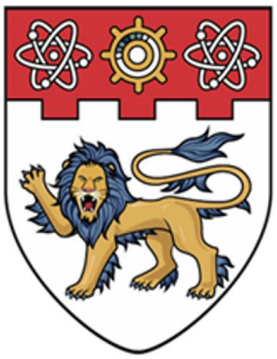

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CCDS24-0193: Hardware/Software Test and Validation System

Submitted By: Chan Yon Nan

Matriculation Number: U2223291D

Bachelor of Engineering in Computer Engineering

Project Supervisor: Mr. Oh Hong Lye

Examiner: Dr. Tan Wen Jun

College of Computing and Data Science

Academic Year: 2024/2025

Abstract

The MDP is an essential module for students in the College of Computing and Data Science emphasizing collaboration between teams managing various components to build a functional robot car. The utilization of diverse sensors and hardware components particularly the STM32, plays a critical role in the project. However, challenges often arise in testing and validating the functionality of these components, hindering students' progress and resulting in inefficiencies during the development of the robot car.

Currently, no standardized system exists to address the unique challenges of the MDP. This forces students to write separate code to test individual hardware components of the robot, leading to wasted time and effort.

To address this, the objective of this project is to develop a comprehensive Hardware/Software Test and Validation System tailored for MDP students. A desktop app named **MDPHelper** was developed to assist students in testing and validating their embedded systems. The app provides tools for hardware testing, communication with STM32, flashing custom firmware to the STM32, and visualizing real-time data from connected hardware components.

The system was developed using .NET MAUI for the frontend, with a .NET Core and MongoDB backend to manage firmware files and app updates. This solution simplifies the testing and validation process, enabling students to focus on the design and integration aspects of their projects while improving overall efficiency and collaboration.

Acknowledgement

First and foremost, I would like to express my deepest gratitude to Mr. Oh Hong Lye for his invaluable guidance, support, and insightful suggestions throughout my Final Year Project. His advice has been instrumental in shaping this project into a meaningful and impactful tool for students undertaking the MDP.

I would also like to extend my sincere appreciation to my peers who contributed to this project, whether through participating in surveys or providing feedback. Your time and effort have been invaluable, and I truly appreciate your willingness to support my work and its development.

Table of Contents

List of Figures

Acronyms

MDP	Multidisciplinary Design Project
STM32	STMicroelectronics 32-bit Microcontroller (STM32F407VET6)
App	Application
MAUI	Multi-platform App UI
XAML	Extensible App Markup Language
UART	Universal Asynchronous Receiver-Transmitter
HTTP	HyperText Transfer Protocol
I2C	Inter-Integrated Circuit
SPI	Serial Peripheral Interface
API	App Programming Interface
HAL	Hardware Abstraction Layer
GUI	Graphical User Interface
PWM	Pulse Width Modulation
RPM	Rotation Per Minute

Chapter 1: Introduction

1.1 Background

MDP challenges student teams to develop a robotic car system that can autonomously complete two tasks within a 10-week period:

1. **Obstacle Navigation and Image Recognition:** Navigate through obstacles in a designated area while capturing and recognizing images placed on those obstacles.
2. **Shortest Path Challenge:** Move toward two obstacles, recognize the direction of arrow images placed on them, and navigate left or right based on these arrows. Finally, return to the original starting point.

The teams then compete based on their performance, measured by completion time and the number of recognized images in these two tasks. A critical element of the robotic system is the STM32, which handles essential low-level commands and interacts with the robot's hardware[2]. STM32 plays a pivotal role in the robot's overall functionality[2], making testing its integration with both hardware and software components crucial.

1.1.1 Challenges Faced by Students Handling STM32 in MDP

A major challenge for students handling the STM32 was determining whether issues originated from the uploaded software or the robot's hardware. When issues arose, students struggled to identify whether the problem lies in the hardware or the software implementation. For example, a motor malfunction could result from either improper coding logic or a faulty motor driver circuit. Without a structured way to test both aspects, students are forced to conduct time-consuming trial-and-error investigations, such as rewriting sections of code or manually inspecting hardware components. This troubleshooting process led to significant delays, as students repeatedly alternate between testing software fixes and verifying hardware functionality.

In line with this, several peers provided insights into the difficulties they encountered during the development of STM32-based systems. **Appendix A** contains the verbatim responses from students, shedding light on their experiences with troubleshooting and identifying whether issues stemmed from software or hardware. As noted by one of the respondents, "The biggest issue was debugging the software on the STM32 and determining whether the issue was with the uploaded code or the robot's hardware". Another respondent shared, "One of the main problems I faced was trying to troubleshoot the robot's hardware without being sure if the issue was related to software. It took a lot of time to isolate the problem". These responses reflect the widespread nature of this challenge among students.

Moreover, research consistently shows that the absence of a structured testing strategy in embedded systems development can lead to higher failure rates, longer project timelines, and increased debugging complexity [3], [4], [5]. Addressing these challenges through a

dedicated validation system would significantly enhance students' ability to diagnose issues efficiently and progress with confidence.

1.2 Scope and Objectives of Project

The scope of this project covers the development of a desktop app, MDPHelper, aimed at simplifying the STM32 testing process for students involved in MDP. The system provides a user-friendly interface that supports firmware flashing, peripheral testing, and real-time data visualization, thereby eliminating the need for students to manually write test code for each component.

The backend infrastructure supports core functionalities such as firmware management, app updates, and communication with the STM32.

The primary objectives of this project were:

- **Providing a User-Friendly Testing Tool:** To deliver a practical and accessible platform that reduces the technical barrier for students when working with STM32.
- **Streamlining the Testing Process**: Automating and integrating testing features into a unified platform to reduce the need for manual test code development.
- **Enhancing Learning Efficiency**: Allowing students to focus on high-level system integration and design rather than spending time on repetitive debugging and low-level testing tasks.
- **Reducing Time Delays**: Minimizing inefficiencies caused by trial-and-error debugging, enabling students to complete their tasks within the project timeline.
- **Providing a Scalable and Extensible Foundation**: Establishing a framework that can be expanded in future iterations to include more features.

Ultimately, this project aims to empower MDP students with a reliable and accessible tool that simplifies testing process, improves learning outcomes, and serves as a foundation for future development.

1.3 Project Timeline

The project spans two semesters, starting from August 2024 and concludes in May 2025. The project is divided into 4 main phases:

- Planning and Design
- App Development
- Testing and Documentation
- Project Presentation

Hardware/Software Test and Validation System for MDP	Aug-24				May-25					
PROJECT NAME	START DATE				END DATE					
	AY 2024/2025 Semester 1					AY 2024/2025 Semester 2				
Tasks	**Aug**	**Sep**	**Oct**	**Nov**	**Dec**	**Jan**	**Feb**	**Mar**	**Apr**	**May**
Planning and Design										
Research method to implement the system										
Design application architecture										
System Architecture Design										
Design the individual components (UI, firmware, backend)										
App Development										
Develop core functionality: flashing STM board										
Develop general user interface										
Develop a server to retrieve/store firmware										
Research methodology for h/w testing										
Implement hardware testing in the app (embedded Programming)										
Initial testing and debugging										
Testing and Documentation										
Documentation: User guide & technical documentation										
Conduct UX testing										
optimize based on initial testing results										
Project Demo										
Launched Beta version and collect user feedback										
Analyze feedback results and make adjustments										
Project Presentation										
Interim Report Submission										
Final Report Submission										
Amended Final Report Submission										
Oral Presentation										

Figure 1: Project Timeline

1.4 Contributions

A summary of key contributions made in this project is as follows:

- **Development of an Integrated Full-Stack Testing Tool:**

  Designed and implemented a cross-platform desktop app, MDPHelper, that streamlines the testing process. The app supports firmware flashing, real-time data visualization, and peripheral control through a user-friendly GUI. It also incorporates backend features such as firmware management and automatic app updates to ensure long-term maintainability.

- **Custom Firmware for STM32 Communication and Validation:**

  Developed and deployed custom firmware on the STM32 to enable stable UART communication with the desktop app. The firmware supports sensor data acquisition and peripheral control, enabling real-time monitoring and functional validation of hardware components.

- **Simplification of Manual Hardware Testing for MDP Students:**

  Reduced the need for students to write individual test scripts by introducing a menu-based interface that allows selection and execution of hardware tests. This approach improves efficiency, lowers the learning curve for hardware interaction, and minimizes trial-and-error debugging.

- **Foundation for Future Enhancements:**

  The current implementation, though focused on functional validation, provides a scalable foundation for future features such as automated firmware code testing, result verification, RPM accuracy checks, and data logging for in-depth analysis.

1.5 Organization of Report

Chapter 1: Introduction – This chapter introduces the background of the project, outlines its objectives and scope, project timeline and highlights the contributions made throughout the course of the project.

Chapter 2: Literature Review – This chapter reviews existing test and validation systems, explores various embedded system testing approaches, and discusses the challenges faced in the field of hardware/software testing. It also details how this project addresses these challenges with its solution.

Chapter 3: System Design – This chapter describes the system architecture, detailing the hardware and software design, communication protocols, and database structure. It also covers the requirements, including functional and non-functional requirements, and includes use case diagrams and activity diagrams.

Chapter 4: Implementation – This chapter provides an overview of the system's implementation, discussing both frontend and backend development. It covers the API, database, firmware, and updater app optimization, as well as how the peripherals and sensors are integrated.

Chapter 5: Results and Discussion – This chapter evaluates the system's performance, discussing flashing speeds and database efficiency. It also includes a detailed analysis of the user feedback collected, as presented in **Appendix C**. Finally, the chapter outlines key limitations identified in the current implementation.

Chapter 6: Conclusion and Future Work – This chapter summarizes the key outcomes of the project and proposes future enhancements to improve usability, accuracy, and scalability. It also addresses the limitations discussed in Chapter 5.

Chapter 2: Literature Review

2.1 Overview of Existing Test and Validation Systems

Existing test and validation systems in embedded hardware and software development primarily focus on hardware testing frameworks and software debugging tools. While these systems are widely adopted in various domains, they lack the specificity required for the MDP, where hardware and software components must be validated within an integrated testing environment.

One prevalent category of test systems involves dedicated hardware testing kits designed for evaluating individual sensors and actuators. Commercially available solutions, such as Arduino sensor kits and Raspberry Pi HATs, provide pre-configured tools for testing components like ultrasonic sensors, motors, and communication modules. These kits are beneficial for general-purpose apps and rapid prototyping but present significant integration challenges when working with STM32, which require additional programming and interface adaptation. Furthermore, such kits typically lack comprehensive validation frameworks that can systematically assess the performance of multiple interacting hardware components within a unified system [5].

Another category includes embedded software debugging tools, such as Keil uVision, which provide step-by-step code execution, breakpoints, and hardware simulation for STM32 development [6]. While these tools are indispensable for debugging firmware, they primarily focus on software logic validation rather than hardware functionality testing. Additionally, these debugging environments often require a steep learning curve, making them less accessible to students with limited embedded systems experience.

Beyond hardware debugging, robotic simulation platforms like Gazebo and Webots offer virtual environments for testing autonomous system behaviour and algorithm performance [7]. These platforms enable users to simulate control strategies, path planning, and sensor fusion techniques, reducing the need for early-stage physical testing. However, their primary limitation lies in hardware validation—they do not provide real-world feedback on physical constraints, sensor accuracy, or actuator performance, which are critical factors in MDP.

2.2 Embedded System Testing Approaches

Embedded systems, such as those based on STM32, require thorough validation to ensure hardware components function reliably before software integration. Various testing methodologies are employed to assess performance and detect faults, including manual hardware testing, software-based simulation, and automated hardware validation.

2.2.1 Manual Hardware Testing (Current Solution)

Manual hardware testing involves direct interaction with embedded components to observe their performance. In this project, individual sensors, actuators, and communication modules are tested through controlled input conditions and physical measurements.

Advantages:

- Straightforward implementation with minimal setup requirements.
- Direct observation of hardware behaviour, allowing real-time feedback.

Disadvantages:

- Time-consuming for large-scale testing.
- Prone to human error, as results depend on manual operation.
- Limited scalability, making automation difficult for repetitive tasks.

Application in This Project:

- Verifying sensor responses under various conditions.
- Testing actuator outputs through direct observation.
- Checking communication module integrity and data transmission accuracy.

Manual testing is a practical approach for small-scale validation but becomes inefficient for large-scale or long-term monitoring.

2.2.2 Software-Based Simulation Testing

Software simulation tools offer virtual environments for testing sensor interactions, control algorithms, and embedded software logic before deploying to physical hardware [8].

Advantages:

- Enables early detection of software issues before deploying on hardware.
- Reduces the risk of damaging physical components during initial testing.
- Allows rapid iteration cycles for debugging control algorithms.

Disadvantages:

- Does not validate real-world sensor accuracy or actuator behaviour.
- May produce unrealistic results due to idealized simulations.
- Limited in detecting hardware faults such as sensor noise or wiring issues.

Relevance to This Project: While software-based simulations assist in algorithm validation, they do not replace the need for real-world sensor testing, which this project aims to address.

2.2.3 Automated Hardware Validation (Future Enhancement)

Automated hardware validation combines real-time data logging, sensor calibration, and automated test scripts to streamline embedded system testing. These systems use microcontroller-based test rigs to evaluate component performance systematically [9].

Advantages:

- Reduces human error by standardizing testing procedures.
- Improves efficiency by automating sensor and actuator validation.
- Enables real-time fault detection with logging and alerts.

Disadvantages:

- Requires additional development effort to implement automation tools.
- Initial setup can be costly depending on hardware requirements.

Future for This Project: While the current solution focuses on manual testing, future iterations will integrate automated scripts for sensor validation to improve efficiency. Implementing real-time data monitoring tools could provide better debugging and system diagnostics.

2.2.4 Comparison of Testing Approaches

Feature	Manual Testing (Current)	Software Simulation	Automated Hardware Validation (Future)
Accuracy	Prone to human error	Slow, requires manual checking	High accuracy with automated verification
Time Efficiency	Slow, requires manual checking	Fast, software-based	Faster than manual, real-time monitoring
Scalability	Not scalable for large projects	Scalable for multiple software tests	Scalable for multiple hardware components
Hardware Dependency	Requires actual hardware	No physical hardware needed	Requires hardware but automates testing

2.2.5 Justification for Manual Hardware Testing in This Project

Although automated testing offers significant advantages, it requires additional development time and resources. For this project, manual hardware testing is the most practical approach because:

- **Validation of Physical Components**: The primary objective of this project is to ensure the proper functioning of sensors and actuators before integrating the software. Manual testing allows for a hands-on verification of hardware performance in real-world conditions.
- **Early Development Stage**: Given that the project is still in its early phases, direct interaction with hardware offers immediate insights into the behaviour of physical components, allowing for rapid troubleshooting and adjustment.
- **Foundation for Future Automation**: While manual hardware testing is prioritized for the current phase, the insights gained through this approach will serve as the foundation for future iterations. Automated testing will be incorporated to improve efficiency and scalability in later stages.

This project prioritizes manual hardware testing as an immediate solution, while considering future enhancements to incorporate automated validation.

Chapter 3: System Design

3.1 System Architecture

The system architecture integrates hardware, software, and communication layers to enable efficient management of firmware updates, hardware interaction, and system diagnostics. A modular layered design was adopted to ensure scalability, maintainability, and ease of troubleshooting.

The system is available in two versions:

1. **Local Version** – The app operates independently without requiring an internet connection. Firmware files are stored directly on the user's device, eliminating the need for an online connection to access firmware updates.
2. **Online Version** – The app is connected to a backend server. Firmware files are stored in a database, allowing the app to retrieve firmware updates and receive app updates from the backend.

Both versions share a similar layered architecture but differ in how firmware files are managed. **While the local and online versions share core design principles, the remainder of this report assumes the online version as the primary implementation. Therefore, discussions on design, requirements and system implementation will focus on the online version's architecture.**

3.1.1 Local Version Architecture

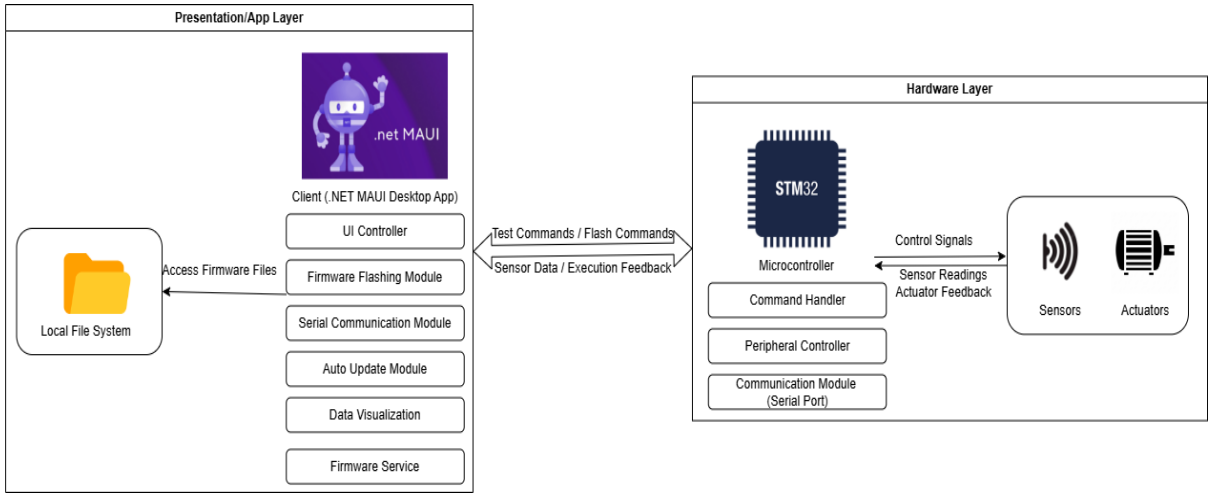

Figure 2: System Architecture (Local)

The local version operates without requiring an internet connection, as firmware files are stored directly on the user's device. This eliminates the need for a backend database, simplifying deployment since no server infrastructure is required.

The architecture consists of the following components:

- Presentation/App Layer (**.NET MAUI**) – Provides a graphical interface for users to select firmware, communicate with the microcontroller, and visualize test results. It also facilitates interaction with the local file system to access stored firmware files.
- Hardware Layer (**STM32**) – Executes the uploaded firmware and controls connected peripherals, such as sensors and actuators. It communicates with the desktop app via serial communication to receive commands and return test data.
- Local File System – Stores firmware files directly on the user's device, ensuring offline accessibility and fast file retrieval during firmware flashing.

This architecture ensures a lightweight and efficient system, allowing users to manage firmware updates and hardware testing entirely offline.

3.1.2 Online Version Architecture

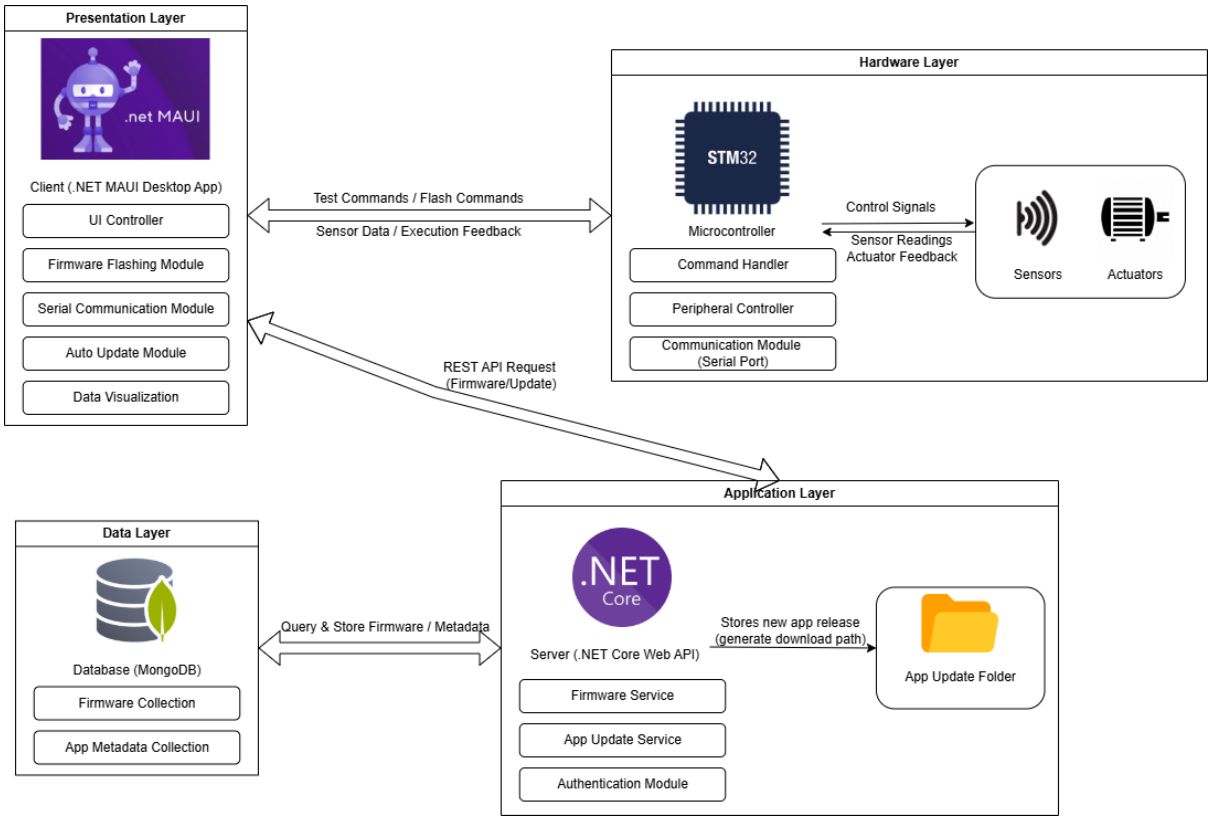

Figure 3: System Architecture (Online)

The online version connects the desktop application to a remote backend infrastructure, allowing real-time access to firmware files and application updates. Firmware binaries and metadata are stored in a cloud-based MongoDB database, and the desktop client interacts with a .NET Core Web API server to manage firmware flashing, version control, and application updates. The architecture consists of the following layers:

- Presentation Layer (**.NET MAUI**) – Provides a user interface for managing firmware updates, issuing test commands to the STM32 hardware, and visualizing sensor data. The app sends REST API requests to the backend to fetch or upload firmware and version metadata.

- App Layer (**.NET Core**) – Handles backend operations such as serving firmware files, responding to version queries, and managing application updates. It also retrieves and stores data in the database and provides access to the latest app executable stored on the server.
- Hardware Layer (**STM32**) – Executes embedded firmware, controls connected peripherals (motors, sensors, etc.), and responds to test commands from the app layer.
- Data Layer (**MongoDB**) – Stores firmware metadata and updated app information ensuring efficient data retrieval and storage.

This layered architecture enables centralized firmware management, simplified version control, and remote update functionality, all while maintaining a clean separation of concerns for easier system maintenance and future scalability.

3.2 Software Design

3.2.1 Frontend Design

MDPHelper was developed using .NET MAUI to provide a user-friendly interface for managing hardware testing and firmware updates. A desktop app was chosen instead of a web-based solution due to the need for direct USB/serial communication, which is not feasible in standard web browsers due to security consideration.

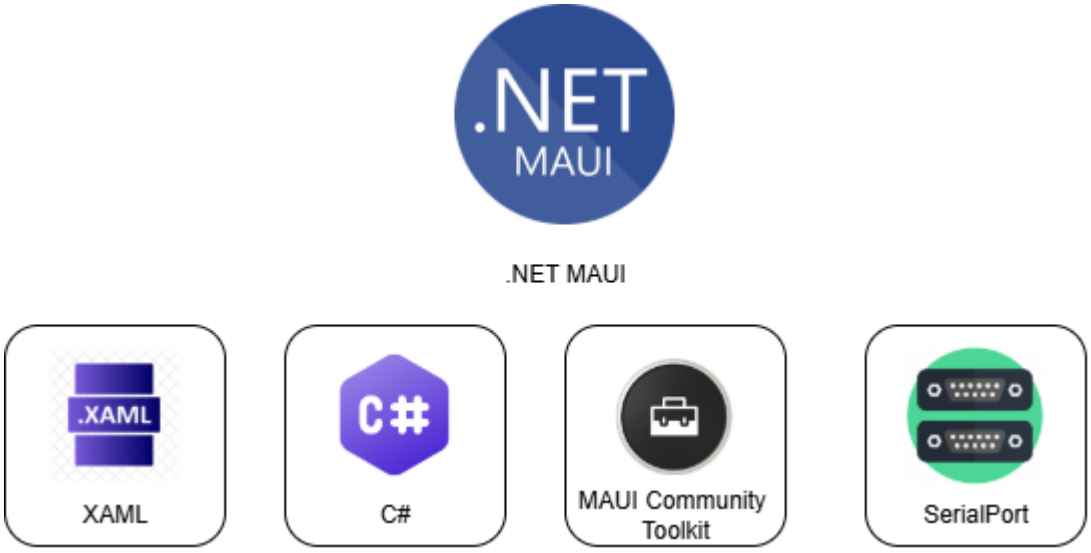

Figure 4: Frontend Technology Stack

Technology Stack and Tools

Tool/Framework	Purpose
.NET MAUI	Provides a cross-platform UI framework, allowing seamless deployment on Windows and macOS while maintaining native performance.
XAML	Defines the UI structure and layout, ensuring a responsive, scalable, and maintainable interface.
C#	Used as the primary programming language to manage application logic, handle user interactions, and coordinate communication with the backend.

MAUI Community Toolkit	Offers prebuilt UI controls, animations, and behaviors, reducing development time while improving the user experience.
SerialPort	Enables direct USB/serial communication between STM32 and desktop app to handling firmware flashing, real-time diagnostics, and sensor data visualization.

Key Features of the Frontend

The MDPHelper app serves as the primary interaction point for users, allowing them to efficiently manage hardware testing. Key functionalities include:

- **Intuitive Test Configuration:** Provides an interface for configuring test parameters, initiating test sequences, and visualizing real-time sensor data from the STM32.
- **Firmware Management:** Simplifies retrieval and flashing of firmware.
- **User-Friendly Navigation:** Features a menu-driven system that enables users to easily navigate.
- **Real-Time Data Visualization:** Displays real-time readings from the STM32, aiding in hardware validation.
- **Integrated Support & Troubleshooting:** Provides quick access to user guides, QnAs, and debugging tools within the app.
- **Automated Updates:** Incorporates an automated update mechanism that keeps the app synchronized with the latest backend-released features and fixes.

3.2.2 Backend Design

The backend plays a pivotal role in centralizing the management of firmware files and system updates. Its robust architecture ensures a secure and scalable infrastructure that supports seamless interactions with the frontend app. Key functionalities of the backend include managing firmware storage, facilitating app updates, and providing an authentication layer to ensure secure access.

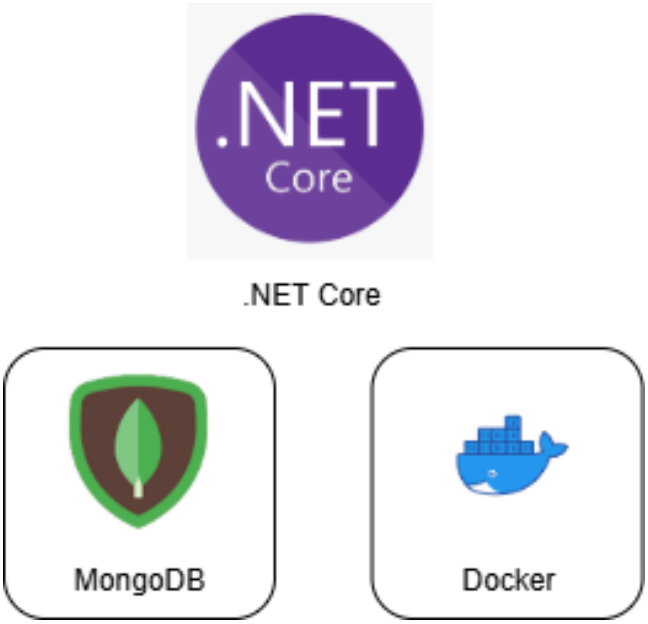

Figure 5: Backend Technology Stack

Technology Stack and Tools

Tool/Framework	Purpose
ASP.NET Core	Powers the backend API server, providing high performance and cross-platform support for scalable web services.
MongoDB	A NoSQL database used for storing firmware metadata, user configurations, and system logs. Its flexible schema supports efficient data retrieval and scalability.
Docker	Containerizes the backend, ensuring easy deployment on cloud-based or local infrastructure, while enabling future scalability via microservices.

Key Features of the Backend

The backend is designed to manage the core tasks of firmware storage, app updates, and user authentication, with the following key features:

- **Firmware Management:** Firmware files are stored in a database, allowing version tracking, retrieval, and seamless updates.
- **API Server:** The backend provides RESTful API endpoints for secure communication between the frontend app and the database.
- **App Update Distribution:** Hosts and manages updated versions of the desktop app, enabling users to download the latest releases directly from the system.
- **Asynchronous Request Handling:** Optimized to handle multiple concurrent requests, ensuring system responsiveness even under high traffic.
- **Scalable Deployment:** Containerized using Docker, supporting deployment on both cloud-based and local environments.

3.2.3 Firmware Design

The firmware on the STM32 is responsible for managing real-time operations, peripheral control, and communication with external devices. Designed for efficiency and reliability, the firmware ensures seamless interaction between the hardware components and the desktop app.

Figure 6: Firmware Technology Stack

Technology Stack and Tools

Tool/Framework	Purpose
STM32CudeIDE	Provides an integrated development environment for coding, debugging.
HAL	Simplifies peripheral interaction and hardware control, ensuring portability across different STM32 models.
FreeRTOS	Implements task scheduling for real-time operations, ensuring efficient multitasking and resource management.
ICM-20948	Interfaces with 9-axis motion sensor through a C library, enabling accurate and real-time access to accelerometer, gyroscope, and magnetometer readings for motion tracking.

Key Features of the Firmware:

- **Real-time Task Management**: Uses a bare-metal loop or RTOS to handle sensor readings, motor control, servo control and data processing.
- **Peripheral Control**: Interfaces with external sensors and actuators using UART, I2C, and SPI protocols.
- **Command Processing**: Executes commands received from the desktop app, enabling automated hardware validation and remote control.
- **Data Formatting**: Prepares sensor data for transmission, converting raw readings into structured formats for backend storage or real-time visualization.

3.2.4 Data Communication

This section details how different system components exchange data, ensuring real-time responsiveness and efficient system operation. The desktop app facilitates both communication and firmware flashing processes for the STM32.

STM32 and Desktop App Communication (UART & ST-LINK):

Figure 7: UART Connection

- **UART Communication (Serial):** The desktop app communicates with the STM32 via a UART connection, allowing command-based interactions for hardware testing, data retrieval, and firmware flashing over a serial interface.

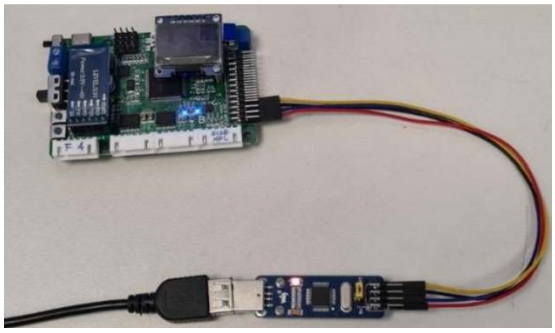

Figure 8: ST-Link Connection

- **ST-LINK Programmer (USB):** Alternatively, users can flash the STM32 firmware using the ST-LINK interface, which connects via USB. This method provides a faster and more reliable flashing process.

Both flashing methods leverage the open-source project [10], which supports programming and debugging STM32. By providing both UART and ST-LINK flashing options, the system offers users the flexibility to choose the most suitable method based on their hardware setup and requirements.

Backend and Frontend Communication (HTTP API):

- **HTTP-Based API Requests:** The frontend retrieves firmware updates and app data from the backend via HTTP requests.
- **Firmware Delivery:** The backend hosts firmware files, allowing the desktop app to fetch and flash the latest versions onto the STM32.

3.3 Hardware Design

3.3.1 STM32

Figure 9: STM32

The STM32 serves as the central processing unit (CPU) of the system, managing real-time operations and coordinating interactions with external hardware components. It interfaces directly with sensors, actuators, and the desktop app, ensuring smooth and efficient system functionality. Equipped with multiple communication protocols, including UART, I2C, and SPI, the STM32 guarantees reliable data exchange between hardware components and the software system.

3.3.2 Sensors and Peripherals

The system integrates a variety of sensors and peripherals to enable data collection, control, and monitoring. Key components include:

Figure 10: ICM-20948

- **Motion Sensors (ICM-20948):** These sensors provide real-time motion and orientation data, critical for apps requiring spatial awareness and dynamic tracking.

Figure 11: HC-SR04

- **Ultrasonic Sensor (HC-SR04):** Used for distance measurement, allowing the system to detect obstacles or map surroundings.

Figure 12: SHARP 2Y0A21

- **Infrared Sensor (SHARP 2Y0A21):** Provides proximity sensing, essential for apps like object detection or environmental monitoring.

Figure 13: JGB37-520

Figure 14: TD-8120MG

- **Motors (JGB37-520) & Servos (TD-8120MG):** Actuators controlled by the STM32 to perform mechanical movements, such as rotating, positioning, or adjusting parts based on sensor input or user commands.

The sensors feed data back to the STM32, which processes the information and communicates it to the desktop app or controls the actuators accordingly.

3.3.3 Communication Protocols

The communication layer in the hardware architecture utilizes several protocols to ensure seamless interaction between STM32 and peripheral:

- **I2C:** A low-speed communication protocol used to interface with onboard sensors. It allows for multiple devices to communicate with the STM32, minimizing wiring complexity.
- **SPI:** A higher-speed protocol used for communication with peripherals requiring fast data transfer, ensuring high-performance interactions.

3.3 Requirements

3.3.1 Functional Requirements

The app must allow the user to perform the following:

- Fetch preloaded firmware files from the system server using an API.
    - The file format must be .bin or .hex.
- Flash a firmware file into STM32.
    - Users can select either a preloaded firmware file from the system server or a custom firmware file from their device.
    - The file format must be .bin or .hex.
    - The flashing process must utilize either a connected ST-LINK programmer or UART.
    - Allow users to cancel the flashing process at any time.
- Establish data communication with the connected STM32.
    - Enable users to send data to and receive data from the STM32.
- Execute hardware tests on the STM32 after flashing.
    - Display real-time logs of the test results in the app.
    - Provide a graphical representation of the test results.
- Manage test result logs.
    - Allow users to delete old or unwanted test result logs.
    - Provide an option to reset the test results displayed in the graph.
- Check for application updates on startup

    - The app must connect to the backend server to check for the latest available version.

- Download and install updated app
    - If an update is available, the app should download the updated .exe or .pkg file automatically.
    - The standalone process should be triggered to replace the existing executable and launch the updated version.
    - The process must clean itself up after the update is complete.

3.3.2 Non-Functional Requirements

The app must be able to meet the following requirements:

- **Performance Requirements**
    - App must not take longer than 2 seconds to fetch preloaded firmware files from the system server.
    - Flashing the STM32 with a firmware file must not take longer than 1 minutes for files up to 500 KB in size.
    - App must not take more than 1 second to display real-time logs during hardware tests.
    - Graphical representation of test results must update in real-time without noticeable delay.
- **Scalability Requirements**
    - App must be able to handle up to 100 concurrent requests for fetching firmware files from the system server.

- **Usability Requirements**
    - App must ensure that all major functionalities (e.g fetch firmware, flashing) can be accessed within 2 clicks.
    - Error messages must be displayed in clear, user-friendly language, with actionable instructions.
    - App interface must support resolutions commonly used for desktops and laptops, ensuring easy navigation.
- **Reliability Requirements**
    - App must recover gracefully within 5 seconds from a lost connection during the flashing process.
    - Test results and logs must persist after an unexpected app crash.

3.3.3 Use Case Diagram

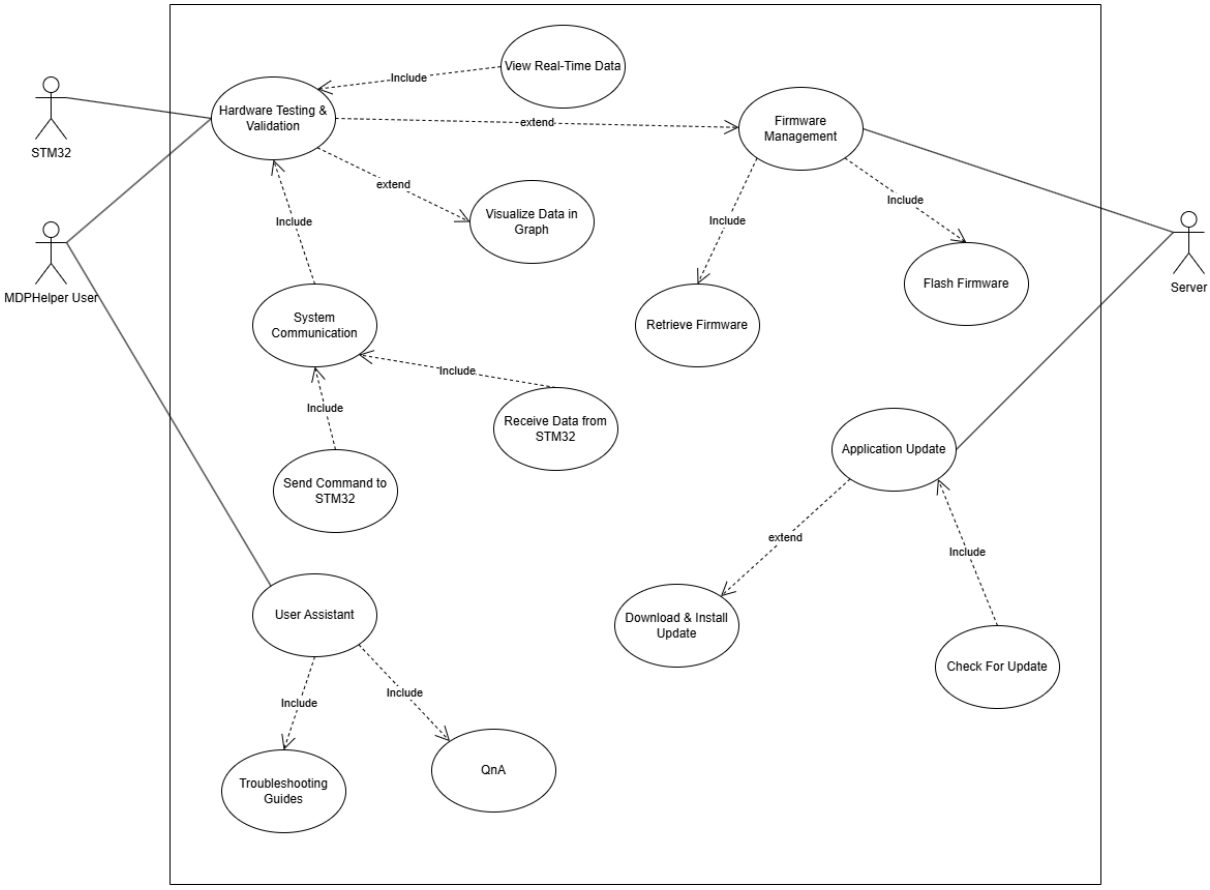

Figure 15: Use Case Diagram

3.3.3.1 Use Case Table

The detailed use case descriptions for each use case are provided in **Appendix B**.

Use Case ID	**Use case**
CASE001	Hardware Testing & Validation
CASE002	System Communication
CASE003	Firmware Management
CASE004	App Update
CASE005	User Assistance

3.3.4 Activity Diagram

Firmware Flashing Workflow:

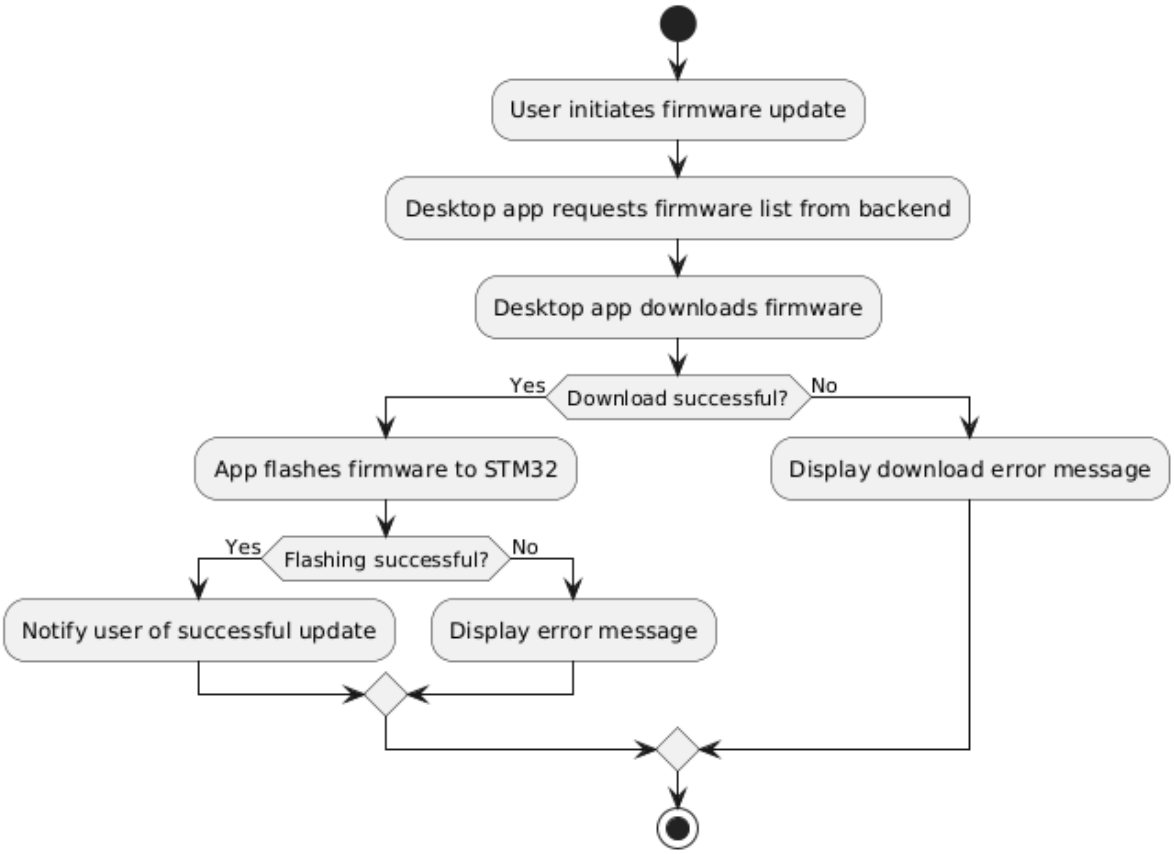

Figure 16: Firmware Flashing Workflow

Hardware Testing Workflow:

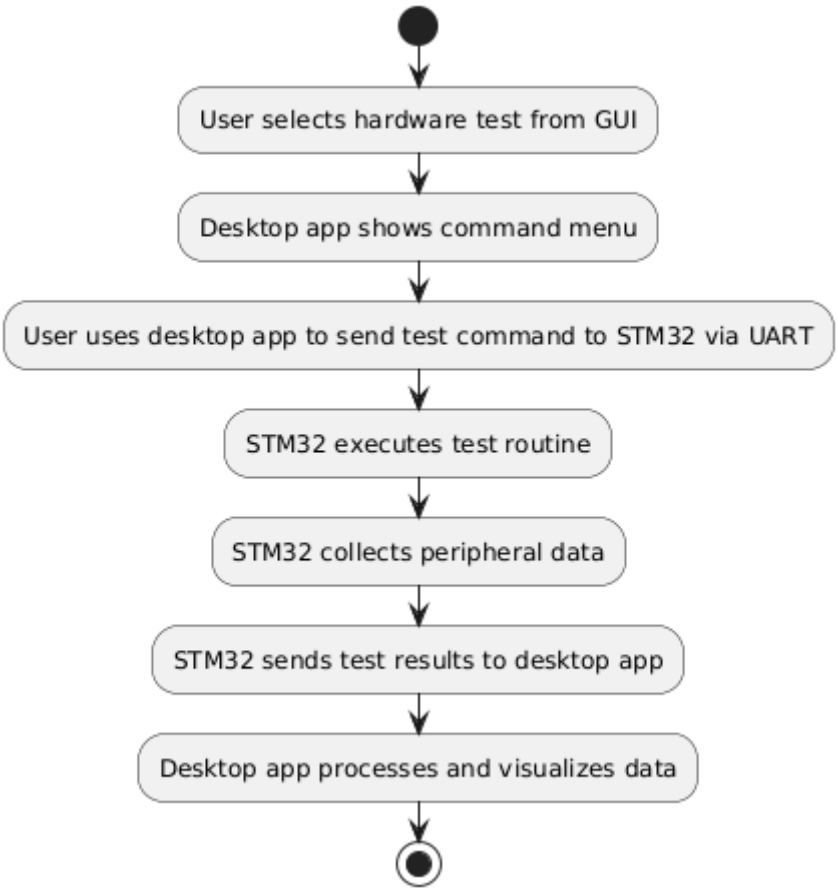

Figure 17: Hardware Testing Workflow

App Update Workflow:

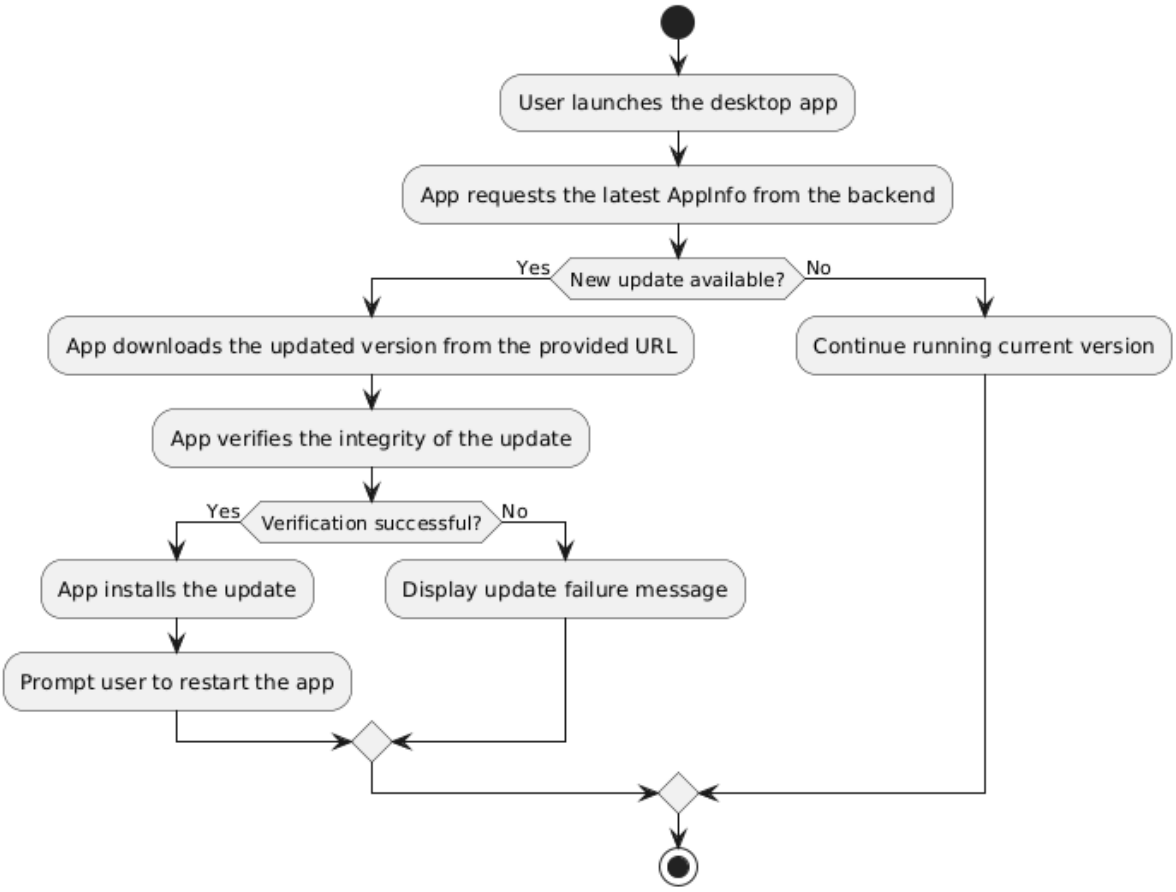

Figure 18: App Update Workflow

Chapter 4: Implementation

4.1 Frontend Implementation

This section outlines the development of the key pages in the app.

4.1.1 GUI Development

The GUI is designed to be intuitive and user-friendly, enabling seamless interactions with the system. The GUI consists of several key pages, each focused on specific user tasks.

4.1.1.1 Landing Page

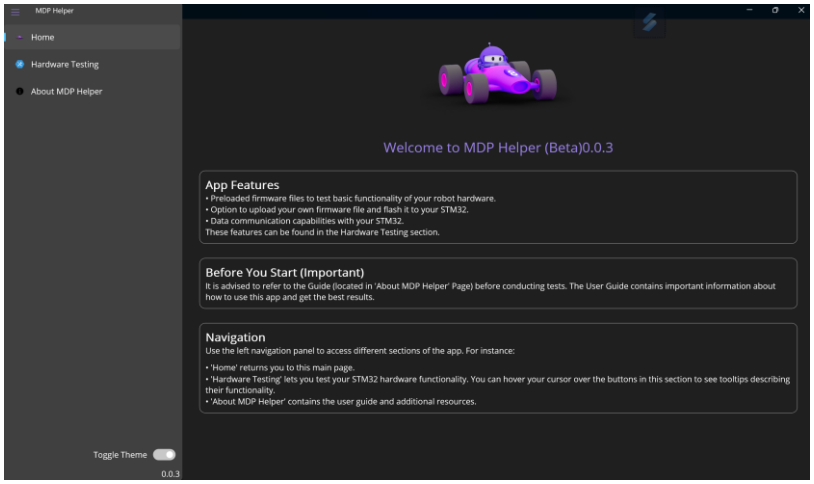

Figure 19: Landing Page

The Landing Page serves as the entry point, providing an overview of the system and easy access to other sections like Hardware Testing Page and About Page.

4.1.1.2 Hardware Testing Page

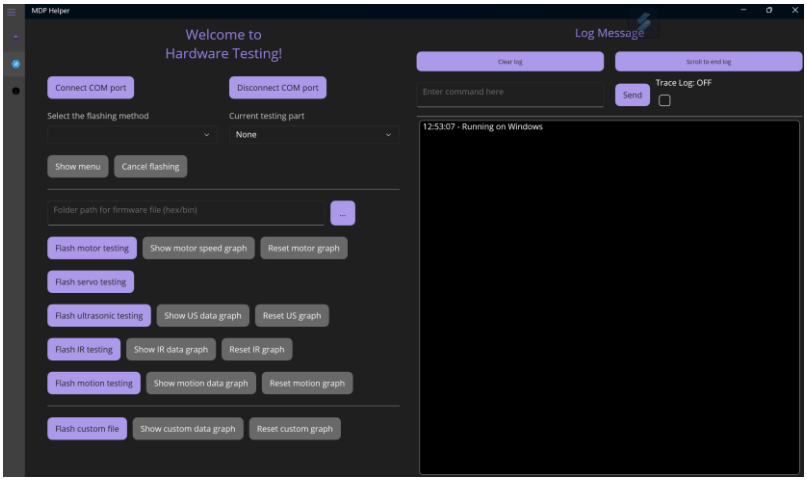

Figure 20: Hardware Testing Page

The Hardware Testing Page enables users to connect/disconnect COM ports, flash firmware, and run tests on motors, servos, ultrasonic sensors, and flash custom firmware. It displays real-time data graphs for performance monitoring and troubleshooting.

4.1.1.2.1 Hardware Testing Page Components

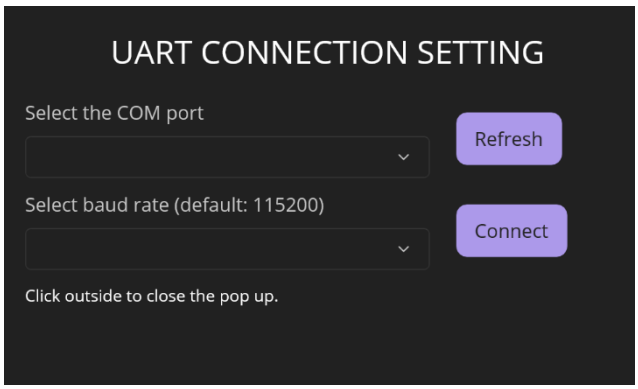

Figure 22: UART Connection Setting

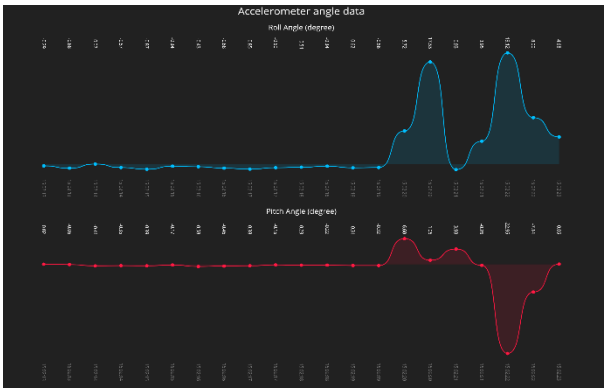

Figure 21: Data Visualisation

4.1.1.3 About Page

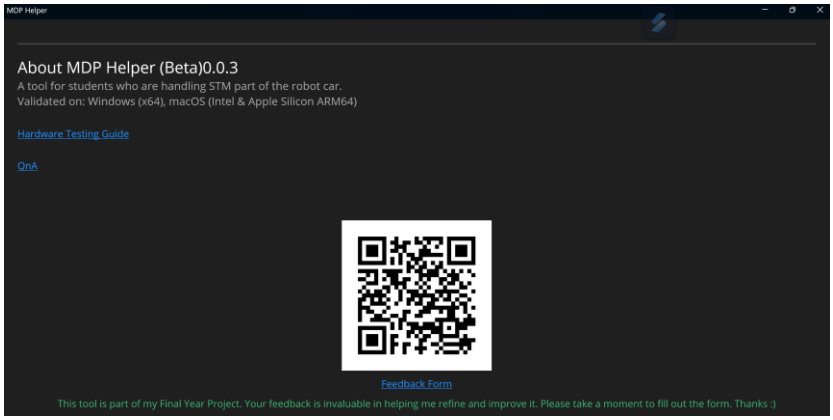

Figure x: About Page

The About Page provides details about the MDPHelper, including its version and platform compatibility. It also includes links to the Hardware Testing Guide, QnA, and a Feedback Form for user suggestions and improvements.

4.1.1.3.1 Hardware User Guide

Figure 23: Hardware Testing User Guide

4.1.1.3.2 QnA

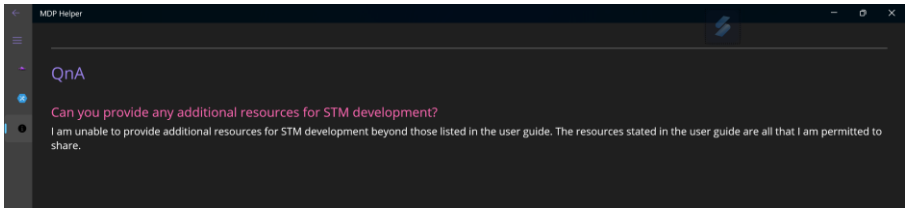

Figure 24: QnA

4.2 Backend Implementation

This section explains the implementation of the API and Database layers of the backend.

4.2.1 API Implementation

The backend exposes several API endpoints to handle requests related to app information and firmware management. Below are the API endpoints developed:

AppInfo Endpoints

Action	Endpoint	Description
POST	/api/appinfo/upload	Uploads new app info
GET	/api/appinfo	Retrieves app information
GET	/api/appinfo/version/{version}	Retrieves specific version of the app
GET	/api/appinfo/latest	Retrieves the latest version of the app
DELETE	/filetype/{fileType}version/{version}	Deletes app info by file type and version
DELETE	/api/appinfo/deleteAll	Deletes all app info

Firmware Endpoints

Action	Endpoint	Description
GET	/api/firmware	Retrieves all firmware
POST	/api/firmware	Uploads new firmware
GET	/api/firmware/name/{firmwareName}	Retrieves a specific firmware by name
DELETE	/api/firmware/name/{firmwareName}	Deletes specific firmware by name
DELETE	/api/firmware/deleteAll	Deletes all firmware

4.2.2 Backend Database Implementation

The backend is connected to a MongoDB database, which stores essential data related to firmware and app information. Below are the two key schemas used in the database:

4.2.2.1 Firmware Schema

The Firmware schema stores information about firmware files used by the system. It includes metadata like the firmware name, author, and the file data. This schema enables the system to manage firmware versions and facilitate updates and retrieval when required.

Key Fields in the Firmware Schema:

Field Name	Description
FirmwareName	The name of the firmware file.
Author	The creator or uploader of the firmware.
FileData	The binary data of the firmware file.

4.2.2.1 AppInfo Schema

The AppInfo schema stores metadata about app files, including details such as the file name, version, download URL, and author. This schema enables tracking of different app versions and ensures that the latest version can be downloaded for updates.

Key Fields in the AppInfo Schema:

Field Name	Description
FileName	The name of the app file.
FileType	The type of the app file (e.g., ".exe", ".pkg").
Version	The version number of the app file.
UpdatedDate	The date the app information was last updated.
DownloadUrl	The URL where the app file can be downloaded.
Author	The creator or uploader of the app.

The database supports CRUD operations (Create, Read, Update, Delete) for both Firmware and AppInfo collections. These operations enable the management of:

- **Firmware Files:** Storing new firmware, retrieving specific versions, and deleting outdated firmware.
- **App Information:** Uploading new app versions, retrieving metadata, and managing updates.

4.2.3 Optimization (Updater App)

Efficient update management is crucial for a seamless user experience and optimal system performance. To address this, the UpdaterApp was created — a lightweight, console-based utility that runs silently in the background, autonomously managing updates without requiring user interaction.

UpdaterApp Process:

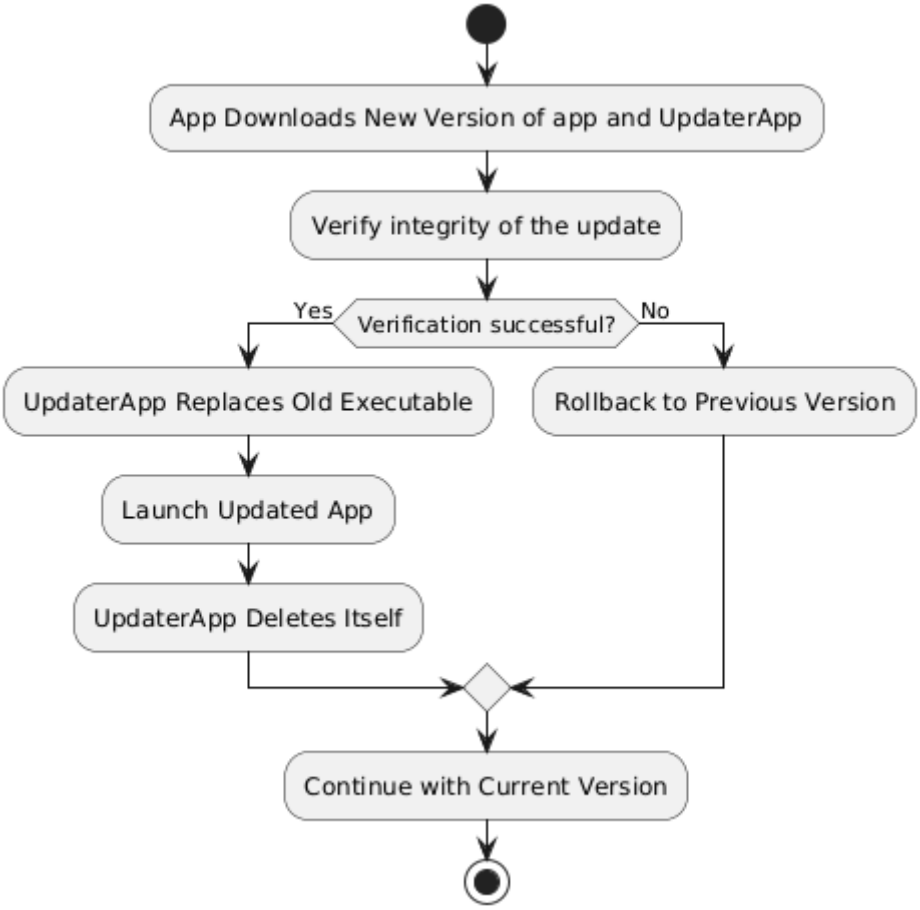

Figure 25: UpdaterApp Workflow

Benefits of the UpdaterApp:

- **Seamless Updates**: Operates quietly in the background, without interrupting the user.
- **Minimal Interaction**: Updates happen automatically, without requiring user intervention.
- **Safety**: The fail-safe mechanism ensures the app remains functional if the update fails.
- **Efficiency**: The separate update process ensures faster updates with minimal UI impact.

4.3 Firmware Implementation

The following outlines the general structure and key considerations during firmware integration:

The general Firmware Structure:

1. **Interrupts for Command Reception**:
   The firmware uses interrupts to receive commands from external devices, ensuring fast response times.
2. **Command Parsing**:
   After receiving the command, the firmware parses it to determine the required action.
3. **Task Execution**:
   Once a command is parsed, the corresponding task is executed, and a response is sent back via UART to acknowledge or send data.
4. **Queueing for Ongoing Tasks**:
   If a task is already running, subsequent commands are stored in a queue to ensure they are processed in order once the current task is completed.

Firmware Flow Diagram

Below is an illustration of the firmware flow, visualizing the command reception, parsing, task execution, and queueing process for ongoing tasks:

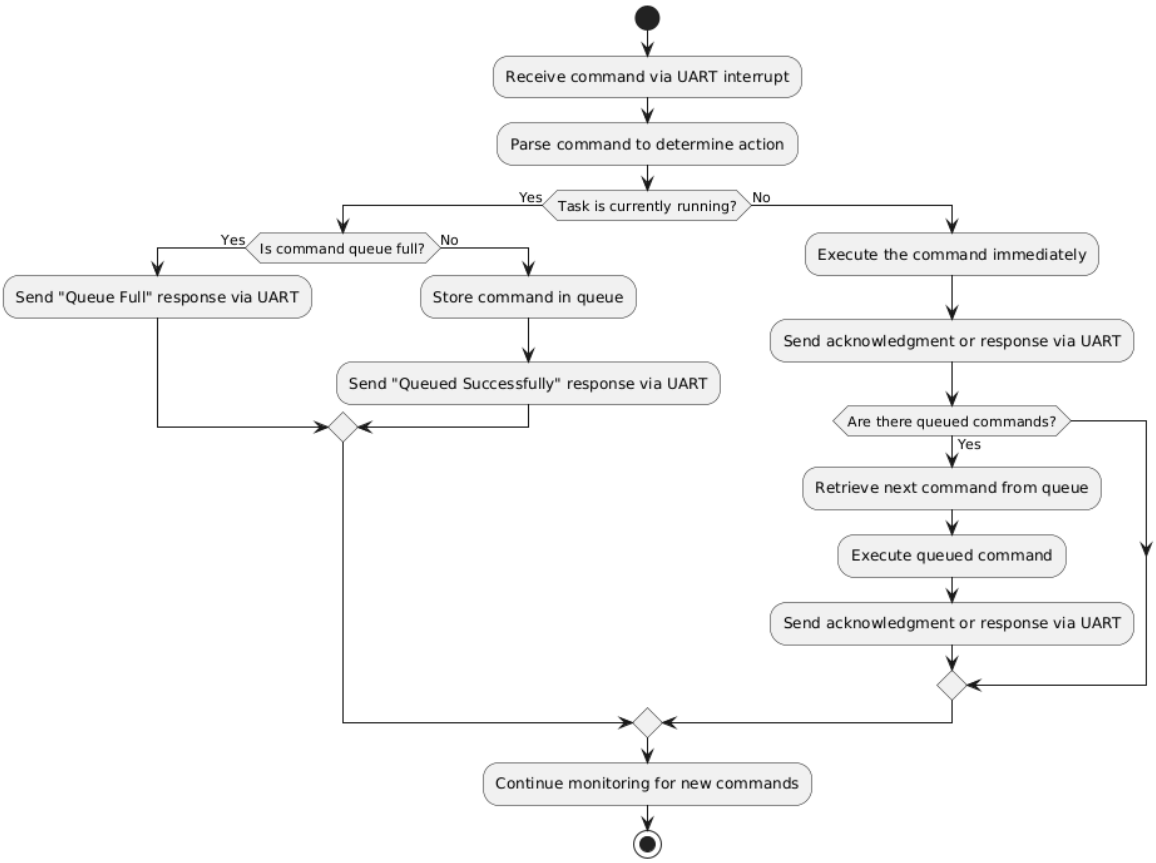

Figure 26: Firmware Workflow

Memory and Performance Optimization

The STM32 has **limited system resources**, with **256 KB to 512 KB of flash memory** and **64 KB to 128 KB of RAM**. To ensure efficient use of memory and maintain high performance, the following strategies were implemented:

- **Conditional Compilation**:
  The firmware uses preprocessor directives **(#ifdef)** to enable or disable features based on hardware configurations, rather than maintaining multiple versions of the firmware.
    - **Benefit**: This reduces code duplication, simplifies updates, and streamlines the development process.
- **Efficient Data Handling**:
  To ensure stable communication via UART and avoid buffer overflows, several data handling techniques were employed:
    - **Ring Buffers**: Implemented to store incoming UART data, efficiently managing high-speed communication and preventing data loss.
    - **Optimized Data Parsing**: Incoming data is processed in chunks, reducing memory usage and preventing large buffer allocations.

4.3.1 Peripherals and Sensor Integration

This section explains the integration of various peripherals and sensors, including motor and servo control via PWM and the handling of sensor data from motion sensors, ultrasonic sensors, and infrared (IR) sensors. Each of these components plays a key role in ensuring accurate and reliable control of the system.

Motor and Servo Control via PWM:

The STM32 uses PWM signals generated by its timers to control both the speed of motors and the position of servos. The duty cycle of the PWM signal determines the motor speed and servo position, providing precise control over the connected actuators.

1. **Motor Control via PWM**: The motor's speed is controlled by adjusting the duty cycle of the PWM signal. A higher duty cycle results in a higher motor speed. The duty cycle can be calculated using the formula:

$$Duty\ Cycle\ (\%) = \left(\frac{Pulse\ Width\ (CCR)}{Period\ (ARR)}\right) \times 100$$

Where:

- **CCR** (Capture/Compare Register) sets the pulse width.
- **ARR** (Auto-Reload Register) determines the total period of the PWM signal.

2. **Servo Control via PWM:** A servo's position is determined by adjusting the pulse width within a fixed 20ms period [11]. A pulse width of 1ms corresponds to 0°, and 2ms corresponds to 180°. The servo position can be calculated as:

$$Servo\ Position\ (degrees) = \frac{Pulse\ Width\ (ms)}{20} \times 180$$

Sensor Data Handling:

Sensors such as **accelerometers**, **gyroscopes**, **magnetometers**, **ultrasonic sensors**, and **infrared (IR) sensors** provide critical data that enables the system to make decisions based on the environment.

Motion Sensor (ICM-20948):

The **ICM-20948** is a 9-axis sensor that includes an accelerometer, gyroscope, and magnetometer. These sensors provide data on motion, orientation, and spatial tracking.

1. **Accelerometer Calculation**: The accelerometer measures acceleration along the X, Y, and Z axes, which can be used to calculate the tilt angle of the device:

$$Row\ angle(degrees) = \arctan\left(\frac{a_y}{\sqrt{{a_x}^2 + {a_z}^2}}\right) \times \frac{180}{\pi}$$

$$Pitch\ angle(degrees) = \arctan\left(\frac{a_x}{\sqrt{{a_y}^2 + {a_z}^2}}\right) \times \frac{180}{\pi}$$

$$Tilt\ angle(degrees) = \arctan\left(\frac{\sqrt{{a_x}^2 + {a_y}^2}}{a_z}\right) \times \frac{180}{\pi}$$

Where:

- a_x , a_y , a_z are the accelerometer data points along the X, Y, and Z axes.

2. **Gyroscope Calculation:** The gyroscope measures the angular velocity around the X, Y, and Z axes. The orientation change over time can be calculated using integration:

$$Row\ angle(degrees) = \int g_x\, dt$$

$$Pitch\ angle(degrees) = \int g_y\, dt$$

$$Yaw\ angle(degrees) = \int g_z\, dt$$

Where:

- g_x , g_y , g_z are the angular velocities around the X, Y, and Z axes, respectively.

To keep the angle values within the range of **-180° to 180°**, the following normalization formula is applied:

Formula for angle normalization:

$$Normalised\ Angle = fmod(Angle, 360.0)$$

If the result exceeds 180°, we subtract 360° to bring the value within the **-180° to 180°** range.

3. **Magnetometer (for Heading Correction):**

The magnetometer provides heading (yaw) information. The yaw angle is calculated using the arctan formula:

$$Heading\ (Yaw)\ Angle\ (degrees) = \arctan\left(\frac{m_y}{m_x}\right) \times \frac{180}{\pi}$$

Where:

- m_x , m_y are the magnetometer data points along the X and Y axes.

The magnetometer provides the absolute heading (yaw) angle, which is used to correct any drift in the yaw angle derived from the gyroscope. However, it can be susceptible to noise and interference from external magnetic fields, which may affect its accuracy.

Ultrasonic Sensor (HC-SR04) Integration:

The HC-SR04 ultrasonic sensor is used to measure distances to nearby objects. The sensor emits a burst of ultrasound at a frequency of 40 kHz and listens for the reflected signal. The time it takes for the pulse to return is proportional to the distance to the object.

Sensor Operation:

1. **Trigger**: A 10µs high pulse on the **Trig** pin triggers the ultrasonic burst.
2. **Echo**: The sensor outputs a high-level pulse on the **Echo** pin, and the width of this pulse is proportional to the time taken for the pulse to travel to the object and back.
3. **Distance Calculation**: The time for the pulse to return is measured, and the distance is calculated using the formula:

$$Distance\ (cm) = \frac{Time\ (us)}{2} \times 0.0343 cm/us$$

Where:

- **Time (µs)** is the duration the **Echo** pin stays high.
- Divided by **2** since the pulse travels to the object and back.
- **0.0343 cm/µs** is the speed of sound in air (approximately 343 m/s).

Infrared (IR) Sensor Integration:

The **IR sensor** is used to measure the distance to an object based on the reflected infrared light. The sensor's ADC (Analog-to-Digital Converter) readings are used to calculate the distance, utilizing a formula derived from the sensor's calibration.

Distance Calculation: The IR sensor outputs an ADC value, which is averaged over multiple samples to minimize noise. The distance is then calculated using the following formula:

$$Distance\ (cm) = \frac{IR\_CONST\_A}{\frac{IR\_data\_raw\_acc}{dataPoint} - IR\_CONST\_B}$$

Where:

- **IR_CONST_A** and **IR_CONST_B** are calibration constants.
- **IR_data_raw_acc** is the accumulated raw ADC value.
- **dataPoint** is the number of samples taken.

4.4 Testing and Validation

This section covers the methods and processes used to test the system, ensuring its reliability and functionality.

4.4.1 Firmware Testing

Firmware testing is crucial to ensure that the firmware functions as expected on the STM32. This involves verifying the proper operation of various peripherals, sensors, and actuators, ensuring that the system responds to inputs and outputs as designed.

Field Name	Description	Expected Result
PWM Motor Control	Test PWM signals controlling motor speed. Verify that the motor speed is adjustable via PWM duty cycle.	Motor speed changes according to the PWM duty cycle.
PWM Servo Control	Test PWM signals controlling servo position. Verify that the servo moves smoothly across the full 180° range.	Servo position corresponds to the PWM pulse width.
Motion Sensor	Verify data from the ICM-20948 motion sensor (accelerometer, gyroscope, magnetometer). Ensure proper readings of roll, pitch, and yaw.	Accurate tilt and orientation data for the system.
Ultrasonic Sensor	Test the ultrasonic sensor (HC-SR04) for distance measurement. Verify that the sensor returns the correct distance based on object proximity.	Accurate distance measurement to nearby objects.
IR Sensor	Test the IR sensor for proximity detection. Verify that the sensor detects objects at varying distances.	Correct detection of objects based on IR sensor readings.
Encoder Feedback	Verify that encoder feedback is accurately processed, and motor RPM is correctly calculated.	Correct motor RPM and direction determined by encoder data.

4.4.2 Software Testing

The software testing phase focuses on verifying the interaction between the MDPHelper app, the server, and the database. Testing ensures that the app can retrieve firmware files from the server, perform updates, and that the database stores and retrieves necessary information correctly.

Test Area	Description	Tools/Methods	Expected Outcome
MDPHelper App	Test the desktop app's ability to retrieve firmware files from the server.	MDPHelper App, Server Logs	App can retrieve firmware files.
Server Communication	Test server responses to ensure the app can fetch the correct firmware.	Network Tools (e.g., Wireshark), Server Logs	Server responds with the correct firmware file based on firmware name.
Database Functionality	Ensure the database stores and retrieves firmware and app metadata.	MongoDB, Queries	Data is accurately stored and retrieved from the database.
Firmware flashing	Verify the app can flash firmware to STM32.	MDPHelper App, Firmware Files, STM32	Firmware is flashed into STM32 successfully with no errors.
App Update	Verify the app can download and apply the latest version of itself.	MDPHelper App, Server	App updates to the latest version without issues.
Error Handling	Test how the system responds to errors such as missing files or server downtime.	MDPHelper App, Server Logs	App handles errors gracefully, displaying appropriate messages.
End-to-End Flow	Test the full update cycle from firmware download to installation.	MDPHelper App, Server, Database, STM32	Full update cycle completes successfully without issues.

4.5 Deployment

For deployment, Docker was utilized to create a consistent and portable environment for running the app. This ensures the system can operate across various machines with minimal setup.

Deployment Step	Description	Expected Output
Create Docker Image	Build a Docker image containing all dependencies for both the app and server.	Image successfully built with the tag.
Create Docker Containers	Create containers for the app and server for isolated execution.	Containers running with ports mapped (e.g., 8080:8080).
Test Deployment	Verify containers run correctly across multiple systems.	Log output showing that the containers are running and functioning properly.
Environment Consistency	Ensure the app behaves identically on different systems using containers.	The app functions consistently across different systems without errors.

Chapter 5: Results and Discussion

5.1 System Performance

5.1.1 Flashing Speed Comparison

To evaluate the efficiency of the firmware flashing process, a comparison was conducted between the implemented MDPHelper flashing methods (UART and ST-LINK) and the existing third-party flashing tool, FLYMCU.exe, which was introduced for STM32 development by the course instructor in MDP. The flashing time for each method was measured over five trials, and the results are summarized in below figure.

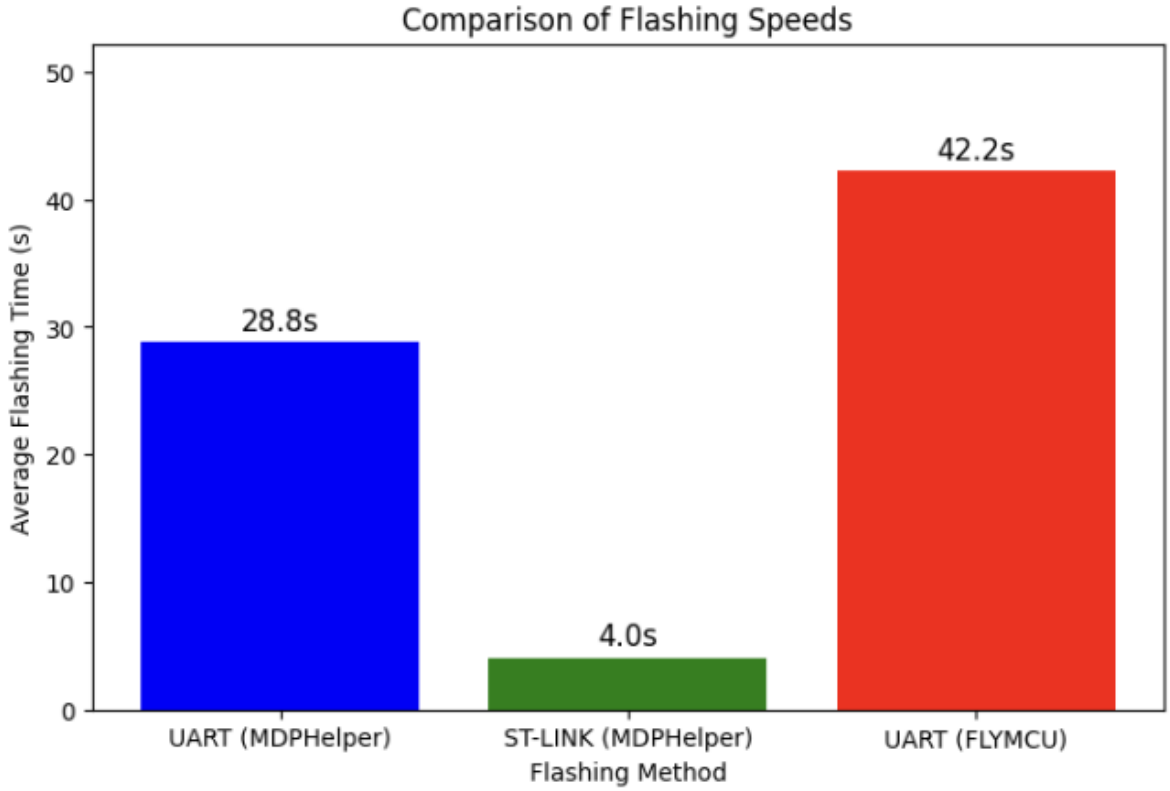

Figure 27: Comparison of firmware flashing time across different methods

As shown in Figure 27, the ST-LINK flashing method in MDPHelper significantly outperformed both UART flashing methods in terms of speed. The average flashing time for ST-LINK was approximately **4.0 seconds**, making it the fastest option. The UART flashing method in MDPHelper took **28.8 seconds** on average, which is much faster than FLYMCU, which averaged **42.2 seconds**. This demonstrates that the MDPHelper flashing method provides better efficiency in firmware transfers compared to the third-party tool.

5.1.1 Backend and Database Performance

To evaluate the efficiency of the backend and database system, key API endpoints were tested to assess response times, data transfer sizes, and overall reliability. These endpoints are critical for retrieving firmware metadata, fetching application update information, and downloading the updated app files from the server. Tests were conducted over five trials using tools such as Postman and curl to simulate client requests.

The following tests were conducted over a local network:

Endpoint	Description	Response Time (Average)	Response Size
GET /api/firmware	Retrieves firmwares	12 ms	82.56 KB
GET /api/appinfo	Retrieves latest app update info	4 ms	150 B
Download via DownloadUrl (.exe)	Downloads Windows app update (250 MB)	~9.5 s	250 MB
Download via DownloadUrl (.pkg)	Downloads macOS app update (45 MB)	~1.8 s	45 MB

The test results demonstrate excellent backend performance for metadata retrieval, with both firmware and app info endpoints responding in under 15 milliseconds. File downloads from the provided DownloadUrl were also efficient, with the 250 MB .exe file downloading in approximately 9.5 seconds and the 45 MB .pkg file in 1.8 seconds on the same local network.

5.2 User Feedback

5.2.1 Feedback Responses

The user feedback survey provided valuable insights into the functionality, usability, and areas for improvement of the MDPHelper app. A summary of the key findings is presented below. The details of the survey and responses can be found in **Appendix C**.

Key Features and Most Useful Functions:

Users highlighted the following features as the most useful in the app:

- **Servo Testing:** The servo testing feature was particularly appreciated for its ease of use in validating hardware performance.
- **IR Testing:** This feature stood out for its effectiveness in testing the functionality of infrared sensors.
- **Custom Testing:** Users found the custom testing feature highly useful, as it allowed for tailored testing scenarios, adding flexibility.
- **Ultrasonic Testing:** The accuracy and ease of use of the ultrasonic sensor were frequently mentioned as valuable features.
- **Sensor Connection Guide:** The guide for connecting sensors with the STM32 was a key resource that users found helpful.
- **Gyro Drift Feature:** The gyro drift feature was identified as essential, particularly for motion-based testing.

Issues Encountered:

Despite the overall positive reception, some users encountered the following challenges:

- **Freezing or Crashing:** 20% of users reported experiencing software freezes or crashes, particularly when queuing commands during IR testing or after executing one command.
- **UI and Performance:** While the UI was generally deemed clean, several users suggested improvements, particularly in colour schemes and contrast for better readability.

- **Instability:** Some users mentioned that the app felt unstable during certain tasks, such as graph generation or when switching between commands, which resulted in minor delays or hangs.

Suggestions for Improvement:

The following suggestions were provided to enhance the app's performance and usability:

- **UI Enhancements:** Users recommended improving the contrast and providing more colour customization options. The ability to display real-time sensor data was also suggested as a useful feature.
- **Additional Features:** Users requested the addition of an auto-calibration function for sensors, real-time sensor data display, and the ability to export test results.
- **Documentation:** Although most users found the documentation clear, some suggested that it could be more concise and clearer, particularly for first-time users. Additional troubleshooting sections and QnA would enhance its usefulness.
- **Stability Improvements:** Feedback indicated that addressing the frequent crashes, particularly during IR testing, should be a priority for improving overall stability.

Overall Rating and Recommendations:

The software received an average rating of **3.9/5**, with **70%** of users rating it **4 (Satisfied)**, and **20%** rating it **3 (Neutral)**.

- **Most Liked Features:** Users appreciated the app's hardware testing capabilities, especially the servo testing and the speed of firmware flashing. The intuitive interface was also frequently praised.
- **Recommendation: 40%** of users indicated they would **"very likely"** recommend the app to others working with STM32 in MDP, while **50%** were **"somewhat likely"** to recommend it.

5.2.2 Analysis of Feedback

The feedback analysis highlights both the strengths of the MDPHelper app and areas that require further attention:

Strengths:

- **Ease of Use:** The app's core features, such as motor and servo testing, are intuitive and easy to access, contributing to positive user experiences.
- **Time Efficiency:** The app effectively streamlines hardware testing and firmware flashing processes, significantly saving users time.

Areas for Improvement:

- **System Stability:** Stability remains a critical concern, with users reporting crashes and freezes, particularly during specific tests (e.g., IR and motion testing). Stability improvements are essential for ensuring consistent performance.
- **UI Enhancements:** Several users recommended improvements to the UI, including better contrast and more colour customization. Additionally, the ability to display real-time sensor data would greatly enhance usability.
- **Documentation:** While the user guide is comprehensive, it could benefit from being more concise and clearer, particularly for first-time users. Adding more troubleshooting and QnA sections would further improve the guide's usability.

5.3 System Limitations

While the current implementation of MDPHelper successfully streamlines hardware testing and firmware interaction for MDP students, several limitations were observed during development and testing. These limitations impact the depth, reliability, and scalability of the system:

- **Manual Observation Without Automated Validation**

  The testing process currently depends on user observation to determine whether a peripheral behaves as expected. There is no automated mechanism to compare actual outputs with predefined expected values, which reduces the precision and repeatability of test results.

- **Functional Rather Than Quantitative Testing**

  The current system primarily verifies basic functionality, for instance, confirming that a motor spins or a sensor responds. However, the system does not assess whether outputs meet specific quantitative criteria, such as verifying the motor's RPM matches the PWM duty cycle input.

- **Absence of Cross-Validation for Encoder Feedback**

  The system assumes that the encoder provides accurate feedback when measuring motor speed. However, there is no secondary method or reference implemented to verify encoder accuracy. This lack of cross-validation could result in undetected errors or misleading test outcomes.

- **No Built-in Accuracy Checks or Threshold Alerts**

  There is currently no mechanism for defining acceptable thresholds for sensor values or actuator responses, nor does the system provide alerts when readings deviate from expected performance. This limits its utility for formal validation, where users would benefit from clear pass/fail indicators or automated warnings.

- **Limited Error Handling and Data Logging**

  Errors are minimally reported, and test data is not automatically logged or stored. Without a logging system, users cannot review past test results, making fault analysis and troubleshooting more difficult.

Chapter 6: Conclusion and Future Work

6.1 Conclusions

This project aimed to develop a desktop-based hardware and software testing system to assist students working with STM32 in MDP. The system successfully streamlined the testing process by providing essential features such as sensor validation, motor and servo control, real-time feedback, and firmware flashing through a user-friendly interface. User feedback confirmed the system's usefulness, and performance evaluations demonstrated improved efficiency compared to existing tools.

The current implementation lays a functional and practical foundation, but its true potential lies in future enhancements. By incorporating features outlined in Section 6.2, the system can evolve into a more robust and intelligent validation platform. This project not only addresses the immediate testing needs of MDP students, but also paves the way for more scalable, accurate, and data-driven validation solutions in embedded systems education.

6.2 Future Works

Future work on this project could focus on two main directions which are enhancing the system and expanding its testing capabilities.

Future work on this project can focus on two key areas: enhancing the system's usability and robustness, and expanding its testing and validation capabilities to address current limitations.

6.2.1 Enhancing the System

To improve usability, the desktop app's interface could be made more intuitive and customizable. Suggested features include real-time graphical feedback for sensors, and a responsive layout that adapts to user workflows.

To address the current limitation of limited error handling and data logging, future versions should include:

- A built-in logging system to automatically record test results, user actions, and system errors.
- Downloadable logs for post-test analysis and reporting.

Additionally, to improve maintainability and scalability:

- Backend infrastructure can be upgraded to support better file versioning, higher data throughput, and concurrent users.
- Implementing HTTPS communication, user authentication, and access control would strengthen data security and prevent unauthorized access to firmware and test logs.

6.2.2 Expanding Testing Capabilities

Another key area for future development is expanding the system's testing capabilities to enhance accuracy, automation, and reliability.

First, the system can be extended to support **firmware code testing**, allowing students to validate the functionality and reliability of their embedded code before deployment. This would reduce trial-and-error debugging and improve overall development efficiency.

To minimise manual intervention, **automated test suites** can be implemented to execute tests consistently and generate objective results. This would help ensure reproducibility and free students from having to interpret results solely through visual observation.

In addition, the system should support **quantitative testing**, enabling users to define performance thresholds and verify whether sensor readings or actuator responses meet those criteria. For example, the system could validate if a motor's RPM aligns with the intended PWM duty cycle.

To enhance data reliability, **cross-validation techniques** such as calibration routines or reference models could be introduced. These would help verify the accuracy of encoder or sensor feedback and detect inconsistencies.

Finally, incorporating **machine learning models** to analyse historical test data could enable predictive maintenance. By identifying patterns or anomalies in hardware readings, the system could alert users to potential hardware issues before they lead to failure.

Collectively, these enhancements would shift MDPHelper from a functional testing tool into a more comprehensive and intelligent validation platform.

References

[1] “STM32 32-bit Arm Cortex MCUs - PDF Documentation,” STMicroelectronics. Accessed: Sep. 22, 2024. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus/documentation.html.

[2] “Why Software Projects Fail: Top 10 Reasons and How to Avoid Them,” Solveit. Accessed: Sep. 21, 2024. [Online]. Available: https://solveit.dev/blog/why-software-projects-fail.

[3] “Top Software Failures Due to Lack of Testing - BugRaptors,” BugRaptors. Accessed: Sep. 21, 2024. [Online]. Available: https://www.bugraptors.com/blog/top-software-failures-due-to-lack-of-testing.

[4] Algoteque, “The Importance of Testing for a Successful Project,” Algoteque Software Development & Nearshoring. Accessed: Sep. 21, 2024. [Online]. Available: https://algoteque.com/the-importance-of-testing-for-a-successful-project/.

[5] Raspberry Pi Foundation, “Sense HAT Documentation,” Raspberry Pi. Accessed: Mar. 20, 2025. [Online]. Available: https://www.raspberrypi.com/documentation/accessories/sense-hat.html.

[6] Arm Keil, “Keil uVision Debug Guide,” Arm. Accessed: Mar. 20, 2025. [Online]. Available: https://keil.com/support/man/docs/uv4/uv4_dg_debug.asp.

[7] Open Robotics, “Gazebo Simulation,” Open Robotics. Accessed: Mar. 20, 2025. [Online]. Available: https://gazebosim.org/home.

[8] R. T. Alexander and J. M. Bieman, "Testing Embedded Software Using Simulated Hardware" *Proceedings of the International Conference on Software Engineering Research and Practice*, 2003. Accessed: Mar. 20, 2025. [Online]. Available: https://www.researchgate.net/publication/250427239_Testing_Embedded_Software_using_Simulated_Hardware.

[9] A. E. Bedoya, “A Review on Verification and Validation for Embedded Software,” *IEEE Latin America Transactions*. Accessed: Mar. 21, 2025. [Online]. Available: https://www.researchgate.net/publication/305721561_A_Review_on_Verification_and_Validation_for_Embedded_Software.

[10] ST-Link Org, “ST-Link Open Source Tools for STM32,” GitHub Repository. Accessed: Mar. 20, 2025. [Online]. Available: https://github.com/stlink-org/stlink

[11] Sainapse, "TD-8120MG 20kg Digital Servo," Sainapse. Accessed: Mar. 20, 2025. [Online]. Available: https://www.sainapse.com.my/td8120mg-20kg-digital-servo.

Appendix A: Peer Responses to STM32 Development Challenges

Response No.	Peer Response
Response 1	"The biggest issue was debugging the software on the STM32 and determining whether the issue was with the uploaded code or the robot's hardware. It was challenging to figure out if the problem was with the coding logic or if it was an issue with the hardware, especially when I was testing in real-time."
Response 2	"One of the main problems I faced was trying to troubleshoot the robot's hardware without being sure if the issue was related to software. It took a lot of time to isolate the problem, and I had to check each component individually."
Response 3	"I struggled with ensuring that the sensors were correctly calibrated. However, once I narrowed down whether the issue was in the hardware or software, things went much smoother."
Response 4	"Debugging on the STM32 was challenging. I often found myself questioning whether the software was functioning as expected or if the hardware setup was faulty. It required a lot of back-and-forth, constantly switching between the software code and the physical setup."
Response 5	"The toughest part was testing the integration between software and hardware. Without a clear testing framework, I was left to manually inspect each piece of hardware while tweaking software code. This trial-and-error process consumed a lot of time."

Appendix B: Use Case Description

Use Case	**ID: CASE001**
Use Case Name	Hardware Testing & Validation
Actor	MDPHelper User
Description	The user performs hardware testing using the desktop app by sending commands to the STM32, receiving data, and validating hardware functionality.
Preconditions	1. The STM32 is connected to the desktop app. 2. The required firmware is installed on the STM32. 3. The desktop app is running. 4. User is at Hardware Testing page
Postconditions	1. The system displays real-time test results. 2. The user can validate hardware performance based on the data.
Priority	High
Frequency of Use	High
Flow of Events	1. The user selects testing part from the desktop app. 2. The system displays available test commands. 3. The user selects a test command. 4. The app sends the command to STM32 via UART. 5. STM32 executes the test and collects data. 6. STM32 sends data back to the desktop app. 7. The app processes and displays the results.
Alternative Flows	1. If the connection is lost, the system prompts the user to reconnect the hardware.
Includes	1. System Communication (Sending/Receiving Data) 2. User Assistant (Guidance & Troubleshooting)
Assumptions	1. The STM32 is properly connected and powered. 2. USB-UART drivers are installed.

Use Case	**ID: CASE002**
Use Case Name	System Communication
Actor	MDPHelper User, STM32
Description	Handles bidirectional communication between the desktop app and STM32, including sending commands and receiving test data.
Preconditions	1. STM32 is powered on and connected via USB. 2. The desktop app is running. 3. User is at Hardware Testing page
Postconditions	1. The system successfully sends and receives data.
Priority	High
Frequency of Use	High
Flow of Events	1. The system initializes a UART connection with STM32. 2. The user selects a command to send. 3. The app sends the command via UART. 4. STM32 executes the command and collects data. 5. STM32 sends the data back. 6. The system receives, processes, and displays the data.
Alternative Flows	-
Includes	1. Send Command to STM32 2. Receive Data from STM32
Assumptions	1. STM32 firmware supports the required test commands. 2. UART connection is stable.

Use Case	**ID: CASE003**
Use Case Name	Firmware Management
Actor	MDPHelper User, Server
Description	The user retrieves and flashes firmware onto STM32 through the desktop app.
Preconditions	1. STM32 is powered on and connected via USB. 2. The desktop app is running. 3. User is at Hardware Testing page 4. The desktop app is connected to the server.
Postconditions	1. The firmware is successfully flashed onto STM32.
Priority	High
Frequency of Use	High
Flow of Events	1. The user selects firmware to flash. 2. The system retrieves firmware from the server. 3. The system flashes the firmware onto STM32. 4. The system verifies firmware installation.
Alternative Flows	-
Includes	1. Retrieve Firmware 2. Flash Firmware
Assumptions	-

Use Case	**ID: CASE004**
Use Case Name	App Update
Actor	MDPHelper User, Server
Description	The app checks for updates and installs new versions when available.
Preconditions	1. The desktop app is connected to the internet. 2. The server has an update available.
Postconditions	1. The system is updated to the latest version.
Priority	Medium
Frequency of Use	Low
Flow of Events	1. The user starts the app. 2. The system checks for updates in the background. 3. If an update is available, the system downloads it. 4. The system installs the update. 5. System prompt user to restart the app.
Alternative Flows	-
Includes	1. Check for Update 2. Download & Install Update
Assumptions	-

Use Case	**ID: CASE005**
Use Case Name	User Assistance
Actor	MDPHelper User
Description	The system provides troubleshooting guides and Q&A support to assist users with hardware testing and validation.
Preconditions	1. The desktop app is running. 2. The user encounters an issue or requires assistance.
Postconditions	1. The user receives guidance.
Priority	Medium
Frequency of Use	Medium
Flow of Events	1. The user navigates to the About Page. 2. The system displays a menu with troubleshooting guides and Q&A support. 3. The user selects either troubleshooting guides or Q&A. 4. If troubleshooting is selected, the system provides step-by-step solutions. 5. If Q&A is selected, the system displays frequently asked questions with answers. 6. The user follows the provided steps to resolve the issue.
Alternative Flows	1. If the issue is hardware-related, the system may suggest rechecking hardware connections.
Includes	1. Troubleshooting Guides 2. Q&A
Assumptions	1. Users follow the provided instructions correctly.

Appendix C: Survey Form for User Feedback

No.	Survey Question	Question Type	Response
1	Are features (e.g motor testing, servo testing, etc) easy to access and understand?	Yes/No	Yes (90%), No (10%)
2	Are the hardware testing features working as expected?	Yes/No	Yes (90%), No (10%)
3	Which features did you find most useful?	Open-ended Text	Servo Testing, IR Testing, Custom Testing, Ultrasonic Testing, Gyro Drift Feature, User Guide for Sensor Connections
4	During use, how many times did this software freeze, crash, or stall long enough to be disruptive to your work at any one given time?	Multiple Choice (0 – 2 times) (3 – 5 times) (6 – 9 times) (10 or more times)	0-2 times (80%), 3-5 times (20%)
5	If more than 2 times, please describe the issue in as much detail as possible.	Open-ended Text	Frequent crashes during IR testing when queuing the next command, board hangs when executing commands.
6	Is the user guide clear and complete?	Yes/No	Yes (80%), No (20%)
7	Are instructions easy to follow?	Yes/No	Yes (100%)
8	Any suggestions or feedback on the following areas: • **UI**: e.g How can the user interface be improved? • **Features**: e.g Are there any features you'd like to see added? • **Documentation**: e.g How can the documentation be made more helpful? • **Other Feedback**: e.g Any additional comments or suggestions?	Open-ended Text	UI improvements (color, contrast), add sensor auto-calibration, real-time logging, export test results. Documentation could be more concise. Other feedback includes general improvements for stability and usability.
9	Overall, how would you rate this software? (1 = Very Dissatisfied, 5 = Very Satisfied)	Multiple Choice (1-5 Scale)	3.90 (Average)
10	What did you like most about the app?	Open-ended Text	Hardware testing features, flashing STM32, servo testing

			functionality, intuitive interface.
11	Would you recommend this app to others who are handling STM in MDP?	Multiple Choice (Very likely) (Somewhat Likely) (Neutral) (Somewhat Unlikely) (Very Unlikely)	Very likely (40%), Somewhat likely (50%), Neutral (10%)