Politecnico di Torino

Politecnico di Torino

INGEGNERIA INFORMATICA (COMPUTER ENGINEERING)
Academic Year 2024/2025
Graduation Session 10/2025

Development of a testing framework enabling Continuous Integration with Hardware-In-the-Loop for Arduino cores

Supervisors:
prof. Alessandro Savino
prof. Stefano Di Carlo
eng. Arturo Guadalupi

Candidate:
Ruggero Rossino

Summary

Continuous practices such as Continuous Integration (CI) and Continuous Deployment (CD) allow developers to release code faster and, most importantly, more reliably. CI, in particular, automates the integration of new code into the existing one and is often paired with a testing phase executed before actually merging the modifications. This enables earlier bug discovery, significantly improving software quality, while reducing the costs of detecting and fixing issues.

Although these practices are widely appreciated and are becoming the standard in software development, they struggle to be adopted when hardware testing is necessary, for instance in the domain of embedded systems. In fact, effective tests should be reliable, deterministic, and isolated, but these characteristics are difficult to achieve when interacting with real hardware, as failures may be caused not only from the code under test, but also from external factors such as hardware defects, inconsistent states resulting from previous operations, or transient external conditions. Other major pitfalls include the cost of building an adequate infrastructure and setting up the testing environment appropriately, which often require significant resources in terms of time, money, and effort. Nevertheless, code validation cannot be disregarded.

In this context, Arduino seeks to enhance CI/CD for its microcontroller cores (i.e., the abstraction layer that enables exposing the same API independently of the underlying OS). This choice is primarily driven by the need to abandon the Mbed OS platform and migrate to the Zephyr Real Time Operating System (RTOS), which requires changing a substantial portion of the codebase. This is not only limited to this context, as it will eventually be applied to the other cores as well.

A basic automated pipeline is already in use within the organization, and it consists of a GitHub Action that gets triggered whenever a new push or pull request takes place on the core's GitHub repository. Still, the current workflow only verifies successful compilation, but does not assess whether regressions are introduced. This thesis aims to fill this gap by developing a testing framework that validates communication protocols through Hardware-in-the-Loop testing and integrating it in Arduino's CI process.

The designed framework comprises a set of Arduino programs, called sketches,

which actually conduct the tests, and an orchestrator that coordinates the whole process. The tests adopt a loopback approach to validate both the receiver and transmitter of a protocol's endpoint, and involve one board acting as Validator, and another as Device Under Test (DUT). Both boards were carefully selected to be golden samples, which means devices that are free from hardware-related issues.

A unit test follows these steps:

1. the Validator generates a sequence of bytes of a given length and calculates its 8-bit CRC value;

2. it sends the sequence to the DUT;

3. the DUT sends the data back without modifying them;

4. the Validator computes the 8-bit CRC value of the incoming message and compares it with the original one;

5. if the values match, the test is successful; otherwise it fails.

This design enables verification of data integrity during both the transmission and reception of messages. Thanks to its simplicity, it can be applied to every communication protocol. The test is carried out with an increasing number of bytes, from 1 to the limit imposed by the protocol (for example, the library for I2C communications allows transferring at most 32 bytes messages), making sure that there are no issues caused by specific message lengths. Furthermore, by changing and combining each protocol's parameters, such as baud rates, clock speed, and repeating the test for each port available on the board, it is possible to cover a wide range of scenarios.

To organize, coordinate, and execute the pipeline, an orchestrator program runs on a host machine. Its duties are the setup of the execution environment, the construction of the test suites, and the collection of the results. It proceeds as follows:

1. it detects the boards connected to the host machine and collects the necessary information to uniquely identify them, such as the serial number and the Fully Qualified Board Name (FQBN);

2. it loads a configuration file, that specifies the roles played by the boards, the core and its version that each device should run, the parameters with which the tests must be executed;

3. it matches the detected board with the desired configuration, assigning them the respective roles and installing the required cores;

4. for each test suites specified by the arguments passed when launching the program, it assembles the test case to cover every possible parameter combination based on the configuration file;

5. it monitors and logs the results of each unit test, collecting and aggregating them in a separate file for easier inspection

The approach employed during the development of the orchestrator aims at maximizing the modularity and maintainability of the system, making it easy for developers to modify existing test suites or add new ones without restriction to the loopback approach, nor to communication protocol tests only. Instead, it is enough for a test suite to implement the TestSuite interface and add it to a registry to be automatically detected by the framework and included in the pipeline. Moreover, the configuration file, written in JSON, provides a single place to specify the entire execution environment, including the cores and version each board should run and the parameters with which the test suites must be assembled, enhancing the usability and versatility of the whole framework.

Continuous Integration was enabled through a new GitHub Action, which executes the tests whenever a push or a pull request occurs on the core's repository. It involves a workflow that is carried out on a self-hosted runner on a Raspberry Pi. It is made up of one job that ensures isolation by clearing the environment from any files, directories, or artifacts potentially left by previous runs, clones the repository where the framework is situated, installs the appropriate toolchain and dependencies, and finally executes the tests, monitoring the program to detect errors or failures. Once the file containing the results is available, it is uploaded as an artifact so that it can be accessed from the repository's Web page.

Given the modularity of the framework's implementation, an attempt was made to integrate previously developed PWM tests. These relied on a host machine commanding an Arduino board to generate PWM signals and on laboratory instruments, such as oscilloscopes and multimeters, controlled via SCPI commands to measure them. However, the libraries used to control these instruments were poorly structured, comprising over 500 functions that differ only in their SCPI strings, hindering both usability and maintainability.

Several redesign strategies were explored. The first aimed to abstract the common logic while preserving the functions as wrappers of a command. This approach would have ensured high compatibility, as the changes would have been internal to the libraries, but the number of wrapper functions would have remained the same. Hence, it was discarded in favor of a dictionary-based approach, in which a TOML file defines common commands together with their parameters, while the library offers a single method to dynamically construct the SCPI string. To assess its feasibility, a use case was implemented in which a Go program interacted with an oscilloscope to capture a waveform screenshot upon a trigger event. The

implementation showed that this strategy worked well for simple commands, but more complex ones required custom functions, limiting scalability. Moreover, a complete redesign of the previously written tests would have been necessary, since the library interface was entirely different. Finally, another solution was proposed, leveraging the hierarchical structure of the SCPI standard to provide few functions that could be chained to build SCPI strings, keeping it compact and avoiding programmers from having to remember the exact syntax. However, due to time constraints, the focus shifted back to the development of the main pipeline, and this strategy was not tested.

To evaluate the system, the UART, I2C, and CAN protocols were tested under multiple configurations, for a total of 17 distinct test cases executed not only on the Zephyr-based core, since it is still in beta and lacks some functionalities, but also on the Mbed-based one, which served as an additional reference. The framework consistently verified data integrity and reliably detected both induced and genuine faults, demonstrating determinism and repeatability. All outcomes were further confirmed through a logic analyzer, ensuring the fidelity of the framework's behavior. Integration with GitHub Actions was also effective, as the pipeline executed the entire process without requiring any human intervention, ensuring fully automated and isolated runs, which are fundamental traits of continuous processes. However, execution overhead emerged as a limitation, since two-thirds of the total pipeline time is spent compiling and uploading the test sketches. This issue should be addressed via parallelization and other techniques for scaling out.

Overall, the project achieved its intended objectives, delivering a robust and modular proof-of-concept that integrates HIL testing into Arduino's continuous pipeline. Although focused on communication protocols, the adopted architecture can be expanded and future improvements can further enhance it by addressing some limitations such as fault tolerance, scalability, and execution time to allow its systematic adoption within the organization.

Table of Contents

List of Tables

List of Figures

Acronyms

API

Application Programming Interface

ARM

Advanced RISC Machines

ASCII

American Standard Code for Information Interchange

ATU

Arduino Test Utility

CAN

Controller Area Network

CD

Continuous Deployment

CDE

Continuous Delivery

CI

Continuous Integration

CLI

Command Line Interface

CRC

Cyclyc Redundancy Check

DFU

Device Firmware Upgrade

DUT

Device Under Test

EDF

Earliest Deadline First

EDK

Extension Development Kit

ELF

Executable and Linkable Format

FPGA

Field-Programmable Gate Array

FQBN

Fully Qualified Board Name

GPIO

General Purpose Input/Output

GPOS

General Purpose Operating System

HAL

Hardware Abstraction Layer

HIL

Hardware-in-the-Loop

IDE

Integrated Development Environment

IoT

Internet of Things

IP

Internet Protocol

IRQ

Interrupt ReQuest

I2C

Inter-Integrated Circuit

JSON

JavaScript Object Notation

LLEXT

Linkable Loadable Extensions

sps

samples per second

NIST

National Institute of Standards and Technology

OS

Operating System

PCB

Printed Circuit Board

PID

Product Identifier

POSIX

Portable Operating System Interface

PWM

Pulse Width Modulation

RAM

Random Access Memory

RISC

Reduced Instruction Set Computer

RTOS

Real Time Operating System

SCPI

Standard Commands for Programmable Instruments

SoC

System-on-Chip

SSH

Secure Shell

TCP

Transmission Control Protocol

TOML

Tom's Obvious, Minimal Language

UART

Universal Asynchronous Receiver-Transmitter

UI

User Interface

USB

Universal Serial Bus

VCS

Version Control System

YAML

YAML Ain't Markup Language

Chapter 1

Introduction

Requirements for modern software applications are becoming increasingly demanding. Whether they are intended for a large audience, such as social media platforms, gaming applications, and productivity tools, or highly specialized, such as firmware for embedded systems ranging from IoT devices to avionic control systems, today's software is expected to deliver rich functionality, high performance, and efficiency, while continuously evolving over time to offer new features and adapt to new contexts.

As a consequence, it is no longer possible for organizations to utilize small teams to develop and release these applications and then move on with the next project. Instead, they must rely on multiple teams, often spread across the globe, that collaborate on a single project, trying to balance all these requirements while ensuring long-term maintainability. This significantly complicates the development process, and as a consequence new methods and tools have emerged. Among the most effective ones are *continuous practices*. When implemented correctly, these techniques provide a process which helps developers by facilitating collaboration, enhances the quality of their work through accurate and automated testing, significantly reducing the cost of fixing defects, and accelerates the release of applications and updates. Their proven effectiveness resulted in widespread adoption across the industry.

Continuous practices typically benefit from being executed in purely virtual environments, which abstract away hardware dependencies and provide controlled conditions. This is particularly valuable during the testing phase, where repeatability and reliability are essential to obtain meaningful results. However, this concept does not hold when it comes to embedded systems, where software is tightly coupled to the physical hardware it runs on. Therefore, new challenges are introduced: tests are harder to isolate, external factors can affect their outcomes, and the infrastructure must be adapted to integrate the hardware into the process. For this reason, when hardware testing is involved, the adoption of continuous practices

becomes more complex and new strategies must be found to balance reliability with cost.

Arduino is currently facing these challenges while in the midst of a significant architectural transition for some of their products. With Arm announcing the retirement of their Mbed Operating System in 2026, the embedded system industry has been forced to find alternatives. Arduino met this announcement by choosing to migrate its firmware to the Zephyr RTOS. However, Zephyr is a fundamentally different environment, which makes the migration process delicate and complex. Therefore, Arduino set out to refine and strengthen its adoption of continuous practices. Although the organization already employs a basic continuous process, it remains limited and lacks a proper testing phase, capable of validating new code effectively.

This project aims at developing a proof-of-concept of a testing framework with Hardware-in-the-Loop that can be fully integrated into the continuous process of Arduino. Although the scope of the project revolves around validating functionalities of communication protocols, the broader goal is to provide a modular and centralized mechanism that can be extended with additional types of test and adapted to other contexts as well, serving as the backbone of an enhanced continuous process.

This thesis seeks to illustrate the developed framework, while providing the reader the necessary background to understand its implementation strategy, evaluate design choices, and assess its results. The structure is as follows:

- **Chapter 2** introduces continuous practices, presenting their characteristics, benefits, and challenges. It also analyzes their application in embedded systems context and, specifically, how Arduino currently implement them;

- **Chapter 3** presents the concept of Real Time Operating Systems and examine the Zephyr RTOS main characteristics. It concludes with an overview of Arduino's Zehpyr-based firmware;

- **Chapter 4** analyzes the implementation of the testing framework, detailing the adopted strategy, the tools utilized, and the system's functioning. Finally, it presents the challenges faced during development.

- **Chapter 5** discusses the attempt to integrate previous work on automated testing into the new architecture. It describes the encountered issues, the strategies explored to overcome them, and a practical evaluation of one proposed solution;

- **Chapter 6** reports the results of the testing framework and provides a qualitative assessment of whether the initial goals were achieved;

- **Chapter 7** summarizes the achievements of this project, together with its limitations and outlines possible future work.

Chapter 2

Background and state of the art

This chapter offers an overview of what continuous practices are, why they are important in the software development process, and how they are employed when hardware must be dealt with. After this introductory explanation, the focus is shifted on how Arduino currently implements them, what are its long-term goals, and what has already been done to achieve them.

2.1 Continuous practices

Continuous practices is a general term that encompasses different techniques, specifically, Continuous Integration (CI), Continuous Deployment (CD) and Continuous Delivery (CDE). These practices are fundamental when many developers must work together to release any piece of code and are becoming widespread in organizations that produce any kind of software. If implemented correctly, they offer many benefits, such as automation of recurring processes, improved code quality, and fast delivery times [1] [2] [3]. *The State of Developer Ecosystem 2023* [4], a survey conducted by Jet Brains, shows that up to 45% of developers employed by a company or organization regularly use such tools.

Although this thesis revolves around Continuous Integration, each continuous practice will be briefly described, as they are all important parts of the process that bridges a software product from its development to its release (Figure 2.1).

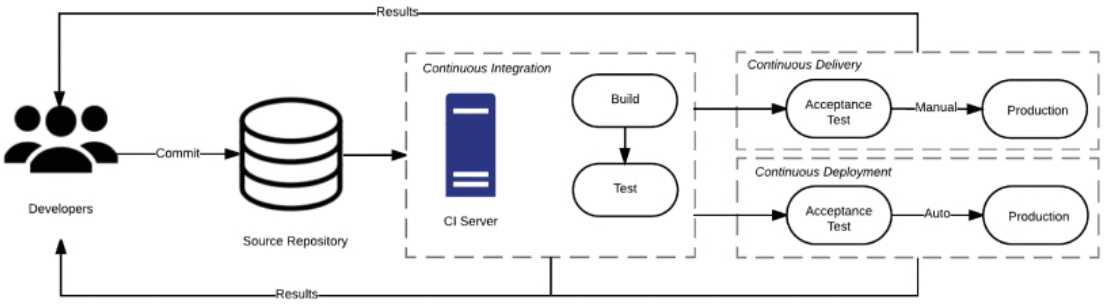

Figure 2.1: CI, CDE, and CD relationship.

2.1.1 Continuous Integration

Continuous Integration is a practice that involves developers frequently merging their work into a shared repository. Its primary goals are to enable easy collaboration among multiple developers and improve software quality and efficiency by following a process, referred to as the CI pipeline, which is triggered whenever developers commit their code [2]. This pipeline usually comprises the following steps [1][3]:

- **Code integration**: this is the preliminary step to the actual pipeline. Changes are merged into the repository on a daily basis to avoid large and complex integrations that often cause major conflicts, hard to understand and resolve. By integrating updates frequently, fewer conflicts are likely to arise and those who do are simpler to deal with;

- **Automated build**: the committed code is automatically converted into executable files. This saves developers significant time and effort, boosting productivity;

- **Automated testing**: after a successful build, a suite of automated tests is run to validate the work. This step is crucial to ensure that no regressions have been introduced and that full functionality of the application is preserved.

Automation is the key enabler of Continuous Integration. Developers are notified immediately if issues occur and can take corrective actions if necessary. This allows for earlier defects discovery, which helps greatly reducing the cost of fixing bugs, both in terms of effort and money (Figure 2.2).

Implementing the CI process requires dedicated tools: a Version Control System, which is the central repository hosting the code base, a CI server that monitors the repository for changes and triggers the pipeline, and an automated test suite capable of reliably validating each update [5].

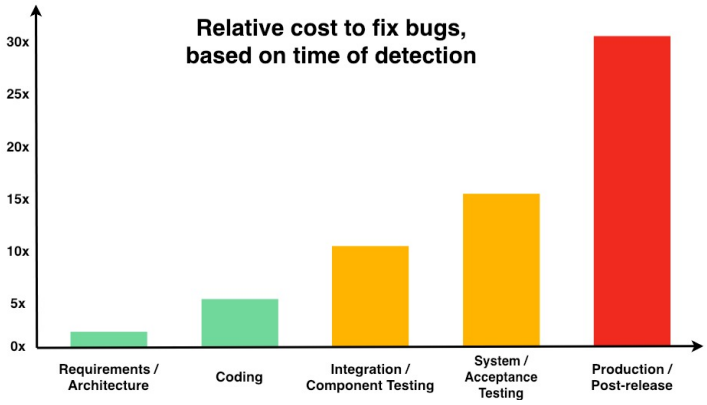

Figure 2.2: Costs of fixing defects based on time of detection. Source: NIST.

The next subsections will describe its most important characteristics of the Version Control System and of the test suite, while the CI server will be described after having introduced also CD and CDE.

The Version Control System

Version Control Systems (VCSs) manage the central repository where developers merge their code. A VCS provides a framework that allows multiple developers to work on the same project at the same time, even from different locations. It prevents them to overwrite each other's work by raising conflicts whenever modifications are incompatible, and offers branching mechanisms where new lines of development can be created to develop and tests features without affecting the reset of the project. VCSs also keep track of the entire history of the project, including the author of every update. This enables reverting the codebase to previous versions, providing the means to undo mistakes and roll back to a stable state, without having to keep manual copies of the work [6] [7].

There are two main categories of VCSs: centralized and decentralized (also known as distributed). The first (Figure 2.3) is based on a client-server model, where a central server stores the master copy of the project files along with their version history. Every action a developer wants to take on the repository, such as commit changes, access history, or check out the code, requires a connection to the central server. This approach is generally simple to understand and manage, but the central server constitutes a single point of failure [6]. Subversion is an example of VCS adopting such an approach. Instead, in decentralized Version Control Systems (Figure 2.4) each developer owns a full local copy of the entire repository, including its history. Even if a central repository for coordination purposes usually exists, developers can perform any action without the need for a

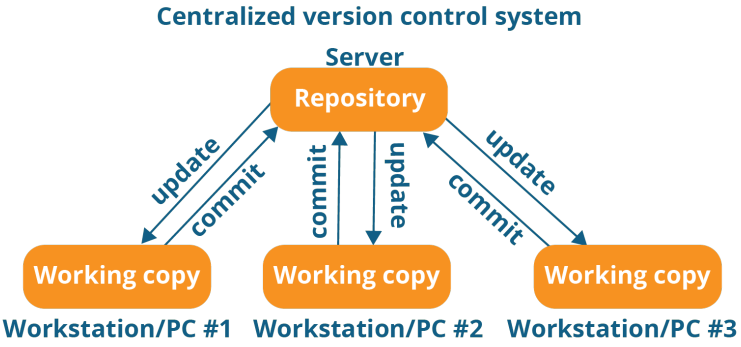

Figure 2.3: Centralized Version Control System. Source: edureka

persistent connection. It is only required of them to connect to that repository to "push" their local changes or "pull" others'. While more complex than the previous one, this strategy enhances performance, reliability, and flexibility. Git, the most widespread VCS, is based on this very approach [6] [8].

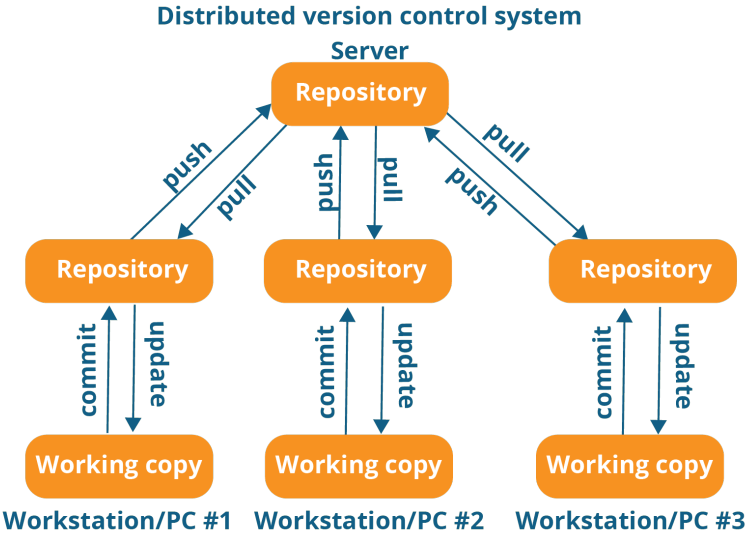

Figure 2.4: Decentralized Version Control System. Source: edureka

GitHub and GitLab are the two most popular web-based platforms that allow creating and managing Git-based repositories. They both offer all the necessary features of a Version Control System, such as the possibility of creating public and private repositories, branching mechanisms, code review tools, and issue tracking. However, over time they expanded to become fully-fledged CI/CD platforms [9].

The test suite

An effective test suite is not merely a collection of tests, but it must be carefully designed taking in to consideration some principles, such as automation, speed, reliability, and coverage.

As previously mentioned, **automation** is fundamental to support the goals of CI. The included tests should be easy to automate and should not require human intervention. This reflects also on another key principle: **speed**. A slow test suite hinders CI adoption by preventing rapid feedback [10]. Hence, execution times should be minimized. Various techniques can be leveraged to reduce execution times, such as parallelization, but ultimately it comes down to the suite's architecture. Some tests (such as UI tests) are inherently slower than others, therefore they should be run last, after having carried out all the quicker ones. The ideal architecture should support the "fail fast" principle, which states that the suite should be composed of stages of increasing complexity and execution time, allowing it to fail as early as possible [11][5]. These concepts are well demonstrated by the *testing pyramid* (Figure 2.5), introduced by Mike Cohn in "Succeeding with Agile" [12]. It is an abstraction that illustrates the relationship between the costs, speed, and coverage of different kinds of tests. According to it, the suite should heavily rely on unit tests. This is a kind of test that verifies single components of a system in isolation, providing fast and inexpensive validation. They should aim to maximize code coverage, a fundamental metric that indicates how much of the code is being evaluated. On top of unit tests, other types can be implemented, such as service, component, and integration tests, which, instead, check how those components interact with each other. At the top of the pyramid UI testing can be found. It is the most expensive and slow type of all, requiring complex setups and tools (but is also the least relevant in the context of this thesis) [13].

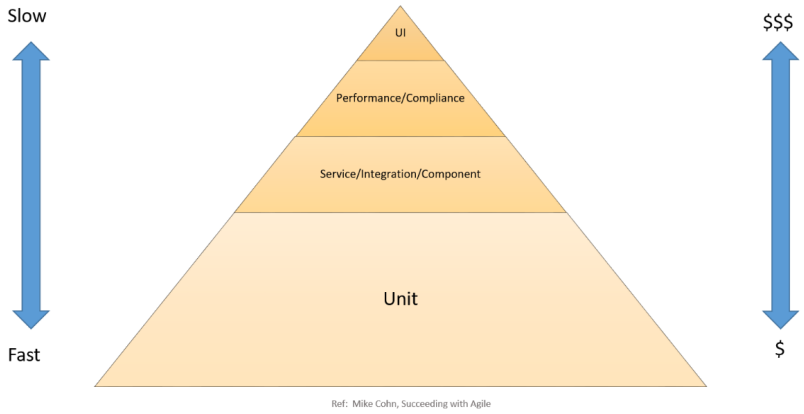

Figure 2.5: The *testing pyramid*.

Two critical factors that are tightly related are **reliability** and **repeatability** of the test suite. Tests must produce clear and consistent results, without any ambiguity or unpredictability. To achieve this, they must be carried out in a controlled environment, ensuring that failures are determined by a code change and not by external factors. If developers cannot trust and rely on the results, the adoption of the whole CI process is fundamentally undermined [11].

Finally, as software evolves, the suite must evolve with it. This implies that **maintainability** should not be overlooked. On the contrary, it should be easy to modify and expand the test suite to adapt it to new contexts. Otherwise, a suite that may work fine at a given time could become completely useless as the project expands [2].

2.1.2 Continuous Delivery and Continuous Deployment

After validating the code quality via CI, other steps are required for an effective deployment. In this context, the terms Continuous Delivery and Continuous Deployment are often used interchangeably, as they both build upon the CI process, toward the release of the code. However, they differ in how the release step is managed. CDE ensures that an application is always in a production-ready state, but it holds it in a pre-production environment, waiting for a human to actually deploy live. Instead, CD also automates that last step, ensuring that the code that passes the CI pipeline is promptly released to the public [10].

The ability of these two practices to carry out quality checks on the code is limited, as their focus is to make sure that it can be released. This once again highlights the importance of CI, as it is the only barrier that prevents faulty code from reaching the customer.

When employed in conjunction with a well-structured CI process, CDE and CD eliminate the need for "release days", which put high pressure on developers. Instead, multiple smaller releases are easier to manage and enable faster feedback from clients, providing them with continuous value, which in turn enhances satisfaction [10] [14].

The CI/CD server

The CI/CD server orchestrates the whole continuous pipeline. It monitors the VCS to detect new commits, runs build and test scripts, and provides feedback to developers. Many tools can be found on the market. For example, Jenkins is a self-hosted solution that offers great customization possibilities, but it can be difficult to configure and maintain. As a consequence, the trend of migrating projects from self-hosted servers to online platforms has been established over the last few years. Among the most famous platforms are GitLab CI/CD and GitHub

Actions, which are highly integrated with the respective VCSs (i.e., GitLab and GitHub), offering easy to set up triggers that execute the pipeline based on events happening on the repository, such as new commits. They also allow defining the entire pipeline via YAML files instead of relying on a UI-based approach, making its configuration documented and tracked like any other file in the VCS. In addition, developers can choose to run the pipeline on self-hosted machines or on cloud runners, that is, virtual machines managed by the platform themselves[15] [16].

2.2 Continuous practices with Hardware-in-the-Loop

Despite its challenges, the adoption of continuous practices is generally straightforward when it concerns pure software applications and can benefit from virtualization and containerization services that provide a controlled and isolated execution environment [5]. However, this is not the case when software is tightly coupled with physical hardware, such as in embedded systems, where new kinds of issues pose an obstacle to their adoption.

There are mainly two approaches to solve these issues: one consists in building a software simulation of the hardware, known as *digital twin*, while the other integrates it directly into the infrastructure. The first method requires a set of digital models that accurately represent the physical system [17]. While a digital twin provides an isolated environment with repeatable conditions that would otherwise be very difficult to reproduce, the effort required to build such abstraction, both in terms of complexity and money, is considerable and requires rigorous processes that ensure its fidelity to the real world [18]. This explains why it does not represent a solution applicable to all scenarios, and is prevalently used when safety critical systems have to be designed or high budget is available. Instead, if real hardware is involved, a way must be found to verify that it behaves as intended. This requires modifying the architecture of the CI pipeline. The CI server cannot leverage a purely virtual environment anymore, but a **local executor machine** must be added (Figure 2.6). Such machine must be able to receive requests for test execution from the CI server, interact with the system under test, and report results back to the server. The link between the executor machine and the hardware is of great importance. Its nature can range from a simple UART connection to a complex setup containing hardware debuggers, power supplies, and other equipment. However, it must at least provide a way to deploy the software on the device, restart it, and communicate with it to run the tests and collect their outcomes [19]. These operations usually require considerably more time than those carried out in a virtual environment, greatly affecting the speed of the CI process. Therefore, even more attention should be paid to the design of the pipeline structure, enhancing the “fail fast” principle to

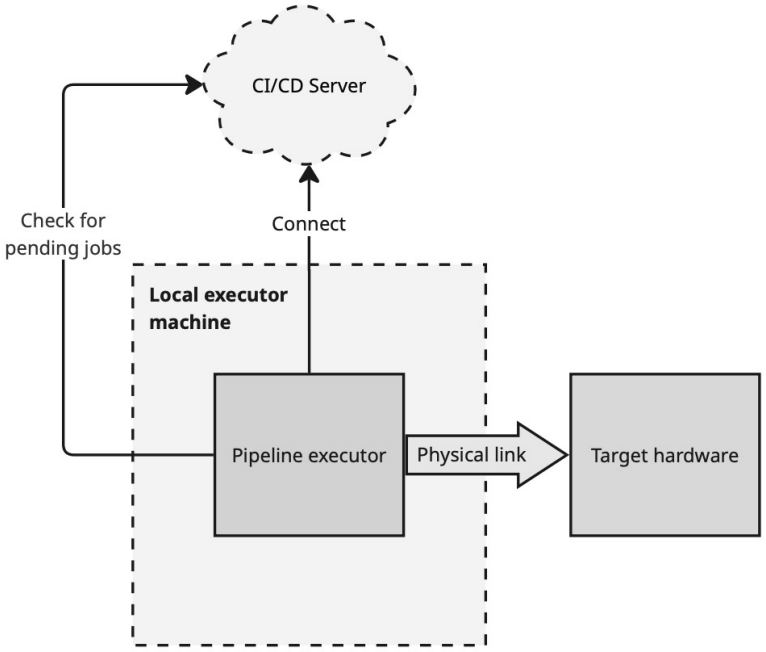

Figure 2.6: Architecture of CI process with Hardware-in-the-Loop

avoid wasting resources.

Configuration and dependency management is another key aspect that must be accounted for. Each hardware platform may have its own requirements and this makes build and test setup failures prevalent [11]. Such challenges significantly impact repeatability of the test suite, which has to be carefully designed to deal with these inconsistencies. Moreover, it must be ensured that the physical devices are free of hardware-related issues, which requires dedicated tests.

Lastly, if the developer team is geographically dispersed, security becomes a concern too. In a purely software-based context, the CI pipeline can be entirely delegated to a third party CI/CD platform, which will take care to implement all the necessary strategies to make it secure. However, when physical access to the hardware is required, this is prevented by the need for the local executor machine, which must be accessible from the CI server through the internet. This requires organizations to implement their own security strategies, which, if not designed correctly, may result in disruptive attacks [5] [19].

Despite these obstacles, HIL with physical hardware offers a cheaper and quicker way to test the code in realistic scenarios with respect to simulation, avoiding

entirely model inaccuracies. It also allows testing of the complete physical interface in a production-like environment, increasing confidence in the CI process.

2.3 Continuous practices in Arduino

This section explores to what extent continuous practices are currently employed in Arduino, their purpose, their drawbacks, and what corrective actions are being taken to address them.

The standard procedure implemented in Arduino consists of a CI/CD pipeline triggered when developers try to merge their code in a repository. However, the Continuous Integration strategies adopted in this pipeline are basic and present major drawbacks.

As an example, the pipelines used on the Zephyr-based and MbedOS-based firmware repositories will be analyzed. Beyond quality checks that enforce some rules about code formatting and commit messages' structure, both pipelines try to build sample programs to ensure that compilation is still working. These checks are run on every pull request or push. Instead, when a new version of the core is ready to be released, an additional workflow packages the code and publishes it for distribution.

Although they successfully implement the first step of a CI pipeline (i.e., the automated build), they clearly lack automated tests, which are instead done manually by developers. This, over time, resulted in the publication of faulty code that, even though built successfully, still introduced regressions in various components, such as communication protocols, causing failures. As a consequence, Arduino has decided to start developing a new pipeline that would also validate the functionality of the updates. This requires introducing real hardware tests that must be executed on the physical boards and carefully validate that all basic functions are free of issues.

The first phase of this rework produced functional tests capable of checking PWM (Pulse Width Modulation) signals. These tests consist of a host machine controlling an Arduino board that generates PWM waveforms. These are monitored and measured via oscilloscopes and multimeters to verify that they follow the given specifications. However, this represents only the first step toward the construction of an automated pipeline, aiming to reliably verify the correctness of all functionalities of a firmware, and still lacks integration with the continuous pipeline.

Chapter 3

Context and motivation

Since Arduino is transitioning from Mbed OS to Zephyr, it is important to understand its main characteristics. Therefore, this chapter describes what are Real Time Operating Systems and the main differences with General Purpose Operating Systems. Zephyr RTOS will then be analyzed, showing its features and how they are leveraged by Arduino to implement its new core.

3.1 Real Time Operating Systems

Real Time Operating Systems (RTOSs) differ from General Purpose Operating Systems (GPOSs) in their primary goal. GPOSs, such as Microsoft's Windows, Apple's macOS, or Linux are designed for versatility and user experience. They support a wide range of applications and needs and offer a rich User Interface. Their design focuses on ensuring a certain amount of **fairness** among processes execution, meaning that the scheduling algorithm they implement tries to balance resources between many tasks in a more or less fair way. However, this results in unpredictable latencies between an event's occurrence and the system's response. Moreover, to enhance such versatility, GPOSs offer features such as virtual address mapping and separate kernel and user address spaces, which lead to a large memory footprint [20]. These properties make them ideal for running on devices with high computational power and memory capacity, such as personal computers, mobile devices, or servers, but hinder their application in contexts where resources are limited, such as embedded systems. Instead, RTOSs are designed to manage hardware resources in resource-constrained and time-critical environments, where the correctness of the performed operations depends not only on the logical output but also on its timing. To satisfy this requirement, RTOSs are designed for **determinism**, which is the ability to calculate the response time for a given event

beforehand. It is important to note that determinism does not necessarily mean improved performances but rather enhanced predictability. Thanks to this property, specialized scheduling algorithms can be employed to prioritize and execute tasks based on their timing requirements. Moreover, to achieve low latency, the kernel is designed to avoid masking interrupt requests for long periods of time and to keep its critical regions as short as possible. When used in conjunction with preemptive multitasking, a widespread technique that allows a higher-priority task to interrupt a lower-priority one, this design ensures that critical operations are executed as soon as possible. An intuitive scheduling algorithm that uses preemptive multitasking is called Earliest Deadline First (Earliest Deadline First), which assigns the highest priority to the task with the closest deadline [20] [21]. Another fundamental feature is the small memory footprint of RTOSs, originating from the union of the kernel and user address spaces and, above all, the modularity and configurable nature of the kernel itself, which allows developers to include only the features they actually need [21].

RTOSs are further categorized based on the strictness of the timing constraints, or, in other words, on how they behave when deadlines are missed:

- **Hard real-time** systems do not tolerate missing deadlines, which would cause a total system failure. For this reason they are used in safety critical applications;
- **Firm real-time** systems tolerate to an extent missing a few deadlines. However, if the threshold is reached, a system failure occurs;
- **Soft real-time** systems continue operating even when missing multiple deadlines, but with degraded performance.

These few characteristics demonstrate why RTOSs are a valid choice when predictability, efficiency, and low latency are essential.

3.2 Zephyr RTOS

Zephyr is a Real Time Operating System specifically designed for low-power, resource-constrained devices. It is an open-source project hosted by the Linux Foundation, strongly relying on its community to evolve and develop new capabilities and features. The collaborative nature of the Zephyr project allows it to adapt to the vast and dynamic space of embedded systems with which organizations alone struggle to keep up. For this purpose, the Zephyr project is structured with a main repository, containing the source code, configuration files, and build system, and a collection of externally maintained repositories, called modules, that

provide third-party integrations. Zephyr leverages the West meta-tool to manage these modules. Through West, users can create a workspace for their application, specifying the modules they need, and keep them updated to their correct versions [22].

3.2.1 The Kernel

The Zephyr kernel has been designed to achieve high flexibility, providing many essential services. It offers multithreading capabilities, with both preemptive and non-preemptive threads, adhering to the POSIX "pthreads" standard. For synchronization and data passing, it provides mechanisms such as mutexes, semaphores, and message queues. Furthermore, it supports both fixed-size and dynamic-size memory blocks for memory allocation, advanced power management at system and driver level, and multiple scheduling algorithms, including EDF. Moreover, its modular architecture allows developers to select only those features that are necessary for their application, greatly reducing its memory footprint, which, in turn, enables devices with as little as 8 kB of memory to run it [23]. These features make it compatible with a wide range of SoCs and different architectures, such as ARM, x86, RISC-V and others.

The Hardware Abstraction Layer

In order to run on such diverse systems, Zephyr employs a Hardware Abstraction Layer (HAL), which provides a consistent interface for managing drivers and making them reusable across different platforms. There are two types of HALs: vendor-provided, developed and maintained by the vendors for their specific hardware, and external ones, which are community-driven and hosted in dedicated modules. This abstraction is achieved through the *Device Driver Model* and the *Devicetree*. The first provides a neutral and device-independent API, which hides the driver's implementation details, allowing applications to interact with the same API, independently of the underlying hardware. Instead, the *Devicetree* is a hierarchical data structure used to describe the hardware on which Zephyr is running and its boot-time configuration. It is implemented as a tree where each node represents a hardware component, such as an I2C controller, or a sensor, and stores important information about it, such as its identifier, its status, and, for addressable resources, the address space.

Together, the *Devicetree* and the *Device Driver Model* indicate what hardware Zephyr is running on, its configuration, and how it can be used [23].

3.2.2 The Build and Configuration system

Zephyr's build process (Figure 3.1) is defined as "application-centric" [23], meaning that the application initiates and controls the build of both the application itself and the kernel, compiling them into a single binary.

Before any build occurs, West is invoked to retrieve all the modules the project is composed of. Only then the build process can start. The process is divided into two phases: configuration and build. The configuration stage is driven by CMake, a free and cross-platform build system. Each Zephyr project includes a `CMakeLists.txt` file, which serves as script for integrating the application's source files with the Zephyr build system. It specifies the essential build configurations, such as the target hardware board and any application-specific software features. Based on this file and together with the *Devicetree* and the *Kconfig* configuration system, CMake produces a set of output files that will be used in the next phase. While the *Devicetree* allows the application and driver code to access hardware details, *Kconfig* is the means through which developers can configure the software features of the kernel, enabling or disabling them based on their needs. The build phase can be executed with the Make or Ninja build systems and is further divided into four stages [23] [22]:

1. **Pre-Build stage**: before actually compiling, the necessary headers are generated;

2. **First-Pass binary**: the build system compiles the source files into object files and links them into an initial binary;

3. **Final binary**: optimizations are applied and final linking is executed to produce an executable `.elf` file;

4. **Post-processing**: if necessary, the output file is converted to the format required by the loader or flash tool of the target platform, such as an `.hex` or `.bin` file.

3.2.3 Linkable Loadable Extensions

To achieve even greater flexibility, Zephyr provides the Linkable Loadable Extensions (LLEXT) framework, which allows extending an application's functionality at **runtime**. An extension is essentially a precompiled executable in ELF format that can be verified, loaded into memory, and dynamically linked to the main Zephyr binary that is already running on a device [23] [22].

By separating an application's functionalities into different extensions, it is possible to achieve great modularity and efficiency. In fact, the main application

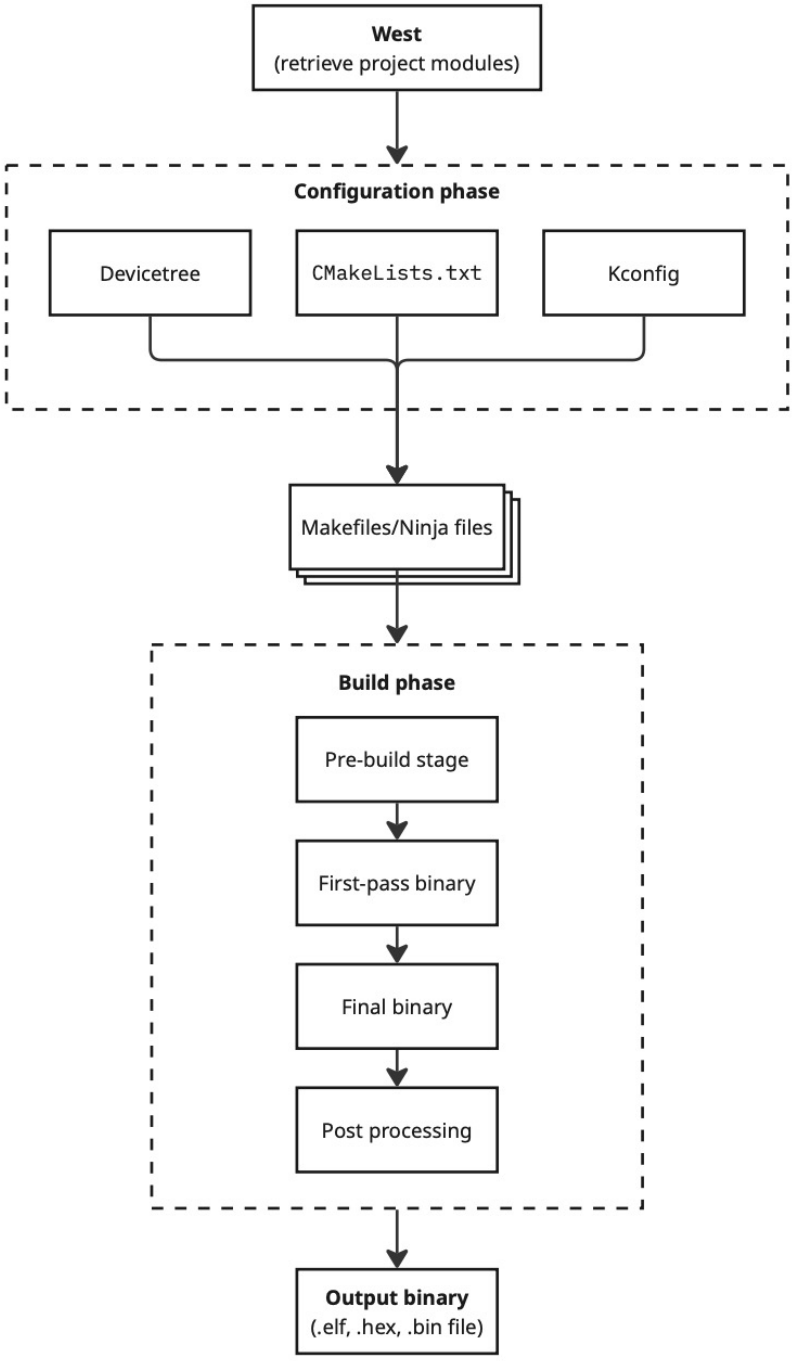

Figure 3.1: The Zephyr Build Process

firmware can remain static, while individual features can be added or updated directly on the running device, without performing a full firmware update. This leads to faster compilation cycles and smaller updates, as only the extensions must be recompiled.

Zephyr provides two methods for building extensions. The first is to build them directly as part of the main application through the native Zephyr CMake functionalities, ensuring that the exact same parameters are used. Instead, when extensions are developed separately, the LLEXT EDK (Extension Development Kit) can be leveraged, which is a tarball that contains the necessary headers and compile flags [23]. This allows for a completely decoupled extension development process.

An extension's lifecycle comprises four phases:

1. **Building**: the extension is compiled into an ELF file;
2. **Loading**: at runtime, the ELF data is loaded into memory and liked with the main application;
3. **Execution**: once loaded, the application is able to call the extension functions or access its variables;
4. **Unloading**: when the extension's functionality is no longer required, it can be unloaded, freeing all the memory it was using.

LLEXT allow for more dynamic system architectures compared to the one previously described, where both the application and the kernel code are compiled in a single binary.

3.3 The Arduino core based on Zephyr

An Arduino core is the implementation of the Arduino programming language for a particular microcontroller architecture. Its goal is to expose the same API regardless of what board is in use, acting as a middle layer between user applications and the underlying OS [24].

In a traditional Arduino core, such as those based on Mbed OS, the user programs (called *sketches* in Arduino terminology) are compiled along with the kernel into a standalone binary file and flashed on the microcontroller, which runs them without any external dependency. The integration of the Arduino cores and Zephyr RTOS introduces a significant architectural change. It moves from the monolithic build process to a model that takes advantage of the modularity and flexibility brought by the new OS [25].

The Zephyr kernel, along with its essential services, is compiled into a binary firmware image called `loader`, which is flashed onto the Arduino board using the

"Burn Bootloader" function provided by the Arduino IDE. The `loader` provides the runtime environment that manages the interaction between the sketches and the underlying Zephyr system. Therefore, the compilation and upload of the `loader` can be executed only once, while to compile and upload sketches, the LLEXT mechanism is leveraged instead. Sketches are compiled as ELF files, which are then dynamically loaded and linked to the `loader`. Moreover, the `loader` utilizes the *Devicetree*, which makes it work across different hardware platforms [26] [27].

Since only the user code and a few libraries have to be compiled, this new architecture offers faster compilation times and smaller binary files.

Chapter 4

Implementation

Now that the whole context in which this thesis project is set has been presented, it is possible to focus on the implementation of the testing framework. This chapter describes the overall strategy adopted, along with the implementation details and challenges faced during its development process.

The resulting system comprises a set of sketches (i.e., the programs running on the boards), which execute the hardware tests, and a Go program acting as orchestrator, that combines such tests into a cohesive and automated framework that can be run on any host machine.

4.1 Strategy

Validating a communication protocol requires verifying that a board can both receive and send data reliably. To this end, a loopback approach with a dual board setup is employed. One device, referred to as DUT (Device Under Test), runs the core under evaluation, while the other, denoted as Validator, runs a core known to be stable and reliable. The testing procedure is carried out as follows:

1. the Validator generates a sequence of data of a given length and computes its CRC (Cyclyc Redundancy Check);

2. it sends the data to the DUT;

3. the DUT receive the message and echoes it back without modifications;

4. the Validator computes the CRC of the DUT's response and matches it against the original one. If the values are equal, the test is successful; otherwise, it fails.

To minimize the risk of false negatives and isolate the behavior of the DUT as much as possible, both boards are carefully selected to be golden samples, that is, tried and tested devices free of hardware-related issues.

Despite its simplicity, this approach effectively verifies that data integrity is preserved in both communication flows in a way that can be applied to many protocols. Moreover, by changing the combination of various parameters, such as baud rate, clock speed, and ports, it is possible to create comprehensive test suites that thoroughly validate the communication interface.

As an example, the UART communication protocol can operate with several baud rate values, and a single Arduino board usually provides multiple serial ports. By repeating the test across various baud rates and message lengths on all available ports, a broad set of scenarios can be evaluated.

As for the orchestrator, its role is to set up the environment and assemble the test suites according to a configuration file provided by the developer, compile and upload the test sketches to the boards, and collect their outcomes.

Finally, this whole process is automated through the CI/CD platform, which executes it whenever new code is merged in the repository and gives accurate feedback to the developers (Figure 4.1).

One of the main advantages offered by this design is the flexibility and portability of the entire framework. It is not limited to run only on the CI pipeline, but developers can also run it locally to test their code even before committing it. Thus, reducing the strain on the pipeline itself. Another benefit is the relatively small computational power required of the host machine. This makes it possible to use a small, affordable, and low-energy computer to execute the framework, which is important since it must run continuously and helps to cut the infrastructure costs.

4.2 Tools and technologies

This section presents the hardware and software stack utilized to implement the strategy, along with additional tools that supported the development process.

4.2.1 Hardware

The Arduino GIGA R1 WiFi (Figure 4.2) board was primarily used, both as Validator and DUT. This device was chosen because it is the most features-rich in the standard line of Arduino products. It includes a 32-bit dual-core microcontroller and offers a wide range of hardware ports and GPIOs, making it an ideal testing platform. Moreover, it is the device for which the migration to Zephyr RTOS is most progressed. Limited tests were also conducted to verify basic compatibility of

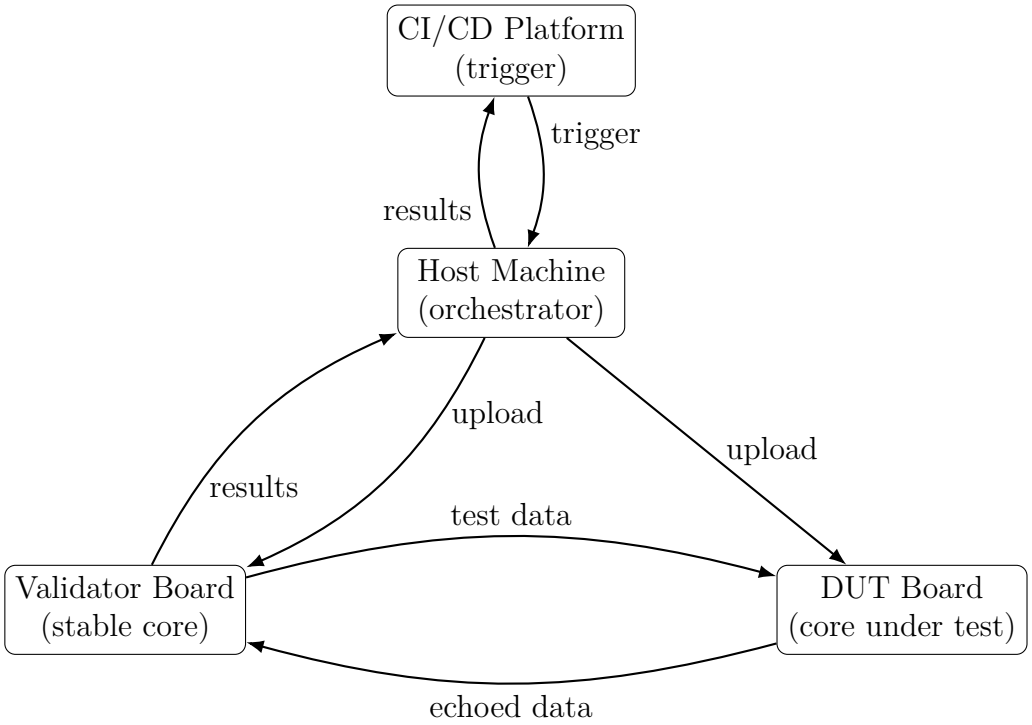

Figure 4.1: Overall testing strategy: the CI/CD platform triggers the host machine, which orchestrates the validation between the DUT and the Validator.

the framework with different cores via the Arduino Due R3 and Arduino Mega 2560 boards which share the same form factor and other similarities with the GIGA.

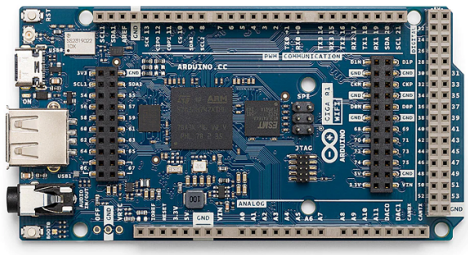

Figure 4.2: The Arduino GIGA R1 WiFi board.

A major drawback of the adopted strategy is the need to manually wire the two boards to enable communication. Since each protocol may require more than a single wire per port, and tests should be carried out for each port, the wiring can quickly become intricate and prone to mistakes. For this reason, two custom boards were designed: the Ruché v1 (Figure 4.3) and the Taurasi v1 (Figure 4.4).

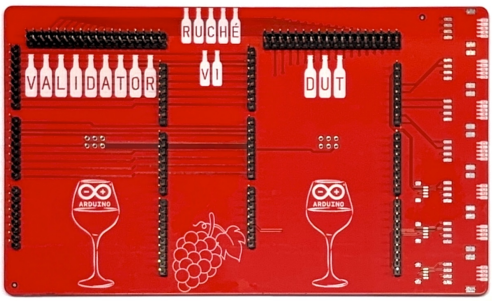

Figure 4.3: The Ruché board.

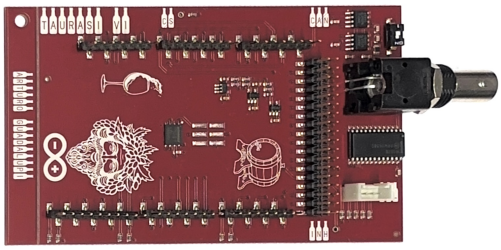

Figure 4.4: The Taurasi board.

Although their functions differ, they share the common goal of providing a compact and plug-and-play solution to the wiring issue. The Ruché is a simple PCB that connects to both the DUT and the Validator, replacing the jumper wires. It routes the signals coming from one board's pins to the corresponding pins on the other board. However, this alone is not sufficient because some protocols have additional requirements. For example, in a UART communication the transmission pin of one device must be wired to the receiving pin of its counterpart; also, to use some I2C ports, external pull-up resistors are required. The Taurasi was designed to address these protocol-related needs, acting as a middle layer between one of the two boards and the Ruché (Figure 4.5). This setup effectively achieves its intended objectives, but comes with the cost of requiring Ruché and Taurasi variations based on the form factor of the boards.

Lastly, as host machine, a Raspberry Pi 4 (Figure 4.6) was employed. This small computer strikes a good balance between performance, costs, and energy efficiency, making it suitable as the backbone of the CI pipeline.

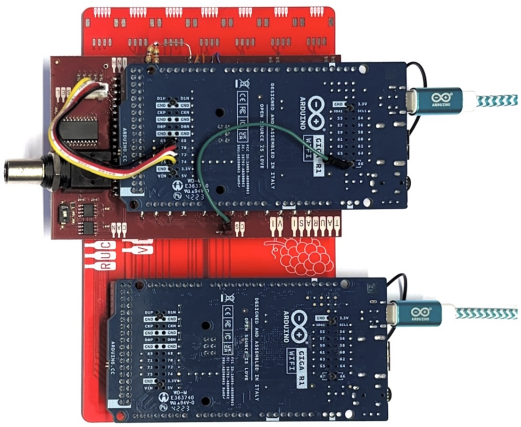

Figure 4.5: The setup of the Validator and the DUT using the Ruché and Taurasi to communicate with each other.

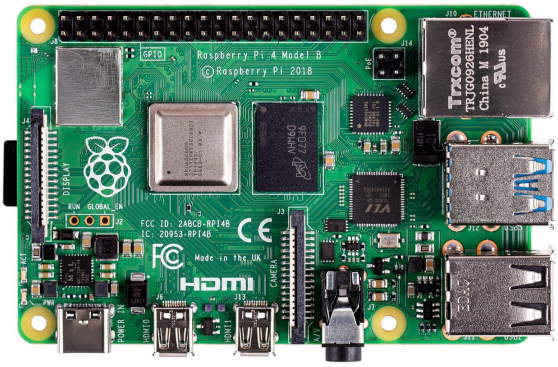

Figure 4.6: The Raspberry Pi 4.

4.2.2 Software

Regarding the cores, the Validator exclusively ran the *Mbed OS for Giga boards*. Different versions were used (mainly v4.2.4 and v4.3.1), as bug fixes required upgrading. The DUT instead used alternatively the *Arduino Zephyr Boards* (up to v0.3.2) and the same *Mbed OS for Giga boards* core as the Validator, including also v4.4.1, which is the latest available. Mbed was employed as a control mechanism for the framework itself, since the Zephyr-based core is still in beta, which makes it unstable and lacks some features.

Unit tests were written in the Arduino programming language, based on C++, while the orchestrator was implemented in Go. Go was chosen because it provides

strong cross-platform support for all major operating systems and well-established libraries for hardware-related tasks, such as serial communication. This ensures that the framework can run seamlessly on Windows, macOS, and Linux/UNIX systems.

To enable interaction between the host machine and the boards, the Arduino CLI is used. It provides all the necessary tools needed to use any Arduino compatible board from the command line [28]. Beyond compiling and uploading sketches, the Arduino CLI also supports important management operations such as installing and updating cores. In addition, the Arduino Test Utility (ATU) library is leveraged. This internally developed library allows sending custom commands to the board via Serial USB.

The selected CI/CD platform to automate the testing process is GitHub Actions because it integrates seamlessly with GitHub, the VCS utilized within the organization. It enables the creation of workflows that are triggered whenever an event occurs in the repository, such as the opening of pull requests. Each workflow is made up of one or more jobs which run on dedicated runners. In general, a runner is a machine, possibly a virtual one, on which a single job can be executed at a time [29]. By default, virtual runners are provided directly from GitHub, but in order to access hardware features, such as USB devices, a dedicated machine with a self-hosted runner is required.

4.2.3 Additional tools

Additional laboratory instruments, such as oscilloscopes and multimeters, were occasionally used to diagnose and analyze communication issues. Among them, the RedPitaya STEMlab 125-14 (Figure 4.7) has been particularly valuable. It is a compact, open-source measurement and control platform built around an ARM SoC combined with an FPGA, which enables real-time processing. The device features two analog inputs and two analog outputs sampled at 125Msps (samples per second) with a 14-bit resolution, through which it provides multiple software-defined instruments, including an oscilloscope, a signal generator, and a signal processor with logic, bode, and spectrum analysis capabilities.

Its functionality can be accessed either locally or remotely, using a Web interface or an SSH connection. In addition, users can program it through the SCPI standard, by uploading C and Python programs, or even by reprogramming the FPGA.

These features make it a highly versatile and flexible platform, capable of adapting to different contexts and several use cases. Furthermore, it offers an all-in-one and affordable alternative to traditional laboratory setups that would require multiple dedicated instruments.

For the purpose of this project, its oscilloscope and logic analyzer with protocol decoding capabilities proved particularly useful, as it allowed to accurately interpret

the signals exchanged between the boards, facilitating the identification of the cause of many failures.

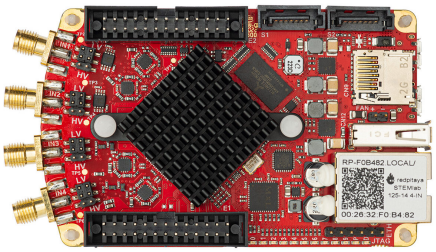

Figure 4.7: The RedPitaya STEMlab 125-14.

4.3 Implementation

The project is organized as a series of Go packages (Figure 4.8), each with a specific scope. The `main` package is the entry point and coordinates the workflow: it sets up the environment, runs the test suites, and collects the results into a file. To do so, it interacts with the `config` package, which loads the configuration file, validates its content, and populates the data structures required by the orchestrator. For each peripheral to be tested, a dedicated package is provided along with a pair of sketches, one for the Validator and the other for the DUT. These packages handle the compilation and upload of the sketches, send the Validator the necessary commands, and monitor its output to determine whether the test has passed or not; instead, the sketches carry out the unit tests. Finally, the `common` package contains shared utility methods.

This modular structure enhances the maintainability and scalability of the framework, making it easy to integrate new tests and possibly modify existing ones.

4.3.1 Validator and DUT sketches

All sketches share the same structure, consisting of a `setup()` function that is executed once, when the sketch is uploaded, and a `loop()` function that is instead

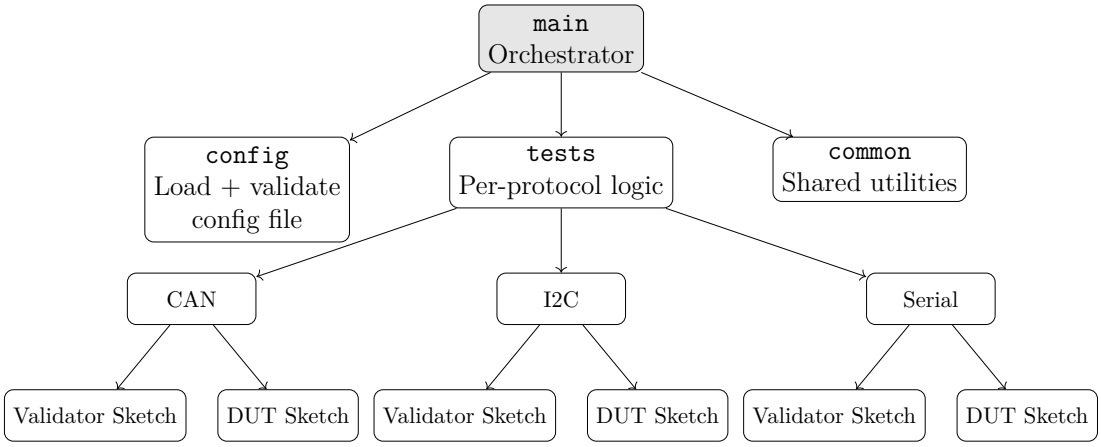

Figure 4.8: Organization of the project into Go packages and test sketches.

executed in a loop indefinitely.

The `setup()` function of both the Validator and the DUT initializes the ATU library, configures the ports based on the flags used during the compilation process, and sets the `Serial1` port for output logging. The Validator then waits to receive the command `SEND;X` where `X` is the number of bytes to send. Once the command has been read and parsed using the ATU library, it checks that the length is a positive number and does not exceed the limits imposed by the protocol specifications (for example, the buffer utilized for I2C communication limits the message length to maximum 32 bytes). Subsequently, it populates an array with `X` bytes, all with values ranging from 1 to 254, and computes its 8-bit CRC. Finally, it writes on the appropriate port the length of the array it is going to send, followed by the array itself, and waits for the DUT's response. The response bytes are counted and stored in a different array. If their number is not equal to the number of bytes sent, the test is rejected immediately. Otherwise, a new 8-bit CRC is computed and compared with the original one. In case they correspond, the test succeeds; differently, it fails. Finally, the result is logged via the ATU library, adhering to a precise pattern that allows the orchestrator to easily interpret the test results. In case of success, the output is "`RESULT:OK`"; on the contrary, the output reports the cause of failure (such as "`LESS_BYTES_RECEIVED`") followed by "`RESULT:KO`". If an error occurred instead, the "`ERROR`" string is outputted along with its explanation.

The DUT instead, once it reads the data sent by the Validator, it simply sends them back. Additional steps are introduced if the sketch is compiled in debug mode. Specifically, the data are logged along with a newly calculated 8-bit CRC value to help troubleshoot issues.

In order to enhance robustness, precautions are taken to avoid buffer overflow when reading a message. This is achieved by defining the maximum number of

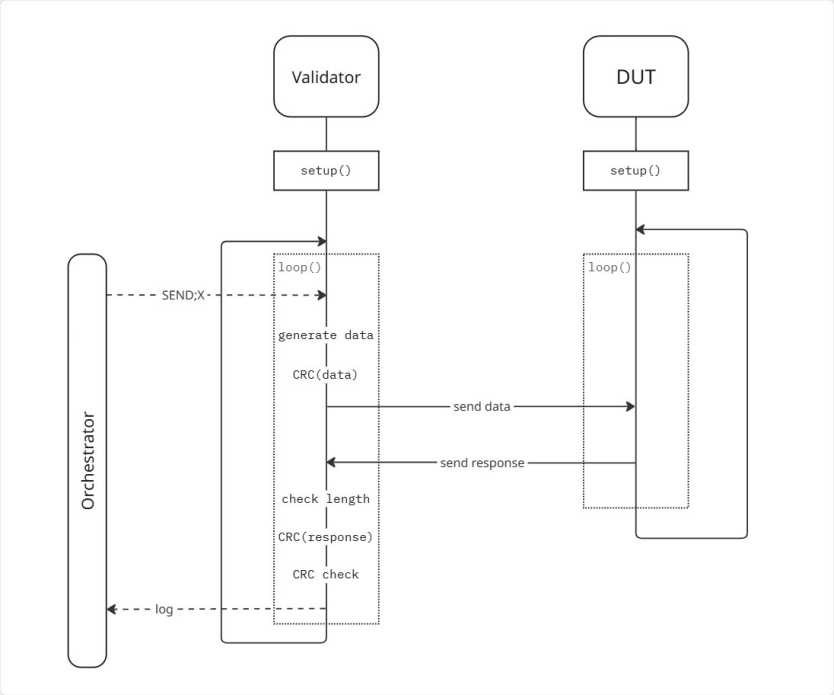

Figure 4.9: Diagram of the testing process.

bytes that can be received and keeping track of how many have been read. If the threshold is reached, the remaining incoming bytes will be discarded but still counted to produce a more accurate report.

```
while (SERIAL_TEST.available()) {
  if (bytesReceived < MAX_LEN)
    receivedData[bytesReceived++] = SERIAL_TEST.read();
  else {
    bytesReceived++;
    SERIAL_TEST.read();
  }
}
```

4.3.2 Orchestrator

Before running the tests, the orchestrator must configure the execution environment. To this end, the following sequence of operations is carried out:

- command line arguments parsing;
- board detection;

- configuration loading;
- configuration matching;
- test execution.

Since the orchestrator is the only contact point with the CI platform, it also features a structured exit-code system where a dedicated code is assigned to each error category. This design facilitates integration with the CI pipeline, where these codes can be parsed to determine whether a failure occurred and for what reason.

```
Exit Codes:
  0 - Success: All tests passed
  1 - Invalid Arguments: Missing or invalid command line arguments
  2 - Boards Error: Error detecting or configuring boards
  3 - Config Error: Error loading configuration
  4 - Core Install Error: Error installing Arduino cores
  5 - Test Failure: Test execution failed or test cases failed
  6 - File Write Error: Error writing results file
```

Command line arguments parsing

The `main` package exposes a command-line interface that lets users select which test suites to run using flags. The `main()` function parses the command line argument to extract the names of the tests that must be executed. These names must match those contained in the test registry, a data structure that maps the test packages with their names. This implementation makes the framework extensible, allowing one to introduce new tests by simply adding an entry to this map without altering the remaining logic.

```
var testRegistry = map[string]common.Test{
    "serialToHost": serialtohost.New(),
    "i2c":          i2c.New(),
    "serialMS":     serialMs.New(),
    "can":          can.New(),
}
```

Board detection

Board detection and assignment are handled within this package as well by querying the Arduino CLI for connected devices. The CLI returns detailed information describing each board, but only the attributes required to uniquely identify a board are retained and stored in a `Board` data structure. Precisely:

- the **serial number**: a 16 or 24 digits hexadecimal string, unique for each board;
- the **Fully Qualified Board Name (FQBN)**: a string specifying the make, architecture, and model of the board;
- the **port** through which the board is connected to the host machine;
- the **core name and version** that determine the runtime environment for test execution;
- the **board role** indicating whether the board is acting as the Validator, the DUT, or is still undefined.

```
type Board struct {
    SerialID string
    FQBN     string
    Port     string
    Core     Core
    Role     BoardRole
}

type Core struct {
    Name    string
    Version string
}

type BoardRole int

const (
    Undefined BoardRole = iota
    Validator
    DUT
)
```

At detection time, the `Core` field remains empty, as the Arduino CLI does not provide that information. However, it is later initialized together with the `Role` field, when the appropriate configuration is applied.

Configuration loading

The `LoadConfig()` function from the `config` package is then invoked. It parses the configuration file, validates its contents, and returns a reference to a `Config` data structure.

The configuration file defines both the hardware setup and the parameters for the test execution. Its structure is divided into two main sections:

- `boards`: specifies the FQBN and the serial number of both the Validator, and optionally the DUT, to ensure the roles are fixed and consistent. The core name and version that each board should run are also specified here.

- `tests`: lists the configurations for each test. Each entry contains the parameters required to build and run the corresponding sketches.

Below an example of a configuration file is presented, along with Table **??** which clarifies its meaning.

```
{
    "boards": {
        "validator": {
            "FQBN": "arduino:mbed_giga:giga",
            "serial_id": "0031001D3033510334323437",
            "Core": {
                "Name": "arduino:mbed_giga",
                "Version": "4.3.1"
            }
        },
        "dut": {
            "FQBN": "arduino:zephyr:giga",
            "serial_id": "002D00343033511934393531",
            "Core": {
                "Name": "arduino:zephyr",
                "Version": "0.3.1"
            }
        }
    },
    "tests": {
        "serial_master_slave": {
            "serial_test": [
                "Serial1",
                "Serial2"
            ],
            "baud_rate": [
                9600,
                115200
            ],
            "serial_out": "Serial"
        },
        "i2c": {
            "wire_test": [
                "Wire"
            ],
            "clock": [
                100000
```

```
38                ],
39                "slave_addr": 16,
40                "serial_out": "Serial"
41            }
42        }
43    }
```

Field	Description
`boards.validator.FQBN`	Fully Qualified Board Name of the Validator
`boards.validator.Core.Name`	Name of the core used by the Validator
`boards.validator.Core.Version`	Version of the Validator core
`boards.dut.FQBN`	Fully Qualified Board Name of the DUT
`boards.dut.Core.Name`	Name of the core used by the DUT
`boards.dut.Core.Version`	Version of the DUT core
`tests.serial_master_slave.serial_test`	List of serial ports to test
`tests.serial_master_slave.baud_rate`	List of baud rates for testing
`tests.serial_master_slave.serial_out`	Serial port for logging
`tests.i2c.wire_test`	I2C bus to test
`tests.i2c.clock`	I2C clock speeds in Hz
`tests.i2c.slave_addr`	I2C slave address
`tests.i2c.serial_out`	Serial port for logging
`tests.can.bit_rate`	CAN bus bit rates
`tests.can.serial_out`	Serial port for logging

Table 4.1: Example of configuration file structure

This design allows users to adapt to different hardware setups or parameter values by simply editing this file without modifying the code base.

Configuration matching

After installing or updating the required cores, the `main` function calls the `matchBoardsWithConfig()` function, which tries to match the previously detected boards

with the loaded configuration. The boards are assigned their role based on their serial numbers.

Here, a special case is treated. As explained in the previous chapter, when switching from a traditional core to the Zephyr-based one, the Zephyr kernel must be flashed first on the board (through the "Burn Bootloader" function) before being able to upload any sketch. To handle this, the FQBN of the detected board is compared with the FQBN of the required configuration: if the latter is `arduino:zephyr:giga` while the first is not, it means that the bootloader must be flashed. This operation is executed by the `BurnBootolader()` function situated in the `common` package, which forces the board into DFU mode by briefly opening the serial port at 1200 baud, locates the appropriate DFU-util binary and firmware image, and flashes the binary at the target memory address.

Finally, the `Board` data structures are updated to match the desired configuration. Thus, completing the setup of the execution environment.

Test execution

To ensure clarity and enhance modularity, a three level test taxonomy was defined:

- **test suite**: validation suite for one communication protocol across all possible parameter combinations;
- **test case**: a fixed parameters configuration over which different sequence lengths tests are executed;
- **unit test**: atomic test that uses fixed sequence length.

The code was structured to reflect this taxonomy. Each test suite is represented by a Go package exposing a constructor. This constructor returns a value implementing the `TestSuite` interface, which provides two methods:

- `Name()`: returns the name of the test suite. It can be compared with the arguments passed to the `main` function to determine which test should be run;
- `Run()`: the method that actually executes the tests. It takes as arguments the configuration and an array of boards, thus not limiting tests execution to a dual-board setup, but also enabling for other configurations. The method returns an array of `TestCase` data structure or an error in case a failure occurs during the process.

```
1 type TestSuite interface {
2     Name() string
```

```
    Run(config config.Config, boards []Board) ([]TestCase, error)
}

type TestCase struct {
    Name    string
    Params  map[string]string
    Results []UnitTestResult
    Success bool
}

type UnitTestResult struct {
    Name    string
    Success bool
    Message string
}
```

To execute the tests, the `main()` function invokes the `Run()` method for each test suite. This method instantiates the test cases based on the parameters provided by the `Config` data structure. For each test case, it compiles and uploads the sketches via Arduino CLI. Then, it opens the serial port to send the Validator the appropriate commands and reads its response, storing the results in an array of `UnitTest` while also logging them to the terminal. Here, the success or failure of the unit test is also detected based on the response pattern (that is, the text "`RESULT:OK`" or "`RESULT:KO`").

Once execution of all test suites is complete, the results are aggregated and written to a timestamped JSON file reporting detailed outcomes of each unit test along with the configuration that produced them. An example of this file is shown below.

```
{
  "I2C Test": [
    {
      "Name": "Wire_100000",
      "Params": {
        "CLOCK": "100000",
        "SERIAL_OUT": "Serial",
        "SLAVE_ADDR": "16",
        "WIRE_TEST": "Wire"
      },
      "Results": [
        {
          "Name": "Send 1 bytes",
          "Success": true,
          "Message": "RESULT:OK"
        },
        {
```

```
            "Name": "Send 2 bytes",
            "Success": true,
            "Message": "RESULT:OK"
          },
          ...
        ]
      },
      {
        "Name": "Wire1_100000",
        ...
      },
      ...
  ],
  "CAN Tests": [
      ...
  ],
  ...
}
```

This design brings flexibility to the framework, allowing to easily introduce new unit tests and test suites. One can choose to take advantage of built-in functions, as long as they adhere to the predefined messaging scheme, or to adopt entirely different strategies provided that it implements the `TestSuite` interface.

4.3.3 Integration with GitHub Actions

The GitHub Action is executed on a self-hosted runner hosted on the Raspberry PI and is automatically triggered by a push or pull request on the core's repository. It can also be started manually with the option of providing the names of the tests to run. The Action consists of a workflow written in YAML, which runs a single job performing the following steps:

1. Clean up the workspace by removing any directory and files that may have been left from previous run. This step ensures isolation between executions;
2. cloning of the repository containing the framework to make sure that the most up-to-date version is used;
3. set up of the Go toolchain and installation of the required dependencies;
4. execution of the framework by constructing and invoking the appropriate command;
5. output monitoring and classification through exit codes and logs inspection. In this way, if a test does not succeed or an error occurs, the workflow is tagged as failed;

6. upload of the results file as an artifact to make it available directly through the repository's Web page;

7. final clean up of the workspace.

As shown in the snippet presented below, errors in the testing process are detected by running a shell script that checks exit codes and scans the logs for error keywords.

```
    error_found=false
    if [ -f "test_output.log" ]; then
      if grep -iq -E "(error|fail|failed|failure)" test_output.log;
    then
        error_found=true
        echo "Error keywords detected in test output"
      fi
    fi

    if [ $exit_code -ne 0 ] || [ "$error_found" = true ]; then
      if [ $exit_code -ne 0 ]; then
        echo "::error::Hardware tests failed with exit code:
    $exit_code"
      fi
      if [ "$error_found" = true ]; then
        echo "::error::Hardware tests failed: Error keywords detected
     in output"
      fi

      exit 1
    fi
```

To avoid concurrency issues, this Action is configured so that if a new run is requested while another is still in progress, the request is stalled until the previous one has completed. This precaution is necessary because only one hardware environment is available, and thus only one request at a time can be fulfilled.

4.4 Challenges

Many challenges arose during the development of this project. Most of them ascribe to the instability of some Arduino platforms, including those presumed to be more reliable. While this corroborates the need for a more grounded approach in the validation of newly released code, it also made the construction of the framework more complex.

First of all, the Zephyr-based core, being still in beta, lacked some features that either forced the adoption of alternative strategies or prevented some components

from being tested at all. For example, the `Wire` library (which supports I2C communications) normally offers interrupt-based mechanisms to handle data receptions or requests. However, these mechanisms have yet to be implemented in the Zephyr cores, and hence a polling approach had to be used. To support both situations, the sketches use preprocessor directives to conditionally compile pieces of code based on the underlying core.

```
1   void setup() {
2       // initialization of the required protocols
3       ...
4
5   // if not using ZEPHYR, use interrupts
6   #ifndef ARDUINO_ARCH_ZEPHYR
7       WIRE_TEST.onReceive(receiveEvent);
8       WIRE_TEST.onRequest(requestEvent);
9   #endif
10  }
11
12  void loop() {
13  // if using ZEPHYR, use polling
14  #ifdef ARDUINO_ARCH_ZEPHYR
15      readData();
16      WIRE_TEST.write(receivedData, dataLen);
17  #endif
18      ...
19  }
```

Although effective, this solution implies that once the interrupt mechanism is ported to the Arduino Zephyr core, the sketches will require modifications. Other limitations of Zephyr cores include the absence of the CAN library, which restricted the related tests to the Mbed OS-based core only, and the presence of various bugs in some important functionalities, an example of which will be described in chapter 6.

Since the Zephyr-based core was expected to present this kind of issues, testing was also carried out on the Mbed OS-based core. Nevertheless, it still presented some problems. As an example, while developing the I2C tests, an anomalous behavior was encountered, where data were occasionally lost during transmission. Analysis of the signals with the STEMlab 125-14's logic analyzer revealed that the receiving end discarded incoming messages twice before finally accepting them. Testing across different core versions revealed that v4.2.1 worked correctly. Based on this information, developers responsible for the Mbed-based cores were able to identify and fix the issue, which stemmed from a read operation performed inside a critical section. Because this operation was relatively slow, it masked interrupts for an extended period of time, resulting in message rejections.

Modifications to the standard configuration of the Arduino environment were also required. By default, when installing a new core version, if one was already present, it is removed. This prevents two sketches from running with two different versions of the same core, which is instead needed in order to test new releases against older ones. To overcome this issue, two separate environments had to be created. This was achieved by creating two different Arduino configuration files, one for the DUT's environment and the other for the Validator's, which specify the directories where the cores and the packages must be installed. Arduino CLI supports this kind of configuration by offering the `-config-file` option, available for several commands, that lets one indicate the path to the configuration file that must be used. Thus, it was sufficient to modify the orchestrator by adding this option whenever needed (for example, when installing new cores or compiling and uploading sketches).

A crucial aspect that must be closely monitored is the amount of time required to execute the framework, which may hinder its scalability. A substantial portion of this time comes from the compilation and upload steps, which take significantly longer than all other stages. For this reason, an attempt was made to parallelize these operations for the DUT and the Validator by using goroutines (i.e. lightweight threads managed by the Go runtime). However, this approach consistently failed because when uploading the sketches, the boards enter their bootloader mode, and this prevents the DFU tool from distinguishing them, causing the process to abort. Due to time constraints, this attempt was abandoned, but a successful implementation could substantially reduce the overall execution time of the framework.

This is not the only issue related to DFU as sometimes, during the upload process, DFU-util fails to detect the connected USB device, resulting in an error. A simple workaround consists of putting the board in bootloader mode before starting the upload. Since the same operation is also required by the `BurnBootloader()` function, its logic was abstracted into a dedicated function named `TouchPort()`. Both `BurnBootloader()` and `UploadSketch()` were updated to invoke this new one.

```
1  func TouchPort(board Board) error {
2      mode := &serial.Mode{
3          BaudRate: 1200,
4          Parity:   serial.NoParity,
5          DataBits: 8,
6          StopBits: serial.OneStopBit,
7      }
8
9      port, err := serial.Open(board.Port, mode)
10     if err != nil {
```

```
11          return fmt.Errorf("error: error opening port: %v", err)
12      }
13      defer port.Close() // Closing the port after use
14
15      time.Sleep(100 * time.Millisecond)
16
17      return nil
18  }
```

Other challenges also emerged during the development of the framework, including issues related to limited stability of Windows 11 for low-level development, which ultimately forced switching to Linux Ubuntu. Nevertheless, the examples presented above constitute a representative overview of the difficulties encountered, highlighting how achieving reliability in complex systems may be challenging.

Chapter 5

Additional work

Thanks to the modularity of the framework, it is possible to integrate more test suites to expand its capabilities. As mentioned in section 2.3, earlier tests of PWM functionalities were already developed. These are based on an orchestrator that utilizes the ATU library to control the DUT and sends SCPI commands to laboratory instruments. Therefore, they align well with the design of the presented pipeline and could be adapted to leverage the STEMlab 125-14, making it relatively easy to integrate them. However, due to limitations of their implementation, a major refactor would be necessary.

To understand the root cause of the issue, it is necessary first to briefly introduce the SCPI standard.

5.1 The SCPI standard

Standard Commands for Programmable Instruments (SCPI) is a standardized interface language that defines the organization and content of messages for communication between computers and laboratory instruments, such as oscilloscopes, power supplies, and multimeters. Its aim is to provide a consistent programming environment across devices and manufacturers. This consistency is achieved both by defining standard commands within a specific class of instruments (vertical consistency) and by using the same commands to control similar functions across different classes of instruments (horizontal consistency). These commands are organized in a tree structure that groups related functions together under a common node. To execute a command, its full path in the hierarchy must be specified.

Commands are case-insensitive[1] self-explanatory ASCII strings, making them

[1]SCPI commands are often written in mixed-case. Uppercase letters indicate the mandatory part of the command, while lowercase ones are optional. However, they remain case-insensitive.

easy to learn and compatible with a variety of programming languages and environments. Each command starts with a column `:` and can take various parameters of different types that are needed to configure instruments. For instance, addressing a specific oscilloscope channel is achieved with the command `:CHANnel<n>`, where `n` is an integer; setting the trigger mode is done via `:TRIGger:MODe <mode>`, where `mode` can assume one of several predefined strings values, such as `EDGE`, `SLOPe`, or `PULSe`. Furthermore, commands can also be used in query form via the `?` operator, which enables retrieving information from the instrument. For example, sending the command `:TRIGger:STATus?` requests the trigger status, which is returned as a string.

Through this mechanism, SCPI provides comprehensive remote control of an instrument, eliminating the need for physical interaction.

5.2 Integration

The PWM tests were implemented using three libraries that provide some Go functions to connect to and command an oscilloscope, a multimeter, and a power supply. Each library exposes a dedicated function for each SCPI command, resulting in an enormous number of functions that differ only in the command they send to the device. As an example, the functions to set the trigger mode of an oscilloscope and to set it to autoscale its axes are presented.

```
1  func SetSingleTriggerMode(conn net.Conn) error {
2      cmd := ":SINGle"
3      _, err := conn.Write([]byte(cmd + "\r\n"))
4      if err != nil {
5          return fmt.Errorf("failed to send SINGLE trigger mode command
       : %v", err)
6      }
7      time.Sleep(100 * time.Millisecond)
8      return nil
9  }
10
11 func AutoScale(conn net.Conn) error {
12     cmd := ":AUTOscale"
13     _, err := conn.Write([]byte(cmd + "\r\n"))
14     if err != nil {
15         return fmt.Errorf("failed to send AUTOscale command: %v", err
       )
16     }
17     time.Sleep(100 * time.Millisecond)
18     return nil
19 }
```

It is clear that both of these functions are almost identical and could benefit from a redesign to reduce the amount of duplicated code.

Although this approach makes it straightforward for the orchestrator to interact with SCPI devices, it does not scale. Because the SCPI standard comprises a very large set of commands, the libraries grew to unmanageable sizes (e.g., 10,000 lines of code for the oscilloscope one, and 5,000-7,000 lines for the other two), making them difficult to use and nearly impossible to maintain. For this reason, the integration process with the new pipeline was halted to find an alternative design approach.

Some basic metrics were established to evaluate new design proposals:

- **feasibility**: the effort required to adapt the libraries;
- **compatibility**: the effort required to adapt the orchestrator to the new implementation;
- **usability**: the ease with which the new libraries can be used and maintained.

In addition, it was important to try preserving an interface that would prevent programmers from remembering the exact SCPI syntax. Instead, they should be able to call the functions directly, at least for the most used ones.

5.2.1 Design A: wrapper functions

The first approach proposed abstracting the logic for sending SCPI commands into a single function, while keeping all other existing functions as wrappers around the new one.

Following the example above, this design would yield a similar result:

```
func sendSCPIcommand(conn net.Conn, cmd string) error {
    _, err := conn.Write([]byte(cmd + "\r\n"))
    if err != nil {
        return fmt.Errorf("failed to send %s command: %v", cmd, err)
    }
    time.Sleep(100 * time.Millisecond)
}

func SetSingleTriggerMode(conn net.Conn) error {
    return sendSCPIcommand(conn, ":SINGle")
}

func AutoScale(conn net.Conn) error {
    return sendSCPIcommand(conn, ":AUToscale")
}
```

In terms of feasibility, this design requires little modification to the libraries and also preserves full compatibility, as the external interface remains unchanged. From a usability perspective, the lines of code would be reduced to approximately 33% of the original size, but it would still retain a very large number of similar wrapper functions, making the benefits in maintainability and scalability marginal.

5.2.2 Design B: dictionary

Another evaluated design consisted of defining a JSON schema containing the commands, along with related information, such as their parameters. In this way, the library would only need to consult the JSON dictionary to build the command string and send it to the instrument. For example, the command for setting an oscilloscope channel's scale would be represented by the following schema:

```
{
    "commands": {
        "SetChannelScale": {
            "template": ":CHANnel{channel}:SCALe {scale}",
            "params": {
                "channel": "int",
                "scale": "float"
            }
        }
    }
}
```

The orchestrator would call an intermediary method, passing it the command's name and the parameters required:

```
    Call("SetChannelScale", 1, 1.0)
```

The intermediary method would need to locate the appropriate command, check that the parameters are correct, build it, and send it to the instrument.

In terms of feasibility, this approach requires completely discarding the existing libraries, focusing the effort on designing the intermediary method. As a result, compatibility would also be negatively affected, since a completely different interface is introduced. However, usability would be improved: this design would eliminate the need for separate libraries for each lab instrument and significantly reduces code size. Adding new commands would require only creating new entries in the dictionary, which is a more user-friendly way than creating wrapper functions, which does not require changing the codebase. Moreover, the orchestrator would only need to invoke the intermediary method. The major drawback would be the

long-term maintainability of the JSON schema, which would eventually suffer from the same scalability issues as the previous approaches.

While other design approaches were still being defined, the decision was taken to proceed with this solution, implementing a simple use case to test its actual practicality. At this stage, it was also decided to replace JSON with TOML, which offers similar functionality and is already employed within Arduino for other purposes.

The use case

The chosen use case consisted of taking a screenshot of a signal waveform with an oscilloscope. To achieve this, the program needs to:

1. connect to the oscilloscope via TCP;
2. set the proper scale for the utilized channel;
3. adjust the trigger mode, edge slope, and level;
4. wait for a signal to be detected;
5. once detected, take a screenshot of the signal's waveform and save it.

This scenario provides a set of SCPI commands with various parameters and formats. Therefore, it serves as a meaningful test case for validating the dictionary-based approach, assessing the library's ability to cope with different commands, and the ease with which the master program can leverage the library.

Implementation of the library prototype

The first step was to define the structure of the dictionary. Each entry must offer a user-friendly name, the corresponding SCPI string, the parameters' type and format, and the return type. The following snippet shows an implementation of two commands: one for setting the trigger sweep and the other to take the screenshot of the waveform.

```
1 [command.SetTriggerSweep]
2 type = "command"
3 template = ":TRIGger:SWEep {sweep}"
4 params = { sweep = "string" }
5 [command.SetTriggerSweep.validations]
6     sweep = ["AUTO", "NORMal", "SINGle"]
7
```

```
[command.QueryDisplayData]
type = "query"
template = ":DISPlay:DATA? {color},{invert},{format}"
params = { color = "bool", invert = "bool", format = "string" }
returns = "bytes"
```

Compared to the conceptual JSON schema shown above, this design introduces some modifications: the `type` field that is utilized to distinguish commands from queries and binary queries, which must be handled by the library in different ways, enhanced parameters' validation through a list of accepted values, and finally the support for optional parameters, indicated by placing a `?` after their type (e.g., params = { invert = "bool?"}, means that `invert` is optional and may not be provided at all).

The library entry point is the `NewTCPDevice()` function, which establishes a TCP connection with an instrument given its IP address and port number, loads the TOML command dictionary, and returns a `TCPDevice`. The `TCPDevice` type provides the means to manage the connection and gives access to the dictionary itself, so that developers do not have to remember the commands.

```
func NewTCPDevice(ip string, port string, delay time.Duration) (*
    TCPDevice, error)

type TCPDevice struct {
    Conn     net.Conn
    Delay    time.Duration
    commands map[string]CommandDefinition
}
```

The core functionality is provided by the `Call()` method:

```
func (d *TCPDevice) Call(name string, args ...any) (any, error)
```

This is the intermediary method that enables calling the SCPI commands. It takes as input the user-friendly name of a command and its parameters. To support multiple parameters of different types, it leverages two Go features: the `any` type (alias for an empty interface) to store variables of any type and the variadic `...` operator to accept an indefinite number of arguments. The method first checks whether the command exists and then constructs the SCPI string while validating its parameters. Finally, based on the `type` field, invokes the `Send()` or `Query()` methods to actually send the command. It returns the result data, in case of a query, or an error if something went wrong.

At this stage it was also realized that an additional `type` of command was required, that is, binary queries. This new type is reserved to those queries

that, instead of returning results in string format, return raw bytes, which must be handled differently. In this use case, this applies to the `QueryDisplayData` command, which retrieves an image.

The `buildParamsWithValidation()` function is responsible for building and validating the parameters that will be substituted in the SCPI templates.

```
func buildParamsWithValidation(cmdDef CommandDefinition, args []any,
    name string) (map[string]any, error) {
    params := make(map[string]any)
    i := 0

    for paramName, paramType := range cmdDef.Params {
        isOptional := strings.HasSuffix(paramType, "?")

        isOptional := strings.HasSuffix(paramType, "?")
        if i >= len(args) && !isOptional {
            return nil, fmt.Errorf("missing required parameter %s",
    paramName)
        }

        params[paramName] = args[i]

        if validationList, ok := cmdDef.Validation[paramName]; ok {
            if !isValidValue(argStr, validationList) {
                return nil, fmt.Errorf("invalid value for %s",
    paramName)
            }
        }

        i++
    }

    return params, nil
}
```

The function checks if a value was supplied for each parameter defined in the command specification. If not and the parameter is not optional, it returns an error. If the command definition provides a set of valid values, it also checks that the arguments provided match one of them via the `isValidValue()` function, preventing execution with invalid options. Since this was a quick prototype, type validation was temporarily omitted, but it could be performed in this function as well. Finally, if all checks pass, a map that matches the parameters' names with their values is returned.

The map is used by the `buildCommand()` function, which replaces all placeholders in the command with their actual value. For example, in the template `:DISPlay:DATA? {color},{invert},{format}`, the placeholder `{color}`, `{invert}`, and `{format}` are substituted with their respective values provided by the map. This function also converts boolean values in their SCPI-compliant equivalents `ON` and `OFF`.

Finally, to give developers the option of issuing commands that are not yet in the dictionary, the library exposes two additional methods: `Send()` and `Query()`. These methods accept a raw string representing a SCPI command and try to execute it directly, without any type of validation. Therefore, responsibility of ensuring the correctness of the command rests entirely on the caller.

Evaluation

With the architecture described above, the master program could connect to the oscilloscope and invoke the `Call()` method for each required command. An excerpt of its code is shown below:

```
func main() {
    osc, err := scpidevice.NewTCPDevice(*ip, *port, 150*time.
    Millisecond)
    defer osc.Close

    _, err = osc.Call("SetChannelScale", 1, 1.0)
    _, err = osc.Call("SetTriggerMode", "EDGE")

    for {
        status, err := osc.Call("QueryTriggerStatus")

        if strings.TrimSpace(status) == "STOP" {
            break
        }
    }

    response, err := osc.Call("QueryDisplayData", true, false, "PNG")
}
```

This implementation successfully executed the entire use case and captured the desired waveform (Figure 5.1), proving the functionality of the design. Nevertheless, it also exposed its expected limitations, along with others that were not anticipated.

One of the main challenges that arose is the difficulty of expressing complex SCPI commands within a TOML dictionary. While this approach works for many commands with few parameters, it becomes hard to adapt for instruments such as multimeters, which often provide multiple command variants, parameter

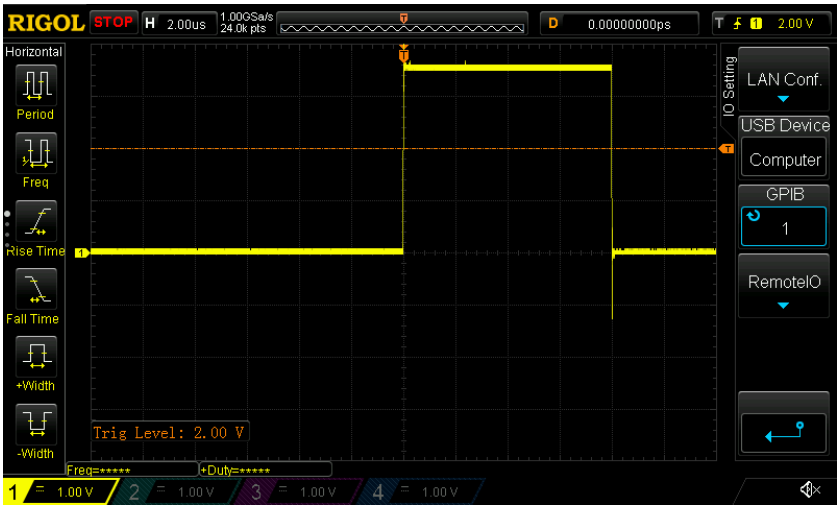

Figure 5.1: Screenshot of a PWM signal taken by commanding a Rigol DS1054Z oscilloscope through SCPI.

set, or parameters dependencies. This complexity also affects the library itself, which cannot rely only on generic methods for sending commands and interpreting responses, but needs to adapt to their peculiarities (the need for implementing the `QueryBinaryData()` function instead of using the `Query()` one is an example of this problem).

Moreover, the dimensions of the dictionary were underestimated: the number of possible commands, together with their query variant, makes it impossible to represent them all in a dictionary, even if it was split in multiple files.

These findings demonstrated that, although functional, the dictionary-based design lacks long-term scalability and maintainability. For this reason, its development was halted.

5.2.3 Alternative design proposal

The challenges faced during the development of the dictionary-based design made it clear that a different approach could bring significant benefits. One possible alternative would be to mimic the inherent tree structure of SCPI commands. This could capture their versatility and variants, while still freeing developers from remembering the correct syntax and keeping the dimension of the library under control.

A simple and practical example can better illustrate the rationale behind this proposal. The `:CHANnel<n>` command branches into multiple subcommands, such as `:COUPling`,`:DISPlay`, and `:SCALe`, all related to a single channel. Attempting to represent each of these as separate command, each validating the parameter

`n`, has proved impractical. Furthermore, some subcommands recur across other different commands. A natural approach would be to assign a function to each subcommand, which would take the required parameters and validate them, and then allow chaining these functions together in the same way as SCPI itself. The following illustrative snippets show how this could look:

```
func Channel(n int) string;
func Coupling(type string, isQuery bool) string;
func Display(isOn bool, isQuery bool) string;
func Scale(mVolt float, isQuery bool) string;
```

Commands could then be built like:

```
oscilloscope.Channel(1).Coupling("AC", false);
oscilloscope.Channel(2).Scale(0.0, true);
```

With a careful design, it could be possible to validate method chaining, enabling developers to leverage IDE autocompletion. However, this approach would require a significant implementation effort.

The major recurring challenge of all these approaches lies in the requirement of avoiding developers to writing SCPI string, while providing a scalable and maintainable library.

However, this remains a theoretical proposal that has not been further explored, as time constraints shifted the focus back on the main objective of the thesis: the pipeline. Future work could reevaluate this proposal or investigate alternative designs.

Chapter 6

Results

The main goal set for this project was to build a proof-of-concept of a Continuous Integration pipeline with Hardware-in-the-Loop capable of validating communication protocols on Arduino boards whenever a new version of the Zephyr-based core is ready for release.

In this chapter, it will be discussed whether this goal was met and to what extent, by performing a qualitative assessment and presenting output logs and screenshots from different runs.

6.1 Qualitative assessment

The evaluation of the pipeline is based on three main properties that are considered key enablers for its systematic adoption and integration in the current Arduino workflow:

- **functionality**: its ability to execute reliably and produce consistent results;
- **usability**: the effort required to configure it, run it, and interpret its outcome;
- **maintainability**: the ease with which it can be modified, extended, or adapted to new requirements.

Before presenting the evaluation, the experimental setup used to execute the pipeline is described.

6.1.1 Evaluation setup

As shown in Table 6.1, the final setup with which the pipeline was tested consisted of two different configurations to overcome the issues of the Zephyr-based core discussed in the previous chapter.

Board	Core	Version
Validator	Mbed OS	4.3.1
DUT	Mbed OS	4.4.1
	Zephyr RTOS	0.3.1, 0.3.2

Table 6.1: Test setup.

UART, I2C, and CAN are the communication protocols that were targeted for the framework's development. This resulted in the implementation of four distinct test suites: one each for I2C and CAN, and two for UART to validate both board-to-board and board-to-host communication. However, as already mentioned, the CAN test suite could only be executed on the Mbed OS core.

All board-to-board suites include multiple test cases that combine the most common parameter values, while the board-to-host test suite uses the standard configuration typically employed for serial communication (i.e., 115200 baud). This approach ensures comprehensive validation of the protocols under realistic conditions. The specific test parameters are summarized in Table 6.2.

Test suite	Test parameters	Nr. test cases
UART board-to-host	Baud rate: 115200 Port: Serial Max length: 32B	1
UART board-to-board	Baud rate: 9600, 115200 Port: Serial 1, Serial 2, Serial 3, Serial 4 Max length: 128B	8
I2C	Clock speed: 100000, 400000 Port: Wire 1, Wire 2, Wire 3 Max length: 32B	6
CAN	Bit rate: 125000, 250000 Max length: 8B	2

Table 6.2: Test suites and parameters for supported protocols.

The GitHub Action was implemented in two repositories forked from the official Arduino ones, namely `ArduinoCore-mbed` and `ArduinoCore-zephyr`. This was done to avoid testing on the production environment, which would have been unsafe and required special permissions for workflow execution.

Finally, figure 6.1 shows the physical setup on which the pipeline was run, demonstrating its compactness. In detail:

a) the laptop from which the pipeline is triggered and monitored. It is used to simulate the actions from a remote developer;

b) the Raspberry Pi executing the framework;

c) the STEMlab 125-14 connected to the board to capture UART and I2C communications;

d) the Validator and DUT boards connected through the Ruché and Taurasi.

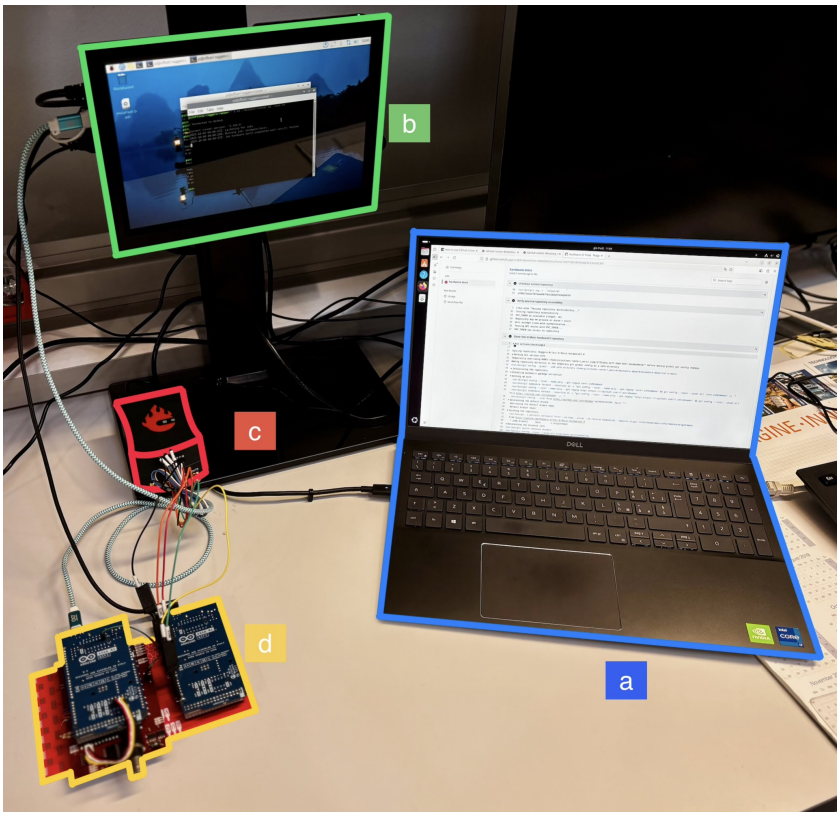

Figure 6.1: The physical setup on which the workflow runs on.

6.1.2 Evaluation

Each component of the project will be analyzed with respect to the key properties described above to provide an overall qualitative assessment.

Configuration mechanism

The configuration mechanism gives developers full control over the execution environment. Through a single file, they can define in detail the boards' setups, indicating their roles, along with their respective cores and versions, and every test parameter to achieve both highly specific and comprehensive suites covering several different configurations. Instead, specifying the appropriate flags when running the program allowed to selectively execute the test suites.

From a usability perspective, the single-file approach simplifies management: the entire pipeline's execution environment is described and maintained in a single place, that is also easy to share across different teams if needed. Moreover, the format chosen (i.e., JSON) is self-explanatory and human-readable.

The flexibility of such approach also enhances its maintainability. The JSON schema can be easily extended to support more tests and different setups. The wide support of the format also determines availability of efficient parsing libraries that can manage even large files without excessively affecting performance, making the system scalable.

The main drawback concerns its long-term usability. As the system evolves, a single file could become large, error-prone, and difficult to navigate. Future iterations could split the configuration across multiple files, or offer a UI that generates the JSON automatically.

Orchestrator

The orchestrator proved capable of reliably interpreting the configuration and setting up the pipeline accordingly. It consistently identified the boards and assigned their roles without mix-ups independently of the USB port they were connected to (Figure 6.2). This behavior is particularly important, as it eliminates potential wiring errors and, when combined with the Ruché and the Taurasi, makes the whole pipeline user-friendly and robust by abstracting away physical connections.

In case the hardware could not support the requirements (such as trying to use the wrong board type or a non-existent core version), the pipeline aborted gracefully. Otherwise, it executed only the requested suites and correctly assembled test cases with the specified parameters. The modular approach adopted also demonstrated highly effective for maintainability and usability. New test suites can be added, and existing ones can be extended without modifying the rest of the codebase as long as the defined interface is respected.

The usability of the framework is also supported by the feedback mechanism. It was able to accurately produce both real-time logging in the terminal (Figure 6.3) and detailed results files. The logs provide accurate feedback at every step of the pipeline, which improves the readability of the system. The hierarchical

```
Installing core arduino:mbed_giga@4.4.1...
Installing Validator core arduino:mbed_giga@4.3.1...
Matching detected boards with configuration...
- Validator board found: Serial 0031001D3033510334323437 -> Port /dev/ttyACM1, Detected FQBN arduino:mbed_giga:giga
- DUT board auto-assigned: Serial 30335119002D0034 -> Port /dev/ttyACM0, Detected FQBN arduino:zephyr:giga
- DUT board updated: Serial 30335119002D0034 -> Port /dev/ttyACM0, Config FQBN arduino:mbed_giga:giga
- Validator board updated: Serial 0031001D3033510334323437 -> Port /dev/ttyACM1, Config FQBN arduino:mbed_giga:giga
Successfully matched 2 board(s) with configuration
```

Figure 6.2: Log of a successful match of the hardware environment with the configuration. In the picture, it is noticeable how the DUT's FQBN changes from `zephyr:giga` to `mbed_giga:giga`, confirming the configuration was matched successfully.

organization of the results file allows in-depth inspection of each unit test outcome while still providing a way to gain a general overview. However, it shares the same concerns as the configuration file: as more test suites are added, it could become cumbersome to navigate. Hence, a more practical visualization method would improve scalability.

Although the current implementation of the framework is capable of detecting failures and abort gracefully, its robustness can be strengthened. Failure tolerance mechanism can be introduced, such as through retries and timeouts, along with failure recovery systems that would allow the framework to carry on with execution despite some failures, preserving partial results. As a practical example, if a developer mistakenly included the CAN suite in a pipeline run on the Zephyr core (which does not support CAN yet), failure in the compilation of the test sketches would cause the orchestrator to abort, wasting time and losing all previously gathered results. A more robust implementation would preserve earlier results, report the failure, and continue with the rest of the suites.

Unit tests

The adopted unit tests strategy proved effective in validating the communication protocols, producing consistent results for both successful and failing cases. To verify correctness, cross-checks were carried out using the STEMlab 125-14, comparing the tests' outcomes with the observed signals. This confirmed that the test results actually reflected the system behavior.

To highlight this, a practical case that occurred during the last stages of development of the framework is presented. When updating from Zephyr v0.3.1 to v0.3.2, the pipeline consistently failed for three distinct reasons:

1. the inability to put the board in bootloader mode before uploading a sketch;
2. the DUT responding to its first incoming UART communication (consisting of a single byte), with a 257-bytes string, and occasionally with random bytes;

```
Command: Send 1 bytes
Command: Send 2 bytes
Command: Send 3 bytes
Command: Send 4 bytes
Command: Send 5 bytes
Command: Send 6 bytes
Command: Send 7 bytes
Command: Send 8 bytes
Command: Send 9 bytes
Command: Send 10 bytes
Command: Send 11 bytes
        Response: LESS_BYTES_RECEIVED:0; RESULT:KO

Command: Send 12 bytes
Command: Send 13 bytes
Command: Send 14 bytes
Command: Send 15 bytes
Command: Send 16 bytes
Command: Send 17 bytes
Command: Send 18 bytes
Command: Send 19 bytes
Command: Send 20 bytes
Command: Send 21 bytes
Command: Send 22 bytes
        Response: LESS_BYTES_RECEIVED:0; RESULT:KO

Command: Send 23 bytes
Command: Send 24 bytes
        Response: LESS_BYTES_RECEIVED:0; RESULT:KO

Command: Send 25 bytes
        Response: LESS_BYTES_RECEIVED:0; RESULT:KO

Command: Send 26 bytes
Command: Send 27 bytes
Command: Send 28 bytes
Command: Send 29 bytes
Command: Send 30 bytes
Command: Send 31 bytes
Command: Send 32 bytes
***Test results:***
4 tests failed
```

Figure 6.3: Results log of a UART test. Some failures were induced to demonstrate how failure logs report the error occurred.

3. the I2C protocol not working at all.

Figure 6.5 shows a screenshot of the logic analyzer demonstrating that the DUT did not accept I2C communications, while Figure 6.5 shows the output of the UART unit test sketches run in debug mode. When contacted, the Zephyr-core developers confirmed that they had intentionally disabled the bootloader command, that they had added debugging strings to the UART port, and that the I2C issue was indeed caused by a bug that was being investigated.

These findings demonstrate the functionality of the tests, which are capable of detecting and reporting protocol issues, while the orchestrator can handle them gracefully, without crashing or halting the pipeline.

Regarding their usability, the adoption of the ATU library ensures the unit tests

```
• ◆◆[8D[J[00:00:01.208,000] [0m<inf> usb_cdc_acm: Device suspended[0m
• [1;32muart:~$ [m[8D[J[00:00:01.557,000] [0m<inf> usb_cdc_acm: Device configured[0m
• [1;32muart:~$ [mCIAO
```

Figure 6.4: Screenshot of the output of the UART unit tests for the. The Validator sends the string `CIAO`, but the DUT prints debugging strings along with the received message.

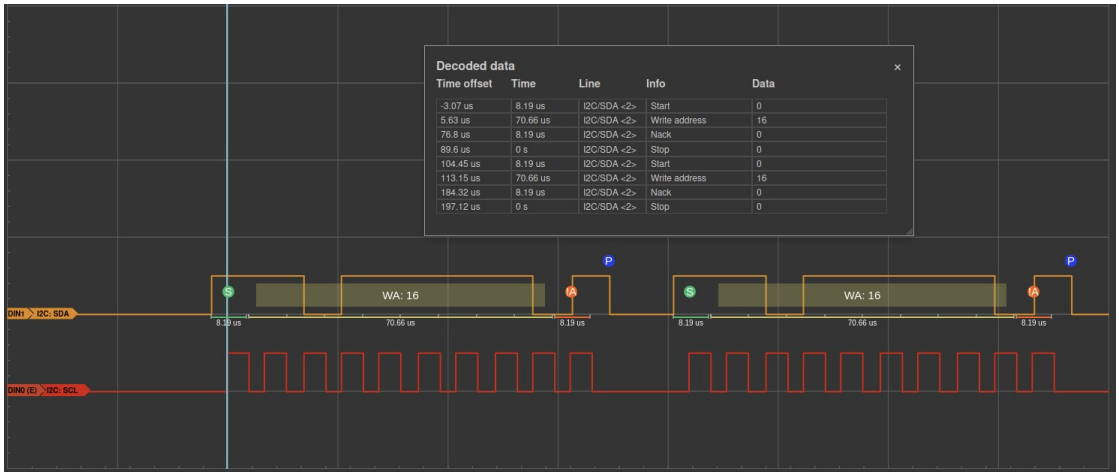

Figure 6.5: Screenshot of the logic analysis of the STEMlab 125-14 of an I2C communication, showing that to each byte sent by the Validator, the DUT would not acknowledge the transmission.

are not limited to run within the framework. Instead, they can also be executed independently, for example via the Arduino IDE, while maintaining their full functionality. In addition, they can be run in debug mode to produce more detailed logs, useful for tracking and diagnosing issues. This flexibility gives developers the ability to test their code on-the-fly without the overhead of running the entire pipeline.

From a maintainability perspective, the mechanism with which they interact with the rest of the pipeline allows to freely modify them as long as they produce the output required from the orchestrator. While the current string-based design could be replaced with a more robust code-based interface, it does not hinder maintainability or scalability.

Integration with GitHub Action

In terms of functionality, the workflow executed successfully, showing the complete output log produced by the framework, uploading the results file as artifact (Figure

6.6), and correctly categorizing each run as either passed or failed (Figure 6.7 and Figure 6.8).

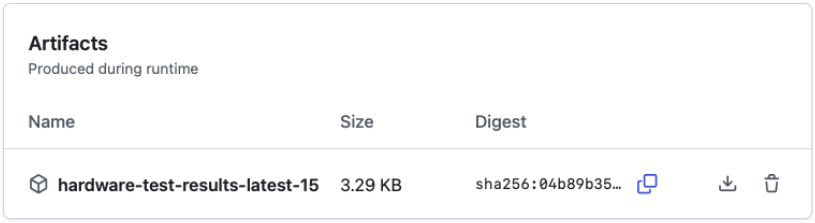

Figure 6.6: Screenshot of the results file uploaded as artifact.

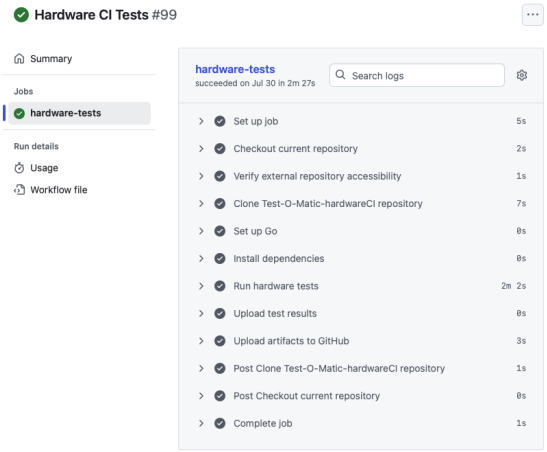

Figure 6.7: Screenshot of a successful GitHub Action run.

As for usability, the workflow could be triggered both automatically upon pushes and pull requests, and manually via the GitHub UI. The manual trigger also makes it possible to select the repository hosting the framework and which test suites to run, thus offering high flexibility (Figure 6.9). When building the workflow, the assumption was made that the master branch of the repository would also be the stable one, but future versions could adopt tagging as an additional precaution.

Its maintainability is inherent to the workflow definition mechanism used by GitHub Action: the YAML file where the job is defined allows easy customization and tracking in the VCS. However, as discussed in the previous chapter, concurrency remains an open problem, as only one hardware setup is currently available, and hence simultaneous requests have to be queued up. The system could scale out by adding more hardware setups and a proxy mechanism capable of dynamically distributing requests.

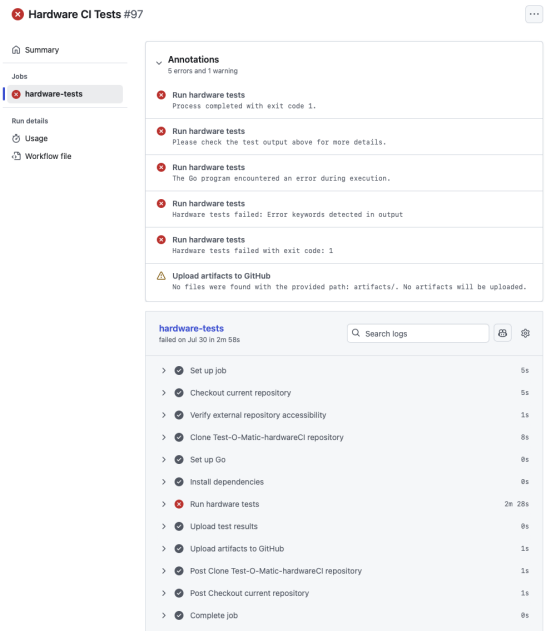

Figure 6.8: Screenshot of a GitHub Action run failed due to an inducted crash.

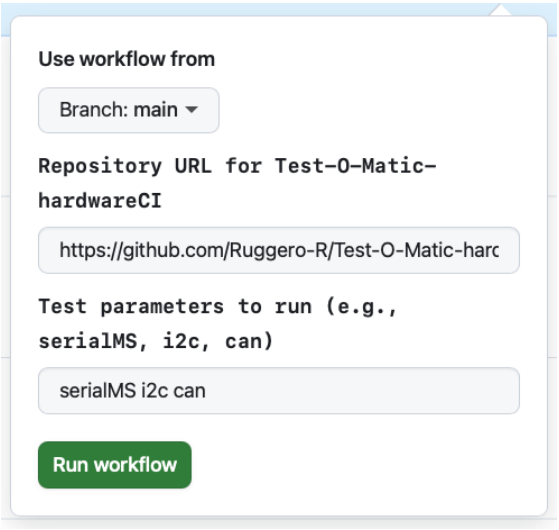

Figure 6.9: Screenshot of the dialog window that lets manually trigger the GitHub Action workflow run.

6.1.3 Execution time

Finally, execution time will be briefly analyzed, as it is an important enabler for an effective CI pipeline. Long-running tests can significantly inhibit the benefits of

Continuous Integration, as developers lose the ability to receive rapid feedback [10].

Both pipelines took approximately 40 minutes to complete. Specifically, the one executed on the Mbed repository took 42 minutes and 32 seconds, while the Zephyr-based one took 38 minutes and 18 seconds, a difference that can be mainly attributed to the absence of the CAN test suite.

These measurements do not include the time required to update a core to a new version, since as a minor impact on the total execution time of an extended pipeline (for example, it takes between 30 to 45 seconds to update or downgrade the Zephyr core, depending on the power of the machine). On the contrary, a clean installation of an entirely new core may require several minutes, but that is a rare operation performed when configuring a new machine. Therefore, it has been excluded from the analysis.

In both pipelines, a significant portion of the time was spent compiling and uploading the sketches. Precisely, on the Raspberry Pi, compiling a single sketch takes around 30 seconds and uploading about 15. As each test case comprises a Validator and a DUT sketch, these operations are performed 34 times in total. Consequently, out of the 40 minutes taken by the entire pipeline, almost 26 are dedicated to compilation and upload, which is approximately 64% of the entire execution time.

This clearly suggests that reducing the time of these steps would greatly benefit the entire pipeline. As mentioned in section 4.4, an attempt was made to parallelize the compilation and upload of the sketches, which would notably reduce execution time. However, that alone would not be enough, since running several tests on a single configuration remains a limiting factor. Hence, new approaches to achieve scalability should be explored.

Interestingly, there is almost no difference between the Zephyr- and Mbed-based pipelines. This is unexpected, as Zephyr's LLEXT mechanism should in principle provide a clear advantage. Initially, it was thought that the Raspberry Pi was acting as a bottleneck, preventing Zephyr from working properly. For this reason, additional measurements were conducted on a laptop[1] running Ubuntu. The results revealed that, although the compilation was nearly three times faster, the ratio between the Mbed and Zephyr remained essentially unchanged. Mbed took on average just a couple of seconds longer than Zephyr, despite producing significantly larger binaries (e.g., for the I2C DUT sketch, approximately 121 kB for Mbed and only 3 kB for Zephyr). Also the upload step was behaving strangely: when the board was first placed in bootloader mode before trying to upload the sketch,

[1]The laptop features 8 GB of RAM and an 11th generation Intel i7 processor. Hence, no performance-related limitations should be present.

Mbed and Zephyr required roughly 14 seconds to upload. However, if the board was not put in bootloader mode, Mbed could reduce that time to only 5 seconds. This scenario could not be replicated with Zephyr, as uploads would fail when the board was not already in bootloader mode due to DFU detection issues.

Table 6.3 shows a summary of the analysis on execution time of the pipeline (note that all results are approximated to the nearest second for readability purposes).

Platform	**Core**	**Compilation time** [s/sketch]	**Upload time** [s/upload]
Raspberry Pi	Mbed OS	30	15
	Zephyr		
Laptop	Mbed OS	9	14 / 5
	Zephyr	7	14 / fail

Table 6.3: Summary of pipeline execution times on Raspberry Pi and laptop. Upload times on the laptop report the distinction regarding the activation of bootloader mode.

This unexpected phenomenon raises many questions about its causes and should be further investigated.

Summarizing the qualitative assessment, the system proved functional with respect to its intended purpose of providing a CI pipeline for the validation of communication protocols. Both usability and maintainability emerged as strong points: the current design lays a strong foundation on which it is possible to customize and extend the entire pipeline for broader scenarios. Regarding performance, execution time analysis revealed that over two-thirds of execution time is spent compiling and uploading sketches, highlighting the importance of reducing this overhead. The main architectural limitations consist of the system's scalability and fault tolerance, which need to be improved in future versions to facilitate its adoption. However, it should be possible to address these flaws without needing a major refactor.

Chapter 7

Conclusion

This project contributes to demonstrating how the challenges that typically hinder the adoption of continuous practices within embedded systems can be mitigated through the development of a proof-of-concept Continuous Integration pipeline.

The design and implementation of a modular testing framework with Hardware-in-the-Loop constitute the main contributions of this work. The framework enables the validation of communication protocols and seamlessly integrates with GitHub Actions, extending the Arduino's continuous process with true hardware testing. Its key properties are modularity and flexibility, which translate into several benefits:

- **maintainability**: developers can extend the system by adding new test suites and modifying existing ones with minimal effort, allowing it to evolve over time;
- **openness**: the framework can be adapted to other contexts other than the Zephyr transition, extending its support to cover other cores;
- **flexibility**: the system is not limited to run on the CI pipeline, but it can be used by developers locally on their machines to conduct accurate and targeted tests on specific functionalities.

These properties contribute to achieve effectiveness and usability, which are fundamental to promote the framework's adoption.

Nevertheless, some limitations emerged:

- **scalability**: concurrent executions are not yet possible, which is a problem that needs to be addressed before integration into production workflow;
- **fault tolerance**: discarding meaningful data when partial failures occur is costly and should be avoided;

- **execution time**: the duration of the pipeline should be reduced to speed up the overall process and allowing for more rapid feedback.

Moreover, integration with previous work remains an open issue that must be carefully investigated to find a suitable solution that balances usability and maintainability.

Before expanding the framework's capabilities, resolving these issues should be prioritized in order to strengthen its foundations. Importantly, these improvements should not require any major redesign of the system, but can be integrated in its current structure, benefiting from the architecture's modularity.

The experience gained during the development of this project shows the difficulty in achieving reliability when hardware is involved. The challenges posed by HIL testing must be addressed with different strategies, ranging from custom physical devices such as Ruché and Taurasi, to software techniques that ensure accurate validation at every stage of the pipeline. At the same time, this work also showed that reliable hardware testing does not necessarily require expensive infrastructure, but rather a careful design of the underlying architecture.

In conclusion, this thesis represents a concrete step towards bridging continuous practices and embedded systems. It shows that applying common software design principles, such as modularity, openness, and maintainability, can make hardware testing practical and scalable.

Bibliography

[1] Sten Pittet. *Continuous Integration vs. delivery vs. deployment*. Accessed 18-09-2025. URL: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment#:~:text=This%20means%20that%20on%20top%20of%20automated (cit. on pp. 4, 5).

[2] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. «CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study». In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2021 (cit. on pp. 4, 5, 9).

[3] E Naresh, S V N Murthy, N. Sreenivasa, Seshaiah Merikapudi, and C R Rakhi Krishna. «Continuous Integration, Testing Deployment and Delivery in Devops». In: *2024 International Conference on Knowledge Engineering and Communication Systems (ICKECS)*. 2024 (cit. on pp. 4, 5).

[4] JetBrains. *The State of Developer Ecosystem 2023*. Accessed 18-0-2025. URL: https://www.jetbrains.com/lp/devecosystem-2023/ (cit. on p. 4).

[5] C. Serrano, M. Betz, L. Doolittle, S. Paiagua, V. K. Vytla, and LBNL Berkeley, USA. «Hardware-In-the-Loop testing of accelerator firmware». In: *17th International Conference on Accelerator and Large Experimental Physics Control Systems*. 2019 (cit. on pp. 5, 8, 10, 11).

[6] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. «Version Control System: A Review». In: *Procedia Computer Science* (2018) (cit. on pp. 6, 7).

[7] GitLab. *What is version control?* Accessed 21-09-2025. URL: https://about.gitlab.com/topics/version-control/ (cit. on p. 6).

[8] Atlassian. Accessed 21-09-2025. URL: https://www.atlassian.com/git/tutorials/what-is-git (cit. on p. 7).

[9] Trupti Chavan Gayatri Makrand Ghodke. «An Overview of Git». In: *International Journal of Scientific Research in Modern Science and Technology* (2024) (cit. on p. 7).

[10] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. «Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices». In: *IEEE Access* (2017) (cit. on pp. 8, 9, 59).

[11] Han Fu, Sigrid Eldh, Kristian Wiklund, Andreas Ermedahl, and Cyrille Artho. «Prevalence of continuous integration failures in industrial systems with hardware-in-the-loop testing». In: *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2022 (cit. on pp. 8, 9, 11).

[12] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison‡ Wesley Professional, 2009 (cit. on p. 8).

[13] AWS. *Practicing Continuous Integration and Continuous Delivery on AWS*. Tech. rep. Amazon Web Services, 2024 (cit. on p. 8).

[14] Mabl. *The State of Testing in DevOps 2021*. 2021 (cit. on p. 9).

[15] Charanjot Singh, Nikita Seth Gaba, Manjot Kaur, and Bhavleen Kaur. «Comparison of Different CI/CD Tools Integrated with Cloud Platform». In: *2019 9th International Conference on Cloud Computing, Data Science and Engineering (Confluence)*. 2019 (cit. on p. 10).

[16] Pooya Rostami Mazrae, Tom Mens, Mehdi Golzadeh, and Alexandre Decan. «On the usage, co-usage and migration of CI/CD tools: A qualitative analysis». In: *Empirical Software Engineering* (2023) (cit. on p. 10).

[17] Christian Dufour, Zareh Soghomonian, and Wei Li. «Hardware-in-the-Loop Testing of Modern On-Board Power Systems Using Digital Twins». In: *2018 International Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM)*. 2018 (cit. on p. 10).

[18] Jim A. Ledin. *Embedded Systems Programming*. 1999 (cit. on p. 10).

[19] ERNI – Swiss Software Engineering. *Why you should consider connecting your continuous integration pipeline to your hardware*. `https://www.bett erask.erni/why-you-should-consider-connecting-your-continuous-integration-pipeline-to-your-hardware/`. Accessed 19-09-2025. 2022 (cit. on pp. 10, 11).

[20] K. C. Wang. «Embedded and Real-Time Operating Systems». In: Springer International Publishing, 2023. Chap. Embedded Real-Time Operating Systems (cit. on pp. 13, 14).

[21] Walter Cedeño and Phillip A. Laplante. «An Overview of Real-Time Operating Systems». In: *JALA: Journal of the Association for Laboratory Automation* (2007) (cit. on p. 14).

[22] The Zephyr Project Contributors. *About the Zephyr Project*. Accessed 23-09-2025. URL: https://www.zephyrproject.org/learn-about/ (cit. on pp. 15, 16).

[23] The Zephyr Project Contributors. *Zephyr Project Documentation*. Release 4.1.0 (cit. on pp. 15, 16, 18).

[24] Arduino Team. *The end of Mbed marks a new beginning for Arduino*. Accessed 24-09-2025. URL: https://blog.arduino.cc/2024/07/24/the-end-of-mbed-marks-a-new-beginning-for-arduino/ (cit. on p. 18).

[25] Arduino Team. *Introducing Arduino cores with ZephyrOS (beta): take your embedded development to the next leve*. Accessed 23-09-2025. URL: https://blog.arduino.cc/2024/12/05/introducing-arduino-cores-with-zephyros-beta-take-your-embedded-development-to-the-next-level/ (cit. on p. 18).

[26] Arduino Team. *Arduino Core for Zephyr*. Accessed 23-09-2025. URL: https://github.com/arduino/ArduinoCore-zephyr/blob/main/README.md (cit. on p. 19).

[27] Arduino Team. *Updated Arduino cores with ZephyrOS (beta)*. Accessed 23-09-2025. URL: https://blog.arduino.cc/2025/08/06/updated-arduino-cores-with-zephyros-beta/ (cit. on p. 19).

[28] Arduino Team. *Arduino CLI*. Accessed 24-09-2025. URL: https://docs.arduino.cc/arduino-cli/ (cit. on p. 25).

[29] GitHub. *Understanding GitHub Actions*. Accessed 24-09-2025. URL: https://docs.github.com/en/actions/get-started/understand-github-actions (cit. on p. 25).