Joan Ignasi Cid Guardia

DESIGN OF A REAL-TIME COLLISION AVOIDANCE MODULE FOR UAVs

Grau d'Enginyeria Informàtica

Final Degree Project

Directed by

Sr. David Gamez Alari

UNIVERSITAT ROVIRA I VIRGILI

Tarragona, 2025

Acknowledgments

I would like to thank my parents, Maite and Joan, for their constant support, both moral and financial, to carry out this project.

I would also like to thank my partner, Gina, for her support at times when my work has been more complicated.

Also, to my tutor, David, for guiding me, advising me and solving my doubts when I needed it.

Resum

Aquest projecte pretén ser una introducció als sistemes de control d'aviònica, en ell es realitzarà un anàlisi de com es pot fabricar un mòdul, tant la part de hardware, com la de software, enfocant-nos amb aquesta última. Es desenvoluparà com un sistema encastat, on es processaran senyals, dades de sensors, s'aplicaran algoritmes de correcció d'errors i s'implementaran diferents algorismes de control de vol. Enfocant-se principalment en la seguretat del sistema.

Resumen

Este proyecto pretende ser una introducción a los sistemas de control de aviónica, en él se realizará un análisis de cómo se puede fabricar un módulo, tanto la parte de hardware, como la de software, enfocándonos con esta última. Se desarrollará como un sistema embebido, donde se procesarán señales, datos de sensores, se aplicarán algoritmos de corrección de errores y se implementarán diferentes algoritmos de control de vuelo. Todo ello enfocándonos en la seguridad del sistema.

Abstract

This project aims to be an introduction to avionics control systems, in which an analysis will be made of how a module can be manufactured, both the hardware and software parts, focusing on the latter. It will be developed as an embedded system, where signals and sensor data will be processed, error correction algorithms will be applied, and different flight control algorithms will be implemented. All this focusing on the safety of the system.

Index

ACKNOWLEDGMENTS 2
RESUM 3
RESUMEN 3
ABSTRACT 3
INDEX 4
INTRODUCTION 7
KEYWORDS 8
OBJECTIVES OF THE PROJECT 8
DISCLAIMER 10
PLANNING 11
PROJECT REQUIREMENTS 12
RULES AND REGULATIONS 12
FUNCTIONAL REQUIREMENTS 13
NON-FUNCTIONAL REQUIREMENTS 14
PROJECT DESIGN 15
SYSTEM ARCHITECTURE 15
USE CASE DIAGRAM 16
CLASS DIAGRAMS 23
IMPLEMENTATION 30
HARDWARE 31
SOFTWARE 36
MPU6050 ORIENTATION SENSOR 48
MAIN APP 50
FLIGHT CONTROL 54
WEB INTERFACE 59
EVALUATION OF PERSONNEL AND MATERIAL COSTS 60
PROJECT EVALUATION 63
RESOURCES 64

Figure 1 UML Diagram 11

Figure 2 System architecture diagram 15

Figure 3 Use Case Diagram visual representation 16

Figure 4Main app class diagram 23

Figure 5 Main Initialization Sequence 24

Figure 6 Rc driver class diagram 26

Figure 7 RC data communication flow 27

Figure 8 IMU driver class diagram 28

Figure 9 Distance Sensor class diagram 29

Figure 10 Component Integration Architecture 30

Figure 11 VL53L0X TOF sensor. 32

Figure 12 FOV of the Module— 32

Figure 13 MPU6050 33

Figure 14 Wiring diagram for the module. 34

Figure 15 Render of the Designed Circuit Board 34

Figure 16 Bottom side of the Module Assembled 35

Figure 17 Top side of the Module Assembled 35

Figure 18 Development Tools Overview. 36

Figure 19 RcDriver update function. 39

Figure 20 Example RcDriver usage on main application 39

Figure 21 RcDriverStrategy data processing. 41

Figure 22 Incoming Data Process Flow 42

Figure 23 iBus data frame snippet. 43

Figure 24 Decode function snippet. 44

Figure 25 Encode function snippet 44

Figure 26 Initialization Sequence for VL53L0X 45

Figure 27 Orientation data gathering process. 48

Figure 28 Heigh simulation result. 55

Figure 29 Throttle simulation result. 55

Figure 30 Simulation result of the collision avoidance mechanism. 57

Figure 31 Simulation result of the distance monitoring 57

Figure 32Simulation result of the command evolution 58

Figure 33 Simulation result of teh Position-Velocit Phase. 58

Figure 34 Dashboard with 2 working sensors   Figure 35 Dashboard with 4 working sensors 59

Introduction

This project is about implementing a flight control system for an FPV drone. These drones typically lack a flight computer; they are controlled entirely by the operator, and even the more advanced models only include basic systems for positioning or altitude control. Therefore, a module will be developed to provide this device with computing capabilities.

First, we need to understand how to interact with the drone, how it is controlled, what data is sent, how frequently, and in what format it is transmitted. Depending on the communication protocol, we will choose the appropriate development board to meet the minimum requirements for this type of application.

When the study of communications is complete, we must analyse how the drone moves through the air, what allows it to gain altitude and navigate through the environment. Based on this understanding, we will proceed to design the algorithms needed for altitude control, automatic landing, autonomous navigation, and obstacle avoidance.

To perform these control tasks, several sensors will be required. These will be discussed in detail. We will determine what needs to be measured and define the necessary specifications in terms of accuracy and response time. From there, we will select the sensor models that best fulfil these requirements.

Once the hardware study is finished, the software will be designed. This will allow critical decisions to be made in real time, where each one of them will be backed up with a security check. It is going to be selected in which software it is going to be used. The selection will be based on the system's requirements for responsiveness, reliability, and scalability. A real-time operating system (RTOS) may be chosen to ensure deterministic behaviour, enabling concurrent task management, precise timing control, and improved system robustness. The integration between the hardware and software layers will be carefully tested to guarantee seamless operation and adherence to safety standards.

The structure of the tasks will be defined, including their execution intervals, order, and the handling of potential errors, as well as data input and output. In parallel, a class and action design will be developed to ensure proper architectural organization, improving code readability, maintainability, and scalability.

After the structural design, the algorithms necessary to perform the obstacle avoidance or height control behaviours at ground level will be developed.

Finally, a legislative analysis of the project will be carried out.

Keywords

UAV	Unmanned Aerial Vehicle
FPV	First Person View
I2C	Inter-Integrated Circuit
ToF	Time of Flight
PCB	Printed Circuit Board
GPS	Global Positioning System
IMU	Inertial Measurement Unit
PWM	Pulse Width Modulation
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
UART	Universal Asynchronous Receiver/Transmitter
SPI	Serial Peripheral Interface
CPU	Central Processing Unit
RAM	Random Access Memory
ROM	Read Only Memory
GPIO	General Purpose Input/Output
BLE	Bluetooth Low Energy
MCU	Microcontroller Unit
LIDAR	Light Detection and Ranging
ISR	Interrupt Service Routine

Objectives of the project

The objective of this project is to create a module to provide avionics to an aerial vehicle that does not have them. Focusing more on the software section that corresponds to us as computer engineers.

Given that the field of research is not electronics, we will use generic components, both for the MCU and the required sensors. This does not detract from the fact that we will perform the analysis and design of the physical circuit board.

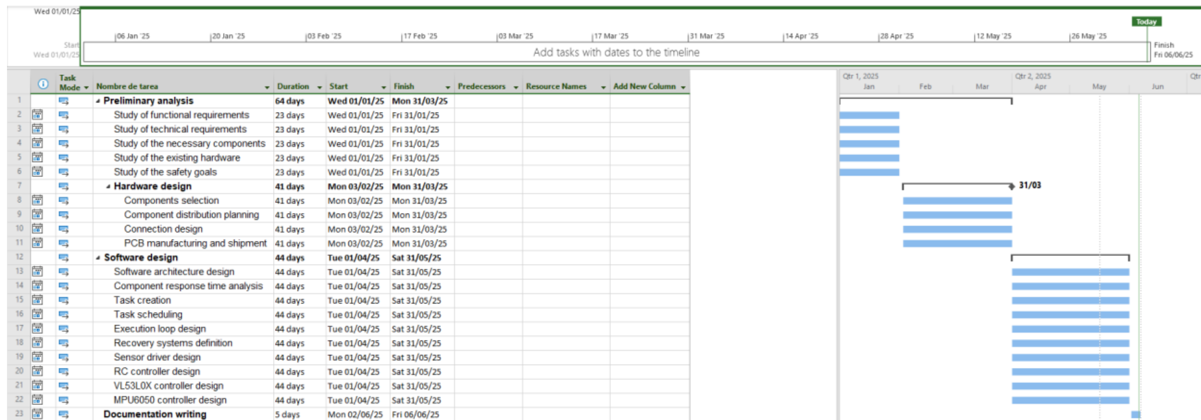
During the development of the project, topics that have been taught during the course of the computer science degree, such as memory management, interrupts, task planning, design of the software architecture and structure, and the diagrams necessary for its maintenance and extension, will be covered.

Disclaimer

All designs presented in this project are for educational and experimental purposes only. Whereas all efforts have been made to ensure accuracy and safety, the author and all respective institutions shall not be held liable for damages, injuries, or losses arising in the physical building, modifying, or use of FPV drones bearing avionics modules described in this document. The readers and practitioners are urged to obey national and local rules governing unmanned aerial vehicles, perform a thorough risk assessment, and wear proper personal protective equipment when working with electronic components and performing flight tests.

This project is not a professional certificate in avionics, aeronautics, or electrical engineering. Any practical implementation and construction shall be carried out only by people with relevant experience and wherever required, under the direct supervision of duly competent people. The author disclaims all responsibility for any and all unauthorized or negligent uses of the information herein.

Planning



	Nombre de tarea	Duration	Start	Finish	Predecessors	Resource Names	Add New Column
1	**Preliminary analysis**	**64 days**	**Wed 01/01/25**	**Mon 31/03/25**			
2	Study of functional requirements	23 days	Wed 01/01/25	Fri 31/01/25			
3	Study of technical requirements	23 days	Wed 01/01/25	Fri 31/01/25			
4	Study of the necessary components	23 days	Wed 01/01/25	Fri 31/01/25			
5	Study of the existing hardware	23 days	Wed 01/01/25	Fri 31/01/25			
6	Study of the safety goals	23 days	Wed 01/01/25	Fri 31/01/25			
7	**Hardware design**	**41 days**	**Mon 03/02/25**	**Mon 31/03/25**			
8	Components selection	41 days	Mon 03/02/25	Mon 31/03/25			
9	Component distribution planning	41 days	Mon 03/02/25	Mon 31/03/25			
10	Connection design	41 days	Mon 03/02/25	Mon 31/03/25			
11	PCB manufacturing and shipment	41 days	Mon 03/02/25	Mon 31/03/25			
12	**Software design**	**44 days**	**Tue 01/04/25**	**Sat 31/05/25**			
13	Software architecture design	44 days	Tue 01/04/25	Sat 31/05/25			
14	Component response time analysis	44 days	Tue 01/04/25	Sat 31/05/25			
15	Task creation	44 days	Tue 01/04/25	Sat 31/05/25			
16	Task scheduling	44 days	Tue 01/04/25	Sat 31/05/25			
17	Execution loop design	44 days	Tue 01/04/25	Sat 31/05/25			
18	Recovery systems definition	44 days	Tue 01/04/25	Sat 31/05/25			
19	Sensor driver design	44 days	Tue 01/04/25	Sat 31/05/25			
20	RC controller design	44 days	Tue 01/04/25	Sat 31/05/25			
21	VL53L0X controller design	44 days	Tue 01/04/25	Sat 31/05/25			
22	MPU6050 controller design	44 days	Tue 01/04/25	Sat 31/05/25			
23	**Documentation writing**	5 days	Mon 02/06/25	Fri 06/06/25			

Figure 1 UML Diagram

This project has been developed together with other subjects of the computer engineering degree, that is why there is no detailed planning. But four phases can be defined, the preliminary study, the development of the hardware, the development of the software and finally the writing of this document. The whole task has been carried out by the same person, which is why he/she has been alternating between the different sections of each of the phases.

Project Requirements

Rules and Regulations

01. Compliance with Aviation Standards
This module and its integration must comply with local and international UAV regulations.

02. Weight and Payload Limits
The combined weight of the drone and this module must not exceed the manufacturer's specified maximum take-off weight.

03. Power Consumption Limits
The implementation of this module must operate within the UAV's power budget, without causing power losses, power distribution problems to it or to the UAV itself.

04. Secure Mounting and Installation
All components must be securely mounted and tightly mounted by using approved materials and methods to withstand vibration, shock, and environmental conditions during flight.

05. Flight Testing Protocols
All flight tests must be conducted in approved areas, following pre-flight checklists, safety briefings, and risk assessments.

06. Maintenance and Inspections
Before and after each flight, and inspection must be conducted to this module, to detect wear, damage or malfunctions.

07. Pilot and Operator Certification
All operators and pilots involved in testing and operation of the system must hold the necessary licenses or certifications as required by aviation authorities.

08. Incident Reporting
Any system failure, unexpected behaviours or safety incident during operation must be documented and reported to the corresponding project manager immediately.

Functional Requirements

1. **RC Signal Interception and Processing:**
   The system must intercept RC commands from a standard RC receiver using any desired protocol, process them, and forward potentially modified commands to the flight controller. This protocol will be configured by the user following the specific interface for it.
2. **Multi-Directional Obstacle Detection:**
   The system must support an array of distance sensors to be capable of detecting obstacles in its surroundings, these must cover the 360 degrees that surround it, and must be categorised with the names: LEFT, FRONT, RIGHT, BACK, BOTTOM.
   The system must operate sensors in interrupt-driven mode with configurable distance thresholds for obstacle detection.

3. **Real-Time Multi-Tasking Operation:**
   The system must implement a real-time scheduler with priorities for managing deterministically all the tasks. None of the tasks shall block the flow of execution.

4. **Sensor Reliability and Error Recovery:**
   The system must include a mechanism to monitor the responsiveness and to address it in the event of any problem.

5. **Signal Loss Detection and Handling:**
   The system must detect RC signal loss conditions when no packets are received for more than the specified time and provide appropriate warnings and safety responses.

6. **Collision Avoidance Command Modification:**
   The system must provide framework for modifying RC commands when obstacles are detected, with the capability to alter throttle, yaw, pitch, and roll values before forwarding to the flight controller.

7. **Web-Based Real-Time Monitoring:**
   The system shall provide a web dashboard accessible through a Wi-Fi access point for real-time monitoring of system information.

Non-functional requirements

1. **Performance requirements:**
   The system implements strict real-time performance requirements with a multi-priority real-time architecture.

2. **Reliability Requirements:**
   The system includes comprehensive fault tolerance and recovery mechanisms:
   i. Sensor Watchdog.
   ii. Automatic recovery.
   iii. Signal Loss Detection.
   iv. Thread Safety.
   v. Error Handling.

3. **Scalability Requirements:**
   The architecture supports modular expansion such as:
   - Dynamic Sensor configuration: Using a constant to declare how many sensors are on the systems, then tagging them for usage.

   - Configurable Sensor Positions: Each sensor will be assigned a logical label corresponding to its spatial role, allowing the MCU to make decisions based on positional context rather than specific sensor models. Enabling modularity and simplifying integration of different sensor types without altering the decision-making logic.

   - Configurable RC Strategy: The user is given the possibility to create his own RC signal translation model, he will have to convert all the parameters needed to fill in the fields in which the modules work.

   - Multi-Core Task Distribution: Tasks can be distributed among the different cores on the implemented board to make better use of the board's resources.

4. **Maintainability Requirements**
   The code will emphasize code quality and debugging support by using:
   - Conditional Debug Compilation: Preprocessor flags for selective debug output.
   - Comprehensive Error Messages: Detailed error reporting with specific sensor identification and failure codes.
   - Clear Code Structure: Well-defined interfaces with separation of concerns between sensor management, communication, and control.
   - Self-Monitoring Capabilities: Built-in diagnostic features including sensor status reporting and performance metrics.

Project design

The abstraction of the intended behaviours consists of receiving the signal from the remote control, decode the signal, receiving the information from the sensors, applying corrections to the control signal, encode it and finally sending it to the FC.

It will be developed as an embedded system for Unmanned Aerial Vehicles (UAVs) to detect and avoid obstacles during flight, among other programable flight behaviours. This system will act as an intermediary between an RC receiver and a flight controller, capable in real time, of modifying the control signals to accomplish its stablished in-flight functionality.

System Architecture

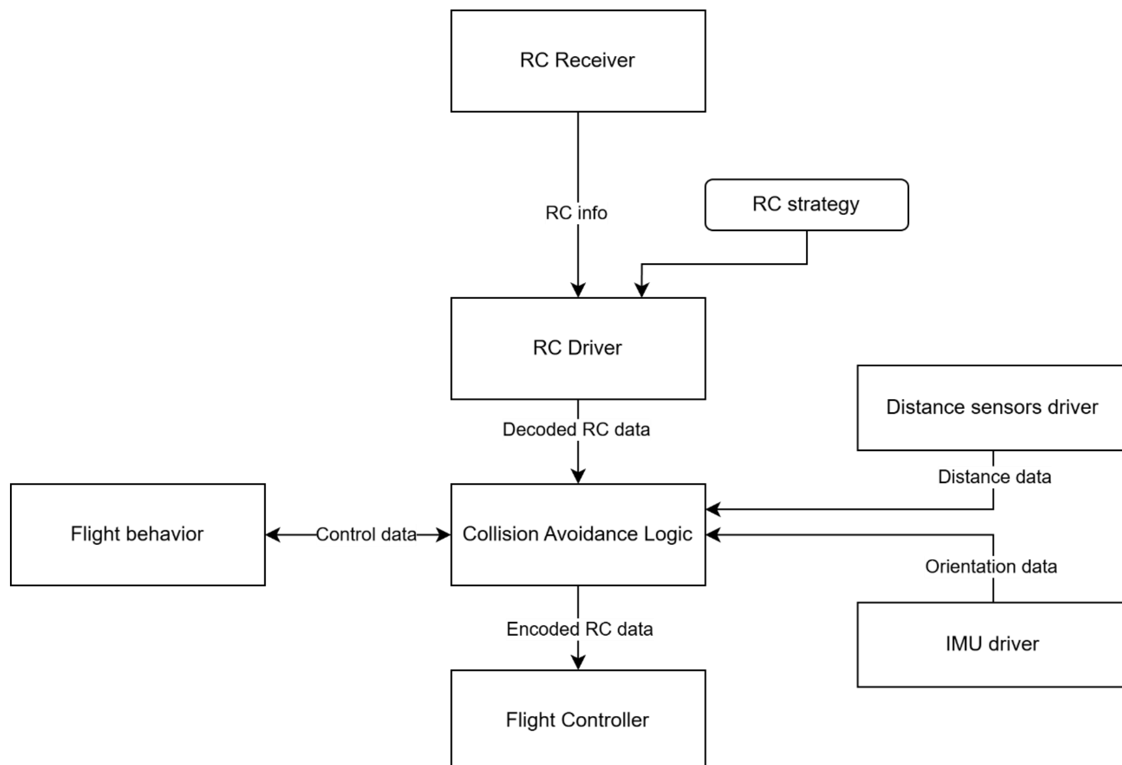


Figure 2 System architecture diagram

The architecture design is based on the segmentation of complex tasks. Each part will perform a specific set of operations, returning only an error code and a data structure that will contain a timestamp.

The main logic will process this information, making decisions based on the state of the system, the data obtained and the established operational mode.

Since it is a real time system and therefore the processes must be deterministic, its behaviour will be controlled according to the time stamp of the incoming data; and may be processed differently or even be rejected causing the change of operating mode in case of non-compliance.

The same applies for the flight behaviour system, in case a decision is not made within a specified time, the controller will discard this flight mode, consider it to be unsafe and switch to the default mode.

At all times of execution, control of what is happening in the system must be taken, therefore an error control system has been created to unify the handling of possible unforeseen events that occur during flight time. This will be explained in more detail in the implementation section.

Use case diagram

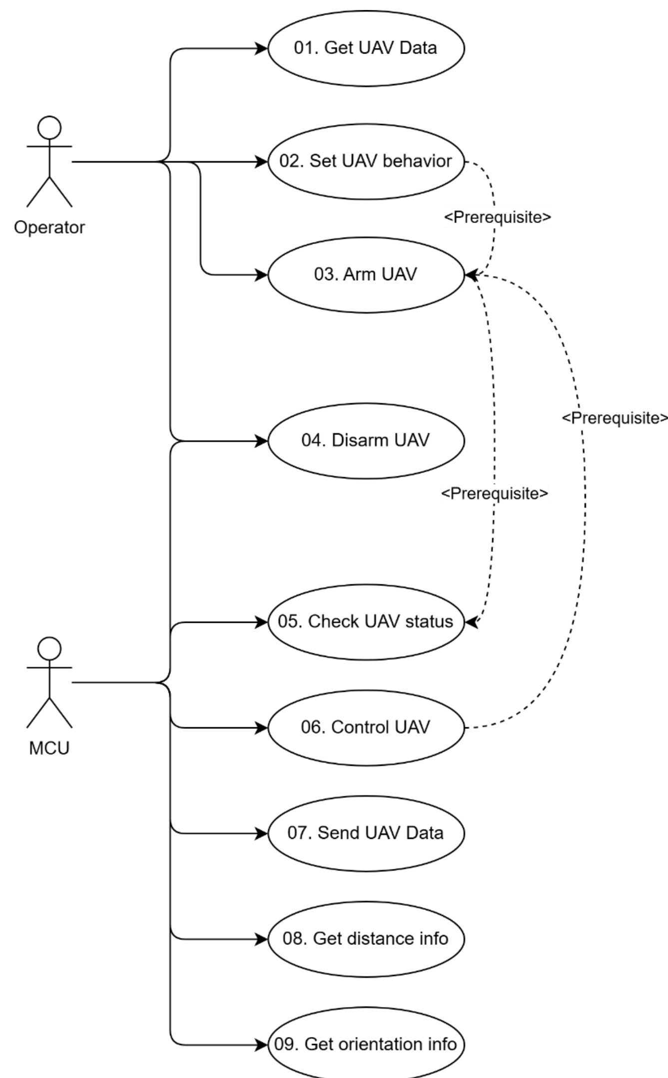


Figure 3 Use Case Diagram visual representation

Figure 3 shows the use case diagram. Two actors are differentiated, the Operator and the MCU. The operator, in other words, is the pilot of the aircraft, the one who will control it remotely.

The Operator is solely authorized to arm the UAV, meaning they have exclusive control over enabling the main propulsion system. Conversely, both the Operator and Supervisor are permitted to disarm the UAV—i.e., deactivate the propulsion—for safety-critical interventions. Disarming is treated as the final operational safeguard to prevent unintended behaviour.

Textual specification of use cases diagram

UC 01. Get UAV Data

Functionality brief: The operator will receive the information from the UAV to its mobile device.

Input parameters: Distance info, RC info, IMU info.

Output parameters: None

Actors: Operator

Precondition: None

Postconditioning: None

Nominal main process:

01. Json reception.
02. Json parsing.
03. Data formatting.
04. Data display.

UC 02. Set UAV behaviour

Functionality brief: The operator will set any of the available operating modes of the UAV.

Input parameters: Available operating mode

Output parameters: Operating mode

Actors: Operator

Precondition: Drone is armed

Postconditioning: None

Nominal main process:

01. The system displays the operating modes.
02. The Operator selects the operating mode.
03. The system checks if it is possible to select the operating mode.
04. The system returns OK.
    a. The system returns NOK.
    b. The system does not set the new operating mode.
05. The system sets the operating mode to the selected one.

UC 03. Arm UAV

Functionality brief: The operator activates the main motor of the UAV.

Input parameters: None

Output parameters: None

Actors: Operator

Precondition: Pre-checking is done successfully.

Postconditioning: None

Nominal main process:

01. The Operator sends the Arm command.
02. The system does the 06. Check UAV status.
03. The system returns OK
    a. The system returns NOK
    b. The drones does not arm
04. The drone arms

UC 04. Disarm UAV

Functionality brief: The operator or MCU deactivates the main motor.

Input parameters: None

Output parameters: None

Actors: Operator, MCU

Precondition: None

Postconditioning: None

Nominal main process:

01. Operator or MCU sends the disarm command.
02. UAV shut its main motor.

UC 05. Check UAV status

Functionality brief: The MCU does a pre-flight checking.

Input parameters: Distance info, RC info, IMU info.

Output parameters: Status.

Actors: MCU

Precondition: None

Postconditioning: Correct preflight check.

Nominal main process:

01. MCU checks RC driver status.
02. RC driver status is OK.
    a. RC driver status is NOK.
    b. MCU returns NOK.
03. MCU checks IMU driver status.
04. IMU driver is OK.
    a. IMU driver is NOK.
    b. MCU sets IMU error flag to the corresponding error.
05. MCU checks distance driver status.
06. Distance driver is OK.
    a. Distance driver is NOK.
    b. MCU sets IMU error flag to the corresponding error.
07. MCU returns Status flag

UC 06. Control UAV

Functionality brief: The MCU modifies the RC information as its operating pattern specifies.

Input parameters: Distance info, RC info, IMU info.

Output parameters: RC info.

Actors: MCU

Precondition: None

Postconditioning: None.

Nominal main process:

01. The MCU receives all input parameters.
02. The MCU identifies the currently active operating mode.
03. The MCU evaluates the input data from all the controllers.
04. The system determines the necessary adjustments to the RC data.
05. The system modifies the RC data.
06. The system encodes the RC data according to its driver.
07. The system sends the RC data through the bus.

UC 07. Send UAV Data

Functionality brief: The MCU parses and send to its server the data from the UAV.

Input parameters: Distance info, RC info, IMU info.

Output parameters: UAV data.

Actors: MCU

Precondition: None

Postconditioning: None.

Nominal main process:

01. The MCU receives all input parameters.
02. The MCU encodes the information in JSON format.
03. The MCU posts this encoded data.

UC 08. Get distance info

Functionality brief: The MCU gets the distance info from the distance driver.

Input parameters: Distance info.

Output parameters: Distance info.

Actors: MCU

Precondition: None

Postconditioning: None.

Nominal main process:

01. The MCU asks the distance driver for the distance sensor information.
02. The MCU gets the new information.
    a. The MCU detects an error from the distance sensor information
    b. The MCU activates the corresponding flag for errors.
03. The MCU propagates the new information globally.

UC 09. Get orientation info

Functionality brief: The MCU gets the orientation info from the IMU driver.

Input parameters: IMU info.

Output parameters: IMU info.

Actors: MCU

Precondition: None

Postconditioning: None.

Nominal main process:

01. The MCU asks the IMU driver for the orientation sensor information.
02. The MCU gets the new information.
    a. The MCU detects an error from the IMU sensor information
    b. The MCU activates the corresponding flag for errors.
03. The MCU propagates the new information globally.

Class diagrams

Main App class diagram

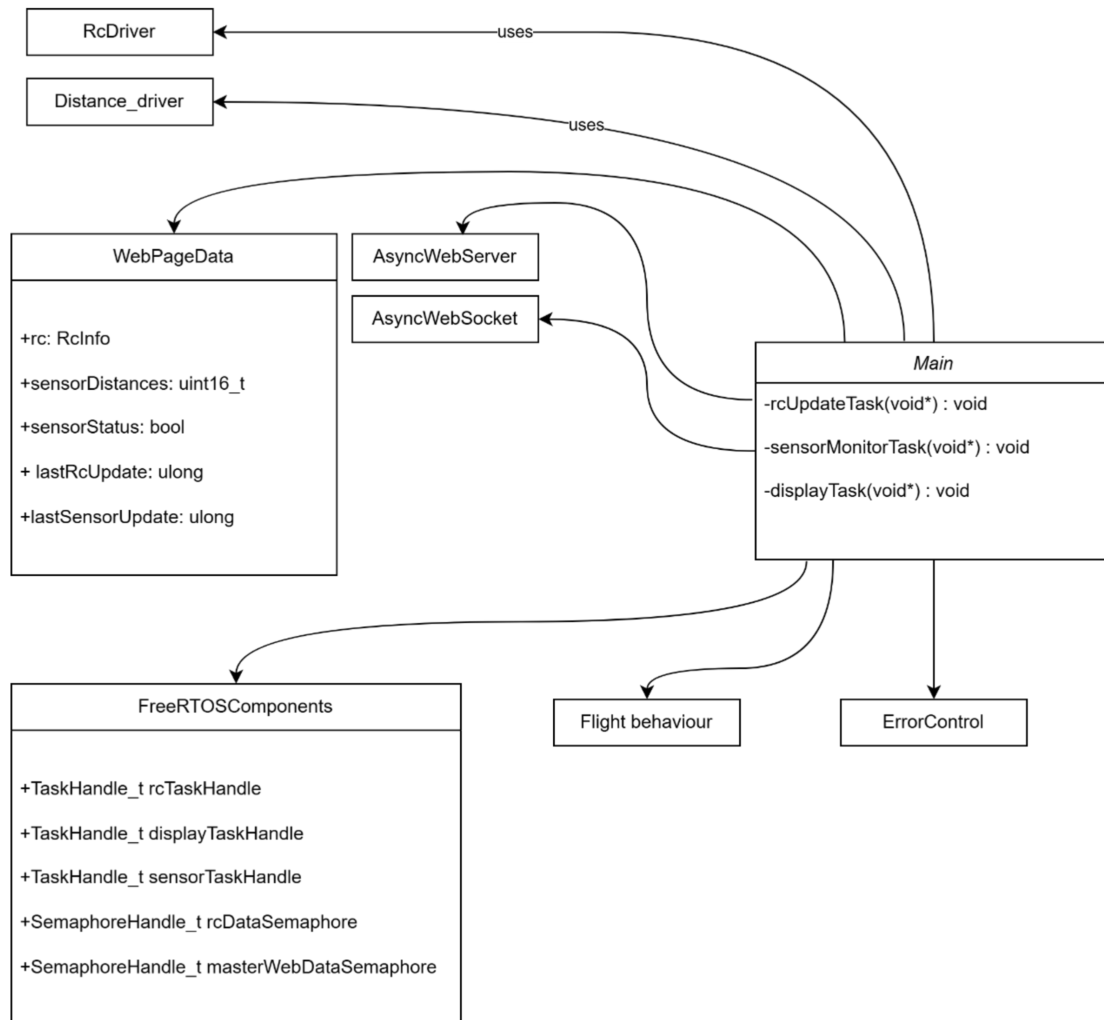


Figure 4Main app class diagram

The main class will orchestrate the entire system using concurrent multi-tasking, managing several critical subsystems in parallel. These include RC communications, which handle bidirectional signal exchange between the remote controller and onboard systems; sensor monitoring for both distance (e.g., ultrasonic, lidar) and orientation (e.g., IMU, gyroscope, accelerometer) to ensure precise spatial awareness; a web interface providing real-time system status, telemetry, and control access over a network; an error control and logging subsystem responsible for detecting, categorizing, and persistently recording system faults or anomalies; and finally, the real-time flight control loop, which continuously adjusts the required systems to maintain stable and responsive flight. The main class ensures that these tasks run simultaneously without resource contention, using real-time scheduling, to maintain low-latency, high-reliability operation on all critical paths.

The initialization sequence is represented and structured as:

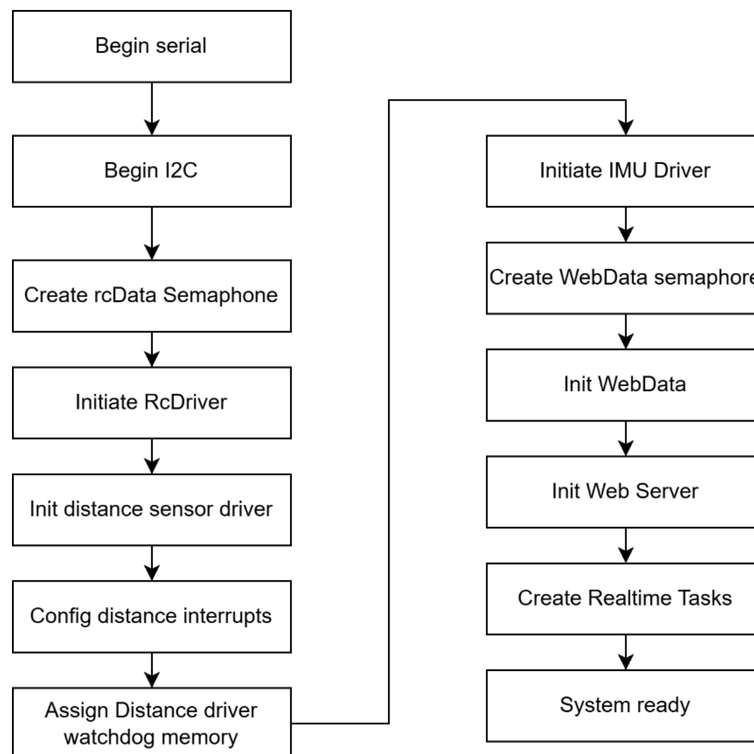


Figure 5 Main Initialization Sequence

1. Initialization of the serial communication:
   The system initializes the serial communication bus using a predefined baud rate (to be specified in the implementation section), enabling data exchange between the MCU and the host computer. This setup is essential for telemetry, diagnostics, and control commands.

2. I2C bus initialization:
   The system initializes the I2C bus for communication with the other sensors.

3. Creation of Synchronization Semaphores:
   A new semaphore mutex is created to protect the remote-control data access.

4. RC Driver initialization:
   The remote-control manager driver is configured and initialised with the specified strategy.

5. Initialisation of Distance Sensors:
   The entire distance monitoring system is initialised, and its status is checked.

6. Sensor Interrupt Configuration:
   The interrupts to be called by each of the sensors are configured and the threshold at which they are to be activated is specified.

7. Allocate working memory for the watchdog of the distance sensors:
   The system allocates dedicated working memory for the watchdog mechanism monitoring the distance sensors. This memory is used to track sensor update timestamps and timeout thresholds, ensuring timely data acquisition and detecting sensor failures or communication loss.

8. Initiate IMU Driver:
   Initialise the orientation sensor controller, together with the mode of operation and the interrupt to call.

9. Create Web Data semaphore:
   In order to provide the data visualisation server with the necessary information without affecting the correct performance of the system, an access control system to the shared data has to be created, therefore this semaphore is initialised.

10. Initiate Web Data:
    Web Data will be the data structure used to display the drone information.

11. Initiate Web Server:
    The Web Server system is initiated with the corresponding credentials.

12. Create Realtime tasks:
    The Realtime tasks are created, and the system starts its normal functionality.

The initialization sequence is strictly ordered because each subsystem has upstream dependencies that must be satisfied to ensure deterministic startup behaviour. Semaphores are created prior to the spawning of any tasks or threads that reference them, guaranteeing that synchronization primitives are fully established before concurrency is introduced. Hardware peripherals (e.g., timers, communication interfaces, sensors) are initialized and verified before enabling associated ISRs, since premature activation of interrupts on uninitialized hardware can result in undefined states, spurious interrupts, or data corruption. The system employs a fail-fast strategy: if any initialization step — whether semaphore creation, hardware configuration, or system resource allocation — returns an error or enters an invalid state, the main thread halts execution by entering an infinite loop, effectively freezing the system. This immediate termination approach prevents partial system activation, mitigates cascading failures, and simplifies fault diagnosis by making the failure point explicit and reproducible.

Rc_driver class diagram

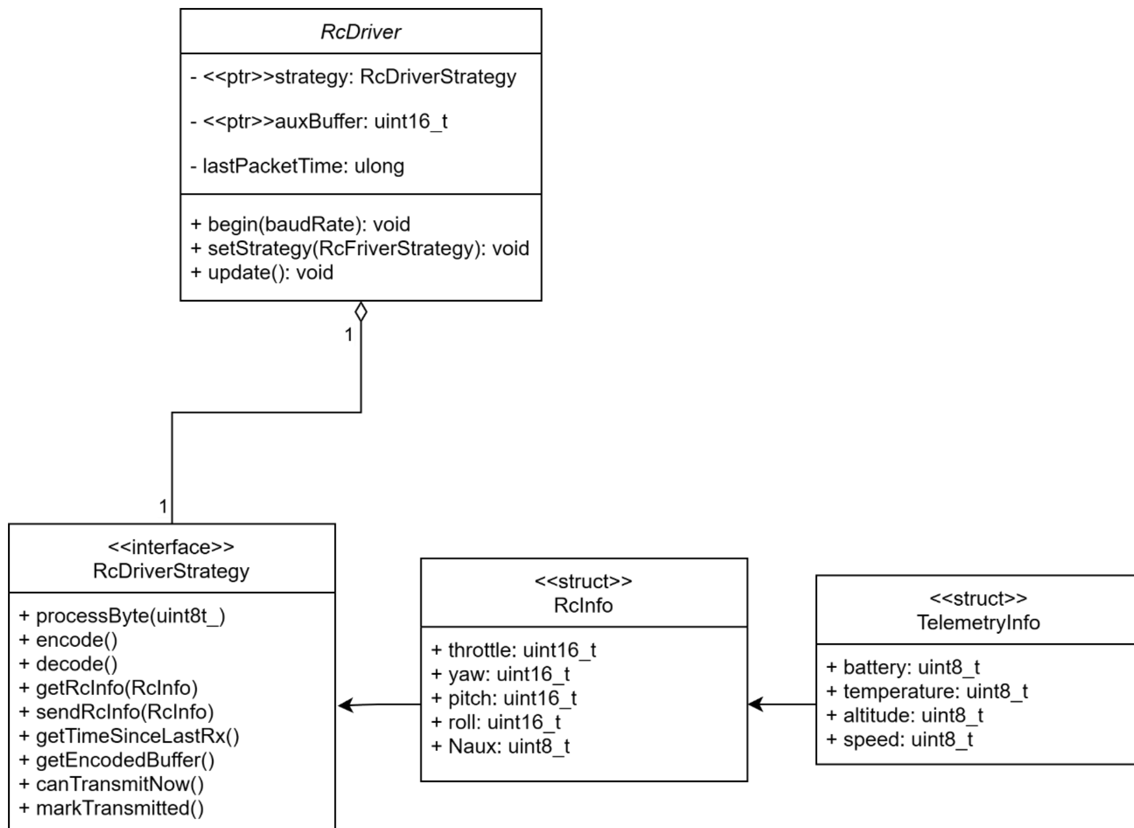


Figure 6 Rc driver class diagram

The RC Communication System is a core component of the UAV collision avoidance module, responsible for receiving, processing, and potentially modifying remote control commands. This subsystem enables the collision avoidance logic to interface with the RC controller inputs and provides a framework for implementing different RC protocols.

It handles the bidirectional flow of control data between the Rc receiver and the flight controller, with the ability to intercept and modify commands when necessary. This system implements a strategy pattern to support different RC protocols, currently focusing on the iBus protocol due to hardware availability.

The remarkable methods of the class are:

- update: sends the next byte on the RX2 queue to the RcDriverStrategy to be processed.
- getRcInfo: Retrieves the latest Rc data.
- sendRcInfo: Sends the data to the flight controller through the TX2 pin.

The RcInfo struct is designed to provide with the minimum required data to let an FPV drone fly. Thus, consisting of the throttle, yaw, pitch and roll. Then, the auxiliar channels, customized within all different protocols and radio controller models, and the timestamp.

The communication flow can be represented as:

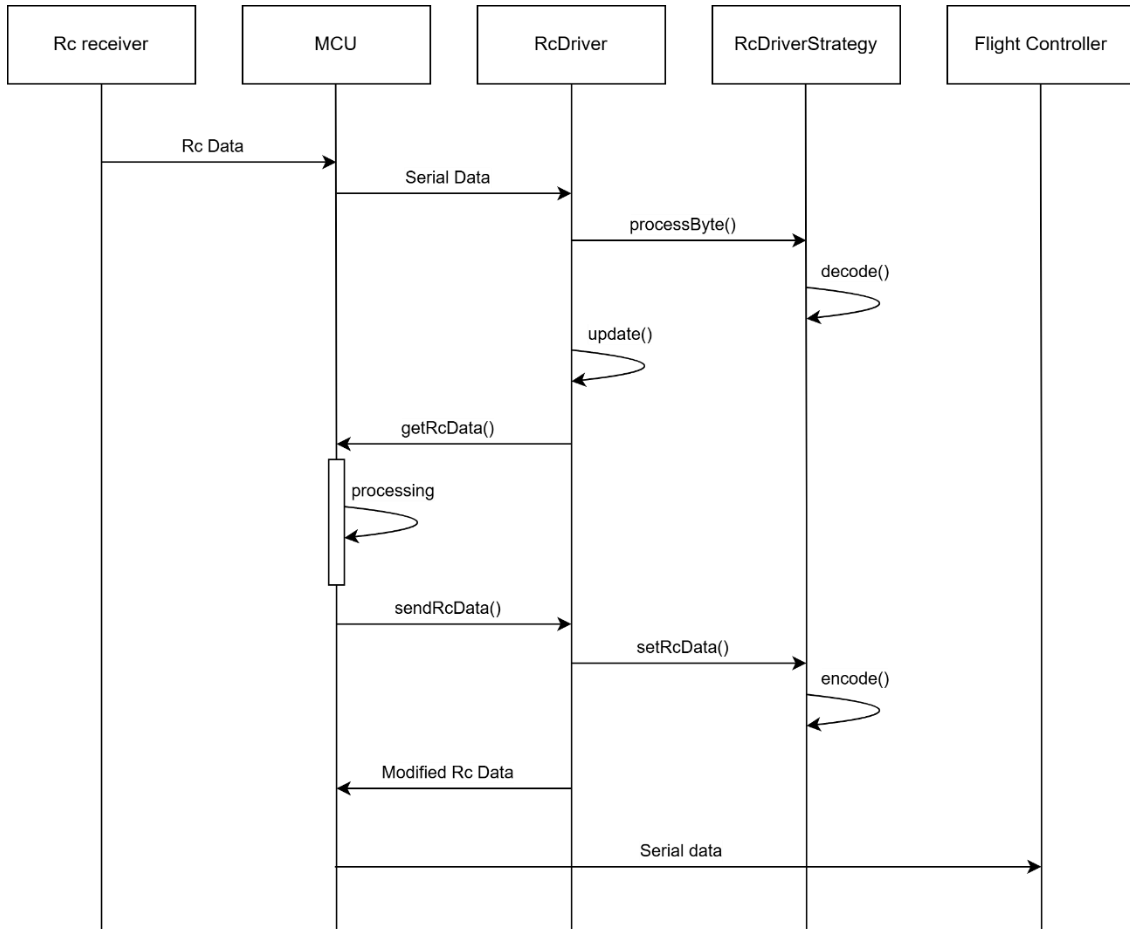


Figure 7 RC data communication flow

IMU driver class diagram

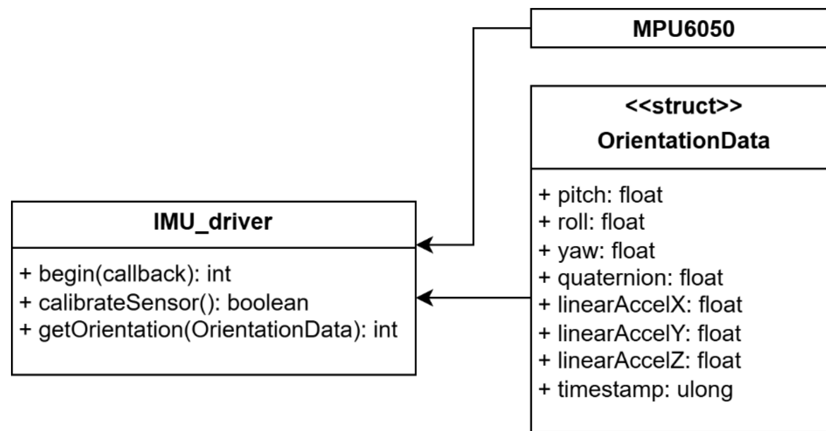


Figure 8 IMU driver class diagram

To obtain the orientation data, a driver has been designed to control all the necessary data and to manage the state of the gyroscope. For reasons of limited size that will be detailed later, only one orientation sensor has been implemented, there is no redundancy within the module; although we do have the UAV flight controller IMU, which we do not have access to, but that will keep the drone in flight in case of failure of this one.

A system of interruptions is implemented for the sensor, with the same one indicating when a new measurement is available. This allows the use of different types of IMU sensors with different response times. The MPU6050 sensor has been chosen for the tests.

Distance_Sensor_driver class diagram

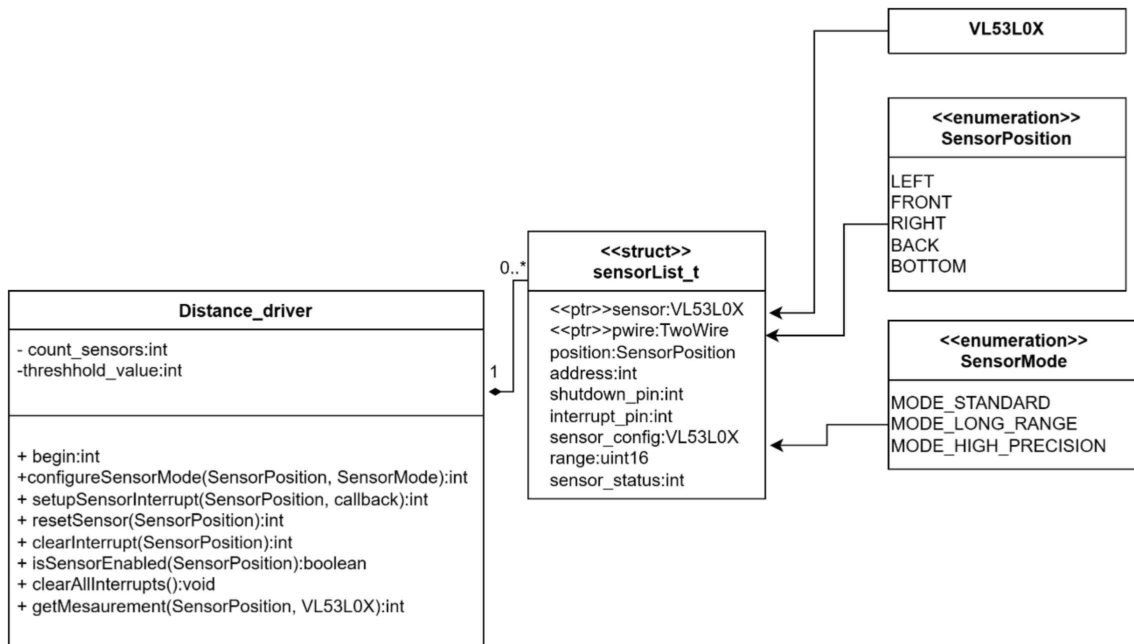


Figure 9 Distance Sensor class diagram

Continuing with the modular architecture, it will be the user himself who will finish the implementation of the sensors, selection of the model and quantity needed. For this reason, the facade pattern is implemented on the Distance_driver class, providing an encapsulation of the sensor control, allowing the main program some simple function calls to control the whole distance monitoring system. For testing purposes, I have implemented five VL53L0X distance sensors.

The controller will have to configure the sensors to ensure concurrent and error-free operation of the entire system. In addition, given the temporal constraints, we will implement interruptions for taking measurements, so the driver will not wait for the sensor to provide a new reading, instead it will be the sensor itself that signals that a new measurement is ready. How this has been implemented will be explained in more detail in the implementation section.

Implementation

Although not the core objective of this project, a physical prototype has been developed to support the validation of the software. For this purpose, a specific test module has been created with commercially available components, which allowed for a realistic simulation of real behaviour. This strategy made it possible to verify the performance of the system in practice and to detect possible integration problems at an early stage.

As for the software, PlatformIO was chosen as the development framework, integrated in the Visual Studio Code IDE. This has been chosen for its flexibility, cross-platform compatibility and efficient management of hardware-specific dependencies and build configurations. To streamline the processing of the various devices integrated in the physical module, libraries provided by the manufacturers were used.

In this section, we will start by explaining the hardware development. Introducing it first will provide essential context for understanding the logic behind various software-related decisions that were made later in this project.

By clarifying beforehand the limitations of the board, the chosen components and the system architecture, a clearer idea of how these factors influenced the structure, functionality and implementation details of the software will result.

All hardware components are implemented by their specific driver.

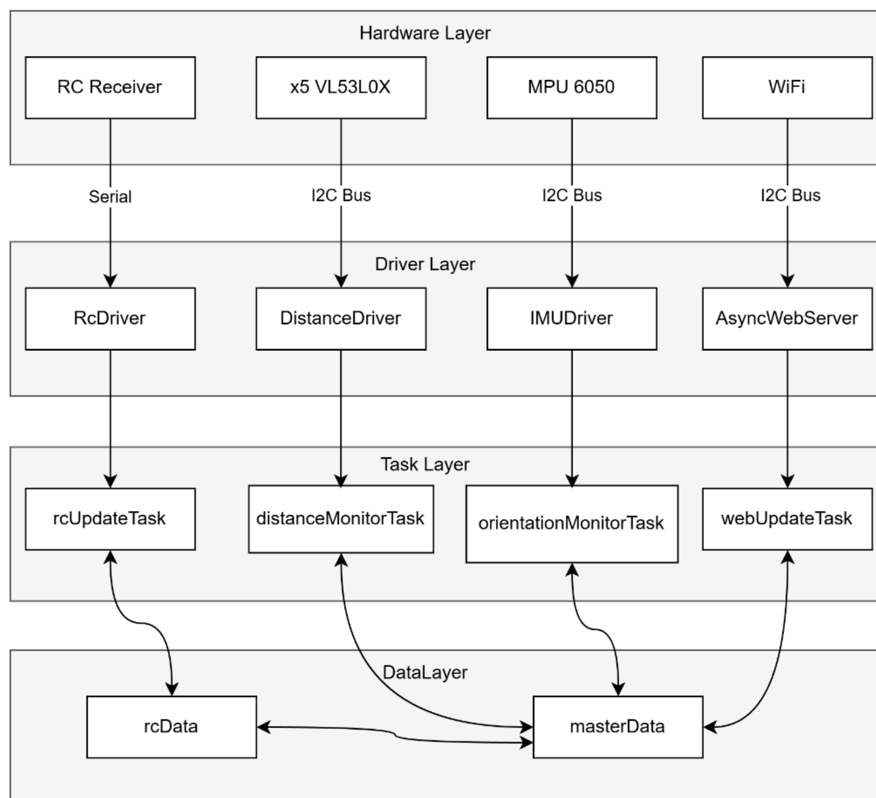


Figure 10 Component Integration Architecture

Hardware

The aircraft in which the module will be integrated is an FPV drone, which has its own flight stabilisation system. It is controlled by a remote control that communicates with the receiver built into the drone and sends a signal in iBus format to the flight controller board. This flight controller board is designed solely for the tasks of signal processing and stabilisation of the aircraft in flight. We will take advantage of the fact that the drone itself is already stabilised, so our objective is to intercept the aircraft operator's commands and modify them based on the inputs from the sensors and the flight operating mode.

Physical module

After an extensive comparison between three major chips, the Arduino Uno, the ESP32 and the STM32 Nucleo, it was decided to use the ESP32. This is the comparison:

Feature	Arduino Uno	ESP32	STM32
Price	3€ to 5€	5€ to 10€	5€ to 20€
Speed	16 MHz	Up to 240 MHz	32 to 480 MHz
Flash Memory	32 KB	Up to 4 MB	64 KB
RAM	2 KB SRAM	~520 KB RSAM	8 KB SRAM
I/O Pins	14 digital + 6 analog	30 GPIO	~16 GPIO
Size	68.6 mm x 53.4 mm	52 mm x 30 mm	68.6 mm x 53.4 mm

The memory characteristics, price and I/O pins make the ESP32 an ideal candidate for the project. In addition to having high processor frequencies, it has dual cores, which will allow us to separate critical tasks from common ones.

Distance Sensors

In order to meet the requirements set out above in relation to how the system is to perform, the VL53L0X distance sensors have been selected. This model is within the working voltage range of the ESP32, has an I2C interface and an interrupt control pin.

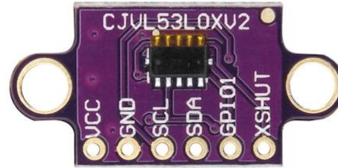


Figure 11 VL53L0X TOF sensor.

The VL53L0X offers distance monitoring in the range of 3cm to 2 metres, with millimetre accuracy and 3% of deviation and 25 degrees field of vision. Its small size will help to reduce the overall module footprint.

As indicated in the requirements, 5 of them will be used, FORWARD, REAR, LEFT, RIGHT and DOWN, thus providing peripheral distance monitoring.

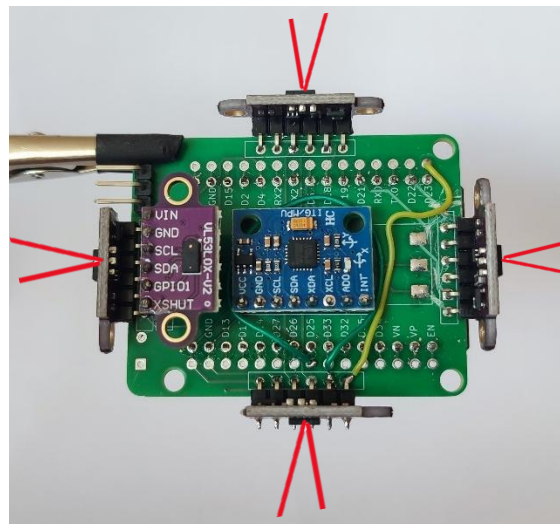


Figure 12 FOV of the Module

IMU

To detect orientation, the MPU6050 sensor will be used, which is a combination of accelerometer and 3-axis gyroscope. It works at a range of up to 16g and 2000 degrees per second, with 16-bit resolution and interrupt output.

Figure 13 MPU6050

This model allows a quick and accurate reading of the orientation that can be obtained with the accelerometer and gyroscope using Madgwick filters.

RC receiver

The receiver used will be the FS Flysky X6B, it is the receiver for the model of the rc controller that will be used for this project. This module receives the radio signal from the controller and sends it as an iBus signal.

PCB

Given the space constraints, a printed circuit board has been designed to compact all sensors into a small space. This is the wiring diagram:

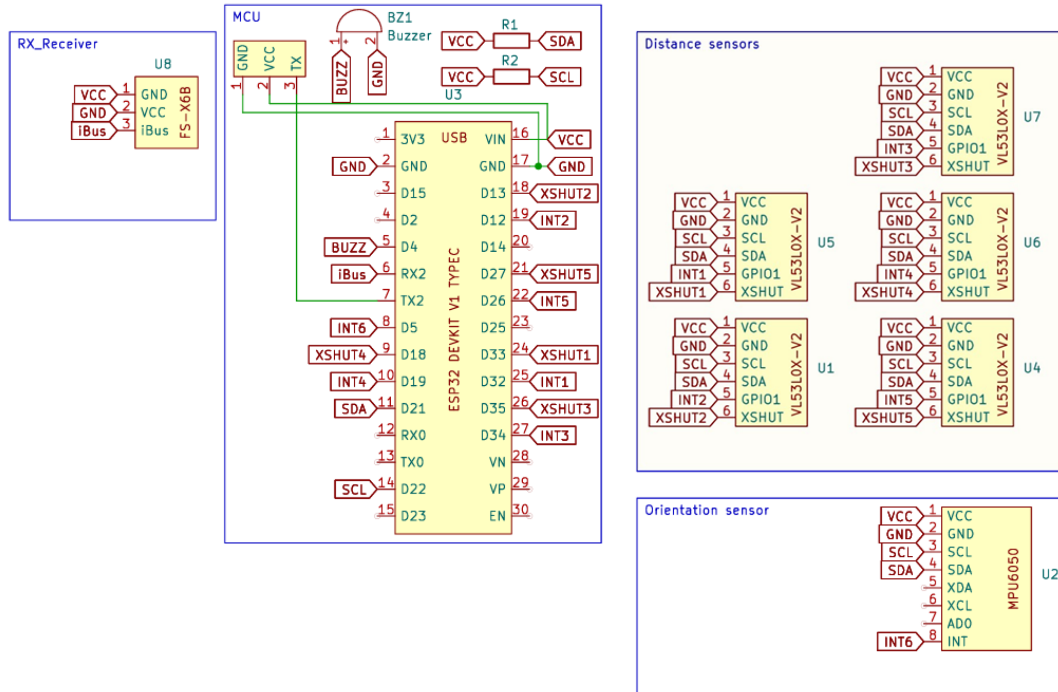


Figure 14 Wiring diagram for the module.

Notice that most of the pins of the ESP32 have been used, mainly by the reset and interrupt pins of all the sensors.

By following this diagram, the pins that have been selected in the software section will be understood.

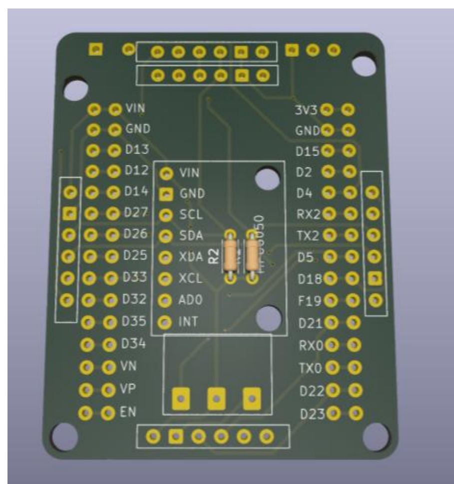


Figure 15 Render of the Designed Circuit Board

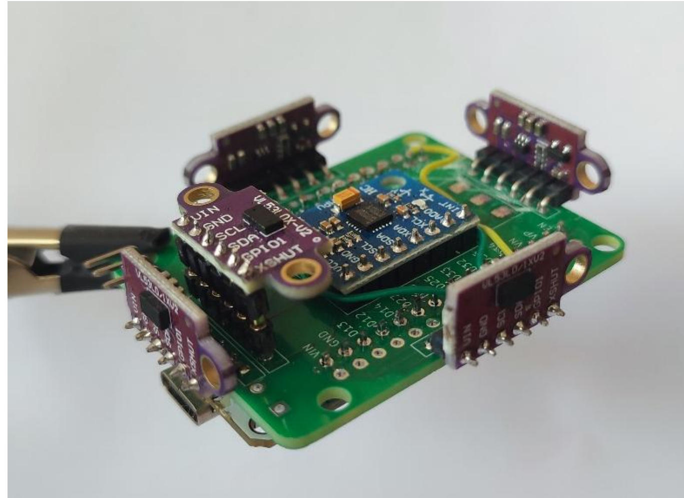


Figure 16 Bottom side of the Module Assembled

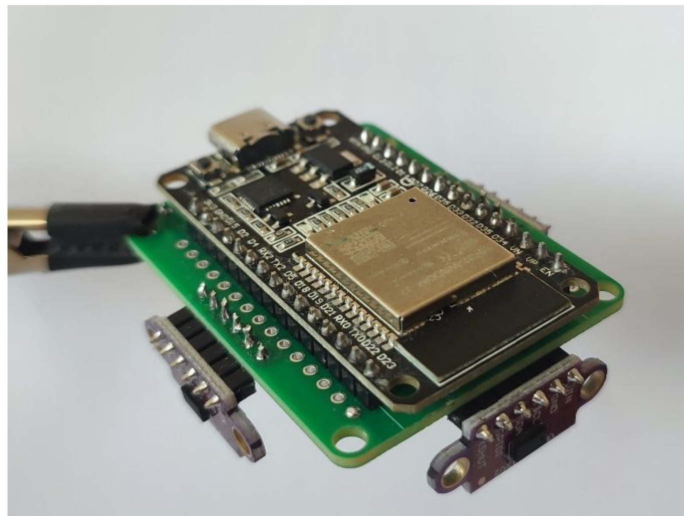


Figure 17 Top side of the Module Assembled

Software

The framework for this module has been developed using PlatformIO as the build system and dependency manager, within Visual Studio Code as the corresponding and recommended IDE. The code targets the ESP32 microcontroller platform using the Arduino Framework.

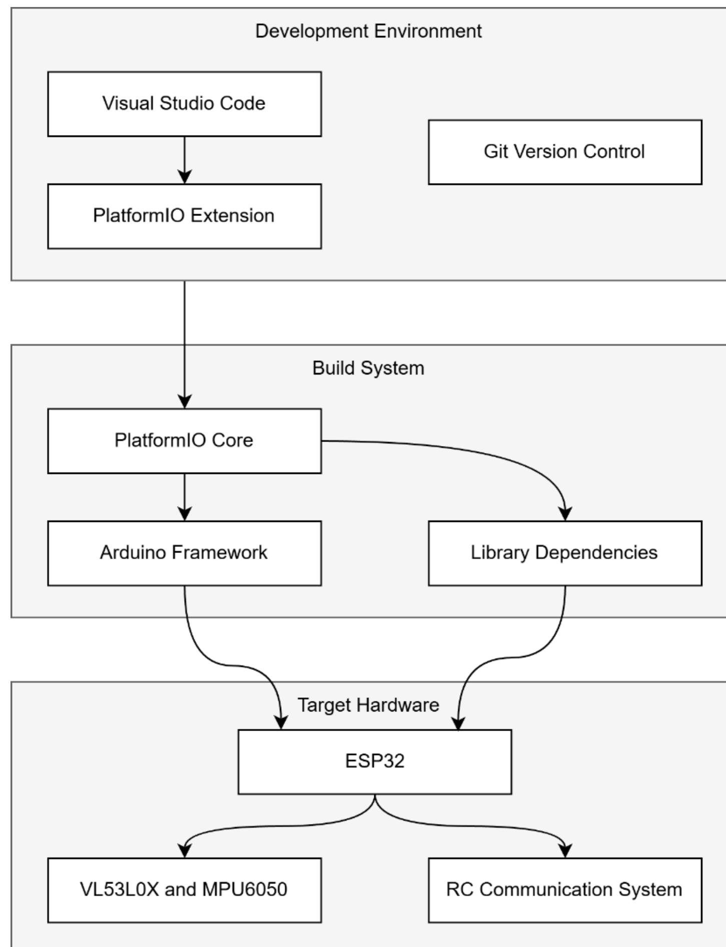


Figure 18 Development Tools Overview.

As the diagram shows, the development environment will be supported by the Git version control system, the build environment will be based on the arduino Framework. The environment will be Arduino as its developers offer great support and constant updates and improvements.

The alternative would be Espressif's own framework, ESP-IDF. It is written entirely in C and allows for almost bare-metal programming. However, support is lacking. With each update, the framework tends to change in ways that break existing libraries, and the performance gains just aren't significant enough to justify the extra effort required to adapt everything.

Project Libraries

The Json, TCP and AsyncWebServer libraries will help us to create the web interface for monitoring data over WiFi.

No existing code will be created, the development will focus on creating the drivers and tasks and the system that controls them and system error control.

Library	Version	Code Implementation	Purpose
Adafruit_VL53L0X	1.2.4	VL53L0X_Manager class	Interface with 5 VL53L0X distance sensors
Adafruit MPU6050	2.2.6	MPU6050_Manager class	Interface with MPU6050 IMU sensor
Adafruit Unified Sensor	1.1.15	Sensor base classes	Common sensor API abstraction
Madgwick	1.2.0	Orientation filter	Sensor fusion for IMU data processing
ArduinoJson	7.3.1	Web API JSON handling	JSON serialization for web dashboard
AsyncTCP	latest	Web server foundation	Asynchronous TCP for ESP32
ESPAsyncWebServer	latest	Web server implementation	Real-time WebSocket dashboard

Getting Started

To set up the development environment for the first time:

1. Install Visual Studio Conde
2. Install the PlatformIO extension from the VS Code marketplace.
3. Clone the repository: "**git clone https://github.com/JoanICG/Collision_avoidance_module_for_UAV".**
4. Open the cloned folder in VS Code
5. PlatformIO will automatically detect the project and install required dependencies.
6. Connect your ESP32 board via USB.
7. Use the PlatformIO build and upload buttons to compile and flash the firmware.

Building and Uploading

1. Open the project in VS Code with PlatformIO extension installed.
2. Connect the ESP32 development board via USB.
3. Use PlatformIO's build command or the build button in the PlatformIO sidebar.
4. Use PlatformIO's upload command to flash the firmware to the ESP32.
5. Use the serial monitor (115200 baud) to view debug output.

Code implementation

RcDriver

The RC Communication System provides a flexible, protocol-independent interface for receiving remote control commands. The system uses the Strategy design pattern to decouple protocol-specific implementation from the main driver logic, currently supporting the iBus protocol with provisions for future protocol extensions.

The RcDriver class will implement efficient memory management for auxiliary channel data to prevent repeated allocations during high-frequency RC updates. The pre-allocated auxBuffer eliminates dynamic allocation overhead during runtime RC processing, ensuring predictable performance for real-time applications.

The main methods are:

- begin(int baudRate = 115200)
  Initializes Serial2 communication and sets default iBus strategy if none configured.
- setStrategy(RcDriverStrategy* newStrategy)
  Changes the RC protocol strategy at runtime, properly cleaning up the previous strategy.
- update()
  Processes all available bytes from Serial2 buffer and delegates to the current strategy for protocol-specific handling.

```
void RcDriver::update() {
    // Process all available bytes in the serial buffer
    while (Serial2.available() > 0) {
        uint8_t incomingByte = Serial2.read();

        // Let the strategy handle the byte
        if (strategy != nullptr) {
            strategy->processByte(incomingByte);
        }
    }
}
```

Figure 19 RcDriver update function.

```
// Example usage pattern from main application
RcDriver rcDriver;
rcDriver.begin(115200);
rcDriver.setStrategy(new RcDriveriBus());

// In main loop
rcDriver.update();
RcInfo rcInfo;
rcDriver.getRcInfo(rcInfo);
```

Figure 20 Example RcDriver usage on main application

Data Flow and Processing

The data comes out of the RC receiver, it is sent through GPIO16. These data frames will be processed within a specific time frame to avoid adding delay and packet loss. These packets are arriving between 7 and 10 ms, with a baud rate of 115200 bps and a size of 32 bytes.

By default, the ESP32 UART buffer is 128 bytes, enabling an accumulation of 128 / 32, 4 iBus packets, taking about 40 ms maximum. This capacity could be increased to allow more information to be buffered, but due to timing constraints, the signal will have to be processed each 10 ms at most, so minimum delay be added.

Core functions:

- getRcInfo(RcInfo& info)
  Retrieves the latest parsed RC data through the strategy, managing the auxiliary channel buffer internally.
- sendRcInfo(const RcInfo& info)
  Transmits RC data if the protocol supports bidirectional communication, handling encoding and timing constraints.

When sending new information is needed, the function sendRcInfo will be called with the RcInfo structure as parameter, the driver will receive the pointer and before continuing with the process, it will check if it can transmit it by calling the function of the implemented strategy canTransmitNow, where it will answer if it is ready to send the information or not. Afterwards, the pointer will be sent to the interface to be copied so this data modification does not interfere in the main process, and finally encoding the information for sending it.

Each step will check for errors and return if anything happens with the appropriate error code.

Data will be retrieved at the physical pin GPIO16 and sent back at GPIO17.

Error Handling and Status

The RcDriver provides status information and error codes for robust system integration:

Return Code	Method	Meaning
-1	sendRcInfo()	No strategy set
-2	sendRcInfo()	Not ready to transmit
-3	sendRcInfo()	Encoding failed
-4	sendRcInfo()	Invalid buffer
0	sendRcInfo()	Success

RcDriverStrategy

This class will be the abstraction of all future implementations to translate the data coming from the RC receiver. It contains different data structures shown in the class diagrams above. These have a general purpose, and each user will have to create his own implementation following this scheme for a correct compatibility of the system.

The first important structure is States, this is created as an enumeration, in which there are three states, WAITING_HEADER, WAITING_DATA and WAITING_CHECKSUM. These three states define the state machine on which the driver will rely to check the correct reception of the message frame, being able to perform checks ahead of time and thus reducing the unnecessary waiting time in case of an early error.

When initialized, it will use by default the iBus strategy already implemented, but in the future, it can be changed to any other strategy following the RcDriverStrategy interface.

The task for processing the radio control information will call the update method to collect the new incoming information. The interface that is active at this moment will be in charge of the corresponding processing.

State	Purpose
WAITING_HEADER	Waiting for frame header/start bytes
WAITING_DATA	Receiving frame data payload
WAITING_CHECKSUM	Waiting for frame validation checksum

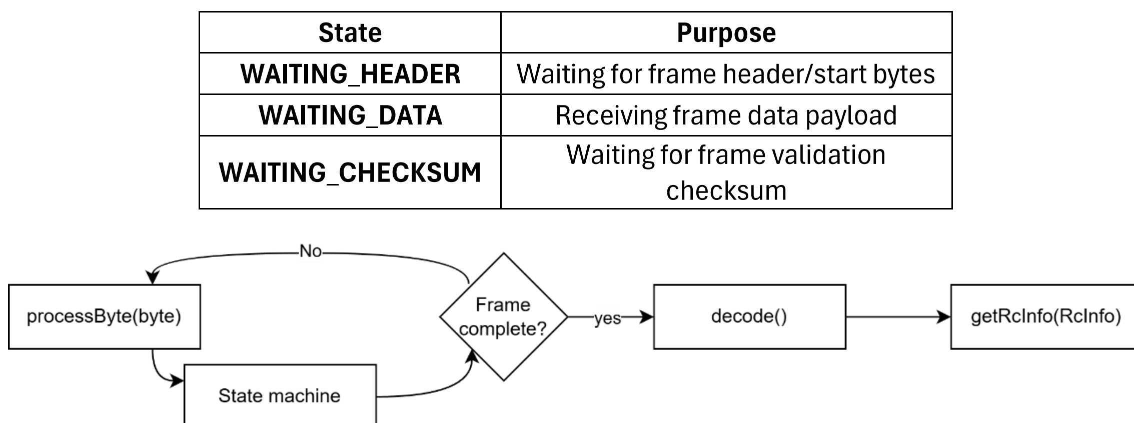


Figure 21 RcDriverStrategy data processing.

Normally UAVs send data with the system information, this is known as telemetry, a structure called TelemetryInfo has been created where this field will be simulated, although the receiver with which it is available hardly uses this information, so it will not be possible to perform tests.

Field	Type	Description
battery	uint8_t	Battery level indicator
temperature	uint8_t	System temperature
altitude	uint8_t	Current altitude
speed	uint8_t	Current speed

To store the necessary control information, the RC info structure has been created, consisting of the 4 main types of flight movements, the Throttle, Yaw, Pitch and Roll, followed by the auxiliary channels. To keep control of how many auxiliary channers are, it will store the quantity inside a variable. In this way, it can vary the number of auxiliary channels that the system has.

Field	Type	Description
throttle	uint16_t	Throttle channel value
yaw	uint16_t	Yaw channel value
pitch	uint16_t	Pitch channel value
roll	uint16_t	Roll channel value
Naux	uint8_t	Number of auxiliary channels
aux	uint16_t*	Pointer to auxiliary channel array
telemetry	TelemetryInfo	Telemetry data structure
timestamp	unsigned long	Last update timestamp

Each time it gets its update function called, it will store the new value and act accordingly to the following state machine schema.

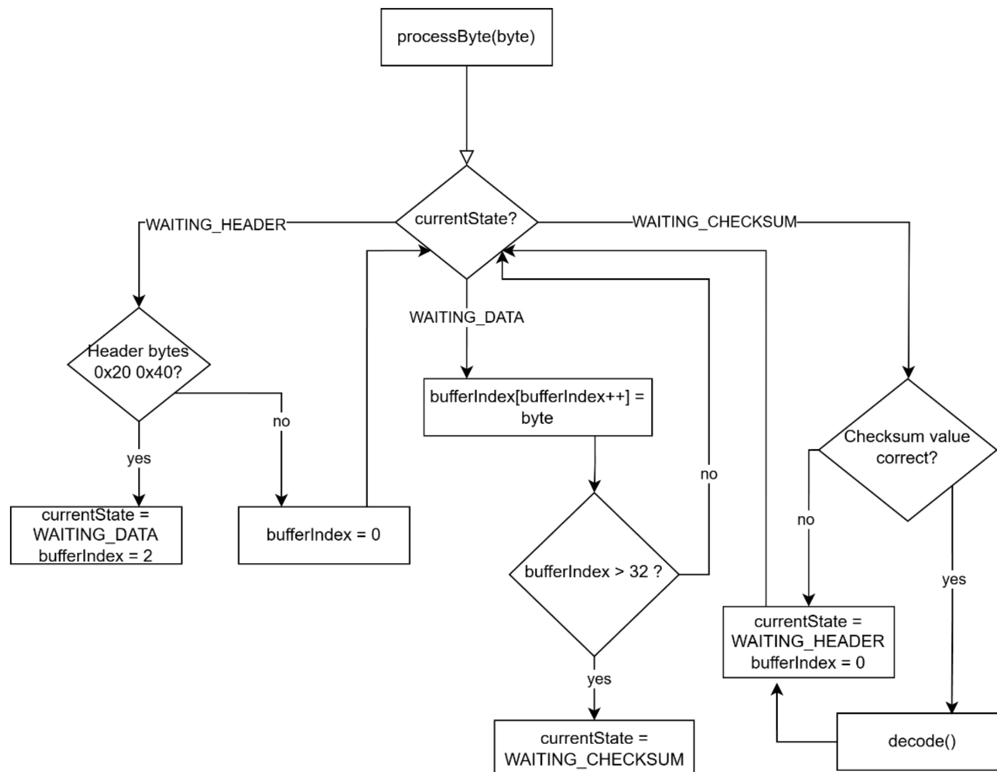


Figure 22 Incoming Data Process Flow

iBus Strategy

As the components that work with iBus are present for testing pourposes, the iBus protocol has been implemented as an example. This protocol has the following characteristics:

- Digital Serial Communication
  iBUS uses a digital UART-based protocol to transmit data between receiver and flight controller or other modules.
- Error Detection
  The protocol includes basic checksum-based error detection to improve reliability over noisy connections.
- Compact Data Format
  iBUS uses a compact binary format, reducing the amount of data transmitted per channel and improving efficiency.
- Supports Multiple Channels
  Typically supports up to 14 or more channels, depending on the receiver and transmitter firmware.
- 3.3V/5V Logic Compatibility
  Although often used with 5V logic levels, iBUS can also operate with 3.3V systems, making it flexible for microcontrollers like ESP32 or STM32.
- Simple Wiring
  Only three wires are needed: VCC, GND, and a single signal wire.
- Proprietary Protocol
  iBUS is proprietary to FlySky, which means documentation might be limited, though information about it can be find over the internet.
- Fixed Baud Rate
  iBUS operates at a standard baud rate of 115200 bps, which must be matched on the microcontroller's UART.

```
20 40 DC 05 DC 05 EF 03 DC 05 E8 03 E8 03 E8 03 E8 03 E8 03 E8 03 DC 05 DC 05 DC 05 DC 05
20 40 DC 05 DC 05 EF 03 DC C1 E8 60 DF 60 DF 60 DF 60 DF 60 DF 60 AF 18 AF 30 AF 30 AF 18
20 40 DC 05 DC 05 EF 03 DC 05 E8 03 E8 03 E8 03 E8 03 E8 03 E8 01 DC 05 DC 05 DC 05 10 40
20 40 DC 05 DC 05 EF 03 DC 05 E8 03 E8 03 E8 03 E8 03 E8 03 E8 03 DC 05 DC 05 DC 05 DC 05
20 40 DC 05 DC 05 EF 03 DE 05 E8 03 E8 03 E8 03 E8 03 E8 03 E8 03 DC 05 DE 05 DC 05 DC 05
20 40 DC C1 DC 05 EF 03 DC 05 E8 03 F4 03 E8 03 E8 03 E8 03 F4 03 DC 05 DC 05 DC 05 EE 05
```

Figure 23 iBus data frame snippet.

The iBUS signal is structured as a binary frame composed of a header, 14 channels, and a checksum at the end. Each part is divided into 2-byte blocks, and all multi-byte values are encoded in little-endian format—meaning the least significant byte comes first.

The frame starts with a 2-byte header, 0x20 0x40, followed by the 14 data channels ranging between the 1000 to 2000 value, and finally it ends with a 2-byte checksum, making the structure 32 bytes long.

- 2 bytes for the header.
- 28 bytes for the c14 channels (2 x 14).
- 2 bytes for the checksum.

The implementation for the decode method extract the 16-bit channel values from the buffer using little-endian byte order:

```
// Main channels (bytes 2-9)
rcInfo.throttle = buffer[2] | (buffer[3] << 8);
rcInfo.yaw = buffer[4] | (buffer[5] << 8);
rcInfo.pitch = buffer[6] | (buffer[7] << 8);
rcInfo.roll = buffer[8] | (buffer[9] << 8);

// Auxiliary channels (bytes 10-29)
for (int i = 0; i < rcInfo.Naux; i++) {
    int bufferIndex = 10 + (i * 2);
    rcInfo.aux[i] = buffer[bufferIndex] | (buffer[bufferIndex + 1] << 8);
}
```

Figure 24 Decode function snippet.

For encoding, the process is similar, but in reverse and calculating the checksum by adding the values and subtracting the sum at 0xFFFF:

```
    txBuffer[0] = IBUS_HEADER_BYTE1;
    txBuffer[1] = IBUS_HEADER_BYTE2;

    // Channel data (16-bit values in little-endian format)
    // Store as-is without mapping

    // Transformacio a little endian
    // Store main channels
    txBuffer[2] = rcInfo.throttle & 0xFF;
    txBuffer[3] = (rcInfo.throttle >> 8) & 0xFF;
.
.
.
   // Store aux channels
    for (int i = 0; i < rcInfo.Naux && i < (IBUS_MAX_CHANNELS - 4); i++){
        int bufferIndex = 10 + (i * 2);

        txBuffer[bufferIndex] = rcInfo.aux[i] & 0xFF;
        txBuffer[bufferIndex + 1] = (rcInfo.aux[i] >> 8) & 0xFF;
    }

// Calculate the checksum
    uint16_t checksum = 0xFFFF;
    for (int i = 0; i < IBUS_FRAME_LENGTH - 2; i++) {
        checksum -= txBuffer[i];
    }
```

Figure 25 Encode function snippet

VL53L0X Distance Sensors

The implementation that has been carried out is using 5 TOF VL53l0X sensors that are going to be used together by the class VL53L0X. Using the facade design pattern, this middleware is created so that the 5 sensors can be used together or independently in a very easy way.

The SensorPosition enumeration will be created.

- LEFT
- FRONT
- RIGHT
- BACK
- BOTTOM

Using the Adafruit_VL53L0X.h library, we can configure each of the sensors to set which address it is configured to, which pin the shutdown and interrupt pins are connected to. To do this we have created a data structure that contains all this information for subsequent easier management. This and with the SensorPosition enumeration, the user will be able to access just by sending the corresponding SensorPosition value, for example:

```
vl53l0xSensors.getMesaurement(LEFT, sensorMeasures[i]);
```

This line is going to store the measurement from the left sensor on its corresponding position of the sensorMesaures array.

Distance Sensors Initialization Sequence

As described by the manufacturer, the sensors have the predefined address 0x20 on the I2C bus. This is a serious problem as it will cause collisions with the other 4 sensors. For this issue, there is a function to change the memory address, but it requires that only one of them is active. The XSHUT pin allows us to turn off the sensor by setting the state to LOW. Therefore, it can perform the following sequence five times to set all the sensors with different addresses:

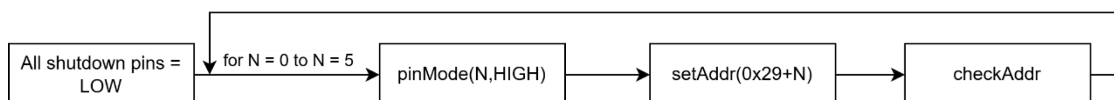


Figure 26 Initialization Sequence for VL53L0X

The begin function of the Adafruit_VL53L0X.h library functionality it is not suitable for the project. Is based on making several attempts in case a sensor does not respond, taking an indeterminate time that breaks the deterministic behaviour of our project. Not only this function, but all the functions from the library also have this procedure.

To solve this inconvenience, we obtain the address of the module to which we want to send the command, and we check that it has response in the following method:

```
        Wire.beginTransmission(sensors[i].address);
        byte address_check_error = Wire.endTransmission();
if (address_check_error == 0) {. . .
```

These two lines of code create and close a connection to the device that responds to the specified address. If this procedure is performed correctly, the error code will be 0, indicating that a device is indeed enabled. In short, it is performing a ping that will prevent the m from a system crash.

After initializing the sensors, the interrupts will be configured. They can be configured to change the state of the interrupt pin from HIGH to LOW or vice versa.

The sensor will be told from which mode it will trigger the interruption: when a new measurement is taken or when a new measurement is taken within the indicated threshold.

After configuring the pin behaviour, the function attachInterrupt is called, to which the corresponding interrupt pin will be passed and the function to be called (this of the specific type IRAM_ATTR).

```
sensors[sensor].psensor->setGpioConfig(VL53L0X_DEVICEMODE_CONTINUOUS_RANGING,
                                       VL53L0X_GPIOFUNCTIONALITY_THRESHOLD_CROSSED_LOW,
                                       VL53L0X_INTERRUPTPOLARITY_LOW);
```

Then, the sensor will be instructed to take measurements constantly, and when a measurement passes the threshold, it will have to set the interrupt pin to LOW.

And finally the startMesaurement command is sent:

```
sensors[sensor].psensor->startMeasurement();
```

Interrupts will simply trigger a flag to indicate that the sensor is available with a new measurement within the desired range.

Subsequently, by means of a referenced interrupt system, the value will be collected and the interrupt will be restarted.

These sensors are designed for recreational use and are flawed in terms of accuracy and performance. During the development of the project, it has been detected that they stop giving interruptions after an indefinite period of time, that is why a function has been created to reset the sensor that is giving problems, disconnecting it from the system in case it stops responding even after restarting it.

The driver also has a list where it keeps track of which sensors are available, this gives us a double layer of security to avoid accessing a sensor that is not working. It will also avoid unnecessary work if we want to use a general configuration function.

The operating modes of the sensor are high accuracy, standard mode, and long range. They can be alternated between by changing the budget they have to take measurements.

Each of these modes sacrifices accuracy for speed and maximum measured distance, with the standard and high accuracy modes being the most limited, starting to fail after 1200mm, and the long-range mode having times ten times longer than the high-speed mode, but reaching almost 2000mm of maximum measurement distance.

MPU6050 Orientation Sensor

The MPU6050 is a 6-axis (gyroscope + accelerometer) motion tracking device that provides raw motion data. This data is processed to determine the UAV's:

- **Pitch**: Forward/backward tilt
- **Roll**: Left/right tilt
- **Yaw**: Rotation around vertical axis

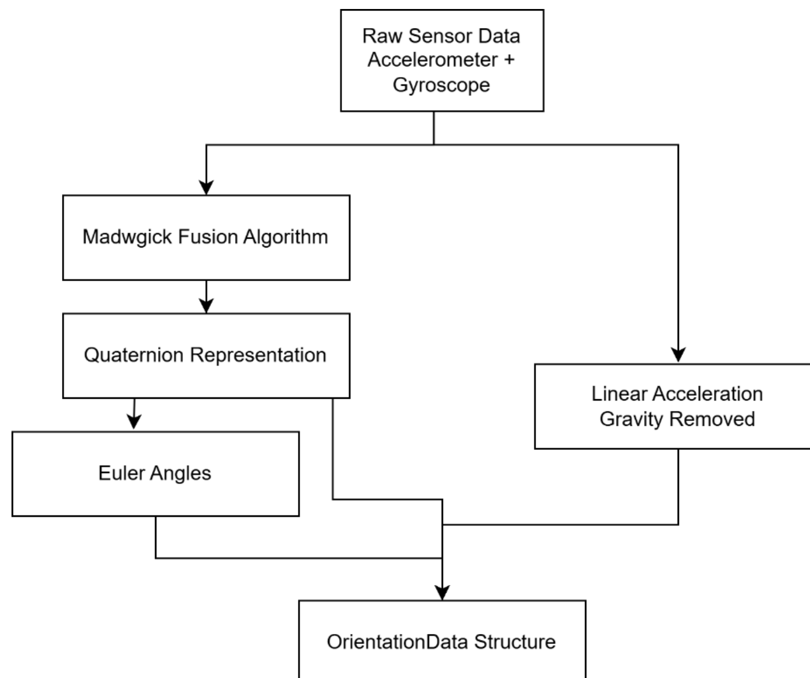


Figure 27 Orientation data gathering process.

The MPU6050_Manager class offers methods to access both raw sensor data and processed orientation information. And for testing purposes, allows the sensor to be operated in polling mode or interrupt-driven mode.

The structure supports both Euler angles for intuitive understanding and quaternions for advanced 3D calculations without gimbal lock issues.

```
bool begin(bool useInterrupts = true, bool performCalibration = true,
           MPU6050InterruptCallback interruptCallback = nullptr);
```

The begin class allows this initialisation mode to be easily selected, allowing the use of interrupts, an initial calibration and the passing of a function for the interrupt call.

The sensor MPU6050 is an IMU containing an accelerometer and a gyroscope, which measures linear and angular acceleration, however, taking this data without any processing could be error-prone, as it is full of noise and cumulative errors.

The accelerometer is very sensitive to noise and vibrations, which is very common in aircraft where this module is to be incorporated.

The gyroscope has cumulative errors, or drift. As time goes by, it becomes less and less accurate.

Therefore, a filtering algorithm such as Madgwick is implemented.

This algorithm, apart from having its own library for easy implementation, is one of the most suitable algorithms for an embedded system like our module, its accuracy is much higher than its complexity. It is also optimised for IMU sensors like the one used for this project.

It allows us to efficiently merge the gyroscope and accelerometer data and correct the drift that both provide, all quickly and in real time.

```
madgwickFilter.begin(0.1f);
madgwickFilter.setSampleFreq(MPU_SAMPLE_RATE);
```

The filter will operate at 100hz (the same as the sensor) with a beta value of 0.1.

The MPU6050 is configured for optimal precision:

- Accelerometer range: ±4G (MPU6050_RANGE_4_G)
- Gyroscope range: ±250°/s (MPU6050_RANGE_250_DEG)
- Filter bandwidth: 10 Hz (MPU6050_BAND_10_HZ)

Integration with Main System

The class offers the method getOrientation as a primary interface, populating the structure OrientationData that its pointer is shared.

Only the interrupt mode has been tested, working in the same way as the distance sensors, when the interrupt is triggered, the frag will be activated indicating that the data structure can be refreshed.

Main App

The Main Controller performs comprehensive system initialization through the setup() function, establishing all hardware interfaces, FreeRTOS synchronization primitives, and task creation.

It will initialize multiple hardware interfaces in the order described before to ensure proper operation.

Interface	Configuration	Purpose
Serial	115200 baud	Debug output and system monitoring
I2C (Wire)	Default pins	VL53L0X sensor communication
Serial2	115200 baud	RC iBus protocol communication
WiFi	Access Point mode	Web dashboard connectivity

Two critical mutex semaphores ensure thread-safe access to shared data structures:

- rcDataSemaphore: Protects currentRcInfo structure containing RC input data
- masterWebDataSemaphore: Protects masterWebData structure for web interface updates

The main controller implements a multi-core task architecture with prioritised scheduling to ensure real-time performance of critical RC processing, while maintaining sensor monitoring and user interface responsiveness. Task partitioning allows us to separate critical tasks from non-critical tasks.

The critical safety requirements are to ensure that the drone does not fail (motors shut down) or lose control. There are different types of situations that can cause these undesired events, which can be classified as follows:

Category	Examples	Priority
Critical Errors	Memory corruption, hardware failure	High
Communication Errors	Timeouts, checksum failures	Medium
Protocol Errors	Invalid data, frame errors	Medium
Resource Errors	Memory allocation failures	High
Timing Errors	Protocol timing violations	Low

Critical errors are those that will cause catastrophic failure of the aircraft (fail from the sky). For example, an access to an incorrect memory area or a memory overflow. The ESP32 board has a recovery mechanism that restarts immediately, which together with the fast system initialisation system would not be a problem if the error was a one-off error, but there is no guarantee that the error will not occur again, so an error detection and correction system has to be implemented.

FreeRTOS provides a mechanism of hooks that are triggered in case of different events. These are named after the task that generated the error. With this we can enable or disable flags that block the execution of the task in the future or change its behaviour.

For instance:

```
void helloTask(void* param) {
    while (true) {
        if (!helloTaskEnabled || systemError) {
            vTaskDelay(pdMS_TO_TICKS(500));
            continue;
        }
        Serial.println("Hello from helloTask!");
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

In case this task generates a critical error (unlikely), an exeption will be generated and the hook will take its defined action to detect who has generated this error and subsequently disable the helloTaskEnabled flag.

Communication errors are less critical, i.e. if they occur they will not cause the loss of the aircraft, as long as they do not occur continuously.

These errors can be timeouts, checksum errors or incorrect values. The driver most prone to generate these errors is the RcDriver, the packets could be delayed or even lost. Therefore, the handling of this error has been implemented in the class itself.

Depending on the return value of the function calls to get the data and even send them, we have to act in such a way that we control the state continuously.

The driver may return an error when sending the packet, either because it is incorrectly formatted, or prematurely. We will have to register this incident with a counter and establish a tolerance threshold to transform this incident into a critical error, since it will indicate that there is an error in the driver that does not allow to take control of the aircraft. On the other hand, if this is a punctual error, it will not cause this failure.

By implementing these two forms of error detection, by message passing and hooks, a prioritisation of error handling is achieved, with critical errors being the first to be responded to.

Finally, timing errors are those that are generated when timing requirements are not met. For these cases, the timestamp has been implemented in the data structures so that each time they are processed, it can be checked if it is within its time margin. These errors are minor and will simply result in a degradation of the service. As with the previous ones, if they are repeated, they will become moderate errors that require a solution in order not to escalate to a major problem.

The main program will also monitor the data for consistency, e.g. if the distance sensor suddenly gives wildly varying values, it will indicate that there is a problem with the sensor readings. Or if the control data suddenly starts to oscillate, it would indicate that something is indeed happening with the receiver and will need to be verified.

System Tasks

In order to provide the system with a deterministic behaviour, the tasks have to be planned in advance, a time frame must be established in which all tasks have to be fulfilled in a specific order.

The top priority is the control of the aircraft, so we have to make sure that the incoming data is processed and sent without adding too much delay. Taking into consideration that an iBus packet takes at most 10ms, we will base the whole system on this interval.

Subsequently, measurements have to be taken of how long each task takes to complete:

Task	Min	Max	Average
RC Task	21	414	37
Disntance Task	312	527	384
Orientation Task	324	786	492
Web Task	457	921	652

Based on these values, tasks can be created to run within the 10ms margin of the iBus, time slots will be created to work with the sensors in case of interruption. In the worst case, the remaining time is 7ms, which will be used for flight calculations.

It is not possible to reserve a fixed time slot for sensor data collection, as the sensors are interrupt-based, therefore the deferred interrupts methodology will be applied, allocating a time slot of 2 ms to deal with all sensors in the worst-case scenario that all sensors are activated.

FreeRTOS also allows to assign priorities to tasks, with the highest priority task being the one with the highest number. The order of priorities, from highest to lowest, will be:

RCTask -> OrientationTask -> Distance Task ->FlightCorecctionTask -> Web Task

It also allows tasks to be separated by processors, the ESP32 has two computational units. As explained above, we will assign critical tasks to core 0 and non-critical tasks to core 1.

This leaves RCTask, OrientationTask and DistanceTask, FlightCorrectionTask on core 0, and Web Task on processor 1.

This assignment can be changed very easily using the task creation parameters, for example:

```
    xTaskCreatePinnedToCore(
        rcUpdateTask,
        "RCTask",
        4096,
        NULL,
        3,
        &rcTaskHandle,
        0
    );
```

The first parameter is the function to be executed, the second the name to identify it, then the memory we assign to it, the following parameters, the priority, the task controller and finally the attached core.

Once the tasks have been created, they will start running.

Flight Control

The flight control class is responsible for applying the necessary corrections for the selected flight behaviour. This will be an interface having the ControlData structure, which contains a structure to store all necessary information, consisting of information from all sensors and information from the radio control.

The modes will be used following the strategy pattern, and can be changed at flight time, for safety reasons, there will be a hardcoded protocol in case the pointer to this new implementation is erroneous or cannot be executed, the flight control will not be stopped.

Height control

The objective is to keep the drone in a fixed height position, not fixed in space as there is no way to control the actual position, by using the distance sensor labelled BOTTOM and disabling the others. In case this sensor is not available, this mode will not be executed.

As the aircraft is a drone, it regulates its height by modifying the throttle value, the higher it is, the more thrust it generates and the higher it rises.

Therefore, the height compensation will be done by a PID control, where it will modify the value of the throttle by calculating the difference of the height measurement taken by the distance sensor and the desired height value (keeping this limited to the maximum range of the sensor).

A PID system will not be implemented, as there are already proven libraries that perform this function.

The control flow will be:

1. Data recollection
    a. Obtain $h_{mesaured}$
    b. Obtain pitch and roll angles.
    c. Read $h_{desired}$
2. Calculate error:

$$e(t) = h_{desired} - h_{mesaured}$$

3. Compute PID:

    The function from the library will be used with configurable Kp Ki and Kd values.

4. Apply Tilt Compensation:
    a. Compensatoin Factor = $\frac{1}{cos(pitch)*cos(roll)}$
    b. Ensure compensation factor is reasonable
    c. Compensated pid = original PID * Compensation Factor

5. Map to Throttle Range
    a. Map compensated PID to the max and min value of the throttle, in this case 1000 and 2000.
    b. Assume a value for hovering, for this drone is 1400, then add the mapped value with a scaling factor to avoid abrupt changes.
    c. Ensure the new value is between the min max values of the corresponding rc protocol.
6. Smooth Throttle Changes
    a. Apply low-pass filter.
7. Output Throttle

Given the dangerous nature of the tests, a simulation has been created in Python where the characteristics of the drone are parameterised, and the behaviour of the drone can be visualised:

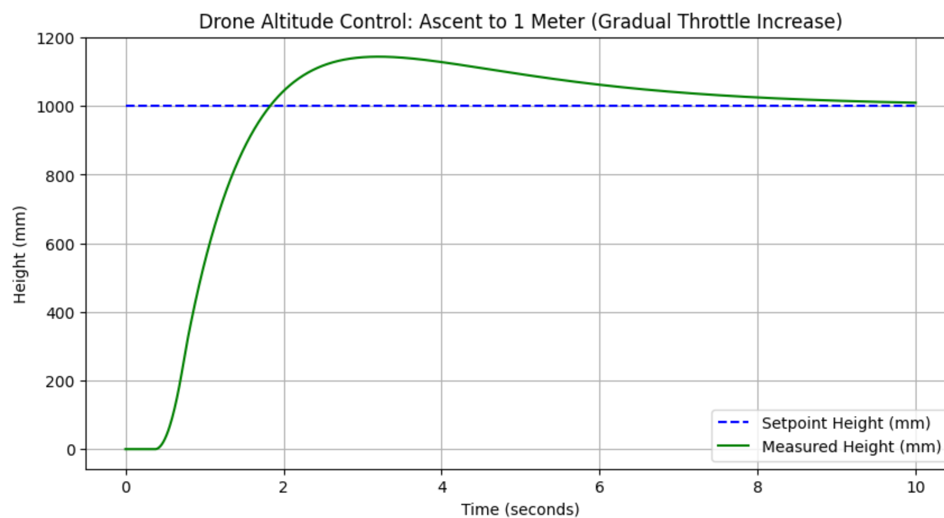


Figure 28 Heigh simulation result.

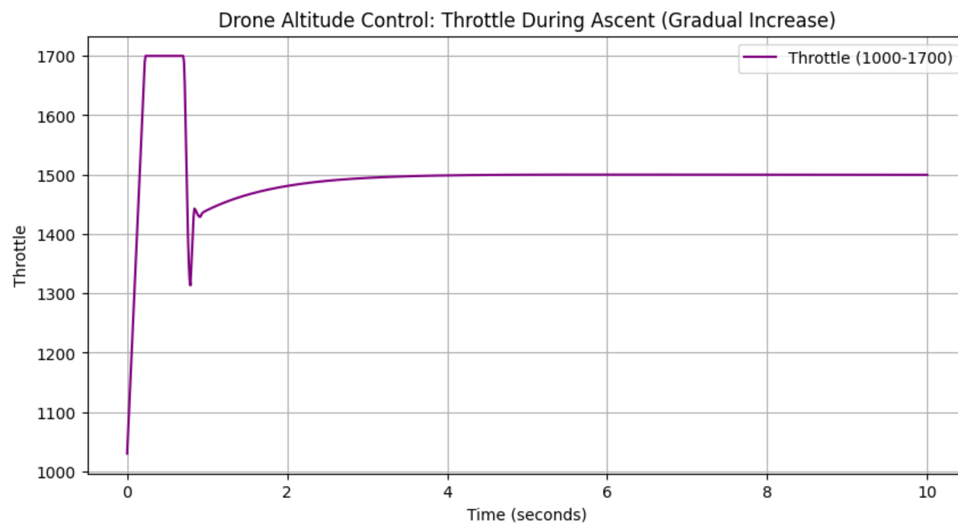


Figure 29 Throttle simulation result.

In the simulations, the behaviour of the algorithm can be visualised. The first graph shows the change in height based on the Throttle applied in the second graph. Safety measures such as the applicable throttle limitation can be observed. The simulation is only an approximation of reality that requires fine-tuning, but it serves to test the performance of the algorithm without compromising the safety of anything or anyone.

Anti-collision control

The drones move by tilting towards where they want to go. We will take advantage of one of the stabilisation modes that drones have, this is called horizon, and it consists of imitating the position of the control stick of the remote control, that is, the drone will be parallel to the ground when the stick is centred, it will tilt towards the direction that the stick points, 1500 being the central value, and 1000 and 2000 the maximum value, those will be mapped to a defined max angle. In this way, the drone will never flip over on itself.

The algorithm will be created on this basis. It will collect the values from all sensors except the one labelled BOTTOM, and by analysing the difference in distance based on time it will be able to detect if any object or wall will collide with it.

Control logic:

1. Read distance sensor values.
2. Calculate distance variation: $\Delta D/\Delta t$
3. If variation is over V_crit and distance is lower than D_safe, adjust the roll or the pitch to counter this movement towards the obstacle.
4. Send those new values to the drone.

Python simulations, path analysis and signal modification have also been carried out for this algorithm.

In the simulation, the situation is posed in which the drone is floating vertically and moves by inertia, it is going to collide against a side wall and a front wall, so it will have to modify the corresponding parameters in order not to collide.

Those are the results:

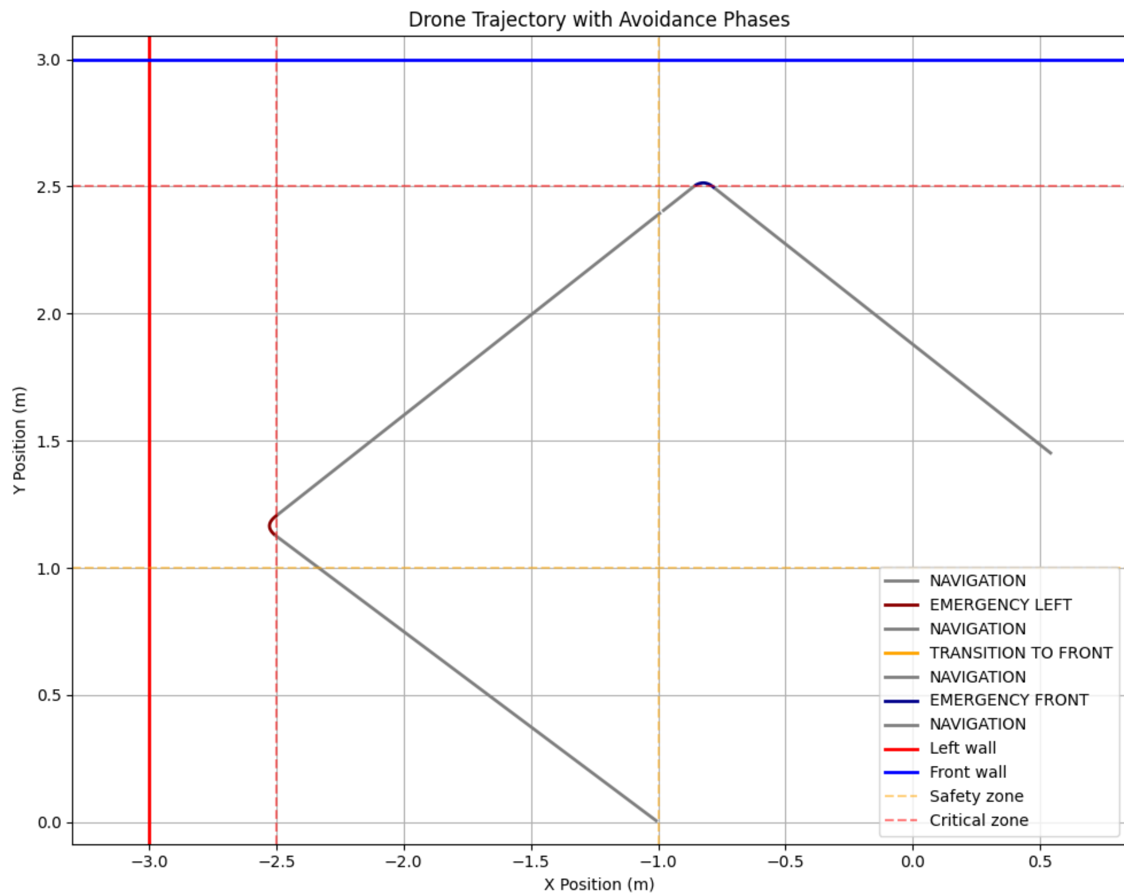


Figure 30 Simulation result of the collision avoidance mechanism.

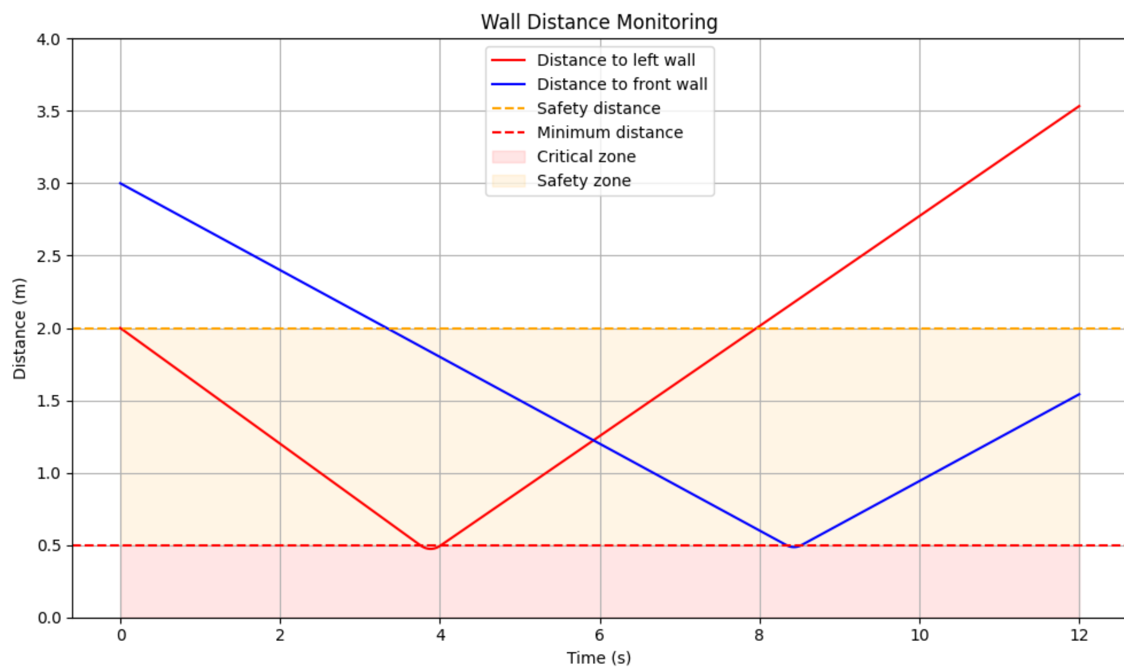


Figure 31 Simulation result of the distance monitoring

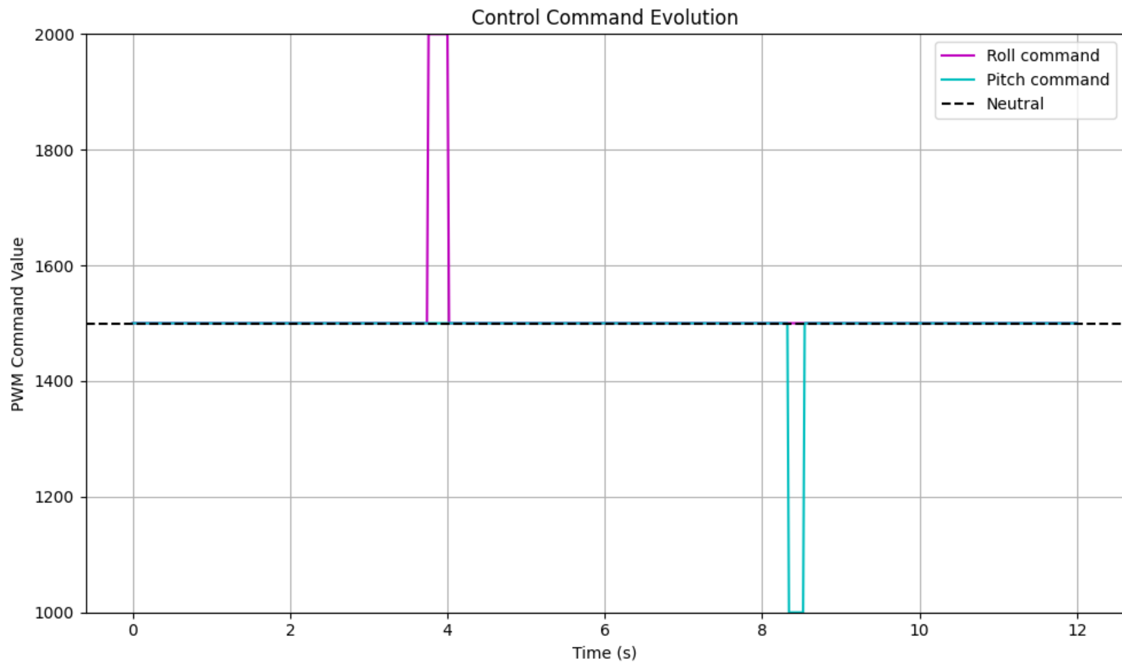


Figure 32Simulation result of the command evolution

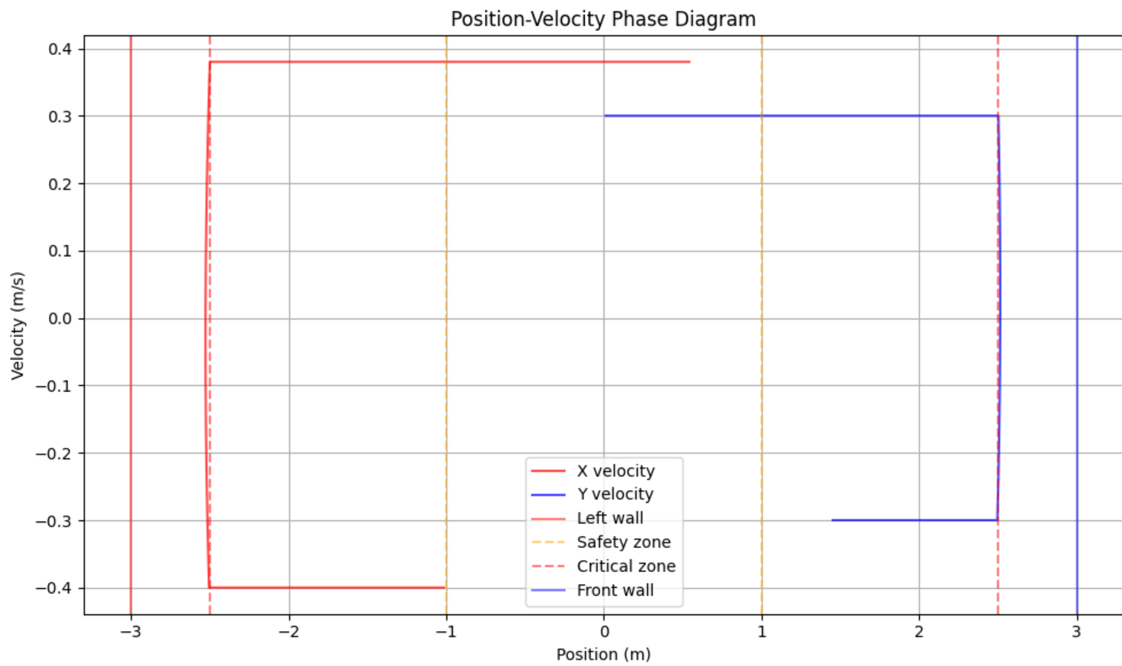


Figure 33 Simulation result of teh Position-Velocit Phase.

Web Interface

This provides a real-time monitoring dashboard by creating a WiFi access point and serving an HTML dashboard that displays sensor data, RC controller inputs, and gyroscope readings via WebSocket connections. The interface enables operators to monitor system status and sensor readings in real-time through any web browser.

The web server operates as an embedded HTTP server with WebSocket support, providing real-time bidirectional communication between the ESP32 and web clients. The system is built on the ESP32's WiFi capabilities using the ESPAsyncWebServer library.

It uses a data structure to organize and transmit all the sensor information efficiently, this structure has the following fields:

- RcInfo
- sensorDistance
- sensorStatus
- OrientationData

The WebSocket system enables real-time bidirectional communication between the ESP32 and connected web clients. Data flows from sensors through the ESP32 to the web dashboard via JSON messages.

Display Structure

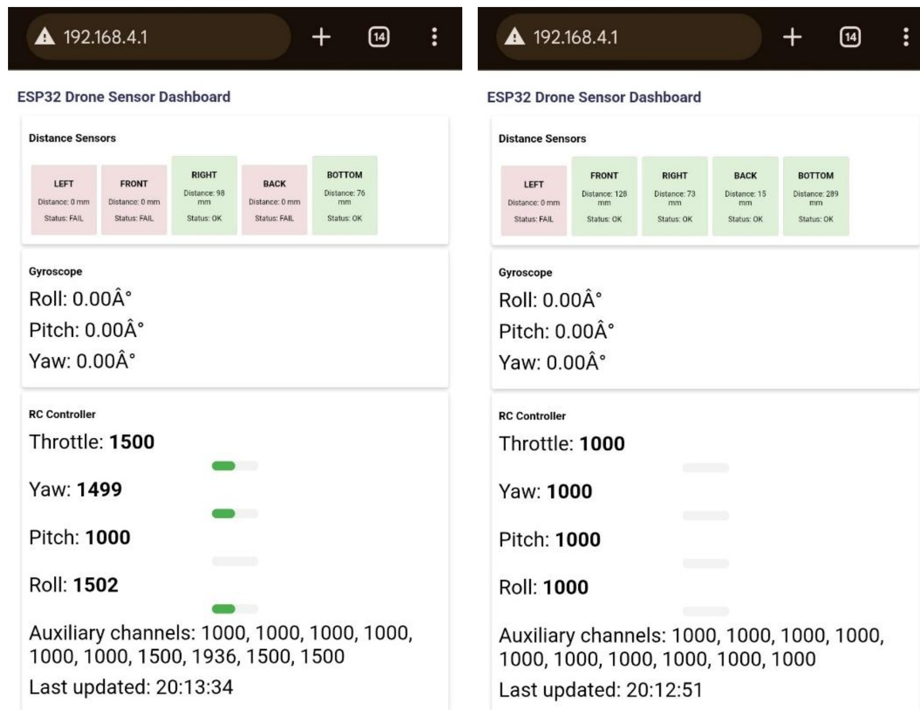


Figure 34 Dashboard with 2 working sensors *Figure 35 Dashboard with 4 working sensors*

Evaluation of personnel and material costs

Resource	Quantity	Price for unit	Total price
ESP32	1	5	5,00 €
VL53L0X	5	4,19	20,95 €
MPU6050	1	3,25	3,25 €
PCB	1	2	2,00 €
PCB Shippment	1	23	23,00 €
Engineer	300	20	6.000,00 €
			6.054,20 €

An ESP32, five VL53L0X, an MPU6050, a PCB and a junior computer engineer were used for this project.

Product prices were taken June 2025.

The hourly rate for the junior computer engineer is €20. He has dedicated a total of approximately 300 hours, from January to June.

The total cost of the project would amount to 6.054,20 euros in total.

Legislation and data protection

The realization of this project does not deal with personal data, but it must comply with the legislation concerning airspace.

If tests are to be performed, they must comply with the following conditions:

a) At a minimum distance of 8 km from the reference point of airport or airfield and the same distance from the axes of the runways and their extension, in both headwaters, up to a distance of 6 km counted from the threshold in the direction away from the runway. This minimum distance may be reduced when it has been agreed with the airport manager or responsible for the infrastructure, and, if any, with the air traffic services provider, and the operation shall be in accordance with the provisions established by them in the corresponding coordination procedure. Infrastructures intended for the use of RPAS can and must have a coordination procedure if they are located in the area affected by an aerodrome.
b) Outside controlled airspace, flight information zones (FIZ) or any aerodrome traffic zone (ATZ). You can find a map to consult the airspace restrictions in ENAIRE's web application: https://drones.enaire.es/
c) At a maximum height above ground no higher than 400 ft (120m), or over the highest obstacle located within 500 ft (150m) of the aircraft.
d) In daytime flight and under visual flight meteorological conditions, except in the case of aircraft weighing less than 2 kg operating at a maximum altitude of no more than 50 meters.
e) Within the visual range of the pilot, without the aid of optical or electronic devices, except corrective lenses or sunglasses. In the event that first-person view (FPV) devices are used, the operation must be performed within visual range, without the aid of such devices, of observers who remain in permanent contact with the pilot.
f) Giving priority to all other categories of aircraft, except for aircraft weighing less than 2 kg operating at a maximum altitude of no more than 50 meters.
g) Outside agglomerations of buildings in cities, towns or inhabited places or outdoor gatherings of people, except for aircraft of less than 250 g operating at a maximum height of no more than 20 meters.
h) Outside the areas reserved, prohibited or restricted to air navigation, as well as over the installations referred to in article 32, with the limits foreseen in said precept, except in coordination with the owners responsible for the same.

Ethical, equity and environmental implications

From an ethical point of view, the development and use of drones raises several relevant issues. In this project, the drone has been reconditioned for experimental, educational and potentially civilian purposes. Its use for military or intrusive surveillance applications, which could involve violations of fundamental rights such as privacy, is neither envisaged nor contemplated.

In addition, safety principles have been followed in the design of the avionics system, minimizing risks during testing and ensuring that the drone does not pose a danger to people, animals or infrastructure. This position responds to a commitment to the responsible use of technology.

Access to technologies such as drones and the knowledge related to their development is still restricted in many regions or social sectors. This work starts from the reuse of an old drone and accessible components, with the objective of demonstrating that it is possible to make relevant technical advances without great economic resources.

Promoting reuse and technical training from recycled or low-cost materials can contribute to greater technological equity, facilitating learning and innovation also in educational contexts with limited resources.

Refurbishing an existing drone has a positive impact from an environmental point of view, avoiding the generation of electronic waste and the purchase of new equipment.

However, it is important to note that the electronic components used, such as sensors or microcontrollers, involve manufacturing processes with a considerable ecological footprint. Therefore, priority has been given to the use of off-the-shelf components, extending their useful life cycle and reducing the consumption of new resources.

Project Evaluation

The realization of this project has been a highly enriching experience both academically and personally. Throughout its development, I have had the opportunity to apply in a practical way knowledge acquired in key areas such as real-time systems, algorithmics, and signal processing and filtering systems. This integration of disciplines has allowed me to consolidate essential technical skills and to understand in greater depth the interrelationship between different fields of engineering.

In addition, the project has given me the opportunity to explore in greater detail a specific field that arouses great interest in me and in which I would like to focus my professional career. This immersion has reinforced my motivation to continue training in this line and has confirmed my vocation towards this field. In summary, I consider that this work has not only contributed significantly to my academic development, but has also been key to define more clearly my future professional goals.

I must add that this project is only the base of what could be something bigger, adjustments in the algorithms and signal processing are still needed, and above all, many hours of flying.

That will be solved in my career as a professional in this sector.

Resources

FreeRtos - URL www.freertos.org

VL53L0X Libraries – URL https://github.com/adafruit/Adafruit_VL53L0X

MPU6050 Libraries – URL https://docs.arduino.cc/libraries/mpu6050/

PlatformIO – URL https://platformio.org/

Madgwick Filtering – URL https://medium.com/@k66115704/imu-madgwick-filter-explanation-556fbe7f02e3

PID Explanation – URL https://pidexplained.com/pid-controller-explained/

ESP32 – URL https://www.espressif.com/en/products/socs/esp32