Politecnico di Torino

Politecnico di Torino

Master's Degree in Mechatronic Engineering

Academic year 2025/2026

Graduation Session: July 2026

Design and Implementation of a Modular HIL-Inspired Test Bench for Automated Validation of an Agricultural Rover VCU

Supervisors:

Alessandro Savino

Stefano Di Carlo

Vincenzo Cacciatore

Candidate:

Davide De Bortoli

Abstract

This thesis presents the design and implementation of a modular Hardware-in-the-Loop-inspired test bench. This bench is developed for the validation of the Vehicle Control Unit (VCU) of an agricultural rover within the e-RO.Alps project, an initiative aimed at developing a compact electrically powered tracked robot for precision viticulture in mountain environments. At the beginning of the work, the final rover hardware was not yet available, making it impossible to validate the VCU firmware directly with the target system. The thesis responds to this constraint by developing a testing infrastructure capable of emulating the behaviour of missing peripherals and validating the communication and input/output layer of the VCU through automated and repeatable test procedures, independently of the final hardware configuration. The proposed architecture consists of three main components: a host PC running the OpenHTF-based orchestration software, a test board based on the STM32F103C8T6 microcontroller running the CHAOS real-time operating system, and the VCU under test. The test board acts as the autonomous test executor, generating and receiving CAN messages, digital signals, analog voltages, and PWM waveforms, while OpenHTF handles session management, structured reporting, and real-time logging. Communication between the host PC and the test board occurs via a UART serial interface, which is used to transfer encoded test definitions and to collect step-by-step execution updates. Tests are defined as JSON files, encoded into binary HEX strings, and transferred to the board at runtime, enabling new test cases to be added without modifying the firmware. The execution model, based on periodic RTOS tasks with configurable priorities, ensures that test timing is governed internally by the embedded scheduler and is independent of host PC latency. A hierarchical test structure, consisting of tests, steps, and actions, supports bidirectional CAN communication verification, digital and analog signal validation, PWM generation, runtime value propagation, and conditional branching logic. The system was validated on an experimental setup in which a second identical board simulated the VCU behaviour. Two representative test scenarios were executed, covering bidirectional digital and CAN communication with conditional branching, and an end-to-end analog acquisition to PWM control chain via CAN transmission. All implemented functionalities operated correctly, confirming the feasibility of the proposed approach. The thesis also discusses current limitations and outlines future developments toward integration with the complete rover system.

Table of Contents

Chapter 1

Introduction

The increasing adoption of automation and intelligent control technologies in modern agriculture has led to the development of a new generation of robots capable of operating in complex environments. In particular, precision agriculture applications require autonomous or semi-autonomous systems able to interact safely and reliably with dynamic outdoor scenarios, while maintaining high levels of efficiency and operational accuracy. In this context, embedded electronic control systems represent a fundamental component, coordinating sensing, actuation, communication, and decision-making functionalities.

The development of embedded systems for robotic applications involves the integration of hardware, so low-level firmware, communication interfaces, and high-level control software. One of the main challenges in such projects is the strong dependency between software validation and hardware availability. During development processes, software and hardware rarely evolve synchronously, and software verification often needs to begin before the final platform is completely available. Therefore, modern embedded systems engineering increasingly uses structured testing methodologies for decoupling software validation from hardware availability. Among these methodologies, Hardware-in-the-Loop (HIL) testing represents one of the most widely adopted approaches, enabling the interaction between embedded software and simulated hardware environments through signal emulation and communication interfaces [1, 2]. Such approaches allow us to validate communication layers, input/output management, and control functionalities before complete system integration.

This thesis is developed within the context of the e-RO.Alps project, an initiative funded under the Smart Specialisation Strategy of the Aosta Valley region. The project aims to develop a compact electrically powered rover for precision viticulture applications in mountain environments. The rover is designed to operate in steep vineyards characterised by narrow inter-row spacing and limited accessibility, where conventional agricultural machinery cannot safely operate.

At the core of the rover electronic architecture there is a Vehicle Control Unit (VCU), responsible for coordinating sensors, actuators, communication interfaces, and high-level control functions. The development of the VCU was assigned to the company Mind the BIT, under the supervision of Vincenzo Cacciatore.

However, at the beginning of this work, the final structure of the rover and its peripherals had not yet been defined. This situation motivated the development of a dedicated test bench capable of supporting VCU software development and validation independently from the final structure.

The work presented in this thesis therefore focuses on the design and implementation of a modular Hardware-in-the-Loop-inspired test bench, created to validate the communication and input/output handling of the rover VCU through automated testing procedures and peripheral signal emulation.

1.1 Research Context and Motivation

Modern agriculture is currently facing a significant technological transformation driven by the integration of robotics, sensing technologies, artificial intelligence, and advanced embedded systems. This evolution, commonly referred to as Agriculture 4.0, aims to improve productivity, sustainability, and operational efficiency through the adoption of intelligent automated solutions.

Precision viticulture is therefore a challenging application area. Vineyard environments are often characterised by irregular terrain, narrow paths, steep slopes, and highly variable environmental conditions. These characteristics make many agricultural operations difficult to automate using conventional machinery and create demand for compact robotic platforms capable of operating safely in constrained environments.

The e-RO.Alps project specifically addresses the challenges of mountain viticulture in the Aosta Valley region, where vineyards are frequently located on slopes exceeding 30 degrees and feature inter-row spacing as narrow as 1.10 metres. These so-called “heroic vineyards” require extensive manual labour for pruning, monitoring, treatment, and harvesting activities. These working conditions increase operational costs and expose the operators to physical risks.

The rover developed within this project is a compact electrically powered tracked robot that helps operators to perform these tasks in a safer and more efficient way in this challenging environment. The system integrates multiple technological subsystems, including traction control, sensor acquisition, remote communication interfaces, and artificial intelligence modules for plant health monitoring and targeted treatment support.

From an engineering perspective, the rover has a distributed architecture in which all subsystems are coordinated by a central Vehicle Control Unit. The

VCU communicates with sensors and actuators, executes control logic, handles safety conditions, and supervises data exchange across the rover electronic network through automotive-oriented communication protocols such as CAN bus and UART interfaces.

The correct operation of the VCU is essential for ensuring both system functionality and operational safety. Failures in communication handling, sensor acquisition, or actuator management may compromise the behaviour of the entire robot. Therefore, the validation process of the embedded software becomes a critical activity during system development [3, 4]. However, embedded systems testing presents several technical challenges. The embedded firmware strongly depends on interactions with physical hardware devices, real-world electrical signals, and communication buses. Therefore, testing requires realistic conditions and sufficient control and observability of the system behaviour.

In this project, these challenges are amplified by the fact that the final hardware platform is not yet fully defined, so sensors, actuators, drivers, and communication peripherals are not yet available. Under these conditions, the need for a flexible testing system becomes fundamental for the software development. Therefore, the approach adopted in this work follows the principle of development independent from target hardware [5]. Several software functionalities related to communication management and input/output handling need to be designed and tested before the complete rover hardware becomes available.

1.2 Problem Statement

At the beginning of this thesis activity, the rover platform was still in an initial development phase. While the general architecture of the system had already been defined, the hardware components, such as sensors, actuators, motor drivers, were either unavailable or not yet selected.

This condition created a significant development constraint. The absence of the final hardware platform limited the possibility of performing testing directly on the target system. As a consequence, the development and validation of the Vehicle Control Unit software required an alternative approach capable of reproducing the behaviour of the missing peripherals and communication interfaces.

This scenario reflects a common problem in embedded systems engineering, where software development often progresses in parallel with hardware design. In such contexts, postponing software validation until the final hardware is available may lead to significant integration issues and delayed debugging activities. Therefore, it is very important to decouple software validation from hardware availability.

More specifically, several technical aspects must be addressed. First, the VCU

communication interfaces must be validated independently from the physical peripherals. The control unit exchanges information through multiple interfaces, including CAN bus and UART communication channels, requiring verification of message transmission and reception.

Second, the input/output subsystem must be tested through realistic signal stimulation. The VCU interacts with digital, analog, and PWM signals originating from sensors and actuators. In the absence of the real devices, these electrical signals must be emulated.

Third, the testing environment must provide sufficient observability over the system behaviour. Embedded debugging activities require monitoring of communication traffic, internal states, and signal responses in order to identify anomalies and validate functional correctness.

Another important challenge concerns flexibility and scalability. Since the rover hardware architecture is expected to evolve during the next project stages, the testing infrastructure must be modular and adaptable, allowing the integration of new peripherals, interfaces, and test scenarios without requiring substantial redesign.

The core problem addressed by this thesis is therefore the design and implementation of a testing infrastructure capable of validating the VCU embedded functionalities in the absence of the final rover hardware while ensuring repeatability, modularity, scalability, and support for automated testing workflows.

1.3 Objective and Contributions of the Thesis

The primary objective of this thesis is the design and implementation of a modular test bench for the validation of the rover Vehicle Control Unit. The proposed system supports the development and verification of VCU firmware before the availability of the complete rover hardware by emulating external peripherals and reproducing their behaviour through communication and signal simulation. To achieve this objective, a Hardware-in-the-Loop-inspired methodology is adopted, for a functional validation of embedded interfaces and communication behaviours through controlled input and output monitoring.

The work begins with an analysis of the rover electronic architecture and the VCU functional responsibilities, focusing on communication interfaces, I/O management, and testing requirements. Based on this analysis, a modular test bench architecture is designed, integrating hardware interfaces, signal emulation mechanisms, and automated testing software.

A software infrastructure for automated testing is then developed, including communication management tools, signal generation, test execution workflows, and data logging functionalities. The framework is designed to be modular and

extensible, supporting repeatable validation procedures.

The proposed system enables validation of several VCU functionalities before the availability of the final rover system, including:

- CAN bus and UART communication management;
- digital, analog, and PWM input/output handling;
- signal acquisition and response verification;
- automated execution of functional test procedures.

The main contributions of this thesis can be summarised as follows:

- design of a modular HIL-inspired test bench architecture for VCU validation;
- development of signal simulation and communication emulation mechanisms;
- the implementation of an automated, repeatable and scalable testing framework and methodology, adaptable to the future development stages and hardware evolution.

Beyond the specific e-RO.Alps application, the work also demonstrates how structured testing methodologies can support incremental development processes in embedded robotic systems characterised by evolving hardware specifications and partial hardware availability.

1.4 Organization of the Thesis

The remainder of this thesis is organised as follows.

Chapter 2 presents an overview of the e-RO.Alps agricultural rover system, describing the application context, the electronic architecture, and the main subsystems composing the platform, with particular focus on the Vehicle Control Unit and its interactions with sensors and actuators.

Chapter 3 defines the technical problem addressed by the thesis and analyses the main challenges associated with embedded systems testing in the presence of incomplete hardware availability. The chapter also introduces the functional decomposition of the VCU and discusses the separation between communication, input-output functionalities and control logic.

Chapter 4 reviews the principal testing methodologies adopted in embedded systems engineering, including automated testing approaches and Hardware-in-the-Loop techniques. Open-source and commercial testing solutions are analysed and compared in order to justify the technological choices adopted for this project.

Chapter 5 describes the design of the proposed test bench architecture. The hardware platform, communication interfaces, signal simulation strategies, and system requirements are presented with the functional and non-functional characteristics of the system.

Chapter 6 presents the implementation and validation activities carried out during the thesis work. The chapter details the hardware configuration, software structure, communication management, implemented test cases, and experimental results obtained during the validation.

Finally, Chapter 7 summarises the main outcomes of the work, discusses the current limitations of the developed infrastructure, and outlines possible future developments related to the integration of the complete rover platform and the extension of the testing capabilities.

Chapter 2

Agricultural Rover System Overview

This chapter presents an overview of the e-RO.Alps agricultural rover system, with particular focus on its electronic architecture and on the role of the Vehicle Control Unit (VCU) within the overall platform. The chapter starts introducing the general system architecture and communication structure of the rover, and then moves on to the principal categories of sensors, actuators, and communication interfaces connected to the VCU. Finally, it discusses the functionalities of the VCU more in detail.

At the time this work was carried out, several hardware components of the rover platform had not yet been fully defined. Consequently, the architectural choices presented in this chapter were developed with flexibility and modularity as primary design criteria. Both the VCU firmware and the testing infrastructure were therefore conceived to support different hardware configurations and future integration stages as the project evolves.

2.1 System Architecture of the Rover

The e-RO.Alps rover is a compact electrically powered tracked ground vehicle designed for operation in mountain vineyards and other constrained agricultural environments. The robot is intended to assist operators during agricultural activities performed on steep terrain, where conventional agricultural machinery cannot safely operate. The main features of the platform are illustrated in Figure 2.1.

The rover operates in remote-controlled mode through a radio communication system that allows the operator to supervise vehicle motion and tool actuation from a safe position. In parallel, the system also provides some embedded intelligence functions, including monitoring, communication management, and support

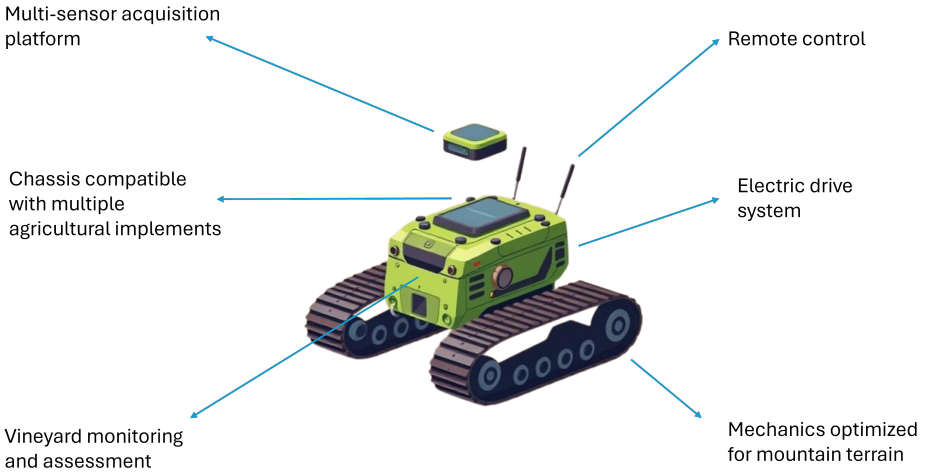

Figure 2.1: Overview of the e-RO.Alps rover platform, highlighting its main features.

functionalities. It also includes an artificial intelligence module dedicated to plant health monitoring and agronomic support functions.

The electronic architecture of the rover is based on a distributed embedded system, where multiple peripheral subsystems communicate through different interfaces and are coordinated by a central Vehicle Control Unit, as shown in Figure 2.2. The VCU acts as the main supervisory node, collecting sensor data, running control logic, handling safety functions, and generating commands for actuators and peripheral devices.

This distributed setup improves modularity and makes the system easier to scale and maintain. Since low-level tasks are handled by dedicated nodes, the VCU can focus on higher-level coordination and decision-making. This separation also simplifies integration, as individual components can be updated or replaced with minimal impact on the rest of the system.

Communication between the VCU and peripheral nodes is primarily based on the CAN (Controller Area Network) bus. CAN is commonly used in automotive and industrial embedded systems due to its robustness and its ability to support deterministic communication in distributed networks [3]. In this rover, it is used to communicate with the motor inverter, the Battery Management System (BMS), and the on-board PC used for development and diagnostics. There is also an Ethernet network, that provides higher-bandwidth communication among the higher-level nodes, including the onboard PC, the HMI, and the perception subsystem. The relationship between the Vehicle Control Module and the Ethernet network had not been fully defined at the time of this work. For this reason, Ethernet communication is outside the scope of the test bench developed in this thesis, and only the interfaces

directly managed by the VCU, such as CAN bus, serial communication, and direct electrical signals, are considered.

The VCU also interacts with different peripherals through direct electrical signals. These include PWM outputs for actuator control, analog inputs for continuous measurements, and digital I/O signals for discrete commands and safety-related signals. A serial communication interface is also used for devices such as the radio frequency transceiver, used for remote operation commands.

Using multiple communication protocols and signal types increases the complexity of the embedded system, since each interface introduces its own timing constraints, electrical characteristics, and validation requirements. These aspects must be considered both during firmware development and during testing, especially when reproducing realistic operating conditions on the test bench.

At the time of development, several hardware specifications had not yet been completely finalised. Consequently, the VCU firmware and testing infrastructure were designed to remain as hardware-independent as possible, allowing future modifications and peripheral integration without requiring important architectural changes.

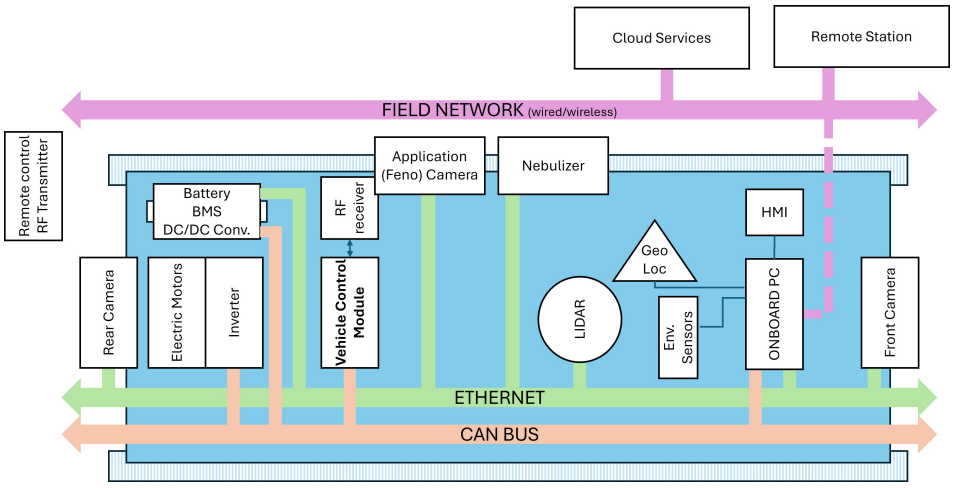

Figure 2.2: Representation of the electronic architecture of the rover, showing the main subsystems and communication networks. The Vehicle Control Module acts as the central node.

2.2 Sensors and Actuators Overview

Understanding the sensors, actuators, and communication interfaces connected to the VCU is necessary to define the validation requirements of the test bench. Since the objective of the proposed infrastructure is to emulate the behaviour of external

peripherals, the analysis of signal types and communication mechanisms represents a fundamental step in the development process.

The rover electronic system includes multiple categories of interfaces, each characterised by different electrical behaviours, timing constraints, and communication requirements. The coexistence of network communication, analog acquisition, PWM actuation, and digital signals increases the complexity of both firmware development and embedded system validation.

CAN Bus Peripherals

The CAN bus represents the primary communication system of the rover electronic architecture. The main nodes connected through the CAN network are the motor inverter, the Battery Management System, and the on-board PC used for supervision and diagnostics. The inverter exchanges real-time information with the VCU, including motor speed, current measurements, temperature data, and operating status flags. In parallel, the VCU transmits command references related to traction control and vehicle motion. Since traction management is one of the most critical functionalities of the rover, reliable and deterministic communication with the inverter is essential. The Battery Management System periodically transmits battery-related information such as cell voltages, pack current, battery temperature, state of charge (SoC), state of health (SoH), and fault and protection states. These parameters are continuously monitored by the VCU to implement energy management strategies and safety protections.

CAN communication is managed using a DBC (Database CAN) file defining frame identifiers, signal encoding rules, scaling factors, timing requirements, and message structures. The DBC file is very important to define and standardise the communication between all CAN nodes in the system.

From the perspective of embedded testing, CAN communication introduces several validation requirements. Besides verifying correct message transmission and reception, the testing infrastructure must also evaluate timing behaviour, timeout management, fault detection, and response to corrupted or missing communication frames.

Pulse Width Modulation Outputs

Pulse Width Modulation (PWM) outputs are used by the VCU to control actuators requiring variable command signals. The duty cycle of the PWM signal encodes the desired command reference, such as speed, torque, or flow regulation. In this rover, PWM signals are primarily used for traction-related actuation, pump control for the nebulizer system, or auxiliary actuator management.

PWM actuation introduces timing constraints because frequency, duty cycle resolution, and waveform stability affect actuator behaviour. Therefore, the validation system must check the correct generation of PWM commands.

Analog Inputs

Several sensors interfaced with the rover provide analog voltage signals proportional to measured physical quantities. These sensors are connected to the analog input channels of the VCU and acquired through the integrated Analog-to-Digital Converter (ADC). Typical analog measurements may include temperature sensing, potentiometer-based position feedback, or auxiliary monitoring signals. The correct acquisition of analog signals depends on multiple factors, including ADC resolution, sampling frequency, signal conditioning, electrical noise, and sensor calibration. Consequently, analog signal validation requires the capability to reproduce controlled voltage levels representative of both nominal and fault operating conditions.

From a testing perspective, analog interfaces are particularly important because incorrect signal interpretation can affect control stability and safety-related decisions.

Digital Input and Output Interfaces

Digital input and output channels are used for management of discrete signals and safety-related functionalities. Typical digital inputs include emergency stop signals, limit switches, enable signals, or fault flags Digital outputs are instead used for relay activation, auxiliary enable commands, signalling indicators, or safety control.

Although digital interfaces are electrically simple, they are frequently involved in safety-critical functions of the system. Consequently, reliable validation of digital signal management is essential to ensure correct rover behaviour under both nominal and emergency operating conditions.

Particular attention must also be dedicated to fault scenarios, including signal interruptions and asynchronous transitions that may occur during real-world operation.

Serial Communication and Auxiliary Interfaces

The radio frequency (RF) transceiver, responsible for receiving operator commands, communicates with the VCU through a serial interface. At the time of this work, the definitive communication protocol and hardware interface had not yet been chosen. For this reason, only part of the RF communication system can be validated in this development stage. Additional auxiliary sensors, such as IMUs, gyroscopes,

or environmental monitoring devices, may also interface with the VCU through protocols that depend on the selected hardware components.

Since several of these peripherals were still under evaluation during the development of this thesis, the test bench architecture was intentionally designed to remain modular and extensible, allowing future integration of additional communication interfaces and signal emulation.

Beyond the interfaces directly managed by the VCU, the rover also integrates a set of higher-level perception and supervision nodes, that communicate over the Ethernet network and interact mainly with the onboard PC. These include: a LIDAR unit for spatial scanning and obstacle detection, front and rear cameras providing visual coverage of the operating area for both remote operator and AI-based plant monitoring; a geolocalisation module (GeoLoc) for field positioning and operational logging; environmental sensors for ambient condition monitoring; and a Human-Machine Interface (HMI) for system configuration and status supervision. The agricultural tool placed on the rover is the nebulizer, whose actuation is managed by the VCU.

2.3 Role and Responsibilities of the Vehicle Control Unit (VCU)

The Vehicle Control Unit is the central element of the rover's electronic architecture. As shown in Figure 2.3, the VCU communicates with the motor inverter, the Battery Management System, and the onboard PC via CAN bus, while direct electrical interfaces include PWM outputs for actuator control, analog inputs for sensor acquisition, and digital I/O for discrete signals and safety-related functions. It also collects sensor data, executes control algorithms, manages communication with peripheral systems, applies safety logic, and generates commands for the actuators. Because of this central role, the VCU is also the main focus of the validation activities described in this thesis. Understanding its responsibilities is therefore essential to define what the test bench must reproduce and verify.

The VCU software is organised as a set of periodic tasks running at different frequencies, depending on the timing requirements of each subsystem. Time-critical functions, such as safety supervision and communication management, require deterministic execution and bounded response times, making timing behaviour a key aspect of system validation. The main VCU responsibilities can be grouped into five principal functional areas: motion control, energy management, tool control, safety management, and communication management.

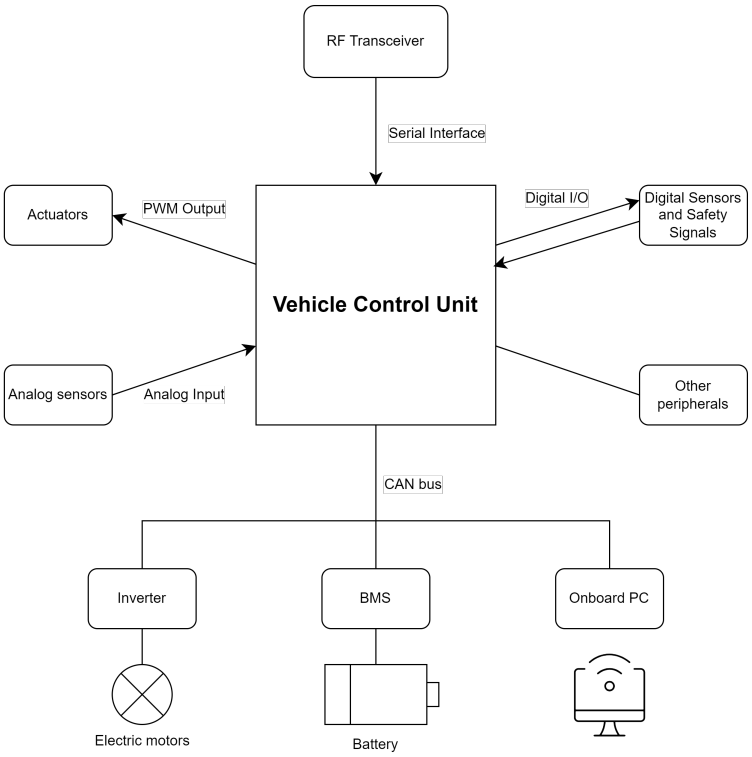

Figure 2.3: Block diagram of the Vehicle Control Unit and its peripheral interfaces. The VCU communicates with the motor inverter, the Battery Management System, and the onboard PC via CAN bus, and with the RF transceiver via serial interface. Direct electrical connections include PWM outputs for actuator control, analog inputs for sensor acquisition, and digital I/O for discrete signals and safety-related functions.

Motion Control

The VCU receives operator commands from the RF transceiver and converts them into traction references for the left and right tracks of the rover. Differential steering is achieved by independently controlling the velocities of the two traction sides.

Since the rover operates on steep and irregular terrain, motion control functionalities must also include traction supervision and stability management. In future development stages, the VCU must include additional functionalities such as slip reduction strategies, slope-dependent speed limitation, inclination monitoring through IMU feedback, and controlled deceleration during emergency conditions.

These functionalities require reliable interaction between communication interfaces, sensor acquisition, and actuator command generation.

Energy Management

The VCU continuously supervises the state of the battery system through communication with the Battery Management System. Energy management functionalities include monitoring of battery operating conditions, power limitation under low SoC conditions, thermal protection mechanisms, operator warning generation, and controlled shutdown procedures in case of critical faults.

Since the rover is fully electrically powered, reliable battery supervision is fundamental both for operational continuity and for safe operation of the platform.

Tool Control

Besides traction management, the VCU is also responsible for controlling the agricultural tools integrated on the rover platform. In the current configuration, this involves generating PWM commands directed to the nebulizer and managing the associated auxiliary outputs.

The tool control architecture was intentionally designed to remain modular, allowing the integration of different agricultural implements without major modifications to the overall software structure.

Safety Management

Safety management represents one of the most critical responsibilities of the VCU. The rover operates in outdoor environments and potentially in proximity to human operators, making reliable fault detection and safe-state management essential. Safety-related functionalities include monitoring of RF communication integrity, emergency stop handling, supervision of inverter and BMS fault conditions, monitoring of digital safety signals, and activation of safe operating states in case of failures.

In the presence of communication failures, invalid sensor measurements, or hardware fault conditions, the VCU must guarantee predictable and safe system behaviour while maintaining operator awareness of the detected anomalies.

The validation of these functionalities is particularly important because many safety mechanisms depend on correct interaction between software logic and hardware interfaces.

Communication Management

The VCU is responsible for maintaining a correct communication with all electronic nodes connected to the rover network. Communication management includes the transmission of periodic CAN frames, the reception and decoding of incoming messages, the fault detection and recovery, the serial communication management,

and the synchronisation of distributed subsystem information. Since communication timing directly affects the behaviour of distributed embedded systems, the validation of communication interfaces constitutes one of the principal objectives of the proposed testing infrastructure.

The VCU must also guarantee sufficient robustness against communication errors, bus interruptions, and invalid data conditions, ensuring that appropriate fallback strategies are activated whenever required.

Given the complexity of the system, with multiple communication interfaces and heterogeneous signal types, a flexible and modular testing infrastructure becomes necessary. The test bench developed in this thesis is designed to reproduce these conditions as realistically as possible, allowing the behaviour of the VCU to be evaluated under both normal and fault scenarios.

Chapter 3

Problem Definition and Challenges

This chapter introduces the technical problem that motivated the design of the test bench developed in this thesis. It starts from the constraints imposed by the early stage of the e-RO.Alps project and then moves to a conceptual separation of the Vehicle Control Unit (VCU) functionalities to divide what can already be validated from what still depends on the final hardware. It closes with the main challenges of embedded system testing.

3.1 System Development Constraints

As introduced in the Section 2.1, at the beginning of this work, the e-RO.Alps rover was still in an early phase of development. Although the general system architecture had been defined, several key hardware components had not yet been fully selected. In particular, no physical sensors or actuators were available, the motor inverter had not been definitively chosen, and the target microcontroller platform for the Vehicle Control Unit (VCU) was still under evaluation. This situation is quite common in embedded and robotic projects, where mechanical, electronic, and software development proceed in parallel. This approach accelerates development, but it also introduces a clear limitation: software cannot yet be validated against a fully defined physical system.

In practice, this has concrete consequences. Without real sensors, signal acquisition and conditioning cannot be checked. Without actuators, there is no way to observe the physical effect of control commands. Without a defined inverter or communication node, CAN interactions cannot be compared with a real counterpart. And without a final microcontroller platform, low-level assumptions such as timer configuration, ADC characteristics, or peripheral mapping are still open to change.

A simple reaction would be to postpone firmware development until the hardware is available. In practice, that choice is inefficient and creates real risks for schedule and integration effort. In embedded development, defects become more expensive to track down and fix as the system moves toward integration and field testing. Early validation becomes important to avoid concentrating all debugging effort in the final phase. For this reason, this thesis adopts a test infrastructure that emulates the behaviour of the missing hardware components. In this way, the VCU firmware can be tested without waiting for the final system implementation.

To define the scope of this approach properly, it is necessary to separate the parts of the VCU that can be validated under hardware abstraction from those that cannot. This distinction is introduced in the next section. In this context, it is useful to split the VCU into two conceptual layers based on their dependence on physical hardware. This distinction is not a software architecture model, but it is a practical way to define what can be tested. The two layers are the *Control Logic Layer*, which relies on the physical behaviour of the system and therefore needs full integration for complete validation, and the *Communication and Input/Output Layer*, which depends directly on hardware interfaces but can still be emulated. The separation between these two layers is illustrated in Figure 3.1.

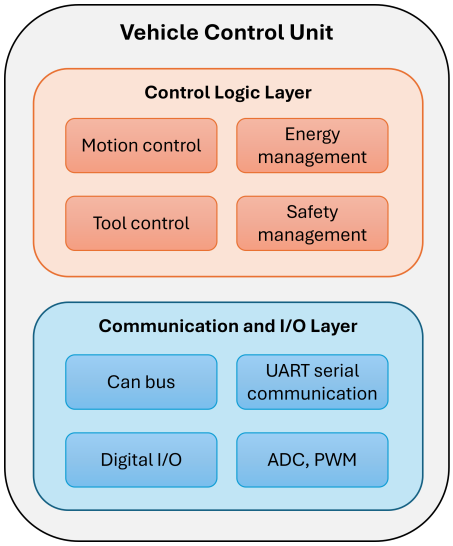

Figure 3.1: Conceptual separation of the VCU firmware into two layers. The Control Logic Layer groups the high-level algorithms of the rover behaviour, as motion control, energy management, tool control, and safety management. The Communication and I/O Layer includes all functions responsible for exchanging data between the VCU and external devices through CAN bus, digital signals, analog acquisition, PWM generation, and serial communication.

Control Logic Layer

The Control Logic Layer includes all the high-level algorithms that govern the rover's behaviour. Motion control, traction management, energy regulation, tool actuation, and safety decision-making all belong to this layer.

One of the most important aspects is that this layer is strongly dependent on the physical characteristics of the system. Motion control performance, for instance, depends on the dynamics of the motors, drivetrain, and terrain interaction. In the same way, energy management strategies depend on the battery chemistry, discharge curves and the behaviour of the Battery Management System (BMS). Safety logic based on inertial measurements also depends on sensor noise, calibration accuracy, and mounting configuration.

Because of these dependencies, proper validation is only possible with the real system. Simplified simulations can support preliminary design, but they are not enough to guarantee correct behaviour in real operating conditions. For this reason, the Control Logic Layer is kept outside the scope of the test bench developed in this thesis. Its complete verification is postponed to later integration stages, when the full hardware configuration becomes available.

Communication and Input/Output Layer

The Communication and Input/Output Layer covers all the functions used to exchange data between the VCU and external devices. CAN communication, digital input/output management, analog signal acquisition, and PWM signal generation are all part of it. Unlike the Control Logic Layer, this one can be developed and tested without depending on the specific hardware variant chosen for the final system. CAN behaviour, for example, is determined by message definitions, timing constraints, and protocol implementation rather than by the specific physical node connected to the bus. In the same way, digital inputs and PWM outputs can be reproduced and measured independently of the final actuators or sensors.

This is why most of the early validation focuses on this layer. Known input signals can be generated to verify firmware behaviour, while output signals can be measured and analysed without depending on the final hardware. As a result, this layer becomes the main target of the test bench developed in this work. Validating it ensures that the VCU correctly implements communication protocols, signal acquisition, and actuation interfaces, all of which are prerequisites for full system integration. Validating this layer early also reduces integration risk, because communication issues are often among the hardest problems to debug once everything is connected.

3.2 Challenges in Embedded System Testing

The development constraints described in the previous section reflect broader challenges in embedded system validation. These issues are well known in industrial and automotive development and have contributed to the adoption of structured testing methodologies for this type of validation [4]. Looking at them in detail is essential both to justify the design choices made in the test bench and to define the requirements that any proposed solution must satisfy.

Hardware Dependency and Validation Constraints

Embedded systems are strongly coupled to the hardware they control. Unlike general-purpose software, firmware runs directly on microcontrollers and interacts with physical peripherals through registers, interrupts, and hardware timers. As a result, system behaviour depends not only on software correctness but also on hardware configuration and electrical characteristics.

Each interface brings its own dependencies. CAN communication relies on transceiver behaviour and bus topology; PWM generation depends on timer resolution and clock frequency; analog acquisition depends on ADC reference voltage, sampling rate, and signal conditioning circuitry. This tight coupling makes embedded system testing heavily dependent on hardware availability. In early development stages, that dependency becomes a limiting factor and pushes the use of alternative approaches to reproduce missing hardware behaviour.

When the final hardware is not available, testing must rely on emulated or simulated representations of sensors, actuators, and communication nodes, but those models remain only approximations of the real system. A simplified sensor model may reproduce nominal output values but miss noise, drift, or transient behaviour. A simulated CAN node may behave correctly under ideal timing conditions but still fail to reproduce bus contention, delays, or fault states. In the same way, digital signal emulation may not capture electrical effects such as bouncing or transient glitches. These discrepancies introduce a fundamental limitation: the fidelity of the test environment directly affects the validity of the obtained results [5].

Debugging and Observability in Embedded Systems

Debugging embedded systems is difficult because behaviour depends not only on logic, but also on timing between concurrent tasks, interrupts, and external events. Common issues such as race conditions, missed deadlines, or communication timeouts may show up only under specific timing conditions, which makes them hard to reproduce. Therefore, effective testing requires a setup that ensures repeatable

execution and good observability of system behaviour without interfering too much with real-time execution.

Another fundamental challenge in embedded testing is limited observability. The internal state of a microcontroller is only partially exposed through external interfaces, and many relevant variables such as task states, intermediate control values, or fault flags are not directly accessible during normal operation. As a result, looking only at external signals is often not enough to understand what is happening inside the system. A proper testing infrastructure therefore needs mechanisms to capture communication data, timing information, and system responses with enough.

Flexibility and Evolution of Requirements

Finally, embedded system test infrastructures must be designed to support changing requirements. In early-stage projects, hardware specifications and system architecture often change during development. A rigid test environment connected to a specific configuration would need continuous redesign as components evolve. For that reason, modularity and scalability are essential properties of any test framework intended for long-term use. The ability to add new interfaces, modify signal definitions, and extend test scenarios without redesigning the entire infrastructure is a key requirement for keeping development efficient.

Chapter 4

Testing Strategies and Development Approach

The development constraints described in the previous chapter highlight the need to validate the VCU firmware before the final rover hardware is available. Meeting this requirement involves selecting an appropriate testing methodology and supporting tools. This chapter analyses the available options and justifies the decisions made in this work. It begins by defining what the test infrastructure must be able to do, then examines the testing approaches used in embedded systems and discusses where the approach adopted in this project fits within them. Then, it analyses the tools considered for test session management, compares them against the defined requirements, and concludes by presenting the development strategy adopted, describing the means and principles guiding the implementation of the test system.

4.1 Requirements for the Test Infrastructure

Given the hardware availability constraints analysed in Chapter 3, the test infrastructure must satisfy a set of requirements that guide both the selection of tools and the architectural choices described in the following chapters. The main requirements are presented below.

Automated and repeatable execution. The test infrastructure must be able of executing test procedures without manual intervention at each step. In embedded development, the same test must be runnable multiple times as the firmware evolves, producing comparable results. The manual interaction, like observing a terminal, pressing buttons, writing down results, introduces variability and does not scale as the number of test cases grows. Automation is therefore essential to ensure efficient and repeatable testing.

Modularity and extensibility. The hardware specifications of the project

are expected to evolve in future development stages. The test infrastructure must be designed so that new test cases, new signal types, and new communication interfaces can be added incrementally without requiring a redesign of the existing structure. This property allows the test system to adapt as the firmware evolves.

Support for heterogeneous hardware interfaces. The VCU communicates with its peripherals through several different interface types: CAN bus, serial communication, digital inputs and outputs, analog signals, and PWM channels. Therefore, the test infrastructure must be able of interacting with all of these. This requirement reflects a key difference between embedded testing and pure software testing, in which the system under test exposes only a software interface.

Structured result reporting. The output of a test session must be more than a pass/fail indication. The infrastructure must produce structured documentation recording which tests were executed, what signals were applied, what responses were observed, and what the overall outcome was. This documentation serves both as an analysis and debugging tool during development and as evidence of validation completeness for later project stages. It is also important that the infrastructure allows real-time updates on test progress to be captured, for example through a logging system that reports the execution state as it evolves, so that system behaviour can be analysed during execution, not only after it completes.

Decoupling of execution timing. In embedded systems, correct behaviour often depends on precise timing conditions. A test infrastructure that manages the timing of individual test steps directly from the PC introduces a dependency on the communication latency between the host and the hardware, which may vary unpredictably due to operating system scheduling. It is therefore important to consider approaches that separate test execution timing from host communication, giving temporal control to components with more predictable behaviour. This aspect influences the architectural choices of the infrastructure and must be considered in the selection of tools and in the definition of the development strategy.

Independence from the final hardware target. The test system must be able to operate without the final rover hardware, interacting with components that emulate peripherals through the available interfaces. This requirement is directly linked to the hardware-independent development principle discussed in Chapter 3.

4.2 Testing Approaches in Embedded Systems

Embedded system validation is performed at different levels during development, each focusing on different aspects of system behaviour and requiring specific infrastructure. A widely used classification in the software and embedded system engineering divides testing into three level: unit testing, unit integration testing, and system testing [18]. This structure is commonly used in industry and is reflected

in frameworks such as Automotive SPICE and ISO 26262 [8, 16]. Understanding where the approach adopted in this project fits within this classification is important for justifying the design choices described in the following chapters.

The **unit testing** (or component testing) target is the individual software modules in isolation. External dependencies, such as other modules, hardware drivers, or peripheral registers, are replaced by software substitutes, allowing the logical behaviour of a function to be verified independently of the hardware context. Unit testing can be executed on a development PC without any target hardware, making it fast and accessible in the early stages of development. However, it has limitations in the embedded domain. The behaviour of hardware interfaces, like interrupt timing, peripheral register configuration, or driver characteristics, cannot be captured by the software substitutes. A module that passes all unit tests may still fail when integrated with the real hardware due to timing dependencies or hardware-specific interactions invisible at the unit level. Unit testing is therefore a useful complement but insufficient as the primary validation method for a system whose correctness depends on hardware interaction.

The **unit integration testing** (or component integration testing) verifies that software modules interact correctly with each other and with the real hardware interfaces. At this level, the firmware runs on the actual microcontroller and is tested through its physical interfaces: CAN messages are transmitted and received, digital inputs are driven and read back, PWM outputs are generated and measured. This is the level most directly relevant to the objectives of this project. Validating that the VCU correctly manages its communication and I/O interfaces requires running the real firmware on real hardware with controlled stimuli applied through the physical interfaces.

The **system testing** verifies the behaviour of the complete system under conditions representative of real operation. In the embedded domain, this corresponds to Hardware-in-the-Loop testing, where missing physical components are replaced by a simulation system that reproduces their electrical and communication behaviour. In a full HIL configuration, the embedded controller runs its firmware and interacts with a real-time simulator that models the dynamic behaviour of the physical process, as motors, actuators, and sensors, closing the control loop [9, 15, 10]. This approach enables the validation of control algorithms and overall system behaviour under realistic operating conditions, without requiring the complete hardware system.

The approach adopted in this project is positioned between integration testing and system testing. On one side, the VCU firmware runs on a real microcontroller with controlled signals applied through physical interfaces, as in classical integration testing. On the other, missing peripherals are replaced by components that emulate their behaviour, similarly to the HIL approach. However, unlike a full HIL setup, the objective is not to simulate the rover's dynamic behaviour in a closed loop, but

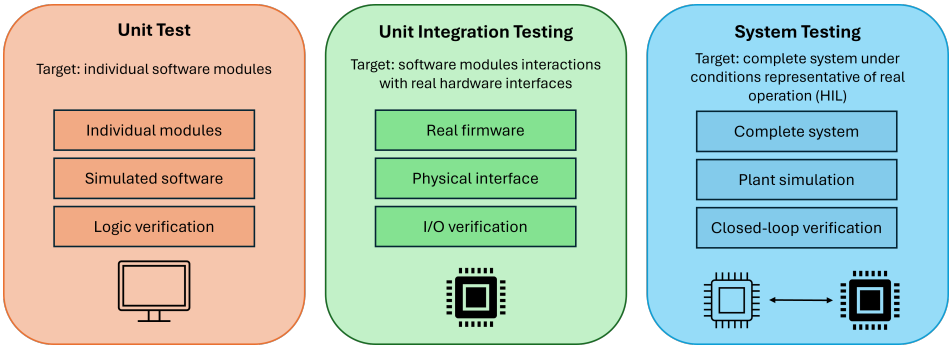

Figure 4.1: Classification of testing levels in embedded systems development.

to verify that the VCU correctly manages its communication and I/O interfaces through structured and controlled tests.

4.3 Frameworks for Test Session Management

The test session management layer is the software component running on the host PC that coordinates the session, interacts with the test hardware, and collects results. Therefore, it must be implemented using an appropriate framework. Given the research context of this project, open-source solutions were considered as the main option, together with an analysis of the main commercial alternatives. This section examines the evaluated options and compares them with the requirements.

4.3.1 Open-Source Testing Solutions

Python-Based Testing Frameworks

Python is widely used for test automation in embedded systems due to its simplicity, readability, extensive library ecosystem, and strong support for hardware communication interfaces. In this context, libraries such as `python-can` and `pyserial` enable direct interaction with CAN and serial interfaces from a host computer, making Python a practical choice for test development.

Within the Python testing ecosystem, `pytest` is one of the most widely used frameworks for automated test execution [17]. It follows a minimal and function-based design, allowing test cases to be written as simple functions instead of class-based structures. This reduces repetitive code and improves readability, especially in large group of tests. A key feature of `pytest` is its fixture system, which allows test resources to be defined once and reused across multiple tests. This is particularly useful in embedded testing, where communication interfaces such as CAN or serial connections can be initialized once per session and reused,

reducing extra work and improving consistency. In addition, parameterization support allows the same test to be run with different input values, avoiding repeated code and improving test coverage. Overall, these features make `pytest` well suited for implementing and executing automated test logic, especially at the software level and in embedded environments.

However, when applied to hardware-oriented test sessions, `pytest` shows some limitations. In particular, its standard output model is mainly designed for software unit testing and does not naturally support structured documentation of hardware validation activities. There is no built-in way to connect measurement data with pass conditions, and no standard format to describe the full test session, including device configuration, execution context, and detailed results for each step. As a result, creating complete and traceable reports requires additional development work and custom extensions. Furthermore, `pytest` does not provide a system to manage interactive test workflows, such as operator confirmation steps or real-time decisions during execution. Similarly, real-time logging of hardware responses is only supported in a basic form and is not structured to integrate with test results and reporting outputs. For these reasons, `pytest` alone is not sufficient as the core framework for full session management in this project, although it remains a valuable tool for implementing low-level and software tests.

OpenHTF

OpenHTF (Open Hardware Testing Framework) is an open-source Python framework developed by Google, released under the Apache 2.0 licence [11]. It was designed for hardware manufacturing test environments, where consistent, structured, and auditable test execution is required. Its architecture is centred on test phases, typed measurements, and structured output records, so this makes it well suited to the management of embedded system validation sessions. The fundamental abstractions of OpenHTF are the *test phase* and the *measurement*. A phase is a Python function representing a discrete step in the test session, annotated with data such as name, description, and timeout. A measurement is a typed, named value recorded during a phase, optionally associated with validators that define acceptance criteria. The framework executes phases sequentially, records all measurements, evaluates validators, and produces a structured JSON output record at the end of each session, containing the name and result of every phase, the values of all recorded measurements, timestamps, and any exceptions encountered. This record provides a complete and persistent trace of the test session, suitable for both result analysis and project documentation. Beyond structured reporting, OpenHTF integrates a logging system that captures and saves messages produced during execution in real time. This is particularly relevant in an embedded context, where the system under test may produce continuous updates on the progress of

operations, for example through a serial interface, that are important to monitor, record, and analyse alongside the session results. The OpenHTF logger can be configured to acquire these information streams and include them in the session documentation, making available not only the final result but also the detailed trace of what occurred during execution. OpenHTF also provides a web-based frontend that displays test progress in real time and allows the operator to interact with the session: confirming that the device is ready, entering parameters, or responding to framework prompts before execution continues. This guided interaction capability is absent in `pytest` and represents a concrete advantage in scenarios where the test session requires active operator involvement. From a hardware integration point of view, OpenHTF is independent to the specific interfaces used: all communication with the device under test is implemented within phase functions using standard Python libraries. This allows `pyserial` for serial communication and, if needed, `python-can` for CAN bus to be integrated directly without any changes to the framework, while maintaining compatibility with the VCU's different interfaces.

4.3.2 Commercial Testing Solution

NI TestStand

NI TestStand is a commercial test management software developed by National Instruments [12, 13]. It is widely used in industrial, automotive, and aerospace embedded testing due to its complete feature set, graphical test sequencing environment, and strong integration with NI hardware platforms and LabVIEW. TestStand provides a graphical sequence editor where test steps are organized in a hierarchical structure, with built-in support for setup, main, and cleanup phases. It supports multiple programming languages for implementing test steps, including LabVIEW, C/C++, Python, and .NET. It also includes integrated database logging, report generation, and configurable operator interface features. In automotive and embedded applications, it is often used in hardware-in-the-loop test setups, where real-time simulation runs on dedicated hardware and TestStand manages test execution and reporting.

With respect to the requirements of this project, TestStand offers strong capabilities for sequence management and reporting, but also introduces important limitations. First, it is a commercial product with high licensing costs, which are not compatible with this work. Second, it is mainly designed to work within the NI hardware ecosystem, which makes integration with non-NI hardware more difficult without additional development. Third, its execution model, where the framework controls the timing of each individual test step from the host, creates a dependency between test execution and host communication latency. This is not ideal in contexts where timing control must be deterministic. For these reasons,

TestStand was not selected, although it remains a useful reference for understanding the requirements of a mature test session management framework.

4.3.3 Selection Criteria and Comparative Evaluation

The solutions examined, `pytest`, OpenHTF, and NI TestStand, can be compared against the requirements defined in Section 4.1. The evaluation is qualitative and focuses on how each tool supports the requirements in practice. The comparison is summarised in Table 4.1.

Requirement	**pytest**	**OpenHTF**	**NI TestStand**
Automated execution	Full	Full	Full
Modularity	Full	Full	Full
Hardware interface support	Partial (libraries)	Full	Partial (NI ecosystem)
Structured reporting	Limited (extensions)	Full	Full
Real-time logging	Limited	Full	Full
Operator interaction / GUI	No	Full	Full
Hardware independence	Full	Full	Partial
Availability	Open-source	Open-source	Commercial licence

Table 4.1: Comparison of test session management tools against project requirements.

The `pytest` framework satisfies the basic automation and modularity requirements and integrates well with hardware libraries, but it does not provide built-in structured reporting, real-time session logging, and operator interaction capabilities. NI TestStand provides a mature and complete solution but is excluded due to licensing costs, hardware-specific dependencies, and an execution model that couples test timing with host communication latency. OpenHTF meets all requirements by providing structured reporting and real-time logging, including a web-based operator interface, integrating with Python hardware libraries, and being open-source. Its phase-based execution model, in which each phase manages one aspect of the session without controlling every step, is suitable for architectures where timing control is delegated to components with more deterministic behaviour. OpenHTF was therefore selected as the test session management framework for this project. Its open-source nature also ensures that the test infrastructure remains accessible and extensible for future development of the project.

4.4 Development Strategy

The analyses presented in the previous sections on testing approaches, test management frameworks, and the HIL methodology form the basis of the development

strategy adopted in this project. This section explains how the selected solutions are translated into a practical approach, defining the tools and principles that will guide the development of the test system. The detailed implementation is described in the following chapters.

The main objective is to develop a system for validating the VCU communication and I/O functions through structured and automated tests before the final rover hardware is available. The chosen methodology is functional HIL testing: the VCU firmware runs on the target microcontroller and interacts with a test board that emulates the behaviour of the missing peripherals by generating controlled input signals and measuring output responses. This approach preserves the main characteristic of HIL testing, so real firmware running on real hardware, while adapting it to the goals of this project, which focus on verifying interface behaviour rather than simulating the rover dynamics.

Test session management is handled by OpenHTF running on the host PC. Its role is to coordinate the test session by verifying that the connection with the test board is active, sending the information required for execution, starting the test, and collecting the final results. The internal test logic, including the execution sequence, signal handling, condition evaluation, and detailed reporting, is executed autonomously by the test board. This approach satisfies the execution timing decoupling requirement discussed in Section 4.1. Delegating the execution control to the embedded board, the test timing becomes independent of communication latency and operating system scheduling on the host PC, resulting in more deterministic behaviour. During the execution, the board continuously reports test progress through the serial interface. Information such as the status of each step, the actions performed, the measured values, and intermediate results is transmitted in real time. The OpenHTF logging system captures and stores this information together with the test session, providing not only the final outcome but also a complete execution trace for analysis and documentation.

The overall architecture of the test system, with OpenHTF acting as the orchestrator and the embedded board acting as the autonomous test executor, reflects the modular structure of the system and the ability to be extended with new tests and interfaces. New tests can be created and transferred to the board without modifying the orchestration framework, while additional interfaces can be integrated into the board without affecting the session management logic running on the PC. This separation of responsibilities allows the system to adapt to future changes in the rover hardware and supports the hardware-independent development principle discussed.

Chapter 5

Test Bench Architecture and Requirements Definition

This chapter describes the architecture of the test bench developed for validation of the future VCU. Starting from the system's main objectives, this section presents the selected hardware and software components, explains why they were chosen, and provides an overview of the communication protocols and signal types used. The chapter concludes with the definition of the functional and non-functional requirements of the system, which serve as the reference for verifying the results presented in the following chapter.

5.1 Test Bench Objectives and Design Goals

The test bench is the system used to validate VCU firmware in the absence of the final rover hardware, within the scope defined in Chapter 3. It must reproduce, in a controlled and repeatable way, the conditions the VCU will encounter during real operation, limited to the communication and I/O layer.

The design of the test bench is guided by several key objectives. First, it must support peripheral emulation by generating and receiving all signal types expected by the VCU in the real system, including CAN messages defined by the DBC database, discrete digital signals, analog voltages, and PWM signals. This allows the test bench to replace the physical sensors, actuators, and other electronic nodes of the rover. In addition, the system must enable communication validation, ensuring that the VCU correctly transmits and receives CAN messages, handles serial communication properly, and reacts appropriately to error conditions such as timeouts, missing messages, or out-of-range values. Another essential requirement is observability and reporting, meaning that the VCU state must be visible during testing through signal monitoring, CAN traffic capture, and structured logging of

results; this role is performed by OpenHTF on the host PC, which collects both final outcomes and execution traces. Finally, the test bench must satisfy the modularity and hardware-independence requirements defined in Section 4.1, allowing it to be extended as the project evolves without requiring major redesigns. These objectives guide the architectural choices presented in the following sections, including system architecture, hardware integration, operating system, communication protocols, and signal management.

5.2 HIL-Inspired Test Bench Architecture

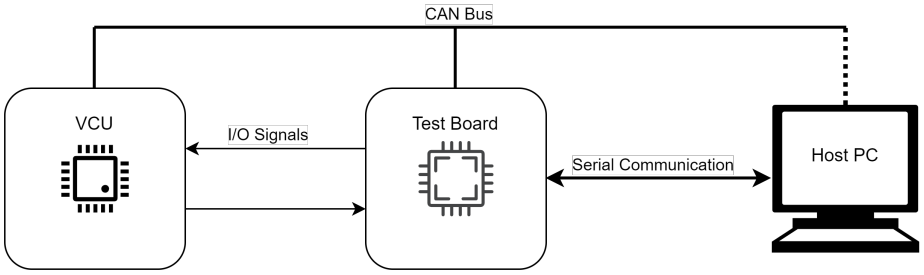

Figure 5.1: Scheme of the Hardware-In-the-Loop-Inspired Test Bench architecture.

The test bench consists of three main elements: the host PC, the test board, and the VCU under test. Figure 5.1 illustrates the architectural scheme with the connections between components.

The **host PC** runs the OpenHTF-based orchestration software. Its role, as described in Chapter 4, is to coordinate the test session: verifying the connection with the test board, transferring the test to be executed, initiating execution, and collecting the final result. Communication between the host PC and the test board occurs through a UART serial interface, which is the primary channel for test buffer transfer and for receiving execution progress updates. In addition, the host PC can be connected to the CAN bus via a CAN-USB adapter, in this case a Vector VN1610 device [6]. This enables real-time CAN traffic monitoring during test execution without interfering with the communication between the test board and the VCU. This additional CAN connection is useful for debugging and for independent verification of the content of exchanged messages.

The **test board** is the central component of the test bench. It contains the dedicated firmware that manages reception of the test buffer from the serial communication with the PC, autonomous execution of the test steps according to an internal timing determined by the operating system tasks, and communication with the VCU through the CAN bus and physical digital signals. During execution, the

board produces step-by-step updates through the serial interface, communicating the status of each action, the values of read signals, and the partial result. At the end, it communicates the overall pass/fail result to the PC.

The **VCU under test** is connected to the test board through the CAN bus, which constitutes the primary communication interface between the two devices, and through direct physical signals, including digital lines for simulation of discrete commands, safety flags, and enable signals. The physical connection between test board and VCU is completed by the sharing of a common ground reference, necessary for correct interpretation of electrical signals.

The control flow is therefore as follows: the PC prepares and transfers the test to the board; the board executes it autonomously, exchanging messages and signals with the VCU; the PC monitors progress through the serial interface and the CAN bus; at the end, OpenHTF collects the result and produces the session report. This scheme reflects the separation between orchestration and execution discussed in Chapter 4: timing and execution logic reside in the embedded board, while the PC handles session management and documentation.

5.3 Test Board and Operating System

Hardware Platform: STM32F103C8T6 and Carrier Board

The test board is based on the STM32F103C8T6 module, commonly known as the *Blue Pill*, mounted on an experimental carrier board designed by Francesco Ficili, whose description is available on GitHub. The choice of this platform is motivated by both practical and technical reasons. The STM32F103C8T6 is a 32-bit ARM Cortex-M3 microcontroller produced by STMicroelectronics, operating at a maximum clock frequency of 72 MHz. It provides 64 KB of Flash memory, 20 KB of SRAM, and a rich set of integrated peripherals: up to 37 GPIO, 10 channels of 12-bit ADC, three USART interfaces, two SPI interfaces, two I2C interfaces, a CAN controller, a Full Speed USB interface, and seven 16-bit timers supporting PWM generation on up to 15 channels. The Blue Pill module exposes all these peripherals through standard connectors, making it accessible in prototyping and experimental contexts. From an economic perspective, the Blue Pill module is extremely affordable, making it suitable for a research and development context where low-cost hardware availability is a practical requirement. Despite the low cost, the performance characteristics and available peripheral set are sufficient for the test bench requirements: the 72 MHz clock frequency ensures adequate response times for test operations, the native CAN controller enables direct communication with the VCU without additional circuits, and the availability of ADC, GPIO, and timers provides the flexibility needed to emulate the different signal types required.

The carrier board integrates, in addition to the physical support for the Blue

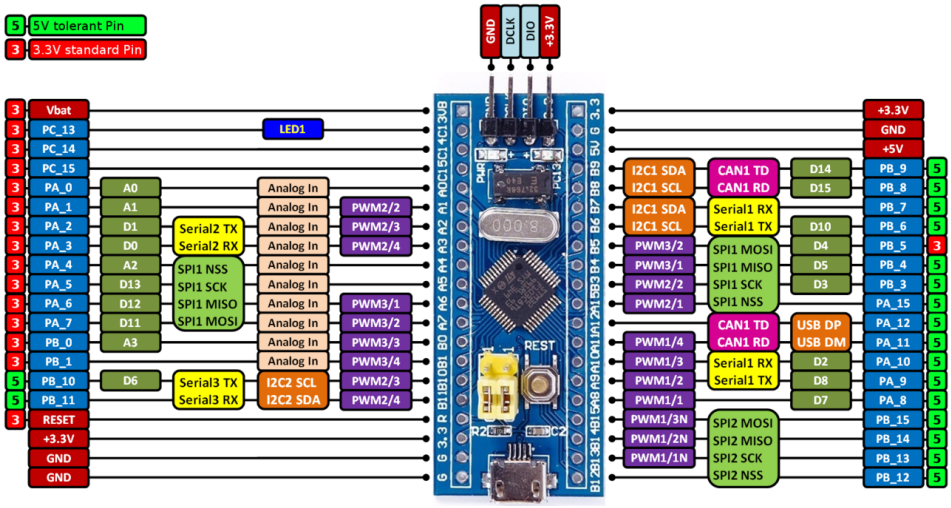

Figure 5.2: Pinout schematic of the STM32F103C8T6 microcontroller, commonly known as the "Blue Pill" [14].

Pill module, several components directly useful for test activities: status LEDs for visual feedback of state during execution, user buttons for manual interaction and simulation of discrete inputs, a potentiometer for generation of variable analog signals, a CAN transceiver for connection to the bus with standard differential CANH/CANL signalling, and a dedicated USB/serial interface for communication with the PC. The carrier board also provides space for additional components, including connectors for additional communication protocols, such as SPI and I2C, and interfaces for real sensors and actuators, making the platform extensible for later project stages.

Operating System: CHAOS RTOS

The test board firmware is developed on CHAOS, an actively developed real-time operating system (RTOS) also created by Francesco Ficili and available on GitHub. CHAOS is inspired by the principles of AUTOSAR OS, but it is not certified or fully compliant with that standard. Its execution model is based on tasks with periodic scheduling at defined priority levels, which is the paradigm best suited to the context of this project. In CHAOS, the fundamental software entities are *tasks*, that are functions executed cyclically at defined time intervals. Each task is assigned a priority that determines the execution order in case of conflict between tasks ready to run at the same instant. The period of each task is configurable and represents the fundamental parameter for ensuring that time-critical operations are executed at the required cadence. CHAOS manages task scheduling, ensuring that timing is respected independently of the overall computational load, provided

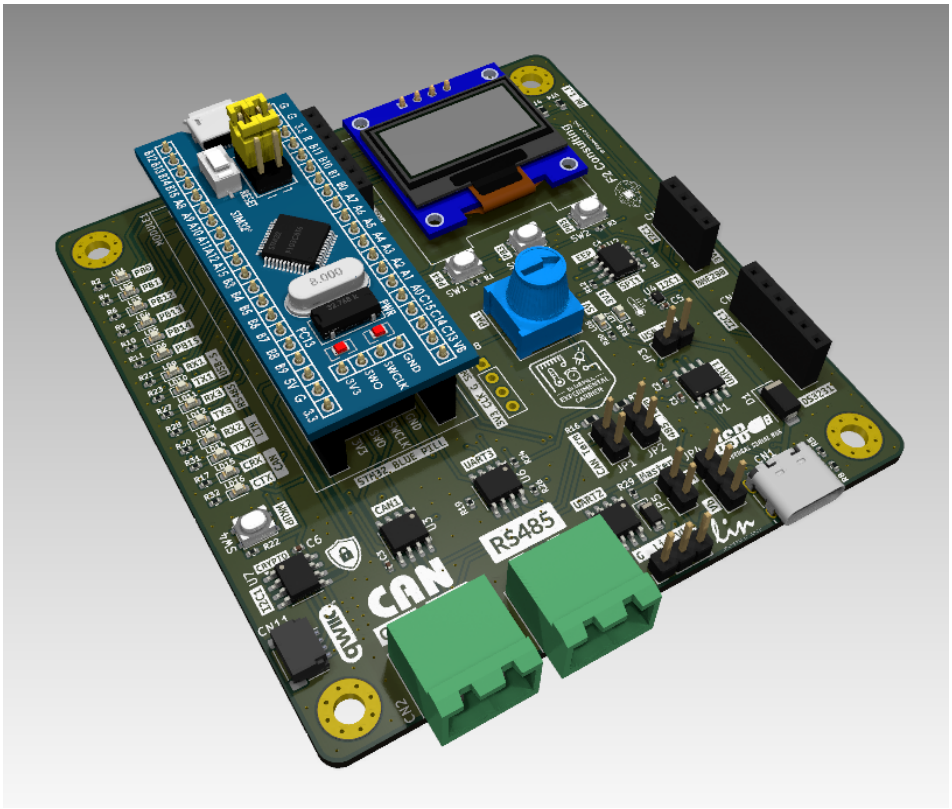

Figure 5.3: Blue Pill experimental carrier board.

the system is not overloaded. This structure of periodic tasks with priorities is appropriate for the test bench for two reasons. First, it allows a clear separation of responsibilities between the different firmware functions: one task is dedicated to managing serial communication with the PC, another to managing the CAN bus, another to executing test steps. Second, it ensures that the timing of test execution is determined internally by the board, independently of the communication latency with the host PC. The highest priority is assigned to tasks that monitor critical error conditions, such as a CAN communication error or a violated safety condition, so that they can interrupt test execution.

Development Tool: Sinergy

Firmware project development for the test board is supported by Sinergy, a graphical interface application developed by Francesco Ficili. It is a configuration and project generation tool that allows the software architecture of the firmware to be defined visually, without having to manually write all the infrastructure code. Through Sinergy it is possible to define system tasks with their respective priorities and

execution periods, configure CAN communication by directly importing a DBC file as the message database, and specify the ECU with the transmitted and received messages, automatically generating the data structures, signal read and write functions, and message send and receive functions. Sinergy then generates a complete project ready for the STM32CubeIDE development environment, also integrating the hardware configuration produced by STM32CubeMX, with basic peripherals already enabled, including the status LEDs. The additional peripherals needed for the test bench, such as analog inputs, PWM outputs, user buttons, need to be enabled manually in the STM32CubeMX configuration after the initial project generation. The integration of Sinergy into the development workflow significantly reduces the time needed to configure CAN communication and ensures consistency between the DBC database and the generated firmware code.

5.4 Communication Interfaces and Signal Simulation

This section describes the communication protocols and signal types involved in the test bench, providing for each a theoretical description of their operation and explaining the role they play in the test system.

UART - Universal Asynchronous Receiver/Transmitter

UART is an asynchronous serial communication protocol commonly used in embedded systems for its simple hardware implementation and the absence of a shared clock signal. Communication occurs over two lines, TX (transmit) and RX (receive), which are connected in a crossed configuration, where the TX of one is connected to the RX of the other. Data is transmitted in frames that usually include a start bit, a configurable number of data bits, an optional parity bit, and one or more stop bits. Since no clock signal is exchanged, both devices must be configured with the same baud rate and frame format before communication can occur reliably. In the test bench, UART is the communication channel between the host PC and the test board. Through this interface transit the test buffer sent from the PC to the board, the step-by-step updates produced by the board during execution, and the final result of the test session.

CAN - Controller Area Network

CAN is a serial communication protocol developed by Robert Bosch in the 1980s and later standardized in ISO 11898. It is designed for distributed embedded systems operating in electrically noisy environments, such as automotive and

industrial applications, and is widely used because of its robustness, built-in error handling, and priority-based access to the bus. CAN uses a two-wire differential bus, CANH and CANL, with a bus topology and termination resistors at both ends. Communication is broadcast-based: every transmitted frame is visible to all nodes on the network, while the identifier of the frame determines both its semantic meaning and its transmission priority, with lower identifiers having higher priority. The non-destructive arbitration mechanism allows simultaneous transmission attempts without data corruption, since the highest-priority frame wins access to the bus. In the test bench, CAN is the main communication channel between the test board and the VCU. The test board can transmit messages that emulate peripheral nodes such as the inverter and the BMS, and it receives and verifies the frames generated by the VCU according to the DBC definitions. The CAN network also allows a PC equipped with a CAN interface to observe traffic for debugging and validation purposes. The DBC file formally defines each message by specifying its identifier, transmitter, DLC, and signal properties, such as bit length, scaling, offset, and valid range. In this project, the DBC acts as the interface contract between the nodes and is therefore a central element for communication testing.

Digital Signals

Discrete digital signals are the simplest electrical interface type present in the system. A digital signal assumes one of two logic states, high or low, corresponding typically to supply voltage (3.3V or 5V) and ground, and is used to communicate binary conditions: a command is active or inactive, an error is present or absent, a limit has been reached or not. In the test bench, digital signals are generated by the test board through GPIO configured as outputs and sent physically to the corresponding input lines of the VCU. In the opposite direction, the VCU's digital outputs are read by the test board through GPIO configured as inputs. The status LEDs on the carrier board provide immediate visual feedback on the state of digital signals during test execution, which is useful for rapid debugging without having to analyse the serial trace. The user buttons on the carrier board can be used to simulate specific discrete inputs, such as an emergency stop button or an enable signal, during interactive test execution.

Analog Signals

Analog signals are continuous voltages used to represent physical quantities such as temperature, position, or current. In this test bench, analog behaviour is not produced by a dedicated analog output stage; rather, controlled input voltages are acquired through the ADC and used to emulate sensor-like conditions for

validation purposes. The carrier board includes a potentiometer that enables manual adjustment of the input voltage within the STM32F103C8T6 ADC range, supporting interactive testing and verification of the analog acquisition path. The 12-bit ADC provides 4096 discrete levels across the reference range, which is adequate for observing the VCU reaction to representative analog input variations.

PWM Signals

Pulse Width Modulation (PWM) is a digital modulation technique that encodes an analog command value into a periodic signal by varying the proportion of time the signal remains in the high state within each period, known as the duty cycle. A duty cycle of 50% means that the signal is high for half of the period, while a duty cycle of 100% corresponds to a constant high logic level. The STM32F103C8T6 timers support PWM generation on multiple independent channels, with configurable frequency and duty cycle. In the test bench context, PWM signals can be used to emulate actuator commands.

5.5 Functional Requirements for VCU Testing

This section translates the test bench capabilities into concrete functional requirements, organised by interface category. These requirements define what the test system must be able to verify about the VCU.

5.5.1 Input/Output Handling

The test bench must be able to generate digital, analog, and PWM signals directed at the VCU and read the corresponding responses. For digital signals, the system must be able to set and read the state of individual GPIO lines, simulating discrete commands, safety flags, and enable signals. For analog signals, the system must be able to apply controlled input voltages and verify that ADC acquisition returns the expected values. For PWM signals, the system must be able to generate waveforms with defined frequency and duty cycle. I/O management must be integrable with automated test logic: test steps must be able to set initial conditions through physical signals and verify VCU responses through physical signal reading and CAN message analysis.

5.5.2 Communication Management

The test bench must manage communication at two distinct levels. The first level is UART serial communication with the host PC, through which the test buffer, execution progress updates, and the final result are transferred. The test board

must ensure correct reception of the buffer, verifying the integrity of received data, and reliable transmission of updates during execution. The second level is CAN communication with the VCU, through which peripheral simulation messages are transmitted and received. At this level, the test bench must be able to transmit messages with identifiers, DLC, and data defined by the DBC, receive and decode messages produced by the VCU, verify that the content of received signals falls within expected values, manage periodic message transmission with the timing defined in the DBC, and detect error conditions on the bus. Error management must also be implementable through dedicated digital signals, allowing the VCU to communicate error conditions directly and the test board to inject fault signals to verify VCU behaviour in response.

5.5.3 Timing and Execution Verification

Embedded systems with periodic tasks have temporal requirements that must be respected to ensure correct operation. The test bench must be able to verify that VCU tasks are executed with the expected period by monitoring the frequency of periodic CAN messages and periodic signals produced by the VCU. On the test board side, timing is handled by CHAOS tasks, which ensure that test actions run with a fixed and deterministic timing, independent of PC communication delays. The separation between the serial communication task, which handles data exchange with the PC, and the test execution task ensures that delays in PC communication do not affect the internal timing of tests.

5.6 Non-Functional Requirements

Beside the functional requirements that define what the test bench must do, there are also higher-level properties that describe the overall quality of the system and its suitability for the project.

Modularity. As required in Section 4.1, the test system must accommodate new test cases without requiring changes to the board firmware or to the orchestration framework. Concretely, each test is an independent unit transferred via serial interface to the board, which receives, decodes, and executes it using the same internal logic regardless of content. On the communication side, updating the DBC file is sufficient to modify or extend the set of CAN messages handled by the system, without structural changes to the firmware.

Reusability. The test bench is not intended for a single VCU version, but for a class of interfaces and protocol definitions. If the VCU is updated or replaced by a different implementation that preserves the same external communication interfaces, the same test bench can be reused without structural modifications, requiring only updates to the DBC file and to the test cases.

Scalability. The carrier board is designed to support the integration of additional components and connectors, making it possible to add new physical peripherals such as sensors, actuators, or further CAN nodes, as the rover hardware evolves.

Independence from the final hardware target. As established in Chapters 3 and 4, all components of the test bench are selected and configured independently of hardware specifications still under definition. In practice, this means that the test board interfaces, the CHAOS task configuration, and the OpenHTF orchestration layer do not assume any specific VCU hardware variant, allowing the infrastructure to remain valid across different implementation choices.

Chapter 6

Implementation and Validation

This chapter presents the experimental setup, the firmware implementation, and the validation activities carried out during the thesis work. The first section introduces the hardware configuration and the monitoring tools used during development. After that, there is a description of the firmware architecture, communication management, and signal simulation. Then, the structure of the test cases and the encoding mechanism used to transfer tests to the board are presented. The next section outlines the orchestration workflow managed by OpenHTF. Finally, the experimental results are reported and the current limitations of the system are discussed.

6.1 Experimental Setup and Hardware Configuration

The experimental setup was realised using two Blue Pill experimental carrier boards described in Chapter 5. One board acted as the test board, executing the test firmware developed in this work, while the second was configured to simulate the VCU through a simple loopback firmware that simulates the VCU behaviour. This approach made it possible to verify the correctness of the physical connections, communication protocols, and test execution logic before the final hardware platform became available.

The physical connections between the components of the setup are as follows. Both boards are powered via USB-C: the VCU simulation board receives power from a dedicated USB port, while the test board is connected to the host PC through USB-C, which provides both power and the serial communication channel.

The CAN bus is established by connecting the CANH and CANL signals between the CAN transceivers of the two boards, with 120 Ω termination resistors at both ends of the bus, as required by the ISO 11898 standard. It is also worth noting that the CAN transceiver on the carrier board requires a 5 V supply, which is provided directly by the USB connector. The Vector VN1610 adapter is also connected to the CAN bus via a DB9 connector, enabling real time monitoring of CAN traffic from the host PC without interfering with the communication between the two boards. Digital signals between the boards are managed through cables connected directly to GPIO pins configured as inputs or outputs. Also a common ground connection is required to ensure the correct interpretation of the logic levels. Firmware flashing on both boards was performed using an ST-Link V2.

For CAN traffic monitoring and debugging from the PC, SavvyCAN was used, an open-source software tool for CAN bus analysis [7]. SavvyCAN displays CAN frames transmitted on the bus in real time, showing the identifier, DLC, payload, and timestamp of each message. This functionality is particularly useful during the early development phases to verify the correct transmission and reception of messages defined in the DBC database, as well as the timing behaviour of periodic frames.

6.2 Test Bench Implementation

6.2.1 Hardware and Firmware Configuration

The test board firmware was developed starting from the project automatically generated by Sinergy, the configuration tool described in Section 5.3. Sinergy generated the base STM32CubeIDE project with the CHAOS operating system already integrated, the task scheduling configured, and the CAN communication imported directly from the DBC file.

The tasks defined in CHAOS follow a priority-based scheduling scheme. The following tasks were configured, listed in order of decreasing priority:

- **CAN task** (10 ms period): manages the reception and transmission of CAN messages, updates incoming signal variables, and cyclically transmits messages configured as persistent. It has the highest priority to ensure deterministic communication timing.
- **read_cmd task** (100 ms period): manages the reception of commands received through the UART interface from the host PC.
- **1000 ms task**: executes tests assigned to `task_id = 1000`.
- **500 ms task**: executes tests assigned to `task_id = 500`.

- **100 ms task**: executes tests assigned to `task_id = 100`.
- **10 ms task**: executes tests assigned to `task_id = 10`.

This structure allows each test to be assigned to the task most suitable for its timing requirements. As discussed in Section 4.4, this design ensures that test execution timing is governed internally by the RTOS scheduler, independently of any latency in UART communication with the host PC.

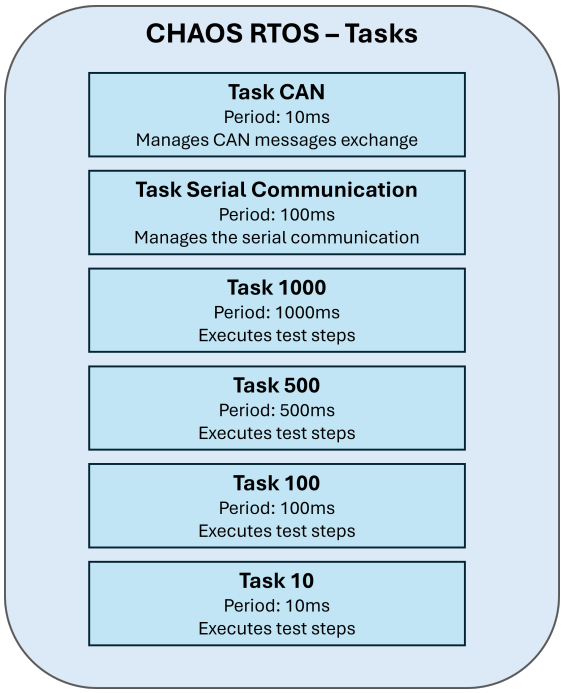

Figure 6.1: Firmware architecture of the test board based on the CHAOS RTOS. Tasks are scheduled according to priority and executed periodically by the RTOS scheduler.

Peripheral initialisation was performed using STM32CubeMX, integrating the project generated by Sinergy. The initialised peripherals include: status LEDs on pins PB0, PB1, PB12, PB13, and PB15 configured as GPIO outputs; the LED on PB14 configured as a PWM output on the TIM1 timer channel for visual debugging of PWM intensity; button SW1 on PB4 configured with EXTI interrupt and SW2 on PB5 configured as a GPIO input; PC13 configured as GPIO output and PC14 as GPIO input; the analog channel on PA1 connected to the carrier board potentiometer and acquired through the 12-bit ADC; UART serial communication via USART1 in asynchronous mode; and the CAN controller with the external transceiver on the carrier board.

6.2.2 Communication Management

UART Serial Communication

Serial communication between the host PC and the test board is managed through the `serial_communication` module. Reception is interrupt driven: each byte received on UART is accumulated in an internal buffer via the `HAL_UART_RxCpltCallback` callback, which updates the timestamp of the last received byte and immediately re-enables reception of the next one. Message completion is detected by an inactivity timeout: if no new bytes are received within 10 ms, the string is considered complete.The receive buffer has a size of 512 bytes, chosen to hold the complete encoded test string within a single UART transfer, as discussed in Section 6.3.2.

The module provides functions for both transmission and reception over UART. Reception is confirmed by monitoring an inactivity timeout on the received byte stream, after which the accumulated string is made available for processing. Incoming strings are parsed to recognise a set of predefined commands; when the `test` command is received, the board enters buffer reception mode and interprets the next incoming string as the encoded HEX buffer of the test to be executed.

CAN Communication

The CAN database was defined using CANdb++ and imported into Sinergy for the automatic generation of data structures and message handling functions. The database used is a simple example that includes six messages across two nodes, VCU and Test_Board, and operates at 500 kbit/s. The messages are summarised in Table 6.1 and described below.

Message name	ID	TX	RX	Signals
VCU_status_msg	0x001	VCU	Test_Board	`VCU_state` (8 bit) `VCU_error_flag` (1 bit)
TestBoard_status_msg	0x002	Test_Board	VCU	`TestBoard_state` (8 bit) `TestBoard_error_flag` (1 bit)
Digital_in_msg_01	0x111	VCU	Test_Board	`Digital_in_01–04` (1 bit each)
Digital_out_msg_01	0x112	Test_Board	VCU	`Digital_out_01–04` (1 bit each)
Analog_in_msg_01	0x101	VCU	Test_Board	`Analog_in_01`, `Analog_in_02` (16-bit signed each)
Analog_out_msg_01	0x102	Test_Board	VCU	`Analog_out_01`, `Analog_out_02` (16-bit signed each)

Table 6.1: CAN messages defined in the DBC database.

The naming convention follows the perspective of the test board: 'in' and 'out'

refer to the direction of data with respect to the Test_Board, mapping each message directly to its role in the simulation. This approach improves clarity during test development and execution, as it directly maps each message to its role in the simulation and validation process.

In particular, the status messages (`VCU_status_msg`, `TestBoard_status_msg`) serve as synchronisation and monitoring frames: each board transmits its current execution state and an error flag, allowing the other node to track progress and detect fault conditions during a test session. The digital messages carry binary flag signals used to simulate discrete commands and digital acknowledgements between the two nodes. The analog messages carry 16-bit signed integer values used to exchange numeric data such as ADC readings or reference setpoints.

Sinergy automatically generated, from the DBC file, the data structures for each message, the functions to set and read individual signals, and the functions for CAN frame transmission and reception. This significantly reduced configuration effort and guaranteed consistency between the message database and the generated firmware code.

Signal Simulation

The physical signals exchanged between the test board and the VCU are managed using the STM32 HAL drivers.

Digital signals are managed via GPIO pins configured as outputs or inputs. Writing a signal uses `HAL_GPIO_WritePin`, while reading uses `HAL_GPIO_ReadPin`. The status LEDs provide immediate visual feedback during test execution, enabling quick debugging without analysing the serial log. Button SW1, configured with an EXTI interrupt, can also be used to simulate asynchronous discrete signals such as an emergency stop or an enable line.

The **analog signal** is acquired via the 12-bit ADC on channel PA1, connected to the carrier board potentiometer. The value is read through `HAL_ADC_Start`, `HAL_ADC_PollForConversion`, and `HAL_ADC_GetValue`, returning an integer in the range 0-4095, corresponding to the full converter span. During test execution, this value can be saved as a runtime value and subsequently transmitted via CAN to the VCU, as demonstrated in the test cases described in Section 6.3.3.

The **PWM signal** is generated by timer TIM1 on pin PA8, with configurable frequency and duty cycle. The output is started with `HAL_TIM_PWM_Start`, while duty cycle updates are applied using the function `__HAL_TIM_SET_COMPARE`. The LED on PB14 is connected to the complementary channel of the same timer and provides visual debugging of the PWM output: luminous intensity increases proportionally with duty cycle, making it easy to verify the correct behaviour of PWM generation without additional measurement equipment.

6.3 Development of Test Cases

6.3.1 Test Structure and Execution Management

Tests are defined through a three levels hierarchical structure consisting of the test, steps, and actions. This structure is based on the `TestBuffer_t` type defined in `test.h` and managed by the `test` module.

The **test** is the main execution unit, identified by a unique `test_id` and associated with a specific CHAOS task through the `task_id` field. At each cyclic invocation of the assigned task, the function `Test_ServiceForTask` advances the test by one step. Execution timing is therefore determined entirely by the RTOS scheduler.

Each **step** represents an execution unit corresponding to a single task invocation. A step is composed of at most `TEST_MAX_ACTIONS_PER_STEP` actions (currently set to 4), executed sequentially within the same task cycle. Steps support a conditional branching mechanism based on runtime values: through the fields `branch_runtime_index`, `branch_next_step_if_zero`, and `branch_next_step_if_nonzero`, it is possible to define conditional jumps to non-sequential steps depending on a value saved at runtime. A step can also be marked as terminal via the `is_final` flag, which causes the test to conclude immediately after its execution regardless of remaining steps.

To support data exchange between steps, the framework provides a runtime value system consisting of a shared array of 32-bit signed integers accessible from all actions through `Test_RuntimeSetValue` and `Test_RuntimeGetValue`. It provides a simple mechanism to pass information between steps, enabling the use of dynamically acquired values, such as a potentiometer reading or a received CAN signal, as input parameters for subsequent actions.

The actions represent the elementary operations executed within a step. They operate on runtime values and interface with the available hardware and communication channels. The supported action types are listed below, grouped by interface category.

Digital I/O:

- `GPIO_WRITE`: sets a GPIO pin to SET or RESET;
- `GPIO_CHECK`: reads a GPIO pin and compares it with the expected state, marking the step as failed if the values differ;
- `GPIO_READ_SAVE`: reads a GPIO pin and stores the value in the runtime value array.

Analog acquisition:

- `READ_ANALOG`: acquires the ADC value on a specified channel and verifies that it falls within a configurable range;
- `READ_ANALOG_WAIT`: acquires the ADC value with periodic serial output at a configurable rate, waits for operator confirmation via UART, and saves the final value as a runtime value.

PWM generation:

- `SET_PWM`: sets the duty cycle of a PWM channel, optionally using a previously saved runtime value instead of a static parameter.

CAN communication:

- `CAN_WRITE`: transmits one or more CAN messages. If the `persistent` flag is set, the transmitted values are also stored in the CAN database, allowing the cyclic CAN task to continue sending them periodically until they are changed. Otherwise, the message is transmitted only once.
- `CAN_READ`: verifies that a CAN message is received within a specified timeout and that the value of the indicated signal falls within the expected range;
- `CAN_READ_SAVE`: reads the value of a signal from a received CAN message and saves it as a runtime value.

Flow control:

- `INSERT_VALUE`: writes a value directly into the runtime value array;
- `WAIT`: suspends step progression for a configurable number of task invocations;
- `WAIT_CONFIRM`: suspends execution until an explicit operator confirmation is received via the serial interface.

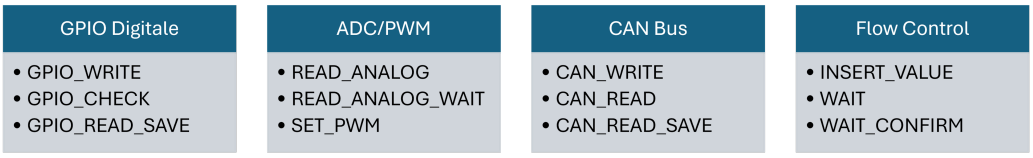

Figure 6.2: Overview of the action types supported by the test execution framework, organised by hardware interface category, including GPIO, ADC/PWM, CAN, and flow control actions

The firmware also provides a **global fault mechanism**, managed by the volatile variable `g_test_fault`. A fault can be raised from any execution context, including interrupt callbacks and CAN receive handlers, and causes the running test to be aborted immediately.

6.3.2 Test Buffer Encoding and Decoding

To transfer a test definition from the host PC to the test board over UART, a binary encoding and decoding system was developed. Tests are defined as JSON files containing an array of test objects, each specifying the test identifier, the associated task, and the sequence of steps and actions with their parameters.

The Python script `test_encoding.py` converts each test object into a structured binary buffer by serialising the data structures into an ordered byte sequence according to a fixed internal format. The binary buffer is then represented as an uppercase ASCII hexadecimal string (HEX string), ready to be transmitted over UART in a single transfer. The script `pack_all_tests.py` automates this process for all JSON files in the `utils/tests/` directory and produces a human readable report file, `tests_report.txt`, summarising for each test the identifier, description, transmittable HEX string, and the full detail of every step and action. This report serves as the primary reference document for test inspection and manual execution.

The HEX string begins with the ASCII header `TBUF`, which allows the test board to confirm that the received string is a valid test buffer before attempting to decode it. A CRC-16 check is used as an additional integrity measure to detect transmission errors and prevent loading corrupted test structures. The CRC is appended to the end of the binary buffer and verified on reception by recomputing it over the received data. If the CRC does not match, the test buffer is rejected and not decoded.

When received, the test board decodes the buffer through the `test_decoding` module, in particular with the `TestDecoding_DecodeHexBuffer` function. This function reconstructs the complete `TestBuffer_t` structure from the HEX string, including all steps and actions with their respective parameters.

The 512-byte UART receive buffer is sized to accommodate the HEX representation of the largest possible test under the current structural limits: 5 steps and 4 actions per step (`TEST_MAX_STEPS` and `TEST_MAX_ACTIONS_PER_STEP`). This guarantees that any valid test is transferred in a single UART transaction without fragmentation. The same limits also ensure that the decoded structure fits within the 20 KB SRAM available on the STM32F103C8T6.

6.3.3 Example Test Cases

Test case development followed an incremental approach. In the initial phase, elementary tests were written to validate each action type in isolation: GPIO write and read tests, ADC acquisition tests, PWM generation tests, and CAN transmission and reception tests with payload verification. These unit level tests confirmed the correct behaviour of each firmware component before composing more complex scenarios.

In the next phase, two more complex tests were developed, combining multiple action types in a structured sequence. These represent the most significant validation cases implemented in this work and are described below.

Test 01 exercises bidirectional digital and CAN communication between the test board and the simulated VCU. The test is organised into five steps. In step 1, the test board sets a debug LED and transmits a `TestBoard_status_msg` with `TestBoard_state = 1`. In step 2, it verifies that the simulated VCU has acknowledged the message by asserting pin PC14, then advances the status to state 2 and sets an additional LED, indicating the correctness of the VCU response. In step 3, the value of signal `Digital_in_01` is read from the incoming `Digital_in_msg_01` frame and stored as runtime value with index 0. The conditional branch configured on step 3 then directs execution: if the signal is 0, step 4 is executed, turning on the LED associated with that outcome and setting `TestBoard_state = 4`; if the signal is 1, step 5 executes, activating the corresponding LED and state 5. Both steps 4 and 5 are marked as final. The test therefore verifies bidirectional CAN communication, GPIO monitoring, the runtime value mechanism, and the conditional branching logic within a single session.

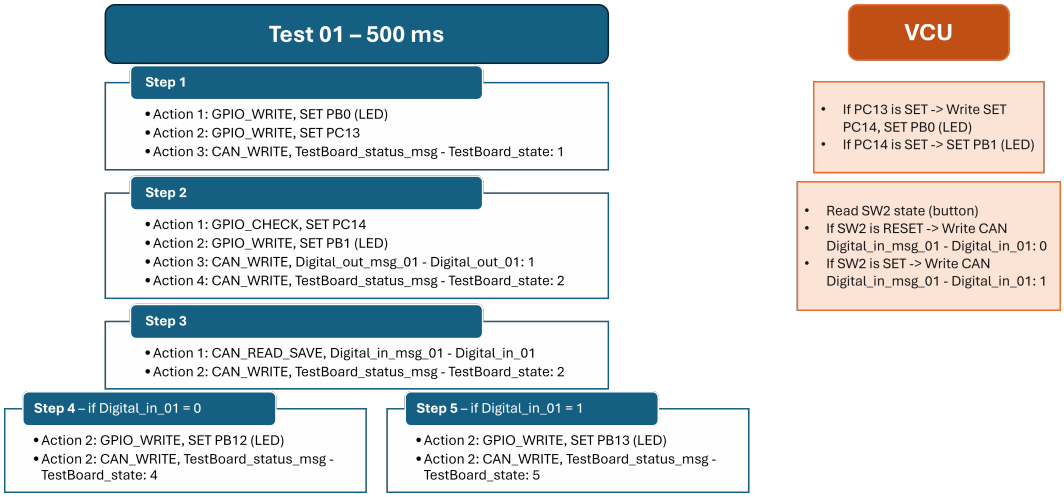

Figure 6.3: Sequence diagram of Test 01, illustrating the test steps and actions as well as the VCU behavior during test execution. The board executes test steps periodically under RTOS control. The conditional branch is shown at the bottom, with the two possible execution paths.

Test 02 verifies the signal flow from analog acquisition on the test board to PWM duty cycle generation on the simulated VCU, based on a CAN reference value. Steps 1 and 2 replicate the connection verification sequence of Test 01. In step 3, the test board starts continuous ADC acquisition on the potentiometer channel, printing the current value to the serial interface every 500 ms and waiting for the

operator to confirm the desired value via UART; the confirmed ADC reading is saved as runtime value 0. In step 4, the saved value is placed in the `Analog_out_01` signal of `Analog_out_msg_01` and transmitted persistently via CAN, so that the VCU receives the reference. The flag `Digital_out_03` in `Digital_out_msg_01` is also set to instruct the simulated VCU to apply the received value as the PWM duty cycle reference. This test validates analog signal acquisition, runtime value propagation across steps, transmission of numerical data over CAN, operator interaction through the serial interface, and the complete path from analog input to PWM output.

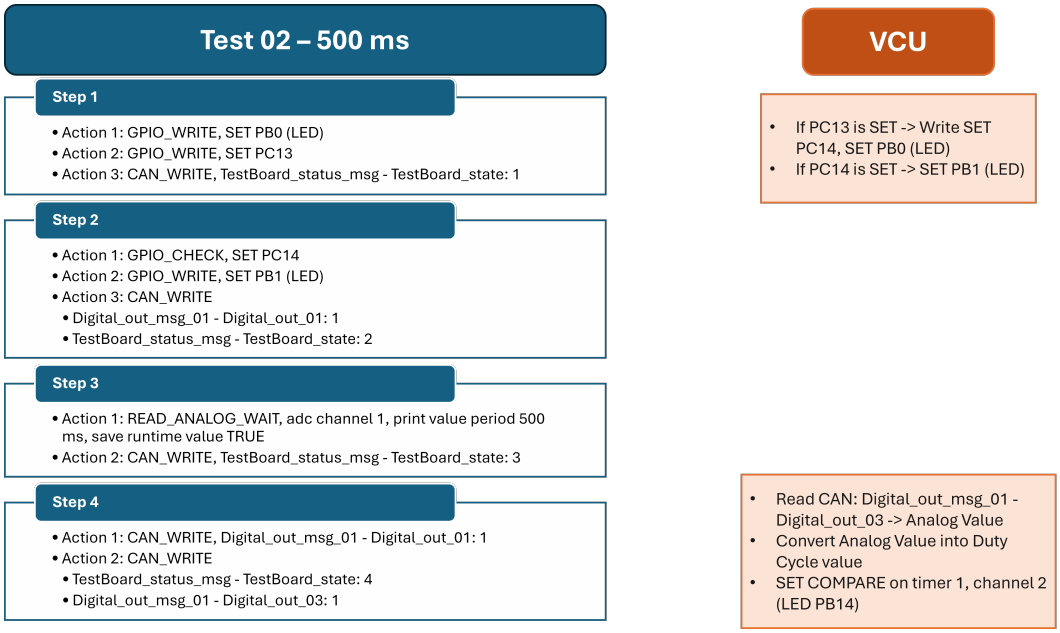

Figure 6.4: Sequence diagram of Test 02, illustrating the test steps and actions, and the VCU behavior during test execution. The board executes test steps periodically under RTOS control.

6.4 Test Execution Workflow with OpenHTF

The test session is orchestrated by the host PC using the OpenHTF framework [11], implemented in `main.py` with the individual test phases defined in `openhtf_serial_phases.py`. The session proceeds through four sequential phases, as illustrated in Figure 6.6.

The first phase, `serial_connection_check`, verifies that the serial link between the PC and the test board is active. The PC transmits the `ping` command and waits for the corresponding `ping` response within a configurable timeout. This

phase confirms that the physical connection is intact and that the board firmware is running before any test is loaded.

The second phase, `user_test_selection`, presents the available tests to the operator through the OpenHTF web GUI and waits for a selection. Each test is listed by name and description. If a test case has not yet been integrated into the Python project, the operator can alternatively provide the HEX string directly, enabling manual transfer without modifying the orchestration code.

The third phase, `firmware_load_test`, transfers the selected test to the board. The PC first sends the `test` command to activate the board's buffer reception mode, then transmits the complete HEX string in a single UART transfer. The phase monitors the board response to confirm that the buffer was received and decoded successfully.

The fourth phase, `firmware_execute_test`, triggers and monitors test execution. OpenHTF sends the start confirmation, after which the board executes the test autonomously according to its internal RTOS schedule. During execution, the board transmits step-by-step updates via UART reporting, for each step and action, the name, execution status, measured or read values, and partial result. The OpenHTF logger captures these messages in real time and attaches them to the session record. At the end of execution, the board transmits the overall pass or fail outcome, which OpenHTF records as the final phase measurement.

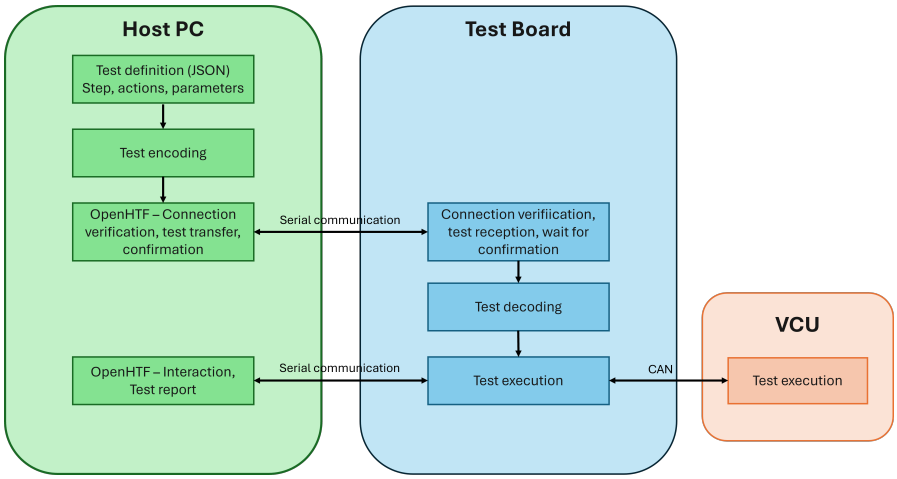

Figure 6.5: Test execution workflow across Host PC, Test Board, and VCU. The Host PC defines the test in JSON format, encodes it into a buffer, and transfers it via serial communication using OpenHTF for connection management and confirmation. The Test Board receives the buffer, decodes the test definition, and executes it locally. During execution, the Test Board interacts with the VCU via CAN bus. Execution status and results are reported to the Host PC, where OpenHTF generates the final report.

The web GUI displays session progress, logger output, and measurement values in real time. When a test step requires operator input, such as the potentiometer confirmation in Test 02, the GUI presents an interactive prompt and suspends execution until the operator responds. Figure 6.6 shows the web GUI during an active test session.

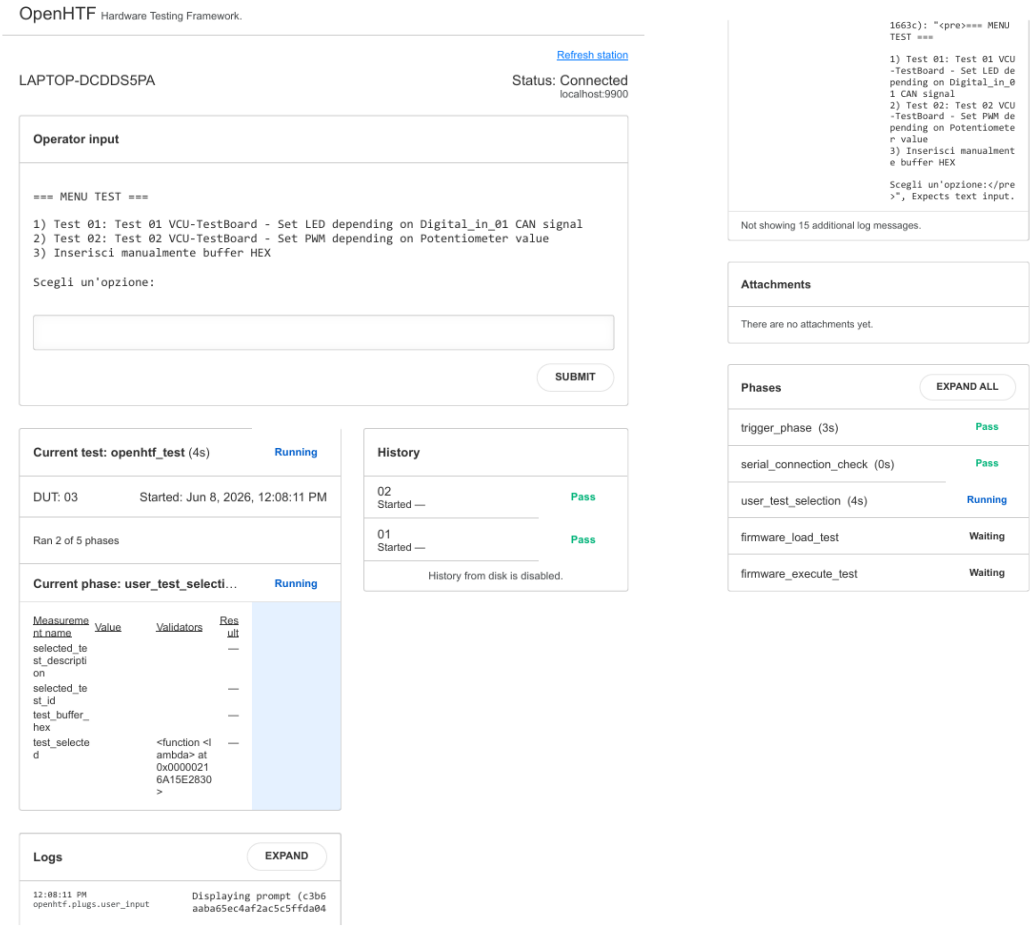

Figure 6.6: OpenHTF web GUI captured during test execution. The figure shows operator input via the command menu, the current test, the real-time log, the report history, and the status of past, passed/failed, ongoing, and upcoming test phases.

At the end of the session, OpenHTF produces a structured JSON report containing the session identifier, start and end timestamps, the outcome of each phase, all recorded measurements with their acceptance criteria, and the complete execution log. A CSV copy of the report is also produced through a custom output callback registered in `main.py`. Together, these documents provide a complete

and auditable trace of the test session, suitable for both development analysis and project validation documentation. An excerpt of the produced JSON report is shown in Figure 6.7.


```
{
  "dut_id": "01",
  "start_time_millis": 1780914323307,
  "end_time_millis": 1780914332593,
  "outcome": "PASS",
  "metadata": {
    "test_name": "openhtf_test",
    "config": {
      "station_id": "LAPTOP-DCDDS5PA"
    }
  },
  "phases": [
    {
      "name": "serial_connection_check",
      "outcome": "PASS",
      "measurements": {
        "ping_sent": "ping",
        "ping_received": "ping",
        "connection_ok": true
      }
    },
    {
      "name": "user_test_selection",
      "outcome": "PASS",
      "measurements": {
        "selected_test_id": "01",
        "selected_test_description": "Test 01 VCU-TestBoard - Set LED depending on Digital_in_01
        CAN signal",
        "test_buffer_hex": "..."
      }
    },
    {
      "name": "firmware_load_test",
      "outcome": "PASS",
      "measurements": {
        "test_loaded": true,
        "buffer_sent": true
      }
    },
    {
      "name": "firmware_execute_test",
      "outcome": "PASS",
      "measurements": {
        "test_ok": true,
        "steps_total": 4,
        "steps_passed": 4
      }
    }
  ]
}
```


Figure 6.7: Excerpt of the OpenHTF JSON report for Test 01, showing session metadata, phase outcomes, and recorded measurements.

6.5 Experimental Results

The two scenario tests were executed on the setup described in Section 6.1, with the test board connected to the VCU simulation board via CAN bus and direct digital signal lines. The results confirmed the correct operation of all the main implemented functionalities.

Serial communication operated correctly across all test sessions. Message reception based on interrupts, combined with an inactivity timeout, reliably handles incoming messages, including full test HEX strings up to the maximum supported length. All predefined commands were correctly recognised by the system, and the test buffer reception mode was activated and cleared as expected.

Figure 6.8 shows the complete UART serial output captured during a Test 01 session in which `Digital_in_01` was read as 0, directing execution to step 4. The test was also executed with `Digital_in_01 = 1`, in which case the branch correctly directed execution to step 5, activating the corresponding LED and state, as shown in Figure 6.9.

```
---- Sent utf8 encoded message: "test" ----
[TEST] Paste the HEX buffer of the test and press ENTER.
[TEST] When ready, the board will ask you YES/NO.
---- Sent utf8 encoded message: "5442554605010 0F401050003000100000100020D000109A50102000000000801
000000000000000010003010 20E010000010100010 9A5021201000000080100000000000000020000000008020000000000
0000001040003040 20B11010000006400000001001353616C766120446967697461 6C5F696E5F303109A5010200000000
0803000000000000000010102000 10C000109A5010200000000080100000000000000001010200010D000109A5010200000
000080500000000000000001A5C0" ----
[TEST] Test 01 loaded (ID 1). Start the test? YES/NO
---- Sent utf8 encoded message: "yes" ----
[TEST] Starting Test 01
[TEST] Test 01 - step 1/5
[TEST] Executing step 1:
[TEST]   action PB0 ON -> wrote SET
[TEST]   action PC13 ON -> wrote SET
[TEST]   action CAN TX x1 -> TX ID 0x2 CH 0 DLC 8 DATA 01 00 00 00 00 00 00 00
[TEST]   action CAN TX x1 -> Com_SetSignal id=1 val=1
[TEST]   action CAN TX x1 -> Com_SetSignal id=2 val=0
[TEST] Step 1 executed correctly
[TEST] Executing step 2:
[TEST]   action PC14 IS ON -> expected SET, read SET
[TEST]   action PB1 ON -> wrote SET
[TEST]   action CAN TX x2 -> TX ID 0x112 CH 0 DLC 8 DATA 01 00 00 00 00 00 00 00
[TEST]   action CAN TX x2 -> TX ID 0x2 CH 0 DLC 8 DATA 02 00 00 00 00 00 00 00
[TEST]   action CAN TX x2 -> Com_SetSignal id=3 val=1
[TEST]   action CAN TX x2 -> Com_SetSignal id=4 val=0
[TEST]   action CAN TX x2 -> Com_SetSignal id=5 val=0
[TEST]   action CAN TX x2 -> Com_SetSignal id=6 val=0
[TEST]   action CAN TX x2 -> Com_SetSignal id=1 val=2
[TEST]   action CAN TX x2 -> Com_SetSignal id=2 val=0
[TEST] Step 2 executed correctly
[TEST] Executing step 3:
[TEST] CAN_READ_SAVE saved value[0] = 0
[TEST]   action CAN TX x1 -> TX ID 0x2 CH 0 DLC 8 DATA 03 00 00 00 00 00 00 00
[TEST]   action CAN TX x1 -> Com_SetSignal id=1 val=3
[TEST]   action CAN TX x1 -> Com_SetSignal id=2 val=0
[TEST] Step 3 executed correctly
```

```
[TEST] Executing step 4:
[TEST]   action PB12 ON -> wrote SET
[TEST]   action CAN TX x1 -> TX ID 0x2 CH 0 DLC 8 DATA 01 00 00 00 00 00 00 00
[TEST]   action CAN TX x1 -> Com_SetSignal id=1 val=1
[TEST]   action CAN TX x1 -> Com_SetSignal id=2 val=0
[TEST] Step 4 executed correctly

[TEST] REPORT - Test 01
[TEST] Task ID: 500
[TEST] Step 1 - : OK
[TEST] Step 2 - : OK
[TEST] Step 3 - : OK
[TEST] Step 4 - : OK
[TEST] Step 5 - : SKIPPED
[TEST] Result: PASSED (OK=4, FAIL=0, SKIPPED=1, TOTAL=5)
```

Figure 6.8: Complete UART serial output captured during Test 01 execution with `Digital_in_01 = 0` including command exchange, step progression, and final report.

```
...

[TEST] Executing step 3:
[TEST] CAN_READ_SAVE saved value[0] = 1
[TEST]   action CAN TX x1 -> TX ID 0x2 CH 0 DLC 8 DATA 03 00 00 00 00 00 00 00
[TEST]   action CAN TX x1 -> Com_SetSignal id=1 val=3
[TEST]   action CAN TX x1 -> Com_SetSignal id=2 val=0
[TEST] Step 3 executed correctly
[TEST] Executing step 5:
[TEST]   action PB13 ON -> wrote SET
[TEST]   action CAN TX x1 -> TX ID 0x2 CH 0 DLC 8 DATA 05 00 00 00 00 00 00 00
[TEST]   action CAN TX x1 -> Com_SetSignal id=1 val=5
[TEST]   action CAN TX x1 -> Com_SetSignal id=2 val=0
[TEST] Step 5 executed correctly

[TEST] REPORT - Test 01
[TEST] Task ID: 500
[TEST] Step 1 - : OK
[TEST] Step 2 - : OK
[TEST] Step 3 - : OK
[TEST] Step 4 - : SKIPPED
[TEST] Step 5 - : OK
[TEST] Result: PASSED (OK=4, FAIL=0, SKIPPED=1, TOTAL=5)
```

Figure 6.9: Excerpt of the UART serial output captured during Test 01 execution with `Digital_in_01 = 1`, in which case the branch correctly directed execution to step 5.

CAN communication at 500 kbit/s operated correctly in all tested scenarios. All messages defined in the DBC database were transmitted and received with the expected identifier, DLC, and payload content. Periodic messages were transmitted at the cadence of the 10 ms CAN task, which was confirmed by inspecting the

inter-frame intervals in SavvyCAN captures. The persistent `CAN_WRITE` mechanism correctly maintained updated signal values across multiple test steps, so that the simulated VCU always received current data without requiring retransmission from the test logic. Figure 6.10 shows the SavvyCAN interface during an execution.

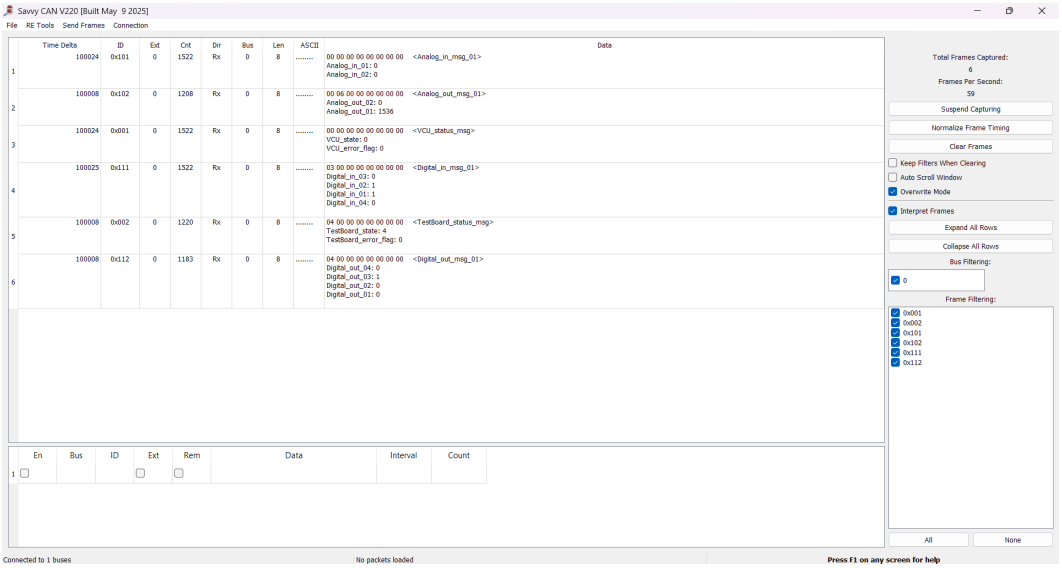

Figure 6.10: SavvyCAN interface showing real-time CAN bus monitoring, including transmitted and received frames.

GPIO actions produced results consistent with the expected values in all test steps. The LED feedback on the carrier board provided immediate confirmation of digital write operations during development, without requiring serial log analysis.
Analog acquisition delivered values consistent with the physical position of the potentiometer across the full 12-bit range (0-4095). The `READ_ANALOG_WAIT` action correctly printed the current ADC value to the serial interface at the configured rate, allowing the operator to observe the value in real time and confirm when the desired position was reached.
PWM generation was verified both through the visual intensity of the LED on PB14, which tracks the duty cycle applied to the complementary timer channel, and through the correct reception of the duty cycle value in the VCU simulation logic. In Test 02, the ADC reading confirmed by the operator was propagated correctly through the runtime value system, transmitted via CAN, and applied as a PWM duty cycle reference by the simulated VCU.
Conditional branching in Test 01 directed execution to the correct step in both possible outcomes, based on the value of `Digital_in_01` read from the simulated VCU, confirming the correct interaction between the `CAN_READ_SAVE` action and the runtime value branching logic.

Regarding **OpenHTF report readability**, the JSON report correctly captures the overall structure of the test session, including the outcome of each phase and the measurements explicitly recorded by the orchestration layer. However, the detailed execution trace produced by the test board is still sent via serial messages and stored mainly as log output, rather than being structured into step and action level. As a result, the final report is useful for documenting the test session, but still requires manual inspection for more detailed analysis of the execution flow.

6.6 Discussion of System Limitations

The implemented system achieved the main objectives defined in Chapter 5, but several limitations remain that are worth discussing explicitly, both as an honest assessment of the current state of the work and as a basis for the future developments outlined in Chapter 7.

Report granularity and log readability. The JSON report produced by OpenHTF correctly records the session metadata, the outcome of each phase, and the measurements explicitly collected during orchestration. Nevertheless, the detailed execution trace generated by the test board is currently transmitted as serial text and recorded primarily as log output, which limits the readability of the final report when a step-by-step analysis is required. A useful improvement would be to modify the OpenHTF report structure to include UART messages as hierarchical measurements associated with individual steps and actions.

Structural limits on steps and actions. The values of `TEST_MAX_STEPS = 5` and `TEST_MAX_ACTIONS_PER_STEP = 4` were chosen to respect the constraints of the 20 KB SRAM and the 512-byte UART buffer simultaneously. These limits are sufficient for the test cases developed in this work, but they may become restrictive for more articulated scenarios in later integration stages. A possible approach to relaxing them without changing the buffer size limit would be to adopt a more compact binary serialisation for action types that carry few parameters, reducing the per-action overhead in the HEX string.

VCU simulation fidelity. The second Blue Pill experimental carrier board used as a VCU simulator reproduces only a functional subset of the real VCU's behaviour, sufficient to exercise the test execution logic and the communication interfaces. A complete validation process will require the developed test cases to be executed against the actual VCU hardware, verifying that the real firmware responds to the test board signals as expected and that any discrepancies are identified and resolved before system integration.

Modularity and scalability. As a concluding remark, it is worth noting that the modularity and scalability requirements defined in Section 4.1 were met by the implemented architecture. Adding a new test case requires only authoring a

JSON file and running `pack_all_tests.py`; no firmware or framework changes are needed. Extending the CAN message set requires updating the DBC file and regenerating the project through Sinergy, with minimal impact on the existing firmware. Additional physical peripherals can be integrated using the expansion connectors available on the carrier board, preserving full compatibility with the existing project structure.

Chapter 7

Conclusions and Future Work

7.1 Summary of the Developed Test Bench

This thesis presents the design and implementation of a modular HIL-inspired test bench for the validation of the Vehicle Control Unit of the e-RO.Alps agricultural rover. The work was motivated by the early development stage of the project, in which the final hardware platform was not yet available, making it necessary to develop an alternative infrastructure capable of supporting firmware validation independently of the physical system. The test bench is composed of three main elements. The host PC runs the OpenHTF orchestration software, which manages the test session by verifying the serial connection with the test board, presenting test options to the operator, transferring the selected test as an encoded HEX string over UART, starting execution, and collecting the final result in a structured JSON report. The test board, based on the STM32F103C8T6 microcontroller mounted on an experimental carrier board and running the CHAOS RTOS, acts as the autonomous test executor. It receives and decodes the test buffer, executes the defined steps and actions according to its internal RTOS schedule, exchanges CAN messages and physical signals with the VCU under test, and reports progress and results in real time via the serial interface. The VCU under test is connected to the test board through the CAN bus and direct digital signal lines, with a shared ground reference ensuring correct logic level interpretation. Tests are defined externally as JSON files, encoded into compact binary HEX strings by a Python utility, and transferred to the board at runtime without requiring firmware recompilation. The internal test execution model is organised as a three-level hierarchy of tests, steps, and actions. Steps are executed at the cadence of the RTOS task to which each test is assigned, ensuring deterministic timing independently of host communication

latency. Actions cover all supported interface types: GPIO write and read, ADC acquisition, PWM generation, CAN message transmission and reception, runtime value management, conditional branching, and operator interaction. A runtime value array provides a shared data space accessible across steps, enabling dynamic test scenarios in which values measured at one step, such as an ADC reading or a received CAN signal, can be used as parameters in subsequent actions. The system was validated in an experimental setup in which a second identical carrier board simulated the VCU behaviour through a loopback firmware. Two scenario tests were executed successfully, covering bidirectional CAN and digital communication with conditional branching and an end-to-end chain from analog acquisition to PWM duty cycle control via CAN-transmitted reference values. CAN traffic was independently monitored using SavvyCAN and a Vector VN1610 adapter, confirming correct message content and transmission timing.

7.2 Main Contributions of the Thesis

The main contributions of this thesis can be grouped into four areas.

Design of a modular HIL-inspired test bench architecture. The proposed architecture positions the test infrastructure between classical integration testing and full HIL simulation. The firmware runs on a real microcontroller and interacts with physical interfaces, while missing peripherals are replaced by a dedicated test board that emulates their communication and signal behaviour. This approach allows the validation of embedded interfaces in the absence of the final rover hardware.

Development of a flexible test execution framework. The test execution model, based on a hierarchical structure of tests, steps, and actions, is encoded in JSON and transferred at runtime as binary HEX strings. This allows new test cases to be defined and executed without modifying the board firmware or the orchestration layer. Delegating execution control to the embedded RTOS decouples test timing from host communication latency, addressing one of the main requirements of embedded test infrastructures. The runtime value system and conditional branching mechanism enable data-driven and state-dependent test scenarios, going beyond simple sequential verification.

Integration of open-source tools for session management and reporting. The selection and integration of OpenHTF as the session management framework provides structured and traceable test documentation, real-time logging of execution traces, and an operator interaction interface through a web-based GUI. The comparative analysis of available tools, including `pytest` and NI TestStand, demonstrated that OpenHTF meets all the requirements defined for this project while remaining open-source and hardware-independent. Its integration with Python

hardware libraries such as pyserial enables direct interaction with serial interfaces without modifications to the framework itself.
Validation of the embedded communication and I/O layer. The test bench was used to validate the correct implementation of CAN message transmission and reception according to the DBC database, digital signal generation and acquisition, analog signal reading and propagation, PWM generation, and the interaction between these interfaces in multi-step test scenarios. The experimental results confirmed the functional correctness of all implemented components and demonstrated the feasibility of the proposed approach as a basis for future integration stages with the complete rover platform.

7.3 Limitations of the Present Work

Although this work achieves its main objectives, this thesis has several limitations.
Absence of the real VCU hardware. The most significant limitation is that the test bench was validated using a second Blue Pill carrier board running a simplified loopback firmware rather than the actual Vehicle Control Unit. This simulation board reproduces only a functional subset of the real VCU behaviour, sufficient to exercise the test execution logic and the communication interfaces, but not representative of the full complexity of the real firmware. A complete validation process will require the developed test cases to be executed against the actual VCU hardware, verifying that the real firmware responds to the test bench signals as expected.
Limited signal emulation fidelity. Any test infrastructure operating without the final hardware can only approximate the real operating conditions. The current implementation does not model electrical non-idealities such as ADC noise, signal bounce on digital lines, bus contention on the CAN bus under heavy load, or transient glitches. Analog signal generation is limited to the potentiometer available on the carrier board, which provides manual control but does not allow automated generation of controlled voltage profiles. Similarly, PWM measurement capabilities on the test board side are currently limited to visual inspection via the LED indicator, without precise frequency and duty cycle capture.
Report granularity. The JSON report produced by OpenHTF correctly records the session metadata, the outcome of each phase, and the measurements explicitly collected during orchestration. However, the detailed execution trace generated by the test board is transmitted as serial text and recorded primarily as log output, rather than being parsed and structured into step and action level measurements within the OpenHTF report. This limits the readability and traceability of the final documentation when a detailed step-by-step analysis is required.
Structural limits on test complexity. The values of `TEST_MAX_STEPS = 5`

and `TEST_MAX_ACTIONS_PER_STEP = 4` were chosen to respect the constraints of the 20 KB SRAM of the STM32F103C8T6 and the 512-byte UART receive buffer. These limits are sufficient for the test cases developed in this work but may become restrictive for more complex scenarios in later integration stages. Relaxing them within memory constraints would require a more compact binary format for actions with few parameters.

Partial coverage of the VCU interface set. The current implementation does not cover all the interfaces present in the rover electronic architecture. Ethernet communication was excluded from the scope of the test bench because the relationship between the VCU and the Ethernet network had not been defined at the time of this work. Serial communication with the RF transceiver was only partially addressed, as the definitive communication protocol and hardware interface had not yet been selected. These aspects will need to be addressed as the project evolves and the hardware specifications are finalised.

7.4 Future Developments

7.4.1 Integration with Complete Rover System

The most important next step is the execution of the developed test cases against the actual VCU hardware, once the target microcontroller platform and the associated peripherals become available. This will require verifying that the physical connections between the test board and the VCU are compatible with the electrical specifications of the real platform, and modifying the CAN message database according to the one used in the final VCU firmware. Any discrepancies between the expected and actual behaviour observed during this integration phase will provide direct feedback for both firmware debugging and test case refinement. As the rover hardware progressively becomes available, the test bench should be extended to cover additional interfaces. The integration of the RF transceiver communication channel and the inclusion of the remaining analog and digital peripherals defined in the rover architecture will allow the tests to be expanded toward the complete Communication and Input/Output Layer. When the full hardware configuration is assembled, the test bench can also serve as a regression testing tool during firmware development, enabling automated re-execution of the validated test suite after each firmware update to detect regressions in the communication and I/O management functions.

7.4.2 Development of Full VCU Control Logic

The scope of the work presented in this thesis was intentionally limited to the Communication and Input/Output Layer of the VCU, leaving the Control Logic

Layer outside the validation scope. As the rover hardware becomes available and the physical parameters of the system are characterised, the development and validation of the full VCU control logic will become the next main activity. This will include the implementation and testing of motion control algorithms, including differential steering, traction supervision, and slope-dependent speed limitation; energy management strategies based on Battery Management System data; tool control functions for the nebulizer and future agricultural implements; and safety management logic covering fault detection, emergency stop handling, and safe-state activation. Validating these functionalities will require a more complete HIL simulation environment in which the dynamic behaviour of the motors, battery, and terrain is reproduced by a real-time simulation model, closing the control loop between the VCU and the simulated physical plant. The test bench developed in this thesis provides a natural starting point for this evolution, since its modular architecture and the CHAOS task structure can be extended to host more complex simulation logic as the project requirements grow.

7.4.3 Extension of Test Bench Capabilities

Several improvements to the test bench are needed in future work. The report structure could be improved by better organizing the UART execution trace captured by the OpenHTF logger. An improvement would consist in parsing these messages on the host side and presenting them in a more readable and structured format within the existing log section of the JSON report, for example by grouping messages by step, adding clear separators, and highlighting pass or fail outcomes for each step. This would make the execution trace significantly easier to inspect and analyse without requiring changes to the test execution model or to the internal firmware logic, and would improve the overall readability of the session documentation. The signal emulation capabilities could be extended by adding a dedicated digital-to-analog converter to the carrier board, enabling automated generation of controlled analog voltage profiles rather than relying exclusively on manual potentiometer adjustment. This would allow analog test cases to be executed without operator intervention, improving repeatability and enabling automated regression testing of analog acquisition functions. PWM measurement capabilities could be added through hardware timer input capture, allowing the test board to precisely measure the frequency and duty cycle of PWM signals generated by the VCU and compare them with the expected values within the test framework, rather than relying on visual LED inspection. Finally, the structural limits on test complexity could be revisited by adopting a more compact binary serialisation format for action parameters, reducing the per-action overhead in the HEX string and allowing a larger number of steps and actions to fit within the existing buffer and memory constraints. Alternatively, a fragmented transfer

protocol could be developed for the UART channel, enabling the transmission of larger test buffers across multiple serial transfers without requiring an increase in SRAM allocation.

Bibliography

[1] Franc Mihalič, Mitja Truntič, and Alenka Hren. «Hardware-in-the-loop simulations: A historical overview of engineering challenges». In: *Electronics* 11.15 (2022), p. 2462 (cit. on p. 1).

[2] P.C. Nissimagoudar, Venkatesh Mane, Gireesha H M, and Nalini C. Iyer. «Hardware-in-the-loop (HIL) Simulation Technique for an Automotive Electronics Course». In: *Procedia Computer Science* 172 (2020). 9th World Engineering Education Forum (WEEF 2019) Proceedings, pp. 1047–1052. ISSN: 1877-0509. DOI: 10.1016/j.procs.2020.05.153. URL: https://www.sciencedirect.com/science/article/pii/S1877050920314836 (cit. on p. 1).

[3] Yuehao Wang and Jian Xu. «Vehicle Real-time Condition Monitoring Based on CAN Bus». In: *Journal of Physics: Conference Series* 2405.1 (Dec. 2022), p. 012004. DOI: 10.1088/1742-6596/2405/1/012004. URL: https://doi.org/10.1088/1742-6596/2405/1/012004 (cit. on pp. 3, 8).

[4] Jernej Zabavnik, Andreas Riel, M. Marguč, and Miran Rodič. *Knowledge and skills requirements for the software design and testing of automotive applications*. 2019. arXiv: 1910.13128 [eess.SY]. URL: https://arxiv.org/abs/1910.13128 (cit. on pp. 3, 19).

[5] Mohammad Abboush, Christoph Knieke, and Andreas Rausch. «A Virtual Testing Framework for Real-Time Validation of Automotive Software Systems Based on Hardware in the Loop and Fault Injection». In: *Sensors* 24.12 (2024). DOI: 10.3390/s24123733. URL: https://www.mdpi.com/1424-8220/24/12/3733 (cit. on pp. 3, 19).

[6] Vector Informatik. *VN1600 Interface Family Manual*. 2026. URL: https://cdn.vector.com/cms/content/products/VN16xx/docs/VN1600_Interface_Family_Manual_EN.pdf (cit. on p. 30).

[7] Collin80. *SavvyCAN*. 2026. URL: https://github.com/collin80/SavvyCAN (cit. on p. 40).

[8] International Organization for Standardization. *ISO 26262-6:2018 Road Vehicles – Functional Safety – Part 6: Product Development at the Software Level*. 2018. URL: https://www.iso.org/fr/standard/68388.html (cit. on p. 23).

[9] Bo Li, Weina Wang, Long Jia, Dafang Wang, and Anran Kong. «Study on HIL system of electric vehicle controller based on NI». In: *IOP Conference Series: Materials Science and Engineering* 382.5 (July 2018), p. 052033. DOI: 10.1088/1757-899X/382/5/052033. URL: https://doi.org/10.1088/1757-899X/382/5/052033 (cit. on p. 23).

[10] Mengyuan Wang, Cong Zhang, and Jianjun Yang. «Research on hardware-in-the-loop test technology for hybrid vehicle VCUs». In: *International Conference on Computer Application and Information Security (ICCAIS 2024)*. Vol. 13562. SPIE, 2025, 135621N. DOI: 10.1117/12.3060898. URL: https://doi.org/10.1117/12.3060898 (cit. on p. 23).

[11] OpenHTF Project. *OpenHTF Documentation*. 2026. URL: https://www.openhtf.com/ (cit. on pp. 25, 48).

[12] Patrik Wennberg and Viktor Danielson. «Evaluation of a Testing Process to Plan and Implement an Improved Test System: A Case Study, Evaluation and Implementation in LabVIEW/TestStand». Master's Thesis. Chalmers University of Technology, 2016. URL: https://www.diva-portal.org/smash/get/diva2:954626/FULLTEXT01.pdf (cit. on p. 26).

[13] National Instruments. *TestStand*. 2026. URL: https://www.ni.com/en/shop/electronic-test-instrumentation/application-software-for-electronic-test-and-instrumentation-category/what-is-teststand (cit. on p. 26).

[14] Anonymous. In: *International Journal of Engineering & Extended Technologies Research (IJEETR)* 8.2 (Mar. 2026), pp. 2246–2250. DOI: 10.15662/IJEETR.2026.0802202. URL: https://ijeetr.com/index.php/ijeetr/article/view/612 (cit. on p. 32).

[15] Pengpeng Nie, Youyu Wu, and Xiaoyu Liang. «Design of HIL Test System for VCU of Pure Electric Vehicle». In: *Proceedings of the 2017 2nd International Conference on Materials Science, Machinery and Energy Engineering (MSMEE 2017)*. Atlantis Press, May 2017, pp. 867–871. ISBN: 978-94-6252-346-3. DOI: 10.2991/msmee-17.2017.168. URL: https://doi.org/10.2991/msmee-17.2017.168 (cit. on p. 23).

[16] Xiyang Wang, Hongpeng Li, Wenxia Xi, Dianming Zhang, and Wei Guo. «Design and verification of VCU system based on ISO26262». In: *Journal of Physics: Conference Series* 2491.1 (Apr. 2023), p. 012036. DOI: 10.1088/1742-6596/2491/1/012036. URL: https://doi.org/10.1088/1742-6596/2491/1/012036 (cit. on p. 23).

[17] pytest Development Team. *pytest Documentation*. 2026. URL: https://docs.pytest.org/en/stable/ (cit. on p. 24).

[18] ISTQB. *ISTQB Certified Tester Foundation Level Syllabus v4.0.1*. 2024. URL: https://istqb.org/wp-content/uploads/2024/11/ISTQB_CTFL_Syllabus_v4.0.1.pdf (cit. on p. 22).

[19] Jacopo Sini, Massimo Violante, and Fabrizio Tronci. «A Novel ISO 26262-Compliant Test Bench to Assess the Diagnostic Coverage of Software Hardening Techniques against Digital Components Random Hardware Failures». In: *Electronics* 11.6 (2022). DOI: 10.3390/electronics11060901. URL: https://www.mdpi.com/2079-9292/11/6/901.