Indoor Surveillance Drone

ELEC5552 Team 04

Author(s):

Sofia Khokhlenok (23099645)
Sunny Guo (23345534)

Final Design Report

Project Partners:	Michal Zawierta & Dilusha Silva, UWA Aviation Labs
UWA Supervisor:	Jega Gurusamy
Team Number:	04
Meeting Time:	Thursday, 10AM
Word Count:	<#>
Word Limit:	7000
Version:	1

17 October 2025

Revision History

Date	Version	Description	Author
<dd/mm/yyyy>	1.0	Initial Draft	Sofia Khokhlenok, Sunny Guo
<dd/mm/yyyy>	<x.x>	<Insert description of changes>	<Author Name>

Executive Summary

Contents

List of Figures

List of Tables

Acronyms

DC Direct Current.

ESP-IDF Espressif IoT Development Framework.

GPS Global Positioning System.

I^{2}C Inter-Integrated Circuit.

IMU Inertial Measurement Unit.

LiPo Lithium Polymer battery.

MCU Microcontroller.

NGO Need, Goals, Objectives.

PCB Printed Circuit Board.

PID Proportional-Integral-Derivative.

RTOS Real Time Operating System.

SPI Serial Peripheral Interface.

ToF Time-of-Flight Sensor.

UART Universal Asynchronous Receiver/Transmitter.

.

UI User Interface.

USB Universal Serial Bus.

UWA The University of Western Australia.

UWAAL UWA Aviation Labs.

WPA2 Wi-Fi Protected Access 2.

1 Introduction

Unmanned Aerial Vehicles (UAVs), commonly referred to as drones, have become an essential tool across a wide range of applications, including surveying, inspection, and research. Their ability to operate in confined or inaccessible environments makes them particularly valuable for close-range observation and monitoring tasks. Unlike outdoor drones that rely on satellite-based positioning systems such as GPS, indoor drones must depend entirely on onboard sensing, control algorithms, and communication systems to maintain stability and navigate safely. Operating in enclosed environments presents additional challenges, including limited spatial awareness, airflow disturbances, and heightened safety risks due to proximity to structures and personnel. Consequently, the design of an indoor drone requires careful consideration of system architecture, control robustness, and physical safety features.

The UWA Aviation Labs (UWAAL) has commissioned the development of an indoor drone system capable of autonomous and stable flight within a confined environment. The drone will be used to survey and inspect indoor areas such as garages or laboratory spaces, following predefined flight paths and maintaining stable altitude and position despite the influence of external airflow. The system must integrate onboard sensing, control, communication, and power management within a compact and lightweight platform, designed from first principles to ensure safety, efficiency, and reliability. This project explores the design and implementation of such a system, aligning with UWAAL's objective to develop a safe, functional, and adaptable indoor drone platform for future research and educational applications.

1.1 Scope of This Report

This report outlines the process undertaken to design and implement a functional indoor drone system for operation within confined environments. It defines the technical, safety, and regulatory boundaries within which the project was executed, as well as the methods used to evaluate performance against requirements agreed upon with consoltation with the client. The scope includes system-level design, hardware and firmware integration, testing, and validation to ensure stable flight, effective altitude control, reliable communication, and key safety features.

The document also presents the rationale behind key design choices and the underlying design archiecture and philosophy. It also includes specific design elements, including component selection, PCB schematic design and layout, flight control systems, and safety mechanisms, while addressing relevant standards governing electrical safety, risk management, and compliance. Consideration is given to manufacturability, maintainability, and adherence to the project's cost and resource constraints. Beyond the immediate development outcomes, the report establishes a framework that can support future extensions of the platform within the UWAAL.

1.2 Purpose of the Design

The primary objective of this project is to design, develop, and validate a low-cost indoor drone system capable of achieving stable flight and hovering performance in the absence of GPS-based navigation. The system is intended for application within the UWAAL to perform autonomous operations maintaining a specified altitude and compensating for environmental disturbances such as localised airflow.

The proposed system architecture comprises a custom-designed Printed Circuit Board (PCB) flight controller integrating onboard power distribution, sensing, and control modules. Flight stabilisation and altitude maintenance are achieved through sensor fusion of inertial and distance measurements processed via integrated control algorithms. The design should incorporate safety features including shrouded propellers or propeller guards, low-battery auto-landing capability, and an emergency failsafe mechanism

that initiates a controlled landing or motor shutdown following a communication loss exceeding thirty seconds. Moreover, the drone is constrained to operate on 1S or 2S lithium polymer batteries with a maximum capacity of 800 mAh and is required to sustain a minimum flight duration of three minutes. All components and materials must conform to the allocated project budget of AUD $350. The overarching purpose of this design is to deliver a robust, safe, and open-source indoor drone platform that satisfies all operational, safety, and budgetary requirements.

1.3 Background

UAVs combine sensing, computation, and actuation to achieve controlled flight through coordinated motor thrust. Their operation depends on precise feedback control systems that maintain stability and orientation while responding to pilot or autonomous commands. This is typically accomplished through onboard microcontrollers running real-time control algorithms that process data from inertial and range sensors to adjust motor outputs dynamically.

The ESP32 microcontroller has emerged as a compact and cost-effective platform for such systems, integrating dual-core processing, Wi-Fi connectivity, and sufficient computational power to handle both flight control and communication tasks. Its compatibility with open-source development frameworks such as Espressif's Espressif IoT Development Framework (ESP-IDF) enables flexible low-level implementations of control loops, sensor fusion, and telemetry within a single embedded system.

In parallel, advances in printed circuit board (PCB) design allow the integration of flight electronics, power management, and motor drivers onto lightweight, custom boards that minimize wiring and improve reliability. Together, these developments make it feasible to design small-scale, low-cost autonomous drones capable of stable flight and sensor-based navigation for educational, research, and experimental applications.

1.4 Summary of Contributions

To-do

1.5 Report Structure

This report is structured to provide a comprehensive overview of the design and development of the autonomous ESP32-based drone. Section 1 introduces the project, outlining its scope, purpose, background, and key contributions. Section 2 details the design process, including system requirements, hardware and firmware architecture, component selection, and integration. It also addresses testing procedures, safety, ethical considerations, and identified risks. Section 3 presents recommendations for further development and improvement. Section 4 provides the user manual for operation and maintenance of the system, and Section 5 lists all referenced materials and supporting documentation. Any additional material can be found in the Appendices.

2 Design

2.1 Requirements

Project requirements define the measurable outcomes of the design process and establish a shared understanding between the client and the development team. Each requirement must be technically sound, verifiable, and achievable within the defined project constraints.

Requirements were derived from the project brief, client discussions, and technical analysis. Constraints and assumptions were first identified, followed by a prioritisation process based on technical criticality, safety relevance, and feasibility. A detailed record of this process is available in the *Requirements Report*.

Table 7 describes the constraints for the requirements which are given in Appendix A.1. The ranked requirements are given below in 2.

Table 2: Ranked Requirements

Rank	No.	Requirement	Constr.
1	R.2	The drone shall be capable of hovering and flying.	C.1
2	R.30	The drone must power off after 30 s of disconnection.	C.6
3	R.1	The drone shall operate in an indoor area.	C.3
4	R.17	The drone shall utilise shrouded propellers or propeller guards.	C.6
5	R.3	The drone shall navigate without the use of Global Positioning System (GPS).	C.3
6	R.4	The drone shall maintain a set hovering altitude during autonomous surveillance.	C.3
7	R.14	A manual override system shall be available to the user.	C.2, C.7
8	R.27	The drone shall autonomously land safely when battery levels are low.	C. 3
9	R.24	Any third-party software used for control must be open-source and freely available.	C.2
10	R.23	The drone design shall remain within a budget of $350.00.	C.4
11	R.20	The drone shall be capable of flight stabilisation.	C.2, C.7
12	R.5	The drone shall maintain a default altitude between 1.2–1.3m during operation.	C.2
13	R.18	Power shall be supplied by one 1S or 2S Lithium Polymer battery (LiPo) battery ($<$800 mAh).	C.6
14	R.15	The flight controller shall use a custom PCB integrating sensors and power distribution.	C.2
15	R.21	The drone shall maintain stability under wind disturbances.	C.2, C.7

There are also several lower-priority requirements that are not essential to meeting the primary project goals but remain background considerations. These features may still be implemented if they can be added with minimal time or resource overhead. The additional requirements are given in Appendix A.2.

The ranked requirements form the foundation for architectural, hardware, and software decisions presented in the following sections.

2.2 Design Architecture

The design architecture defines how major subsystems interact to meet functional, safety, and performance requirements. It establishes clear boundaries between sensing, control, communication, and actuation components, ensuring reliable operation and outlines the integration of these components with eachother.

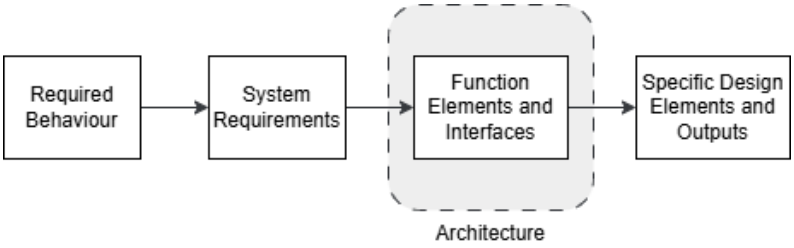

Figure 1: System Architecture Process

While the desired behaviour of the systems defines the system requirements, along with additional aspects such as safety and any additional constraints that alter the final outcome, the system architecture is the high-level realisation of these requirements into system which clearly defines the functional elements and interfaces between them. From this, specific design components are chosen to best represent the architecture.

2.2.1 Overall Architecture

The overall system architecture, shown in Fig. 2, is divided into three main layers: the User–Web Interface, the Web–Firmware Interface, and the Firmware–Hardware Interface. These layers communicate through well-defined protocols to ensure reliable and modular operation.

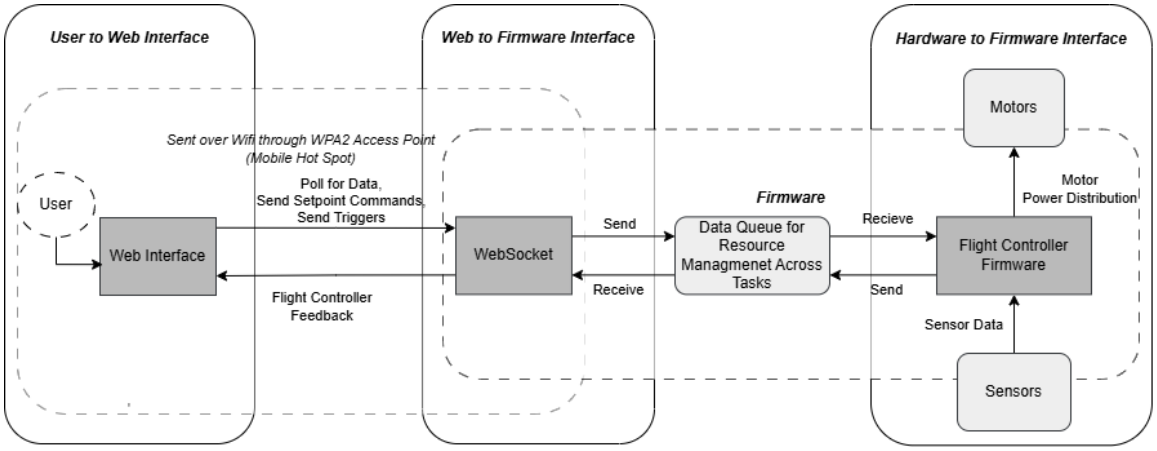

Figure 2: Overall System Architecture with Hardware and Firmware Interfaces

User to Web Interface: The operator controls the drone and monitors system feedback via a web interface supporting both manual and autonomous flight. Real-time telemetry, sensor feedback, and emergency motor shutdown are provided over a WPA2-secured Wi-Fi connection.

Web to Firmware Interface: A persistent WebSocket link connects the web interface and the flight controller. Three message types are supported: (i) *data requests* for sensor feedback, (ii) *setpoint commands* for control updates, and (iii) *control triggers* for testing or initiating sequences. The WebSocket and control processes run as independent Real Time Operating System (RTOS) tasks, communicating via thread-safe queues to prevent race conditions.

Firmware to Hardware Interface: The flight controller directly interfaces with onboard sensors and motor drivers. Sensor data is continuously processed to generate thrust commands that maintain stable flight. The control loop responds in real time to user or autonomous inputs, ensuring stable hover and smooth manoeuvres.

The following sections describe the hardware and firmware architectures in further detail.

2.2.2 Hardware Architecture

The hardware architecture integrates sensing, control, communication, and power management into a single lightweight PCB mounted on a safety-framed airframe. Fig. 3 illustrates the main components and their interconnections.

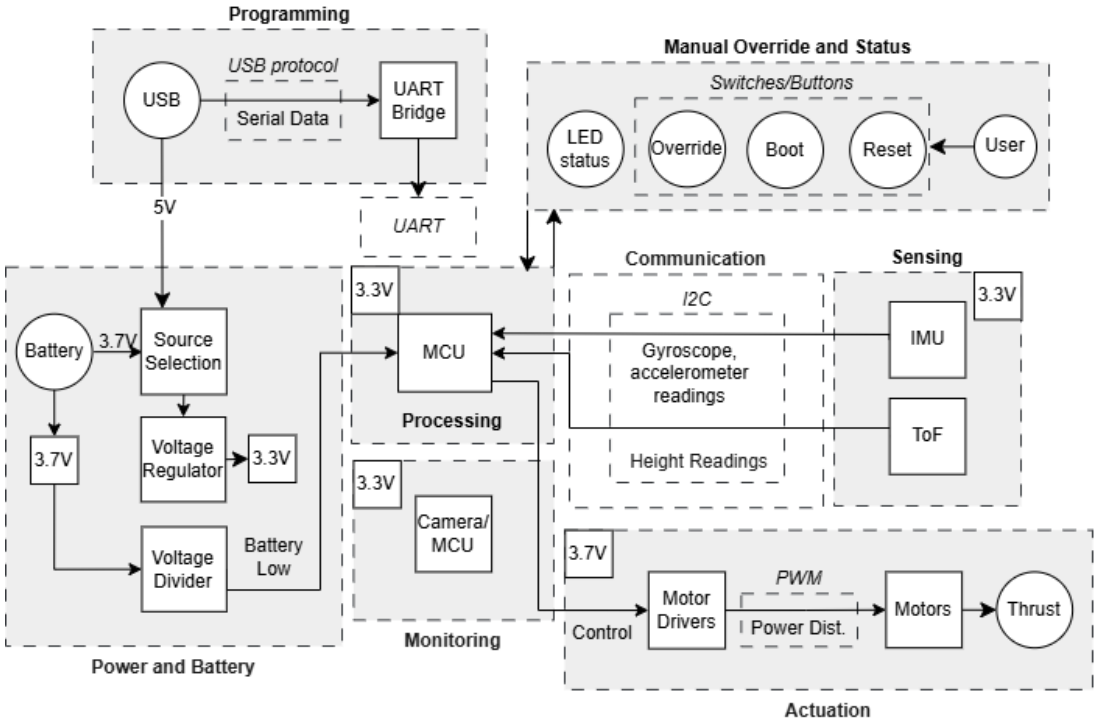

Figure 3: Hardware Architecture

A description of each of the architectural elements is given below in Table 3. ach element of the hardware architecture supports a specific functional or safety requirement, as outlined in Table 3. Additionally, the PCB design must maintain signal integrity, minimise power losses, and allow efficient component routing (R.15). The structural frame supports the PCB, battery, and motors, incorporating propeller guards to satisfy safety requirements (R.17).

Table 3: Hardware Elements and Requirements

Architectural Element	Purpose	Requirement(s)
Actuation and Sensing	Provides continuous data needed for state estimation (position and orientation of the drone), control stability, and safety.	R.2, R.1, R.3, R.4, R.20, R.5, R.21
Power and Battery	Regulates power supply input and manages power from battery.	R.27, R.18
Processing, Programming, Communication	Microcontroller (MCU) executes control loops and manages peripheral communication. Programming allows firmware upload and monitoring of diagnostics. Communication provides telemetry between sensors, actuators, and processing tasks.	All except R.17, R.14, R.18, R.15
Actuation	Converts control signals into thrust outputs.	R.2, R.1, R.4, R.27, R.29, R.5
Manual Override and Status	Provides manual override and status via LEDs.	Additional
Monitoring	Provides a means for surveillance of the environment.	Additional

2.2.3 Hardware Justification

The architecture was informed by several open-source designs—CircuitDigest's ESPDrone, Espressif's ESP-Drone, and Max Imagination's ESP-FLY - alongside Espressif's official ESP32 DevKit schematics. These references provided a practical reference guide into lightweight drone design, component choices, and standard circuit integration. The final design combines these layouts with project-specific adaptations. [Sources]

Table 4: Design Element Justification

Element	Rationale
Power and Battery	The ESP32 requires a stable 3.3 V supply. A source selection circuit allows both battery and Universal Serial Bus (USB) connection, with priority given to USB. Battery low levle is monitored via simple voltage divider to warn when charge is low.
Programming	Firmware uploading and debugging require a USB-to-Universal Asynchronous Receiver/Transmitter (UART) bridge, providing direct serial communication with the MCU.
Sensing and Communication	An Inertial Measurement Unit (IMU) is necessary for flight stability and position control. A Time-of-Flight Sensor (ToF) sensor simplifies height measurement and reduces error accumulation from integration. Sensors use standard protocols such as I^2C for reliable communication with the MCU.
Actuation	Direct Current (DC) motors controlled via Pulse Width Modulation (PWM) generate thrust. Individual motor control ensures balance and stabilisation. Simple motor driver circuits (Integrated Circuits (ICs) or Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) switches) provide reliable control for each motor.
Monitoring	A separate MCU for the surveillance camera ensures independent processing from the flight controller for modularity.
Manual Override and Status	Physical switches and LEDs provide user feedback and safety. Manual power-off allows immediate shutdown. Status LEDs indicate power and connectivity.

The design elements and the rational begind them is given in Table 4. Add description of changes we made for the research

2.2.4 Firmware Architecture

The firmware architecture defines the drone's control logic and operational behaviour. It manages sensor acquisition, state estimation, and motor actuation within a real-time stabilisation loop. Sensor fusion combines IMU and time-of-flight (ToF) data to estimate orientation and altitude, while cascaded Proportional-Integral-Derivative (PID) controllers maintain attitude and height stability.

A WebSocket-based communication layer enables remote telemetry and command control via a Wi-Fi access point. Safety routines include manual override and automatic motor shutdown in case of communication loss or detected fault conditions.

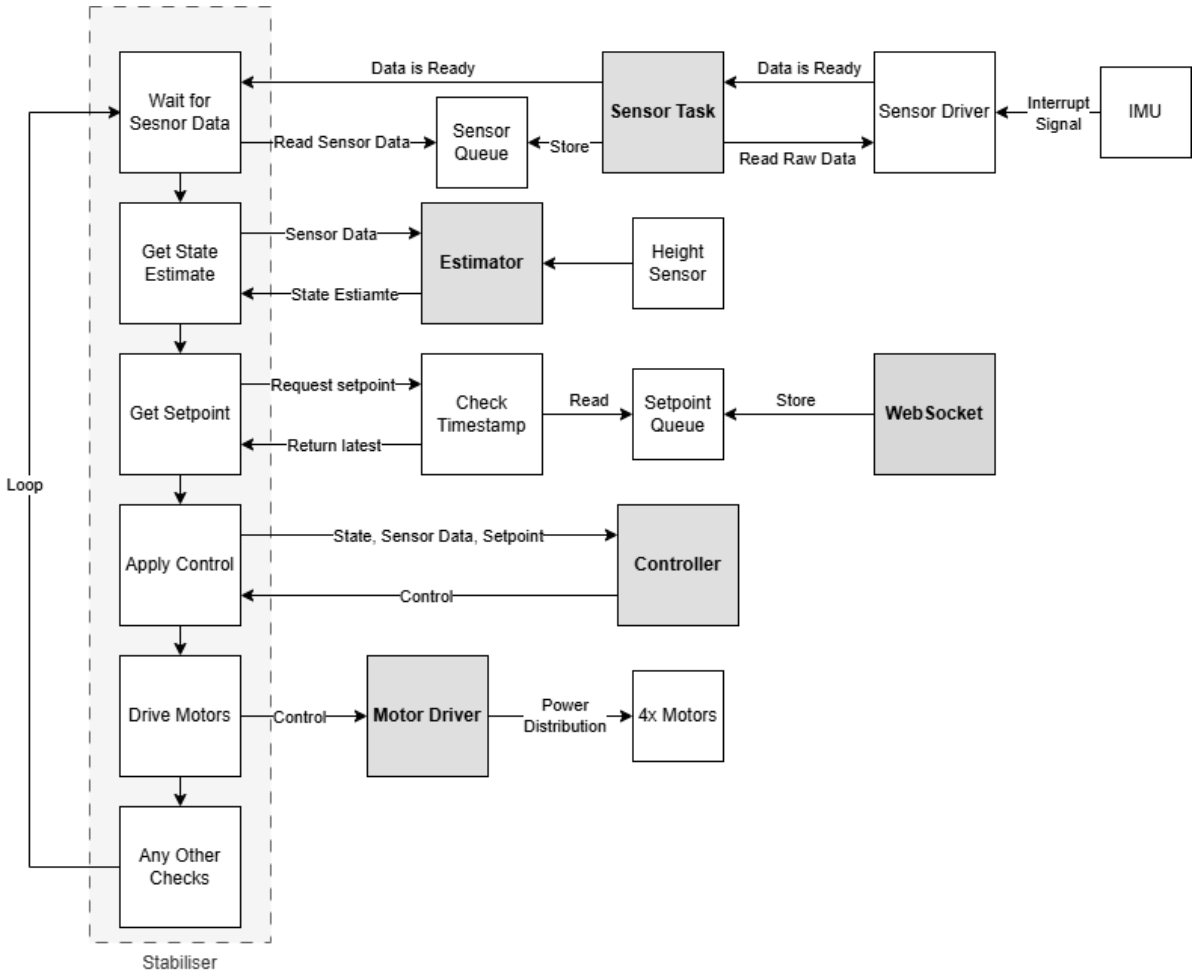

Figure 4: Firmware Architecture

2.2.5 Firmware Architecture Justification

Firmware using the ESP32 was reviewed for the firmware architecture including Espressif's ESP-Drone (or more so, Circuit Digest's modified version [D], with fixed UDP driver needed for Crazyflie Client (CFClient)). It is built on the ESP-IDF framework and derived from the Crazyflie open-source project (GPL-3.0) [C] and was assessed as a baseline for stable flight and mobile control. Phadte's ESP32 Flight Controller [E], developed with the Arduino framework, was reviewed a simple custom implementation of a flight controller, built for a larger brushless motor drone, and the PID tuning methodology.

Based on this analysis, a custom firmware was developed using ESP-IDF v5.5.0 to provide long-term flexibility, low-level control, and improved integration. This approach simplified the generalised Crazyflie libraries of ESP-Drone, which contain dozens of source files and many macros and generalisations to work with multiple drone designs. Then, the lower-level ESP-IDF environment was chosen over Arduino to achieve more precise timing, and compatibility with ESP-IDF libraries.

This configuration ensures stable flight under constrained indoor conditions while maintaining extensibility for future autonomous functionality and research development within the UWAAL.

Table 5: Firmware Option Comparison

Feature	**ESP-Drone**	**Phadte**	**Custom**
Framework	ESP-IDF v4.4.8	Arduino	ESP-IDF v5.5.0
Control Type	Manual	Manual	Manual/Autonomous
Complexity	High	Low	Moderate
Flexibility	Moderate	Low	High
Integration	CFClient only	Arduino libraries	ESP-IDF libraries
Primary Focus	General purpose	Flight stabilisation only	Project specific surveillance

The final architecture balances flexibility and control, centring on a stabilisation loop that integrates IMU feedback, cascaded PID control, and motor actuation derived from the Crazyflie core library, as shown in Fig. 4.

Table 6: Firmware Architecture Rationale

Element	**Purpose**	**Rationale**
Sensor Task	Acquire data from onboard sensors.	Sensor inputs are required to estimate the drone's state. IMU and ToF sensors were used to meet project requirements while remaining compatible with open-source third-party software if needed.
State Estimator	Combine sensor inputs to estimate attitude (roll, pitch, yaw) and altitude.	State estimation is necessary for the drone to determine its orientation, needed for stabilisation.
Motor Driver	Convert control signals into motor speed commands via PWM or driver interface.	Motor control generates thrust based on output from the PID control system.
Override	Handle manual override, loss of communication, and fault detection.	Safety functions include fault detection, manual override, and controlled handling of errors to prevent unsafe operation.
Communication	Provide telemetry, control input, and data logging through WebSocket or serial link.	Communication with the WebSocket is required for remote activation or deactivation, command transmission, and system feedback. It also supports debugging and monitoring.
Timing/Stabiliser	Coordinate execution timing using an RTOS.	Timing management is critical for stabilisation, ensuring rapid response to changes in drone position or environment.
User Interface	Present data and system status to the operator via wireless connection.	The user interface enables remote deactivation, monitoring of system status, and basic control of the drone.

2.3 Design Elements

2.3.1 Firmware Elements

The firmware elements, which are described in Fig. 4, are described in detail below. The detailed implementation of these elements in given in [1].

2.3.1.1 Sensor Task *sensors.c*

The sensor task provides an interface for the MEMS Accelerometer and Gyroscope IC (MPU6050) 6-axis IMU through Inter-Integrated Circuit (I^2C), which provides real-time data from the accelerometer and gyroscope on the sensor. This is done on a register level using Espressif's *I2C* library.

The raw data obtained from the MPU6050 sensor are 16-bit integer values representing the gyroscope and accelerometer measurements. These values are first converted into physical units (SI units) based on the selected sensitivity configuration of the sensor. The conversion from raw readings to angular velocity and linear acceleration is expressed as:

$$\boldsymbol{\omega} = (\mathbf{G}_{\text{raw}} - \mathbf{G}_{\text{bias}}) \times S_{\text{gyro}}, \tag{1}$$

$$\mathbf{a} = \mathbf{A}_{\text{raw}} \times S_{\text{acc}}, \tag{2}$$

where $\mathbf{G}_{\text{raw}}$ and $\mathbf{A}_{\text{raw}}$ are the raw gyroscope and accelerometer readings, $\mathbf{G}_{\text{bias}}$ is the gyroscope bias, which is calculated when the filter is intialised. S_{gyro} and S_{acc} are the corresponding sensitivity scaling factors.

To reduce measurement noise, a second-order digital biquad low-pass filter is applied to the sensor data. This filter attenuates high-frequency noise while preserving the motion dynamics of interest. Cutoff frequencies of 30 Hz and 80 Hz are used for the accelerometer and gyroscope, respectively. Additionally, the MPU6050 incorporates an internal digital low-pass filter on the gyroscope measurements, which is also enabled for further signal smoothing.

The digital biquad filter can be expressed in the z-domain as:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}, \tag{3}$$

$$w(n) = x(n) - a_1 w(n-1) - a_2 w(n-2), \tag{4}$$

$$y(n) = b_0 w(n) + b_1 w(n-1) + b_2 w(n-2), \tag{5}$$

where $x(n)$ is the current input sample, $y(n)$ is the filtered output, and $w(n)$ represents the internal state variables in Direct Form II. The coefficents can be found in *filter.c*.

A interrupt service routine (ISR) is attached to the MPU6050 interrupt, this means that instead of the sensor task continuously reading from the MPU6050, it waits for the trigger to signify that data is ready to be read. The data is then stored for a queue and an additional signal is sent for the stabiliser task to signify the processed sensor data is ready.

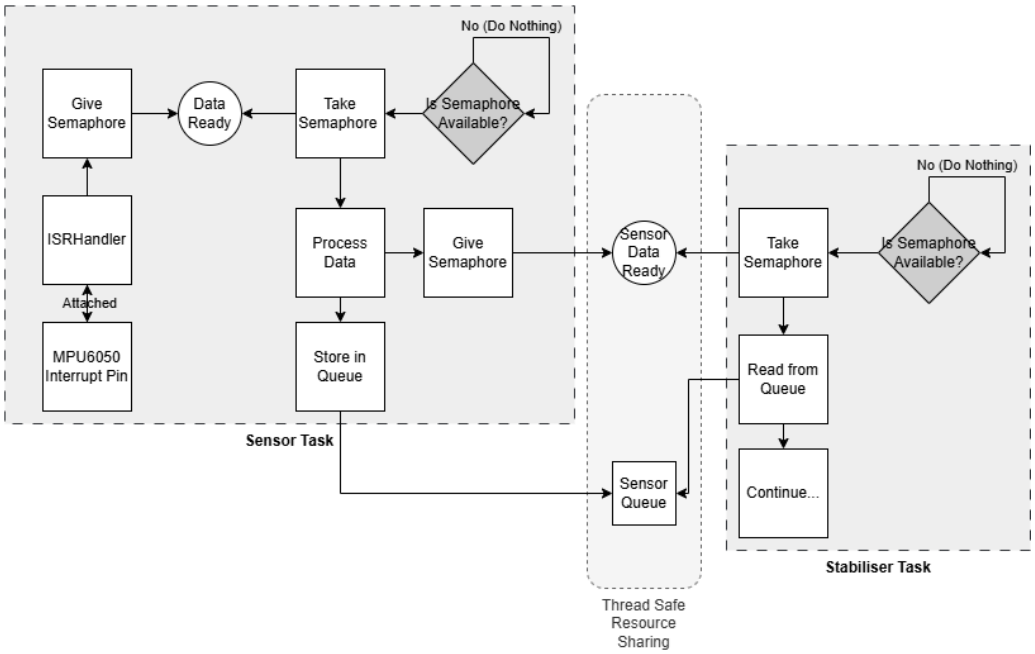

Figure 5: Thread-safe Data Availability Signalling Heurisitic

The code for this can be found in A.3.1.

2.3.1.2 State Esitmation *estimator.c*

The MPU6050 measures angular rate from the gyroscope and linear acceleration from the accelerometer, which do not directly provide information about its orientation or position in space. The drone does not have inherent knowledge of its position relative to world coordinates. Simple integration can be used to estimate this, but it leads to large errors due to accumulation in the integration. A more complex technique is required.

The quaternion number system extends complex numbers to three dimensions. The detailed mathematics are omitted here, but practically it solves issues with the Euler angle system. One issue is gimbal lock: when two of the three Euler axes align, a degree of freedom is lost. This does not mean the axes are physically locked, but smooth motion to all orientations is disrupted until the axes "unlock."

This number system and an AHRS algorithm called Mahony's algorithm are used in the CrazyFlie firmware to achieve state estimation. This estimate is sent to the WebSocket for visualisation.

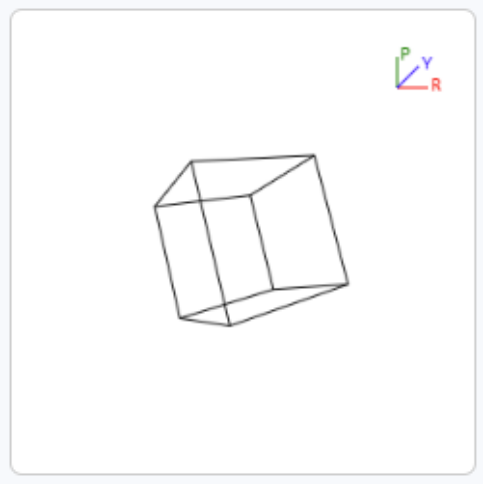

Figure 6: Real-Time State Estimate Display on WebSocket

2.3.1.3 Controller *controller.c*

Once the state estimate is and the setpoint is recieved, essentially saying ẗhis is my current stateänd ẗhis is the state I want to be in; the realisation of the control to reach the setpoint is met through the use of a control system, in this case a cascaded PID loop.

[PID diagram]

Short explanation

2.3.1.4 Stabiliser *stabiliser.c*

The stabiliser ensures that all other processes in the flight controller run on time and repeatedly, acting as the centralised task from which other functions are called. A key feature is its timing system: different parts of the controller run at different frequencies using a tick-based approach. This applies both in the state estimator, which updates the attitude and position estimates, and in the controller, which computes the control outputs.

For stabilisation, the attitude loop, controlling roll, pitch, and yaw, runs at a high frequency (500 Hz) to respond quickly to rapid angular changes. The position loop, controlling altitude and horizontal position, runs at a lower frequency (100 Hz) since translational motion is slower. This setup ensures fast dynamics are stabilised quickly, while slower dynamics are controlled efficiently.

2.3.1.5 WebSocket *websocket.c*

The WebSocket interface provides a real-time connection between the flight controller and a web-based user interface. It allows the drone to send sensor readings, state estimates, and control outputs to the browser, while receiving setpoints and manual control commands from the user.

The HTML interface displays IMU data, attitude, position, and motor outputs, and provides sliders and buttons for manual control. The WebSocket ensures that these updates are transmitted asynchronously, so the stabiliser and controller loops continue running at their required rates. Users can enable or disable motors, adjust setpoints, and request telemetry in real time, with the interface handling both periodic state updates and on-demand commands efficiently.

[Image]

2.3.1.6 Motors *motors.c*

The motors module converts the outputs of the stabiliser and controller into signals for the four drone motors. The controller outputs *thrust* and attitude control corrections (*roll, pitch, yaw*) which are distributed to each motor to achieve the desired motion. This is known as power distribution, and for a quadcopter in an X-configuration, the motor commands are calculated as:

$$
\begin{aligned}
M_1 &= T - \frac{R}{2} + \frac{P}{2} + Y \\
M_2 &= T - \frac{R}{2} - \frac{P}{2} - Y \\
M_3 &= T + \frac{R}{2} - \frac{P}{2} + Y \\
M_4 &= T + \frac{R}{2} + \frac{P}{2} - Y
\end{aligned}
$$

where M_i is the command for motor i, T is the total thrust, R, P, Y are the roll, pitch, and yaw corrections respectively. The roll and pitch contributions are halved to balance their effect across opposing motors.

Each motor command is then limited to a minimum *idle thrust* to ensure stable spinning even when the drone is hovering:

$$
M_i = \max(M_i, T_{\text{idle}})
$$

Finally, these commands are converted to PWM signals for the motor ESCs, with a linear mapping from the 16-bit thrust command to the duty cycle of the PWM hardware:

$$
\text{PWM}_i = \text{motorConv16ToBits}(M_i)
$$

This mapping allows the controller outputs to directly control motor speeds, which in turn generate the required forces and torques to stabilise and manoeuvre the drone.

2.4 Testing

The testing phase was conducted to validate the design elements, demonstrate proof of concept, and identify areas requiring modification. Testing was split into *Initial Stationary Testing*, which assessed sensors, communication, and basic control logic without flight, and *Dynamic Testing*, which evaluated flight performance, stability, and PID control during lift-off and hovering.

2.4.1 Initial Stationary Testing

Initial testing did not involve flight but focused on validating core hardware and firmware components:

1. MPU6050 IMU and VLX distance sensors were tested separately to verify functionality and accurate readings.
2. Sensors were integrated with the Espressif EPSDrone firmware to check compatibility and basic data retrieval via CFClient.
3. IMU calibration and testing confirmed gyroscope and accelerometer measurements, noting minor inconsistencies that could be compensated through bias correction.

4. WebSocket interface development enabled wireless command and telemetry communication, progressively tested to ensure correct reception and feedback (linked to R.25 - User Interface).
5. State estimation algorithms were implemented and verified visually to confirm correspondence between actual and estimated orientation.
6. PID control and motor command modules were developed concurrently while awaiting PCB delivery.
7. PCB issues were addressed, ensuring proper hardware integration before flight tests.
8. Initial motor testing on a fixed rig revealed erratic behaviour, indicating potential motor alignment or calibration issues.
9. Early hovering attempts were limited by motor instability and resource constraints, providing insight into control tuning requirements.

Test No. XX	**R.30 - Disconnection Response** (Firmware)
Test Specification	Ensure the drone powers off safely after loss of connection, observing motor shutdown and null setpoints.
Test Description	1. Connect drone to WebSocket interface. 2. Send setpoints and verify terminal output. 3. Close the control webpage to simulate disconnection. 4. Observe if the drone detects disconnection and motors shut off.
Test Result Analysis	The drone successfully detected the disconnection and immediately ceased motor activity. This confirms proper handling of lost connection and application of null setpoints, meeting R.30.

Test No. XX	**R.3 - WebSocket State Estimate** (Firmware)
Test Specification	Validate the state estimation algorithm by comparing measured drone orientation with computed estimates.
Test Description	1. Tilt and manipulate drone manually while connected to WebSocket. 2. Monitor state estimate output and compare with actual drone movements.
Test Result Analysis	The algorithm accurately tracked the drone's orientation under moderate movements. Rapid changes introduced minor lag and drift over time, indicating acceptable performance within normal operational limits.

2.4.2 Dynamic Testing

Dynamic testing focused on lift-off, hovering, altitude maintenance, and PID stabilisation.

Test No. XX	**R.2 - Capable of Hovering** (Hardware/Firmware)
Test Specification	Validate that the drone can lift and maintain altitude, assessing motor thrust, PID control, and manual tuning.
Test Description	1. Connect drone to WebSocket interface. 2. Gradually increase thrust with PID disabled until lift-off occurs. 3. Observe behaviour at higher thrust levels. 4. Re-enable PID and attempt manual tuning of pitch/roll gains to assess stabilisation.
Test Result Analysis	The drone achieved lift-off; however, uneven thrust caused flipping tendencies at higher throttle. PID control did not improve stability, and manual tuning produced the same result. Likely causes include thrust misalignment or motor calibration errors rather than PID tuning.

Test No. XX	**R.4 / R.2 - Hovering and Altitude Maintenance** (Hardware/Firmware)
Test Specification	Assess the drone's ability to achieve and maintain a stable hover at set altitude.
Test Description	Attempt hover with PID stabilisation active, monitoring altitude and control response via WebSocket.
Test Result Analysis	Hover could not be maintained due to instability. This prevented altitude evaluation and suggests issues with motor output consistency or control tuning.

Test No. XX	**R.20 - Flight Stabilisation** (Hardware/Firmware)
Test Specification	Evaluate PID-based stabilisation and control signal application during flight.
Test Description	Monitor control signals and motor output during hovering attempts, observing PID response and integral windup.
Test Result Analysis	Hover could not be achieved, limiting assessment. Control signals were active and adjusted, but incomplete testing prevented full stabilisation. Integral windup was noted but could not be resolved.

2.4.3 Outcome of Testing

The testing process successfully validated key hardware and firmware components, including sensors, communication, motor control, and state estimation. Initial stationary tests confirmed that sensors and WebSocket communication functioned as intended. Dynamic tests demonstrated that while lift-off was achievable, full hover and altitude stability were limited by motor calibration and thrust inconsistencies. PID control operated correctly but could not compensate for underlying hardware issues. Overall, the tests identified areas requiring refinement before reliable autonomous flight could be achieved.

2.5 System Integration

Scope: Integration of flight controller, power subsystem, communication modules, and mechanical frame with defined signal/power interfaces and timing dependencies. Interfaces are version-controlled and validated per subsystem test logs.

2.6 Stakeholder Engagement

Engagement Summary:

Activity	Outcome / Actions Closed
–	–

2.7 Safety Issues

This safety plan applies to bench electronics work, soldering and rework, and controlled indoor hover testing. All activities are conducted in designated laboratory spaces with restricted access during active testing. Risk ratings follow a consequence–likelihood–exposure (C×L×E) model. Mitigations apply the hierarchy of hazard control: elimination, substitution, engineering control, administrative control, and PPE as a final layer. Full details are contained in the risk register.

Operational Boundaries and Controls:

Aspect	Defined Limits and Requirements
Environment	Indoor test room only; access restricted during hover tests with signage placed outside.
Training	All personnel must complete soldering and Li-Po handling inductions before handling any equipment.
Hover Testing	Hover altitude limited to 1.25 m AGL; any adjustment beyond this permitted only during setup phases with props disarmed.
Battery Management	Only 1S/2S Li-Po $<$800 mAh packs approved; no swollen or damaged cells. Charging supervised with fire blanket or sand bucket available.
Personnel	Minimum two people: Test Lead with authority to abort and Spotter monitoring environmental conditions and escape paths.
PPE	Safety glasses and closed footwear mandatory at all times; nitrile gloves required for chemical or flux handling tasks.

Acceptance Criteria: All tests must demonstrate thermal compliance, fail-safe functionality, clean electrical behaviour, and incident-free operation. Logs and checklists must be completed to satisfy traceability requirements.

The following table summarises key hazards, their associated risks, and control approaches.

Hazard	Risks/Mechanism	Controls Implemented	Procedure
Heating	Localised MOSFET heating or soldering burns	Copper pours, current limiting, heat mats	IR checks, switch-off when idle
Cuts/Abrasions	Propeller lacerations or tool injuries	Shrouds, guarded tools, PPE	Bench test first, enable arming LED cues
Sparking	Short circuits or solder bridging	Heat-shrink, clearance design	continuity test, no live soldering
Battery	Thermal runaway or over-discharge	Voltage alarms, visual inspection	Remove damaged cells, fire control kit on-site
Trips/Spills	Loose cables/liquids in walkways	Cable trays, dry floor policy	End-of-session clearance audit
Collisions	Impact during hover test	Altitude limit, prop shields	Incident log, abort protocol
Fumes	Flux fumes, UFPs from printing	Extraction fans, gloves	Ventilation check, SDS access

2.8 Ethical Issues

Testing takes place in a shared laboratory environment, so ethical considerations focus on safety, responsible data handling, and clear usage boundaries for open-source release. Exclusion zones will be clearly marked, and written approval is obtained from space owners before any powered testing. A spotter is always present to ensure that no unauthorised person enters the test area.

Data and Recording: The onboard camera remains disabled by default and is only activated during scheduled and approved hover trials. Any footage containing identifiable individuals will require written consent before capture. Recordings, if taken, are stored in encrypted form, retained for no longer than seven days and permanently deleted upon request.

Wireless Communication: Wi-Fi telemetry operates under least-privilege access principles. No personal device identifiers are logged. RF transmissions remain within laboratory and local regulatory limits. Channels that may conflict with co-located research equipment are excluded.

Open-Source Release and Responsible Use: All CAD, firmware, and PCB files will be published under a permissive open-source license. Documentation will clearly state:

- Intended operation is indoor hover testing only, with shrouded rotors and fully functional fail-safes.
- Prohibited uses include surveillance without explicit consent, public outdoor operation, weaponisation, or disabling of safety systems.
- Recommended safe substitution components, correct Li-Po disposal procedures, and repair instructions using reprintable PETG parts.

This ensures that any derivative builds acknowledge ethical and safety boundaries aligned with the project's academic intent.

2.9 Top 5 Risks and Mitigations

The following risks have the highest inherent risk ranking based on (C×L×E). Mitigations follow the hierarchy of controls, and each risk includes a defined verification test to confirm that controls are effective.

RISK-01: Slips & Trips

Inherent Risk	1500 (Very High)
Key Causes	Loose cables, tools obstructing walkways, congested bench area.
Mitigations	• **Elimination:** Keep walkways clear, remove floor clutter. • **Engineering:** Use cable trays, tape down leads. • **Administrative:** End-of-session clearance checks, clear-aisle policy. • **PPE:** Closed-toe shoes.
Verification Test	Walkway audit: zero loose cables/tools; aisles $\geq$900 mm clear. Pass, no hazards found
Residual Risk	Low

RISK-02: Poor Ventilation / Asphyxiation

Inherent Risk	750 (Very High)
Key Causes	Soldering without fume extraction; extended work in enclosed areas.
Mitigations	• **Substitution:** Use low-odour, rosin-free flux. • **Engineering:** Enable local fume extraction with filters. • **Administrative:** Ventilation checks before work, time limits on solder tasks. • **PPE:** Optional respirator if extraction unavailable.
Verification Test	Fume extractor active, filter in place, ventilation checklist signed. Pass, all conditions met
Residual Risk	Low

RISK-03: Cuts / Abrasions

Inherent Risk	450 (High)
Key Causes	Sharp tools or part edges, damaged 3D prints or PCBs.
Mitigations	• **Elimination:** Deburr and smooth printed/metal edges. • **Engineering:** Use tool guards and retractable blades. • **Administrative:** Tool-use training and blade return protocol. • **PPE:** Safety glasses; cut-resistant gloves if needed.
Verification Test	All sharp edges smoothed; blades stored safely; PPE worn during cutting. Pass, all conditions met
Residual Risk	Low

RISK-04: Burns (Soldering / Hot Components)

Inherent Risk	450 (High)
Key Causes	Contact with hot soldering iron tips or overheated MOSFETs.
Mitigations	• **Engineering:** Use tip stands and heat-proof mats. • **Administrative:** Do not leave powered irons unattended. • **PPE:** Safety glasses; heat-resistant gloves if required.
Verification Test	Iron stored in stand; heat mat in use; motor driver $< 85°C$ after 5 min hover. Pass, all conditions true
Residual Risk	Low

RISK-05: Emissions (Solder / 3D-Print Fumes)

Inherent Risk	270 (Medium)
Key Causes	Flux fumes, ultrafine particle emissions from filament heating.
Mitigations	• **Substitution:** Lead-free solder and low-emission filament. • **Engineering:** Local fume extraction for soldering and printing. • **Administrative:** Store chemicals sealed; SDS available. • **PPE:** Safety glasses; nitrile gloves for solvent handling.
Verification Test	Extraction active and SDS visible; PPE worn when handling resin/solvents. Pass, all conditions true
Residual Risk	Low

2.10 Design Outputs

Stored at: GitHub repository /templink. **Includes:**

2.11 Final Costs

3 Recommendations

4 Manual

5 References

[1] S. G. Sofia Khokhlenok, *Flight controller design project*, https://github.com/koshchey/flight-controller-design-project, Accessed: 2025-10-17, 2025.

A Appendices

A.1 Constraints

Table 7: Constraints for Drone Design (Updated)

No.	Constraint	Description	Assumptions
C.1	Weight	The total weight of the drone must allow for continuous and stable flight.	The quad-mounted motors can support a maximum payload of 60 g.
C.2	Software	Only free and open-source flight controller software shall be used.	The firmware will incorporate third-party open-source code, which will be modified to suit the application (e.g., autonomy and feedback).
C.3	Operating Environment	The drone is limited to operation within a fixed indoor area containing immovable fixtures (e.g., wall-mounted displays).	–
C.4	Budget	The design and construction of the drone shall remain within the allocated project budget.	–
C.5	Network Speed	Wireless bandwidth and latency may limit the system's ability to stream live video.	Streaming will occur via The University of Western Australia (UWA) Wi-Fi or a mobile hotspot.
C.6	Safety	The drone must be safe for operation by the user and must not pose any physical risk to personnel.	–
C.7	Sensor Selection	Available sensors (such as IMU and ToF) are constrained by cost, availability, and ability to interface with the ESP32 microcontroller.	Supported interfaces include UART, I^2C, and Serial Peripheral Interface (SPI).

A.2 Additional Requirements

Table 8: Additional Requirements

No.	Requirement
R.16	The PCB shall include custom motor drivers.
R.19	The drone shall be able to fly for a minimum of 3 minutes.
R.28	The surveillance camera shall stream a live camera feed.
R.26	The User Interface (UI) shall include a battery level monitoring system for detecting low battery levels.
R.29	The surveillance camera shall record video for playback.
R.25	The UI shall include an error diagnostic and feedback system.
R.6	The hovering altitude shall be adjustable prior to operation, within a range of 0.5 to 2 metres.
A.R.2	The drone's operational capabilities shall be powered by a single battery.
A.R.4	A comprehensive operation manual and user guide shall be provided to the client for use, understanding, and further development.
A.R.6	Placement of components on the PCB shall minimise signal distortion and heat generation to ensure accurate flight paths and predictable behaviour.
A.R.7	The drone shall feature a visual (LED) or audible warning system to alert personnel during flight initiation.
A.R.8	The drone shall provide a UI alert in cases of overheating.
A.R.9	The drone shall include LEDs to indicate charging mode.
A.R.10	The drone shall feature visual or UI indicators showing when it is powering down or operating autonomously.

A.3 Code

A.3.1 sensor.c

```
//Ref: https://github.com/hibit-dev/mpu6050
//Ref: https://components.espressif.com/components/espressif/mpu6050

#include "i2c.h"
#include "stabiliser.h"
#include "sensors.h"
#include "filter.c"

#define MPU6050_SENSOR_ADDR          0x68
#define MPU6050_WHO_AM_I_REG_ADDR    0x75
#define MPU6050_ACCEL_XOUT_H_ADDR    0x3b
#define MPU6050_GYRO_XOUT_H_ADDR     0x43
#define MPU6050_PWR_MGMT_1_ADDR      0x6b
#define MPU6050_GYRO_CONFIG_ADDR     0x1b
#define MPU6050_ACCEL_CONFIG_ADDR    0x1c
#define MPU6050_INT_CFG_ADDR         0x37
#define MPU6050_INT_ENABLE_ADDR      0x38
#define MPU6050_DLPF_ADDR            0x1a
#define MPU6050_H_RESET              0x80
#define MPU6050_SLV4_CTRL_ADDR       0x34
#define MPU6050_MAST_CTRL_ADDR       0x24
#define MPU6050_SMPLRT_DIV_ADDR      0x19

#define GYRO_FULL_SCALE_250_DPS   0x00
#define GYRO_FULL_SCALE_500_DPS   0x08
```

```
#define GYRO_FULL_SCALE_1000_DPS 0x10
#define GYRO_FULL_SCALE_2000_DPS 0x18
#define ACC_FULL_SCALE_2G        0x00
#define ACC_FULL_SCALE_4G        0x08
#define ACC_FULL_SCALE_8G        0x10
#define ACC_FULL_SCALE_16G       0x18

#define GYRO_250_SENSITIVITY    (float)((2 * 250.0) / 65536.0)
#define GYRO_500_SENSITIVITY    (float)((2 * 500.0) / 65536.0)
#define GYRO_1000_SENSITIVITY   (float)((2 * 1000.0) / 65536.0)
#define GYRO_2000_SENSITIVITY   (float)((2 * 2000.0) / 65536.0)
#define ACC_2G_SENSITIVITY      (float)((2 * 2) / 65536.0)
#define ACC_4G_SENSITIVITY      (float)((2 * 4) / 65536.0)
#define ACC_8G_SENSITIVITY      (float)((2 * 8) / 65536.0)
#define ACC_16G_SENSITIVITY     (float)((2 * 16) / 65536.0)

#define GYRO_LPF_CUTOFF_FREQ 80
#define ACCE_LPF_CUTOFF_FREQ 30

/* The sensitivity must match the range: */
#define GYRO_SENSITIVITY GYRO_500_SENSITIVITY
#define ACCE_SENSITIVITY ACC_8G_SENSITIVITY
#define GYRO_CONFIG_RANGE GYRO_FULL_SCALE_500_DPS
#define ACCE_CONFIG_RANGE ACC_FULL_SCALE_8G

#define CALIBRATION_SAMPLES 200
#define INT_EN 1
bool use_bias = true;

static const char *TAG = "IMU";

QueueHandle_t acce_queue = NULL;
QueueHandle_t gyro_queue = NULL;
static TaskHandle_t sensor_task_handle = NULL;

static SemaphoreHandle_t mpu6050_data_ready;
static SemaphoreHandle_t sensor_data_ready;

static i2c_master_bus_handle_t bus_handle;
static i2c_master_dev_handle_t dev_handle;
static raw_acce_t raw_acce;
static raw_gyro_t raw_gyro;
static gyro_t gyro;
static acce_t acce;
static raw_gyro_t gyro_bias = {.x = 0, .y = 0, .z = 0};
static raw_acce_t acce_bias = {.x = 0, .y = 0, .z = 0};

static lpf_data lpf_data_acce[3];
static lpf_data lpf_data_gyro[3];

static bool sensors_init = false;

/* INIT ********************************************************************/
esp_err_t mpu6050_init(i2c_master_bus_handle_t *bus_handle, i2c_master_dev_handle_t *
    dev_handle) {
    // Intialise the I2C master and device
    i2c_master_init(bus_handle, dev_handle);
    i2c_mpu_device_init(bus_handle, dev_handle);

```

```
84      ESP_LOGI(TAG, "I2C initialized successfully");
85      vTaskDelay(pdMS_TO_TICKS(100));
86
87      // Reset the registers
88      ESP_ERROR_CHECK(mpu6050_register_write_byte(*dev_handle, MPU6050_PWR_MGMT_1_ADDR,
            MPU6050_H_RESET));
89      vTaskDelay(pdMS_TO_TICKS(100));
90
91      // Set the gyro as the reference
92      ESP_ERROR_CHECK(mpu6050_register_write_byte(*dev_handle, MPU6050_PWR_MGMT_1_ADDR, 0
            x01));
93
94      uint8_t data[2];
95      esp_err_t ret = mpu6050_register_read(*dev_handle, MPU6050_WHO_AM_I_REG_ADDR, data, 0
            x01);
96      if (ret != ESP_OK || data[0] != 0x68) return 1;
97
98      // Configure the sensor sensitivity
99      ESP_ERROR_CHECK(mpu6050_register_write_byte(*dev_handle, MPU6050_ACCEL_CONFIG_ADDR,
            ACCE_CONFIG_RANGE));
100     ESP_ERROR_CHECK(mpu6050_register_write_byte(*dev_handle, MPU6050_GYRO_CONFIG_ADDR,
            GYRO_CONFIG_RANGE));
101
102     // Enable digital low pass (reduces sample rate from 8kHz to 1kHz)
103     ESP_ERROR_CHECK(mpu6050_register_write_byte(*dev_handle, MPU6050_DLPF_ADDR, 0x02));
            // 98Hz BW
104     ESP_ERROR_CHECK(mpu6050_register_write_byte(*dev_handle, MPU6050_SMPLRT_DIV_ADDR, 0))
            ;
105
106     // Enable interrupts
107     ESP_ERROR_CHECK(mpu6050_register_write_byte(*dev_handle, MPU6050_INT_ENABLE_ADDR,
            INT_EN));
108     ESP_ERROR_CHECK(mpu6050_register_write_byte(*dev_handle, MPU6050_INT_CFG_ADDR, 0x00))
            ;
109
110     // Wake up the IMU
111     ESP_ERROR_CHECK(mpu6050_register_write_byte(*dev_handle, MPU6050_PWR_MGMT_1_ADDR, 0
            x00));
112     vTaskDelay(pdMS_TO_TICKS(100));
113
114     return ESP_OK;
115 }
116
117 /* De-initialise IMU */
118 void mpu6050_deinit(i2c_master_bus_handle_t *bus_handle, i2c_master_dev_handle_t *
        dev_handle) {
119     i2c_master_bus_rm_device(*dev_handle);
120     i2c_del_master_bus(*bus_handle);
121     ESP_LOGI(TAG, "I2C de-initialized successfully");
122 }
123
124 /* MOTION ************************************************************/
125 /* Read MPU6050 acceleration */
126 void mpu6050_read_motion_acce(i2c_master_dev_handle_t *dev_handle, raw_acce_t *raw_acce,
        acce_t *acce, lpf_data *lpf_data) {
127     // Registers (dec):
128     // [59-64] Accelerometer
129
130     uint8_t buffer[6];
```

```
131     esp_err_t ret = mpu6050_register_read(*dev_handle, MPU6050_ACCEL_XOUT_H_ADDR, buffer,
            sizeof(buffer));
132 
133     if (ret != ESP_OK) {
134         ESP_LOGE(TAG, "Failed to read from accelerometer");
135         return;
136     }
137 
138     raw_acce->x = (((int16_t) buffer[0]) << 8) | buffer[1];
139     raw_acce->y = (((int16_t) buffer[2]) << 8) | buffer[3];
140     raw_acce->z = (((int16_t) buffer[4]) << 8) | buffer[5];
141 
142     float x, y, z;
143     x = ((float)(raw_acce->x)) * ACCE_SENSITIVITY;
144     y = ((float)(raw_acce->y)) * ACCE_SENSITIVITY;
145     z = ((float)(raw_acce->z)) * ACCE_SENSITIVITY;
146 
147     acce->x = apply_lpf(&lpf_data[0], x);
148     acce->y = apply_lpf(&lpf_data[1], y);
149     acce->z = apply_lpf(&lpf_data[2], z);
150 }
151 
152 /* Read MPU6050 gyroscope */
153 void mpu6050_read_motion_gyro(i2c_master_dev_handle_t *dev_handle, raw_gyro_t *raw_gyro,
        gyro_t *gyro,
154                               raw_gyro_t *gyro_bias, lpf_data *lpf_data) {
155     // Registers (dec):
156     // [67-72] Gyroscope
157     uint8_t buffer[6];
158     esp_err_t ret = mpu6050_register_read(*dev_handle, MPU6050_GYRO_XOUT_H_ADDR, buffer,
            sizeof(buffer));
159     if (ret != ESP_OK) {
160         ESP_LOGE(TAG, "Failed to read from gyro");
161         return;
162     }
163 
164     raw_gyro->x = (((int16_t) buffer[0]) << 8) | buffer[1];
165     raw_gyro->y = (((int16_t) buffer[2]) << 8) | buffer[3];
166     raw_gyro->z = (((int16_t) buffer[4]) << 8) | buffer[5];
167 
168     float x, y, z;
169     x = ((float)(raw_gyro->x - gyro_bias->x)) * GYRO_SENSITIVITY;
170     y = ((float)(raw_gyro->y - gyro_bias->y)) * GYRO_SENSITIVITY;
171     z = ((float)(raw_gyro->z - gyro_bias->z)) * GYRO_SENSITIVITY;
172 
173     gyro->x = apply_lpf(&lpf_data[0], x);
174     gyro->y = apply_lpf(&lpf_data[1], y);
175     gyro->z = apply_lpf(&lpf_data[2], z);
176 }
177 
178 void mpu6050_read_motion_raw(i2c_master_dev_handle_t *dev_handle, raw_gyro_t *raw_gyro,
        raw_acce_t *raw_acce) {
179     uint8_t buffer[14];
180     mpu6050_register_read(*dev_handle, MPU6050_GYRO_XOUT_H_ADDR, buffer, sizeof(buffer));
181 
182     raw_gyro->x = (((int16_t) buffer[0]) << 8) | buffer[1];
183     raw_gyro->y = (((int16_t) buffer[2]) << 8) | buffer[3];
184     raw_gyro->z = (((int16_t) buffer[4]) << 8) | buffer[5];
185     raw_acce->x = (((int16_t) buffer[8]) << 8) | buffer[9];
```

```
186     raw_acce->y = (((int16_t) buffer[10]) << 8) | buffer[11];
187     raw_acce->z = (((int16_t) buffer[12]) << 8) | buffer[13];
188 }
189 
190 /* Calibration ************************************************************/
191 void mpu6050_calibrate(i2c_master_dev_handle_t *dev_handle, raw_gyro_t *raw_gyro,
        raw_acce_t *raw_acce, raw_gyro_t *gyro_bias, raw_acce_t *acce_bias) {
192     uint16_t samples = CALIBRATION_SAMPLES;
193     int64_t g_sum_x = 0, g_sum_y = 0, g_sum_z = 0, a_sum_x = 0, a_sum_y = 0, a_sum_z = 0;
194 
195     for (int i = 0; i < samples; i++) {
196         mpu6050_read_motion_raw(dev_handle, raw_gyro, raw_acce);
197         g_sum_x += raw_gyro->x;
198         g_sum_y += raw_gyro->y;
199         g_sum_z += raw_gyro->z;
200         vTaskDelay(pdMS_TO_TICKS(10));
201     }
202 
203     gyro_bias->x = g_sum_x/samples;
204     gyro_bias->y = g_sum_y/samples;
205     gyro_bias->z = g_sum_z/samples;
206 
207     ESP_LOGI(TAG, "Gyro calibrated!");
208 
209     ESP_LOGI(TAG, "Gyro Bias: Gx: %.4f Gy: %.4f Gz: %.4f",
210         (float)(gyro_bias->x) * GYRO_SENSITIVITY,
211         (float)(gyro_bias->y) * GYRO_SENSITIVITY,
212         (float)(gyro_bias->z) * GYRO_SENSITIVITY);
213 }
214 
215 /**************************************************************/
216 /* Signal when data is ready from MPU6050 */
217 void IRAM_ATTR sensorISRHandler(void *arg) {
218     // See: FreeRTOS website, xSemaphoreGiveFromISR()
219     portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
220     xSemaphoreGiveFromISR(mpu6050_data_ready, &xHigherPriorityTaskWoken);
221 
222     if (xHigherPriorityTaskWoken) {
223         portYIELD_FROM_ISR();
224     }
225 }
226 
227 /* Task ************************************************************/
228 static void sensor_task(void *arg) {
229     // ESP_LOGI(TAG, "Ready to read from mpu6050...");
230     while (1) {
231         if ((xSemaphoreTake(mpu6050_data_ready, portMAX_DELAY) == pdTRUE)) {
232         mpu6050_read_motion_acce(&dev_handle, &raw_acce, &acce, lpf_data_acce);
233         xQueueOverwrite(acce_queue, (void *)&acce);
234 
235         mpu6050_read_motion_gyro(&dev_handle, &raw_gyro, &gyro, &gyro_bias, lpf_data_gyro
                );
236         xQueueOverwrite(gyro_queue, (void *)&gyro);
237 
238         xSemaphoreGive(sensor_data_ready);
239         }
240     }
241 }
242 
```

```
void sensor_task_init(void) {
    acce_queue = xQueueCreate(1, sizeof(acce_t));
    gyro_queue = xQueueCreate(1, sizeof(gyro_t));
    mpu6050_data_ready = xSemaphoreCreateBinary();
    sensor_data_ready = xSemaphoreCreateBinary();

    esp_err_t ret = mpu6050_init(&bus_handle, &dev_handle);
    if (ret != ESP_OK) {
        ESP_LOGE(TAG, "Sensor not present or couldn't be read from");
        return;
    }
    vTaskDelay(pdMS_TO_TICKS(100));

    if (use_bias) mpu6050_calibrate(&dev_handle, &raw_gyro, &raw_acce, &gyro_bias, &
        acce_bias);

    for (uint8_t i = 0; i < 3; i++) {
        init_lpf(&lpf_data_acce[i], 1000.0, ACCE_LPF_CUTOFF_FREQ);
        init_lpf(&lpf_data_gyro[i], 1000.0, GYRO_LPF_CUTOFF_FREQ);
    }

    /* Configure interrupt pin */
    gpio_config_t io_conf = {
        .intr_type = GPIO_INTR_POSEDGE, // interrupt of rising edge
        .pin_bit_mask = (1ULL << MPU6050_INT_GPIO_PIN), // bit mask map of the pins for
            esp
        .mode = GPIO_MODE_INPUT, // set as input mode
        .pull_down_en = 0, // disable pull-down mode
        .pull_up_en = 1, // enable pull-up mode
    };

    gpio_config(&io_conf);
    gpio_install_isr_service(0);
    gpio_set_intr_type(MPU6050_INT_GPIO_PIN, GPIO_INTR_POSEDGE);
    gpio_isr_handler_add(MPU6050_INT_GPIO_PIN, sensorISRHandler, (void *)
        MPU6050_INT_GPIO_PIN);

    ESP_LOGI(TAG, "Sensor interrupt intialised");
    vTaskDelay(pdMS_TO_TICKS(100));

    /* Start the task */
    xTaskCreate(sensor_task, "sensor task", 2*1024, NULL, 1, &sensor_task_handle);
    ESP_LOGI(TAG, "Sensor task intialised");
    sensors_init = true;
}

/***************************************************************/
/* Functions to interface with other tasks */
void waitSensorData() {
    xSemaphoreTake(sensor_data_ready, portMAX_DELAY);
}

bool sensorsIsInit(void) {
    return sensors_init;
}

void readIMUData(sensor_data_t *sensor_data) {
    xQueuePeek(gyro_queue, (void *)&(sensor_data->gyro), 0);
    xQueuePeek(acce_queue, (void *)&(sensor_data->acce), 0);
```

```
299 }
300
301 /*************************************************************/
302
303 /* Read from MPU6050 register */
304 esp_err_t mpu6050_register_read(i2c_master_dev_handle_t dev_handle, uint8_t reg_addr,
        uint8_t *data, size_t len) {
305     return i2c_master_transmit_receive(dev_handle, &reg_addr, 1, data, len,
            I2C_MASTER_TIMEOUT_MS / portTICK_PERIOD_MS);
306 }
307
308 /* Write to MPU6050 register */
309 esp_err_t mpu6050_register_write_byte(i2c_master_dev_handle_t dev_handle, uint8_t
        reg_addr, uint8_t data) {
310     uint8_t write_buf[2] = {reg_addr, data};
311     return i2c_master_transmit(dev_handle, write_buf, sizeof(write_buf),
            I2C_MASTER_TIMEOUT_MS / portTICK_PERIOD_MS);
312 }
313
314 /* Initialise MPU device */
315 void i2c_mpu_device_init(i2c_master_bus_handle_t *bus_handle, i2c_master_dev_handle_t *
        dev_handle) {
316     i2c_device_config_t dev_config = {
317         .dev_addr_length = I2C_ADDR_BIT_LEN_7,
318         .device_address = MPU6050_SENSOR_ADDR,
319         .scl_speed_hz = I2C_MASTER_FREQ_HZ,
320     };
321     ESP_ERROR_CHECK(i2c_master_bus_add_device(*bus_handle, &dev_config, dev_handle));
322 }
```

Listing 1: Register Read