra-micro boards (Lumenier Lux Mini F7 V2 or Flywoo Goku Nano) to maximize weight savings.

- If you’re building a 3″ or mild 4″ under 250 g, maybe with a decent camera for aerials → you could go with something like the iFlight Borg F7 Mini or Holybro Kakute F4 V2.4 for more head-room (features, sensors) but still keep the whole build light.

- If you want simplicity and minimal wiring → the Sub250 Redfox A3 4-in-1 AIO is very compelling: fewer separate components → less wiring → less chance of wiring weight adding up.” (ChatGPT, 2025)

ChatGPT began by providing the main criteria that mattered when deciding on which flight controller to use. This covered size, weight, power compatibility, processing power, sensor quality, power supply, and regulation. Then provided a comprehensive list that included the SpeedyBee F405 Mini, Lumenier Lux Mini F7, and Flywoo Goku F722 Pro Mini, among others. It also outlined a comparison of microcontroller types (F4 vs F7 vs H7), mounting patterns, and voltage ranges.

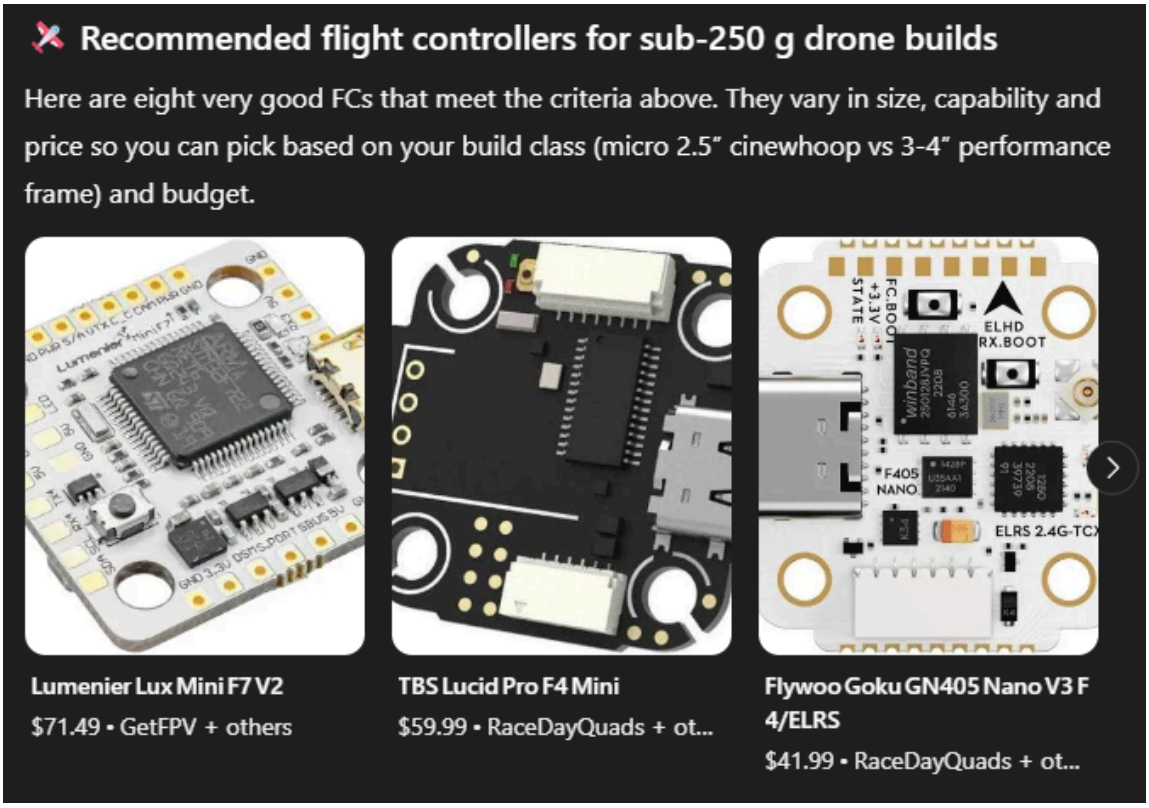


Figure 5.1. *Results of Case Study #1, by ChatGPT*

From the results, the SpeedyBee F405 Mini stood out as the most compatible with our design requirements. ChatGPT noted that it featured an STM32F405 MCU running at 168 MHz, a 4-in-1 ESC connector, and Bluetooth configurability through the SpeedyBee app. All valuable for simplifying assembly and tuning since it stated that it would be good for beginners. Using this information, we verified specifications through the official SpeedyBee datasheet and product manual, confirming details such as its 20 x 20 mm mounting size, 6 UART ports, and 3.3–5 V logic levels.

While ChatGPT provided a strong starting point for flight controller research, it could not substitute for official datasheet confirmation or up-to-date stock and pricing information. However, its structured summaries helped us organize comparison tables quickly and identify

which parameters (MCU series, I/O count, mounting size, and firmware support) mattered most in our selection process.

5.2 Case Study 2

Question Prompt: *What is the best propeller size for a <250g drone?*

> "Short version: for most sub-250 g builds, you'll be happiest in the 2.5–4.0″ range. Pick size by mission:
>
> - Freestyle / agile cruising: 3.0–3.5″ props hit the sweet spot for thrust, control authority, and wind handling while staying under weight. Many sub-250g pilots consider 3.5″ the best all-rounder. Oscar Liang+1
>
> - Cinewhoop / indoors: 2.5–3.0″ (often ducted). Smaller diameter = safer and quieter, but less efficient; use lower-pitch, lightweight props. Oscar Liang
>
> - Endurance / mild long-range (<250 g): 4.0″ with modest pitch can be more efficient (larger disk area → lower disk loading), giving longer flights if the rest of the build is optimized. mdpi.com
>
> How to choose within a size
>
> - Pitch & blades: Start conservative (e.g., 3×3×3 on 3″, 4×2.5×2–3 on 4″). Lower pitch and fewer blades generally improve efficiency and noise; higher pitch/blades add grip and current draw. Oscar Liang
>
> - Match motors/KV: Use community-proven pairings for sub-250g: e.g., 1404 on 3.5″, 1204–1507 on 3″, 1504–1805 on 4″, with KV chosen for your cell count (higher KV for 3S/4S, lower for 4S/6S). FPV Know-It-All - Joshua Bardwell+1
>
> - Weight matters: Lighter props spin up faster and reduce amp spikes—important on tiny rigs. Oscar Liang
>
> Quick picks
>
> - 2.5″ cinewhoop: 2.5×1.9×3 (ducted)
>
> - 3″ general: 3×3×3 (tri-blade) or 3×2.5×2 for efficiency
>
> - 3.5″ freestyle: 3.5×2.8×3 (standard pick)
>
> - 4″ endurance: 4×2.5×2 or 4×2.8×2 (keep pitch modest)

> If you tell me battery (2S/3S/4S), target flight time, and camera payload, I'll suggest exact prop+motor+KV combos with expected current draws."(ChatGPT, 2025)

ChatGPT provided a concise and technically sound breakdown of propeller sizing for sub-250 g drones, dividing its recommendations by use case suggestions like freestyle, cinewhoop, and endurance flight. It suggested an optimal range of 2.5–4.0 inches, noting that 3.0–3.5-inch propellers generally offer the best balance between thrust, control, and efficiency for lightweight builds. The model also explained how pitch and blade count affect flight characteristics: lower pitch and fewer blades improve efficiency, while higher pitch and additional blades increase thrust and responsiveness at the cost of greater current draw.

The most important piece of information ChatGPT offered was the motor-to-propeller pairing and the example of motor types for the size of the propeller. These details helped our team better understand the mechanical and electrical relationships between propulsion components and supported the design of our tri-motor configuration.

5.3 Case Study 3

Question Prompt: *What is the lightest 3D printed material we could use for our drones frame?*

> "Short answer: use a foaming "LW" filament. It's the lightest you can practically 3D-print for airframes.
>
> - LW-ASA (foaming ASA) → effective part density down to ~0.36–0.43 g/cm³ with ~2.5× expansion, plus ~96 °C heat resistance (so it survives sun/motor heat better than PLA). Spectrum Filaments+1
>
> - LW-PLA (foaming PLA) → effective part density about ~0.54 g/cm³ and up to ~65% weight reduction vs regular PLA. Widely used for RC aircraft. esun3d.com+2colorfabb.us+2
>
> By comparison, common light filaments without foaming are heavier: polypropylene ~0.90 g/cm³, PA12/nylon ~1.0–1.2 g/cm³ (CF-filled is stiffer/stronger but not lighter). Formlabs+2Bitfab+2
>
> What I'd do for a sub-250 g frame
>
> - Primary shells/ducts/skins: LW-ASA (best strength-to-heat for ultralight prints). If you don't need heat resistance, LW-PLA is easier. colorfabb.us+1
>
> - Load-bearing bits (motor mounts/arms): reinforce with CF tubes/rods or print those parts in PA12-CF/nylon-CF and keep the rest foamed. (CF nylon isn't lighter, just much stiffer.) sunlu.com
>
> Print tips (ultralight, but strong enough)

- Start at ~40–60% flow with LW-ASA/PLA (where foaming peaks) and tune temperature per vendor to hit target weight. colorfabb.us+1

- Use thicker walls (≥0.8–1.0 mm), low infill (≤10–15%), generous fillets, and embed CF rods in arm channels for rigidity (glue or tie-wrap).

- Keep parts out of direct motor exhaust if using LW-PLA (heat-softening risk); prefer LW-ASA for sunny/hot conditions. colorfabb.us

If you want, tell me your frame size (2.5″/3″/4″) and target battery, and I'll sketch a weight budget plus suggested wall/infill settings for each part." (ChatGPT 2025)

After asking this question, ChatGPT provided a detailed breakdown of the lightest 3D-printed materials suitable for drone airframes. It recommended LW-ASA and LW-PLA foaming filaments, explaining the practical use for drone frames.

However, while ChatGPT presented accurate general data, it did not provide direct manufacturer comparisons or cost estimates, which had to be verified manually through vendor sites like ColorFabb, eSUN, and Spectrum Filaments that it provided in the prompt.

"The lightest 3D-printed material suitable for a drone frame is Lightweight PLA (LW-PLA or PLA Aero), which uses active foaming technology during printing to reach a density of around 0.54 g/cm³, roughly half the weight of standard PLA. While LW-PLA provides the lowest weight, a drone frame also requires sufficient strength and durability, particularly impact resistance. Therefore, the optimal material choice involves balancing low mass with good mechanical properties.

Lightweight PLA (LW-PLA) is ideal when minimizing weight is the primary goal, such as in model aircraft or micro-drone applications. Its foaming property expands the filament during extrusion, allowing for reduced material usage and adjustable density based on temperature and flow settings.

Carbon-fiber-reinforced composites such as Nylon-CF or PA12-CF offer an excellent strength-to-weight ratio and are considered the standard for high-performance drone frames. These materials provide exceptional stiffness, impact resistance, and structural integrity while remaining lighter than traditional plastics, though they are denser than LW-PLA.

Polypropylene (PP) is another lightweight option, with a natural density of about 0.9 g/cm³ and good impact resistance. However, it presents printing challenges due to its tendency to warp and its limited bed adhesion.

In summary, LW-PLA offers the lowest possible weight, carbon-fiber composites provide the highest strength, and polypropylene serves as a balanced alternative when durability and toughness are priorities."(Google, 2025)

Additionally, when the same prompt was tested using Google AI, the output came to a similar result, focusing on the same outcome but a few different other comparisons (LW-PLA, PP, and PA12). ChatGPT's response offered more engineering depth, specifics, actionable detail, and credible links to extra information. Whereas Google AI's result was more surface-level but linked to live product pages, making it better for verifying availability and pricing.

5.4 Case Study 4

Question Prompt: *What are some prices on 1404 motors?*

> "If you're shopping for 1404-size drone motors (popular for sub-250 g builds), here are some current price ranges and tips on what affects the price.
>
> And here's a quick summary of each:
>
> - Flywoo Nin 1404 V2: ~$19.99 (some deals ~$16).
> - Sub250 1404 4500KV Motor: ~$15.99 on sale.
> - SpeedyBee 1404 V2 4600KV Motor: ~$12.99 (one of the lowest).
> - T-Motor F1404: around $18-22.
> - GEPRC SpeedX2 1404: ~$19.99.
> - Brotherhobby TC 1404: ~$22.49 (some discounts bring it lower ~15).
> - RCInPower AOS RC Supernova 1404: ~$18.99 (promoted).
> - iFlight XING2 1404: ~$23.49 (some shops show ~20.99).
>
> What affects the price
>
> Here are the main factors that cause variation:
>
> - Brand & build quality: Premium brands (T-Motor, iFlight) cost more due to materials, bearings, durability.
> - Stator size / KV / voltage rating: Lower KV motors built for long-range/efficiency tend to cost more; very high KV for tiny props might cost less.
> - Weight / materials / features: Ultralight builds (e.g., <9g weight) often use exotic materials and cost more.

- Support & availability: Regional distribution, warranties, and shipping can affect price.

- Sale / stock levels: Many retailers drop to ~$12-15 for good deals. E.g., on one marketplace many 1404s ranged $15-25. eBay+1

- Intended use / specs: If the motor is labeled for 4-6S, heavy duty, etc., price may be higher. Example: the HGLRC AEOLUS 1404 was ~$24.89 from ~$33.99. MyFPVStore.com

My recommendation for your build (<250 g)

Since you're building a sub-250 g drone (which typically means lighter props, smaller motors, efficiency prioritized), you don't necessarily need the highest-cost premium motor. A well-built 1404 in the ~$15-20 range is very good, as long as you check thrust, efficiency, and weight.

If I were selecting one now: I'd pick something in the ~$16–22 range that has good reviews, low weight (<9g), and is matched to your prop size (e.g., 3″ or 3.5″) and battery (2S/3S/4S) for best results." (ChatGPT, 2025)

ChatGPT is able to give me a wide range of 1404 motors that would be useful in our project design. Even providing links to price check that the values are accurate. A few years ago it would have been better to do my own research for this product since ChatGPT was not as reliable. Now it is able to cut down on the research time required for searching up something you may not know much about. This quickly gave me eight different options with all of their represented prices. While also including the main factors between the variance of prices for each motor.

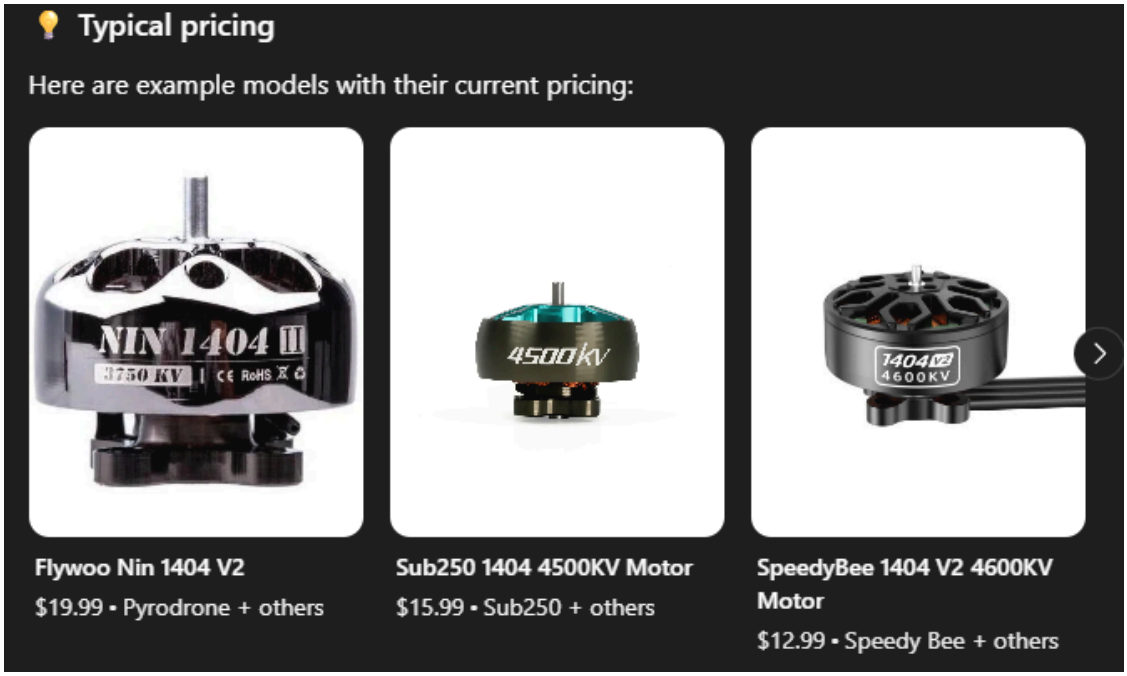


Figure 5.2. *Results for Case Study #4, by ChatGPT*

Also the prompt gave me links and example models of each and every 1404 motor that it referenced. As you can see in Figure 5.2, if you click the arrow to the right it would proceed to give me all eight of the motors listed below.

5.5 Case Study 5

ChatGPT was one of the most valuable tools used throughout the research and documentation process of our drone project. Because a large portion of this report focuses on identifying and analyzing technical components. ChatGPT helped bridge gaps in our understanding and guided us toward reliable starting points for deeper investigation into what was going on for this project. It was especially helpful early in the semester when we were exploring new topics such as flight controllers, motor controllers, sensor technologies, and communication protocols.

When deciding on the best microcontrollers, ChatGPT provided a clear explanation of their fundamental differences, advantages, and trade-offs. It helped us determine which microcontroller was more practical for our sub-250 g drone due to lower power consumption, smaller form factor, and easier integration with existing drone firmware such as Betaflight. The ability to summarize complex concepts and provide concise comparison criteria made the decision process more informed and structured.

While ChatGPT's explanations were accurate at the conceptual level, it occasionally produced information that did not pertain to our project. Or giving us results that ended up not being useful for the type of drone we were developing. To resolve this, we cross-checked all hardware details using manufacturer datasheets and verified component availability through vendors like Digi-Key, Mouser, and GetFPV.

Where ChatGPT truly excelled was in organizing information for sensors, communication systems, and frame materials. It helped us identify the most common technologies used for telemetry (such as UART and I2C) and suggested multiple options for distance sensing, including ultrasonic, LiDAR, and time-of-flight (ToF) sensors. Similarly, when researching lightweight frame materials, ChatGPT introduced us to foaming filaments like LW-PLA and LW-ASA, which were later validated through additional research and matched the weight goals of our design.

ChatGPT also proved useful when writing technical sections of the report. It assisted in finding out protocols for certain things like (UART and I2C). However, other types of AI like Google AI were not needed. AI is so advanced now and ChatGPT was able to pretty much give us any information we needed. ChatGPT used to remain more effective for technical reasoning and background research but it was also able to give us plenty of accurate pricing details, as well as locations on where to purchase the product.

Overall, ChatGPT offered tremendous value as a research assistant. It helped us narrow our scope, refine our documentation, and accelerate technical understanding across multiple areas of the project. Although not always perfect in factual precision, its structured responses and ability to simplify complex subjects made it an indispensable resource for the research and development of our drone

5.6 Case Study 6

Throughout the course of our project, we found that ChatGPT alone was sufficient for nearly all of our research and technical planning needs. While there are other AI tools available, such as Google AI, Copilot, or Claude, we rarely found the need to use them. ChatGPT consistently provided the information, explanations, and comparative insight required for each stage of our design process. We did not run into many issues where ChatGPT was unable to give us credible information. It gave us plenty of resources to back up the information it provided.

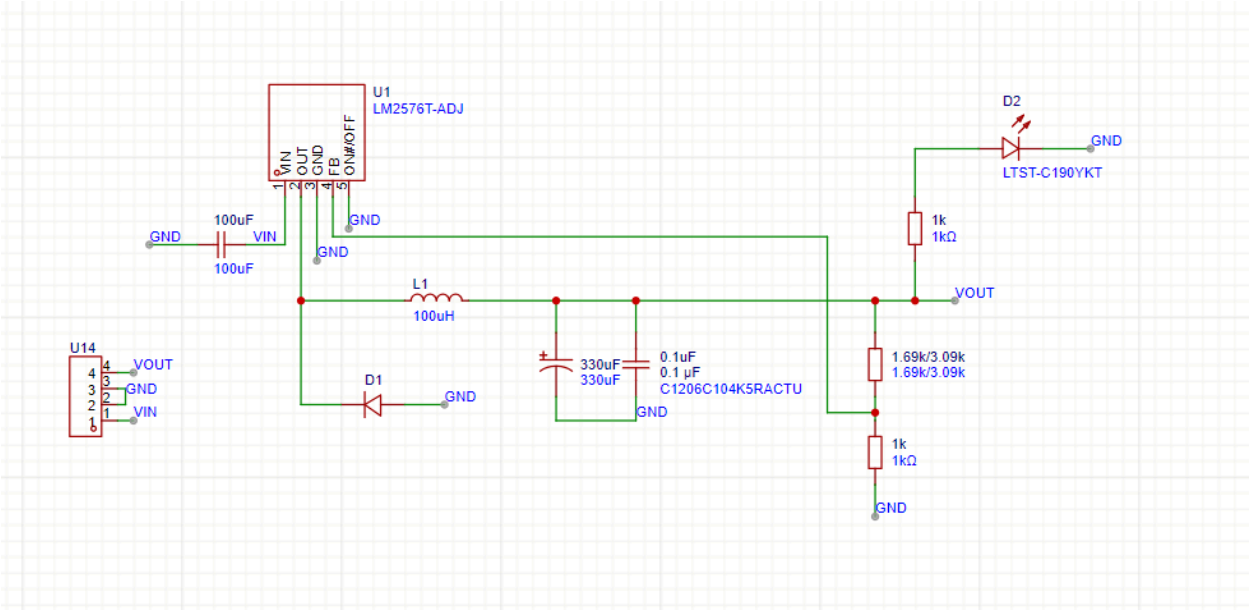
From early component research to understanding firmware compatibility and evaluating design trade-offs, ChatGPT offered detailed and well-structured responses that guided our decision-making. It provided both theoretical explanations and practical context, making it easy to identify viable components, understand their specifications, and cross-reference them with manufacturer datasheets.

In areas where accuracy was critical, such as voltage ratings, current limits, or weight estimations, we supplemented ChatGPT's responses with datasheet verification rather than relying on alternative AI systems. The model's ability to explain concepts clearly and organize technical information proved more valuable than tools that simply listed products or linked search results.

Ultimately, ChatGPT served as a single, centralized research platform that simplified the information-gathering process for our team. Its balance of accessibility, depth, and clarity allowed us to focus more time on design and testing rather than searching through fragmented sources. Because of this, our team did not find a strong need to rely on any other AI model for the completion of this project.

Chapter 6. System Hardware Design

6.1 Voltage Regulators



6.1.1 Overview

The power regulation and distribution network forms the electrical backbone of the Dizzy n-Copter. Its purpose is to convert and stabilize the voltage supplied by the lithium-polymer (Li-Po) battery to multiple regulated levels required by the onboard subsystems, including the microcontroller, servos, sensors, and telemetry components. The circuit ensures safe, efficient, and noise-free operation while remaining lightweight and compact to meet the sub-250 g flight constraint.

6.1.2 Circuit Description

We implemented both the 3.3 V and 5 V power rails using the **LM2576T-ADJ**, taking advantage of its adjustable output capability and high current handling. This device is a switching (buck) regulator, which steps down a higher input voltage efficiently compared to linear regulators. In our design, a nominal 12 V battery input is reduced to the required output voltages by configuring the regulator with an external feedback resistor divider. By simply adjusting the ratio of the resistors connected to the feedback (FB) pin, we were able to reuse the same circuit topology for both voltage rails, minimizing design complexity and allowing for a consistent PCB layout across both regulator implementations.

The regulator circuit includes all standard supporting components recommended by the datasheet. An input capacitor stabilizes the supply and reduces ripple from the battery, while the inductor and catch diode form the core of the buck conversion stage, enabling efficient energy transfer during switching. On the output side, a combination of bulk capacitance (330 µF) and a

smaller ceramic capacitor (0.1 µF) helps smooth the output voltage and reduce high-frequency noise. The feedback network sets the output voltage precisely to either 3.3 V or 5 V, depending on the resistor values selected. Additionally, an indicator LED tied to the output provides a simple visual confirmation that the regulator is functioning correctly.

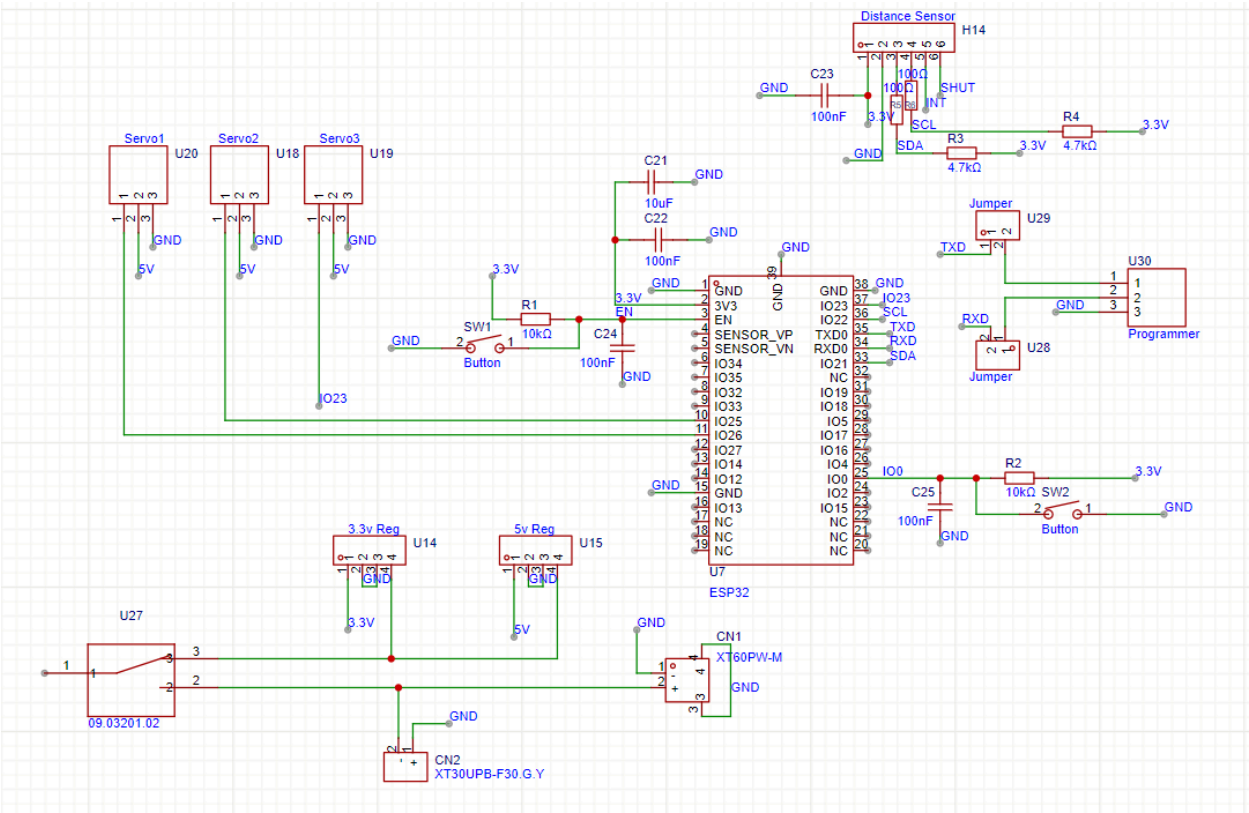
One key advantage of this approach is flexibility and scalability. Because the **LM2576-ADJ** supports up to 3 A of output current, it provides sufficient headroom for both the ESP32 (3.3 V rail) and peripheral loads such as servos (5 V rail). Using the same regulator IC for both rails also simplifies sourcing and assembly, while ensuring consistent performance characteristics. Overall, this design balances efficiency, reliability, and simplicity, making it well-suited for our embedded system application.

6.1.3 Functional Summary

Sub-Circuit	Primary Function	Voltage Domain	Key Components
Input Protection	Reverse-polarity and over-current protection	VBAT	P-MOSFET, PTC fuse
Buck Converter	Step-down 7–12 V → 5 V	+5 V_MAIN	LM2576HV, L1, D1, CIN/COUT
LDO Regulator	Step-down 5 V → 3.3 V	+3.3 V	LD1117-3.3 V, CIN/COUT
Motor Power Switch	MCU-controlled gating	+5 V_MOTOR	P-MOSFET, R5
Status Indicators	Visual power feedback	5 V and 3.3 V	LED1, LED2

This power stage provides high efficiency and fault tolerance while maintaining signal integrity for the sensitive flight electronics.

6.2 ESP32 Core Subsystem



6.2.1 Overview

The ESP32 serves as the primary control unit for the Dizzy n-Copter, integrating computation, communication, and sensor coordination. This subsystem includes the power interface, reset and boot circuitry, and essential decoupling networks to ensure stable operation during power transients and RF communication bursts.

6.2.2 Circuit Description

As shown in *Figure 6.2*, the ESP32 module operates from the +3.3 V rail generated by the LDO regulator. A dedicated 10 µF and 0.1 µF capacitor pair is placed close to the VCC pin to mitigate switching noise. The EN (enable) and IO0 (boot) pins are tied through pull-up resistors to ensure proper boot configuration, with momentary push buttons providing manual reset and programming mode entry.

The UART interface pins (TXD0, RXD0) are labeled to connect directly with the USB-to-serial converter on the programming board. Additional GPIOs are allocated for PWM motor control, servo actuation, ultrasonic echo/trigger, and telemetry sensors.

This circuit ensures reliable startup, low-noise logic operation, and accessible development interfaces for firmware uploading and debugging.

6.2.3 Functional Summary

Signal/Node	Function	Key Components
+3.3 V Power	MCU supply rail	LM2576 output, 10 µF + 0.1 µF decoupling
EN (Enable)	System reset control	Pull-up resistor, push button
IO0 (Boot)	Flash programming mode	Pull-up resistor, push button
TXD0/RXD0	Serial interface	UART communication lines
GPIO Pins	PWM, I²C, ultrasonic, telemetry control	ESP32 GPIOs 2–23

This design provides a stable and configurable core for flight control and data acquisition.

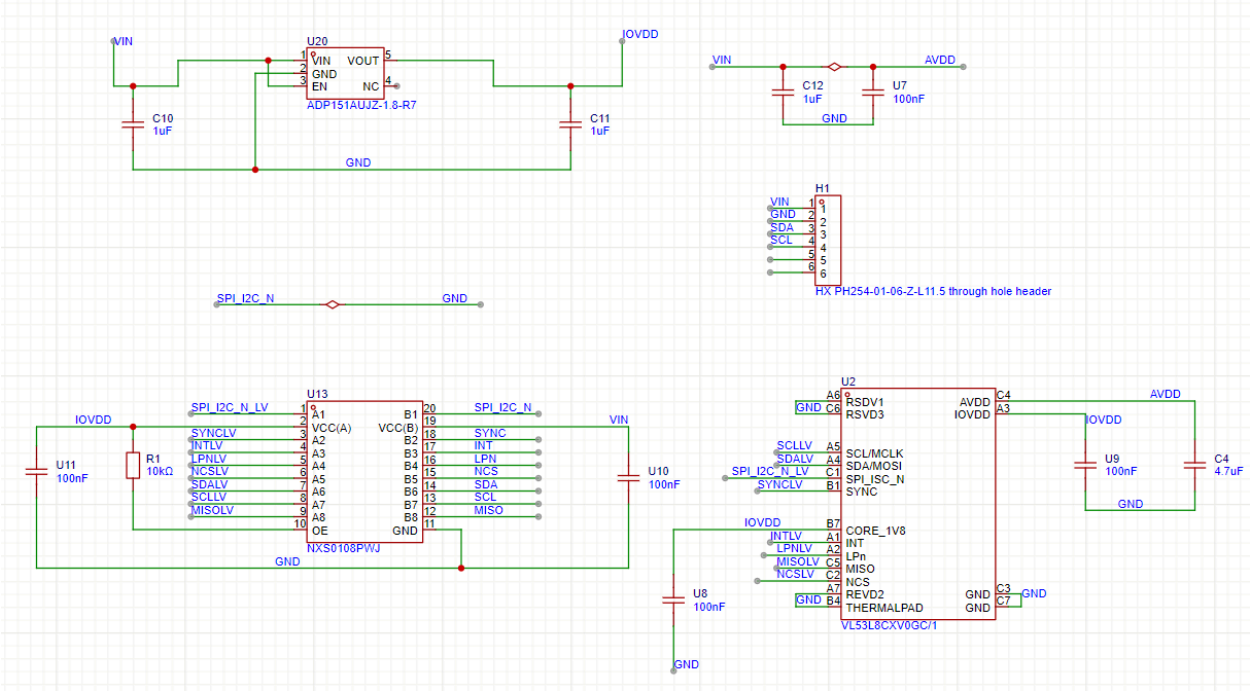
6.2.4 Overview

The electronic speed controller (ESC) developed in this project integrates a custom-designed three-phase power stage with an ESP32 microcontroller to create a modular and programmable motor control solution for multirotor applications. The power section is based on the architecture of STMicroelectronics' STEVAL-ESC001V1, utilizing L6398/L6388 half-bridge gate drivers and six N-channel MOSFETs to generate a robust three-phase output capable of driving brushless DC motors efficiently. A dedicated auxiliary buck regulator (L7986TR) provides a stable 10 V rail for the gate drivers, while a 5 V BEC and local 3.3 V regulator supply the ESP32 logic. The ESP32 replaces the conventional STM32 found in typical ESCs and is responsible for decoding throttle commands from a SpeedyBee F7 V3 flight controller, which provides individual PWM control signals for each motor. Using its MCPWM peripheral, the ESP32 generates complementary high- and low-side PWM signals with precise dead-time insertion to drive the gate drivers, implementing fully customizable 3-phase commutation. Each ESC board operates independently and powers a single motor, resulting in a scalable design where multiple identical ESC modules can be deployed across a multirotor system. This architecture combines

the reliability of a proven hardware power stage with the flexibility of open firmware development, offering a platform suitable for experimentation in custom motor control strategies

6.3 Distance Sensor

6.4.1 Overview



The sensor and telemetry interface connects the ESP32 to the environmental sensors and communication peripherals that provide critical flight data such as temperature, pressure, or battery voltage telemetry.

6.4.2 Circuit Description

For the distance sensing portion of the design, we used the **VL53L8CX** as the core component due to its high accuracy and ability to provide multi-zone ranging data. This sensor operates using a time-of-flight (ToF) principle, where it emits infrared light and measures the return time to determine distance. To properly support the device, we designed the surrounding circuitry to meet its strict power and communication requirements. The sensor requires separate supply domains, including AVDD for analog operation and IOVDD for digital I/O, ensuring clean signal integrity and reliable measurements.

To generate the required lower voltage rail for the sensor core, we implemented a dedicated LDO regulator using the **ADP151AUJZ-1.8**. This regulator steps down the system voltage to 1.8 V, which is used internally by the sensor. Proper decoupling capacitors were placed at both the input and output of the LDO to minimize noise and ensure stable operation. Additional bypass

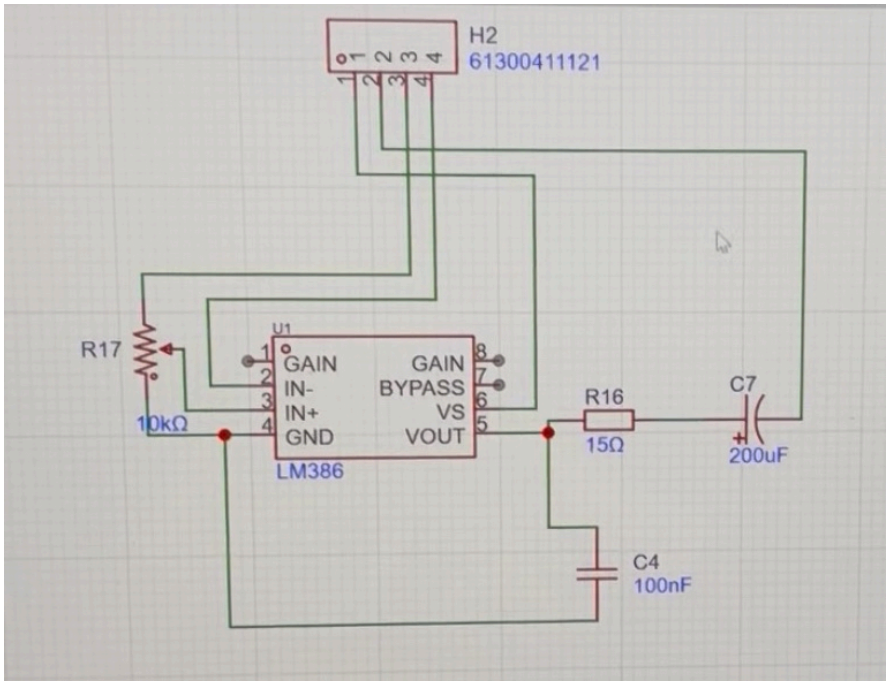
capacitors were also placed close to the sensor's AVDD and IOVDD pins to filter out high-frequency noise and maintain voltage stability, which is critical for accurate distance measurements.

Communication between the sensor and the microcontroller is handled through an I²C interface, with optional SPI capability available. Because the sensor operates at a lower logic level than the ESP32, we included a bidirectional level shifter using the **NXS0108PWJ**. This allows safe voltage translation between the 3.3 V microcontroller domain and the sensor's lower voltage domain. Supporting signals such as interrupt (INT), shutdown (LPn), and chip select (NCS) were also routed appropriately to give full control over the sensor's operation. Overall, this design ensures proper power regulation, signal integrity, and communication compatibility, enabling reliable and accurate distance sensing within the system.

6.5 Audio amplifier addition

Adding this gave us some queues and sound for when our drone was or was not working properly. Our system supplies a limited amount of voltage to this, which then is amped up to supply the needed output to an audio device. The circuit shown is based on the LM386 low-voltage audio amplifier, which is designed to amplify weak analog signals into a level sufficient to drive a small speaker or similar load. In this configuration, the input signal is received through the H2 header and routed through a 10 kΩ potentiometer (R17), which functions as a variable voltage divider for volume control. By adjusting the potentiometer, the amplitude of the signal fed into the non-inverting input (Pin 3) of the LM386 can be regulated, allowing control over the output sound level. The LM386 itself operates as the main amplification stage, powered through Pin 6 (VS) with Pin 4 connected to ground, and internally amplifying the input signal to produce a higher-power output at Pin 5.

To ensure stable operation and reduce high-frequency oscillations, a Zobel network composed of a 15 Ω resistor (R16) and a 100 nF capacitor (C4) is connected at the output stage. This network helps maintain amplifier stability when driving reactive loads such as speakers by damping high-frequency noise and preventing unwanted oscillations. Additionally, a 200 µF coupling capacitor (C7) is placed in series with the output to block any DC component from reaching the load while allowing the amplified AC audio signal to pass through. This protects the connected speaker and ensures that only the intended audio waveform is delivered. Overall, the circuit provides a simple but effective low-power audio amplification system with adjustable gain, stable output behavior, and DC-safe signal coupling suitable for small embedded audio applications.

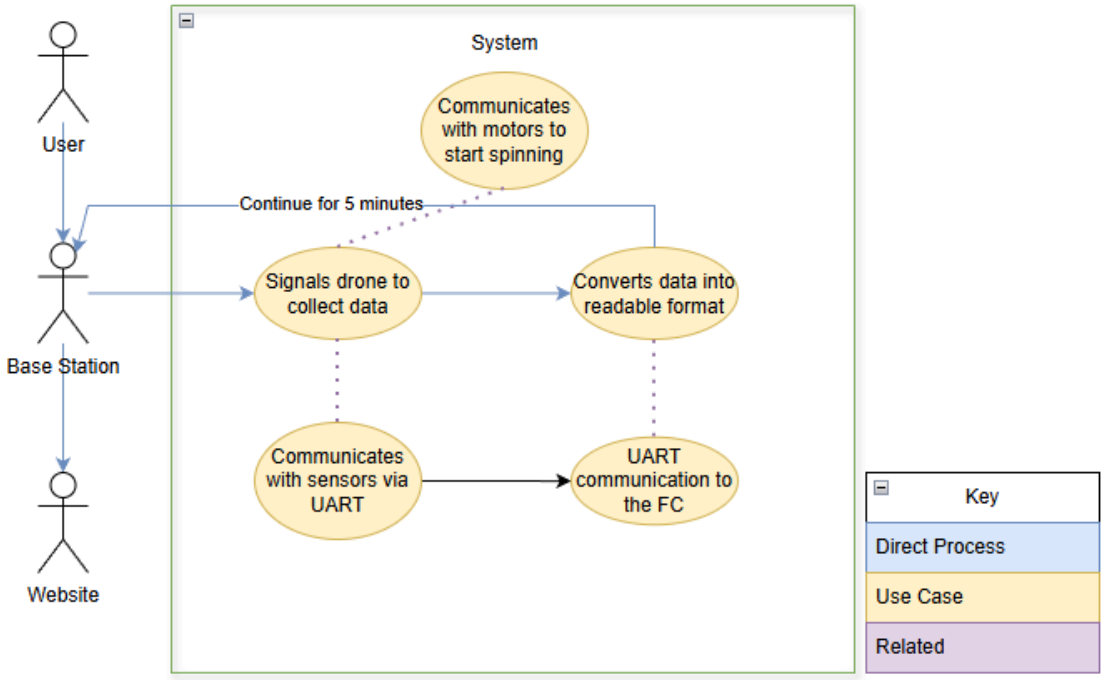


6.6 System Integration Overview

All three hardware subsystems integrate into a cohesive electrical design supporting safe, stable, and efficient drone operation. The power distribution board ensures reliable supply to each domain, while the ESP32 core handles command logic and communication. The servo hub and sensor interfaces are modularly designed for easy maintenance and scalability. Together, these circuits achieve a balance of energy efficiency, weight minimization, and control precision essential for the Dizzy n-Copter’s flight performance.

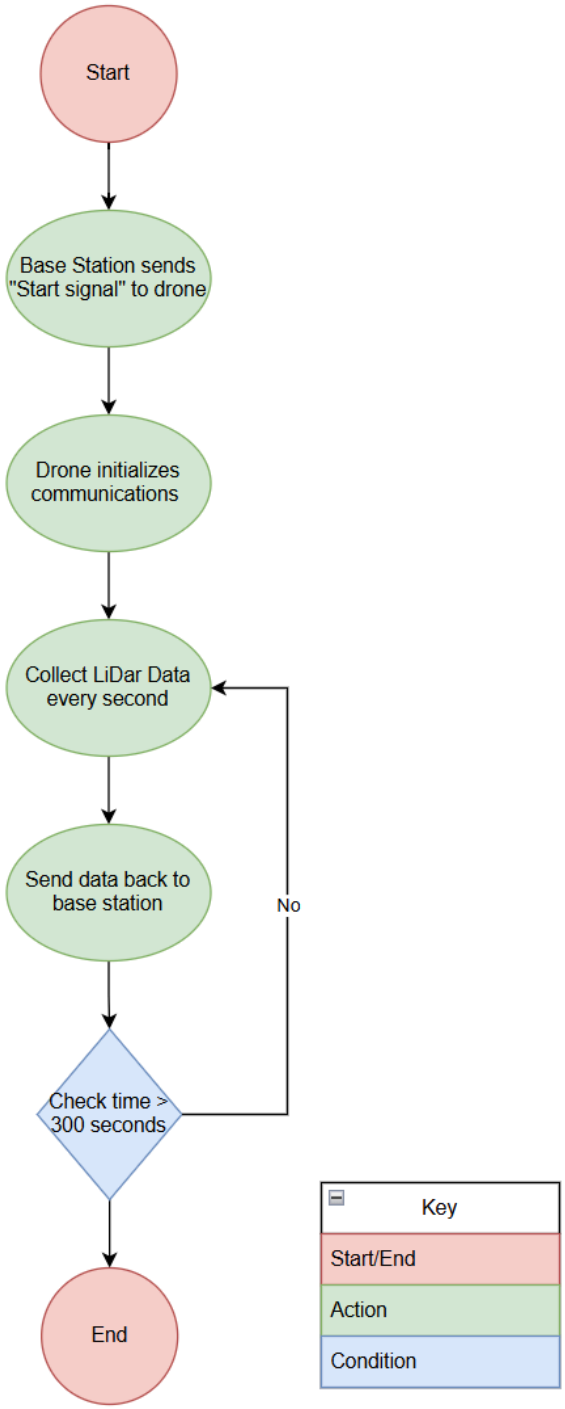
Chapter 7 - Software Design

7.1 Software Use Case Diagram



The system begins when the Base Station signals the drone to start collecting data. The drone responds by activating its motors to begin spinning and initiates data collection from onboard sensors via UART communication. The collected sensor data is then transferred to the flight controller, where it is converted into a readable format. This process continues for approximately five minutes while the drone streams information back to the Base Station. The Base Station subsequently forwards the processed data to the website, where a user can view the results. Throughout this workflow, the user primarily interacts with the system through the Website interface, while the Base Station manages real-time communication with the drone.

7.2 Activity Diagram



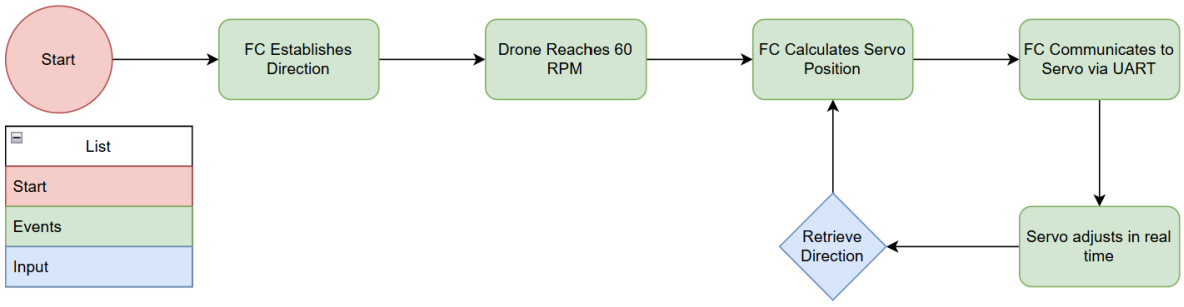
At startup, the drone will remain stationary on the ground while performing all required initialization procedures, including sensor calibration, wireless link verification, and system health checks. After confirming that all subsystems are operating within safe parameters, the

drone will ascend to its designated hover altitude without rotating. Once stable, it will begin a controlled yaw rotation at approximately 60 RPM for a duration of five minutes. During this rotation, the onboard sensors will continuously collect environmental data, which will be transmitted to the base station in real time at one-second intervals. The acquired data will be processed to generate a spatial representation of the drone's surroundings, ultimately producing a visually interpretable map or image accessible through a web interface. This interface will allow external users to monitor the drone's measurements and visualize the mapped environment in near real time.

The Dizzy n-Copter's onboard software acts as the central control system. Coordinating all flight operations, sensor readings, and communication tasks. It manages the three brushless motors, tilt servos, and onboard sensors, ensuring stable flight during the drone's continuous 60 RPM rotation. The software architecture is modular, with subsystems handling flight stabilization, telemetry, motor control, and power management. These modules communicate through standard digital interfaces such as I2C, UART, and PWM, all managed by the STM32-based flight controller.

The control firmware is built on a modified version of Betaflight, optimized for continuous rotation and thrust-vectoring control. It continuously reads data from the IMU and sensors, processes it through PID control algorithms, and generates corresponding PWM outputs for each motor and servo. The software also includes telemetry reporting, safety monitoring, and fault detection routines that ensure stable communication with the ground station and reliable system performance. Together, these elements provide a stable and efficient software foundation that allows the Dizzy n-Copter to maintain controlled flight and responsive maneuvering even while the frame is in constant motion.

7.3 Forward Translation With Directional Stability



This project goal involves adding semi-autonomous forward motion while the drone maintains its controlled rotational scanning behavior. During this mode, the drone would continue spinning at a controlled rate of approximately 60 RPM to collect 360° environmental data; however, it would also introduce a deliberate tilt about the Z-axis (yaw orientation reference) to generate directional thrust and achieve forward translation.

This behavior requires the flight system to manage rotational motion and translation simultaneously. The drone would continually monitor its attitude through onboard kinematic calculations combined with IMU and other sensor inputs. By dynamically stabilizing orientation while adjusting tilt, the system would maintain a consistent net direction of travel even though the craft is continuously rotating.

The algorithm guiding this behavior will be responsible for maintaining the drone's overall orientation in space, ensuring that rotation does not interfere with the intended forward trajectory. This will require real-time estimation of heading, rotational velocity, and positional drift. The control logic will adjust motor outputs to generate the appropriate tilt angle to move forward while compensating for aerodynamic disturbances and maintaining a stable spin.

The proposed approach preserves the primary mapping mechanism, high-frequency environmental data collection during rotation, while expanding the capability of the drone to cover larger regions. Forward translation allows the system to create extended spatial maps rather than a single stationary scan. Data gathered during motion can be processed and fused to generate increasingly comprehensive 3D representations of the surrounding environment, enabling enhanced visualizations and real-time updates on the ground station.

7.4 Real-Time Data Transmission via BLE

The drone communicates directly with a companion mobile application over Bluetooth Low Energy (BLE), eliminating the need for an intermediate base station or cloud infrastructure. The ESP32 microcontroller onboard the drone acts as a BLE peripheral, hosting a custom GATT service that exposes multiple characteristics for sensor data, flight state, and control commands.

Telemetry is streamed continuously via BLE notifications across seven characteristics: floor distance (VL53L8CX ToF sensor averaged across floor zones), battery voltage and current, attitude (pitch, roll, yaw derived from CRSF telemetry), flight mode, and a 72-byte lidar map encoding 36 directional sectors built up as the drone spins. All characteristics use little-endian binary encoding rather than text, minimizing packet size and parse overhead on the receiving end.

The mobile application, built with Flutter, connects to the ESP32 by scanning for the custom service UUID and subscribing to BLE notifications on each characteristic. The application maintains in-memory history buffers for all telemetry streams, capped at 2000 samples, which feed a suite of real-time visualizations accessible through a slide-in telemetry drawer. These include a 360° polar wall map that builds up sector by sector as the drone rotates, a full-session floor distance line graph with a live drone position indicator, and time-series charts for battery, attitude, and RPM.

Control commands flow in the opposite direction: the app writes to writable BLE characteristics to toggle drone power, hover mode, and spin mode. The ESP32 processes these writes via characteristic callbacks on the NimBLE stack, which runs pinned to Core 0, while flight-critical tasks — CRSF output, altitude hold, and sensor reading — run pinned to Core 1 under FreeRTOS, coordinated through mutexes to prevent data races across the shared state.

To handle BLE disconnection gracefully, the ESP32 implements a 1.5-second grace period during which a reconnecting client can cancel an in-progress emergency landing. On the application side, all telemetry history is cleared on disconnect so each session begins with a clean state. This direct peer-to-peer BLE architecture avoids cloud latency entirely, delivering sensor updates to the UI at approximately 5Hz with no intermediate hops.

7.5 Comparison of Backend and Frontend Tool Options

7.5.1 Backend Frameworks

Framework	Language	Pros	Cons
FastAPI	Python	Very fast; easy async support; simple API development; strong typing	Less built-in structure than Django
Django	Python	Very mature; built-in admin + ORM; strong ecosystem	More heavyweight; less ideal for real-time updates
Express.js	JavaScript	Lightweight; highly flexible; widely supported	Less structured; requires more boilerplate
Go + Gin/Fiber	Go	Very fast; compiled; excellent concurrency	Smaller library ecosystem
Flutter	Dart	Fast; Strong BLE ecosystem	Niche language

7.5.2 Front-End Technologies

Library / Framework	Use Case	Pros	Cons

React.js	Full UI	Mature; modular; many examples	Slightly steep learning curve
Vue.js	Full UI	Simpler than React; good documentation	Smaller ecosystem
Chart.js	2D Charts	Simple; great for line/polar/bar graphs	Limited customization
D3.js	Custom charts/maps	Very flexible; powerful visuals	High complexity
Flutter	Dart	Fast; Strong BLE ecosystem	Niche language

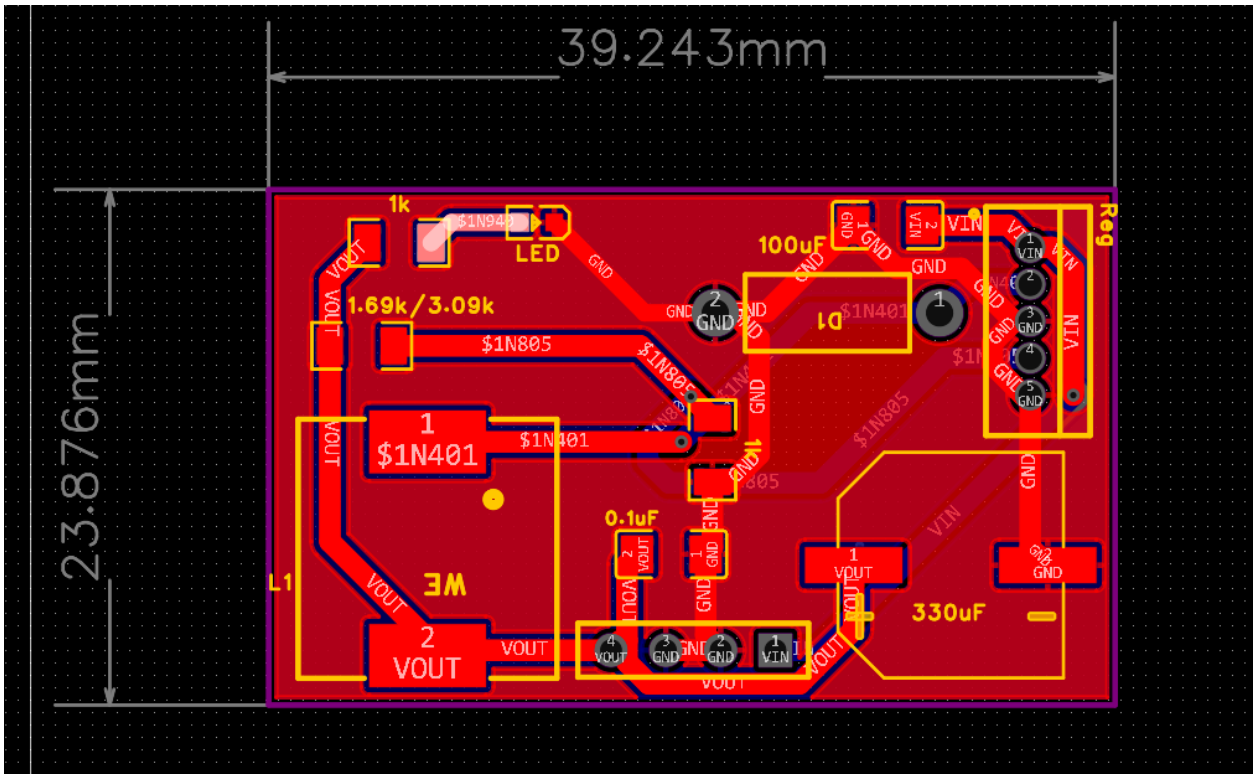
Chapter 8 - System Fabrication/Prototype Construction

8.1 Regulator PCB

For the regulator PCB, we designed a compact and efficient layout centered around the **LM2576T-ADJ**, with careful attention to high-current paths and switching behavior. Because this is a switching regulator, layout is just as critical as the schematic. We kept the input capacitor, regulator IC, catch diode, and inductor tightly grouped to minimize loop area, which helps reduce noise, voltage ripple, and EMI. The high-current traces, particularly those carrying VIN, VOUT, and the switching node, were made significantly wider to safely handle the expected current while minimizing resistive losses and heating.

The PCB also incorporates a solid ground pour across both layers to provide a low-impedance return path and improve thermal performance. Key components such as the diode and regulator were placed with short, direct connections to ground to reduce switching noise. Decoupling capacitors, including the 100 µF input capacitor and the 330 µF output capacitor, were positioned as close as possible to the regulator pins to ensure stable operation. A smaller 0.1 µF ceramic capacitor was added near the output to filter high-frequency noise, further improving output voltage quality.

Additionally, the board was designed with usability and flexibility in mind. Clearly labeled VIN, VOUT, and GND header pins make integration with the rest of the system straightforward, while the onboard LED indicator provides a quick visual confirmation of operation. The feedback resistor network is placed close to the feedback pin to maintain accuracy and reduce susceptibility to noise. Overall, this PCB layout balances compact size with proper power design practices, ensuring reliable performance for both the 3.3 V and 5 V regulator configurations.



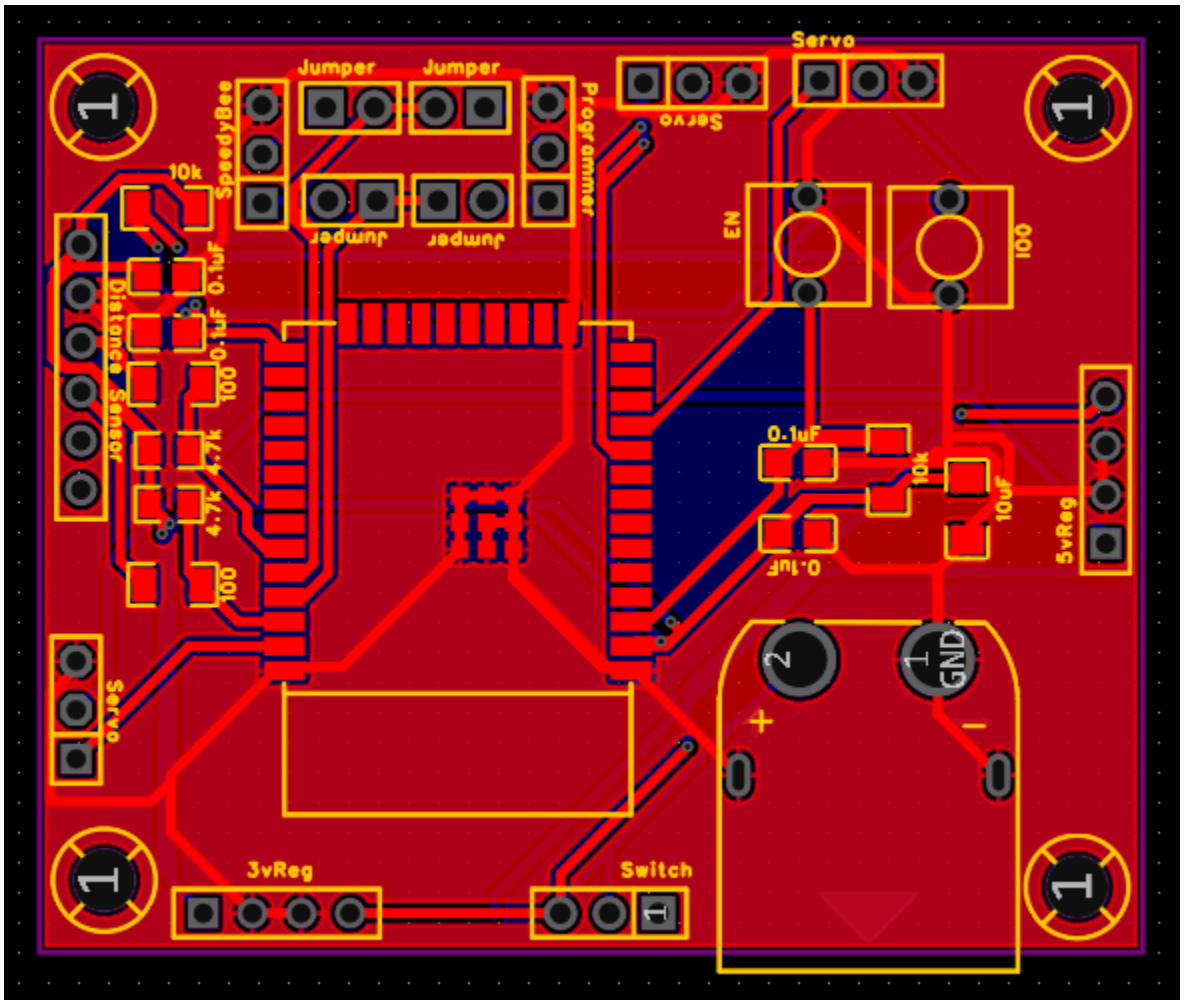
8.2 Communication PCB

For the communication board, the layout was designed to reliably interface the microcontroller with external modules while maintaining clean signal integrity and organized routing. At the center of the design is the **ESP32**, which handles all communication, control, and data processing. The PCB was arranged to keep the ESP32 and its supporting components as compact as possible, while still allowing clear breakout access to important pins such as UART, I²C, SPI, and control signals. This makes the board flexible for debugging, programming, and integration with other subsystems like the flight controller and sensors.

Power distribution on this board was carefully managed to ensure stable operation. A dedicated 5 V input feeds into the onboard regulation stage, which then supplies the 3.3 V rail required by the ESP32 and associated logic. Decoupling capacitors were placed close to the power pins of the ESP32 to filter noise and stabilize the supply, especially important given the switching activity from the communication peripherals. Wide traces and ground pours were used for power

routing to minimize voltage drop and improve overall efficiency, while also providing a solid reference plane for high-speed signals.

Signal routing was organized to separate power and communication paths, reducing the likelihood of interference. Headers labeled for connections such as “SpeedyBee,” “Sensor,” and general-purpose jumpers allow straightforward interfacing with external boards. Critical control signals like EN and IO0 were broken out and paired with appropriate pull-up and control circuitry to support reliable boot and programming behavior. Additionally, the board includes clearly defined connectors for switches and UART communication, enabling easy interaction during testing and deployment. Overall, the communication board layout emphasizes modularity, clean routing, and ease of integration, ensuring dependable communication across the entire system.

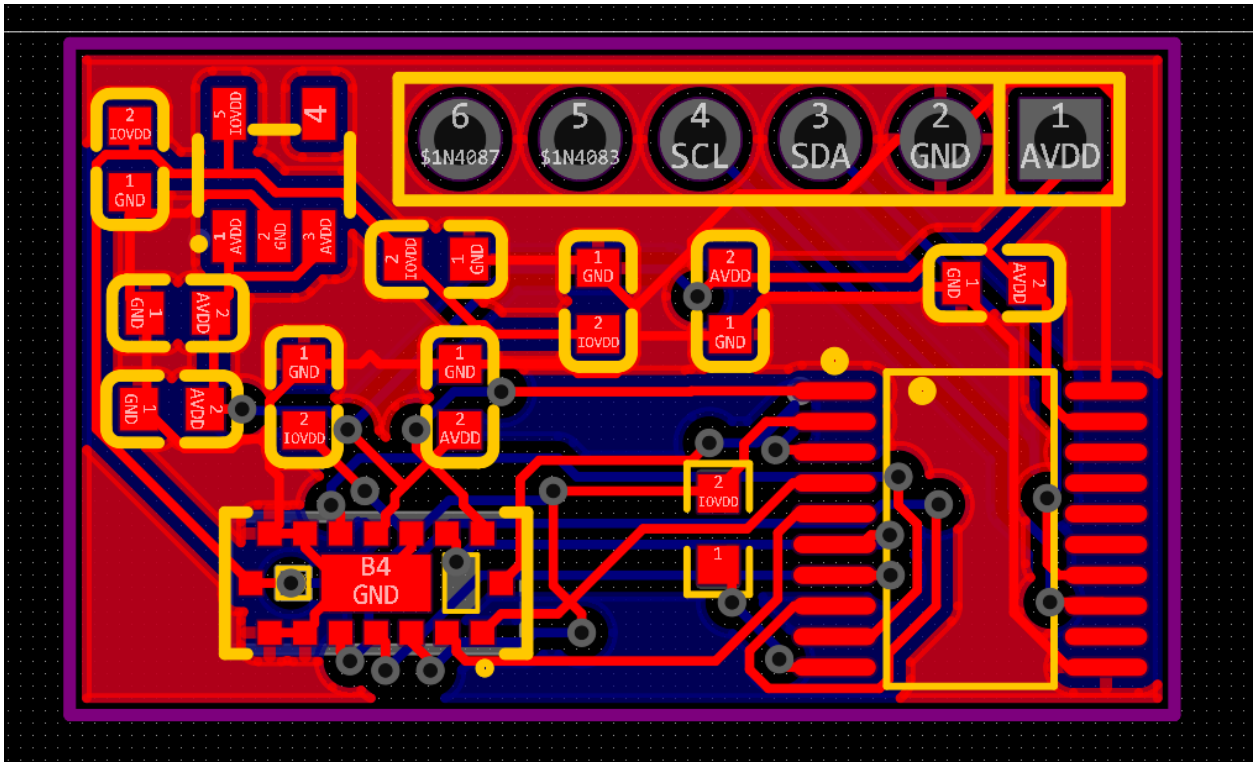


8.3 Distance Sensor PCB

For the distance sensor PCB, the layout was carefully designed to support accurate and low-noise operation of the **VL53L8CX**. Since this device relies on precise optical measurements, special attention was given to both the electrical layout and the physical placement of the sensor. The sensor itself was positioned with a clear, unobstructed area in front of it to avoid interference with the emitted and reflected infrared signals. Additionally, routing was kept away from the region directly beneath the sensing area to prevent any electrical noise from coupling into the sensitive analog front-end.

Power integrity was a major focus in this design due to the multiple voltage domains required by the sensor. Separate routing was used for AVDD and IOVDD to isolate analog and digital supplies, with dedicated decoupling capacitors placed as close as possible to each power pin. The 1.8 V rail generated by the onboard LDO was routed with short traces to minimize voltage drop and noise. Grounding was handled with a continuous ground plane to provide a stable reference and reduce impedance, which is critical for maintaining consistent sensor performance. Care was also taken to ensure that return currents for high-speed digital signals did not interfere with the analog portions of the circuit.

Signal routing for communication lines such as I²C (SDA and SCL) was kept short and clean, with minimal crossings and proper spacing to reduce crosstalk. The level shifter was placed between the sensor and external header to ensure proper voltage translation while keeping the signal paths efficient. Supporting signals like interrupt, shutdown, and chip select were routed cleanly to their respective pins, allowing full control of the sensor. Finally, the board includes a clearly labeled header interface for easy connection to the main system. Overall, this PCB layout prioritizes signal integrity, power stability, and sensor accuracy, ensuring reliable distance measurements in the final system.

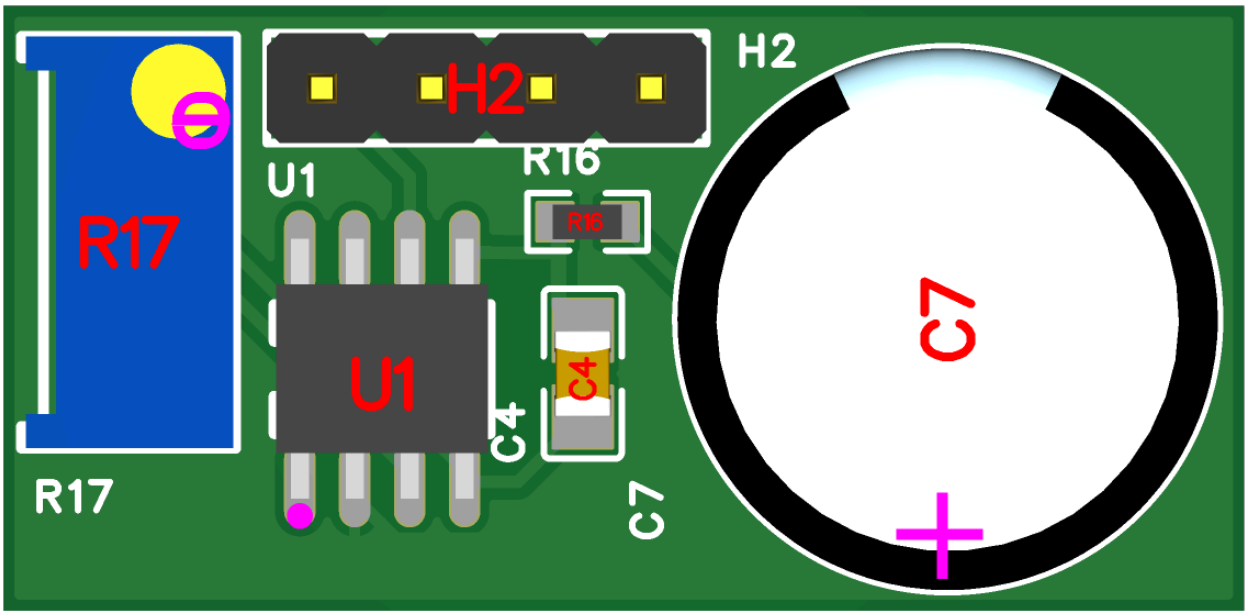


8.4 Audio amplifier PCB

The circuit shown is based on the LM386 low-voltage audio amplifier, which is designed to amplify weak analog signals into a level sufficient to drive a small speaker or similar load. In this configuration, the input signal is received through the H2 header and routed through a 10 kΩ potentiometer (R17), which functions as a variable voltage divider for volume control. By adjusting the potentiometer, the amplitude of the signal fed into the non-inverting input (Pin 3) of the LM386 can be regulated, allowing control over the output sound level. The LM386 itself operates as the main amplification stage, powered through Pin 6 (VS) with Pin 4 connected to ground, and internally amplifying the input signal to produce a higher-power output at Pin 5.

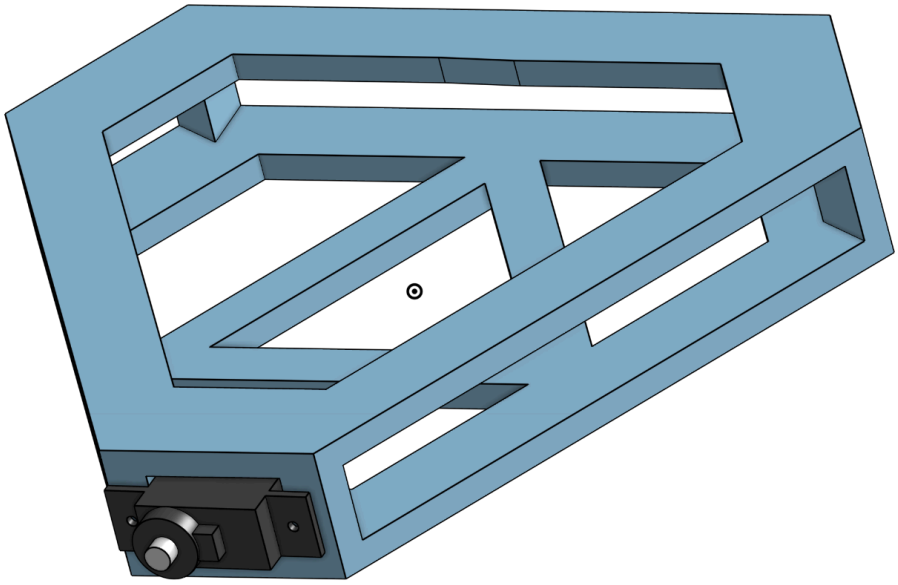
To ensure stable operation and reduce high-frequency oscillations, a Zobel network composed of a 15 Ω resistor (R16) and a 100 nF capacitor (C4) is connected at the output stage. This network helps maintain amplifier stability when driving reactive loads such as speakers by damping high-frequency noise and preventing unwanted oscillations. Additionally, a 200 µF coupling capacitor (C7) is placed in series with the output to block any DC component from reaching the load while allowing the amplified AC audio signal to pass through. This protects the connected speaker and ensures that only the intended audio waveform is delivered. Overall, the circuit provides a simple but effective low-power audio amplification system with adjustable gain, stable output behavior, and DC-safe signal coupling suitable for small embedded audio

applications.

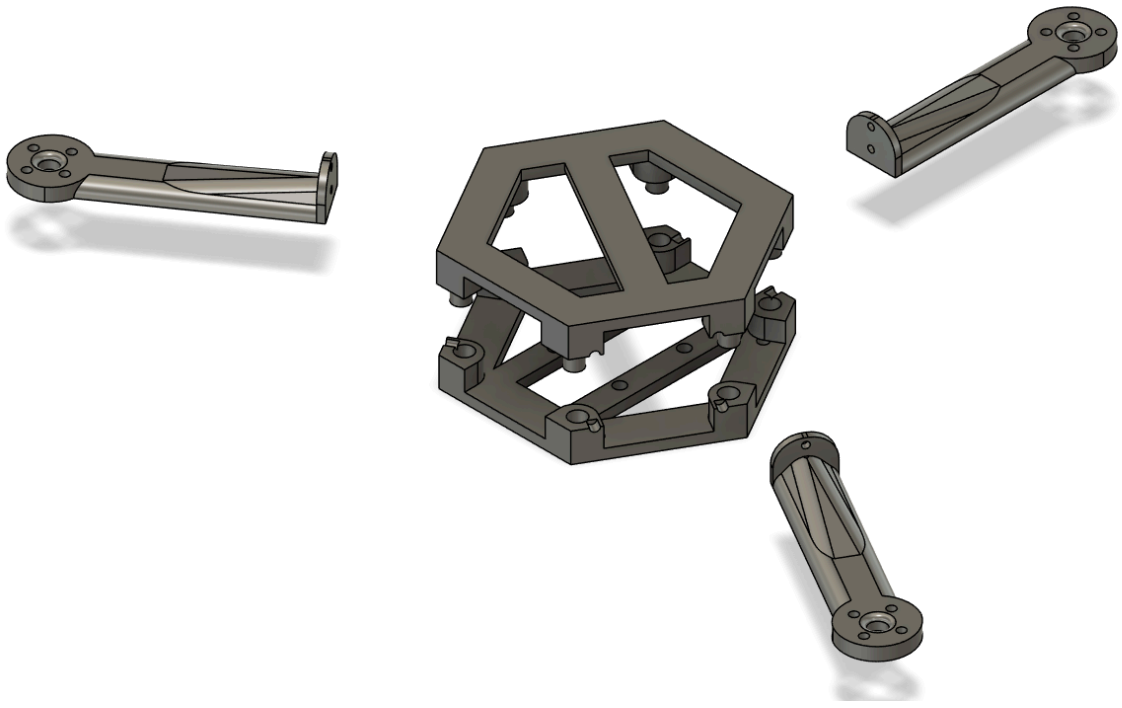


8.5 CAD Design Main body

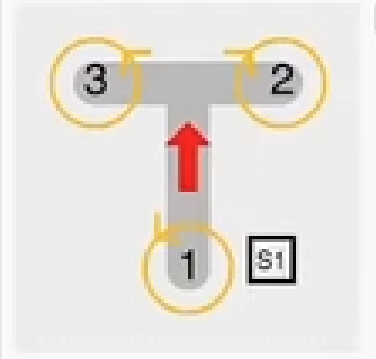
For the design of our main body, we went through 2 iterations. We went through a base model where we wanted to induce spin in our drone with 3 separate servos controlling 3 separate arms. This design we implemented at first bvecasue we culd put the PCB board under our main body with screws and then strap the battery onto the top of our program. This concept we stuck with throughout the whole drone. This is the main body with the singular servo in it to show the placement for it:



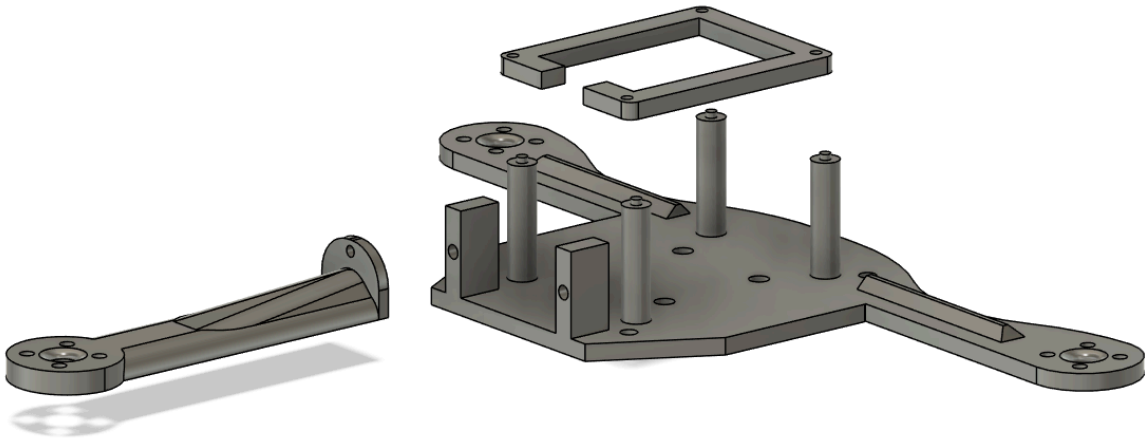
After this base design was filled out, we found a way to separate it and allow us to insert our Flight controller and ESC into the drone without having to break anything. This also shows where our arms were going to be placed in the drone.



The final main body for this program followed the diagram that the Flight program had provided us.



We used this design to then implement a better, more stable program for the drone. We wanted to remove some of the weight from the servos, so we decided on only one servo in the back of the drone. This would allow us to stay stable in the air and to control it if needed. We still needed the 3 layers, one for the battery to strap onto the top, the middle for the SpeedyBee to be screwed into, and then the screw holes that matched our PCB and allowed us to screw the PCB onto the bottom. This allowed us to stay stable with the weight mainly being in the middle and the freedom to work in our drone more freely.



8.6 Drone arms:

The design of the drone's arm assemblies draws inspiration from conventional FPV (First-Person View) drone structures, which are widely recognized for achieving an optimal balance between durability, low mass, and mechanical stability. This baseline architecture initially provided a reliable foundation for mounting the high-performance brushless motors required for our tri-motor configuration. Each arm was engineered to accommodate 3.5-inch propellers, requiring careful geometric planning to ensure adequate clearance during rotation while minimizing unnecessary bulk that would increase weight.

However, during testing, the initial arm design proved to be **too flexible and mechanically unstable under load**, resulting in unwanted vibration and reduced flight stability. As a result, the design was revised to a more rigid configuration in which the motor mounts were **directly**

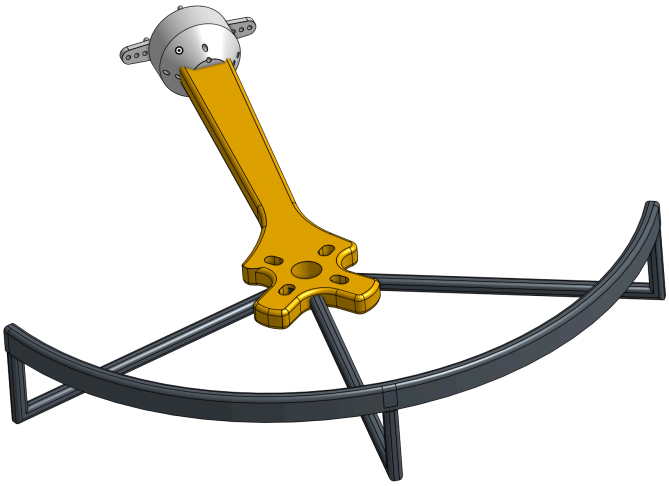
screwed into a reinforced central structure, improving stiffness and significantly reducing arm deflection during operation. This modification greatly improved overall structural integrity and flight stability by eliminating excess compliance in the arm assembly.

To enable the thrust-vectoring capabilities central to our control strategy, a servo-actuation interface was integrated at the base of each arm. This mechanism allows the entire motor assembly to rotate within the operational plane, providing precise directional thrust adjustments essential for yaw control and lateral maneuvering. The servo mount was designed to be structurally simple and mechanically robust, ensuring that it could withstand dynamic loading without compromising motion range.

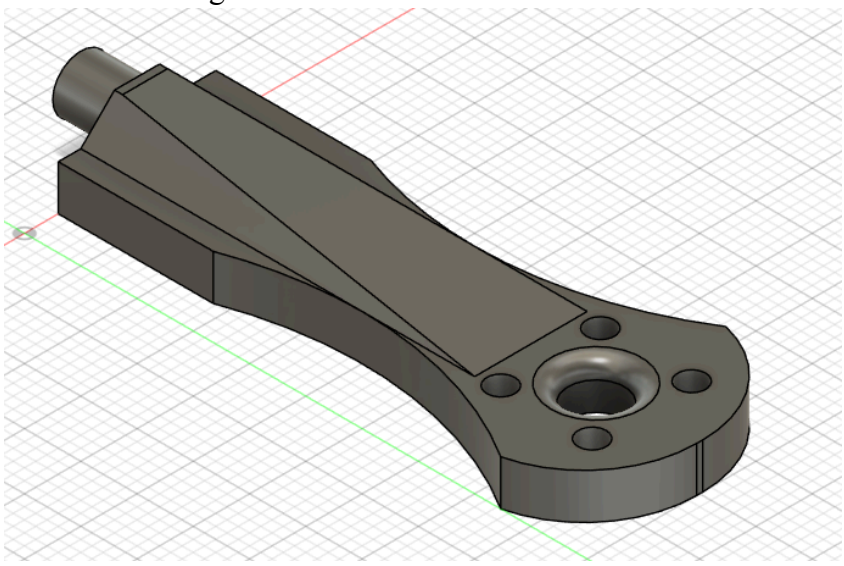
A key constraint of the project was the requirement that all propellers be fully enclosed to meet safety standards. In the initial design phase, these protective guards were 3D printed alongside the frame. However, due to **print fragility, structural weakness, and difficulty achieving consistent strength with thin-walled geometries**, the printed guards proved unreliable during testing. As a result, the design was revised and **commercially manufactured propeller guards were purchased and integrated instead**, providing improved durability, impact resistance, and safety compliance.

Collectively, these design decisions produced an arm assembly that is lightweight, structurally reinforced, compliant with safety requirements, and well-suited to real-world operation. The final configuration improves rigidity, reduces vibration, and enhances overall flight stability while still supporting the drone's thrust-vectoring and tri-motor flight architecture.

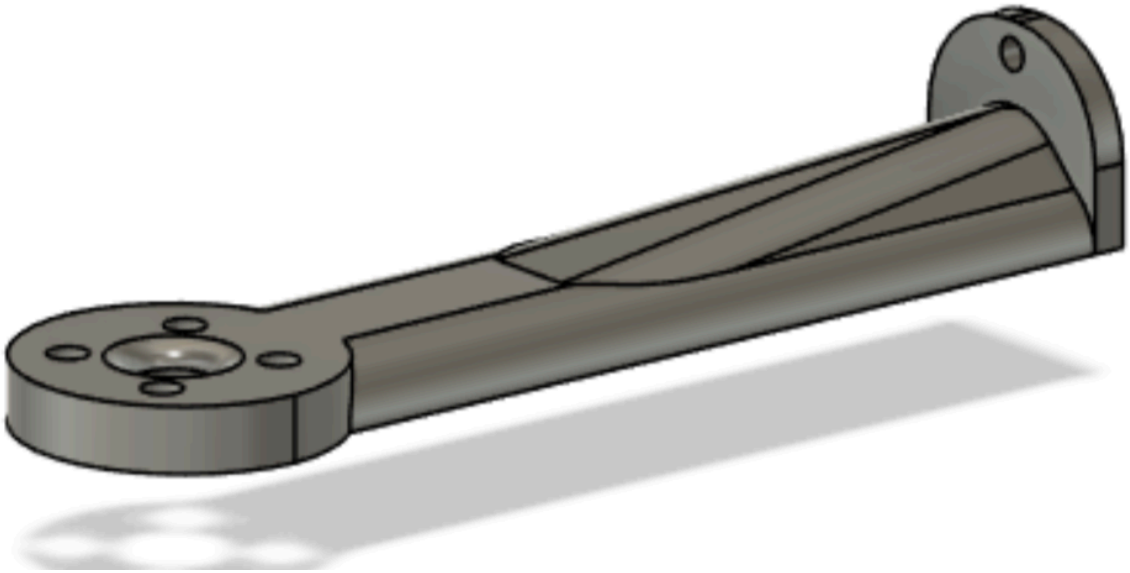
First arm design:



First Tested design:



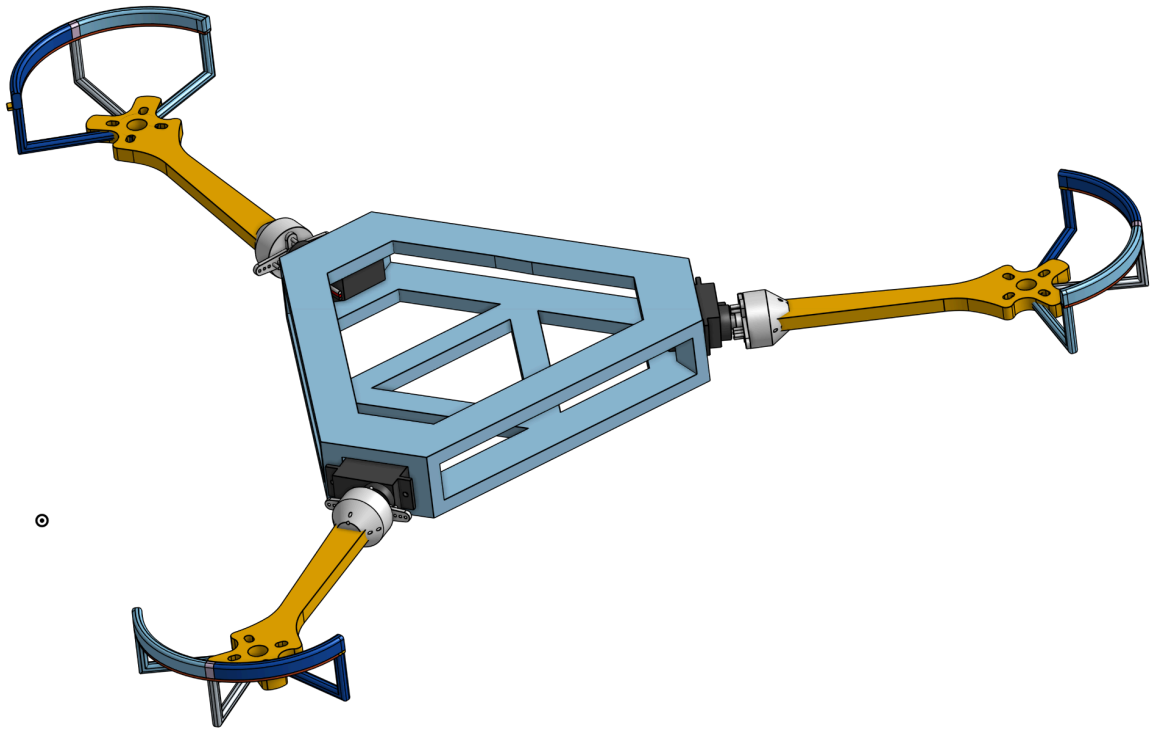
Final Arm Design:



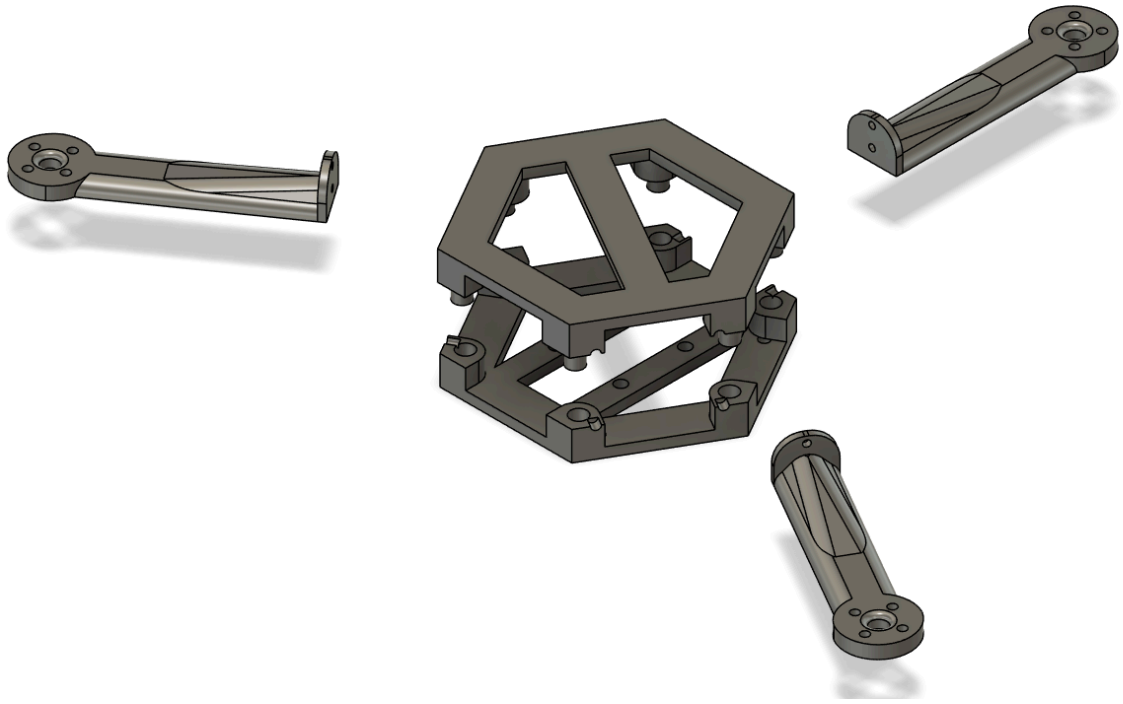
8.7 Full Drone:

In designing the overall airframe, we combined a lightweight central body structure with servo-actuated motor arms to create a fully custom, bidirectional thrust-vectoring drone optimized for our project constraints. The main body was engineered around a minimalistic triangular geometry, selected for its ability to evenly distribute load while providing open

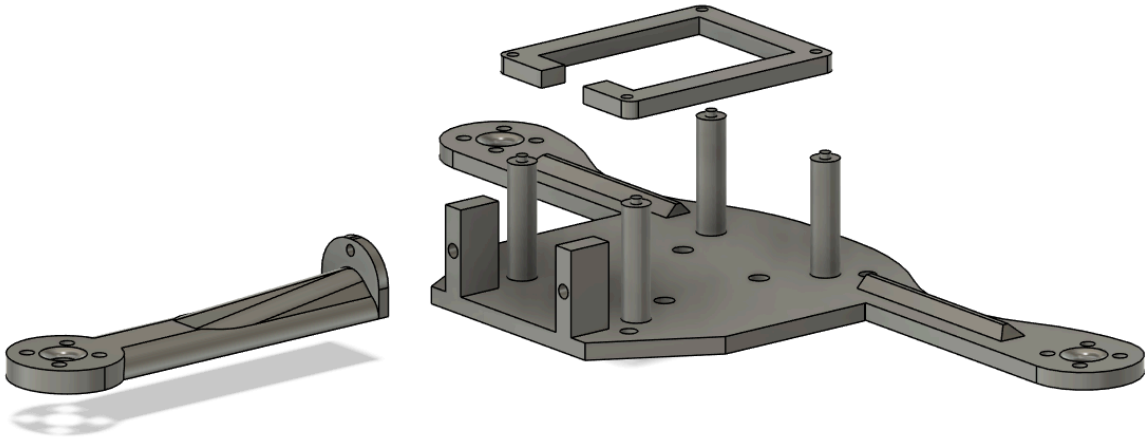
sections for the integration of the flight-controller PCB, ESC, battery, sensor modules, and power-distribution circuitry. This configuration was chosen to meet strict mass limits while maintaining accessibility for assembly and wiring management. Complementing the central frame, the motor arms were adapted from established FPV-drone design principles due to their proven balance of durability and low mass. Each arm was scaled and shaped specifically to accommodate a 3-inch propeller and to mount securely to a servo positioned at its base, enabling active rotation within the x–y plane for thrust-vectoring control. To satisfy safety requirements, circular propeller guards were incorporated around each arm assembly, fully enclosing the propellers while retaining a simple, continuous geometry that supports reliable, low-complexity FDM 3D printing. These combined design choices—lightweight structural framing, servo-rotated motor vectoring, and integrated propeller protection—were selected to ensure mechanical robustness, manufacturability, and compliance with project safety and performance specifications, ultimately resulting in a controllable and efficient tri-motor drone architecture. This first design lacked more modulation and the main ability to connect to the servos well. While trying to print this version out we ran into many issues.



Those issues lead me to this next drone design. The arms being different and the frame being about to come apart was the best way to work inside and around this design.



Even though our last drone design was perfect, we figured that to save even more weight and to implement our single servo design we made this new design.



Chapter 9 - System Testing and Prototype

9.1 Hardware Testing

9.1.1 ESP32

When testing that the ESP32- development boards we purchased work correctly. We first plugged it into power using the USB port on my laptop. The RGB LED is powered on which verifies that the device is receiving power. After, the GPIO pin voltages could be verified when using a multimeter. Then we finally tested the reset button function when pushing the button.

9.1.2 Sensor TOF400C-VL53L8CX

To test the functionality of the VL53L8CX sensor, we wired up the sensor to the ESP32 using the breadboard and properly connected it by pairing the 3V output to VIN, ground, and SDA and SCL to GPIO pins 21 and 22. After wiring it up properly we tested it by running some code since there was no other way to verify it was working properly.

9.1.3 Motors

To test the basic hardware functionality of the 1404 brushless motors, we connected a single motor to an electronic speed controller (ESC) and supplied the ESC with a regulated 5V input from the bench power supply. The signal wire from the ESC was connected to one of the ESP32's PWM-capable GPIO pins through the breadboard, while the ground lines of the ESC, ESP32, and power supply were tied together to ensure a common reference.

Before applying any control signal, we verified that the motor and ESC powered on correctly by confirming that the ESC initialization tones played when voltage was applied. This confirmed that the motor windings were intact and that the ESC hardware was functioning. After confirming proper power delivery and wiring, we used a basic PWM output from the ESP32 to send a minimum-throttle signal, allowing us to confirm that the motor spun up smoothly without stuttering or excessive vibration. This verified that the motor, ESC, and power wiring were all functioning correctly at the hardware level.

9.1.4 Servo Motors

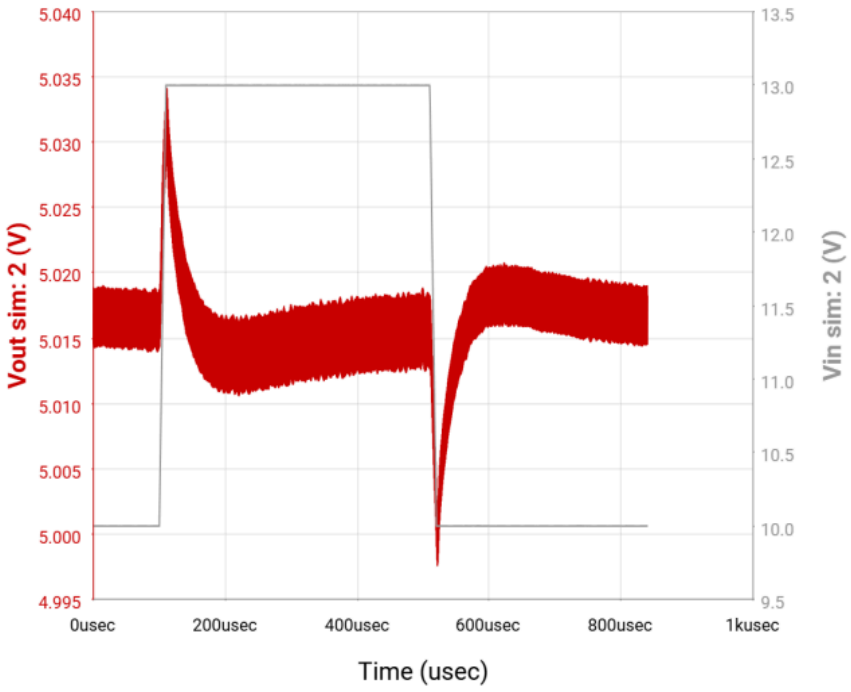
To test the functionality of the micro servo motors, we connected a single servo to the breadboard and powered it using a regulated 5V output from the bench power supply. The ground of the servo was tied to the same ground as the ESP32 to ensure a common reference point for signal control. The control line was then connected to one of the ESP32's PWM-capable GPIO pins.

Before sending any control pulses, we verified that the servo was receiving power by gently touching the output shaft and feeling the slight resistance caused by the servo's internal circuitry engaging when powered. Once this was confirmed, we applied a basic PWM signal from the

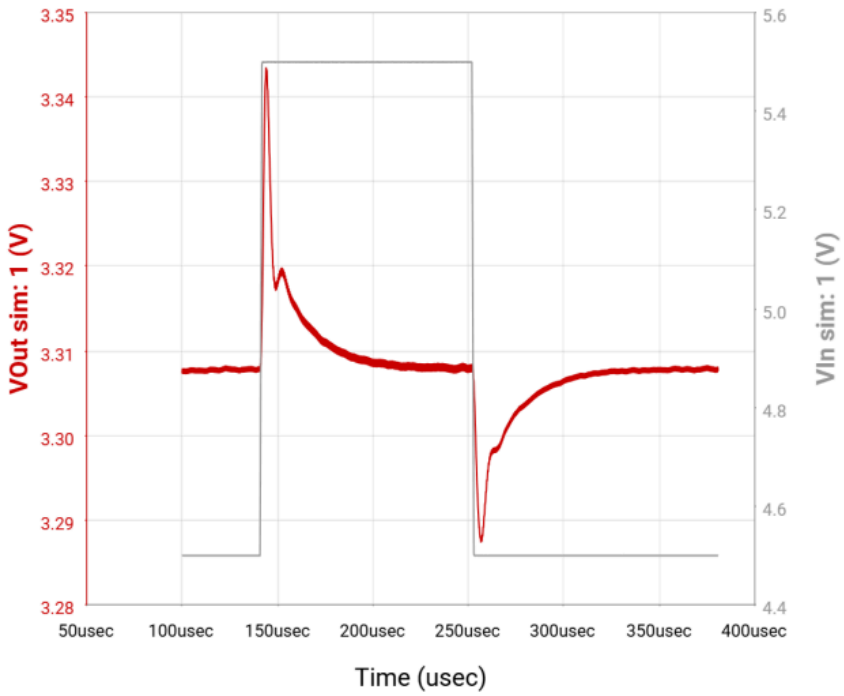
ESP32 to move the servo arm between two fixed positions. The servo successfully rotated to each commanded position without jittering or stalling, confirming that the power delivery, wiring, and internal motor driver circuitry of the servo were all functioning properly at the hardware level.

9.1.5 PCB Simulation

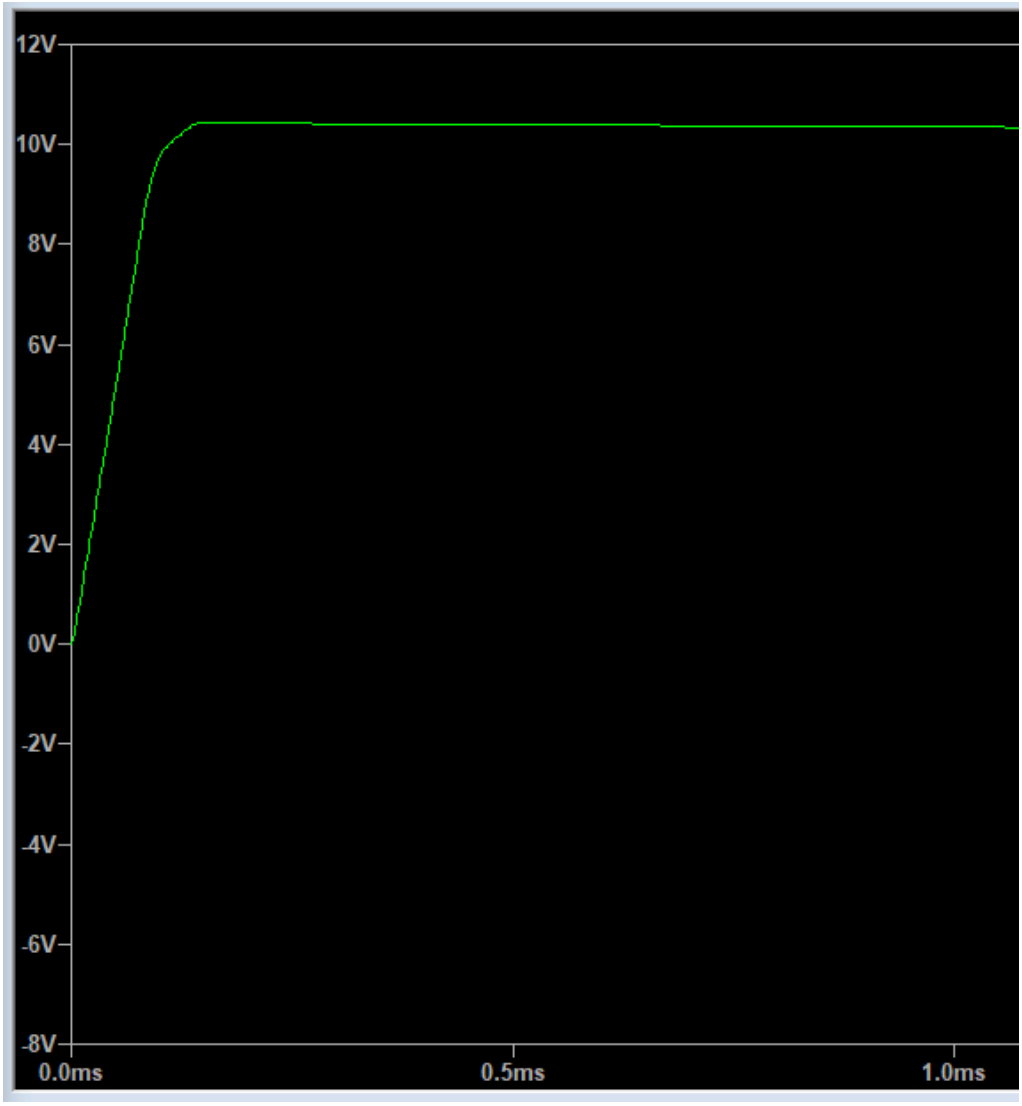
To evaluate the performance of the LMR33630BRNXR regulator under realistic operating conditions, the simulation varies the input voltage between 10–13 V, reflecting the fluctuation range of our 12 V battery. Despite these changes at the input, the converter maintains regulation and the output ultimately settles back to a stable 5 V, demonstrating the robustness of the design.

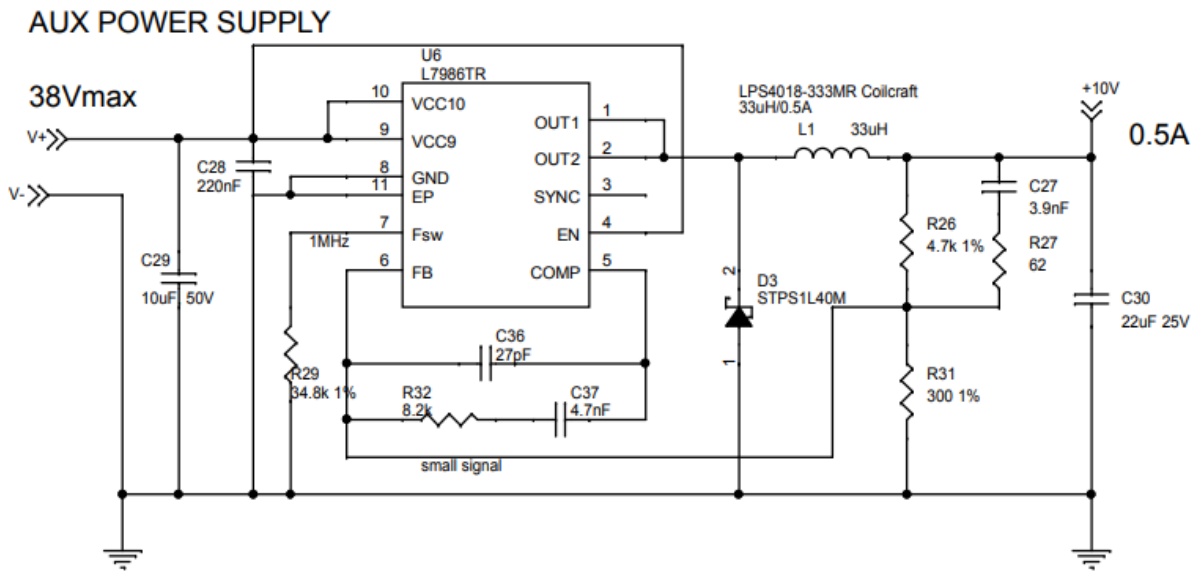


To analyze the behavior of the LMR23625CFDDAR 3.3 V buck converter, the simulation applies an input voltage variation between 4.6–5.4 V, representing the expected fluctuations from the upstream 5 V regulation stage. Even with these rapid changes at the input, the output quickly recovers and settles to a stable 3.3 V, demonstrating the converter's ability to maintain regulation under dynamic conditions.



As part of the ESC power architecture, a dedicated 10-volt regulator was designed and simulated to ensure stable and isolated power delivery to the control and actuation circuits. The regulator accepts the drone's **11.1-volt (3S LiPo) battery input** and steps it down to a clean and reliable 10 V rail, which serves as the intermediate supply for several critical subsystems. This regulated line powers the micro servos responsible for thrust-vectoring at each motor, provides the required operating voltage for the ESC's MOSFET gate-driver circuitry, and feeds downstream converters that generate the 5 V and 3.3 V rails used by the ESP32 flight controller, onboard sensors, and communication electronics. Using this regulation stage ensures that sensitive digital components remain protected from battery-voltage fluctuations and prevents high-current ESC switching noise from affecting servo actuation or microcontroller performance. Within the broader design methodology, the regulator forms the foundation of electrical stability, enabling all ESC and control circuitry to operate consistently and within safe electrical limits. The circuit below was simulated in LTSpice and generated the appropriate voltage we need to input into the phase shifters.


12V
10V
8V
6V
4V
2V
0V
-2V
-4V
-6V
-8V
0.0ms
0.5ms
1.0ms



9.2 Software Testing

9.2.1 ESP32

To begin software testing on the ESP32 development board, we first set up our programming environment using the Arduino IDE, which provided a straightforward workflow for compiling and flashing programs. After installing the ESP32 board support package and selecting the proper COM port, we connected the ESP32 via USB and verified that the device was detected by the IDE.

Our first program uploaded to the board was a simple “Hello World” example. This program prints “Hello World” repeatedly to the serial monitor, allowing us to verify that the ESP32 could compile code correctly, flash firmware without errors, and establish a stable UART communication channel with the host computer. Once uploaded, the serial monitor displayed the expected output, confirming that the USB-to-UART interface and internal bootloader were functioning correctly.

After validating basic serial output, we ran the standard “Blink” example to test GPIO control. This program toggles the onboard LED at a one-second interval, providing a visual confirmation that the board could execute timing functions and manipulate digital output pins. The LED blinked as expected, indicating stable execution of user programs.

With communication and GPIO confirmed, we continued by testing example scripts involving digital input and simple PWM output. These tests allowed us to verify that multiple pins could be configured correctly and that the ESP32 responded predictably when interacting with external components such as ESCs, sensors, and servos in later testing phases.

Overall, these initial programs confirmed that the ESP32 was functioning correctly, fully programmable, and ready for integration with the rest of the system.

9.2.2 Sensor

We started testing by confirming that the distance sensor itself was operating reliably. The first step was simply powering it from the ESP32 and running a basic test sketch that printed raw distance readings to the serial console. This gave us a clean, isolated way to verify that the wiring was correct, the sensor was recognized, and it could produce consistent measurements without any additional system complexity. With the console open, we moved objects closer and farther from the sensor and watched the readings change in real time. Seeing steady, responsive values told us the sensor was healthy and that the ESP32 could communicate with it without errors or timeouts.

This console-based testing also helped establish a baseline for what “normal” readings looked like. If the values flickered, froze, or reported timeouts, we knew something needed attention, whether it was power, alignment, or initialization. Only after we were confident that the sensor responded predictably under simple conditions did we move on to integrating it with the Wi-Fi portion of the project.

Validating the sensor in isolation like this made the later stages much smoother, because we knew that any issues that appeared after that point were coming from networking or server behavior rather than the sensor hardware itself.

9.2.3 Motors

To confirm that the motors behaved correctly, we began with a simple standalone test that powered each motor directly from the ESP32’s control signals and monitored how they responded to fixed speed and delay settings. The goal at this stage wasn’t to integrate the motors into a larger system, but to verify that they could spin reliably at known rates. We uploaded a basic program that set the motor speed, triggered a spin for a set duration, paused, and then repeated. By watching the motor’s movement and listening to its sound, we could tell whether the speed was stable and whether the on/off timing matched the delays we programmed.

This testing made the behavior easy to evaluate. If the motor spun smoothly at the expected pace and started and stopped at predictable intervals, we knew the control logic was working.

9.2.4 Servo Motors

We approached the servo motors with the same “test in isolation” mindset that we used for the earlier motor checks. Instead of jumping straight into full system integration, we loaded a small test program that commanded each servo to sweep to its leftmost angle, pause, then move to its rightmost angle, and repeat. That gave us a predictable motion pattern to look for. Watching the servo arms physically move through those positions made it easy to confirm whether the control signal and power supply were behaving the way we intended.

What made this test especially useful was the timing. Because the servo was supposed to hold each position for a specific delay, we could use those pauses as checkpoints. If the servo reached the correct angles and stayed put for the right amount of time before switching directions, we knew the signal timing was sound. Any jitter, drifting, or failure to hit the expected endpoints would have shown us immediately that something needed correction. Running this left-right sweep several times gave us confidence that the servos were responding reliably before we built them into the larger system.

9.2.5 Integration of Systems

9.2.5.1 Integration of the Motor and Servo Motor

When it came time to test the motor and servo together, the goal was to make sure both devices could operate simultaneously without interfering with each other's timing or response. We combined the two earlier test routines, the fixed-rate motor cycle and the left-right servo sweep, into a single program and watched how both systems behaved when running side by side. This let us confirm right away whether shared power, overlapping control signals, or timing loops caused either device to slow down, jitter, or fall out of rhythm.

The real value of this combined test was in observing how the motions lined up. The motor needed to spin at its steady, predictable rate while the servo continued to hit its left and right endpoints cleanly. If both motions stayed consistent, even when happening at the same time, that told us the ESP32 could manage the control signals and delays for both devices without performance drops.

9.2.5.2 Integration of Sensor into WiFi Stream

To verify that our ultrasonic-style distance sensing system worked over Wi-Fi, we set up the ESP32 to broadcast a simple live-updating webpage and used that as our main test harness. The idea was straightforward: if the ESP32 could measure distance with the sensor, connect to the campus network, and reliably deliver those readings to a browser, then all pieces of the system were functioning as intended. Once the board powered on, we waited for it to authenticate on the university Wi-Fi and checked the serial monitor for confirmation and the assigned IP address. That IP became our entry point for testing—typing it into any browser on the same network immediately brought up the webpage the ESP32 was serving.

From there, validating the system was largely an observational process. The webpage displayed a large distance value that refreshed several times per second, so we could physically move an object in front of the sensor and watch the numbers change in real time. This gave us direct feedback that the sensor readings were being captured, transmitted across the network, and rendered correctly by the browser. When the values updated smoothly, we knew the Wi-Fi connection was stable and that the device could handle continuous requests without dropping the sensor or freezing the interface. If the page ever showed "error," it was a clear sign that something in the chain (sensor, connection, timing) wasn't behaving, which made troubleshooting easier. In practice, being able to interact with the system visually and immediately like this gave us confidence that the Wi-Fi integration and data streaming behaved reliably under normal use.

9.2.6 Plan for SD2

As we transition into Senior Design 2, the focus will shift from preliminary testing and component validation to full system integration and flight-ready development. Over winter break, our team will finalize the PCB schematics and submit the first round of board orders. At the same time, the 3D-printed frame components like our finalized CAD model. These two deliverables (PCB + airframe) will form the foundation of our complete prototype assembly at the start of SD2.

During SD2, we will move straight into full functional development. After assembling the custom PCB and 3D-printed frame, we will integrate the motors, servos, and sensors to begin building the complete working prototype. Our focus will be getting the drone powered, spinning, collecting sensor data, and sending telemetry through the base station/mobile app as quickly as possible.

Once the core system is operational, we will begin testing and refining our stretch goals. These include direction movement as the drone is rotating and improving accuracy during motion as much as possible. The remainder of SD2 will be devoted to iterative testing, tuning, and final adjustments leading up to our demonstration

Chapter 10 - Administrative Content

10.1 Budget

The financing of the drone design and development project is being supported through sponsorship from the DRACO Lab, which has provided the primary funding to cover research, materials, and prototyping expenses. This sponsorship ensures that the team has access to the necessary resources for both the technical and logistical aspects of the project, including hardware components, testing facilities, and development tools. The current estimate for **one** drone is **$197.80**. This, however, would not include defective parts, testing accidents, or software requirements. The DRACO lab would finance our project up to **$250**, with any extra reimbursement occurring at the end of the spring semester if there is any funds left.

10.2 Bill of Materials

Table 10.2. *An Itemized Bill of Materials*

Component	Name	Price ($/per part)	Quantity
Mechanical			
3D printed main body	Printed plastic	$3	2

Bearing/Gyroscope	Bearing	$0.5 - 1	1
propellor	Fpv props	$0.50 - 2	5
Screws (for PCB boards)	M8x0.5	$0.03	10
Electrical			
Motor	Servo motor	$2 - 8 Per motor	9
Gyroscope sensor	Arduino sensor	$10	1
Microcontroller	TBD	$15	1
Antenna	TBD	$0.5	1
Wires/wiring	TBD	$3	1ft of wire (estimated)
PCB boards	Custom	$20	4
Total		**$197.8**	

10.3 Distribution of Work-table

The table below shows the distribution of work and the tasks assigned to each group member. While specific subsystems are assigned to individual members, all components depend on one another, requiring collaboration across the team. This approach ensures that every member remains involved and informed about the design and implementation of all systems, even those they are not directly responsible for.

Table 10.3. *Distribution of Worktable*

Computer Engineer	**Responsibilities**
Corey Barbera	Sensor Selection and Implementation
	MCU Selection and Implementation
	Website Design and Management
	Software Design and Implementation

Computer Engineer	Responsibilities
Carlos Alfonso Vargas	MCU Selection and Implementation
	Sensor Selection and Implementation
	Communication Implementation
	Software Design and Implementation

Electrical Engineer	Responsibilities
Alex Bayda	PCB Design and Implementation
	Power Distribution Design and Implementation
	Circuit Design
	Testing

Electrical Engineer	Responsibilities
James Thompson	PCB Design and Implementation
	CAD Design
	Power Distribution Design and Implementation
	Administrative Documentation

10.4 Project Milestones

To ensure that our project stays on track my team implemented a table of projected milestones. This table will include anticipated time frames throughout the next two semesters, so Senior Design 1 and Senior Design 2. Creating this table will help us be able to visibility see what our

time frame looks like over the next few months and hopefully prevent our project from falling behind.

Table 1: Senior Design I Project Milestones

Task	Time Period	Dates	Responsibility
Project Proposal	1 Week	8/18 - 8/22	All
Research Drones	3 Weeks	8/22 - 9/10	All
D&C Document	2 Weeks	8/22 - 9/5	All
D&C Meeting	30 Min	9/10	All
Research Flight Controllers	2 Weeks	9/5 - 9/19	James / Alex
Research Autonomy / Navigation	2 Weeks	9/5 - 9/19	Corey / Carlos
Research Communication Systems	2 Weeks	9/5 - 9/19	Corey / Carlos
Research Sensor Integration	2 Weeks	9/5 - 9/19	James / Alex
30-Page Milestone	2 Weeks	9/5 - 9/19	All
Start PCB Design	2 Weeks	9/19 - 10/6	James / Alex
CAD Designing	2 Weeks	9/19 - 10/6	Corey / Alex
60-Page Milestone	3 Weeks	9/19 - 10/10	All
Order Drone Components / PCB	1 Week	10/6 - 10/13	All
90-Page Milestone	3 Weeks	10/10 - 10/31	All
Assemble Project Prototype	5 Weeks	10/20 - 11/25	All
120-Page Milestone	2.5 Weeks	10/31 - 11/19	All
Final Document	Ongoing	9/5 - 11/25	All

Table 2: Senior Design II Project Milestones

Task	Time Period	Dates	Responsibility
Prototype Assembled	2 Weeks	01/12 - 1/26	All
Prototype Testing	2 Weeks	1/26 - 2/9	All
Prototype Adjustments	2 Weeks	2/9 - 2/20	All
Project Demo	TBD	TBD	All
Final Team Meeting before Presentation	TBD	TBD	All
Final Documentation	TBD	TBD	All
Final Presentation	TBD	TBD	All

Chapter 11 - Conclusion

In conclusion, the Dizzy n-Copter project successfully demonstrates a lightweight, continuously rotating drone that is capable of real-time sensing and wireless communication. The main objective of this design was to create a sub-250 g aerial platform that can intentionally rotate at approximately 60 RPM while collecting environmental data and maintaining predictable, controllable behavior. Through research, prototyping, and hardware validation, we developed a working foundation for a drone that leverages its rotational motion as a functional advantage rather than a flight limitation.

The system integrates several core subsystems, including brushless motor control, servo-based tilt mechanisms, the VL53L8CX distance sensor, Wi-Fi telemetry using the ESP32, and a modular 3D-printed frame. Each component underwent individual hardware and software testing to ensure reliability before system-level integration. The ESP32 proved to be an effective microcontroller for managing both PWM motor actuation and wireless data streaming, while the distance sensor demonstrated consistent measurement performance during live testing. Our motor and servo evaluations confirmed that the propulsion and actuation hardware responded smoothly to control signals, providing a dependable basis for full-flight implementation.

A key innovation of the Dizzy n-Copter is its use of deliberate constant yaw rotation to broaden its sensing capabilities. Instead of relying on high-cost LiDAR modules, the platform explores how rotational movement combined with a low-cost sensor can achieve 360-degree environmental awareness. This approach has the potential to reduce complexity and cost while still enabling useful spatial mapping for applications such as inspection, indoor navigation, or object detection.

This document has presented the full engineering process behind the drone's design. Covering research, component selection, architecture, hardware validation, and software testing. The results confirm that the platform is technically viable and ready for further development. In the next stage of work, efforts will focus on implementing full flight stabilization, refining the rotating frame mechanics, expanding sensor fusion capabilities, and integrating a mobile app interface for remote control and telemetry visualization.

Overall, the Dizzy n-Copter provides a strong foundation for an innovative, accessible, and low-cost aerial sensing system. The progress made throughout this project demonstrates its feasibility and sets the stage for achieving a fully functional rotating drone with enhanced scanning capabilities in the upcoming development phase.

Appendix - References

[1] “LiDAR Drone Systems: Using LiDAR Equipped UAVs,” DJI Enterprise. Accessed: Sept. 03, 2025. [Online]. Available: https://enterprise-insights.dji.com/blog/lidar-equipped-uavs

[2] OASIS, “Message Queuing Telemetry Transport (MQTT) TC.” Accessed: Sept. 03, 2025. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt

[3] Nireka, “WebRTC vs WebSocket: Key Differences and Which to Use to Enhance Real-Time Communication?” *Dyte*, Oct. 5, 2023. [Online]. Available: https://dyte.io/blog/webrtc-vs-websocket/

[4] D. Droeschel, S. Behnke, and D. Holz, “Omnidirectional perception for lightweight MAVs using a continuously rotating 3D laser scanner,” *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. II-1/W1, pp. 89–96, 2013. doi: 10.5194/isprsannals-II-1-W1-89-2013

[5] M. Müller, C. Pfeiffer, and R. Siegwart, “A self-rotating, single-actuated UAV with extended sensor field of view,” *Science Robotics*, vol. 8, no. 74, p. eade4723, 2023. doi: 10.1126/scirobotics.ade4723

[6] Y. Zhao, J. Guo, and K. Xu, “Rotating 3D LiDAR mapping system for rotorcraft drones,” *Journal of Field Robotics*, vol. 40, no. 5, pp. 1123–1140, 2023. doi: 10.1002/rob.22156

[7] F. Ahmed, T. Dissanayake, and A. Jayasuriya, *Low-Cost Ultrasonic-Based Object Detection and Collision Avoidance for Mobile Robots*. Springer, 2020.

[8] Z. Zhao, P. Wang, and C. Wu, “Review of ultrasonic ranging methods and their current challenges,” *Sensors*, vol. 21, no. 11, p. 3759, 2021. doi: 10.3390/s21113759

[9] “STM32 High-Performance MCUs,” STMicroelectronics. Accessed: Oct. 15, 2025. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32-high-performance-mcus.html

[10] “What is a Drone Flight Controller,” Grepow. Accessed: Oct. 15, 2025. [Online]. Available: https://www.grepow.com/blog/what-is-a-drone-flight-controller.html

[11] “STM32 High-Performance MCUs,” STMicroelectronics. Accessed: Oct. 15, 2025. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32-high-performance-mcus.html

[12] “STM32F405/415 Microcontrollers,” STMicroelectronics. Accessed: Oct. 15, 2025. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f405-415.html

[13] “SpeedyBee F405 Mini BLS 35A 20×20 Stack,” SpeedyBee. Accessed: Oct. 15, 2025. [Online]. Available: https://www.speedybee.com/speedybee-f405-mini-bls-35a-20x20-stack/

[14] “STM32F405xx / STM32F407xx Datasheet,” STMicroelectronics. Accessed: Oct. 15, 2025. [Online]. Available: https://www.st.com/resource/en/datasheet/dm00037051.pdf

[15] “Review: SpeedyBee F405 Mini Stack,” Oscar Liang. Accessed: Oct. 15, 2025. [Online]. Available: https://oscarliang.com/speedybee-f405-mini-fc-stack/

[16] “SpeedyBee F405 Mini BLS 35A 20×20 Stack User Manual V1.0,” Rotorama. Accessed: Oct. 15, 2025. [Online]. Available: https://www.rotorama.cz/cms/assets/docs/419880ae652338ce3f165aad6e3349cc/15189-0/speedybee-f405-mini-bls-35a-stack-user-manual-en.pdf

[17] “STM32F405RGT6 – Mouser Product Page,” Mouser Electronics. Accessed: Oct. 15, 2025. [Online]. Available: https://www.mouser.com/ProductDetail/STMicroelectronics/STM32F405RGT6?qs=Z8%252BeY1k3TIKgj7QWsYGpQw%3D%3D

[18] “STM32F4 Series – STMicroelectronics,” STMicroelectronics. Accessed: Oct. 15, 2025. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f4-series.html

[19] “What is a Drone Flight Controller,” Grepow. Accessed: Oct. 15, 2025. [Online]. Available: https://www.grepow.com/blog/what-is-a-drone-flight-controller.html

[20] “Betaflight Documentation,” Betaflight Wiki. Accessed: Oct. 15, 2025. [Online]. Available: https://betaflight.com/docs/

[21] “Inertial Measurement Unit (IMU): An Introduction,” Advanced Navigation. Accessed: Oct. 15, 2025. [Online]. Available: https://www.advancednavigation.com/tech-articles/inertial-measurement-unit-imu-an-introduction/

[22] R. Roriz, H. Silva, F. Dias, and T. Gomes, "A Survey on Data Compression Techniques for Automotive LiDAR Point Clouds," *Sensors*, vol. 24, no. 10, p. 3185, 2024. [Online]. Available: https://www.mdpi.com/1424-8220/24/10/3185

[23] Hitron Technologies, "802.11ac vs 802.11ax: Which is Better?," 2021. [Online]. Available: https://us.hitrontech.com/learn/802-11ac-vs-802-11ax-which-is-better/

[24] Cisco Systems, "Omni Antenna vs. Directional Antenna," 2007. [Online]. Available: https://www.cisco.com/c/en/us/support/docs/wireless-mobility/wireless-lan-wlan/82068-omni-vs-direct.html

[25] Dacast, "TCP vs UDP - Which Is Better for Streaming?," 2023. [Online]. Available: https://www.dacast.com/blog/tcp-vs-udp/

[26] Ably Realtime, "Pros and cons of WebSockets - the complete guide," 2023. [Online]. Available: https://ably.com/topic/websockets-pros-cons

[27] ACTE, "Cyclic Redundancy Check (CRC): Ensuring Data Integrity," 2025. [Online]. Available: https://www.acte.in/cyclic-redundancy-check

[28] M. Faizullin, A. Kornilova, and G. Ferrer, "Open-Source LiDAR Time Synchronization System by Mimicking GNSS-clock," *arXiv*, 2021. [Online]. Available: https://arxiv.org/abs/2107.02625

[29] J. González-Gómez, J. García, and A. Romero, "Efficiency Analysis of Brushless Motors for Micro-UAV Applications," *Drones*, vol. 7, no. 2, p. 98, 2023. https://doi.org/10.3390/drones7020098

[30] K. Watanabe and M. Fujita, "Optimization of Micro UAV Propulsion Systems for Hover Efficiency," *Aerospace*, vol. 9, no. 3, p. 118, 2022. https://doi.org/10.3390/aerospace9030118

[31] Kovtun, V. (2024). *Influence of the shape and number of blades on the efficiency of the UAV propeller*. *Aerospace and Technical Technology*, 4(S1), 45–52.

[32] GetFPV. (2025). *Gemfan Hurricane 2009 3-Blade Propeller (Set of 8)* [Product page]. Retrieved from https://www.getfpv.com/gemfan-hurricane-2009-3-blade-propeller-set-of-8-1-5mm.html

[33] Emax. (2025). *Emax Avan Blur 2×1.9×3 Propeller* [Product page]. Retrieved from https://emax-usa.com/products/avan-blur-2-inch-prop

[34] HQProp. (n.d.). *HQ Durable Prop T2×2.5×3 (Polycarbonate)* [Product page]. Retrieved from https://www.hqprop.com/hq-durable-prop-t2x25x3-2cw2ccw-poly-carbonate-p0038.html

[35] Sapit, A., Masjan, S., & Shater, T. (2023). *Aerodynamics drone propeller analysis by using computational fluid dynamics (CFD)*. *Journal of Computational Fluids*, 2(1), 17–24.

[36] Susi, L., Unt, A., & Heering, A. (2025). *The efficiency of drone propellers — A relevant step towards sustainability*. *Proceedings*, 90(1), 89. MDPI.

[37] "Basics of UART Communication," Circuit Basics. Accessed: Oct. 15, 2025. [Online]. Available: https://www.circuitbasics.com/basics-uart-communication/

[38] "Basics of the I2C Communication Protocol," Circuit Basics. Accessed: Oct. 15, 2025. [Online]. Available: https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/

[39] "Yaw Spin Recovery and Gyro Overflow Detect," Betaflight Documentation. Accessed: Oct. 25, 2025. [Online]. Available: https://www.betaflight.com/docs/wiki/guides/current/Yaw-Spin-Recovery-and-Gyro-Overflow-Detect

[40] "FPV Drone Flight Controllers Explained: F1, F3, F4, F7, H7 (Latest Version 2024)," MEPSKING. Accessed: Oct. 25, 2025. [Online]. Available: https://www.mepsking.shop/blog/fpv-drone-flight-controllers-explained-f1-f3-f4-f7-h7-latest-version-2024.html

[41] “Quadichopter,” DrehmFlight. Accessed: Oct. 25, 2025. [Online]. Available: https://www.drehmflight.com/post/quadichopter

Appendix - Codes

App Code

Status Page

```
import 'dart:async';

import 'package:flutter/material.dart';

import '../services/ble_telemetry_service.dart';

import '../widgets/telemetry_drawer.dart';



// ---------------------------------------------------------------------------

// Theme palette — matches telemetry_drawer.dart exactly

// ---------------------------------------------------------------------------

const Color _bg       = Color(0xFF0A0E1A);

const Color _surface  = Color(0xFF121828);

const Color _border   = Color(0xFF1E2A40);

const Color _accent   = Color(0xFF00E5FF);

const Color _textPrim = Color(0xFFE8F0FF);

const Color _textSec  = Color(0xFF6B80A0);

const Color _good     = Color(0xFF00E5A0);

const Color _warn     = Color(0xFFFFAA00);

const Color _danger   = Color(0xFFFF3B5C);



// ---------------------------------------------------------------------------

// LedStatusPage
```

```dart
// ---------------------------------------------------------------------------


class LedStatusPage extends StatefulWidget {
 const LedStatusPage({super.key});


 @override
 State<LedStatusPage> createState() => _LedStatusPageState();
}


class _LedStatusPageState extends State<LedStatusPage> {
 final BleDistanceService bleService = BleDistanceService();
 final GlobalKey<ScaffoldState> _scaffoldKey = GlobalKey<ScaffoldState>();


 int? distanceMm;
 double? voltage;
 double? current;
 int? mahUsed;
 int? batteryRemaining;
 double? pitch;
 double? roll;
 double? yaw;
 String? flightMode;
 bool droneOn      = false;
 bool hoverEnabled = false;
 bool spinEnabled  = false;
 bool connected    = false;
```

```
bool connecting   = false;

String statusMessage = "";

double? rpm;



StreamSubscription? distanceSub;

StreamSubscription? batterySub;

StreamSubscription? attitudeSub;

StreamSubscription? flightModeSub;

StreamSubscription? droneOnSub;

StreamSubscription? hoverSub;

StreamSubscription? spinSub;

StreamSubscription? lidarMapSub;



@override

void dispose() {

  bleService.disconnect();

  super.dispose();

}



// ---------------------------------------------------------------------------

// BLE connect / disconnect

// ---------------------------------------------------------------------------



Future<void> _connectBle() async {

  setState(() { connecting = true; statusMessage = "Scanning for ESP32…"; });
```

```
  try {
    await bleService.connect();
    setState(() { connected = true; statusMessage = "Connected to ESP32"; });

    distanceSub = bleService.distanceStream?.listen((v) {
      setState(() => distanceMm = v >= 0 ? v : null);
    });

    batterySub = bleService.batteryStream?.listen((d) {
      setState(() {
        voltage          = d["voltage"];
        current          = d["current"];
        mahUsed          = d["mah_used"];
        batteryRemaining = d["remaining"];
      });
    });

    attitudeSub = bleService.attitudeStream?.listen((d) {
      setState(() {
        pitch = d["pitch"];
        roll  = d["roll"];
        yaw   = d["yaw"];
        rpm   = d["rpm"];
      });
    });
```

```
    flightModeSub = bleService.flightModeStream?.listen((m) {
      setState(() => flightMode = m);
    });

    droneOnSub = bleService.droneOnStream?.listen((v) {
      setState(() { droneOn = v; if (!droneOn) hoverEnabled = false; });
    });

    hoverSub = bleService.hoverStream?.listen((v) {
      setState(() => hoverEnabled = v);
    });

    spinSub = bleService.spinStream?.listen((v) {
      setState(() => spinEnabled = v);
    });

    lidarMapSub = bleService.lidarMapStream?.listen((_) {});

  } catch (e) {
    setState(() { connected = false; statusMessage = "Connection failed"; });
    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text("BLE error: $e"),
          backgroundColor: _danger,
        ),
```

```
      );
    }
  } finally {
    setState(() => connecting = false);
  }
}

Future<void> _disconnectBle() async {
  await distanceSub?.cancel();
  await batterySub?.cancel();
  await attitudeSub?.cancel();
  await flightModeSub?.cancel();
  await droneOnSub?.cancel();
  await hoverSub?.cancel();
  await spinSub?.cancel();
  await lidarMapSub?.cancel();
  await bleService.disconnect();

  // Clear all graph history so next session starts fresh
  bleService.clearHistory();

  setState(() {
    connected        = false;
    spinEnabled      = false;
    hoverEnabled     = false;
    droneOn          = false;
```

```
    distanceMm       = null;
    voltage          = null;
    current          = null;
    mahUsed          = null;
    batteryRemaining = null;
    pitch            = null;
    roll             = null;
    yaw              = null;
    flightMode       = null;
    rpm              = null;
    statusMessage    = "Disconnected";
  });
}


// ---------------------------------------------------------------------------
// Widgets
// ---------------------------------------------------------------------------


/// A standard labelled data row used for most telemetry fields.
Widget _dataRow(String label, String value, {Color valueColor = _textPrim}) {
  return Container(
    margin: const EdgeInsets.symmetric(vertical: 4, horizontal: 16),
    padding: const EdgeInsets.symmetric(vertical: 12, horizontal: 16),
    decoration: BoxDecoration(
      color: _surface,
      border: Border.all(color: _border),
```

```
      borderRadius: BorderRadius.circular(8),
    ),
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: [
        Text(label,
            style: const TextStyle(
                color: _textSec, fontSize: 12,
                letterSpacing: 1.2, fontFamily: 'monospace')),
        Text(value,
            style: TextStyle(
                color: valueColor, fontSize: 15,
                fontWeight: FontWeight.w700, fontFamily: 'monospace')),
      ],
    ),
  );
}

/// Section header with cyan left-bar accent.
Widget _sectionHeader(String title) {
  return Padding(
    padding: const EdgeInsets.fromLTRB(16, 20, 16, 6),
    child: Row(
      children: [
        Container(width: 3, height: 14, color: _accent,
            margin: const EdgeInsets.only(right: 10)),
```

```
        Text(title,
            style: const TextStyle(
                color: _accent, fontSize: 11,
                fontWeight: FontWeight.w800,
                letterSpacing: 2.5, fontFamily: 'monospace')),
      ],
    ),
  );
}

/// Toggle switch row styled to match the theme.
Widget _toggleRow(String label, bool value, bool enabled,
    Future<void> Function(bool) onChanged) {
  return Container(
    margin: const EdgeInsets.symmetric(vertical: 4, horizontal: 16),
    padding: const EdgeInsets.symmetric(vertical: 10, horizontal: 16),
    decoration: BoxDecoration(
      color: _surface,
      border: Border.all(
          color: (enabled && value) ? _accent.withOpacity(0.5) : _border),
      borderRadius: BorderRadius.circular(8),
    ),
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: [
        Text(label,
```

```
            style: TextStyle(
                color: enabled ? _textPrim : _textSec,
                fontSize: 13, letterSpacing: 1,
                fontFamily: 'monospace')),
        Switch(
          value: value,
          onChanged: enabled ? (v) => onChanged(v) : null,
          activeColor: _accent,
          activeTrackColor: _accent.withOpacity(0.25),
          inactiveThumbColor: _textSec,
          inactiveTrackColor: _border,
        ),
      ],
    ),
  );
}

Color _batteryColor(int? pct) {
  if (pct == null) return _textSec;
  if (pct > 50)   return _good;
  if (pct > 20)   return _warn;
  return _danger;
}

Color _voltageColor(double? v) {
  if (v == null) return _textSec;
```

```
  if (v > 11.0)  return _good;
  if (v > 10.5)  return _warn;
  return _danger;
}

@override
Widget build(BuildContext context) {
  return Theme(
    data: ThemeData.dark().copyWith(
      scaffoldBackgroundColor: _bg,
      appBarTheme: const AppBarTheme(
        backgroundColor: _surface,
        foregroundColor: _textPrim,
        elevation: 0,
        titleTextStyle: TextStyle(
          color: _textPrim, fontSize: 14,
          fontWeight: FontWeight.w700,
          letterSpacing: 2, fontFamily: 'monospace',
        ),
      ),
    ),
    child: Scaffold(
      key: _scaffoldKey,
      appBar: AppBar(
        title: const Text("DRONENADO"),
        leading: IconButton(
```

```
          icon: const Icon(Icons.menu, color: _accent),
          tooltip: "Telemetry Graphs",
          onPressed: () => _scaffoldKey.currentState?.openDrawer(),
        ),
        bottom: PreferredSize(
          preferredSize: const Size.fromHeight(1),
          child: Container(color: _border, height: 1),
        ),
      ),
      drawer: TelemetryDrawer(bleService: bleService),
      backgroundColor: _bg,
      body: SingleChildScrollView(
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.stretch,
          children: [


            // ---- CONNECTION STATUS BAR ----
            AnimatedContainer(
              duration: const Duration(milliseconds: 300),
              margin: const EdgeInsets.fromLTRB(16, 16, 16, 0),
              padding: const EdgeInsets.symmetric(vertical: 10, horizontal:
16),
              decoration: BoxDecoration(
                color: _surface,
                border: Border.all(
                    color: connected
```

```
                ? _good.withOpacity(0.6)
                : connecting
                    ? _warn.withOpacity(0.6)
                    : _border),
          borderRadius: BorderRadius.circular(8),
        ),
        child: Row(
          children: [
            // Status dot
            Container(
              width: 8, height: 8,
              decoration: BoxDecoration(
                shape: BoxShape.circle,
                color: connected ? _good
                    : connecting ? _warn
                    : _textSec,
              ),
            ),
            const SizedBox(width: 10),
            Expanded(
              child: Text(
                statusMessage.isEmpty ? "Not connected" : statusMessage,
                style: TextStyle(
                  color: connected ? _good
                      : connecting ? _warn
                      : _textSec,
```

```
            fontSize: 12, letterSpacing: 1,
            fontFamily: 'monospace',
          ),
        ),
      ),
      // Connect / Disconnect button
      GestureDetector(
        onTap: connecting ? null
            : connected ? _disconnectBle
            : _connectBle,
        child: Container(
          padding: const EdgeInsets.symmetric(
              horizontal: 14, vertical: 6),
          decoration: BoxDecoration(
            color: connected
                ? _danger.withOpacity(0.15)
                : _accent.withOpacity(0.15),
            border: Border.all(
                color: connected ? _danger : _accent,
                width: 1),
            borderRadius: BorderRadius.circular(6),
          ),
          child: connecting
              ? const SizedBox(
                  width: 14, height: 14,
                  child: CircularProgressIndicator(
```

```
                            strokeWidth: 1.5, color: _warn))
                    : Text(
                        connected ? "DISCONNECT" : "CONNECT",
                        style: TextStyle(
                          color: connected ? _danger : _accent,
                          fontSize: 11, fontWeight: FontWeight.w800,
                          letterSpacing: 1.5, fontFamily: 'monospace',
                        ),
                      ),
                  ),
                ),
              ],
            ),
          ),

          // ---- DISTANCE HERO ----
          _sectionHeader("FLOOR DISTANCE"),
          Container(
            margin: const EdgeInsets.symmetric(horizontal: 16),
            padding: const EdgeInsets.symmetric(vertical: 20),
            decoration: BoxDecoration(
              color: _surface,
              border: Border.all(color: _border),
              borderRadius: BorderRadius.circular(8),
            ),
            child: Column(
```

```
              children: [
                Row(
                  mainAxisAlignment: MainAxisAlignment.center,
                  crossAxisAlignment: CrossAxisAlignment.end,
                  children: [
                    Text(
                      distanceMm != null ? "$distanceMm" : "--",
                      style: const TextStyle(
                        color: _accent, fontSize: 64,
                        fontWeight: FontWeight.w900,
                        fontFamily: 'monospace', height: 1,
                      ),
                    ),
                    const Padding(
                      padding: EdgeInsets.only(bottom: 10, left: 8),
                      child: Text("mm",
                          style: TextStyle(
                              color: _textSec, fontSize: 18,
                              fontFamily: 'monospace')),
                    ),
                  ],
                ),
                const SizedBox(height: 6),
                // Simple bar indicator 0-2000mm
                Padding(
                  padding: const EdgeInsets.symmetric(horizontal: 32),
```

```
                child: ClipRRect(
                  borderRadius: BorderRadius.circular(4),
                  child: LinearProgressIndicator(
                    value: distanceMm != null
                        ? (distanceMm! / 2000).clamp(0.0, 1.0)
                        : 0,
                    minHeight: 4,
                    backgroundColor: _border,
                    valueColor:
                        const AlwaysStoppedAnimation<Color>(_accent),
                  ),
                ),
              ),
            ],
          ),
        ),

        // ---- CONTROLS ----
        _sectionHeader("CONTROLS"),
        _toggleRow("DRONE POWER", droneOn, connected,
            (v) async {
              setState(() => droneOn = v);
              await bleService.setDronePower(v);
            }),
        _toggleRow("HOVER MODE", hoverEnabled, connected && droneOn,
            (v) async {
```

```
        setState(() => hoverEnabled = v);
        await bleService.setHover(v);
      }),
    _toggleRow("SPIN MODE", spinEnabled, connected && droneOn,
      (v) async {
        setState(() => spinEnabled = v);
        await bleService.setSpin(v);
      }),

    // ---- BATTERY ----
    _sectionHeader("BATTERY"),
    _dataRow("VOLTAGE",
      voltage != null ? "${voltage!.toStringAsFixed(2)} V" : "--",
      valueColor: _voltageColor(voltage)),
    _dataRow("CURRENT",
      current != null ? "${current!.toStringAsFixed(2)} A" : "--"),
    _dataRow("MAH USED",
      mahUsed != null ? "$mahUsed mAh" : "--"),
    _dataRow("REMAINING",
      batteryRemaining != null ? "$batteryRemaining %" : "--",
      valueColor: _batteryColor(batteryRemaining)),

    // ---- ATTITUDE ----
    _sectionHeader("ATTITUDE"),
    _dataRow("PITCH",
      pitch != null ? "${pitch!.toStringAsFixed(1)}°" : "--"),
```

```
            _dataRow("ROLL",
                roll != null ? "${roll!.toStringAsFixed(1)}°" : "--"),
            _dataRow("YAW",
                yaw != null ? "${yaw!.toStringAsFixed(1)}°" : "--"),
            _dataRow("RPM",
                rpm != null ? rpm!.toStringAsFixed(1) : "--",
                valueColor: (rpm != null && rpm!.abs() > 10) ? _accent :
_textSec),


            // ---- FLIGHT MODE ----
            _sectionHeader("FLIGHT MODE"),
            _dataRow("MODE", flightMode ?? "--",
                valueColor: flightMode != null ? _good : _textSec),


            const SizedBox(height: 80),
          ],
        ),
      ),
    ),
  );
 }
}
```

BLE Telemetry

```
import 'dart:convert';
```

```dart
import 'dart:typed_data';

import 'package:flutter/foundation.dart';

import 'package:flutter_blue_plus/flutter_blue_plus.dart';


// ---- UUIDS ----

const String serviceUUID        = "FEED";

const String distanceCharUUID   = "5748";

const String batteryCharUUID    = "5749";

const String attitudeCharUUID   = "574A";

const String flightModeCharUUID = "574B";

const String droneOnCharUUID    = "574C";

const String hoverCharUUID      = "574D";

const String spinCharUUID       = "574E";

const String lidarMapCharUUID   = "574F";


class _YawSample {

 final double yaw;

 final DateTime time;

 _YawSample(this.yaw, this.time);

}


// ---------------------------------------------------------------------------

// Snapshot types

// ---------------------------------------------------------------------------


class BatterySnapshot {
```

```
 final DateTime time;
 final double voltage;
 final double current;
 final int remaining;
 BatterySnapshot({required this.time, required this.voltage,
     required this.current, required this.remaining});
}


class AttitudeSnapshot {
 final DateTime time;
 final double pitch;
 final double roll;
 final double yaw;
 AttitudeSnapshot({required this.time, required this.pitch,
     required this.roll, required this.yaw});
}


class FloorSnapshot {
 final DateTime time;
 final double distanceMm;
 FloorSnapshot({required this.time, required this.distanceMm});
}


class RpmSnapshot {
 final DateTime time;
 final double rpm;
```

```
 RpmSnapshot({required this.time, required this.rpm});
}

// ---------------------------------------------------------------------------
// BleDistanceService
// ---------------------------------------------------------------------------

class BleDistanceService {
 BluetoothDevice? device;

 BluetoothCharacteristic? distanceChar;
 BluetoothCharacteristic? batteryChar;
 BluetoothCharacteristic? attitudeChar;
 BluetoothCharacteristic? flightModeChar;
 BluetoothCharacteristic? droneOnChar;
 BluetoothCharacteristic? hoverChar;
 BluetoothCharacteristic? spinChar;
 BluetoothCharacteristic? lidarMapChar;

 Stream<int>?                  distanceStream;
 Stream<Map<String, dynamic>>? batteryStream;
 Stream<Map<String, double?>>? attitudeStream;
 Stream<String>?               flightModeStream;
 Stream<bool>?                 droneOnStream;
 Stream<bool>?                 hoverStream;
 Stream<bool>?                 spinStream;
```

```
Stream<List<int>>?            lidarMapStream;


static const int kMaxHistory = 2000;


final List<FloorSnapshot>    floorHistory    = [];

final List<BatterySnapshot>  batteryHistory  = [];

final List<AttitudeSnapshot> attitudeHistory = [];

final List<RpmSnapshot>      rpmHistory      = [];

final List<int>              lidarMap        = List.filled(36, 0);


// RPM internals

final List<_YawSample> _yawHistory = [];

double? _lastRawYaw;

double  _continuousYaw = 0;

double? _filteredRpm;

double? _lastRpm;

final double _rpmWindowSeconds = 0.8;

final double _rpmFilterAlpha   = 0.15;   // slower EMA — less reactive to
noise


// Session average RPM (absolute, spinning direction agnostic)

double  _rpmSessionSum   = 0;

int     _rpmSessionCount = 0;

double? sessionAvgRpm;   // public — read by the drawer


// ---------------------------------------------------------------------------
```

```
// clearHistory — wipes all graph buffers and RPM state on disconnect
// ---------------------------------------------------------------------------


void clearHistory() {
  floorHistory.clear();
  batteryHistory.clear();
  attitudeHistory.clear();
  rpmHistory.clear();
  lidarMap.fillRange(0, lidarMap.length, 0);


  _yawHistory.clear();
  _lastRawYaw      = null;
  _continuousYaw   = 0;
  _filteredRpm     = null;
  _lastRpm         = null;
  _rpmSessionSum   = 0;
  _rpmSessionCount = 0;
  sessionAvgRpm    = null;
}


// ---------------------------------------------------------------------------
// connect
// ---------------------------------------------------------------------------


Future<void> connect() async {
  if (!await FlutterBluePlus.isOn) throw Exception("Bluetooth is OFF");
```

```
  device = null;

  await FlutterBluePlus.startScan(timeout: const Duration(seconds: 5));
  final sub = FlutterBluePlus.scanResults.listen((results) {
    for (final r in results) {
      final uuids = r.advertisementData.serviceUuids
          .map((u) => u.toString().toUpperCase());
      if (uuids.contains(serviceUUID)) { device = r.device; break; }
    }
  });
  await Future.delayed(const Duration(seconds: 5));
  await sub.cancel();
  await FlutterBluePlus.stopScan();

  if (device == null) throw Exception("ESP32 not found");

  await device!.connect(autoConnect: false, timeout: const Duration(seconds:
10));
  await device!.connectionState
      .firstWhere((s) => s == BluetoothConnectionState.connected);

  debugPrint("Connected to ESP32");

  final services = await device!.discoverServices();
  final service  = services.firstWhere(
```

```
    (s) => s.uuid.toString().toUpperCase() == serviceUUID,
    orElse: () => throw Exception("Telemetry service not found"),
  );

  for (final c in service.characteristics) {
    final uuid = c.uuid.toString().toUpperCase();
    if (uuid == distanceCharUUID)   distanceChar   = c;
    if (uuid == batteryCharUUID)    batteryChar    = c;
    if (uuid == attitudeCharUUID)   attitudeChar   = c;
    if (uuid == flightModeCharUUID) flightModeChar = c;
    if (uuid == droneOnCharUUID)    droneOnChar    = c;
    if (uuid == hoverCharUUID)      hoverChar      = c;
    if (uuid == spinCharUUID)       spinChar       = c;
    if (uuid == lidarMapCharUUID)   lidarMapChar   = c;
  }

  if (distanceChar == null) throw Exception("Distance characteristic not
found");

  await distanceChar!.setNotifyValue(true);
  if (batteryChar    != null) await batteryChar!.setNotifyValue(true);
  if (attitudeChar   != null) await attitudeChar!.setNotifyValue(true);
  if (flightModeChar != null) await flightModeChar!.setNotifyValue(true);
  if (lidarMapChar   != null) await lidarMapChar!.setNotifyValue(true);

  distanceStream = distanceChar!.onValueReceived.map((v) {
```

```
    if (v.length < 2) return -1;
    final d = v[0] | (v[1] << 8);
    if (d > 0) _addFloor(d.toDouble());
    return d;
  });

  batteryStream = batteryChar?.onValueReceived.map((v) {
    final data = ByteData.sublistView(Uint8List.fromList(v));
    final snap = BatterySnapshot(
      time:      DateTime.now(),
      voltage:   data.getFloat32(0, Endian.little),
      current:   data.getFloat32(4, Endian.little),
      remaining: data.getUint8(12),
    );
    _addBattery(snap);
    return {
      "voltage":   snap.voltage,
      "current":   snap.current,
      "mah_used":  data.getUint32(8, Endian.little),
      "remaining": snap.remaining,
    };
  });

  attitudeStream = attitudeChar?.onValueReceived.map((v) {
    final data  = ByteData.sublistView(Uint8List.fromList(v));
    final pitch = data.getFloat32(0, Endian.little);
```

```
    final roll  = data.getFloat32(4, Endian.little);

    final yaw   = data.getFloat32(8, Endian.little);

    final now   = DateTime.now();



    _addAttitude(AttitudeSnapshot(time: now, pitch: pitch, roll: roll, yaw:
yaw));

    final rpm = _computeRpm(yaw, now);

    if (rpm != null) _addRpm(RpmSnapshot(time: now, rpm: rpm));



    return <String, double?>{

      "pitch": pitch, "roll": roll, "yaw": yaw, "rpm": _filteredRpm,

    };

  });



  flightModeStream = flightModeChar?.onValueReceived.map((v) =>
utf8.decode(v));



  if (droneOnChar != null) {

    droneOnStream = droneOnChar!.lastValueStream.map((d) => d.isNotEmpty &&
d[0] == 1);

  }

  if (hoverChar != null) {

    hoverStream = hoverChar!.lastValueStream.map((d) => d.isNotEmpty && d[0]
== 1);

  }

  if (spinChar != null) {

    spinStream = spinChar!.lastValueStream.map((d) => d.isNotEmpty && d[0] ==
1);
```

```
  }

  if (lidarMapChar != null) {
    lidarMapStream = lidarMapChar!.onValueReceived.map((v) {
      if (v.length >= 72) {
        final data = ByteData.sublistView(Uint8List.fromList(v));
        for (int i = 0; i < 36; i++) {
          final d = data.getUint16(i * 2, Endian.little);
          if (d > 0 && (lidarMap[i] == 0 || d < lidarMap[i])) lidarMap[i] = d;
        }
      }
      return List<int>.from(lidarMap);
    });
  }

  debugPrint("Telemetry streams ready");
}

// ---------------------------------------------------------------------------
// History helpers
// ---------------------------------------------------------------------------

void _addFloor(double d) {
  floorHistory.add(FloorSnapshot(time: DateTime.now(), distanceMm: d));
  if (floorHistory.length > kMaxHistory) floorHistory.removeAt(0);
}
```

```
void _addBattery(BatterySnapshot s) {
  batteryHistory.add(s);
  if (batteryHistory.length > kMaxHistory) batteryHistory.removeAt(0);
}

void _addAttitude(AttitudeSnapshot s) {
  attitudeHistory.add(s);
  if (attitudeHistory.length > kMaxHistory) attitudeHistory.removeAt(0);
}

void _addRpm(RpmSnapshot s) {
  rpmHistory.add(s);
  if (rpmHistory.length > kMaxHistory) rpmHistory.removeAt(0);
}

// ---------------------------------------------------------------------------
// RPM
// ---------------------------------------------------------------------------

double? _computeRpm(double yaw, DateTime now) {
  // -----------------------------------------------------------------
  // 1. YAW UNWRAP — always relative to the previous raw reading.
  //    On the very first sample (_lastRawYaw == null) we just anchor
  //    _continuousYaw without accumulating any delta, preventing the
  //    giant first-frame spike that caused inter-session blow-ups.
```

```
  // ---------------------------------------------------------------
  if (_lastRawYaw != null) {
    double delta = yaw - _lastRawYaw!;
    if (delta >  180) delta -= 360;
    if (delta < -180) delta += 360;
    // Ignore jitter below 0.2° (tightened from 0.05 to suppress noise at
rest)
    if (delta.abs() < 0.2) delta = 0;
    _continuousYaw += delta;
  }
  _lastRawYaw = yaw;

  // ---------------------------------------------------------------
  // 2. WINDOWED SAMPLE BUFFER
  // ---------------------------------------------------------------
  _yawHistory.add(_YawSample(_continuousYaw, now));
  _yawHistory.removeWhere(
    (s) => now.difference(s.time).inMilliseconds / 1000.0 > _rpmWindowSeconds,
  );

  // ---------------------------------------------------------------
  // 3. LINEAR REGRESSION over the window
  // ---------------------------------------------------------------
  double? computedRpm;
  if (_yawHistory.length >= 4) {   // raised minimum from 3 → 4 for stability
    double sumT = 0, sumYaw = 0, sumTYaw = 0, sumTT = 0;
```

```
    final firstTime = _yawHistory.first.time;
    for (final s in _yawHistory) {
      final t = s.time.difference(firstTime).inMilliseconds / 1000.0;
      sumT += t; sumYaw += s.yaw; sumTYaw += t * s.yaw; sumTT += t * t;
    }
    final n     = _yawHistory.length.toDouble();
    final denom = n * sumTT - sumT * sumT;
    if (denom.abs() > 1e-6) {
      final slope = (n * sumTYaw - sumT * sumYaw) / denom;
      final raw   = (slope / 360.0) * 60.0;
      // Hard cap: physically impossible for this drone to exceed 300 RPM
      if (raw.abs() <= 300) computedRpm = raw;
    }
  }

  // ---------------------------------------------------------------
  // 4. EMA FILTER — only feed real values; hold last on null
  // ---------------------------------------------------------------
  if (computedRpm != null) {
    if (_filteredRpm == null) {
      // First valid reading: seed the filter instead of jumping
      _filteredRpm = computedRpm;
    } else {
      _filteredRpm = _rpmFilterAlpha * computedRpm +
                     (1.0 - _rpmFilterAlpha) * _filteredRpm!;
    }
```

```
  }

  // ---------------------------------------------------------------
  // 5. SPIKE GUARD — tightened to 100 RPM/frame (was 200)
  // ---------------------------------------------------------------
  if (_filteredRpm != null && _lastRpm != null &&
      (_filteredRpm! - _lastRpm!).abs() > 100) {
    _filteredRpm = _lastRpm;
  }
  _lastRpm = _filteredRpm;

  // ---------------------------------------------------------------
  // 6. SESSION AVERAGE — only count samples where drone is meaningfully
  //    spinning (|rpm| > 5 to exclude noise at rest)
  // ---------------------------------------------------------------
  if (_filteredRpm != null && _filteredRpm!.abs() > 5) {
    _rpmSessionSum   += _filteredRpm!.abs();
    _rpmSessionCount += 1;
    sessionAvgRpm     = _rpmSessionSum / _rpmSessionCount;
  }

  return _filteredRpm;
}

// ---------------------------------------------------------------------------
// Write helpers
```

```
 // ---------------------------------------------------------------------------


 Future<void> setDronePower(bool on) async => droneOnChar?.write([on ? 1 : 0]);

 Future<void> setHover(bool enabled)      async => hoverChar?.write([enabled ?
1 : 0]);

 Future<void> setSpin(bool enabled)       async => spinChar?.write([enabled ? 1
: 0]);



 Future<void> disconnect() async {

   await device?.disconnect();

   device = null;

 }

}

import 'dart:convert';

import 'dart:typed_data';

import 'package:flutter/foundation.dart';

import 'package:flutter_blue_plus/flutter_blue_plus.dart';



// ---- UUIDS ----

const String serviceUUID        = "FEED";

const String distanceCharUUID   = "5748";

const String batteryCharUUID    = "5749";

const String attitudeCharUUID   = "574A";

const String flightModeCharUUID = "574B";

const String droneOnCharUUID    = "574C";

const String hoverCharUUID      = "574D";

const String spinCharUUID       = "574E";
```

```
const String lidarMapCharUUID   = "574F";


class _YawSample {
 final double yaw;
 final DateTime time;
 _YawSample(this.yaw, this.time);
}


// ---------------------------------------------------------------------------
// Snapshot types
// ---------------------------------------------------------------------------


class BatterySnapshot {
 final DateTime time;
 final double voltage;
 final double current;
 final int remaining;
 BatterySnapshot({required this.time, required this.voltage,
     required this.current, required this.remaining});
}


class AttitudeSnapshot {
 final DateTime time;
 final double pitch;
 final double roll;
 final double yaw;
```

```
 AttitudeSnapshot({required this.time, required this.pitch,
     required this.roll, required this.yaw});
}

class FloorSnapshot {
 final DateTime time;
 final double distanceMm;
 FloorSnapshot({required this.time, required this.distanceMm});
}

class RpmSnapshot {
 final DateTime time;
 final double rpm;
 RpmSnapshot({required this.time, required this.rpm});
}

// ---------------------------------------------------------------------------
// BleDistanceService
// ---------------------------------------------------------------------------

class BleDistanceService {
 BluetoothDevice? device;

 BluetoothCharacteristic? distanceChar;
 BluetoothCharacteristic? batteryChar;
 BluetoothCharacteristic? attitudeChar;
```

```
BluetoothCharacteristic? flightModeChar;
BluetoothCharacteristic? droneOnChar;
BluetoothCharacteristic? hoverChar;
BluetoothCharacteristic? spinChar;
BluetoothCharacteristic? lidarMapChar;

Stream<int>?                  distanceStream;
Stream<Map<String, dynamic>>? batteryStream;
Stream<Map<String, double?>>? attitudeStream;
Stream<String>?               flightModeStream;
Stream<bool>?                 droneOnStream;
Stream<bool>?                 hoverStream;
Stream<bool>?                 spinStream;
Stream<List<int>>?            lidarMapStream;

static const int kMaxHistory = 2000;

final List<FloorSnapshot>    floorHistory    = [];
final List<BatterySnapshot>  batteryHistory  = [];
final List<AttitudeSnapshot> attitudeHistory = [];
final List<RpmSnapshot>      rpmHistory      = [];
final List<int>              lidarMap        = List.filled(36, 0);

// RPM internals
final List<_YawSample> _yawHistory = [];
double? _lastRawYaw;
```

```
 double  _continuousYaw = 0;

 double? _filteredRpm;

 double? _lastRpm;

 final double _rpmWindowSeconds = 0.8;

 final double _rpmFilterAlpha   = 0.15;   // slower EMA — less reactive to
noise



 // Session average RPM (absolute, spinning direction agnostic)

 double  _rpmSessionSum   = 0;

 int     _rpmSessionCount = 0;

 double? sessionAvgRpm;   // public — read by the drawer



 // ---------------------------------------------------------------------------

 // clearHistory — wipes all graph buffers and RPM state on disconnect

 // ---------------------------------------------------------------------------



 void clearHistory() {

   floorHistory.clear();

   batteryHistory.clear();

   attitudeHistory.clear();

   rpmHistory.clear();

   lidarMap.fillRange(0, lidarMap.length, 0);



   _yawHistory.clear();

   _lastRawYaw      = null;

   _continuousYaw   = 0;
```

```
  _filteredRpm     = null;
  _lastRpm         = null;
  _rpmSessionSum   = 0;
  _rpmSessionCount = 0;
  sessionAvgRpm    = null;
}

// ---------------------------------------------------------------------------
// connect
// ---------------------------------------------------------------------------

Future<void> connect() async {
  if (!await FlutterBluePlus.isOn) throw Exception("Bluetooth is OFF");

  device = null;

  await FlutterBluePlus.startScan(timeout: const Duration(seconds: 5));
  final sub = FlutterBluePlus.scanResults.listen((results) {
    for (final r in results) {
      final uuids = r.advertisementData.serviceUuids
          .map((u) => u.toString().toUpperCase());
      if (uuids.contains(serviceUUID)) { device = r.device; break; }
    }
  });
  await Future.delayed(const Duration(seconds: 5));
  await sub.cancel();
```

```
  await FlutterBluePlus.stopScan();


  if (device == null) throw Exception("ESP32 not found");


  await device!.connect(autoConnect: false, timeout: const Duration(seconds:
10));
  await device!.connectionState

      .firstWhere((s) => s == BluetoothConnectionState.connected);


  debugPrint("Connected to ESP32");


  final services = await device!.discoverServices();
  final service  = services.firstWhere(

    (s) => s.uuid.toString().toUpperCase() == serviceUUID,

    orElse: () => throw Exception("Telemetry service not found"),

  );


  for (final c in service.characteristics) {

    final uuid = c.uuid.toString().toUpperCase();

    if (uuid == distanceCharUUID)   distanceChar   = c;

    if (uuid == batteryCharUUID)    batteryChar    = c;

    if (uuid == attitudeCharUUID)   attitudeChar   = c;

    if (uuid == flightModeCharUUID) flightModeChar = c;

    if (uuid == droneOnCharUUID)    droneOnChar    = c;

    if (uuid == hoverCharUUID)      hoverChar      = c;

    if (uuid == spinCharUUID)       spinChar       = c;
```

```
    if (uuid == lidarMapCharUUID)   lidarMapChar   = c;
  }

  if (distanceChar == null) throw Exception("Distance characteristic not
found");

  await distanceChar!.setNotifyValue(true);
  if (batteryChar    != null) await batteryChar!.setNotifyValue(true);
  if (attitudeChar   != null) await attitudeChar!.setNotifyValue(true);
  if (flightModeChar != null) await flightModeChar!.setNotifyValue(true);
  if (lidarMapChar   != null) await lidarMapChar!.setNotifyValue(true);

  distanceStream = distanceChar!.onValueReceived.map((v) {
    if (v.length < 2) return -1;
    final d = v[0] | (v[1] << 8);
    if (d > 0) _addFloor(d.toDouble());
    return d;
  });

  batteryStream = batteryChar?.onValueReceived.map((v) {
    final data = ByteData.sublistView(Uint8List.fromList(v));
    final snap = BatterySnapshot(
      time:      DateTime.now(),
      voltage:   data.getFloat32(0, Endian.little),
      current:   data.getFloat32(4, Endian.little),
      remaining: data.getUint8(12),
```

```
    );

    _addBattery(snap);

    return {

      "voltage":   snap.voltage,

      "current":   snap.current,

      "mah_used":  data.getUint32(8, Endian.little),

      "remaining": snap.remaining,

    };

  });



  attitudeStream = attitudeChar?.onValueReceived.map((v) {

    final data  = ByteData.sublistView(Uint8List.fromList(v));

    final pitch = data.getFloat32(0, Endian.little);

    final roll  = data.getFloat32(4, Endian.little);

    final yaw   = data.getFloat32(8, Endian.little);

    final now   = DateTime.now();



    _addAttitude(AttitudeSnapshot(time: now, pitch: pitch, roll: roll, yaw:
yaw));

    final rpm = _computeRpm(yaw, now);

    if (rpm != null) _addRpm(RpmSnapshot(time: now, rpm: rpm));



    return <String, double?>{

      "pitch": pitch, "roll": roll, "yaw": yaw, "rpm": _filteredRpm,

    };

  });
```

```
   flightModeStream = flightModeChar?.onValueReceived.map((v) =>
utf8.decode(v));



   if (droneOnChar != null) {

     droneOnStream = droneOnChar!.lastValueStream.map((d) => d.isNotEmpty &&
d[0] == 1);

   }

   if (hoverChar != null) {

     hoverStream = hoverChar!.lastValueStream.map((d) => d.isNotEmpty && d[0]
== 1);

   }

   if (spinChar != null) {

     spinStream = spinChar!.lastValueStream.map((d) => d.isNotEmpty && d[0] ==
1);

   }



   if (lidarMapChar != null) {

     lidarMapStream = lidarMapChar!.onValueReceived.map((v) {

       if (v.length >= 72) {

         final data = ByteData.sublistView(Uint8List.fromList(v));

         for (int i = 0; i < 36; i++) {

           final d = data.getUint16(i * 2, Endian.little);

           if (d > 0 && (lidarMap[i] == 0 || d < lidarMap[i])) lidarMap[i] = d;

         }

       }

       return List<int>.from(lidarMap);

     });
```

```
  }

  debugPrint("Telemetry streams ready");
}

// ---------------------------------------------------------------------------
// History helpers
// ---------------------------------------------------------------------------

void _addFloor(double d) {
  floorHistory.add(FloorSnapshot(time: DateTime.now(), distanceMm: d));
  if (floorHistory.length > kMaxHistory) floorHistory.removeAt(0);
}

void _addBattery(BatterySnapshot s) {
  batteryHistory.add(s);
  if (batteryHistory.length > kMaxHistory) batteryHistory.removeAt(0);
}

void _addAttitude(AttitudeSnapshot s) {
  attitudeHistory.add(s);
  if (attitudeHistory.length > kMaxHistory) attitudeHistory.removeAt(0);
}

void _addRpm(RpmSnapshot s) {
  rpmHistory.add(s);
```

```
  if (rpmHistory.length > kMaxHistory) rpmHistory.removeAt(0);
 }

 // ---------------------------------------------------------------------------
 // RPM
 // ---------------------------------------------------------------------------

 double? _computeRpm(double yaw, DateTime now) {
   // ---------------------------------------------------------------------
   // 1. YAW UNWRAP — always relative to the previous raw reading.
   //    On the very first sample (_lastRawYaw == null) we just anchor
   //    _continuousYaw without accumulating any delta, preventing the
   //    giant first-frame spike that caused inter-session blow-ups.
   // ---------------------------------------------------------------------
   if (_lastRawYaw != null) {
     double delta = yaw - _lastRawYaw!;
     if (delta >  180) delta -= 360;
     if (delta < -180) delta += 360;
     // Ignore jitter below 0.2° (tightened from 0.05 to suppress noise at
rest)
     if (delta.abs() < 0.2) delta = 0;
     _continuousYaw += delta;
   }
   _lastRawYaw = yaw;

   // ---------------------------------------------------------------------
```

```
// 2. WINDOWED SAMPLE BUFFER
// ---------------------------------------------------------------------
_yawHistory.add(_YawSample(_continuousYaw, now));
_yawHistory.removeWhere(
  (s) => now.difference(s.time).inMilliseconds / 1000.0 > _rpmWindowSeconds,
);

// ---------------------------------------------------------------------
// 3. LINEAR REGRESSION over the window
// ---------------------------------------------------------------------
double? computedRpm;
if (_yawHistory.length >= 4) {   // raised minimum from 3 → 4 for stability
  double sumT = 0, sumYaw = 0, sumTYaw = 0, sumTT = 0;
  final firstTime = _yawHistory.first.time;
  for (final s in _yawHistory) {
    final t = s.time.difference(firstTime).inMilliseconds / 1000.0;
    sumT += t; sumYaw += s.yaw; sumTYaw += t * s.yaw; sumTT += t * t;
  }
  final n     = _yawHistory.length.toDouble();
  final denom = n * sumTT - sumT * sumT;
  if (denom.abs() > 1e-6) {
    final slope = (n * sumTYaw - sumT * sumYaw) / denom;
    final raw   = (slope / 360.0) * 60.0;
    // Hard cap: physically impossible for this drone to exceed 300 RPM
    if (raw.abs() <= 300) computedRpm = raw;
  }
```

```
}

// ---------------------------------------------------------------------
// 4. EMA FILTER — only feed real values; hold last on null
// ---------------------------------------------------------------------
if (computedRpm != null) {
  if (_filteredRpm == null) {
    // First valid reading: seed the filter instead of jumping
    _filteredRpm = computedRpm;
  } else {
    _filteredRpm = _rpmFilterAlpha * computedRpm +
                   (1.0 - _rpmFilterAlpha) * _filteredRpm!;
  }
}

// ---------------------------------------------------------------------
// 5. SPIKE GUARD — tightened to 100 RPM/frame (was 200)
// ---------------------------------------------------------------------
if (_filteredRpm != null && _lastRpm != null &&
    (_filteredRpm! - _lastRpm!).abs() > 100) {
  _filteredRpm = _lastRpm;
}
_lastRpm = _filteredRpm;

// ---------------------------------------------------------------------
// 6. SESSION AVERAGE — only count samples where drone is meaningfully
```

```
    //    spinning (|rpm| > 5 to exclude noise at rest)
    // ------------------------------------------------------------------
    if (_filteredRpm != null && _filteredRpm!.abs() > 5) {
      _rpmSessionSum   += _filteredRpm!.abs();
      _rpmSessionCount += 1;
      sessionAvgRpm     = _rpmSessionSum / _rpmSessionCount;
    }

    return _filteredRpm;
  }

  // ------------------------------------------------------------------------
  // Write helpers
  // ------------------------------------------------------------------------

  Future<void> setDronePower(bool on) async => droneOnChar?.write([on ? 1 : 0]);
  Future<void> setHover(bool enabled)      async => hoverChar?.write([enabled ? 1 : 0]);
  Future<void> setSpin(bool enabled)       async => spinChar?.write([enabled ? 1 : 0]);

  Future<void> disconnect() async {
    await device?.disconnect();
    device = null;
  }
}
```

Telemetry Drawer

```
import 'dart:math';

import 'package:flutter/material.dart';

import '../services/ble_telemetry_service.dart';


// ---------------------------------------------------------------------------

// Palette & theme constants

// ---------------------------------------------------------------------------

const Color _bg        = Color(0xFF0A0E1A);

const Color _surface   = Color(0xFF121828);

const Color _border    = Color(0xFF1E2A40);

const Color _accent    = Color(0xFF00E5FF);

const Color _accentDim = Color(0xFF0077AA);

const Color _textPrim  = Color(0xFFE8F0FF);

const Color _textSec   = Color(0xFF6B80A0);

const Color _nearColor = Color(0xFFFF3B5C);

const Color _midColor  = Color(0xFFFFAA00);

const Color _farColor  = Color(0xFF00E5A0);


// ---------------------------------------------------------------------------

// TelemetryDrawer — the slide-in panel

// ---------------------------------------------------------------------------


class TelemetryDrawer extends StatelessWidget {

 final BleDistanceService bleService;
```

```
const TelemetryDrawer({super.key, required this.bleService});


@override
Widget build(BuildContext context) {
  return Drawer(
    backgroundColor: _bg,
    width: MediaQuery.of(context).size.width * 0.92,
    child: SafeArea(
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          // Header
          Padding(
            padding: const EdgeInsets.fromLTRB(20, 16, 20, 8),
            child: Row(
              children: [
                Container(
                  width: 3,
                  height: 22,
                  color: _accent,
                  margin: const EdgeInsets.only(right: 12),
                ),
                const Text(
                  "TELEMETRY",
                  style: TextStyle(
                    color: _accent,
```

```
                fontSize: 13,
                fontWeight: FontWeight.w800,
                letterSpacing: 3,
                fontFamily: 'monospace',
              ),
            ),
          ],
        ),
      ),
      const Divider(color: _border, height: 1),
      // Tab view
      Expanded(
        child: DefaultTabController(
          length: 5,
          child: Column(
            children: [
              TabBar(
                isScrollable: true,
                tabAlignment: TabAlignment.start,
                indicatorColor: _accent,
                indicatorWeight: 2,
                labelColor: _accent,
                unselectedLabelColor: _textSec,
                labelStyle: const TextStyle(
                  fontSize: 11,
                  fontWeight: FontWeight.w700,
```

```
                letterSpacing: 1.5,
                fontFamily: 'monospace',
              ),
              tabs: const [
                Tab(text: "WALL MAP"),
                Tab(text: "FLOOR"),
                Tab(text: "BATTERY"),
                Tab(text: "ATTITUDE"),
                Tab(text: "RPM"),
              ],
            ),
            const Divider(color: _border, height: 1),
            Expanded(
              child: TabBarView(
                physics: const NeverScrollableScrollPhysics(),
                children: [
                  _WallMapTab(bleService: bleService),
                  _FloorTab(bleService: bleService),
                  _BatteryTab(bleService: bleService),
                  _AttitudeTab(bleService: bleService),
                  _RpmTab(bleService: bleService),
                ],
              ),
            ),
          ],
        ),
```

```
            ),
          ),
        ],
      ),
    ),
  );
 }
}

// ═══════════════════════════════════════════════════════════════════════
// TAB 1 — Wall Polar Map
// ═══════════════════════════════════════════════════════════════════════

class _WallMapTab extends StatefulWidget {
 final BleDistanceService bleService;
 const _WallMapTab({required this.bleService});

 @override
 State<_WallMapTab> createState() => _WallMapTabState();
}

class _WallMapTabState extends State<_WallMapTab> {
 List<int> _sectors = List.filled(36, 0);

 @override
 void initState() {
```

```
  super.initState();
  widget.bleService.lidarMapStream?.listen((sectors) {
    if (mounted) setState(() => _sectors = sectors);
  });
  // Seed with whatever has already been merged
  _sectors = List<int>.from(widget.bleService.lidarMap);
}


@override
Widget build(BuildContext context) {
  final covered = _sectors.where((v) => v > 0).length;
  return Column(
    children: [
      const SizedBox(height: 12),
      Text(
        "$covered / 36 SECTORS",
        style: const TextStyle(
          color: _textSec,
          fontSize: 11,
          letterSpacing: 2,
          fontFamily: 'monospace',
        ),
      ),
      Expanded(
        child: Padding(
          padding: const EdgeInsets.all(16),
```

```
          child: LayoutBuilder(builder: (ctx, box) {
            final side = min(box.maxWidth, box.maxHeight);
            return Center(
              child: SizedBox(
                width: side,
                height: side,
                child: CustomPaint(
                  painter: _PolarPainter(sectors: _sectors),
                ),
              ),
            );
          }),
        ),
      ),
      // Legend
      Padding(
        padding: const EdgeInsets.only(bottom: 16),
        child: Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            _legendDot(_nearColor, "< 300 mm"),
            const SizedBox(width: 16),
            _legendDot(_midColor, "300–800 mm"),
            const SizedBox(width: 16),
            _legendDot(_farColor, "> 800 mm"),
          ],
```

```
        ),
      ),
    ],
  );
 }

 Widget _legendDot(Color c, String label) => Row(children: [
      Container(width: 10, height: 10, decoration: BoxDecoration(color: c,
shape: BoxShape.circle)),
      const SizedBox(width: 6),
      Text(label, style: const TextStyle(color: _textSec, fontSize: 11,
fontFamily: 'monospace')),
    ]);
}

class _PolarPainter extends CustomPainter {
 final List<int> sectors;
 const _PolarPainter({required this.sectors});

 Color _distColor(int mm) {
  if (mm <= 0)   return Colors.transparent;
  if (mm < 300)  return _nearColor.withOpacity(0.85);
  if (mm < 800)  return _midColor.withOpacity(0.75);
  return _farColor.withOpacity(0.65);
 }

 @override
```

```
void paint(Canvas canvas, Size size) {
  final cx = size.width / 2;
  final cy = size.height / 2;
  final maxR = cx * 0.88;
  final maxDist = 1500.0;

  // Grid rings
  final gridPaint = Paint()
    ..color = _border
    ..style = PaintingStyle.stroke
    ..strokeWidth = 0.8;

  for (final frac in [0.25, 0.5, 0.75, 1.0]) {
    canvas.drawCircle(Offset(cx, cy), maxR * frac, gridPaint);
  }

  // Grid spokes every 30°
  for (int i = 0; i < 12; i++) {
    final angle = i * pi / 6 - pi / 2;
    canvas.drawLine(
      Offset(cx, cy),
      Offset(cx + cos(angle) * maxR, cy + sin(angle) * maxR),
      gridPaint,
    );
  }
```

```
    // Distance labels on rings
    final labelStyle = const TextStyle(
      color: _textSec,
      fontSize: 9,
      fontFamily: 'monospace',
    );
    final tp = TextPainter(textDirection: TextDirection.ltr);
    for (final entry in {0.25: "375", 0.5: "750", 0.75: "1125", 1.0:
"1500"}.entries) {
      tp.text = TextSpan(text: "${entry.value}mm", style: labelStyle);
      tp.layout();
      tp.paint(canvas, Offset(cx + maxR * entry.key + 2, cy - tp.height / 2));
    }

    // Cardinal labels
    final cardStyle = TextStyle(
      color: _textPrim.withOpacity(0.6),
      fontSize: 10,
      fontFamily: 'monospace',
      fontWeight: FontWeight.bold,
    );
    for (final entry in {'N': -pi / 2, 'E': 0.0, 'S': pi / 2, 'W': pi}.entries)
{
      tp.text = TextSpan(text: entry.key, style: cardStyle);
      tp.layout();
      final angle = entry.value;
      final r = maxR + 12;
```

```
    tp.paint(canvas, Offset(cx + cos(angle) * r - tp.width / 2,
                            cy + sin(angle) * r - tp.height / 2));
  }

  // Sector wedges
  final sweepAngle = (2 * pi) / 36;

  for (int i = 0; i < 36; i++) {
    final dist = sectors[i];
    if (dist <= 0) continue;

    final r     = (dist.clamp(0, maxDist.toInt()) / maxDist) * maxR;
    final start = -pi / 2 + i * sweepAngle - sweepAngle / 2;

    // Filled wedge
    final fillPaint = Paint()
      ..color = _distColor(dist)
      ..style = PaintingStyle.fill;

    final path = Path()
      ..moveTo(cx, cy)
      ..arcTo(Rect.fromCircle(center: Offset(cx, cy), radius: r),
              start, sweepAngle, false)
      ..close();
    canvas.drawPath(path, fillPaint);
```

```
    // Outer arc highlight
    final strokePaint = Paint()
      ..color = _distColor(dist).withOpacity(1.0)
      ..style = PaintingStyle.stroke
      ..strokeWidth = 1.5;
    canvas.drawArc(
      Rect.fromCircle(center: Offset(cx, cy), radius: r),
      start, sweepAngle, false, strokePaint,
    );
  }

  // Centre dot
  canvas.drawCircle(
    Offset(cx, cy),
    5,
    Paint()..color = _accent,
  );
  canvas.drawCircle(
    Offset(cx, cy),
    3,
    Paint()..color = _bg,
  );
}

@override
bool shouldRepaint(_PolarPainter old) => old.sectors != sectors;
```

```
}


// ═══════════════════════════════════════════════════════════════════════
// TAB 2 — Floor Distance Line Graph
// ═══════════════════════════════════════════════════════════════════════


class _FloorTab extends StatefulWidget {
 final BleDistanceService bleService;
 const _FloorTab({required this.bleService});


 @override
 State<_FloorTab> createState() => _FloorTabState();
}


class _FloorTabState extends State<_FloorTab> {
 List<FloorSnapshot> _history = [];


 @override
 void initState() {
   super.initState();
   widget.bleService.distanceStream?.listen((_) {
     if (mounted) {
       setState(() {
         _history = List.from(widget.bleService.floorHistory);
       });
     }
```

```
  });
  _history = List.from(widget.bleService.floorHistory);
}

@override
Widget build(BuildContext context) {
  final current = _history.isNotEmpty ? _history.last.distanceMm : null;

  return Column(
    children: [
      const SizedBox(height: 16),
      // Current value hero
      Row(
        mainAxisAlignment: MainAxisAlignment.center,
        crossAxisAlignment: CrossAxisAlignment.end,
        children: [
          Text(
            current != null ? current.toStringAsFixed(0) : "--",
            style: const TextStyle(
              color: _accent,
              fontSize: 52,
              fontWeight: FontWeight.w900,
              fontFamily: 'monospace',
              height: 1,
            ),
          ),
```

```
          const Padding(
            padding: EdgeInsets.only(bottom: 8, left: 6),
            child: Text(
              "mm",
              style: TextStyle(color: _textSec, fontSize: 16, fontFamily:
'monospace'),
            ),
          ),
        ],
      ),
      const SizedBox(height: 4),
      const Text("FLOOR DISTANCE", style: TextStyle(color: _textSec, fontSize:
10, letterSpacing: 2)),
      const SizedBox(height: 16),
      Expanded(
        child: Padding(
          padding: const EdgeInsets.fromLTRB(8, 0, 16, 8),
          child: _history.length < 2
              ? const Center(
                  child: Text("Waiting for data…",
                      style: TextStyle(color: _textSec, fontFamily:
'monospace')))
              : CustomPaint(
                  painter: _FloorLinePainter(history: _history),
                  child: Container(),
                ),
        ),
```

```
      ),
    ],
  );
 }
}


class _FloorLinePainter extends CustomPainter {
 final List<FloorSnapshot> history;
 const _FloorLinePainter({required this.history});


 @override
 void paint(Canvas canvas, Size size) {
  if (history.length < 2) return;


  final w = size.width;
  final h = size.height;
  const padT = 20.0, padB = 30.0, padL = 50.0, padR = 8.0;
  final chartW = w - padL - padR;
  final chartH = h - padT - padB;


  // Y range: 0–2000 mm
  const yMin = 0.0, yMax = 2000.0;


  // X range: time
  final tStart = history.first.time.millisecondsSinceEpoch.toDouble();
  final tEnd   = history.last.time.millisecondsSinceEpoch.toDouble();
```

```
    final tRange = max(tEnd - tStart, 1.0);



    double toX(double t) => padL + ((t - tStart) / tRange) * chartW;

    double toY(double d) => padT + chartH - ((d - yMin) / (yMax - yMin)) *
chartH;



    // Grid

    final gridPaint = Paint()

      ..color = _border

      ..strokeWidth = 0.8;



    final tp = TextPainter(textDirection: TextDirection.ltr);

    final labelStyle = const TextStyle(color: _textSec, fontSize: 9, fontFamily:
'monospace');



    for (final mm in [0, 500, 1000, 1500, 2000]) {

      final y = toY(mm.toDouble());

      canvas.drawLine(Offset(padL, y), Offset(w - padR, y), gridPaint);

      tp.text = TextSpan(text: "${mm}mm", style: labelStyle);

      tp.layout();

      tp.paint(canvas, Offset(0, y - tp.height / 2));

    }



    // Gradient fill

    final fillPath = Path();

    fillPath.moveTo(toX(tStart), toY(history.first.distanceMm));

    for (final s in history) {
```

```
    fillPath.lineTo(toX(s.time.millisecondsSinceEpoch.toDouble()),
toY(s.distanceMm));

  }

  fillPath.lineTo(toX(tEnd), toY(0));

  fillPath.lineTo(toX(tStart), toY(0));

  fillPath.close();



  canvas.drawPath(

    fillPath,

    Paint()

      ..shader = LinearGradient(

        begin: Alignment.topCenter,

        end: Alignment.bottomCenter,

        colors: [_accent.withOpacity(0.3), _accent.withOpacity(0.02)],

      ).createShader(Rect.fromLTWH(0, padT, w, chartH)),

  );



  // Line

  final linePath = Path();

  linePath.moveTo(toX(tStart), toY(history.first.distanceMm));

  for (final s in history) {

    linePath.lineTo(toX(s.time.millisecondsSinceEpoch.toDouble()),
toY(s.distanceMm));

  }

  canvas.drawPath(

    linePath,

    Paint()
```

```
      ..color = _accent
      ..style = PaintingStyle.stroke
      ..strokeWidth = 2
      ..strokeCap = StrokeCap.round
      ..strokeJoin = StrokeJoin.round,
  );

  // Drone icon at current position
  final last = history.last;
  final droneX = toX(tEnd);
  final droneY = toY(last.distanceMm);

  // Glow ring
  canvas.drawCircle(Offset(droneX, droneY), 10,
      Paint()..color = _accent.withOpacity(0.2));
  // Outer ring
  canvas.drawCircle(Offset(droneX, droneY), 6,
      Paint()..color = _accent.withOpacity(0.6)..style =
PaintingStyle.stroke..strokeWidth = 1.5);
  // Centre dot
  canvas.drawCircle(Offset(droneX, droneY), 3, Paint()..color = _accent);

  // Mini drone arms (4 lines radiating at 45°)
  final armPaint = Paint()..color = _accent..strokeWidth = 1.5;
  for (final angle in [pi / 4, 3 * pi / 4, 5 * pi / 4, 7 * pi / 4]) {
    canvas.drawLine(
```

```
      Offset(droneX + cos(angle) * 6, droneY + sin(angle) * 6),
      Offset(droneX + cos(angle) * 10, droneY + sin(angle) * 10),
      armPaint,
    );
    canvas.drawCircle(
      Offset(droneX + cos(angle) * 11, droneY + sin(angle) * 11),
      2,
      Paint()..color = _accent.withOpacity(0.8),
    );
  }
}

@override
bool shouldRepaint(_FloorLinePainter old) => old.history != history;
}

// ═══════════════════════════════════════════════════════════════════
// TAB 3 — Battery
// ═══════════════════════════════════════════════════════════════════

class _BatteryTab extends StatefulWidget {
final BleDistanceService bleService;
const _BatteryTab({required this.bleService});

@override
State<_BatteryTab> createState() => _BatteryTabState();
```

```
}


class _BatteryTabState extends State<_BatteryTab> {
 List<BatterySnapshot> _history = [];


 @override
 void initState() {
   super.initState();
   widget.bleService.batteryStream?.listen((_) {
     if (mounted) setState(() => _history =
List.from(widget.bleService.batteryHistory));
   });
   _history = List.from(widget.bleService.batteryHistory);
 }


 @override
 Widget build(BuildContext context) {
   return _TimeSeriesChart(
     title: "BATTERY",
     series: [
       _Series(
         label: "Voltage (V)",
         color: _farColor,
         values: _history.map((s) => _Point(s.time, s.voltage)).toList(),
         yMin: 9.0,
         yMax: 13.0,
```

```
      ),
      _Series(
        label: "Current (A)",
        color: _midColor,
        values: _history.map((s) => _Point(s.time, s.current)).toList(),
        yMin: 0.0,
        yMax: 30.0,
      ),
    ],
  );
 }
}

// ═══════════════════════════════════════════════════════════════════════
// TAB 4 — Attitude
// ═══════════════════════════════════════════════════════════════════════

class _AttitudeTab extends StatefulWidget {
 final BleDistanceService bleService;
 const _AttitudeTab({required this.bleService});

 @override
 State<_AttitudeTab> createState() => _AttitudeTabState();
}

class _AttitudeTabState extends State<_AttitudeTab> {
```

```
 List<AttitudeSnapshot> _history = [];



 @override

 void initState() {

   super.initState();

   widget.bleService.attitudeStream?.listen((_) {

     if (mounted) setState(() => _history =
List.from(widget.bleService.attitudeHistory));

   });

   _history = List.from(widget.bleService.attitudeHistory);

 }



 @override

 Widget build(BuildContext context) {

   return _TimeSeriesChart(

     title: "ATTITUDE",

     series: [

       _Series(

         label: "Pitch (°)",

         color: _accent,

         values: _history.map((s) => _Point(s.time, s.pitch)).toList(),

         yMin: -90,

         yMax: 90,

       ),

       _Series(

         label: "Roll (°)",
```

```
        color: _nearColor,
        values: _history.map((s) => _Point(s.time, s.roll)).toList(),
        yMin: -90,
        yMax: 90,
      ),
      _Series(
        label: "Yaw (°)",
        color: _midColor,
        values: _history.map((s) => _Point(s.time, s.yaw)).toList(),
        yMin: -180,
        yMax: 180,
      ),
    ],
  );
 }
}


// ═══════════════════════════════════════════════════════════════════
// TAB 5 — RPM
// ═══════════════════════════════════════════════════════════════════


class _RpmTab extends StatefulWidget {
 final BleDistanceService bleService;
 const _RpmTab({required this.bleService});


 @override
```

```
 State<_RpmTab> createState() => _RpmTabState();
}


class _RpmTabState extends State<_RpmTab> {
 List<RpmSnapshot> _history = [];
 double? _sessionAvg;


 @override
 void initState() {
   super.initState();
   widget.bleService.attitudeStream?.listen((_) {
     if (mounted) {
       setState(() {
         _history    = List.from(widget.bleService.rpmHistory);
         _sessionAvg = widget.bleService.sessionAvgRpm;
       });
     }
   });
   _history    = List.from(widget.bleService.rpmHistory);
   _sessionAvg = widget.bleService.sessionAvgRpm;
 }


 @override
 Widget build(BuildContext context) {
   final currentRpm = _history.isNotEmpty ? _history.last.rpm : null;
```

```
return Column(
  children: [
    // ---- Session average banner ----
    Container(
      margin: const EdgeInsets.fromLTRB(16, 12, 16, 0),
      padding: const EdgeInsets.symmetric(vertical: 10, horizontal: 16),
      decoration: BoxDecoration(
        color: const Color(0xFF121828),
        border: Border.all(color: const Color(0xFF1E2A40)),
        borderRadius: BorderRadius.circular(8),
      ),
      child: Row(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        children: [
          // Session avg
          Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              const Text(
                "SESSION AVG",
                style: TextStyle(
                  color: _textSec, fontSize: 9,
                  letterSpacing: 2, fontFamily: 'monospace',
                ),
              ),
              const SizedBox(height: 2),
```

```
            Text(
              _sessionAvg != null
                  ? "${_sessionAvg!.toStringAsFixed(1)} RPM"
                  : "--",
              style: const TextStyle(
                color: _farColor, fontSize: 22,
                fontWeight: FontWeight.w900, fontFamily: 'monospace',
              ),
            ),
          ],
        ),
        // Divider
        Container(width: 1, height: 36, color: const Color(0xFF1E2A40)),
        // Current
        Column(
          crossAxisAlignment: CrossAxisAlignment.end,
          children: [
            const Text(
              "CURRENT",
              style: TextStyle(
                color: _textSec, fontSize: 9,
                letterSpacing: 2, fontFamily: 'monospace',
              ),
            ),
            const SizedBox(height: 2),
            Text(
```

```
                currentRpm != null
                    ? "${currentRpm.toStringAsFixed(1)} RPM"
                    : "--",
                style: TextStyle(
                  color: currentRpm != null && currentRpm.abs() > 5
                      ? _accent
                      : _textSec,
                  fontSize: 22,
                  fontWeight: FontWeight.w900,
                  fontFamily: 'monospace',
                ),
              ),
            ],
          ),
        ],
      ),
    ),

    // ---- Chart ----
    Expanded(
      child: _TimeSeriesChart(
        title: "RPM",
        series: [
          _Series(
            label: "RPM",
            color: _accent,
```

```
              values: _history.map((s) => _Point(s.time, s.rpm)).toList(),
              yMin: -300,
              yMax: 300,
              // Pass session avg so the painter can draw the reference line
              referenceLine: _sessionAvg,
            ),
          ],
        ),
      ),
    ],
  );
 }
}


// ============================================================================
// Shared time-series chart widget
// ============================================================================


class _Point {
 final DateTime time;
 final double value;
 const _Point(this.time, this.value);
}


class _Series {
 final String label;
```

```
 final Color color;

 final List<_Point> values;

 final double yMin;

 final double yMax;

 final double? referenceLine;   // optional horizontal reference (e.g. session
avg)

 const _Series({

   required this.label,

   required this.color,

   required this.values,

   required this.yMin,

   required this.yMax,

   this.referenceLine,

 });

}


class _TimeSeriesChart extends StatelessWidget {

 final String title;

 final List<_Series> series;

 const _TimeSeriesChart({required this.title, required this.series});


 @override

 Widget build(BuildContext context) {

   final hasData = series.any((s) => s.values.length >= 2);


   return Column(
```

```
    children: [
      const SizedBox(height: 12),
      // Legend
      Wrap(
        spacing: 16,
        children: series.map((s) => Row(mainAxisSize: MainAxisSize.min,
children: [
          Container(width: 24, height: 2, color: s.color),
          const SizedBox(width: 6),
          Text(s.label, style: TextStyle(color: s.color, fontSize: 11,
fontFamily: 'monospace')),
        ])).toList(),
      ),
      const SizedBox(height: 8),
      Expanded(
        child: Padding(
          padding: const EdgeInsets.fromLTRB(8, 0, 16, 8),
          child: !hasData
              ? const Center(
                  child: Text("Waiting for data…",
                      style: TextStyle(color: _textSec, fontFamily:
'monospace')))
              : CustomPaint(
                  painter: _MultiLinePainter(series: series),
                  child: Container(),
                ),
        ),
```

```
      ),
    ],
  );
 }
}


class _MultiLinePainter extends CustomPainter {
 final List<_Series> series;
 const _MultiLinePainter({required this.series});


 @override
 void paint(Canvas canvas, Size size) {
   final allPoints = series.expand((s) => s.values).toList();
   if (allPoints.length < 2) return;


   const padT = 12.0, padB = 28.0, padL = 50.0, padR = 8.0;
   final chartW = size.width - padL - padR;
   final chartH = size.height - padT - padB;


   final tStart = allPoints.map((p) =>
p.time.millisecondsSinceEpoch).reduce(min).toDouble();
   final tEnd   = allPoints.map((p) =>
p.time.millisecondsSinceEpoch).reduce(max).toDouble();
   final tRange = max(tEnd - tStart, 1.0);


   // Use first series' Y range for grid; each series normalises individually
   final yMin = series.first.yMin;
```

```
  final yMax = series.first.yMax;
  final yRange = max(yMax - yMin, 0.001);

  double toX(double t) => padL + ((t - tStart) / tRange) * chartW;
  double toY(double v, double mn, double mx) =>
      padT + chartH - ((v - mn) / (mx - mn)) * chartH;

  // Grid
  final gridPaint = Paint()..color = _border..strokeWidth = 0.8;
  final tp = TextPainter(textDirection: TextDirection.ltr);
  final labelStyle = const TextStyle(color: _textSec, fontSize: 9, fontFamily:
'monospace');

  for (int i = 0; i <= 4; i++) {
    final v = yMin + (yMax - yMin) * (i / 4);
    final y = toY(v, yMin, yMax);
    canvas.drawLine(Offset(padL, y), Offset(size.width - padR, y), gridPaint);
    tp.text = TextSpan(text: v.toStringAsFixed(0), style: labelStyle);
    tp.layout();
    tp.paint(canvas, Offset(0, y - tp.height / 2));
  }

  // Lines per series
  for (final s in series) {
    // Reference line (e.g. session avg) — dashed, dimmer
    if (s.referenceLine != null) {
```

```
      final refY = toY(s.referenceLine!, s.yMin, s.yMax);
      final dashPaint = Paint()
        ..color = _farColor.withOpacity(0.55)
        ..strokeWidth = 1.0
        ..style = PaintingStyle.stroke;
      // Manual dash: 6px on, 4px off
      double x = padL;
      while (x < size.width - padR) {
        canvas.drawLine(
          Offset(x, refY),
          Offset((x + 6).clamp(padL, size.width - padR), refY),
          dashPaint,
        );
        x += 10;
      }
      // Label on right edge
      tp.text = TextSpan(
        text: "avg",
        style: const TextStyle(
            color: _farColor, fontSize: 8,
            fontFamily: 'monospace', fontWeight: FontWeight.bold),
      );
      tp.layout();
      tp.paint(canvas, Offset(size.width - padR - tp.width - 2, refY -
tp.height - 1));
    }
```

```
    if (s.values.length < 2) continue;
    final path = Path();
    path.moveTo(
      toX(s.values.first.time.millisecondsSinceEpoch.toDouble()),
      toY(s.values.first.value, s.yMin, s.yMax),
    );
    for (final p in s.values.skip(1)) {
      path.lineTo(toX(p.time.millisecondsSinceEpoch.toDouble()),
                  toY(p.value, s.yMin, s.yMax));
    }
    canvas.drawPath(
      path,
      Paint()
        ..color = s.color
        ..style = PaintingStyle.stroke
        ..strokeWidth = 1.8
        ..strokeCap = StrokeCap.round
        ..strokeJoin = StrokeJoin.round,
    );


    // End dot
    final last = s.values.last;
    canvas.drawCircle(
      Offset(toX(tEnd), toY(last.value, s.yMin, s.yMax)),
      3,
```

```
        Paint()..color = s.color,
      );
    }
  }

  @override
  bool shouldRepaint(_MultiLinePainter old) => old.series != series;
}
```

ESP32 Code

```
#include <Arduino.h>
#include <NimBLEDevice.h>
#include <Wire.h>
#include <vl53l8cx.h>
#include <HardwareSerial.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <freertos/semphr.h>


// ============================================================
//  SECTION 1 — Shared telemetry state
// ============================================================
```

```
struct BatData { float voltage=0; float current=0; uint32_t mah=0; uint8_t
remaining=0; };

struct AttData { float pitch=0;   float roll=0;    float yaw=0; };



BatData batData;

AttData attData;

char    flightMode[32] = "UNKNOWN";



// ================================================================

//  SECTION 2 — UART / CRSF

// ================================================================



HardwareSerial FC(2);



#define CRSF_ADDRESS_FLIGHT_CONTROLLER    0xC8

#define CRSF_FRAMETYPE_RC_CHANNELS_PACKED 0x16

#define CRSF_FRAMETYPE_BATTERY            0x08

#define CRSF_FRAMETYPE_ATTITUDE           0x1E

#define CRSF_FRAMETYPE_FLIGHT_MODE        0x21

#define CRSF_INTERVAL_MS                  5



unsigned long startTime = 0;

bool          shutdownActive  = false;

unsigned long shutdownStart   = 0;
```

```
#define SHUTDOWN_RAMP_MS 2000



// ================================================================

//  SECTION 3 — FreeRTOS primitives

// ================================================================



// Mutexes — one per shared resource group

SemaphoreHandle_t crsfMutex      = nullptr;  // protects sharedThrottle/Aux/Yaw/Pitch

SemaphoreHandle_t sensorMutex    = nullptr;  // protects
lastSensorDistance/lastWallDistance

SemaphoreHandle_t telemetryMutex = nullptr;  // protects batData/attData/flightMode

SemaphoreHandle_t stateMutex     = nullptr;  // protects
droneOn/hover/landing/spin/elapsed



// Shared CRSF output — written by altHold task, read by CRSF task

volatile int sharedThrottle = 1012;

volatile int sharedAux      = 1000;

volatile int sharedYaw      = 1500;

volatile int sharedPitch    = 1500;



// Shared sensor data — written by sensor task, read by altHold task

volatile uint16_t lastSensorDistance = 0;

volatile uint16_t lastWallDistance   = 0;



// Shared flight state — written by BLE callbacks, read by altHold/CRSF tasks
```

```
volatile bool droneOn = false;

volatile bool hover   = false;

volatile bool spin    = false;

volatile bool landing = false;

volatile bool avoiding = false;

unsigned long landingStart  = 0;

unsigned long hoverStartMs  = 0;

unsigned long disconnectTime = 0;

#define BLE_GRACE_MS 1500



// ================================================================

//  SECTION 4 — Distance sensor globals

// ================================================================



VL53L8CX*            sensor           = nullptr;

VL53L8CX_ResultsData measurementData;

unsigned long        lastDistanceRead  = 0;

const int            distanceInterval  = 40;

bool                 sensorReady       = false;



#define SECTOR_COUNT 36   // 10° sectors



volatile uint16_t lidarMap[SECTOR_COUNT];

volatile uint32_t lastMapReset = 0;
```

```
// Spike filter
#define SENSOR_MAX_DELTA_MM  150
uint16_t lastValidDistance  = 0;
uint8_t  sensorBadReadCount = 0;

// ================================================================
//  SECTION 5 — BLE UUIDs & characteristics
// ================================================================

#define SERVICE_UUID         "0000FEED-0000-1000-8000-00805F9B34FB"
#define CHAR_UUID_DISTANCE   "00005748-0000-1000-8000-00805F9B34FB"
#define CHAR_UUID_BATTERY    "00005749-0000-1000-8000-00805F9B34FB"
#define CHAR_UUID_ATTITUDE   "0000574A-0000-1000-8000-00805F9B34FB"
#define CHAR_UUID_FLIGHTMODE "0000574B-0000-1000-8000-00805F9B34FB"
#define CHAR_UUID_DRONE_ON   "0000574C-0000-1000-8000-00805F9B34FB"
#define CHAR_UUID_HOVER      "0000574D-0000-1000-8000-00805F9B34FB"
#define CHAR_UUID_SPIN       "0000574E-0000-1000-8000-00805F9B34FB"
#define CHAR_UUID_LIDAR_MAP  "0000574F-0000-1000-8000-00805F9B34FB"

NimBLECharacteristic* pDistanceChar   = nullptr;
NimBLECharacteristic* pBatteryChar    = nullptr;
NimBLECharacteristic* pAttitudeChar   = nullptr;
NimBLECharacteristic* pFlightModeChar = nullptr;
```

```
NimBLECharacteristic* pDroneOnChar    = nullptr;
NimBLECharacteristic* pHoverChar      = nullptr;
NimBLECharacteristic* pSpinChar       = nullptr;
NimBLECharacteristic* pLidarMapChar   = nullptr;


#define SPIN_YAW         1850
#define AVOID_PITCH_BACK 1470
#define NEUTRAL_PITCH    1500
#define WALL_THRESHOLD   400
#define WALL_CLEAR_MM    500
volatile bool toggleAvoidance = false;


// ================================================================
//  SECTION 6 — Altitude hold & landing
// ================================================================


#define ALT_TARGET_MM              600
#define ALT_DEADBAND_MM            30
#define THROTTLE_MIN               1560
#define THROTTLE_MAX               1720
#define THROTTLE_INIT              1600
#define THROTTLE_STEP_UP           0.3f
#define THROTTLE_STEP_DOWN         0.3f
#define THROTTLE_STEP_FAST_UP      0.6f
```

```
#define THROTTLE_STEP_FAST_DOWN     0.6f

#define THROTTLE_STEP_BRAKE         1

#define ALT_FAST_THRESHOLD          150

#define LAND_DISARM_MM              100

#define LAND_TIMEOUT_MS             15000

#define LAND_DESCENT_RATE_MM        15

#define HOVER_MAX_VERTICAL_SPEED_MM 5



float altHoldThrottle    = THROTTLE_INIT;

bool  altHoldActive      = false;

float stableHoldThrottle = THROTTLE_INIT;

float hoverLearningRate  = 0.02f;

bool  wasInDeadband      = false;

bool  landCompensationApplied = false;



bool     disarmRampActive  = false;

float    disarmThrottle    = 0.0f;

unsigned long disarmStart  = 0;



// Ramp

float         rampThrottle = 1050.0f;

bool          rampComplete = false;

unsigned long rampStart    = 0;
```

```
void resetRamp() {
   rampThrottle = 1050.0f;
   rampComplete = false;
   rampStart    = 0;
}


bool updateRamp(unsigned long now) {
   if (rampComplete) return false;
   if (rampStart == 0) rampStart = now;
   float progress = constrain((float)(now - rampStart) / 1000.0f, 0.0f, 1.0f);
   rampThrottle   = 1050.0f + (560.0f * progress);
   if (progress >= 1.0f) {
       rampThrottle = 1580.0f;
       rampComplete = true;
       return false;
   }
   return true;
}


int batteryCompensation() {
   // Called only from altHold task — no mutex needed for batData read
   // since telemetryMutex is taken before this is called
   if (batData.voltage == 0) return 0;
```

```
    if (batData.voltage <= 10.3f && droneOn && hover) {
        landing    = true;
        landingStart = millis();
        hover      = false;
    }

    float voltage = constrain(batData.voltage, 10.2f, 12.4f);
    int offset    = (int)((12.1f - voltage) * 58.0f);
    return constrain(offset, 0, 150);
}

int computeHoverThrottle() {
    if (!hover && !landing) return 1012;

    uint16_t distance;
    if (xSemaphoreTake(sensorMutex, pdMS_TO_TICKS(2))) {
        distance = lastSensorDistance;
        xSemaphoreGive(sensorMutex);
    } else {
        return (int)altHoldThrottle;  // mutex timeout — hold current
    }

    if (distance == 0) return (int)altHoldThrottle;
```

```
  int batComp = batteryCompensation();



  if (!altHoldActive) {

      altHoldThrottle    = THROTTLE_INIT;

      stableHoldThrottle = THROTTLE_INIT;

      altHoldActive      = true;

  }



  int error = (int)distance - ALT_TARGET_MM;



  // --- LANDING ---

  if (landing) {

      static uint16_t      lastLandDist   = 0;

      static unsigned long lastLandUpdate = 0;



      if (!landCompensationApplied) {

          altHoldThrottle         = constrain(altHoldThrottle + batComp - 20,
1012.0f, (float)THROTTLE_MAX);

          landCompensationApplied = true;

      }



      if (distance <= LAND_DISARM_MM) {
```

```
            if (!disarmRampActive) {
                disarmRampActive = true;
                disarmThrottle   = altHoldThrottle;
                disarmStart      = millis();
            }

            float elapsed = (float)(millis() - disarmStart) / 1000.0f;  // 0.0 → 1.0
over 1s
            elapsed = constrain(elapsed, 0.0f, 1.0f);

            float rampedThrottle = disarmThrottle - (disarmThrottle -
(float)THROTTLE_MIN) * elapsed;

            if (elapsed >= 1.0f) {
                // Ramp complete — disarm
                disarmRampActive = false;
                landing          = false;
                droneOn          = false;
                spin             = false;
                altHoldActive    = false;
                if (xSemaphoreTake(crsfMutex, pdMS_TO_TICKS(5))) {
                    sharedThrottle = 1012;
                    sharedAux      = 1000;
                    xSemaphoreGive(crsfMutex);
                }
```

```
            return 1012;
        }

        return (int)constrain(rampedThrottle, 1012.0f, (float)THROTTLE_MAX);
    }

    if (millis() - lastLandUpdate >= 50) {
        lastLandUpdate = millis();
        int descentRate = (int)lastLandDist - (int)distance;
        lastLandDist = distance;
        if (descentRate > LAND_DESCENT_RATE_MM)
            altHoldThrottle += THROTTLE_STEP_UP;
        else if (descentRate < LAND_DESCENT_RATE_MM)
            altHoldThrottle -= THROTTLE_STEP_DOWN;
    }

    altHoldThrottle = constrain(altHoldThrottle, 1012.0f, (float)THROTTLE_MAX);
    return (int)altHoldThrottle;
}

// --- HOVER ---
static uint16_t      lastHoverDist   = 0;
static unsigned long lastHoverUpdate = 0;
```

```
  int verticalSpeed = (int)distance - (int)lastHoverDist;
  lastHoverDist = distance;
  if (abs(verticalSpeed) < 3) verticalSpeed = 0;

  if (millis() - lastHoverUpdate >= 70) {
      lastHoverUpdate = millis();

      // Ceiling failsafe — independent of PID
      if (distance > 2000) {
          landing      = true;
          landingStart = millis();
          landCompensationApplied = false;
      } else if (abs(error) <= ALT_DEADBAND_MM) {
          wasInDeadband   = true;
          toggleAvoidance = toggleAvoidance || spin;

          if (abs(verticalSpeed) <= 2)
              stableHoldThrottle += hoverLearningRate * (altHoldThrottle -
stableHoldThrottle);
          if (verticalSpeed > 2)
              altHoldThrottle -= THROTTLE_STEP_DOWN;
          else if (verticalSpeed < -2)
              altHoldThrottle += THROTTLE_STEP_UP;
```

```
      } else if (error > 0) {

          if (wasInDeadband) { wasInDeadband = false; }

          if (verticalSpeed > -HOVER_MAX_VERTICAL_SPEED_MM) {

              float step = (error > ALT_FAST_THRESHOLD) ? THROTTLE_STEP_FAST_DOWN :
THROTTLE_STEP_DOWN;

              altHoldThrottle -= step;

          } else {

              altHoldThrottle += THROTTLE_STEP_BRAKE;

          }

      } else {

          if (wasInDeadband) { wasInDeadband = false; }

          if (verticalSpeed < HOVER_MAX_VERTICAL_SPEED_MM) {

              float step = (-error > ALT_FAST_THRESHOLD) ? THROTTLE_STEP_FAST_UP :
THROTTLE_STEP_UP;

              altHoldThrottle += step;

          } else {

              altHoldThrottle -= THROTTLE_STEP_BRAKE;

          }

      }

  }



  altHoldThrottle = constrain(altHoldThrottle, THROTTLE_MIN, THROTTLE_MAX);

  return (int)constrain(altHoldThrottle + batComp, THROTTLE_MIN, THROTTLE_MAX);

}
```

```
int getSector(float yaw)

{

   int sector = (int)(yaw / (360.0f / SECTOR_COUNT));

   if(sector < 0) sector += SECTOR_COUNT;

   if(sector >= SECTOR_COUNT) sector -= SECTOR_COUNT;

   return sector;

}



int findBestSector()

{

   int best = -1;

   uint16_t bestDist = 0;



   for(int i=0;i<SECTOR_COUNT;i++)

   {

       if(lidarMap[i] > bestDist)

       {

           bestDist = lidarMap[i];

           best = i;

       }

   }



   return best;

}
```

```
int computeAvoidancePitch()

{

   int bestSector = findBestSector();

   if(bestSector < 0) return NEUTRAL_PITCH;



   float targetYaw = bestSector * (360.0f / SECTOR_COUNT);



   float yaw;

   if (xSemaphoreTake(telemetryMutex, pdMS_TO_TICKS(2))) {

       yaw = attData.yaw;

       xSemaphoreGive(telemetryMutex);

   }



   float diff = fabs(yaw - targetYaw);



   if(diff > 180) diff = 360 - diff;



   if(diff < 15)

       return 1530;   // move forward



   return NEUTRAL_PITCH;

}
```

```
// ================================================================
//  SECTION 7 — CRC & CRSF Send
// ================================================================

uint8_t crc8(const uint8_t *ptr, uint8_t len) {
   uint8_t crc = 0;
   while (len--) {
       crc ^= *ptr++;
       for (int i = 0; i < 8; i++)
           crc = (crc & 0x80) ? ((crc << 1) ^ 0xD5) : (crc << 1);
   }
   return crc;
}

void sendCRSF(int throttle, int aux, int yaw = 1500, int pitch = 1500) {
   uint16_t channels[16] = {
       992,
       (uint16_t)map(pitch,    1000, 2000, 172, 1811),
       (uint16_t)map(throttle, 1000, 2000, 172, 1811),
       (uint16_t)map(yaw,      1000, 2000, 172, 1811),
       (uint16_t)map(aux,      1000, 2000, 172, 1811),
       1811, 172, 172, 172, 172, 172, 172, 172, 172, 172, 172
   };
   uint8_t payload[22];
```

```
    memset(payload, 0, sizeof(payload));

    int bitIndex = 0;

    for (int i = 0; i < 16; i++) {

        for (int b = 0; b < 11; b++) {

            if (channels[i] & (1 << b))

                payload[(bitIndex + b) / 8] |= 1 << ((bitIndex + b) % 8);

        }

        bitIndex += 11;

    }

    uint8_t frame[26];

    frame[0] = CRSF_ADDRESS_FLIGHT_CONTROLLER;

    frame[1] = 24;

    frame[2] = CRSF_FRAMETYPE_RC_CHANNELS_PACKED;

    memcpy(&frame[3], payload, 22);

    frame[25] = crc8(&frame[2], 23);

    FC.write(frame, 26);

}



// ================================================================

//  SECTION 8 — CRSF Parse

// ================================================================



void parseCRSFTelemetry() {
```

```
   static enum { WAIT_SYNC, WAIT_LEN, WAIT_TYPE, WAIT_PAYLOAD, WAIT_CRC } state =
WAIT_SYNC;

   static uint8_t frameLen = 0, frameType = 0, payload[64], payloadIndex = 0;



   while (FC.available()) {

       uint8_t b = FC.read();

       switch (state) {

           case WAIT_SYNC:

               if (b == 0xC8) state = WAIT_LEN;

               break;

           case WAIT_LEN:

               frameLen = b - 2; payloadIndex = 0; state = WAIT_TYPE;

               break;

           case WAIT_TYPE:

               frameType = b; state = (frameLen > 0) ? WAIT_PAYLOAD : WAIT_CRC;

               break;

           case WAIT_PAYLOAD:

               if (payloadIndex < sizeof(payload)) payload[payloadIndex] = b;

               if (++payloadIndex >= frameLen) state = WAIT_CRC;

               break;

           case WAIT_CRC: {

               uint8_t crcBuf[65];

               crcBuf[0] = frameType;

               memcpy(&crcBuf[1], payload, frameLen);
```

```
                if (b == crc8(crcBuf, frameLen + 1)) {

                    // Take telemetry mutex briefly to update shared data

                    if (xSemaphoreTake(telemetryMutex, pdMS_TO_TICKS(2))) {

                        if (frameType == CRSF_FRAMETYPE_BATTERY && frameLen >= 8) {

                            batData.voltage   = (uint16_t)((payload[0] << 8) |
payload[1]) / 10.0f;

                            batData.current   = (uint16_t)((payload[2] << 8) |
payload[3]) / 10.0f;

                            batData.mah       = ((uint32_t)payload[4] << 16) |
((uint32_t)payload[5] << 8) | payload[6];

                            batData.remaining = payload[7];

                        } else if (frameType == CRSF_FRAMETYPE_ATTITUDE && frameLen >=
6) {

                            attData.pitch = (int16_t)((payload[0] << 8) | payload[1]) /
174.533f;

                            attData.roll  = (int16_t)((payload[2] << 8) | payload[3]) /
174.533f;

                            attData.yaw   = (int16_t)((payload[4] << 8) | payload[5]) /
174.533f;

                        } else if (frameType == CRSF_FRAMETYPE_FLIGHT_MODE && frameLen
>= 1) {

                            payload[frameLen] = '\0';

                            strncpy(flightMode, (char*)payload, sizeof(flightMode) -
1);

                        }

                        xSemaphoreGive(telemetryMutex);

                    }

                } else {
```

```
                    Serial.printf("[CRSF] CRC mismatch type=0x%02X\n", frameType);
                }
                state = WAIT_SYNC;
                break;
            }
        }
    }
}

// ================================================================
//  SECTION 9 — BLE Notify helpers
// ================================================================

void notifyDistance(uint16_t distance) {
    if (!pDistanceChar) return;
    pDistanceChar->setValue((uint8_t*)&distance, sizeof(distance));
    pDistanceChar->notify();
}
void notifyBattery() {
    if (!pBatteryChar) return;
    uint8_t buf[13];
    if (xSemaphoreTake(telemetryMutex, pdMS_TO_TICKS(5))) {
        memcpy(&buf[0], &batData.voltage, 4);
        memcpy(&buf[4], &batData.current, 4);
```

```
        memcpy(&buf[8], &batData.mah,     4);
        buf[12] = batData.remaining;
        xSemaphoreGive(telemetryMutex);
    }
    pBatteryChar->setValue(buf, sizeof(buf));
    pBatteryChar->notify();
}
void notifyAttitude() {
    if (!pAttitudeChar) return;
    uint8_t buf[12];
    if (xSemaphoreTake(telemetryMutex, pdMS_TO_TICKS(5))) {
        memcpy(&buf[0], &attData.pitch, 4);
        memcpy(&buf[4], &attData.roll,  4);
        memcpy(&buf[8], &attData.yaw,   4);
        xSemaphoreGive(telemetryMutex);
    }
    pAttitudeChar->setValue(buf, sizeof(buf));
    pAttitudeChar->notify();
}
void notifyFlightMode() {
    if (!pFlightModeChar) return;
    char mode[32];
    if (xSemaphoreTake(telemetryMutex, pdMS_TO_TICKS(5))) {
        strncpy(mode, flightMode, sizeof(mode));
```

```
        xSemaphoreGive(telemetryMutex);
    }
    pFlightModeChar->setValue((uint8_t*)mode, strlen(mode));
    pFlightModeChar->notify();
}

void notifyLidarMap() {
    if (!pLidarMapChar) return;
    // Pack 36 x uint16 into 72 bytes, little-endian
    uint8_t buf[72];
    for (int i = 0; i < SECTOR_COUNT; i++) {
        uint16_t d = lidarMap[i];
        buf[i * 2]     = d & 0xFF;
        buf[i * 2 + 1] = (d >> 8) & 0xFF;
    }
    pLidarMapChar->setValue(buf, sizeof(buf));
    pLidarMapChar->notify();
}

// ================================================================
//  SECTION 10 — BLE Callbacks
// ================================================================

class DroneOnCallbacks : public NimBLECharacteristicCallbacks {
```

```
void onWrite(NimBLECharacteristic* pChar, NimBLEConnInfo& connInfo) override {

    if (pChar->getLength() >= 1) {

        bool newValue = (pChar->getValue()[0] != 0);

        Serial.printf("[BLE] drone_on = %s\n", newValue ? "true" : "false");

        if (newValue && !droneOn) {

            startTime              = millis();

            landing                = false;

            hover                  = false;

            spin                   = false;

            avoiding               = false;

            altHoldThrottle        = THROTTLE_INIT;

            altHoldActive          = false;

            stableHoldThrottle     = THROTTLE_INIT;

            wasInDeadband          = false;

            landCompensationApplied = false;

            toggleAvoidance        = false;

            disarmRampActive = false;

            disarmThrottle   = 0.0f;

            disarmStart      = 0;

            resetRamp();

            if (xSemaphoreTake(crsfMutex, pdMS_TO_TICKS(5))) {

                sharedThrottle = 1012;

                sharedAux      = 1000;

                sharedYaw      = 1500;
```

```
                    sharedPitch    = 1500;
                    xSemaphoreGive(crsfMutex);
                }
            }
            droneOn = newValue;
        }
    }
};

class HoverCallbacks : public NimBLECharacteristicCallbacks {
    void onWrite(NimBLECharacteristic* pChar, NimBLEConnInfo& connInfo) override {
        if (pChar->getLength() >= 1) {
            bool newValue = (pChar->getValue()[0] != 0);
            Serial.printf("[BLE] hover = %s\n", newValue ? "true" : "false");
            if (newValue) {
                hoverStartMs           = millis();
                landing                = false;
                altHoldThrottle        = THROTTLE_INIT;
                stableHoldThrottle     = THROTTLE_INIT;
                altHoldActive          = false;
                wasInDeadband          = false;
                landCompensationApplied = false;
                toggleAvoidance        = false;
                avoiding               = false;
```

```
            lastValidDistance      = 0;
            sensorBadReadCount     = 0;
            disarmRampActive = false;
            disarmThrottle   = 0.0f;
            disarmStart      = 0;
        } else {
            if (droneOn) {
                altHoldActive          = true;
                landing                = true;
                landingStart           = millis();
                landCompensationApplied = false;
            }
        }
        hover = newValue;
    }
  }
};


class SpinCallbacks : public NimBLECharacteristicCallbacks {
  void onWrite(NimBLECharacteristic* pChar, NimBLEConnInfo& connInfo) override {
    if (pChar->getLength() >= 1) {
        spin = (pChar->getValue()[0] != 0);
    }
  }
```

```
};



class ServerCallbacks : public NimBLEServerCallbacks {

   void onConnect(NimBLEServer* pServer, NimBLEConnInfo& connInfo) override {

       Serial.println("[BLE] Client connected");

       if (disconnectTime > 0 && millis() - disconnectTime < BLE_GRACE_MS) {

           Serial.println("[BLE] Reconnected within grace — cancelling landing");

           landing         = false;

           avoiding        = false;

           toggleAvoidance = false;

           disconnectTime  = 0;

       }

   }

   void onDisconnect(NimBLEServer* pServer, NimBLEConnInfo& connInfo, int reason)
override {

       Serial.println("[BLE] Client disconnected — restarting advertising");

       disconnectTime  = millis();

       hover           = false;

       spin            = false;

       avoiding        = false;

       toggleAvoidance = false;

       if (droneOn) {

           altHoldActive          = true;

           landing                = true;
```

```
            landingStart           = millis();

            landCompensationApplied = false;

        }

        shutdownActive = true;

        shutdownStart  = millis();

        NimBLEDevice::startAdvertising();

    }

};



// ================================================================

//  SECTION 11 — BLE Setup

// ================================================================



void setupBLE() {

    NimBLEDevice::init("ESP32_DRONE");

    NimBLEDevice::setPower(ESP_PWR_LVL_P9);



    NimBLEServer*  pServer  = NimBLEDevice::createServer();

    pServer->setCallbacks(new ServerCallbacks());

    NimBLEService* pService = pServer->createService(SERVICE_UUID);



    pDistanceChar = pService->createCharacteristic(CHAR_UUID_DISTANCE,
NIMBLE_PROPERTY::READ | NIMBLE_PROPERTY::NOTIFY);

    pDistanceChar->createDescriptor("2901", NIMBLE_PROPERTY::READ)->setValue("Distance
mm");
```

```
   pDistanceChar->createDescriptor("2902", NIMBLE_PROPERTY::READ |
NIMBLE_PROPERTY::WRITE);



   pBatteryChar = pService->createCharacteristic(CHAR_UUID_BATTERY,
NIMBLE_PROPERTY::READ | NIMBLE_PROPERTY::NOTIFY);

   pBatteryChar->createDescriptor("2901", NIMBLE_PROPERTY::READ)->setValue("Battery");

   pBatteryChar->createDescriptor("2902", NIMBLE_PROPERTY::READ |
NIMBLE_PROPERTY::WRITE);



   pAttitudeChar = pService->createCharacteristic(CHAR_UUID_ATTITUDE,
NIMBLE_PROPERTY::READ | NIMBLE_PROPERTY::NOTIFY);

   pAttitudeChar->createDescriptor("2901", NIMBLE_PROPERTY::READ)->setValue("Attitude
deg");

   pAttitudeChar->createDescriptor("2902", NIMBLE_PROPERTY::READ |
NIMBLE_PROPERTY::WRITE);



   pFlightModeChar = pService->createCharacteristic(CHAR_UUID_FLIGHTMODE,
NIMBLE_PROPERTY::READ | NIMBLE_PROPERTY::NOTIFY);

   pFlightModeChar->createDescriptor("2901", NIMBLE_PROPERTY::READ)->setValue("Flight
Mode");

   pFlightModeChar->createDescriptor("2902", NIMBLE_PROPERTY::READ |
NIMBLE_PROPERTY::WRITE);



   pDroneOnChar = pService->createCharacteristic(CHAR_UUID_DRONE_ON,
NIMBLE_PROPERTY::READ | NIMBLE_PROPERTY::WRITE);

   pDroneOnChar->createDescriptor("2901", NIMBLE_PROPERTY::READ)->setValue("Drone
On");

   uint8_t off = 0;

   pDroneOnChar->setValue(&off, 1);
```

```
   pDroneOnChar->setCallbacks(new DroneOnCallbacks());



   pHoverChar = pService->createCharacteristic(CHAR_UUID_HOVER, NIMBLE_PROPERTY::READ
| NIMBLE_PROPERTY::WRITE);

   pHoverChar->createDescriptor("2901", NIMBLE_PROPERTY::READ)->setValue("Hover");

   pHoverChar->setValue(&off, 1);

   pHoverChar->setCallbacks(new HoverCallbacks());



   pSpinChar = pService->createCharacteristic(CHAR_UUID_SPIN, NIMBLE_PROPERTY::READ |
NIMBLE_PROPERTY::WRITE);

   pSpinChar->createDescriptor("2901", NIMBLE_PROPERTY::READ)->setValue("Spin");

   pSpinChar->setValue(&off, 1);

   pSpinChar->setCallbacks(new SpinCallbacks());



   pLidarMapChar = pService->createCharacteristic(

       CHAR_UUID_LIDAR_MAP,

       NIMBLE_PROPERTY::READ | NIMBLE_PROPERTY::NOTIFY

   );

   pLidarMapChar->createDescriptor("2901", NIMBLE_PROPERTY::READ)->setValue("LiDAR
Map");

   pLidarMapChar->createDescriptor("2902", NIMBLE_PROPERTY::READ |
NIMBLE_PROPERTY::WRITE);



   // Zero-initialise

   {
```

```
        uint8_t zeroMap[72] = {0};
        pLidarMapChar->setValue(zeroMap, sizeof(zeroMap));
    }

    uint8_t zeroBat[13] = {0};  pBatteryChar->setValue(zeroBat, sizeof(zeroBat));
    uint8_t zeroAtt[12] = {0};  pAttitudeChar->setValue(zeroAtt, sizeof(zeroAtt));
    pFlightModeChar->setValue((uint8_t*)"UNKNOWN", 7);

    pService->start();
    NimBLEAdvertising* pAdv = NimBLEDevice::getAdvertising();
    pAdv->addServiceUUID(SERVICE_UUID);
    pAdv->setName("ESP32_DRONE");
    pAdv->start();

    Serial.println("[BLE] Advertising started");
}

// ================================================================
//  SECTION 12 — Distance sensor setup
// ================================================================

bool isVL53L8CXPresent() {
    Wire.beginTransmission(0x29);
    return Wire.endTransmission() == 0;
```

```
}


void setupDistanceSensor() {


   Wire.begin(21, 22);

   Wire.setClock(400000);

   Wire.setTimeOut(50);


   Serial.println("[VL53L8CX] Waiting for sensor...");


   while (!isVL53L8CXPresent()) {

       delay(20);

   }


   Serial.println("[VL53L8CX] Sensor detected");


   // Allow sensor firmware boot

   delay(400);


   sensor = new VL53L8CX(&Wire, -1);

   sensor->begin();


   while (sensor->init() != VL53L8CX_STATUS_OK) {

       Serial.println("[VL53L8CX] Init retry...");
```

```
      delay(200);
  }

  sensor->set_resolution(VL53L8CX_RESOLUTION_4X4);
  sensor->set_ranging_frequency_hz(20);
  sensor->start_ranging();

  sensorReady = true;

  Serial.println("[VL53L8CX] Initialized");
}

// ================================================================
//  SECTION 13 — Beeper
// ================================================================

#define SPEAKER_PIN  26 // 25
#define LEDC_CHANNEL 0
#define LEDC_RES     8

#define B3  247
#define E4  330
#define F3  175
#define G4  392
```

```
#define FS4 370

#define REST 0



const uint16_t melody[][2] = {

   {B3,320},{REST,80},{F3,320},{REST,80},{F3,180},{REST,80},

   {E4,180},{REST,80},{E4,320},{REST,80},{G4,80},{REST,80},

   {FS4,80},{REST,80},{G4,80},{REST,80},{FS4,80},{REST,80},

   {G4,180},{REST,80},{F3,180},{REST,80},{F3,180},{REST,80},

   {E4,180},{REST,80},{E4,320},{REST,80},

};

const int melodyLen = sizeof(melody) / sizeof(melody[0]);



void tone_(uint32_t freq, uint32_t durationMs, uint8_t volume = 128) {

   if (freq == REST) { ledcWrite(LEDC_CHANNEL, 0); delay(durationMs); return; }

   ledcSetup(LEDC_CHANNEL, freq, LEDC_RES);

   ledcAttachPin(SPEAKER_PIN, LEDC_CHANNEL);

   ledcWrite(LEDC_CHANNEL, volume);

   delay(durationMs);

   ledcWrite(LEDC_CHANNEL, 0);

   delay(20);

}



// ================================================================

//  SECTION 14 — FreeRTOS Tasks
```

```
// ================================================================


// ---- Task 1: CRSF Send — Core 1, highest priority ----
// Runs at exactly CRSF_INTERVAL_MS, never blocked by other work
void crsfTask(void* pvParameters) {
   TickType_t     xLastWakeTime = xTaskGetTickCount();
   const TickType_t xFrequency  = pdMS_TO_TICKS(CRSF_INTERVAL_MS);

   while (true) {
       int t, a, y, p;

       // Non-blocking read of shared CRSF values
       if (xSemaphoreTake(crsfMutex, 0)) {
           t = sharedThrottle;
           a = sharedAux;
           y = sharedYaw;
           p = sharedPitch;
           xSemaphoreGive(crsfMutex);
       } else {
           // Mutex busy — use last known safe values
           t = 1012; a = 1000; y = 1500; p = 1500;
       }

       sendCRSF(t, a, y, p);
```

```
        parseCRSFTelemetry();



        vTaskDelayUntil(&xLastWakeTime, xFrequency);

    }

}



// ---- Task 2: Altitude Hold — Core 1, high priority ----

// Computes throttle and writes to shared CRSF values

void altHoldTask(void* pvParameters) {

    while (true) {

        unsigned long now     = millis();

        unsigned long elapsed = (startTime == 0) ? 0 : (now - startTime);



        // BLE grace period check

        if (disconnectTime > 0 && now - disconnectTime > BLE_GRACE_MS) {

            disconnectTime = 0;

            droneOn        = false;

        }



        int throttle = 1012;

        int aux      = 1000;



        if (droneOn) {

            if (elapsed < 3000) {
```

```
            throttle = 1012; aux = 1000;
            resetRamp();

        } else if (elapsed < 5000) {
            throttle = 1012; aux = 2000;
            resetRamp();

        } else if (elapsed < 7000) {
            throttle = 1050; aux = 2000;
            resetRamp();

        } else if (landing) {
            throttle = computeHoverThrottle(); aux = 2000;

        } else if (!rampComplete) {
            updateRamp(now);
            throttle = (int)rampThrottle; aux = 2000;

        } else if (hover) {
            throttle = computeHoverThrottle(); aux = 2000;

        } else {
            throttle = 1580; aux = 2000;
        }
```

```
    }


    if (millis() - lastMapReset > 1000) {
        for(int i=0;i<SECTOR_COUNT;i++) lidarMap[i] = 0;
        lastMapReset = millis();
    }


    // Compute yaw
    int yaw = 1500;
    if (spin && !landing && elapsed >= 7000) yaw = SPIN_YAW;


    // Compute pitch — avoidance
    uint16_t wallDist;
    uint16_t floorDist;
    if (xSemaphoreTake(sensorMutex, pdMS_TO_TICKS(2))) {
        floorDist = lastSensorDistance;
        wallDist = lastWallDistance;
        xSemaphoreGive(sensorMutex);
    } else {
        floorDist = 0;
        wallDist = 0;
    }


    int pitch = NEUTRAL_PITCH;
```

```
      if(hover && !landing && toggleAvoidance && spin)
      {
          pitch = computeAvoidancePitch();
      }
      if (!hover || landing) avoiding = false;

      // Write to shared CRSF output
      if (xSemaphoreTake(crsfMutex, pdMS_TO_TICKS(3))) {
          sharedThrottle = throttle;
          sharedAux      = aux;
          sharedYaw      = yaw;
          sharedPitch    = pitch;
          xSemaphoreGive(crsfMutex);
      }

      vTaskDelay(pdMS_TO_TICKS(70));  // 50Hz alt hold update rate
   }
}

// ---- Task 3: Sensor Read — Core 1, medium priority ----
// I2C runs on Core 1, never conflicts with BLE on Core 0
void sensorTask(void* pvParameters) {

   // Reinitialise the sensor in-place without blocking other tasks
```

```
auto reinitSensor = []() -> bool {
    Serial.println("[LIDAR] Attempting reinit...");

    if (sensor != nullptr) {
        sensor->stop_ranging();
    }

    // Brief bus recovery pause
    vTaskDelay(pdMS_TO_TICKS(200));

    // Check sensor is still physically present on I2C
    Wire.beginTransmission(0x29);
    if (Wire.endTransmission() != 0) {
        Serial.println("[LIDAR] Sensor not responding on I2C");
        return false;
    }

    if (sensor->init() != VL53L8CX_STATUS_OK) {
        Serial.println("[LIDAR] Reinit failed");
        return false;
    }

    sensor->set_resolution(VL53L8CX_RESOLUTION_4X4);
    sensor->set_ranging_frequency_hz(36);
```

```
    sensor->start_ranging();
    Serial.println("[LIDAR] Reinit OK");
    return true;
};

uint8_t  consecutiveFailures = 0;
uint32_t lastFailLog         = 0;
const uint8_t  MAX_FAILURES  = 5;    // failures before reinit attempt
const uint32_t LOG_INTERVAL  = 1000; // only print failure once per second

while (true) {
    if (!sensorReady || sensor == nullptr) {
        vTaskDelay(pdMS_TO_TICKS(distanceInterval));
        continue;
    }

    uint8_t dataReady = 0;
    uint8_t status    = sensor->check_data_ready(&dataReady);

    if (status != VL53L8CX_STATUS_OK) {
        consecutiveFailures++;

        // Rate-limited log — don't spam serial
        if (millis() - lastFailLog >= LOG_INTERVAL) {
```

```
                Serial.printf("[LIDAR] check_data_ready failed (x%d)\n",
consecutiveFailures);

                lastFailLog = millis();

            }



            if (consecutiveFailures >= MAX_FAILURES) {

                Serial.println("[LIDAR] Too many failures — reinitialising sensor");

                consecutiveFailures = 0;



                sensorReady = false;

                bool ok = reinitSensor();

                sensorReady = ok;



                if (!ok) {

                    // Reinit failed — wait longer before retrying

                    vTaskDelay(pdMS_TO_TICKS(2000));

                }

            }



            vTaskDelay(pdMS_TO_TICKS(distanceInterval));

            continue;

        }



        // Successful status check — reset failure counter
```

```
        consecutiveFailures = 0;



        if (!dataReady) {

            vTaskDelay(pdMS_TO_TICKS(distanceInterval));

            continue;

        }



        status = sensor->get_ranging_data(&measurementData);

        if (status != VL53L8CX_STATUS_OK) {

            Serial.println("[LIDAR] get_ranging_data failed");

            vTaskDelay(pdMS_TO_TICKS(distanceInterval));

            continue;

        }



        auto validStatus = [](uint8_t s) {

            return s == 4 || s == 5 || s == 6 || s == 7 || s == 9 || s == 10;

        };

        auto clampDistance = [](uint8_t s, uint16_t d) -> uint16_t {

            if (s == 4 || s == 7 || s == 10) return 10;

            return d;

        };



        // Floor zones (12-15)

        uint32_t sum   = 0;
```

```
    int      count = 0;
    for (int i = 12; i <= 15; i++) {
        uint8_t  s = measurementData.target_status[i];
        uint16_t d = measurementData.distance_mm[i];
        if (validStatus(s)) { sum += clampDistance(s, d); count++; }
    }

    if (count > 0) {
        uint16_t rawDist = sum / count;
        lastValidDistance  = rawDist;
        sensorBadReadCount = 0;
        if (xSemaphoreTake(sensorMutex, pdMS_TO_TICKS(5))) {
            lastSensorDistance = rawDist;
            xSemaphoreGive(sensorMutex);
        }
    }

    // Wall zones (0-7) — only when spinning
    if (spin) {
        uint32_t wallSum   = 0;
        int      wallCount = 0;

        for (int i = 0; i <= 7; i++) {
            uint8_t  s = measurementData.target_status[i];
```

```
            uint16_t d = measurementData.distance_mm[i];
            if (validStatus(s)) { wallSum += clampDistance(s, d); wallCount++; }
        }

        if (wallCount > 0) {
            uint16_t wallDist = wallSum / wallCount;

            float yaw = 0;
            if (xSemaphoreTake(telemetryMutex, pdMS_TO_TICKS(2))) {
                yaw = attData.yaw;
                xSemaphoreGive(telemetryMutex);
            }

            int sector = getSector(yaw);

            if (xSemaphoreTake(sensorMutex, pdMS_TO_TICKS(5))) {
                lastWallDistance = wallDist;
                if (wallDist > 0) {
                    if (lidarMap[sector] == 0 || wallDist < lidarMap[sector]) {
                        lidarMap[sector] = wallDist;
                    }
                }
                xSemaphoreGive(sensorMutex);
            }
```

```
            }

        }


        vTaskDelay(pdMS_TO_TICKS(distanceInterval));

    }

}


// ---- Task 4: BLE Notify — Core 0, low priority ----

// All BLE operations stay on Core 0 with the NimBLE stack

void bleNotifyTask(void* pvParameters) {

    while (true) {

        uint16_t distToSend;

        if (xSemaphoreTake(sensorMutex, pdMS_TO_TICKS(5))) {

            distToSend = lastSensorDistance;

            xSemaphoreGive(sensorMutex);

        } else {

            distToSend = 0;

        }


        if (distToSend > 0) notifyDistance(distToSend);

        notifyBattery();

        notifyAttitude();

        notifyFlightMode();

        notifyLidarMap();
```

```
        vTaskDelay(pdMS_TO_TICKS(200));
    }
}

// ================================================================
//  SECTION 15 — Setup
// ================================================================

void setup() {
    Serial.begin(115200);

    // Create mutexes before any tasks start
    crsfMutex      = xSemaphoreCreateMutex();
    sensorMutex    = xSemaphoreCreateMutex();
    telemetryMutex = xSemaphoreCreateMutex();
    stateMutex     = xSemaphoreCreateMutex();

    setupDistanceSensor();
    for(int i=0;i<SECTOR_COUNT;i++) lidarMap[i] = 0;

    setupBLE();           // NimBLE binds to Core 0 internally

    pinMode(SPEAKER_PIN, OUTPUT);
```

```
  ledcSetup(LEDC_CHANNEL, 2700, LEDC_RES);

  ledcAttachPin(SPEAKER_PIN, LEDC_CHANNEL);

  ledcWrite(LEDC_CHANNEL, 0);



  FC.begin(420000, SERIAL_8N1, 3, 1); // 3, 1



  Serial.println("Playing song...");

  for (int i = 0; i < melodyLen; i++) tone_(melody[i][0], melody[i][1]);



  esp_reset_reason_t reason = esp_reset_reason();

  switch (reason) {

      case ESP_RST_POWERON:   Serial.println("Power-on reset"); break;

      case ESP_RST_EXT:       Serial.println("External reset"); break;

      case ESP_RST_SW:        Serial.println("Software reset"); break;

      case ESP_RST_PANIC:     Serial.println("Exception/panic"); break;

      case ESP_RST_INT_WDT:   Serial.println("Interrupt watchdog"); break;

      case ESP_RST_TASK_WDT:  Serial.println("Task watchdog"); break;

      case ESP_RST_WDT:       Serial.println("Other watchdog"); break;

      case ESP_RST_DEEPSLEEP: Serial.println("Wake from deep sleep"); break;

      case ESP_RST_BROWNOUT:  Serial.println("Brownout"); break;

      case ESP_RST_SDIO:      Serial.println("SDIO reset"); break;

      default:                Serial.println("Unknown"); break;

  }
```

```
    // ---- Create tasks ----
    // CRSF task — Core 1, priority 5 (highest) — must never be preempted
    xTaskCreatePinnedToCore(crsfTask,      "CRSF",      4096, NULL, 5, NULL, 1);

    // AltHold task — Core 1, priority 3
    xTaskCreatePinnedToCore(altHoldTask,   "AltHold",   8192, NULL, 4, NULL, 1);

    // Sensor task — Core 1, priority 2 (I2C stays on Core 1)
    xTaskCreatePinnedToCore(sensorTask,    "Sensor",    8192, NULL, 3, NULL, 1);

    // BLE notify task — Core 0, priority 1 (NimBLE stack owns Core 0)
    xTaskCreatePinnedToCore(bleNotifyTask, "BLENotify", 4096, NULL, 2, NULL, 0);

    Serial.println("[RTOS] All tasks started");
}

// loop() intentionally minimal — all work is in tasks
void loop() {
    vTaskDelay(pdMS_TO_TICKS(10000));
}
```