

# POLITECNICO DI TORINO

Master's Degree in Embedded Systems



Master's Degree Thesis

## Continuous Integration and Unit Testing with Hardware-in-the-Loop for Enhancing Embedded Software Quality

Supervisors:

Prof. Alessandro Savino

Prof. Stefano Di Carlo

Eng. Arturo Guadalupi

Candidate:

Amirhossein Mohtashami

Academic Year 2024/2025



# Summary

This thesis presents the development of an automated test framework for embedded systems, combining Continuous Integration (CI) principles with real-time validation using laboratory instruments. The goal was to reduce manual effort and make firmware testing more repeatable and reliable, especially for Arduino-based projects.

To build the system, a custom firmware was created for the Arduino GIGA R1 WiFi board. The firmware could receive ATU-style commands over serial and perform actions such as generating PWM signals, writing analog voltages, and switching pins. On the host PC, a Go-based orchestration tool was developed. It compiled and uploaded the firmware using Arduino CLI, communicated with the board via serial, and also controlled external instruments over LAN using SCPI.

Dedicated Go libraries were written to support three SCPI-based instruments used in the lab: the Rigol DP832 power supply, the Rigol DS1054Z oscilloscope, and the Keithley DMM6500 multimeter. Each library mapped the instrument's command set into modular Go functions, allowing automated control of voltage levels, waveform capture, and current/voltage measurements.

Before putting everything together, standalone test programs were written for each instrument to verify SCPI communication and basic control. These included a voltage ramp and current logging test for the power supply, waveform trigger configuration for the oscilloscope, and precision current measurement with the DMM.

In the final integrated test workflow, the system was able to:

- Power the DUT using the Rigol DP832 and measure current draw.
- Send commands to the Arduino to generate specific signals.
- Use the Rigol DS1054Z to capture and save screenshots of PWM waveforms.
- Measure analog output voltage using the Keithley DMM6500.
- Save all results to timestamped JSON files and organize screenshots and logs automatically.

This setup allowed full automation of the test flow. Measurements could be repeated reliably and saved for traceability. Screenshots and structured logs helped validate that the firmware was working as expected.

Although some parts—such as the integration with GitHub Actions—were not completed during this thesis, they were planned as natural extensions. In particular, GitHub Actions was chosen because most of the Arduino ecosystem already relies on GitHub for development and collaboration. The system developed here serves as a foundational step toward building a complete Continuous Integration (CI) workflow for Arduino projects, specifically addressing the hardware validation phase. By automating lab-based measurements and interfacing with test instruments, this work establishes the basis for future CI pipelines with Hardware-in-the-Loop (HIL), and can be adapted for other boards and use cases as the CI infrastructure expands.

# Acknowledgements

I would like to sincerely thank Prof. Alessandro Savino for his his unceasing help and encouragement in the realization of this thesis. His expertise and availability were crucial to designing the project and making sure I was focusing on the right objectives.

I am also grateful to Prof. Stefano Di Carlo, from whom I had the opportunity to learn a great deal from taking courses. His teachings had a meaningful impact on my understanding of embedded systems.

I would like to extend special thanks to Arturo Guadalupi and Danilo Leo from Arduino srl for supervising my work during the project. Their expertise, trustworthiness, and accessibility not only made this fruitful but also very enriching personally and professionally.

Finally, I thank my family and friends for always being so supportive and encouraging.



# Table of Contents

<b>List of Tables</b>	IX
<b>List of Figures</b>	X
<b>Acronyms</b>	XIII
<b>1 Introduction</b>	1
1.1 Motivations . . . . .	1
1.2 Problem Statement . . . . .	2
1.3 Objectives . . . . .	3
1.4 Thesis Structure . . . . .	4
<b>2 Continuous Integration in Embedded Systems: Concepts, Tools, and Challenges</b>	5
2.1 Evolution of CI: From Manual Merges to Automated Pipelines . . .	5
2.1.1 Early CI Practices: Nightly Builds and Static Analysis . . .	6
2.1.2 Modern CI/CD Workflows and Tools . . . . .	7
2.1.3 CI/CD Toolchains: Jenkins, GitLab CI, GitHub Actions . .	9
2.1.4 Benefits and Best Practices of Continuous Integration . . .	12
2.2 Challenges of Applying CI to Embedded Systems . . . . .	14
2.2.1 Real-Time Execution and Hardware Dependencies . . . . .	14
2.2.2 Cross-compilation and Flashing Firmware . . . . .	16
2.2.3 Limited Debugging and Output Capabilities . . . . .	19
2.2.4 Timing, Power, and Environmental Constraints . . . . .	21
2.2.5 CI Bottlenecks in Production-Grade Embedded Workflows .	22
2.3 Practical Techniques and Architectures for Embedded CI . . . . .	23
2.3.1 Hardware-in-the-Loop (HIL) Testing . . . . .	24
2.3.2 Virtual Platforms and Emulation . . . . .	27
2.3.3 Automated Instrumentation (Oscilloscopes, DMMs) . . . . .	28
2.3.4 GitHub Actions: Resource Usage and CI/CD Tool Utilization	29
2.4 Summary and Outlook . . . . .	30

2.4.1	Key Takeaways from Existing CI Tools and Techniques . . .	31
2.4.2	Justification for the Thesis Implementation Strategy . . . . .	32
2.4.3	Planned Use of GitHub Actions in the Arduino Ecosystem . .	32
<b>3</b>	<b>Methods Developed in the Thesis</b>	<b>33</b>
3.1	System Architecture Overview . . . . .	33
3.2	Arduino Firmware and ATU Command Design . . . . .	35
3.2.1	GIGA Board Hardware Features and Role . . . . .	35
3.2.2	Command Structure and Syntax . . . . .	38
3.2.3	Firmware Logic and Command Parser . . . . .	39
3.3	Host-Side Automation Logic in Go . . . . .	41
3.3.1	Overview of the Go Orchestration Program . . . . .	42
3.3.2	Sketch Compilation and Upload (Arduino CLI) . . . . .	44
3.3.3	Serial Connection Management and ATU Communication .	45
3.3.4	SCPI-Based Instrument Control Libraries in Go . . . . .	46
3.3.5	Screenshot Capture and JSON Result Saving . . . . .	61
3.4	Example Execution Flow and File Structure . . . . .	64
3.4.1	Execution Flow Overview . . . . .	64
3.4.2	Folder and File Organization . . . . .	65
3.4.3	File Naming Conventions . . . . .	66
3.5	Test Case Examples and Use Scenarios . . . . .	67
3.5.1	PWM Signal Validation . . . . .	67
3.5.2	Analog Output Verification . . . . .	67
<b>4</b>	<b>Implementation Challenges and Results</b>	<b>68</b>
4.1	Overview of the Development and Debugging Process . . . . .	68
4.2	Hardware Challenges . . . . .	70
4.2.1	USB and Manual Reset Issues . . . . .	70
4.2.2	Serial Communication Timing . . . . .	70
4.2.3	Oscilloscope Trigger Instability . . . . .	70
4.3	Software and Integration Issues . . . . .	71
4.3.1	Arduino CLI and Upload Failures . . . . .	71
4.3.2	SCPI Communication Bugs . . . . .	71
4.3.3	JSON and Data Handling Errors . . . . .	72
4.4	System Limitations . . . . .	72
4.4.1	Manual Intervention for Device Reset . . . . .	72
4.4.2	Lack of GitHub Actions Integration . . . . .	73
4.4.3	Single-Threaded Execution . . . . .	73
4.4.4	Static Test Scripts and Configurations . . . . .	73
4.4.5	Limited Feedback on Failures . . . . .	73
4.4.6	Instrument Dependency and Lab Setup Constraints . . . . .	73



4.4.7	Firmware-Level Safety Checks Not Enforced . . . . .	74
4.5	Summary of Results and Observations . . . . .	74
4.5.1	Functional Achievements . . . . .	74
4.5.2	Observed Stability and Accuracy . . . . .	74
4.5.3	Areas for Improvement . . . . .	75
4.5.4	Final Remarks . . . . .	75
<b>5</b>	<b>Conclusion and Future Work</b>	<b>76</b>
5.1	Summary of Contributions . . . . .	76
5.2	Evaluation of the CI Framework . . . . .	77
5.2.1	Reliability and Stability . . . . .	77
5.2.2	Repeatability and Output Consistency . . . . .	77
5.2.3	Modularity and Extensibility . . . . .	78
5.2.4	Integration Readiness . . . . .	78
5.2.5	Limitations . . . . .	78
5.3	Lessons Learned and Research Reflections . . . . .	78
5.3.1	Practical Debugging is Inevitable . . . . .	79
5.3.2	Hardware Adds Complexity to CI . . . . .	79
5.3.3	Abstraction Improves Maintainability . . . . .	79
5.3.4	Documentation is Part of the System . . . . .	79
5.3.5	CI/CD Adoption in Embedded Systems is Growing . . . . .	79
5.3.6	Human Feedback is Still Valuable . . . . .	80
5.4	Suggestions for Future Development . . . . .	80
5.4.1	Full CI/CD Pipeline Integration . . . . .	80
5.4.2	Dynamic Test Configuration . . . . .	80
5.4.3	Concurrency and Performance Optimization . . . . .	81
5.4.4	Error Classification and Logging Enhancements . . . . .	81
5.4.5	Multi-Board and Multi-Instrument Support . . . . .	81
5.4.6	Integration with Visualization and Dashboard Tools . . . . .	81
5.4.7	Hardware Abstraction and Safety Checks . . . . .	82
5.5	Final Remarks . . . . .	82
	<b>Bibliography</b>	<b>83</b>

# List of Tables

2.1	Comparison of Jenkins, GitLab CI, and GitHub Actions for CI/CD in embedded development workflows. . . . .	11
2.2	Comparison of CI/CD tools used in embedded system workflows . .	18
2.3	Execution Times of DIO Software Component Tests [33] . . . . .	26
2.4	GitHub Actions optimization strategies and their impact [36]. . . .	30

# List of Figures

2.1	Early CI overview such as nightly builds, static analysis, and unit testing, adjusted from [4]. . . . .	6
2.2	Architectural overview of a CI/CD pipeline integrating Gitea and Jenkins [7]. . . . .	8
2.3	Conceptual model of a state-of-the-art CI/CD pipeline showing the key components and stakeholder levels in software delivery [15]. . .	13
2.4	CI/CD flow for a software environment showing automated testing across multiple levels, from unit to system-level validation [22]. . . .	15
2.5	CI/CD flow for hardware environments, involving deployment and testing directly on target hardware platforms [22]. . . . .	16
2.6	CI/CD build flow integrating FPGA design and cross-compilation .	17
2.7	Modular JTAG Debug Architecture with multiple IO Clients [30]. .	20
2.8	Multi-master IO Client accessing FPI Bus for debug communication [30]. . . . .	21
2.9	Workflow of Relative Timing Analysis (ReTA) [32]. . . . .	22
2.10	Test setup for manual tests [33] . . . . .	25
2.11	Hardware/software integration test concept based on host-executed test scripts [33] . . . . .	25
2.12	Integration of virtual platforms with environment models for embedded CI [34] . . . . .	28
3.1	High-level architecture of the automated embedded testing setup. .	34
3.2	Arduino GIGA R1 WiFi – Physical appearance . . . . .	36
3.3	Arduino GIGA R1 WiFi – Top view with labeled components and pin layout. . . . .	37
3.4	Terminal output of a full test execution by the Go orchestration program. . . . .	43
3.5	Terminal output showing voltage–current logging with the Rigol DP800 power supply . . . . .	50
3.6	DP800 panel at 10.00V and 0.0574A . . . . .	51
3.7	DP800 panel at 15.00V and 0.0431A . . . . .	51

3.8	Captured waveform on the Rigol DS1054Z after setting EDGE trigger and 2.0V threshold . . . . .	55
3.9	Terminal output showing current measurement of 0.1709 A using the Keithley DMM6500 . . . . .	58
3.10	Current readings on the DMM6500 front panel under different voltage configurations (10V, 8V, and 6V) . . . . .	59
3.11	Captured oscilloscope screenshot showing PWM waveform with 501 Hz frequency and 30% duty cycle. . . . .	62
3.12	Screenshot of a generated JSON result file as displayed in Visual Studio Code. . . . .	63
3.13	Block diagram of the automated test flow from firmware upload to result capture. . . . .	65
3.14	Example of the generated file and folder structure after a test run. .	66



# Acronyms

**ADC**

Analog-to-Digital Converter

**API**

Application Programming Interface

**ATU**

Arduino Test Utility

**AWS**

Amazon Web Services

**BART**

Build and Runtime Triage

**CD**

Continuous Deployment

**CI**

Continuous Integration

**CLI**

Command Line Interface

**CSP**

Cyber-Physical System

**DMM**

Digital Multimeter

**DSL**

Domain-Specific Language

**DUT**

Device Under Test

**ETU**

Embedded Trace Unit

**FPI**

Flexible Peripheral Interface

**FPGA**

Field-Programmable Gate Array

**FQBN**

Fully Qualified Board Name

**Go**

Go Programming Language

**HIL**

Hardware-in-the-Loop

**IP**

Internet Protocol

**JSON**

JavaScript Object Notation

**JTAG**

Joint Test Action Group

**LAN**

Local Area Network

**ML**

Machine Learning

**OTA**

Over-the-Air

**PC**

Personal Computer

**PID**

Product Identifier

**PWM**

Pulse Width Modulation

**ReTA**

Relative Timing Analysis

**SCPI**

Standard Commands for Programmable Instruments

**SDK**

Software Development Kit

**SLA**

Service-Level Agreement

**SoC**

System-on-Chip

**SUT**

System Under Test

**TDD**

Test-Driven Development

**TCP**

Transmission Control Protocol

**TeSSLa**

Temporal Stream-based Specification Language



**UART**

Universal Asynchronous Receiver-Transmitter

**USB**

Universal Serial Bus

**VID**

Vendor Identifier

**VM**

Virtual Machine

**VP**

Virtual Platform

**WCET**

Worst-Case Execution Time

**YAML**

Yet Another Markup Language

# Chapter 1

## Introduction

### 1.1 Motivations

Embedded systems are used today in many different areas like automotive, industrial automation, medical devices, and even consumer electronics. These systems are usually designed for a specific task, and because of that, they often need to meet strict constraints like power consumption, timing, reliability, and cost [1]. Since each application has different priorities, the design of the system has to be carefully adapted to the context where it will operate.

In some fields, like automotive and aerospace, these constraints are even more critical. Systems must guarantee real-time execution, safety, and long-term stability, even under changing conditions and increased complexity [2]. But in practice, the development and testing process is still very manual. Developers usually have to flash the firmware to the board and measure outputs using external instruments, which can be slow and difficult to repeat.

This becomes a big problem when the firmware needs to be tested often or across different versions. Manual methods introduce delays and errors, especially when testing involves physical signals and hardware behavior. At the same time, the growing demand for reliability, low energy usage, and faster cycles pushes the need for more automated and structured processes [3].

The idea behind this thesis is to apply modern Continuous Integration techniques to embedded firmware validation, with the help of real hardware and laboratory instruments. By automating the compile, upload, and test phases, the goal is to reduce manual steps, improve repeatability, and make testing more efficient and robust for real-world embedded applications.

This thesis is the first structured attempt to introduce Continuous Integration (CI) into the Arduino firmware development cycle, with a focus on real hardware

testing. Prior to this work, no CI framework existed in the Arduino ecosystem for validating embedded firmware against real physical signals using laboratory instruments. By automating test scenarios through hardware-in-the-loop (HIL) techniques and integrating SCPI-controllable instruments, this work lays the foundational infrastructure for future CI/CD workflows at Arduino. It represents a practical shift from isolated manual testing toward systematic, reproducible validation that fits into modern DevOps practices for embedded systems.

## 1.2 Problem Statement

In embedded systems development, validating firmware is still often done manually. This usually involves compiling the code, uploading it to the board using a USB cable or other programmer, and then checking its behavior by monitoring signals with lab instruments like oscilloscopes or multimeters. While this works for small tests, it becomes a problem when the system gets more complex or when the firmware needs to be tested repeatedly in different configurations.

Manual testing is time-consuming and not always consistent. Each time a developer wants to verify something, they need to flash the firmware, connect the instruments, set up the conditions, and read values manually. This makes the whole process harder to repeat and also increases the risk of human error. When the firmware needs to be tested after every small update, this manual process becomes a bottleneck.

Unlike general software projects that rely on Continuous Integration to automatically build and test code, embedded development doesn't usually benefit from the same kind of automation, especially when hardware is involved. Even when some CI is used, the actual behavior of the firmware on the real board is rarely tested. Most automated tests only check that the code compiles or runs in simulation, but they don't validate real electrical signals or physical interactions.

The problem this thesis addresses is how to close that gap. The goal is to build a system where each firmware change can be tested automatically, not just in code but also on the actual hardware, using real instruments. This would allow developers to verify functionality, timing, and signal correctness without needing to be physically present for every test. Without this kind of automation, it's very difficult to scale up testing, catch bugs early, or maintain high reliability in complex embedded projects.

## 1.3 Objectives

The main objective of this thesis is to bring automation into the firmware development and validation process by combining Continuous Integration techniques with real hardware-in-the-loop testing. The idea is to reduce the amount of manual work that goes into compiling, uploading, and verifying firmware on embedded boards, especially when updates are frequent or testing conditions are complex.

The system developed in this work should allow firmware to be compiled automatically, flashed to a target board, and tested using real lab instruments without human interaction. Instead of relying on manual measurements or visual checks, the tests should provide measurable outputs that can be logged, compared, and verified.

To support this goal, the thesis is based on the following specific objectives:

- Develop a robust communication interface between the host PC and the Arduino board using serial protocols for command execution.
- Create modular ATU-based firmware that allows the Arduino to interpret and respond to remote test commands.
- Use Arduino CLI to compile and upload firmware automatically as part of the CI workflow.
- Build SCPI-based libraries in Go to remotely control laboratory instruments including oscilloscopes, power supplies, and digital multimeters.
- Design a centralized orchestration system in Go that coordinates firmware upload, DUT configuration, instrument control, and measurement capture.
- Ensure the system is capable of storing structured test results (e.g., current, voltage, frequency, duty cycle) in machine-readable formats such as JSON.
- Design the architecture to allow future integration with GitHub Actions and other CI tools used within Arduino’s open-source ecosystem.
- Validate the entire flow by executing end-to-end tests on real hardware and confirming correctness via instrument feedback.

The overall aim is to make embedded firmware testing more efficient, repeatable, and scalable by bringing together tools that already exist, but are rarely used in combination for this purpose.

## 1.4 Thesis Structure

This thesis is organized into five chapters, each addressing a specific phase of the project. The structure is designed to lead the reader from the initial motivations and background context to the technical implementation, experimental validation, and forward-looking conclusions.

Chapter 1 introduces the motivation, problem statement, and objectives of the work. It explains the need for automation in embedded-firmware validation and outlines how Continuous Integration (CI) techniques can address longstanding inefficiencies in manual testing processes.

Chapter 2 reviews the state of the art in CI for embedded systems. It begins with an overview of traditional CI practices and modern toolchains, then analyzes the specific technical and organizational challenges of applying CI to hardware-dependent systems. It also explores hardware-in-the-loop testing, virtual platforms, and automated instrumentation using SCPI.

Chapter 3 presents the methods developed in this thesis. It describes the full system architecture, including the ATU firmware on the Arduino GIGA, the ATU-based command interface, and host-side orchestration written in Go. The chapter details how custom SCPI libraries were developed for controlling oscilloscopes, power supplies, and multimeters, and how serial communication was used to automate firmware interaction—laying the foundation for a hardware-in-the-loop CI pipeline.

Chapter 4 discusses implementation challenges and validation results. It chronicles the development and debugging process, highlights hardware and software integration issues, and reports on system stability and accuracy. The chapter also presents real-world test outputs—oscilloscope screenshots and structured JSON measurements—to demonstrate automated firmware testing across multiple scenarios.

Chapter 5 concludes the thesis and outlines future work. It summarizes the key contributions, reflects on lessons learned (timing, reliability, modularity), and proposes enhancements such as full CI/CD integration, dynamic test configuration, parallel execution, and expanded multi-instrument support.

## Chapter 2

# Continuous Integration in Embedded Systems: Concepts, Tools, and Challenges

This chapter gives a technological background for the system implemented later in this very dissertation. It begins with a brief history of CI in general-purpose software, then presents today's popular CI/CD workflows and tools used widely by professionals. After that, it discusses the special limitations and challenges of applying CI to embedded systems — such as dependence on hardware, real-time constraints, and the difficulty of debugging. And finally it evaluates existing solutions for HIL testing or SCPI-based instrumentation now in use. This discussion will help the reader to evaluate technical trade-offs and better understand the design choices made in Chapter 3.

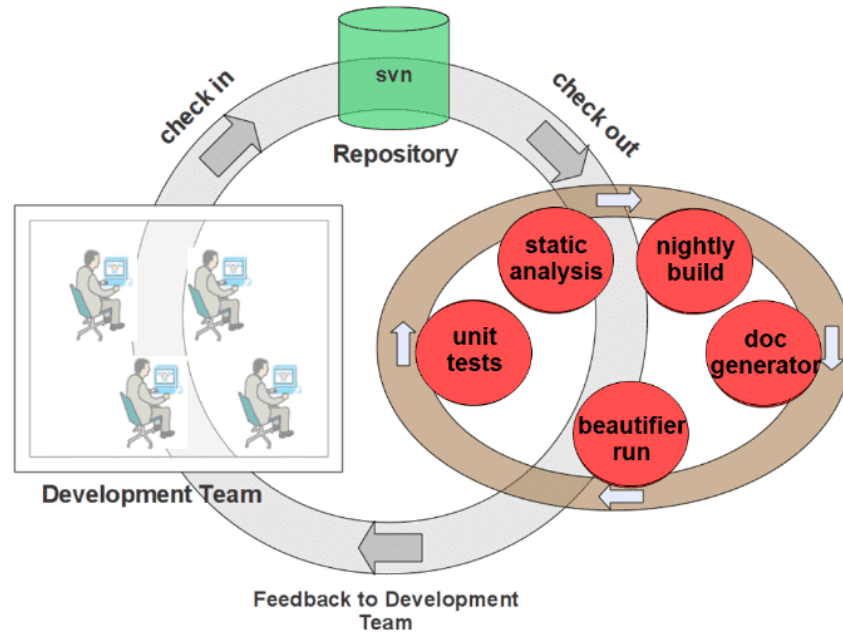
### 2.1 Evolution of CI: From Manual Merges to Automated Pipelines

Continuous integration did not start out as a streamlined, automated process. In the old days of software development, teams often waited until the end of a release cycle to merge their code. Developers worked separately on their code for weeks and then tried to combine everything at once. This typically led to numerous problems such as merge errors, broken builds, and bugs that were hard to track down but difficult to locate.

As software systems became increasingly complex and teams grew in size, this method proved unwieldy and ineffective. One of the first steps toward improvement was the use of a centralized version control system. This allowed developers to make changes more frequently, and also made it possible for what had been altered to be traced back. It was easier to revert unfixable changes in this way when something went wrong [4].

### 2.1.1 Early CI Practices: Nightly Builds and Static Analysis

Another good method was introducing nightly builds. Although no immediate feedback, they still enabled teams to identify their own integration problems in a matter of days. During the nightly builds, some teams also added static code analysis and code formatting tools. All of this contributed to both individual mistake-catching consistency increases [4].



**Figure 2.1:** Early CI overview such as nightly builds, static analysis, and unit testing, adjusted from [4].

Another of the first integrated CI practices was unit testing. It ensured that the code still worked, who after each small thing got changed in each place, sort of an important fail [5]. Even though early CI was not fully automatable, it played

a when destiny called role for modern business processes, reducing defects and increasing collaboration.

But all these things could not be done without proper infrastructure. However, if the build script was not dependable or the test environment inconsistent, sometimes it was CI that delays rather than solves problems. Sometimes, comprehensive technical debt had to be handled by teams because the scripts were outdated or systems were ill-maintained [6].

From these early experiences, more trustworthy and automatic CI tools began to take shape. The next section examines the workflow of CI in the present day and its common limitations for real projects.

### **2.1.2 Modern CI/CD Workflows and Tools**

In modern software engineering, Continuous Integration and Continuous Delivery have become standard practice. In addition to this, they also control processes like build, test deployment so that common frontal developer errors will be reduced. Without the capability of CI/CD, rapid release cycles will be unattainable. CI/CD can associate with events in version control such as commit to ensure that code changes are continually validated and ready for production uses in the case of production use [7].

Popular CI/CD platforms include Jenkins, GitLab CI/CD, and GitHub Actions. These tools allow teams to define pipelines using YAML or declarative syntax and connect various stages such as compilation, testing, and packaging. Depending on the project scale and infrastructure preferences, teams may choose cloud-hosted, self-hosted, or hybrid deployment models for these tools.

In addition to orchestrating a pipeline, containerization is very important for ensuring consistent environments. Docker is widely used to package applications and their dependencies into portable containers. Kubernetes extends this idea to manage container scheduling, scaling, and updates in distributed systems.

Many teams also integrate their pipelines with cloud platforms such as AWS (Amazon Web Services), Google Cloud, or Microsoft Azure. These services provide scalable runners, secure artifact storage, and predefined templates that facilitate deployment of the pipeline [8]. They also greatly ease infrastructure management tasks, thus sparing developers from such boring work to devote all their attention to coding.

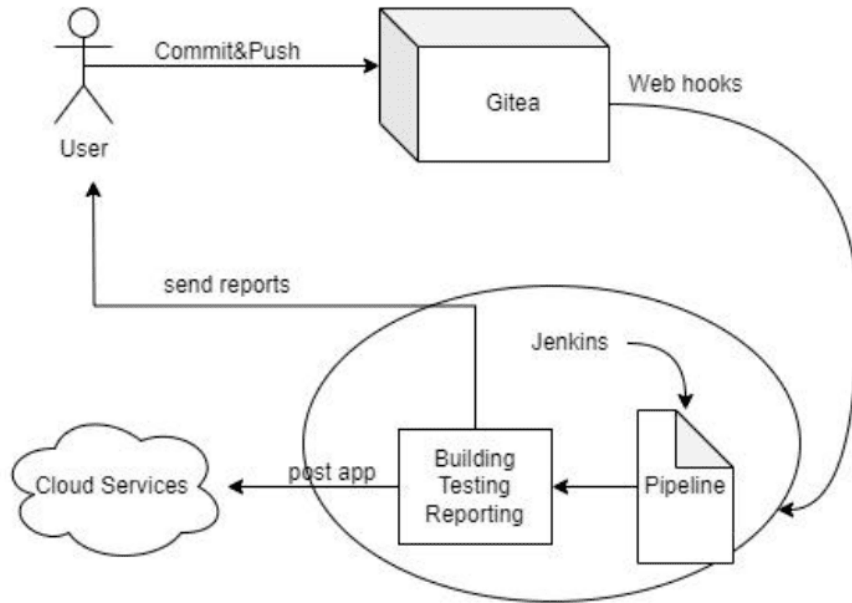
CI/CD adoption provides benefits such as killing two birds with one stone—shorter feedback loops and greater quality—and better collaboration between development and operations teams. It also allows the merging of static code analysis tools, linters, code coverage reports, and vulnerability scanners into automated workflows. These additions help to catch problems early and hold to a greater standard [9].



However, in certain domains, implementing CI/CD is not so straightforward. Legacy applications, monolithic codebases, and embedded systems all bring additional constraints to bear—perhaps needing specific compilers or requiring hardware access during testing [7]. Or they may not be compatible with other tools. These constraints need tailored solutions and may involve hybrid architectures.

A practical example of such a setup is the combination of Gitea and Jenkins. Gitea provides a lightweight self-hosted Git platform, while Jenkins handles the CI/CD orchestration. The interaction follows these steps:

1. Developers push code to private repositories hosted on Gitea.
2. Gitea sends a webhook to Jenkins upon detecting a new commit.
3. Jenkins executes a pipeline with the following stages:
  - Source code compilation and build.
  - Unit and integration tests.
  - Generation of test reports and artifacts.
4. If the pipeline succeeds, Jenkins deploys the output to a target system or embedded device.



**Figure 2.2:** Architectural overview of a CI/CD pipeline integrating Gitea and Jenkins [7].

In summary, modern CI/CD workflows integrate source control, automation servers, containers, and cloud platforms to provide a robust foundation for software delivery. The choice of tools depends on the project context, and embedded development introduces specific constraints that will be addressed in the following sections.

### 2.1.3 CI/CD Toolchains: Jenkins, GitLab CI, GitHub Actions

Continuous Integration and Continuous Deployment (CI/CD) practices have become the cornerstone of most modern development, allowing teams to make software faster, better quality and more reliably. Whether Jenkins, GitHub Actions are among the most frequently used tools for actually working these processes. These have varying emphasis on flexibility and integration but all provide ease of use.

**Jenkins** is a self-hosted automation server with wide capabilities and is well-suited to complex enterprise environments. It includes both scripted and declarative pipelines as well as an extensive library of plugins. However, Jenkins requires many manual configuration steps, particularly when used in embedded systems or hardware-in-the-loop (HIL) scenarios, where custom agents must be deployed and maintained [10].

**GitLab CI** is built directly into GitLab. Instead of managing plugins made by third-parties, you can keep everything in one place, more easily ensuring it all works and is usable. It uses a single YAML file (`.gitlab-ci.yml`) to define jobs, requires less setup than Jenkins and comes equipped with features like security scans, test reports and artifact handling.

For one thing, GitLab CI is not as tightly integrated with the Arduino CLI or hardware automation tools as its rival. Therefore despite the higher rated usability, it receives a mixed ranking in comparison to Jenkins [11]. However, GitLab CI is suitable for the general software engineering project workflow.

**GitHub Actions** has emerged as a preferred option for lightweight and educational workflows, especially for Arduino-based projects. It uses simple YAML files placed under `.github/workflows` and allows automatic triggering of jobs upon events such as push, pull request, or tag creation. Its deep integration with GitHub and native support for community-contributed Actions make it particularly suitable for firmware compilation, testing, and deployment in embedded contexts [12].

**A. CI/CD Pipelines with GitHub Actions:** GitHub Actions offers an end-to-end CI/CD solution integrated directly into the GitHub ecosystem. It can be used to:

- **Build and Test:** Automate sketch compilation using the Arduino CLI and validate functionality across platforms and versions.

- **Deploy:** Automatically upload compiled firmware to a target device, or package it for further testing.

**Benefits:**

- *Efficiency:* Automates time-consuming build and test routines.
- *Consistency:* Ensures repeatable workflows across commits and contributors.
- *Collaboration:* Provides instant feedback via GitHub checks and badges.

**B. GitHub Actions for Testing and Quality Assurance:** GitHub Actions simplifies quality assurance by automating tests and enforcing standards.

**Testing strategies include:**

- *Matrix testing* for evaluating code across multiple hardware targets or configurations.
- *Unit test runners* and static analysis tools.

**Benefits:**

- *Early Detection:* Catch regressions before merge.
- *Maintainability:* Automated checks improve long-term code quality.

**C. GitHub Actions for Security Automation:** Security checks are often integrated into CI pipelines using GitHub Actions.

**Features:**

- Detect vulnerabilities in third-party libraries via **dependabot**.
- Enforce secure coding policies with pre-defined security actions.

**Benefits:**

- *Proactive Defense:* Prevents insecure dependencies from reaching production.
- *Speed:* Reduces the response time to known exploits.

**D. GitHub Workflow for Embedded Projects (Adapted from [12]):**

GitHub Actions supports a common Git-based development flow. The following commands and steps are typically automated through YAML workflows:

1. **Clone the repository:** `git clone https://github.com/user/repo.git`

2. **Create a new branch:** `git checkout -b feature-branch`
3. **Make changes** to the firmware or test scripts.
4. **Stage changes:** `git add .`
5. **Commit with a message:** `git commit -m "Fix: update timing test"`
6. **Push to GitHub:** `git push origin feature-branch`
7. **Create Pull Request** and trigger automated workflows (compilation, test, upload).

**Why GitHub Actions for Arduino-Based Work:** The Github repositories and opened-sourced contributions are an inherent aspect of the Arduino ecosystem. Seamless integration with the Arduino CLI through GitHub Actions, which makes it easy to automate processes such as building, uploading firmware and reporting test results - which can be very important in hardware-in-the-loop orientation test environments.

Tool	Setup and Hosting	Integration and Plugins	Use in Embedded/Arduino	Pros / Cons
<b>Jenkins</b>	Self-hosted; requires manual setup and maintenance	Highly customizable via plugins; supports tools like Docker and SonarQube	Suitable for complex hardware workflows; high setup overhead	Powerful but requires significant effort; ideal for large teams
<b>GitLab CI</b>	Integrated into GitLab; supports both cloud and self-hosted options	Built-in CI/CD, security scanning, version control integration	Common in full-stack DevOps; limited use in Arduino	Complete DevOps suite; steeper learning curve for embedded
<b>GitHub Actions</b>	Native to GitHub; supports cloud and self-hosted runners	YAML-based; integrates with GitHub CLI and Arduino CLI	Best fit for Arduino CI/CD; popular in open-source	Lightweight, developer-friendly; natural workflow for Arduino

**Table 2.1:** Comparison of Jenkins, GitLab CI, and GitHub Actions for CI/CD in embedded development workflows.

To sum up, GitHub Actions is a light and flexible developer-friendly CI/CD environment. In comparison with Jenkins or GitLab CI, it provides the most user-friendly workflow for Arduino development, as it is natively integrated with the GitHub, is free of charge up to a certain tier, and supported by the ecosystem [12].

### 2.1.4 Benefits and Best Practices of Continuous Integration

Continuous Integration (CI) has a great potential for the software process, which can be beneficial for software projects when used in a consistent manner and adapted to project restrictions. In contemporary development, CI now enables developers to detect integration problems earlier, and in the name of productivity, quality and team congeniality. CI leads to faster developer feedback and faster software delivery by automatically validating changes on the baseline [13, 9].

Better quality of the software is the most popularly quoted benefit of CI. Regular integration means that changes are tested as soon as possible, and also in small enough bits that it's much easier for a team to catch regressions before they reach lower-level functions. Automated pipelines that comprise with linters and static analysis tools enforce code of conduct to avoid poor code to be pushed to production [9, 13]. Moreover, CI encourages developers to commit more frequently, making the debugging process faster and more traceable.

And the other big advantage that has is it makes the developer faster. CI reduces the amount of manual testing and deployment work, allowing development teams to focus on feature development and fixing issues [14]. In the long term, teams who val CI have smopoth deploys, and are less distracted on the transitions and join up of engineering and operations.

But industrywide studies suggest that these gains are very context-dependent. In the context of embedded and telecom, the problematic deployment setup of CI is due to legacy systems and the per-customer configuration (done via third party software) [15]. For example, organizations would have to establish a parallel pipeline to cater to traditional release mechanism and new-age CI system leading to increased operational overhead. In such scenarios, the balance between the cost-benefit of CI needs to be weighed very carefully before full implementation.

To maximize the return on CI, several best practices are commonly recommended:

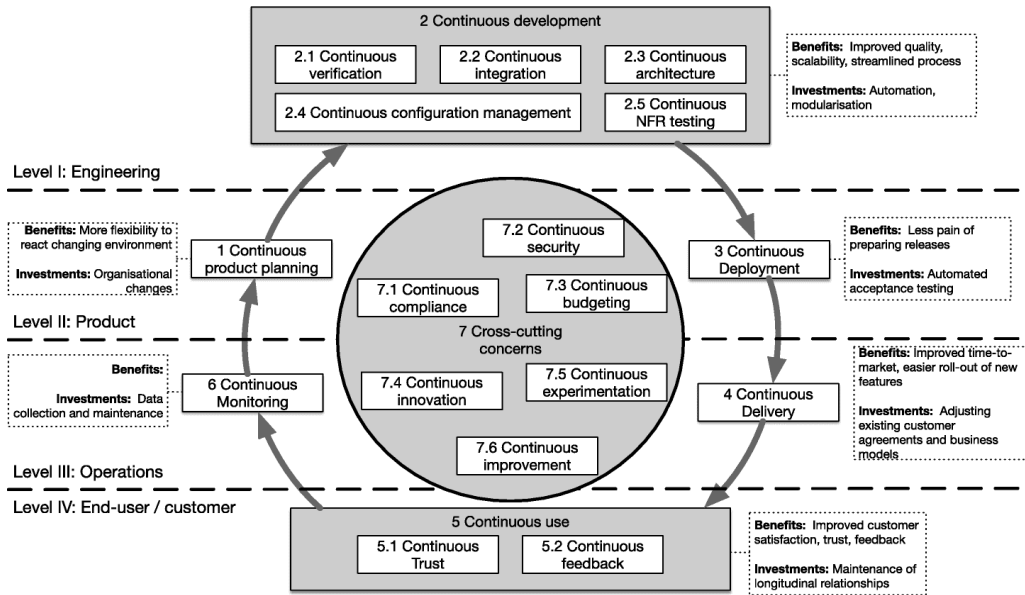
- **Automate tests:** Unit, integration, and static tests should be run with every change to ensure correctness and prevent regressions [9].
- **Integrate frequently:** Teams are encouraged to commit small changes multiple times per day to avoid merge conflicts and increase traceability [13].
- **Monitor build health:** A healthy CI system includes alerts for build failures, detailed logs, and clear feedback to the developers [14].
- **Reduce unnecessary steps:** Redundant scripts or documentation in the delivery process should be eliminated when CI automates those functions [15].

Despite these practices, some barriers remain. Developers, for example, find it challenging to make sense of complex build logs - particularly in large CI pipelines.

BART and build failure classification systems have been presented in order to smooth the bug triage [14]. Other works concentrate on cataloguing and identifying anti-patterns that are associated with the degradation of CI processes, such as not executing tests, infrequent commits, that affect the long-term reliability of the pipeline.

In domains with strong requirements in agreements or governance CI needs to be adaptable rather than rigidly applied. Organizations that operate under strict SLAs or within regulated industry sectors may still have to retain hybrid delivery models, support legacy integration mechanisms, and implement CI only gradually [15].

Overall, the full benefits of CI are to be reaped from the technology or organization only when it can be put into practice. It is only when both the company culture and product infrastructure are ready that CI may grow to fulfil its full potential. Through combining solid engineering practices with the real world constraints that the project faces, teams can develop their CI strategies in a mature manner and so avoid common folly like learning when it's too late.



**Figure 2.3:** Conceptual model of a state-of-the-art CI/CD pipeline showing the key components and stakeholder levels in software delivery [15].

## **2.2 Challenges of Applying CI to Embedded Systems**

While Continuous Integration is extensively used in the context of general purpose software development, its context to the embedded systems world faces several non-trivial issues. In contrast to desktop or cloud software, embedded development is closely related to real hardware, real time constraints and environmental dependencies which make automate testing and repeatable testing difficult[16]. In embedded scenarios, the typical CI workflows are designed for high levels of computational and software abstraction, which are explicitly not guaranteed. Instead, developers need to interact with limited access to target hardware, manually set up test environments for validation purposes, and long feedback loops including Devices under Test (DUTs), signal generators, or measurement tools[17].

Further issues are long cross-compilation times, toolchain incompatibilities, hardware specific build failures which hinder the developers productivity and increase the latencies in terms of defect detection[18]. In life-critical domains, that is, where a software failure can lead to death or injury, e.g., automotive or aerospace, adherence to functional safety standards increases the level of constraints in tool certification, regression coverage, and change control processes[16]. Even today's model-based-design flows struggle with fragmented toolchains and challenges on maintaining their simulation models up to date with deployed firmware [19].

This section presents the technical and process impediments for integrating CI in embedded software development. We'll address them one at a time, with the goal of providing the reader with enough information to evaluate design trade-offs, bootstrap CI approaches, and tune the tools to their specific development scenario.

### **2.2.1 Real-Time Execution and Hardware Dependencies**

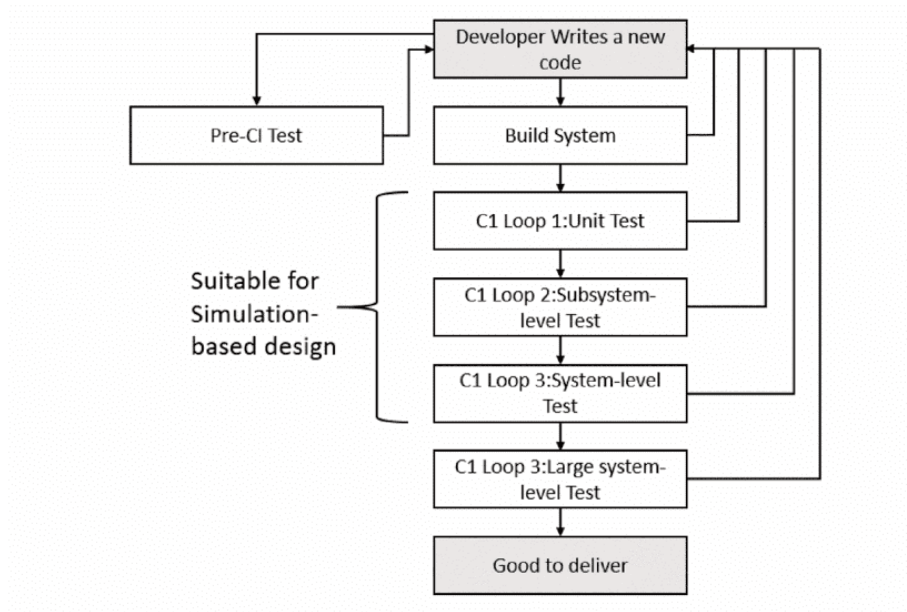
There are known and well-documented benefits of CI practices in recent software engineering deployment models, yet applying those systems to real world, hardware focused embedded systems is not without issues. For an embedded software application, the software must be validated and tested within a hardware partition. This dependency creates bottlenecks on shared resources including DUTs, probes or hardware interfaces that are, for the most part, idle and are not easy to make it available for each CI cycle[17]. Furthermore, real-time requirements add another level of complexity. A great deal of embedded software, for instance, involves software interacting with hardware within relatively tight time limits. It can be challenging to model or reproduce these deterministic operations in a typical CI environment, which ultimately can lead to defects not being detected, or left completely[18, 20].

Long build times and brittle integration processes as well as unavailable hardware

lead to frequent CI pipeline breaks, effectively hampering the fast iteration. Late defect discovery and interrupted flow of development are typical problems that team working in real-time embedded contexts experience [21].

In order to overcome these challenges, some solutions have been suggested by researchers. *virtual platforms* or simulators to decouple the test environment from the hardware where developers can do their early verification without getting blocked on hardware availability [22]. Another approach involves adopting *Hybrid Hardware-in-the-Loop (HIL)* setups, which blend simulation with real hardware validation [23]. Finally, *automated test selection* frameworks help reduce execution load on embedded targets by selecting only the most relevant tests for a given code change [24].

Figure 2.4 shows a common CI/CD process which includes the software ecosystem. Upon successful local development and manual testing, code changes are being committed and are going through a series of automated build stages and testing levels (Unit, Subsystem and full system testing) to verify the correctness and integration [22].

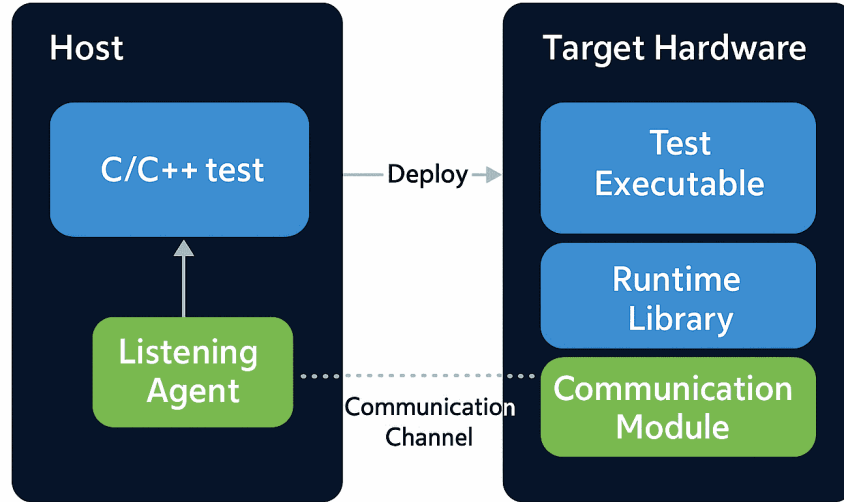


**Figure 2.4:** CI/CD flow for a software environment showing automated testing across multiple levels, from unit to system-level validation [22].

Likewise, Figure 2.5 illustrates the CI/CD flow in the context of hardware.



And here, after I’ve written and pushed that code, I run the test suite directly on the actual target hardware. Unlike a software pipeline, this flow includes flashing firmware, running tests on real boards, and capturing hardware responses. Development boards which are very close to the production target in terms of functionality are frequently exploited to simulate realistic testing contexts[22].



**Figure 2.5:** CI/CD flow for hardware environments, involving deployment and testing directly on target hardware platforms [22].

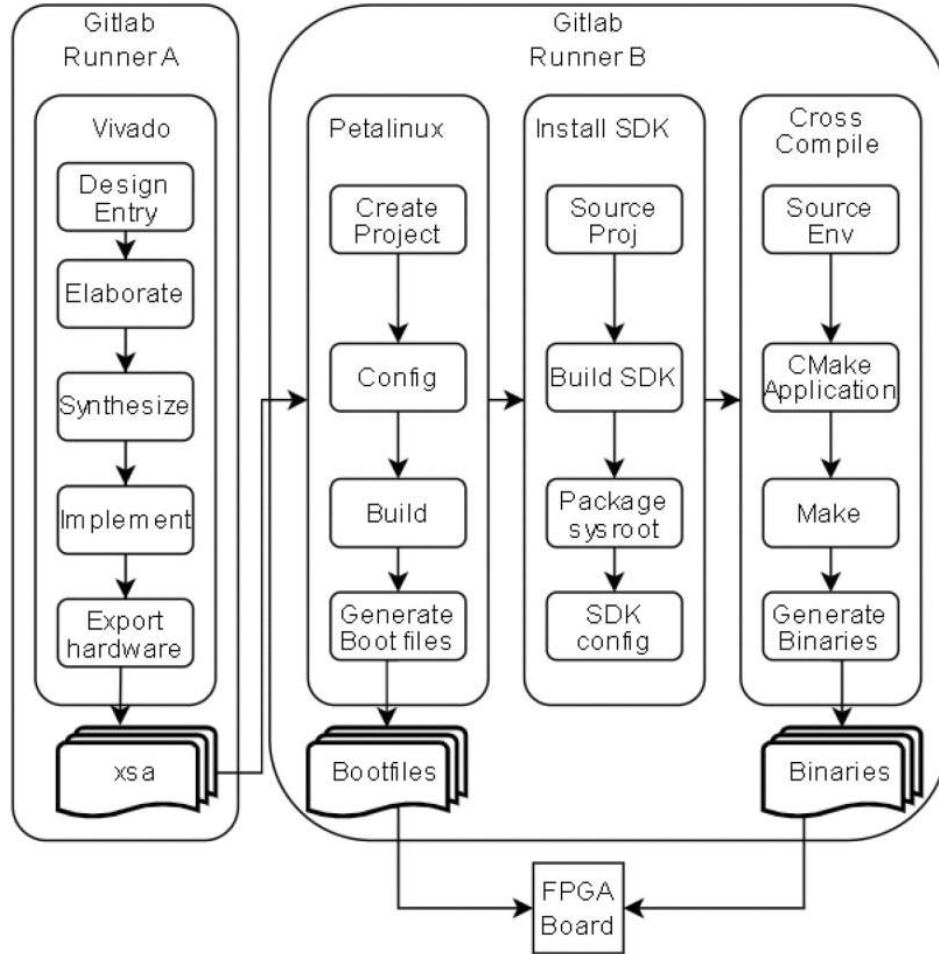
CI pipelines aware of real-time have to be explicitly engineered to adhere to time, power, and system stability constraints. This is particularly important for safety-critical domains like automotive and health care that cannot afford the cost of failure of functional correctness [20]. Such distinctions between software-led and hardware-centred CI/CD are essential to inform the design of automation framework that take into consideration the limitations of embedded development environments. In the next subsection we present how cross-compilation and flashing approaches are modified in order to facilitate these workflows.

### 2.2.2 Cross-compilation and Flashing Firmware

One of the first steps in bringing Continuous Integration (CI) to embedded systems is setting up continuous cross-compilation and firmware deployment. As embedded systems will have a hardware architecture different from the development host (e.g.,

ARM as opposed to x86), we can't use a normal compiler. Divergently, we require development to be based on cross-compilation toolchains, that are set with target-specific headers, libraries, and compilers [25].

Modern CI pipelines enhance this step by introducing modularity and automation across multiple stages. A recent implementation [26] demonstrates a CI/CD flow based on GitLab Runners that separates FPGA hardware design, embedded Linux configuration with PetaLinux, Software Development Kit (SDK) setup, and application cross-compilation. Each stage produces versioned artifacts consumed downstream, offering both traceability and reproducibility.



**Figure 2.6:** CI/CD build flow integrating hardware design, OS setup, SDK configuration, and application cross-compilation for embedded targets [26].

After having compiled the binary, you have to flash it to your target board. Methods for getting the new firmware onto the device span the range from low-level

approaches such as JTAG and SWD debugging connections to higher-level options including bootloaders, over-the-air (OTA) updates, and even remote flashing over SSH/serial links. GitLab CI jobs can take care of these steps and allow for scripted deployment to testbeds after a successful build [26].

In early-stage testing, to alleviate technical problem dependence on physical hardware, people widely use virtual interfaces and simulator technology [22]. This permits developers to run embedded behavior on their workstations, thus greatly boosting productivity and coverage. When linked to container technology, such approaches can maintain the same environment through every stage of CI. This solidly shuts off anything from outside which might affect build performance or behavior.

Tool	Setup and Hosting	Integration and Plugins	Use in Embedded/Arduino	Pros / Cons
<b>GitLab CI</b>	Self-hosted or cloud-hosted runners; flexible YAML configuration	Native integration with Docker, SSH, Git submodules; rich GitOps features	Used in modular flows for FPGA, SDK, and firmware compilation	<b>Pros:</b> Full control over runners, artifacts, secure stages. <b>Cons:</b> Hardware runners require extra configuration
<b>GitHub Actions</b>	Fully hosted on GitHub; minimal setup for public repositories	Vast plugin ecosystem via marketplace; supports matrix builds	Preferred for Arduino-based projects due to GitHub ecosystem integration	<b>Pros:</b> Easy to use, integrates with Arduino CLI, community support. <b>Cons:</b> Less control over runner environment, no direct USB access
<b>Jenkins</b>	Requires manual setup and server maintenance	Highly extensible with plugins (e.g., hardware flashing, Docker)	Historically used in large-scale embedded test automation systems	<b>Pros:</b> Powerful and flexible, custom pipelines possible. <b>Cons:</b> Steep learning curve, plugin maintenance overhead
<b>CircleCI</b>	Cloud-native with optional self-hosted runners	Pre-built Docker images, caching support, limited hardware access	Not commonly used in embedded due to hardware constraints	<b>Pros:</b> Fast setup, efficient for software-only projects. <b>Cons:</b> Limited suitability for hardware-in-the-loop testing
<b>Travis CI</b>	Cloud-based CI for open-source projects	Integrates easily with GitHub; supports Linux, macOS, and Windows	Previously used for open-source Arduino library testing	<b>Pros:</b> Simplicity and historical popularity. <b>Cons:</b> Performance limitations, declining popularity

**Table 2.2:** Comparison of CI/CD tools used in embedded system workflows

**Planned and Recommended Practices** The particular implementation presented here focuses on GitLab CI, but it is worth noting that the original plan was to move over to GitHub Actions. This preference results from Arduino’s strong reliance on GitHub-based workflows and the strong support offered by GitHub CI tools. Integrating hardware-in-the-loop test into GitHub Actions can complete the cycle and unify code, unit test and deploy in one toolkit.

In addition to tooling, some practices of architecture also improve CI reliability for embedded systems [27]. They involve setting up a centralized location for analytics repository so you can keep track version by version, managing configuration parity across environments, and having rollback procedures in place to handle botched deployments. Bringing these principles to bear helps preserve the stability of all kinds of targets in hardware and software.

**Conclusion and Trade-off Evaluation** Creating a cross-compilation and flashing pipeline for a CI/CD service enables embedded teams to shorten integration cycles, minimize risks of human errors, and improve the reliability of their deliveries. But the installation is non-trivial in that you have to have a good working knowledge of tool chains, firmware protocols, and hardware idiosyncrasies. As shown in Table 2.2, each method has trade-offs between complexity, velocity, and hardware sensitivity. At the end of the day, the choice of tools and workflow should ultimately depend on the scale of the project, the hardware you have access to, and long term maintainability.

### 2.2.3 Limited Debugging and Output Capabilities

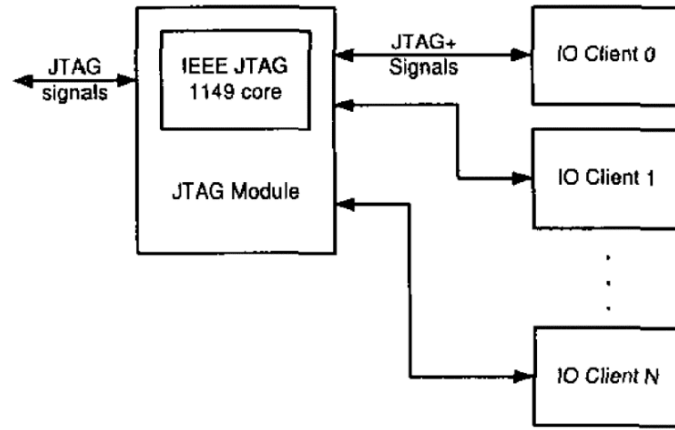
Debugging an embedded system has very different challenges compared to debugging traditional software. As opposed to desktop systems, in embedded platforms there may be very limited I/O, memory, and access to interior system states. This lack of clear visibility clouds fault isolation and extends development cycles. Developers are often left to use indirect methods such as LED status or serial output logs which can provide only so much information about the program flow or system state [28].

One fundamental challenge is that embedded systems are real-time systems. Debugging frequently must be done without the ability to stop code execution, and certain operations are time-critical and can’t be slowed down or aborted without causing the whole system to fail. Also, due to real-time constraints, the use of the default breakpoints or trace tools is restricted and developers need to move to more specialized modalities and methods [29].

As a means of tackling these limitations, embedded platforms take advantage of on-chip debug support (OCDS) mechanisms. One architecture divides the system into three main components for instance: resources for processor-specific debugging,

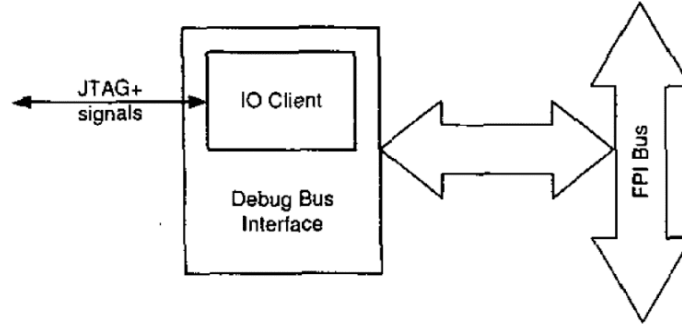
a serial communication interface with host interaction, and I/O clients which are linked to processors and buses in some way [30]. This architecture fits single-core as well as multi-core SOC's with minimal silicon overhead.

The JTAG module functions as a connection between the SoC and host debugger. In an IEEE 1149.1 standard signal format this interface uses JTAG+ signals that allow multiple IO clients to be communicated To. These clients make it possible for processor debug registers, memory, and system buses to be accessed without the need for external instrumentation [30].



**Figure 2.7:** Modular JTAG Debug Architecture with multiple IO Clients [30].

More advanced systems, connecting through shared system buses like the FPI bus, model this architecture to support clients which write data to multiple masters simultaneously. These clients can read or write to any memory location accessible over the bus, as well as any debug register. Most importantly, they can be configured with low-priority access without interfering with real-time tasks of the target system under test, so that the performance remains stable even while active debugging is going on [30].



**Figure 2.8:** Multi-master IO Client accessing FPI Bus for debug communication [30].

As embedded systems increase in complexity, especially with the unfolding of multi-core SoCs, the traditional debugging methods such as JTAG and UART no longer suffice to provide sufficient runtime visibility. In modern platforms, embedded trace units (ETUs) are integrated at last. These units stream compressed trace data depicting the control flow, memory access, and system events. Making possible low-intrusion observation, they greatly exceed the sweep and buffer capacity of their predecessors [31].

In order to address these rigidities, an FPGA-based approach has been suggested which processes trace data as it is created in real time. Rather than collect trace logs to be analyzed offline, this technique provides live reconstruction of execution flow and monitoring of runtime properties utilizing monitors defined in a specification language called TeSSLa. This infrastructure is capable of providing ongoing observability of the system under test (SUT) and supports real world use cases, including timing analysis, code coverage, functional validation and fault detection [31].

When trace-based observability is paired with hardware-based real-time verification, it provides a scalable, effective alternative for traditional debugging methods. It provides better visibility into the internal behavior of embedded software without perturbing execution timing or needing source-level instrumentation.

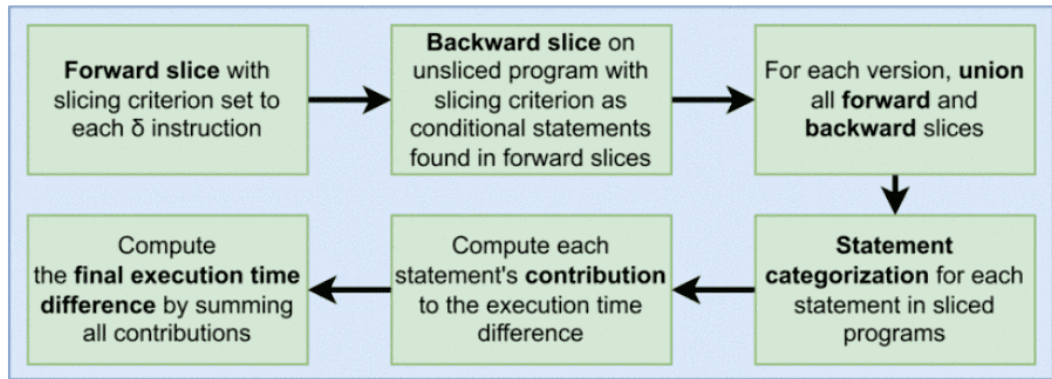
#### 2.2.4 Timing, Power, and Environmental Constraints

Timing validation is critical in embedded systems, especially in safety-critical applications where missing deadlines may result in system failure. Traditional

Worst-Case Execution Time (WCET) analysis methods evaluate the full codebase, which is often impractical for CI/CD workflows involving frequent updates and rapid iterations.

To address this, differential timing techniques such as *Relative Timing Analysis (ReTA)* have been proposed. ReTA targets and examines only the snippets of code affected by recent changes. It performs forward and backward slicing of the device graph in order to determine timing-relevant paths and allow for a fast estimation of the timing delta between firmware versions [32].

The ReTA process is depicted in Figure 2.9, whereby changed instructions are traced and classified to calculate timing deviations without reexecuting the whole binary. The technique has been implemented in the *Delta* tool for Cortex-M processors, and it can provide near-hardware-accurate WCET estimation with up to 45% improvement in analysis time[32].



**Figure 2.9:** Workflow of Relative Timing Analysis (ReTA) [32].

Beyond timing, embedded CI pipelines must balance power and thermal constraints. Repetition of highly prioritized jobs may result in high power consumption or trigger overheating on mobile devices. To mitigate the tradeoff between performance and energy budget during automated test cycles, low-power scheduling, dynamic frequency scaling, and runtime monitoring are frequently used.

Lightweight timing tools such as ReTA enable efficient validation of real-time behavior while capturing its safety and liveness, thereby enabling agile CI/CD operations while maintaining the safety and responsiveness of the system.

### 2.2.5 CI Bottlenecks in Production-Grade Embedded Workflows

In production-grade embedded systems it is usually not as easy to use Continuous Integration systems as with standard software. A cross industrial study (telecommunication and avionic) of two real-world cases, detected several reoccurring

bottlenecks that slow down or complicate CI adoption [16].

Build time is one of the biggest challenges. In embedded systems, where dependencies tend to be tightly coupled, even a small change can cause a full system rebuild to take place, often actually requiring hours. This slows down the integration cycle and lessens the number of commits, while CI demands that developers push and integrate code multiple times a day. A long build flies in the face of “fast feedback,” and eliminates the ability to rapidly iterate, which CI should facilitate.

A big problem is also hardware access. Embedded software is usually tested on one-off custom boards or rigs that are expensive and scarce. Testing on real hardware for each and every commit starts to become unwieldy. Some projects attempt to use simulators, but these fall short when it comes to accuracy or coverage to fully model the behavior of actual devices. This makes it difficult to determine precisely what is meant by “all tests must pass,” particularly when hardware variants are applied.

Plus in regulated industries like aerospace, developers must also cope with strict compliance requirements. This means writing documentation, passing audits, and demonstrating a critical event tracking system. Though important for safety, these commitments can at the same time detract from the software itself. A failed build can become less important than documents for a certification review.

Finally, teams working on embedded systems often span several technology areas—mechanical, electrical and software. This means that workflows are not only isolated from one another but many times engineers don’t have access to the whole system view too, particularly if security policies restrict information sharing. It’s then harder to audit builds globally, see failures in context, or apply CI feedback across whole systems.

Together, most embedded development teams find it hard to fully put CI into practice in its original form. Although the basic concepts of CI still hold true, real hardware constraints, safety standards and complex organization structures often demand that workflows be adjusted accordingly.

## **2.3 Practical Techniques and Architectures for Embedded CI**

This section presents the main issues that differentiate embedded systems from traditional software projects, followed by practical solutions and architectures, which have been developed to provide a way through which Continuous Integration (CI) can be extended towards hardware-aware development environments. Pseudo-soft CI pipeline. Unlike typical software-only CI pipelines, for Embedded Systems, you need to work with something real, getting responses from actual devices, timing



accurate verifications or measurements at the signal level.

In order to meet these requirements, embedded CI solutions use a variety of hybrid strategies, where both software simulation and hardware validation occur. Such strategies embrace Hardware-in-the-Loop (HIL) testing, virtual platforms, SCPI-based automatic instrumentation, and custom CI/CD workflows with tools like GitLab or GitHub Actions. Together, these make reproducible firmware deployment, real-time verification, and effective test automation across a wide range of different embedded targets.

This section provides detailed explanations of each method, with references to documented research and actual implementation. It is designed to give the reader a clear and comparative overview of existing solutions, so that he can understand how in practice such architectures can join together what has previously been only theoretical CI theory.

### **2.3.1 Hardware-in-the-Loop (HIL) Testing**

Hardware-in-the-Loop (HIL) testing is the cornerstone of validating embedded software - putting real hardware into a controlled test environment. Unlike simulations, based on software models, HIL setups are in direct contact with the physical target. As a result they are particularly effective in fields where safety demands cannot be satisfied by software alone: for instance, automotive and aerospace.

This approach is designed to embed the software onto the target platform, but this platform then communicates external components such as oscilloscopes and signal generators with which to carry out two-way testing of system behavior. These peripheral devices provide input stimuli and measure outputs to assess real-time performance. Figure 2.10 shows a typical manually operated HIL setup where an oscilloscope or signal generator is used to generate signals and measure response.

Manual HIL testing is prone to errors, time-intensive and hard to scale in a CI/CD environment. Experiments usually have to be performed in sequence, and a mismatch in timing or signal synchronization may lead to erroneous results. To overcome this, the authors in[33] propose a fully automated alternative presented in Figure 2.11.

This architecture pushes a large amount of control logic directly onto the host PC, eliminating the need for instrumentation on the microcontroller. The debugger links up via JTAG and a host platform runs test script and vehicular simulators, in addition to communicating with an HIL test system. This architecture facilitates better synchronization, lowers the variance of the running time, and overcomes on-chip resource limitations experienced by other approaches.

To illustrate the difference between manual and automated testing, a performance comparison of DIO driver was carried out. As shown in Table 2.3, the automated method drastically reduced test execution time, improving speed and consistency.

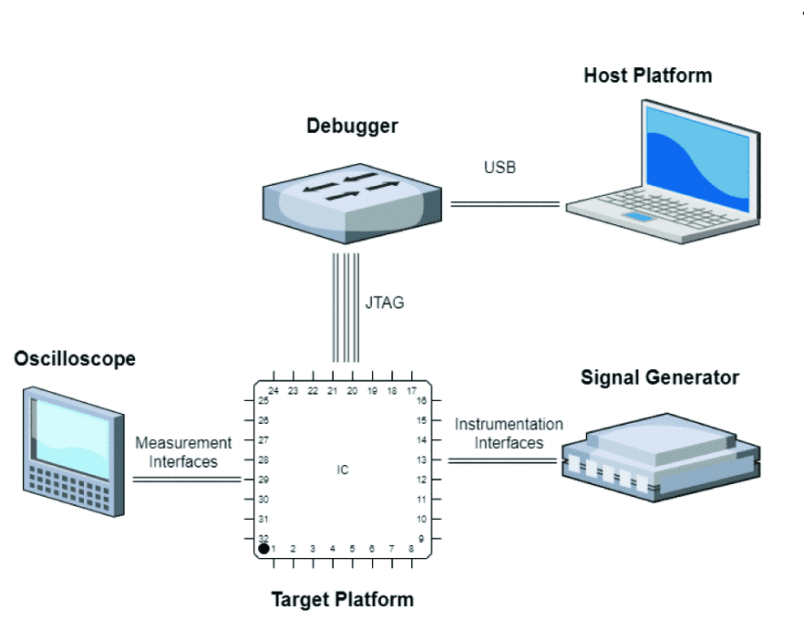


Figure 2.10: Test setup for manual tests [33]

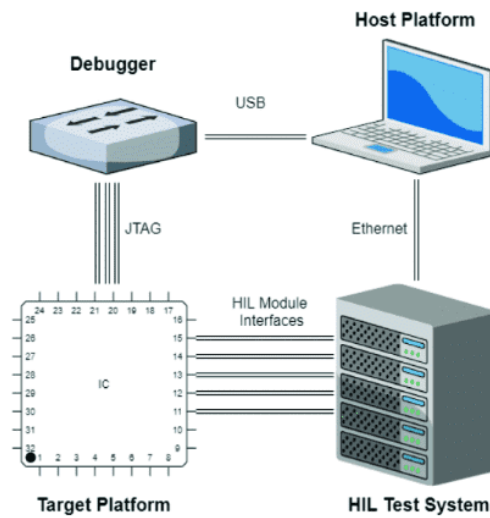


Figure 2.11: Hardware/software integration test concept based on host-executed test scripts [33]

#	Test Description	Manual [s]	Automated [s]
1	Read low level at DIO0	80	3.042
2	Read high level at DIO0	76	3.169
3	Read low level at DIO1	56	3.178
4	Read high level at DIO1	53	3.193
5	Write low to DIO0	51	3.444
6	Write high to DIO0	45	3.377
7	Write low to DIO1	50	3.256
8	Write high to DIO1	40	3.174

**Table 2.3:** Execution Times of DIO Software Component Tests [33]

Statistical analysis was also performed to understand the variability of execution times. The standard deviation for manual tests ( $s_m$ ) and automated tests ( $s_a$ ) were calculated as:

$$s_m = \sqrt{\frac{\sum(t_{\text{runtime}} - \bar{t}_{\text{manual}})^2}{n}} = 13.331 \text{ s}$$

$$s_a = \sqrt{\frac{\sum(t_{\text{runtime}} - \bar{t}_{\text{automated}})^2}{n}} = 0.119 \text{ s}$$

where  $\bar{t}$  is the average runtime and  $n$  is the number of test cases. The large spread for deviation proves that manual tests are not only slower, but also more inhomogeneous and therefore more prone to errors.

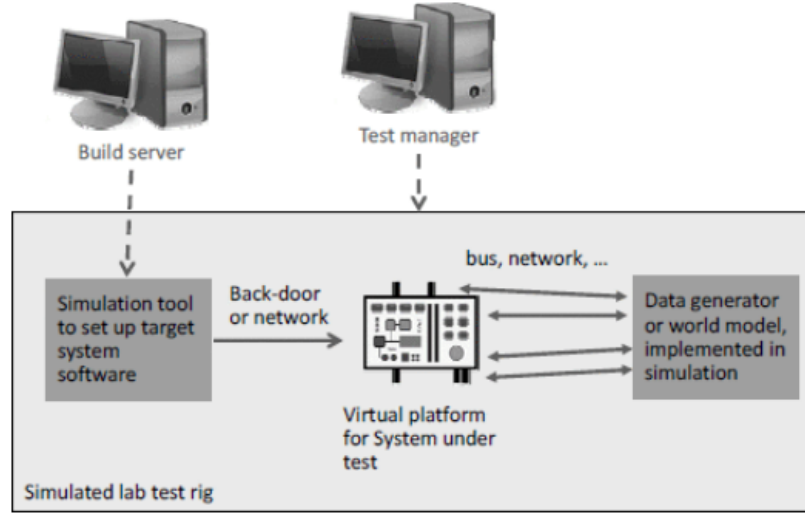
The Go based HIL testing with SCPI connected instruments and Arduino CLI was adopted for work of this thesis. The automated process was similar to the host-controlled one, ensuring appropriate control over the signal generation and even real-time monitoring through oscilloscopes and DMM.

Overall, integrating HIL into CI workflows boosts efficiency and improves confidence in embedded firmware, especially when changes need to be validated continuously. The approach presented in [33] aligns closely with modern embedded development goals, supporting scalable and reproducible validation of hardware-software integration.

### 2.3.2 Virtual Platforms and Emulation

In the world of embedded systems, Continuous Integration (CI) struggles due to hardware that is hard to access, time-consuming test setup, and the complexity that comes in automating an environment that makes mobile development seem like child's play. In such challenges, virtual platforms are considered as a viable alternative to hardware-based testing. A virtual platform (VP) simulates a target hardware at the transaction level, allowing unmodified embedded software to run in a controlled and deterministic environment [34].

Virtual learning tools offer a number of important benefits relative to traditional lab-based experiments. They enable testing in parallel across virtual machines, reduce the latency of tests and streamline test automation. Additionally, developers can simulate the processor and peripherals as well as the interaction of the system with its environment, including sensors or communication buses.



**Figure 2.12:** Integration of virtual platforms with environment models for embedded CI [34]

Figure 2.12 illustrates how a VP can be integrated with simulations of the physical environment. This configuration enables developers to test system behavior under realistic circumstances without needing physical hardware.

Another important feature is checkpoints where the simulation state can be saved and used to make test campaigns faster and reproducible for debugging purposes. This is especially useful for catching the kinds of intermittent or timing-related bugs that would be difficult to diagnose using real hardware.

While physical boards are still needed for final validation, the virtual platforms really lower the reliance on hardware in the earlier stages of CI significantly. This serves the aim of moving testing to occur earlier in the development process, of increasing the frequency of feedback afforded by testing, and accommodates Agile working practices in the domain of embedded software engineering [34].

### 2.3.3 Automated Instrumentation (Oscilloscopes, DMMs)

Automated control of those instruments like oscilloscopes and digital multimeters, is an essential component for embedded CI workflows, especially when testing power levels, pwm signals, and analog outputs in unattended test loops. The contemporary systems use SCPI format (Standard Commands for Programmable Instruments) for consistent communication throughout devices across different vendors, allowing these instruments to be contacted from CI scripts, remote agents, or embedded test managers.

One typical solution is to link measurement devices to an embedded controller

or a host PC using USB, Ethernet, or serial interfaces. By use of some software libraries (e.g., PyVISA – Python, or VISA Interfaces – LabVIEW), SCPI commands can be sent to set trigger levels and capture screenshots; plot voltage or frequency values and compare with some predefined thresholds. Such a technique enables automatic signal validation on firmware flashing or on hardware replacement.

The advantage of SCPI lies in its standardization. SCPI, in contrast to vendor-specific application programming interfaces (APIs), articulates a uniform ASCII hierarchy of commands that is supported by most modern instruments. This regularisation makes scripts that run hardware from any vendor possible. For instance, a typical command to measure signal frequency is:

`MEASure:FREQuency?`

This query functions similarly across many oscilloscope brands, greatly simplifying CI test code reuse.

The use of SCPI in embedded test environments not only reduces manual effort and operator variability but also improves reproducibility across test cycles. Furthermore, since the same command sequences can be exported and run in simulation or hardware, it supports both virtual platform-based and real-hardware-based testing uniformly [35].

### **2.3.4 GitHub Actions: Resource Usage and CI/CD Tool Utilization**

GitHub Actions is now one of the more ubiquitous CI/CD platforms, with native support for GitHub-based repositories and up to 2,000 free VM minutes likely used each month on public projects. Nevertheless, recent works demonstrate that workflow executions may lead to a high consumption VM and inefficiencies at users of paid-tiers [36].

Empirical analysis on 1.3 million runs from GitHub Actions workflows over 30 months shows that 91.2% of the VM compute time is consumed by builds and tests. These are generally run on pull requests, direct pushes, or scheduled workflow runs. Paid repositories used about 5,914 VM minutes a month, or \$504 per year, with free repositories typically doing briefer builds and tests (1.5 minutes vs 9.6 minutes in the case of builds) [36].

The same study reported 17.4% of workflows subjected to the different running time constraints of their paid-tier plans fail, which not only bloats cost with re-runs, but also in lengthy execution, unfolding up to time-outs. These deficiencies point out the importance of proper optimization techniques, some of which are not popular.

Despite the multiple built-in means that GitHub Actions offers for minimizing the resource waste, such as cache, fail-fast, cancel-in-progress and custom timeouts,

they are not popular. For instance, caching was included in only 32.9% of paid repositories, despite saving 3.5% VM time and \$21 per year on average. Only 10.1% took advantage of cancel-in-progress that could save 4.1% VM time [36].

Strategy	Adoption (%)	VM Time Saved	Yearly Savings (\$)
Caching	32.9	3.5%	21.5
Fail-fast	75.9	1.5%	2.1
Cancel in Progress	10.1	4.1%	62.6
Custom Timeouts	14.0	8.1%	58.3

**Table 2.4:** GitHub Actions optimization strategies and their impact [36].

A large-scale study on 1.5 millions of GitHub Actions workflows from almost 33,000 projects, divided the tools used based on their CI/CD domains into five categories, namely Build Automation, Test Automation, Static Code Analysis, Version Control, and CI/CD Servers [37]. The Build Automation category was the most comprehensive (and by far the most popular) with 45% of unique tools. However, Test Automation tools, which are particularly important for CI, had only 6% variety and almost 0% usage share.

These results imply that while the infrastructure level support for CI/CD is quite mature for GitHub Actions, there is the possibility to optimize and take advantage of various testing abilities. Especially in paid-tier scenarios, cost and performance inefficiencies persist due to insufficient use of caching, failure handling, or timeout strategies. Also the low use of automated testing tools suggests that the practice of CI/CD is not fully applied in real-world repos as it could.

Future work could focus on automated, ML-driven recommender systems may be developed that can modify GitHub Actions workflows to make them more cost-effective, robust and follow more closely to CI/CD best practices.

Since most Arduino libraries and examples are created on GitHub, something like GitHub Action is a straightforward way to stay true to the ecosystem philosophy for embedded developers. In this thesis, GitHub Actions was considered for integration due to its relevance in Arduino workflows, but optimization approaches have remained an subject of ongoing experimentation.

## 2.4 Summary and Outlook

This chapter gave a comprehensive understanding of the role that CI plays in embedded system development over time: the origins with software engineering, to challenges of leveraging it on resource-powered environments, as is the case with hardware-limited systems. First, they started by revisiting the history of CI practices, when manual merges were replaced by automated pipelines, and tools

like Jenkins, GitHub Actions, or GitLab CI became critical parts of developer workflows.

The latter half of the chapter discussed challenges that need to be addressed to extend CI to embedded systems. This includes cross-compilation, flashing the firmware, debugging over restricted interfaces, and validating real-time behavior. In addition to these issues, issues such as test resource constraints, long build times, and conformance-related restrictions were mentioned as well.

In order to address these challenges, various alternatives were considered. Hardware-in-the-loop (HIL) testing and virtual platforms were proposed as methods to reduce the hardware dependency and to allow for more scalability in CI. SCPI-controlled instrumentation with Go-based orchestration was developed to programmatically confirm signals and measurements. Recent studies on GitHub Actions were also addressed, notably related to workflow improvement and tooling consumption patterns.

On the whole, the chapter emphasized the fact that embedded CI is much more hands-on than a common software pipeline. It's not simply a matter of doing the build and test automatically — one has to integrate physical measurement, hardware control, and domain specific constraints into a single, coherent process that can be repeated. The next chapter puts all this in perspective, it demonstrates how we applied such strategies in practice to a real working system for embedded test automation.

### **2.4.1 Key Takeaways from Existing CI Tools and Techniques**

From the tools and cases that we have reviewed, some general remarks can be made:

- CI in embedded systems is not plug-and-play. It requires hardware orchestration, cross-compilation, and measurement coordination.
- Tools like Jenkins and GitLab CI are flexible but need a lot of setup and sometimes need to be interfaced with hardware.
- GitHub Actions is widely adopted in open-source firmware projects but is often underutilized in terms of caching, test automation, and optimization.
- HIL setups and virtual platforms help close the gap between purely software-based and physical tested approaches.
- SCPI-based instrumentation brings repeatability and automation to some signal validation that would otherwise have been manual work.



These results guided the design and architecture decisions of the actual implementation of this thesis.

### **2.4.2 Justification for the Thesis Implementation Strategy**

The decisions which is in this thesis were highly affected by the lack and weakness in standard embedded CI pipelines. A host-side orchestration system using Go was chosen for its performance, cross-platform nature, and ability to handle serial communication, SCPI over TCP/IP, and file generation.

The use of Arduino CLI allowed flexible firmware compilation and uploading without the need for a heavy IDE. Vendor-neutral SCPI commands were a way of controlling oscilloscopes, power supplies and digital multimeters in automated tests. This allowed for a modular and repeatable configuration where we could throw lines of compilation, flashes, and measurements and we could call, verify and log.

It's in line with the current direction of embedded testing, to reduce the amount of human interaction and yet have all the traceability and measurement capability saved.

### **2.4.3 Planned Use of GitHub Actions in the Arduino Ecosystem**

Although the prototype described in this thesis runs locally using Go scripts and shell orchestration, the long-term goal was to bring the entire flow into GitHub Actions. This choice is based on the fact that Arduino development already heavily relies on GitHub for code hosting, library distribution, and issue tracking.

GitHub Actions would allow the CI system to be triggered by pushes or pull requests, compile the firmware using Arduino CLI, run logic in Docker containers, and—if connected to hardware test servers—perform remote HIL validation too. While this step was not implemented in full, the system was intentionally designed to be compatible with such a future extension.

This keeps the workflow aligned with Arduino's own development culture and allows future contributors to continue building on top of the existing automation in a way that's scalable and GitHub-native.

## Chapter 3

# Methods Developed in the Thesis

This chapter describes the system that was developed to automate testing and validation for embedded firmware using a Continuous Integration (CI) workflow. The goal was to build a setup where hardware tests could run automatically whenever new code was pushed, without the need for manual interaction with devices or instruments.

I designed the system as a composition of various parts: a Go program running at the host PC, an Arduino GIGA board, acting as the Device Under Test (DUT), a power supply, a DMM, and an oscilloscope as some lab instruments. These instruments are linked via standard interfaces like USB, LAN, and serial ports. The host PC control it all, through SCPI commands and a serial link. In this way, I can flash firmware to the Arduino, generate a control signal, take measurements, and save results in JSON format for later analysis.

Before diving into the details of each component, the next section provides an overview of the full system and its architecture.

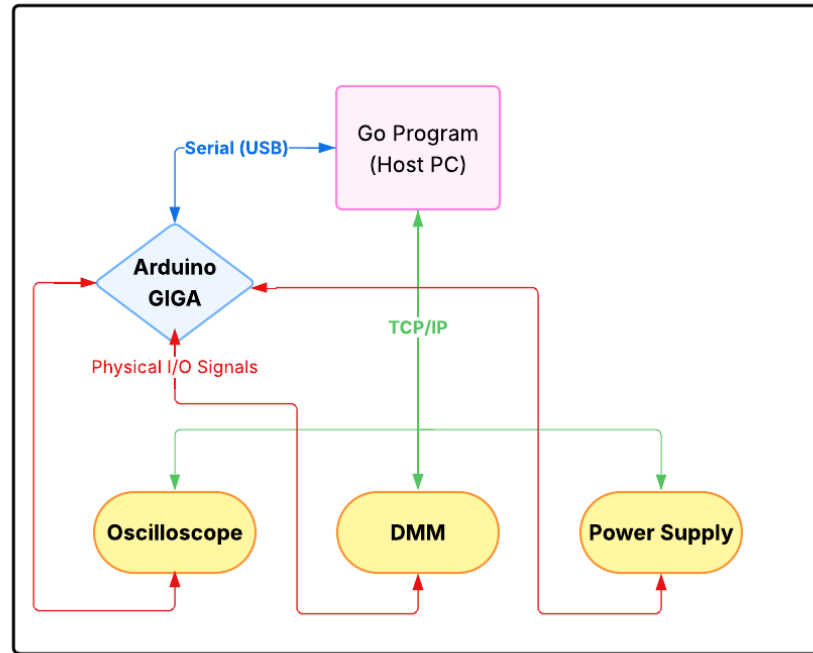
### 3.1 System Architecture Overview

The entire system is designed around the idea of automating real hardware validation within a CI pipeline. As shown in Figure 3.1, the host PC sits at the center of the workflow. It is responsible for compiling the Arduino sketch using Arduino CLI, uploading the firmware to the DUT, sending test commands over serial, and collecting measurements from the lab instruments.

The Arduino GIGA is programmed to understand a custom set of commands,

referred to as ATU (Arduino Test Utilities) commands. These allow the host to request specific behaviors such as generating a PWM signal with a certain duty cycle, setting a voltage on an analog pin, or performing digital reads and writes. The firmware on the Arduino parses these commands and responds accordingly, acting like a small interpreter running on the board.

On the measurement side, the power supply, oscilloscope, and DMM are all controlled using SCPI (Standard Commands for Programmable Instruments). This standard makes it possible to talk to different brands of instruments using the same kind of ASCII-based commands. For example, to read the voltage on a DMM, the host sends a SCPI command like `MEAS:VOLT?`, and the DMM replies with the measured value. This makes it easy to automate checks for things like whether the output voltage is correct or if a PWM signal has the right frequency and duty cycle.



**Figure 3.1:** High-level architecture of the automated embedded testing setup.

The workflow is designed to be modular and easy to extend. If new types of tests are needed, it is usually enough to update the Arduino sketch with new ATU commands and add corresponding logic on the host side. The instruments are already integrated, so any test that can be measured using voltage, current, or signal timing can be added without major changes to the setup.

This architecture has the advantage of combining real hardware validation with a flexible, scriptable test environment. It enables fully automated test runs inside GitHub Actions or other CI platforms, bridging the gap between software-level CI and real embedded hardware testing.

## **3.2 Arduino Firmware and ATU Command Design**

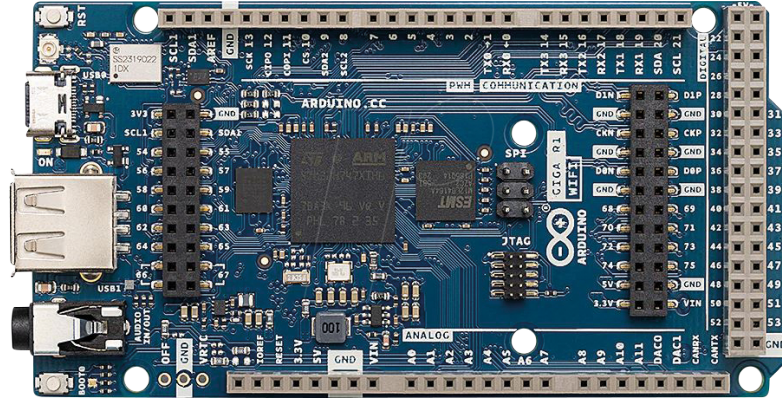
In order to be able to control the DUT in a flexible way, I wrote my own firmware for the Arduino GIGA based on the ATU (Arduino Test Utilities) framework. I expanded the firmware on it to add a variety of serial commands which would initiate various test routines. And that's with things like creating PWM signals, controlling analog output levels with DAC, or reading digital and/or analog inputs of the board. I was also responsible for the command parsing and response formatting logic, so the host application can communicate with the DUT in a consistent manner.

ATU framework was selected for its simplicity, extendability, and compatibility with the Continuous Integration (CI) systems. It abstracts direct register manipulation and offers a command-based API where an external program (such as the Go orchestrator) can request an operation by sending a human-readable string. This makes the communication protocol simple, debuggable, and cross-platform compatible.

The firmware is a simple loop that continuously runs on the Arduino. It sits receiving commands from the serial port, and then parsing them and performing the appropriate operation depending on what was requested. After processing each command, it sends a message back if the operation was successful or not. It makes your communications two-way, no matter what, and easy to debug when you're testing.

### **3.2.1 GIGA Board Hardware Features and Role**

The Arduino GIGA R1 WiFi is the core hardware platform for the Device Under Test (DUT) in this project. It was selected for its extensive I/O, dual-core design, and built-in communications. Its flexibility also makes it versatile for embedded systems, in the context of advanced automation testing and CI workflows.



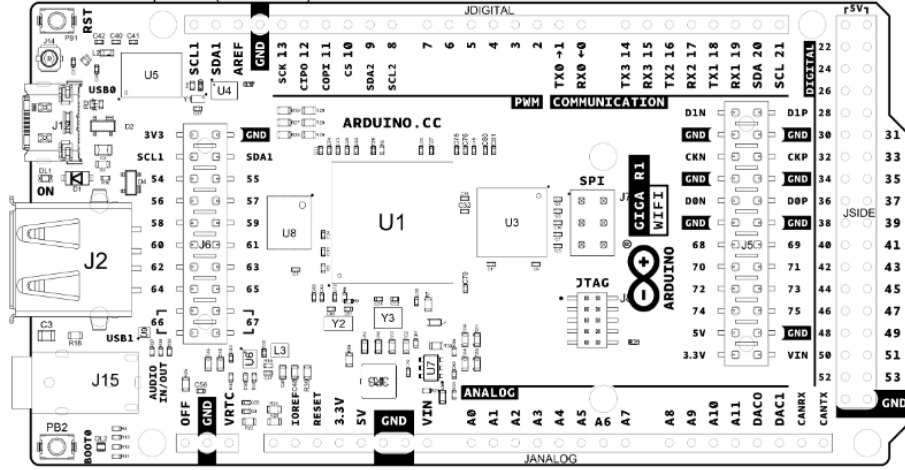
**Figure 3.2:** Arduino GIGA R1 WiFi – Physical appearance

### Key Features:

- **Microcontroller:** STM32H747XI with dual-core architecture:
  - Arm<sup>®</sup> Cortex<sup>®</sup>-M7 @ 480 MHz
  - Arm<sup>®</sup> Cortex<sup>®</sup>-M4 @ 240 MHz
- **Memory:** 2 MB Flash, 1 MB SRAM, 16 MB external NOR Flash (QSPI), and 8 MB external SDRAM.
- **Connectivity:** Integrated Wi-Fi 802.11 b/g/n and Bluetooth Low Energy (BLE) via Murata 1DX module.
- **I/O Capabilities:** 76 digital pins, 14 analog inputs, 13 PWM-capable pins, 2 DAC outputs (up to 12-bit).
- **Interfaces:** USB 2.0 Host (Type-A), USB-C Programming/Peripheral, 4 UART, 3 I<sup>2</sup>C, 2 SPI, 1 CAN (via transceiver).
- **Operating Voltage:** 3.3V logic level, input voltage range from 6–24V via VIN or USB.

In this thesis, the GIGA board serves as a programmable DUT to process test commands sent through a serial interface from the host PC. These strings, separated by semicolons, are parsed and processed by a custom ATU (Arduino Test Utilities) firmware. Depending on the command, the GIGA can produce a PWM signal, place a DAC voltage on an analog output, or use digital I/O. The resulting outputs are then observed using lab equipment, aiming to verify the behaviour of the firmware under the tests.

A key reason for selecting the GIGA was its broad pin availability, which enabled simultaneous use of digital I/O, analog output (DAC), and serial communication for orchestration. For instance, PWM waveforms were generated on D9 or D6, DAC voltages on A12 (pin 84), and analog voltage was read back using the Keithley DMM6500 to ensure voltage accuracy. This made the board well-suited to multi-instrument measurement setups in a hardware-in-the-loop (HIL) testing framework.



**Figure 3.3:** Arduino GIGA R1 WiFi – Top view with labeled components and pin layout.

#### Notable Pin Assignments in This Project:

- **PWM Output:** D9 — used to generate 30% duty cycle PWM.
- **DAC Output:** A12 — used to generate 1.5V for DMM validation.
- **Analog Measurement:** Pin A0 — monitored by oscilloscope and DMM.
- **Serial Interface:** USB-C port — used for ATU command communication at 115200 baud.

The dual-core design of the board and the high-speed peripherals additionally allow for upgrading with real-time signal processing, concurrent monitoring on both cores, or edge AI expansions, and many other features. Although in this work only a single core was actually used, the STM32H7 opens significant headroom to scale up CI test cases or enable autonomous decision-making onboard.

### 3.2.2 Command Structure and Syntax

Each command sent to the Arduino follows a basic pattern of:

`<COMMAND>;<PIN>;<VALUE>`

This format allows the system to remain lightweight and avoid parsing complexity on the microcontroller. The semicolon-separated format is easy to split and validate. Commands are case-sensitive and end with a newline character (`\n`) to trigger parsing.

For example, to generate a PWM signal with 30% duty cycle on pin D9, the following command is sent:

`AW;9;77`

Here, `AW` stands for *Analog Write*, pin 9 is the target output, and 77 is the duty cycle value on a scale from 0 to 255 (where 76.5 maps approximately to 30%).

The firmware includes a predefined list of supported pins, which ensures that only usable I/O lines are initialized and keeps I/O lines that are not used, deactivated. This avoids erroneous definition of reserved or incompatible pins on the GIGA board.

```
1 uint8_t usedPins[] = {
2     // Digital pins D0–D13
3     0, 1, 2, 3, 4, 5, 6, 7,
4     8, 9, 10, 11, 12, 13,
5
6     // Analog pins A0–A5
7     76, // A0
8     77, // A1
9     78, // A2
10    79, // A3
11    80, // A4
12    81, // A5
13
14    // Extended digital pins D22–D53
15    22, 23, 24, 25, 26, 27,
16    28, 29, 30, 31, 32, 33,
17    34, 35, 36, 37, 38, 39,
18    40, 41, 42, 43, 44, 45,
19    46, 47, 48, 49, 50, 51,
20    52, 53,
21
22    // DAC output (A12)
```

```

23 |      84  // DAC_0 / A12
24 | };

```

The ‘setup()’ function initializes serial communication at 115200 baud and enables the ATU interpreter. After initialization, the ‘loop()’ function continuously checks for new commands using ‘ATU.parseCommand()’. Once a command is received, it is processed by ‘ATU.executeCommand()’ and the result is sent back to the host.

```

1 void setup() {
2   ATU.begin();
3   Serial.begin(115200);
4   ATU.setPins(usedPins, sizeof(usedPins) / sizeof(usedPins[0]));
5 }
6
7 void loop() {
8   String command = ATU.parseCommand();
9   if (command != "") {
10    String result = ATU.executeCommand(command);
11    if (result != "") {
12     ATU.result(command, result);
13    }
14  }
15 }

```

While maintaining the logic readable and maintainable, this minimal firmware allows remote-controlled actions with just a few lines of code. It also separates the firmware logic from the test orchestration system so that new capabilities may be included without influencing host-side automation.

### 3.2.3 Firmware Logic and Command Parser

The firmware flashed in the Arduino GIGA is designed using a loop-oriented model, which is commonly used in microcontrollers. It has the duty of being a command interpreter that reacts to serial input, performs a corresponding low-level operation (on a pin), and replies with a result over the same serial interface. It allows remote automation and scripting from the host side, so no onboard UI or manual activation is required.

Once the board is powered and the ‘setup()’ function is executed, the ATU system is initialized by calling ‘ATU.begin()’, and all available pins are registered using ‘ATU.setPins(...)’. This step defines which pins can be addressed later by incoming commands and prevents the firmware from attempting to operate on



undefined or unsafe pins. Serial communication is initialized at a baud rate of 115200, which provides a good balance between speed and reliability for UART communication.

During the `loop()`, the firmware keeps calling `ATU.parseCommand()`, which does not return until one line of a complete command is received (including the newline). When a recognized command is received, it is tokenized and dispatched to the corresponding internal ATU (e.g., digital write, analog read, PWM output). The `ATU.executeCommand(...)` function is an example of this encapsulation, and guards against inappropriate values from getting into the world of specific pin numbers or values that can be written. The firmware then uses `ATU.result(...)` to return the response to the host.

This modularity keeps the parsing, execution, and response generation, making it easier to add new ATU commands in the future or modify existing behavior. For instance, implementing a new command for reading an analog input only would involve extending the parsing logic with a new case like `"AR"` (short for *Analog Read*) and routing it to the corresponding `analogRead(...)` function.

The command parser relies on simple string manipulation functions provided by the Arduino `String` class. While this may not be optimal for performance in very constrained environments, it offers fast development and debugging for moderately sized embedded projects like the one in this thesis. Since only short strings are parsed and no dynamic memory is retained after execution, the risk of fragmentation or overflow remains minimal.

The command is tokenized internally based on semicolon delimiters (`;`) using simple string parsing inspired by the `String` class of Arduino. The parser then examines if the first token is a recognized keyword, `"AW"` (Analog Write), `"AR"` (Analog Read), `"DW"` (Digital Write), or `"DR"` (Digital Read). The command is executed if the keyword is correct and the number of arguments follows the anticipated pattern. If not, then the firmware replies with the smallest error or doesn't take action on the input.

An example of a valid parsing process for the command `'AW;9;127'` would be:

- `'AW'` → Analog Write
- `'9'` → Pin D9
- `'127'` → Duty cycle ( $\approx 50\%$ )

This command would internally trigger ‘`analogWrite(9, 127)`’ on the GIGA board, and the response string ‘`"AW;9;127"`’ would be echoed back to the host to confirm execution.

The firmware is intentionally kept minimal, with all execution logic routed through the ATU library. This makes it easier to test the host-side Go program independently, since any Arduino running the ATU firmware can behave identically.

If corrupt input, unknown commands, or bad pins are entered, the firmware does not give very good feedback to the user at the moment. Perhaps something like number code returns or verbose error output is a possible addition for the future. In reality, however, testing has proven the current structure to be sufficiently strong (at least from a testing standpoint) that automated tests in the CI pipeline are already enabled and can run uninterrupted.

### **3.3 Host-Side Automation Logic in Go**

The host-side logic is where everything comes together. This part of the system runs on the PC and controls the full test process from start to finish. It was written entirely in Go because the language is lightweight, easy to organize into modules, and fast to compile. In addition, it works well with networking and serial communication, which made it a good fit for this project.

The Go program acts like a conductor. It compiles and uploads the firmware to the Arduino using Arduino CLI, then communicates with it using a serial port to perform various ATU commands. It connects to the lab instruments via LAN and sends SCPI commands at the same time. Those devices are a power supply, an oscilloscope, and a DMM. It can tell the instruments to apply voltage, read current, check instrument signal properties, and even take a screenshot of the oscilloscope display.

All measurements are gathered and saved automatically in a structured JSON file. The filename includes a timestamp, so each test run produces a unique output. This is also useful to structure the results in an easily reviewable and comparable format. The tool can be launched manually in a terminal, run as part of GitHub Actions to run the full test pipeline when a new piece of firmware is pushed.

In the next parts, each component of the Go program will be explained. This includes how the firmware is compiled, how the system interacts with the instruments, and how the final test results are stored.

Each part of the workflow is encapsulated by its own function in the Go program, so if one part fails, it won't take everything else down with it. So, if the Arduino fails to start in the right way, the system can progress and take what it can from the lab instruments. This allows the tool to be more fault-tolerant, particularly for automated runs.

The orchestration logic also includes utility functions for formatting values, managing serial buffers, trimming strings, and generating readable filenames. This helps ensure consistency across test runs and simplifies future debugging or validation.

### 3.3.1 Overview of the Go Orchestration Program

The Go orchestration program is the central automation layer that controls the test workflow from start to finish. It starts by parsing runtime configuration parameters passed as flags, such as the path to the sketch, IP addresses and ports of instruments, and USB Vendor ID and Product ID to locate the Arduino GIGA. This allows the same program to work across different setups with minimal changes.

```
1 dataPath := flag.String("dataPath", defaultDataPath, "Path to the  
   sketch data")  
2 fqbn := flag.String("fqbn", "arduino:mbed_giga:giga", "Board name")  
3 port := flag.String("port", "/dev/ttyACM0", "Arduino port")  
4 vid := flag.String("vid", "2341", "Arduino Vendor ID")  
5 pid := flag.String("pid", "0266", "Arduino Product ID")  
6 psIP := flag.String("psIP", "10.130.22.111", "Power Supply IP")  
7 osIP := flag.String("osIP", "10.130.22.229", "Oscilloscope IP")
```

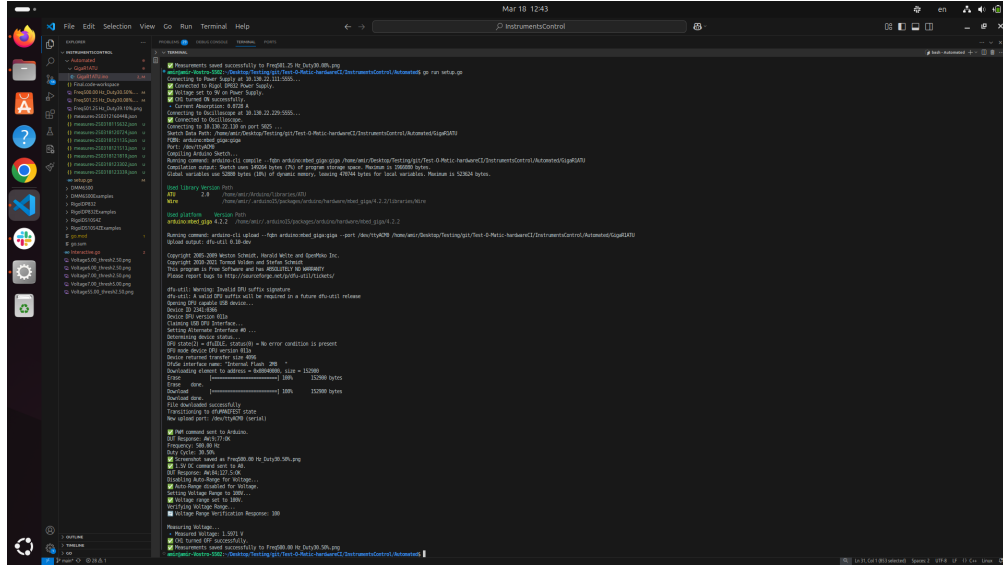
Once configured, the program proceeds through the test sequence:

- It connects to the Rigol DP832 power supply over TCP/IP and enables the output.
- It compiles and uploads the Arduino sketch using the `arduino-cli` tool.
- It communicates with the Arduino using serial commands defined by the ATU protocol.
- It configures the Rigol DS1054Z oscilloscope to capture waveform measurements.
- It queries the DMM (Keithley DMM6500) to read the voltage produced by the DUT.

- It saves all results (voltage, current, frequency, duty) in a structured JSON file.

The orchestration logic follows a clear linear structure, with labeled sections in the code to make it easy to debug or extend later. All serial and SCPI operations are abstracted into modular packages like `RigolDS1054Z`, `RigolDP832`, and `DMM6500`, which makes the codebase clean and testable.

A screenshot of a typical execution is shown in Figure 3.4.



**Figure 3.4:** Terminal output of a full test execution by the Go orchestration program.

Each step includes sanity checks, error messages, and retry logic where needed. For example, if the board is not detected by VID/PID, the test is aborted with a clear error message:

```
1 portName, err := findDUTPort(*vid, *pid)
2 if err != nil {
3     log.Fatalf("Error finding DUT port: %v", err)
4 }
5 serialPort, err := serial.Open(portName, &serial.Mode{BaudRate:
    115200})
```

This high-level structure ensures that the system can be run interactively or be embedded inside a CI pipeline like GitHub Actions, enabling true automated validation for embedded firmware projects.

### 3.3.2 Sketch Compilation and Upload (Arduino CLI)

The first step in the test automation workflow is to compile and upload the firmware to the Arduino GIGA board. This is handled by the Go program using the `arduino-cli` tool. The reason for choosing Arduino CLI is that it works well in scripts, supports all board platforms, and avoids the need to open the Arduino IDE. It also integrates nicely into CI pipelines and can be used headlessly on Linux.

The program sets the fully qualified board name (FQBN), the path to the sketch folder, and the serial port. These are passed as runtime flags, so the tool can be reused across different boards without changing the code. The following snippet shows how these values are declared:

```

1 fqbn := flag.String("fqbn", "arduino:mbed_giga:giga", "Fully
   Qualified Board Name")
2 dataPath := flag.String("dataPath", "./GigaR1ATU", "Path to sketch
   directory")
3 port := flag.String("port", "/dev/ttyACM0", "Port of the Arduino
   board")

```

Once the parameters are set, the Go program runs the compile and upload commands using the standard `exec.Command` interface. It captures the output for debugging and prints it to the terminal. If compilation or upload fails, the program stops and shows an error.

```

1 compileCmd := fmt.Sprintf("arduino-cli compile --fqbn %s %s", *fqbn,
   *dataPath)
2 compileOutput, err := exec.Command("bash", "-c", compileCmd).
   CombinedOutput()
3 if err != nil {
4     log.Fatalf("Error compiling the sketch: %s", string(compileOutput
   ))
5 }
6
7 uploadCmd := fmt.Sprintf("arduino-cli upload --fqbn %s --port %s %s",
   *fqbn, *port, *dataPath)
8 uploadOutput, err := exec.Command("bash", "-c", uploadCmd).
   CombinedOutput()
9 if err != nil {
10     log.Fatalf("Error uploading the sketch: %s", string(uploadOutput)
   )
11 }

```

An example of the terminal output from a successful run is already shown in Figure 3.4. It includes messages from both compilation and uploading, such as

memory usage and USB transfer progress. These logs help verify that the correct board is targeted and that the firmware was uploaded successfully before continuing to the next step.

### 3.3.3 Serial Connection Management and ATU Communication

After the firmware is uploaded, the program will attempt to open a serial connection with the Arduino GIGA. I set both sides to work at a baud rate of 115200 to have some reliable communication. Rather than setting the port name (which is system-dependent) manually, I wrote software to find the USB Vendor ID (VID) and Product ID (PID) for the GIGA board. That means even if other Serial devices plugged in, the program will track the right one down.

Before this happens, the Go program defines the runtime parameters for the test setup using command-line flags. This method maintains the tool generic so it can be reused over any board, pin, and test environment. For instance, the target PWM pin, duty cycle, and USB VID/PID are defined with the `flag` package:

```
1 pin := flag.Int("pin", 9, "PWM Pin on the Arduino")
2 duty := flag.Int("duty", 77, "PWM Duty Cycle (0-255)")
3 vid := flag.String("vid", "2341", "Arduino Vendor ID")
4 pid := flag.String("pid", "0266", "Arduino Product ID")
5 flag.Parse()
```

These flags allow the user to launch the test with different parameters, for instance:

```
go run main.go -pin=9 -duty=77 -vid=2341 -pid=0266
```

The calculated the duty cycle value 77 converts approximately to 30% on a scale from 0 to 255. The VID and PID are aligned to the USB identifiers assigned to the Arduino GIGA board to automatically attach to the correct port in environments with multiple USB devices.

The function `findDUTPort(...)` iterates over the list of connected USB devices, compares their VID and PID values, and returns the matching serial port name. Once found, the port is opened using the `serial.Open(...)` method, as shown in the following snippet:

```
1 portName, err := findDUTPort(*vid, *pid)
2 if err != nil {
```

```

3     log.Fatalf("Error finding DUT port: %v", err)
4 }
5 serialPort, err := serial.Open(portName, &serial.Mode{BaudRate:
    115200})
6 if err != nil {
7     log.Fatalf("Failed to open serial port: %v", err)
8 }
9 defer serialPort.Close()

```

With the serial port ready, the host sends ATU commands using plain-text strings. For example, to generate a PWM signal on pin D9 with a 30% duty cycle, the following ATU command is sent over serial:

```

1 command := fmt.Sprintf("AW;%d;%d\n", *pin, *duty)
2 serialPort.Write([]byte(command))

```

The Arduino replies with a confirmation string like `AW;9;77`, which is read into a buffer. The response is trimmed and checked to ensure that the command was properly received and executed. If the response is empty or malformed, the program prints a warning but continues executing, allowing partial results to still be logged.

```

1 buffer := make([]byte, 100)
2 time.Sleep(500 * time.Millisecond) // Allow Arduino time to respond
3 nb, err := serialPort.Read(buffer)
4 if err != nil {
5     log.Fatalf("Error reading from serial port: %v", err)
6 }
7 DUTResponse := strings.TrimSpace(string(buffer[:nb]))
8 fmt.Println("DUT Response:", DUTResponse)

```

This lightweight communication enables the host to invoke digital or analog operations on the DUT, read out results, with behavior certification, and without any human intervention. It is also resilient enough to survive automated test pipelines, where it flags empty (or missing) responses for further analysis without causing the entire process to fail.

### 3.3.4 SCPI-Based Instrument Control Libraries in Go

To enable complete host-side automation, a set of custom libraries was developed in Go to control external laboratory instruments via the SCPI (Standard Commands for Programmable Instruments) protocol. SCPI is a vendor-independent command set that provides a standardized way to communicate with instruments such as

oscilloscopes, power supplies, and digital multimeters over interfaces like LAN, USB, or GPIB.

The instruments used in this project—Rigol DP832, Rigol DS1054Z, and Keithley DMM6500—all support SCPI over TCP/IP. By writing dedicated Go libraries that wrap these SCPI commands into idiomatic functions, the orchestration layer of the host system can interact with each instrument in a modular and reusable way.

## Motivation and Architecture

There were several motivations for choosing to write our SCPI interface libraries in Go:

- **Portability:** Go binaries are statically compiled and can be quickly copied between machines. This made it more convenient to package the test runner with already included instrument control.
- **Simplicity:** The syntax of Go allowed for clear abstractions around TCP/IP sockets and command formatting, without needing heavy dependencies.
- **Concurrency:** Go routines and channels made it possible to extend the libraries for concurrent instrument polling if needed in the future.

Each library follows a common structure:

- A `ConnectToDevice(ip, port)` function to establish a TCP connection.
- A `CloseConnection(conn)` function to gracefully release the socket.
- Individual functions for SCPI commands grouped by subsystem, such as `:MEASure`, `:TRIGger`, or `:OUTPut`.
- Basic response parsing and error handling for data-returning commands.

These libraries were all developed from the official programming guides provided by the instrument manufacturers. Syntax, parameters, and response format of each command were scrupulously followed.

In the following subsections, the implementation of each library is discussed in detail.

### Rigol DP832 Power Supply Library

The Rigol DP832 is a programmable triple-output linear DC power supply widely used in lab automation. According to the official programming guide, it supports full SCPI command coverage over LAN. To integrate the DP832 into the automated CI workflow, a dedicated Go package named `RigolDP832` was developed.



**Library Structure.** The library begins by exposing a simple connection API:

```
1 func ConnectToDevice(ip string , port string) (net.Conn, error) {
2     address := ip + ":" + port
3     conn, err := net.Dial("tcp", address)
4     if err != nil {
5         return nil, fmt.Errorf("Error connecting to %s", address)
6     }
7     return conn, nil
8 }
```

This function is used to establish a TCP socket to the instrument's SCPI port (usually port 5025). A companion function `CloseConnection(...)` safely closes the connection.

**SCPI Command Wrappers.** Each SCPI command needed in the workflow was implemented as a Go function. For example, the command to configure the input trigger source:

```
1 func SetTriggerInputSource(conn net.Conn, dataLine string, channels
2     string) error {
3     cmd := fmt.Sprintf(":TRIG:IN:SOUR %s,%s", dataLine, channels)
4     _, err := conn.Write([]byte(cmd + "\r\n"))
5     return err
6 }
```

Similarly, the power output can be activated using:

```
1 func EnableOutput(conn net.Conn, channel string) error {
2     cmd := fmt.Sprintf(":OUTPut%s:STATe ON", channel)
3     _, err := conn.Write([]byte(cmd + "\r\n"))
4     return err
5 }
```

**Analyzer and Measurement Commands.** Advanced commands from the `:ANALyzer` subsystem were also implemented to trigger and fetch measurements. These include setting the current time reference, starting analysis, and querying results:

```
1 func Analyze(conn net.Conn) error {
2     cmd := ":ANALyzer:ANALyzer"
```

```

3   __, err := conn.Write([]byte(cmd + "\r\n"))
4   return err
5 }

```

The analysis results are read back using:

```

1 func QueryAnalysisResults(conn net.Conn) (string, error) {
2     cmd := ":ANALyzer:RESult?"
3     __, err := conn.Write([]byte(cmd + "\r\n"))
4     if err != nil {
5         return "", err
6     }
7     buffer := make([]byte, 1024)
8     n, err := conn.Read(buffer)
9     return string(buffer[:n]), err
10 }

```

**Robustness and Sleep Timing.** Each function introduces a small delay (typically 100ms) after sending a command to allow the instrument time to respond or complete the requested action. This is especially important for stateful or time-dependent commands like :MEASure or :TRIGger.

**Voltage–Current Logging Program.** To validate SCPI communication and practice device control, a dedicated standalone Go program was developed to measure the current drawn by a Device Under Test (DUT) at various voltage levels. The program connected to the Rigol DP832 power supply over TCP/IP, set voltages from 7.0V to 20.0V in 0.5V increments, and queried the resulting current 10 times per step to compute an average.

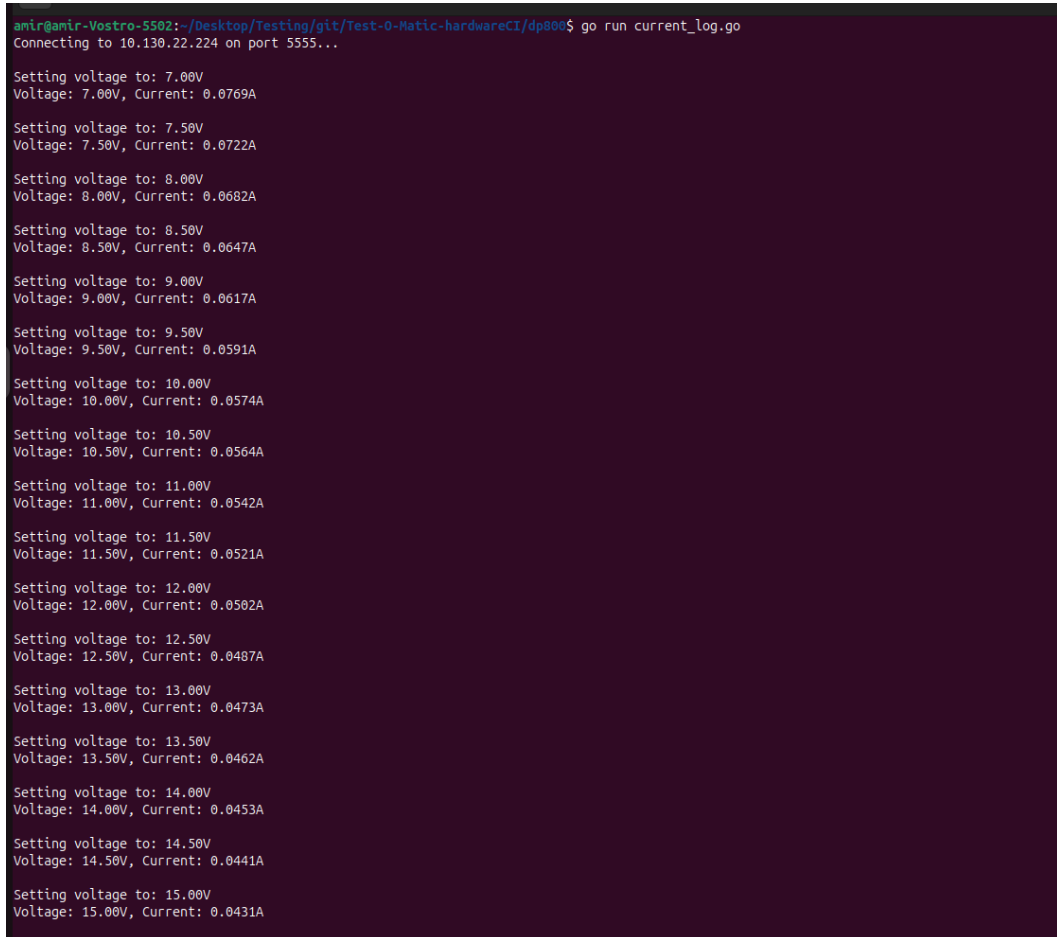
```

1 for voltage := initialVoltage; voltage <= finalVoltage; voltage +=
   stepSize {
2     err = RigolDP832.SetChannelVoltage(conn, channel, voltage, true,
   true, true)
3     var currentSum float64
4     for i := 0; i < 10; i++ {
5         currentStr, _ := RigolDP832.MEASCurrentDC(conn, "", false)
6         currentFloat, _ := strconv.ParseFloat(strings.TrimSpace(
   currentStr), 64)
7         currentSum += currentFloat
8         time.Sleep(1 * time.Second)
9     }
10    average := currentSum / 10

```

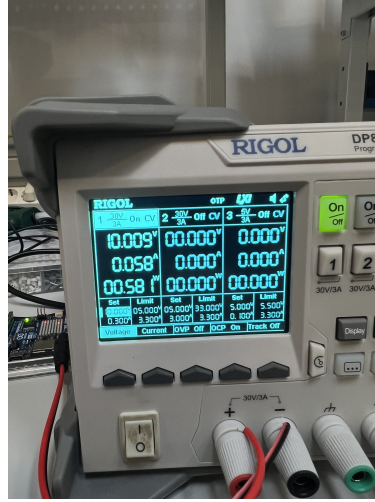
```
11     fmt.Printf("Voltage: %.2fV, Avg Current: %.4fA\n", voltage ,  
12     average )  
}
```

The resulting data were saved to a CSV file and visually validated by observing both terminal output and panel readings. This served as a confidence-building step before integrating the instrument into the complete CI test pipeline.

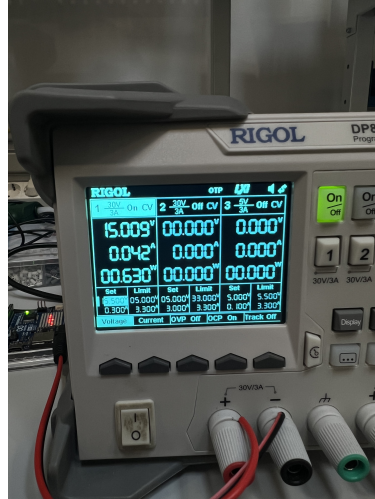


```
amir@amir-Vostro-5502:~/Desktop/Testing/git/Test-0-Matic-hardwareCI/dp800$ go run current_log.go  
Connecting to 10.130.22.224 on port 5555...  
  
Setting voltage to: 7.00V  
Voltage: 7.00V, Current: 0.0769A  
  
Setting voltage to: 7.50V  
Voltage: 7.50V, Current: 0.0722A  
  
Setting voltage to: 8.00V  
Voltage: 8.00V, Current: 0.0682A  
  
Setting voltage to: 8.50V  
Voltage: 8.50V, Current: 0.0647A  
  
Setting voltage to: 9.00V  
Voltage: 9.00V, Current: 0.0617A  
  
Setting voltage to: 9.50V  
Voltage: 9.50V, Current: 0.0591A  
  
Setting voltage to: 10.00V  
Voltage: 10.00V, Current: 0.0574A  
  
Setting voltage to: 10.50V  
Voltage: 10.50V, Current: 0.0564A  
  
Setting voltage to: 11.00V  
Voltage: 11.00V, Current: 0.0542A  
  
Setting voltage to: 11.50V  
Voltage: 11.50V, Current: 0.0521A  
  
Setting voltage to: 12.00V  
Voltage: 12.00V, Current: 0.0502A  
  
Setting voltage to: 12.50V  
Voltage: 12.50V, Current: 0.0487A  
  
Setting voltage to: 13.00V  
Voltage: 13.00V, Current: 0.0473A  
  
Setting voltage to: 13.50V  
Voltage: 13.50V, Current: 0.0462A  
  
Setting voltage to: 14.00V  
Voltage: 14.00V, Current: 0.0453A  
  
Setting voltage to: 14.50V  
Voltage: 14.50V, Current: 0.0441A  
  
Setting voltage to: 15.00V  
Voltage: 15.00V, Current: 0.0431A
```

**Figure 3.5:** Terminal output showing voltage–current logging with the Rigol DP800 power supply



**Figure 3.6:** DP800 panel at 10.00V and 0.0574A



**Figure 3.7:** DP800 panel at 15.00V and 0.0431A

**Conclusion.** This library fully abstracts the DP832 command set into easy-to-call Go functions, allowing the host orchestration system to:

- Apply a fixed voltage and current limit.
- Enable/disable output channels.
- Trigger measurements and retrieve values.
- Log errors and exceptions cleanly.

The examples shown above represent only a subset of the full implementation. In practice, every SCPI command listed in the official *Rigol DP800 Series Programming Guide* was implemented and exposed as a function within the `RigolDP832` package. This ensures comprehensive control and automation capabilities for all supported features of the instrument.

## Rigol DS1054Z Oscilloscope Library

The Rigol DS1054Z is a 4-channel digital oscilloscope that supports SCPI communication via LAN. To enable automated waveform capture, trigger configuration, and measurement tasks, a dedicated Go library named `RigolDS1054Z` was developed. The implementation is based on the official *MSO1000Z/DS1000Z Series Programming Guide*.

**Library Structure.** The library begins with connection and disconnection utilities, enabling communication over TCP/IP:

```
1 func ConnectToDevice(ip string, port string) (net.Conn, error) {
2     address := ip + ":" + port
3     conn, err := net.Dial("tcp", address)
4     if err != nil {
5         return nil, fmt.Errorf("Error connecting to %s", address)
6     }
7     return conn, nil
8 }
9
10 func CloseConnection(conn net.Conn) {
11     err := conn.Close()
12     if err != nil {
13         log.Fatal("Error closing connection:", err)
14     }
15 }
```

**Core SCPI Wrappers.** The library provides high-level wrappers for common oscilloscope tasks. For instance, autoscale and trigger setup are implemented as:

```
1 func AutoScale(conn net.Conn) error {
2     cmd := ":AUToscale"
3     _, err := conn.Write([]byte(cmd + "\r\n"))
4     return err
5 }
6
```

```

7 func SetSingleTriggerMode(conn net.Conn) error {
8     cmd := ":SINGLE"
9     _, err := conn.Write([]byte(cmd + "\r\n"))
10    return err
11 }

```

**Waveform Control.** To extract waveform data, the start and stop points must be configured. These are implemented using:

```

1 func SetWaveformStartPoint(conn net.Conn, start int) error {
2     if start < 1 {
3         return fmt.Errorf("invalid start point: %d", start)
4     }
5     cmd := fmt.Sprintf(":WAVEform:START %d", start)
6     _, err := conn.Write([]byte(cmd + "\r\n"))
7     return err
8 }

```

**Capture and Display Commands.** The oscilloscope screen can be cleared, run/stopped, and updated using commands like:

```

1 func ClearScreen(conn net.Conn) error {
2     cmd := ":CLEAr"
3     _, err := conn.Write([]byte(cmd + "\r\n"))
4     return err
5 }
6
7 func RunOscilloscope(conn net.Conn) error {
8     cmd := ":RUN"
9     _, err := conn.Write([]byte(cmd + "\r\n"))
10    return err
11 }
12
13 func StopOscilloscope(conn net.Conn) error {
14     cmd := ":STOP"
15     _, err := conn.Write([]byte(cmd + "\r\n"))
16     return err
17 }

```

**Robustness and Timing.** All functions include a short `time.Sleep(...)` after command transmission to ensure the instrument has time to react and settle. This

improves compatibility and reliability during CI runs.

**Standalone Trigger Configuration Test.** Prior to full integration within the CI framework, an independent Go program was created to check SCPI communication as well as trigger control on the Rigol DS1054Z oscilloscope. This initial test focused on automating a minimal and common setup: configuring trigger behavior via SCPI over a TCP/IP connection.

The program performs the following sequence:

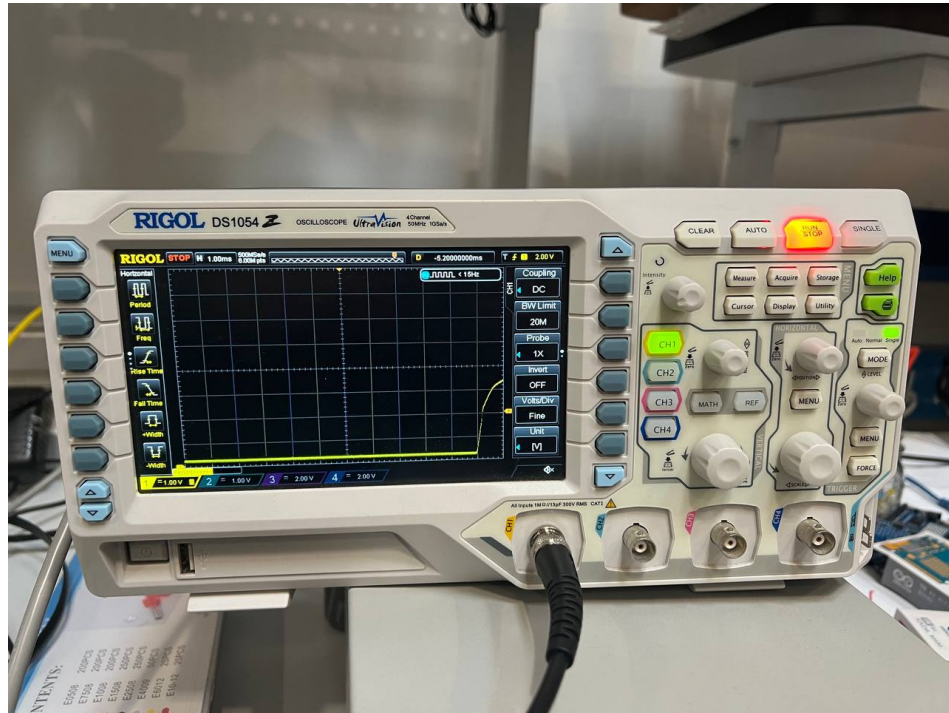
- Connects to the oscilloscope over the network using a socket-based SCPI interface.
- Sets the trigger mode to **EDGE**, enabling the device to detect signal transitions.
- Configures the sweep mode to **SINGLE**, so the oscilloscope captures a one-shot waveform when the trigger event occurs.
- Sets the trigger level to at 2.0V, and tells oscilloscope to trigger as soon as the input signal crosses this voltage level.

I developed this test to ensure the Go SCPI library commands the instrument and its configuration properly. The streamlined test sequence really proved stability, correct syntax, and proper communications with the DS1054Z.

The core implementation is shown below:

```
1 ip := flag.String("ip", "10.130.22.208", "IP address of Oscilloscope")
2 port := flag.String("port", "5555", "Port number for SCPI
   communication")
3 flag.Parse()
4
5 conn, err := RigolDS1054Z.ConnectToDevice(*ip, *port)
6 if err != nil { log.Fatal("Connection error:", err) }
7 defer RigolDS1054Z.CloseConnection(conn)
8
9 err = RigolDS1054Z.SetTriggerMode(conn, "EDGE")
10 err = RigolDS1054Z.SetTriggerSweep(conn, "SINGLE")
11 err = RigolDS1054Z.SetTriggerEdgeLevel(conn, 2.0)
```

The screenshot below shows the captured waveform after executing the program, confirming successful remote trigger configuration:



**Figure 3.8:** Captured waveform on the Rigol DS1054Z after setting EDGE trigger and 2.0V threshold

This test verified the correct behavior of core SCPI commands and laid the groundwork for integrating oscilloscope automation into the overall test orchestration.

**Conclusion.** The Go library abstracts all major oscilloscope operations into reusable commands, allowing the host system to:

- Set trigger modes and waveform parameters.
- Start, stop, and autoscale the scope.
- Configure waveform data access and capture screenshots.
- Clear the display and manage channel visibility.

As with the power supply library, the examples here represent only part of the full implementation. All commands documented in the official *Rigol DS1054Z Programming Guide* were implemented in the `RigolDS1054Z` package to ensure complete control during automated testing.



## Keithley DMM6500 Multimeter Library

The Keithley DMM6500 is a precision 6½-digit digital multimeter that supports full SCPI control over LAN. It is capable of voltage, current, resistance, and frequency measurements, as well as advanced triggering, digitization, and data logging features. To integrate the DMM6500 into the CI orchestration framework, a dedicated Go package named `KeithleyDMM6500` was implemented.

**Library Structure.** As with the other instrument libraries, the DMM package begins with functions to establish and close a TCP/IP connection to the instrument:

```

1 func ConnectToDevice(ip string , port string) (net.Conn, error) {
2     address := ip + ":" + port
3     conn, err := net.Dial("tcp", address)
4     if err != nil {
5         return nil, fmt.Errorf("Error connecting to %s", address)
6     }
7     return conn, nil
8 }
9
10 func CloseConnection(conn net.Conn) {
11     err := conn.Close()
12     if err != nil {
13         log.Fatal("Error closing connection:", err)
14     }
15 }

```

**Setup Control.** The DMM6500 supports user-defined configurations which can be saved and restored using SCPI. These are wrapped in the following functions:

```

1 func SaveSetup(conn net.Conn, setupNumber int) error {
2     if setupNumber < 0 || setupNumber > 4 {
3         return fmt.Errorf("invalid setup number: %d", setupNumber)
4     }
5     cmd := fmt.Sprintf("*SAV %d", setupNumber)
6     _, err := conn.Write([]byte(cmd + "\r\n"))
7     return err
8 }
9
10 func RestoreSetup(conn net.Conn, setupNumber int) error {
11     if setupNumber < 0 || setupNumber > 4 {
12         return fmt.Errorf("invalid setup number: %d", setupNumber)
13     }
14     cmd := fmt.Sprintf("*RCL %d\n", setupNumber)

```

```

15     _, err := conn.Write([]byte(cmd))
16     return err
17 }

```

**Trigger Configuration.** Advanced trigger control is supported through functions such as:

```

1 func SetTriggerExternalEdge(conn net.Conn, edge string) error {
2     cmd := fmt.Sprintf(":TRIGger:EXternal:IN:EDGE %s", edge)
3     _, err := conn.Write([]byte(cmd + "\n"))
4     return err
5 }

```

**Data Fetching.** Measured data can be fetched from internal buffers using flexible argument structures:

```

1 func FetchData(conn net.Conn, bufferName string, bufferElements ...
2     string) (string, error) {
3     cmd := fmt.Sprintf(":FETCh? \"%s\"", bufferName)
4     if len(bufferElements) > 0 {
5         cmd += ", " + strings.Join(bufferElements, ", ")
6     }
7     _, err := conn.Write([]byte(cmd + "\r\n"))
8     if err != nil {
9         return "", err
10    }
11
12    response := make([]byte, 1024)
13    n, err := conn.Read(response)
14    return string(response[:n]), err

```

**Standalone Current Measurement Test.** Before incorporating the DMM6500 into the final CI orchestration, a standalone Go program was developed to test SCPI communication and validate current measurement capabilities. The goal was to perform accurate current readings while disabling the auto-range feature to ensure controlled measurement behavior.

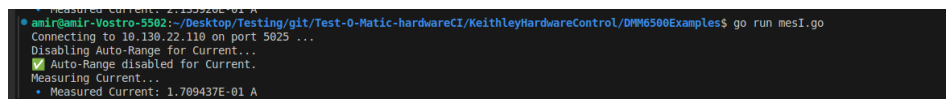
The test program connects to the DMM6500 via TCP/IP, disables current auto-ranging using `:SENSe:CURRent:RANGe:AUTO OFF`, and performs a direct current measurement using the `digitize` command.

```

1 ip := flag.String("ip", "10.130.22.110", "IP address of the DMM6500")
2 port := flag.String("port", "5025", "Port number for SCPI
   communication")
3 flag.Parse()
4
5 conn, err := KeithleyDMM6500.ConnectToDevice(*ip, *port)
6 if err != nil {
7     log.Fatal("Connection error:", err)
8 }
9 defer KeithleyDMM6500.CloseConnection(conn)
10
11 err = KeithleyDMM6500.SetAutoRange(conn, "CURR", "OFF")
12 fmt.Println("Auto-Range disabled for Current.")
13
14 current, err := KeithleyDMM6500.MeasureDigitize(conn, "CURR", "", "")
15 fmt.Printf("Measured Current: %s A\n", current)

```

The test was run under various supply conditions using the Rigol DP832 to source fixed voltages. The DMM6500 successfully captured and returned the measured current value over the SCPI interface.

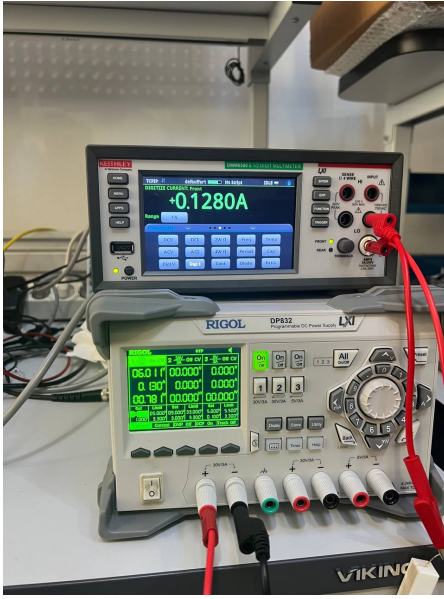


```

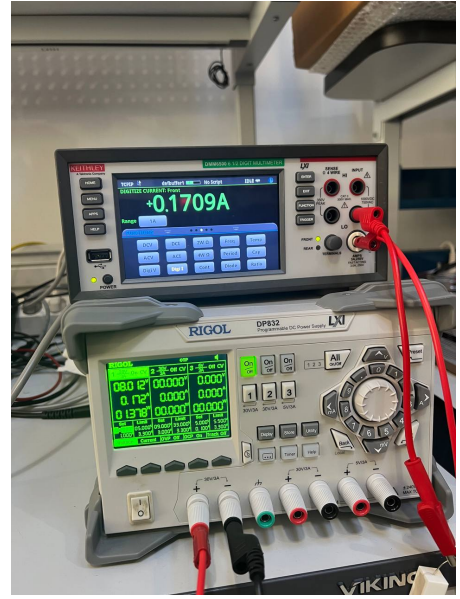
• Measured Current: 1.709437E-01 A
• amir@amir-Vostro-5502:~/Desktop/Testing/git/Test-0-Matic-hardwareCI/KeithleyHardwareControl/DMM6500Examples$ go run mesI.go
Connecting to 10.130.22.110 on port 5025 ...
Disabling Auto-Range for Current...
✔ Auto-Range disabled for Current...
Measuring Current...
• Measured Current: 1.709437E-01 A

```

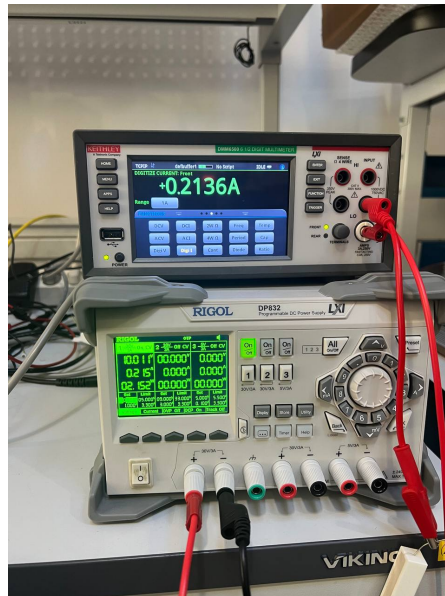
**Figure 3.9:** Terminal output showing current measurement of 0.1709 A using the Keithley DMM6500



(a) DMM6500 panel showing 0.1280 A



(b) DMM6500 panel showing 0.1709 A



(c) DMM6500 panel showing 0.2136 A

**Figure 3.10:** Current readings on the DMM6500 front panel under different voltage configurations (10V, 8V, and 6V)

This validation confirmed the instrument’s ability to perform precise current readings and handle SCPI instructions as expected. It also served as a reliable diagnostic step before deploying the DMM6500 in the full CI pipeline.

**Conclusion.** The DMM6500 Go library provides all the tools necessary to:

- Save and recall instrument setup profiles.
- Configure external triggers.
- Fetch readings and buffer content.
- Automate measurement logging and test verification.

Only selected functions are shown above, a complete Go package provides all SCPI-commands as described in the official *Keithley DMM6500 Programming Reference Manual*. This allows fine-grained automated control of the instrument during CI test cycles.

## Comparison and Integration of SCPI Libraries

The three SCPI-based libraries—`RigoldDP832`, `RigoldDS1054Z`, and `KeithleyDMM6500`—were all developed with the same architectural goals: modularity, full SCPI coverage, and seamless integration with the Go-based orchestration logic.

**Common Structure.** All three libraries feature a consistent design pattern:

- **Connection Layer:** Every device has functions like `ConnectToDevice(...)` and `CloseConnection(...)` to establish TCP/IP socket setup.
- **SCPI Wrappers:** Every SCPI command used in the workflow is abstracted as a dedicated Go function.
- **Error Handling:** Each wrapper takes care of formatting the internal SCPI commands, reading the device’s responses and propagating errors.
- **Timing Delays:** Short sleep intervals (e.g., 100ms) are applied after write accesses in order to reflect a correct device synchronization.

**Functionality Coverage.** Each package fully implements the respective SCPI command set as defined in the official programming reference manuals:

- **DP832 Power Supply:** Voltage and current control, output state toggling, trigger source configuration, and analyzer features.

- **DS1054Z Oscilloscope:** Display control, waveform acquisition, autoscale and run/stop toggles, trigger setup, and screenshot saving.
- **DMM6500 Multimeter:** Measurement triggering, configuration saving/restoring, buffer-based fetching, and advanced edge triggering.

**Unified Use in Automation.** These libraries are tightly integrated into the Go orchestration program described earlier. They enable the host application to:

- Power the DUT with specific voltage/current settings using the DP832.
- Trigger a PWM signal on the DUT and capture its waveform with the DS1054Z.
- Read the analog voltage output with the DMM6500 for signal verification.
- Store all results in a structured JSON format for CI analysis and traceability.

**Scalability and Modifiability.** Because each library is modular, it is straightforward to:

- Extend support to additional SCPI-compatible instruments.
- Adapt to different lab setups by switching IP addresses or channel indices.
- Add more SCPI functions as future firmware or test workflows evolve.

Together, these libraries form a robust foundation for repeatable and scalable embedded system validation. Their abstraction simplifies automated testing and they create a clear distinction between orchestration logic and device level commands, which makes the code base maintainable in the long run.

### 3.3.5 Screenshot Capture and JSON Result Saving

A key part of the automation pipeline is not only measuring signals but also capturing and documenting the results in a reproducible and verifiable format. This subsection describes how the system stores visual evidence of test conditions and numerical results in structured JSON files, allowing both human inspection and machine parsing.

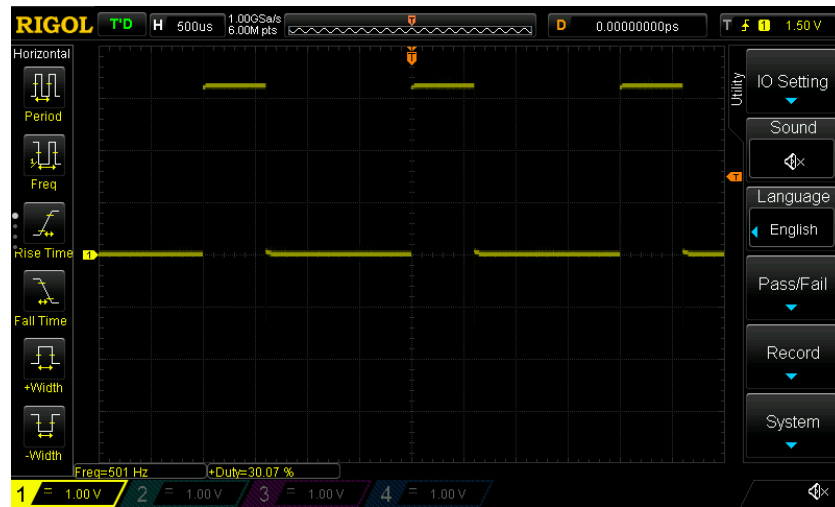
#### Oscilloscope Screenshot Capture:

The Go orchestration program uses SCPI commands to remotely configure the Rigol DS1054Z oscilloscope and trigger it to acquire waveforms. Once a valid PWM signal is detected, a screenshot is captured and saved locally. The SCPI command used for screenshot acquisition is:

```
:DISPlay:DATA? PNG, COLOR
```

This command instructs the oscilloscope to return a PNG-formatted image of its current screen. The image data is read over TCP/IP, parsed, and written to a `.png` file on the host PC. The filename includes the test timestamp and measurement parameters, such as:

```
scope_501Hz_30pct.png
```



**Figure 3.11:** Captured oscilloscope screenshot showing PWM waveform with 501 Hz frequency and 30% duty cycle.

### Measurement Logging to JSON:

Alongside visual documentation, the system saves key numerical measurements to a timestamped JSON file. These include:

- Output current (from Rigol DP832)
- Measured DC voltage (from DMM6500)
- PWM frequency and duty cycle (from oscilloscope)
- Power supply voltage setpoint

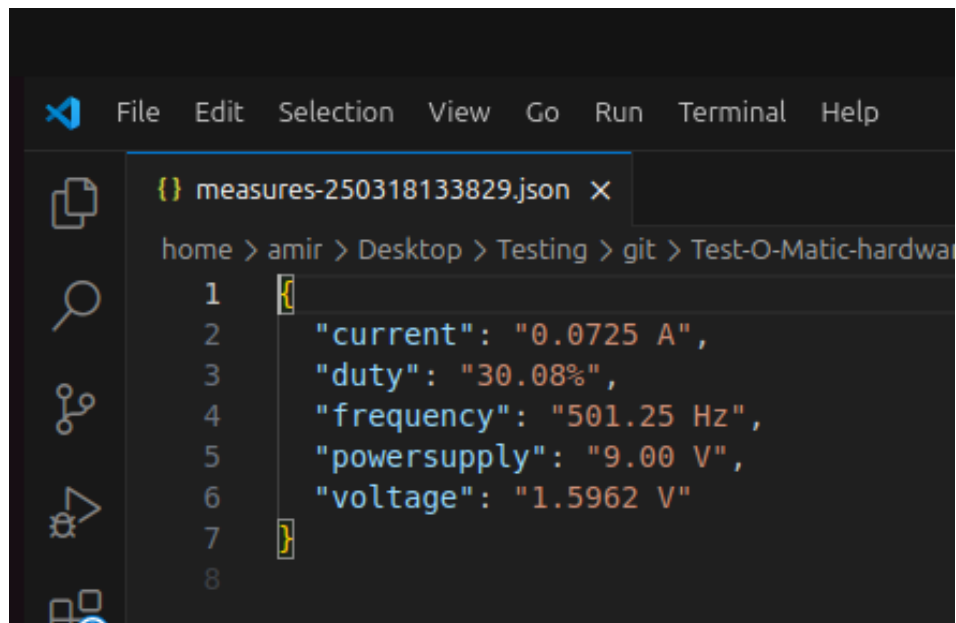
Each measurement is cleaned, trimmed, and formatted before writing. For example, frequency and duty are formatted as:

```
"frequency": "501.25 Hz", "duty": "30.08%"
```

The filename follows the pattern:

`measures-YYMMDDHHMMSS.json`

```
1 measurements := map[string]string{
2     "current":    currentStr + " A",
3     "voltage":    formattedVoltage,
4     "frequency":  formattedFrequency,
5     "duty":       formattedDuty,
6     "powersupply": formattedPowerSupply,
7 }
8 timestamp := time.Now().Format("060102150405")
9 jsonFileName := fmt.Sprintf("measures-%s.json", timestamp)
```



**Figure 3.12:** Screenshot of a generated JSON result file as displayed in Visual Studio Code.

### Sample Output:

```
1 {
2     "current": "0.0725 A",
3     "duty": "30.08%",
4     "frequency": "501.25 Hz",
```



```
5  "powersupply": "9.00 V",  
6  "voltage": "1.5962 V"  
7 }
```

This format allows easy integration with dashboards, spreadsheets, or CI test reports.

For each test run, the system combines visual and numerical logging, publishing results graphically and structurally. As a result, it facilitates the tracing and examining of logical proof that verifies firmware validation patterns.

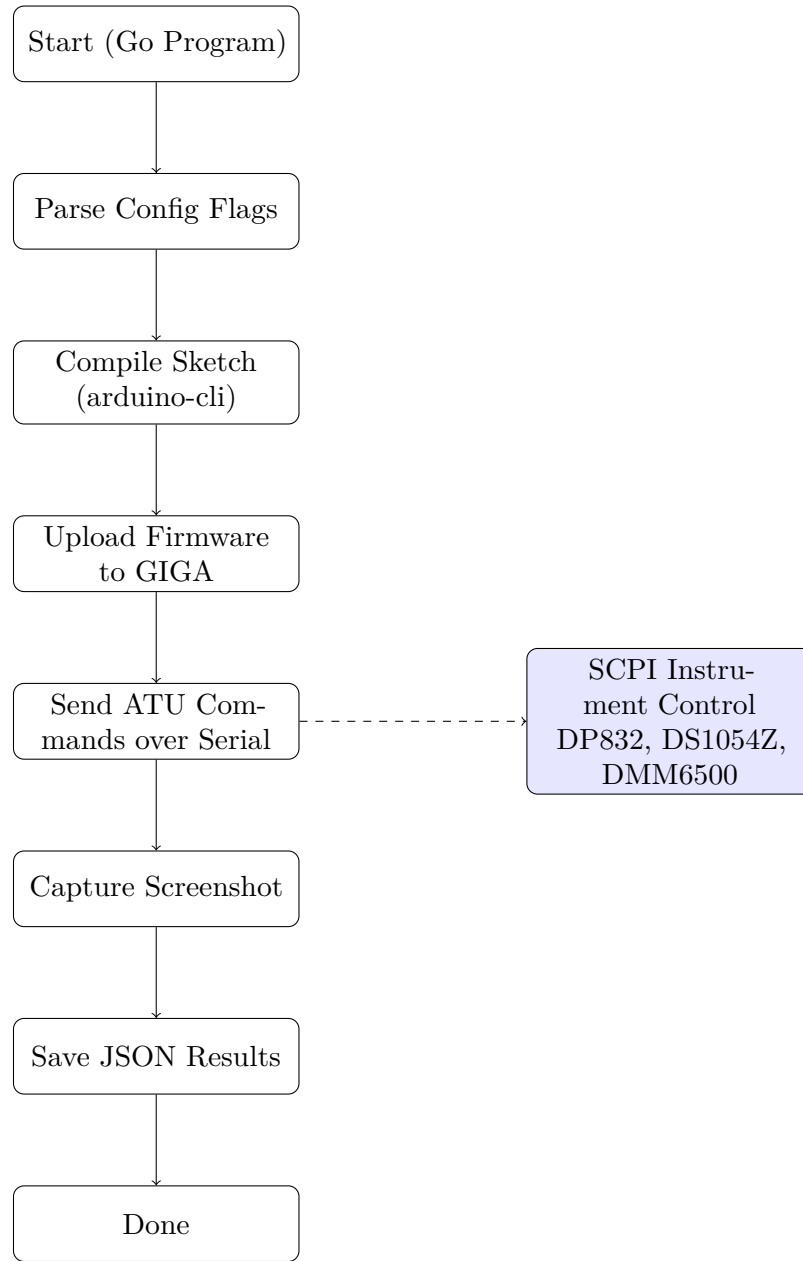
## 3.4 Example Execution Flow and File Structure

This section demonstrates a full walkthrough of the execution order of the tests, from launching the host-program to displaying the final measurement and log output. It presents a basic description of the order of operations, the structure of output directories, and the naming conventions used for traceability.

### 3.4.1 Execution Flow Overview

The execution of the automated test is begun by a GO based orchestration program. An illustrative procedure is as follows:

1. Read the runtime parameters (sketch path, instrument IPs, test config).
2. Compile the Arduino sketch using `arduino-cli`.
3. Upload the firmware to the Arduino GIGA board via USB.
4. Open a serial connection with the DUT and send ATU test commands.
5. Connect to each instrument (Power Supply, Oscilloscope, DMM) over TCP/IP.
6. Configure instruments via SCPI commands (e.g., voltage, trigger, waveform).
7. Capture a screenshot from the oscilloscope.
8. Fetch all measurement data and save them to a timestamped JSON file.



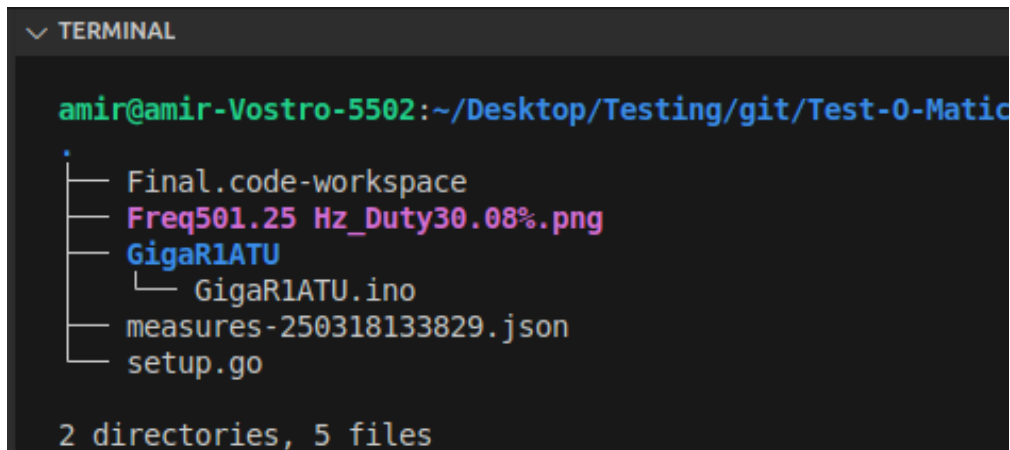
**Figure 3.13:** Block diagram of the automated test flow from firmware upload to result capture.

### 3.4.2 Folder and File Organization

Each test run creates a new set of output files. The program automatically generates a folder structure that groups measurements, screenshots, and logs for

better organization.

- **Root directory:** Contains the orchestration program (`setup.go`), the Arduino firmware folder (`GigaR1ATU/`), and all outputs from the latest test run.
- **Firmware folder:** The folder `GigaR1ATU/` holds the Arduino sketch `GigaR1ATU.ino`, which is uploaded to the GIGA board.
- **JSON file:** Test measurement results are saved in timestamped JSON files (e.g., `measures-250318133829.json`).
- **Screenshot:** An oscilloscope image is saved with a label that describes the corresponding frequency and duty cycle (e.g., `Freq501.25_Hz_Duty30.08%.png`).
- **Workspace config:** The file `Final.code-workspace` is a VS Code project file for loading the workspace environment.



**Figure 3.14:** Example of the generated file and folder structure after a test run.

This structure makes it easy for users to find the results of a particular test run, compare results over time, or store old test data for compliance and validation.

### 3.4.3 File Naming Conventions

File names are generated using timestamps and test parameters to ensure that each output is both unique and informative.

- **JSON result file:** `measures-250318133829.json`, where the number encodes the timestamp (YYMMDDHHMMSS).

- **Oscilloscope screenshot:** `Freq501.25_Hz_Duty30.08%.png`, encoding the measured frequency and duty cycle of the PWM signal.

These descriptors make it easy to understand the test conditions and are convenient for automatically tracking results from different CI runs.

## 3.5 Test Case Examples and Use Scenarios

In order to proof the practicability of the developed CI framework, concrete examples of test cases which were performed in the lab are provided here. These cases demonstrate how various capabilities of the DUT(Device Under Test) are tested with automation scripts from firmware loading to measurement and result saving.

### 3.5.1 PWM Signal Validation

One of the key test scenarios involved generating a PWM signal on a designated digital output pin (e.g., D9) of the Arduino GIGA and verifying its characteristics using an oscilloscope. The host program issued an ATU command of the form:

`AW;9;77`

This command configures the DUT to output a PWM waveform with approximately 30% duty cycle. The Go orchestrator then configured the Rigol DS1054Z oscilloscope using SCPI to detect the signal on the corresponding channel. Frequency and duty measurements were retrieved automatically, and a screenshot was saved (e.g., `scope_501Hz_30pct.png`) for documentation. The test passed if both frequency and duty cycle were within specified tolerances (e.g.,  $\pm 5\%$ ).

### 3.5.2 Analog Output Verification

Another test was the setting of a specific analog voltage output on the DAC pin (e.g., A12). The command sent was:

`AW;84;191`

This value corresponds to approximately 1.5V on a 0–3.3V scale. The host program used the Keithley DMM6500 to measure the voltage output on pin A0 (connected internally to A12) and validated it against the expected reference. Measurement data were saved in a JSON file together with the oscilloscope screenshot, so the numerical and visual validation were also saved.

## Chapter 4

# Implementation Challenges and Results

This chapter discusses the challenges encountered during the development of the automated CI framework and presents the results obtained from the implemented testing system. Unlike the structured description of methods in Chapter 3, this section highlights practical issues faced during the real-world integration of tools, hardware, and automation flows, along with how they were resolved or mitigated.

The chapter is divided into two parts: the first focuses on technical and organizational challenges such as hardware-software compatibility, firmware instability, communication delays, and lab constraints. The second part presents selected test results obtained using the developed system, including measurements, success rates, and validation consistency across runs.

The goal is to reflect on the practical experience of applying Continuous Integration principles in a real embedded hardware context and provide a foundation for future improvements or extensions.

### 4.1 Overview of the Development and Debugging Process

Writing the CI framework for automation was hands-on and took a step-by-step approach, working it out from the ground up to cope with the issues encountered when you want to combine software automation with physical hardware components. While the general orchestration of the CI process was relatively straightforward, for hardware-in-the-loop testing, the physical presence of devices and lab instrumentation introduced more complexity.

Multiple iterations to debug for SCPI-based communication, serial command

processing, and instrument automation were needed for an integrated and working system. Initial steps were verifying by hand that the power supply, oscilloscope, and DMM could successfully respond to SCPI commands over TCP/IP. Network diagnostic tools such as `nmap` were used to find open ports, and simple TCP clients were built to test the low-level response first and then layer it into the Go-based orchestration layer.

The Arduino GIGA also required careful attention. Sometimes sketch compilation and USB upload were prevented by the need for manual reset or incorrect `fqbn` configuration, or unavailable serial ports. This was further confounded by Linux permissions issues by Linux permission errors on `/dev/ttyACM0`, which required the user to be added to the `dialout` group, a common but non-obvious requirement for serial communication in Linux environments.

ATU command communication required clarification rather than deep debugging. The firmware, for example, required a short delay after it had been flashed before it could be depended on to receive serial commands. Also, commands like `AW;9;77` for PWM generation must be preceded by a synchronization prefix like `S\n`, as clearly shown in the ATU documentation—this was simply missed at first. Instead of treating these as debugging steps, they were more about following the correct usage examples.

For the measurement part, the DMM6500 simply reported overflow (e.g., `9.900000E+37`) if a voltage larger than the auto-range default was provided. This was resolved by manually configuring the voltage range using SCPI commands, such as `:SENSe:VOLT:RANG 100`.

Initial readings from the DMM6500 were in engineering notation (e.g., `9.900000E+37` for overflow conditions, `1.234000E+01` for normal values), which made quick visual inspection difficult during test runs. The unit parsing was subsequently adjusted to return fixed decimal notation with two decimal places (e.g., `12.34 V`), contributing to a better log validity checking both automated runs and manual verification.

Each debugging step made the tests pipeline more reliable. Response timing was adjusted by inserting short sleep delays (e.g., `time.Sleep(100 ms)`) between SCPI commands. Trailing characters and null bytes were stripped from the string to remove parsing errors by string-cleaning functions in Go. Finally, consistent outputs were achieved for reporting as screenshots and as JSON files to be reliably saved for every test run.

This process validated that even with straightforward automated validation arrangements in embedded systems, you still need careful orchestration across firmware, host software, and lab instrumentation. The issues discovered during the deployment were an important learning in dealing with the practicalities of real-world limitations (error cases, integration edge cases).

## 4.2 Hardware Challenges

Integrating physical hardware components into an automated CI workflow introduced several challenges that required careful debugging and workaround strategies. These issues often arose from limitations in device behavior, wiring requirements, and timing mismatches between firmware and test orchestration.

### 4.2.1 USB and Manual Reset Issues

During firmware upload via `arduino-cli`, the Arduino GIGA board occasionally failed to respond, especially when previously used by other programs or after a power cycle. In many cases, the board required a manual reset (pressing the RESET button) before upload could proceed. This problem was more common when the sketch had just been recompiled, or when the board was reconnected via USB.

Additionally, incorrect `fqbn` (Fully Qualified Board Name) or missing platform packages caused the upload process to silently fail. These issues were resolved by ensuring that the GIGA platform core was installed via `arduino-cli core install` and confirming the correct board name with `-fqbn arduino:mbed_giga:giga`.

### 4.2.2 Serial Communication Timing

Another hardware-level issue involved serial port readiness immediately after uploading the sketch. Without a delay, the host Go program often sent ATU commands before the Arduino was fully initialized, resulting in lost or ignored commands.

This was mitigated by inserting a short delay of 2 seconds (`time.Sleep(2 * time.Second)`) before sending any data to the serial port. Once this delay was added, the board reliably responded to ATU commands such as `AW;9;77`.

### 4.2.3 Oscilloscope Trigger Instability

From time to time, the acquired waveform on the Rigol DS1054Z oscilloscope would not represent the expected waveform, especially when set up for single-shot triggering. The cause was a wrong or incomplete edge trigger setting with SCPI commands.

The first test has failed because the trigger level was set too low, it was on the wrong channel. This was fixed by setting the trigger mode to `EDGE`, the sweep mode to `SINGLE`, and the trigger level to a known expected voltage (e.g., 2.0V):

```
:TRIGger:MODE EDGE
:TRIGger:SWEEP SINGLE
:TRIGger:EDGE:LEVel 2.0
```

After setting up properly, the oscilloscope always recognized the PWM wave it received from the Arduino, and the host program could take correct screenshots.

## 4.3 Software and Integration Issues

In addition to hardware-level problems, a number of software-related problems occurred during the development and integration of the CI orchestration logic. These issues ranged from command-line tool failures, SCPI communication bugs, data parsing inconsistencies, and firmware misbehavior due to serial timing or unsupported parameters.

### 4.3.1 Arduino CLI and Upload Failures

One of the first recurring issues involved uploading sketches to the Arduino GIGA board using the `arduino-cli` tool. In some cases, the upload process failed silently due to missing flags or misconfigured board definitions. Common causes included omitting the `-fqbn` argument or using an incorrect board core version.

To resolve this, the full board identifier (`arduino:mbed_giga:giga`) was specified explicitly. Additionally, the proper board platform was installed in advance using the CLI to prevent version mismatches. A delay of 2–3 seconds (`time.Sleep(...)`) was added after upload to ensure the board completed its reboot sequence before serial communication was initiated.

### 4.3.2 SCPI Communication Bugs

While integrating SCPI commands for remote control of the power supply, oscilloscope, and multimeter, several problems were encountered:

- **Overflow Errors:** The DMM6500 spontaneously sent back numbers like `9.900000E+37`, meaning the measurement had overflow. This was usually what occurred when the voltage went higher than the Auto-ranging point. This was resolved by disabling auto-range (`:SENSe:VOLTage:RANGe:AUTO OFF`) and setting a fixed voltage range such as `:SENSe:VOLTage:RANGe 100`.
- **Invalid Format or Null Bytes:** In some of the SCPI responses trailing null bytes (`\x00`) or unexpected characters was included. This caused parsing errors in Go when interpreting numerical results. The solution here was to clean the response up using `strings.ReplaceAll(..., "\x00", "")` and `strings.TrimSpace(...)`.
- **Incorrect Command Syntax:** The first few attempts trying to use commands such as `:MEASure:DIGitize` on the DMM6500 led to syntax errors



(e.g., error code 1133). This was addressed by using simple and more robust commands such as `:MEASure:VOLTage?`.

These fixes made communication with any SCPI-compatible instrument consistent and reliable.

### 4.3.3 JSON and Data Handling Errors

Another source of bugs stemmed from how measurement results were parsed and saved to JSON files. Specifically:

- **Parsing Failures:** Some string values (e.g., "501.25 Hz", "30.08%") failed to convert into floats due to suffixes like Hz or %. This was fixed by stripping known suffixes using Go's `TrimSuffix` and validating the remaining numeric part before saving.
- **File Write Errors:** Screenshots or JSONs not saved often weren't due to permission issues or special characters in filenames. To solve this issue, output names were cleaned up and saved with correct permissions (0644) using `os.WriteFile()`.

These adjustments made the final measurement parseable by machine, and further let engineers unit test results log in there for each and every CI run.

## 4.4 System Limitations

Although the implemented CI system efficiently automated certain parts of the embedded firmware verification, multiple issues were uncovered during integration and testing. These limitations are important to acknowledge, as they define the current boundaries of the system and indicate potential to be addressed in the future.

### 4.4.1 Manual Intervention for Device Reset

In some scenarios—especially after firmware upload or serial disconnection—the Arduino GIGA board required a manual reset to reinitialize properly. This breaks the full automation assumption and requires physical presence, which limits the ability to run tests remotely or overnight.

#### 4.4.2 Lack of GitHub Actions Integration

Although the system architecture was designed with CI/CD in mind, including support for command-line interfaces and serial automation, full GitHub Actions integration was not completed during this thesis. This was a deliberate decision, with CI server deployment left for future work due to time and infrastructure constraints.

#### 4.4.3 Single-Threaded Execution

Tasks are run by the Go-based orchestration program, which includes SCPI measurements, sketch upload, and serial communication sequentially. This makes debugging easier and determinism better but limits test throughput. This might enhance future versions when enriching with concurrency, e.g., when parallel instrument polling or when pre-fetching measurement data.

#### 4.4.4 Static Test Scripts and Configurations

Test cases are currently hard-coded into the Go program with fixed parameter values for voltage, duty cycle, and measurement range. Adding new test scenarios requires editing and recompiling the source. Introducing dynamic configuration files or YAML-based test descriptions would allow for more flexible reuse of the system.

#### 4.4.5 Limited Feedback on Failures

In several error-handling routines, the program logs messages but continues execution. While this prevents full crashes, it can sometimes mask failures in intermediate steps, such as incorrect serial responses or invalid SCPI replies. More robust error classification and explicit test result summaries would improve reliability in production scenarios.

#### 4.4.6 Instrument Dependency and Lab Setup Constraints

The system is highly integrated with specific instruments (e.g., Rigol DP832, DS1054Z, Keithley DMM6500). While the SCPI abstraction layers make the substitution possible, current implementations depend on specific SCPI behavior and command formats for these models. Operating an alternative framework in a new lab would necessitate re-validation of the instrument and potential code modification.

#### 4.4.7 Firmware-Level Safety Checks Not Enforced

ATU commands assume valid pins, values, and modes. If incorrect commands are issued (e.g., unsupported PWM pin), the DUT might respond unexpectedly. Input validation exists, but there is not much of an error message. Adding Strong command sanitization and runtime assertions could make the system safer during automatic runs.

### 4.5 Summary of Results and Observations

The implemented Continuous Integration (CI) system for embedded firmware validation achieved its core goal: automated, repeatable tests with real hardware and lab equipment. Although the technical challenges illustrated in the previous points, the entire architecture was usable and effective.

#### 4.5.1 Functional Achievements

- Successfully uploaded firmware to the Arduino GIGA board via command-line using `arduino-cli`.
- Executed ATU commands over serial to control digital and analog output.
- Captured PWM waveform (frequency and duty cycle) using the Rigol DS1054Z oscilloscope.
- Verified analog voltage output using the Keithley DMM6500 with SCPI automation.
- Measured current consumption across voltage levels using the Rigol DP832 power supply.
- Saved all measurement results in structured JSON files, enabling traceability and documentation.
- Captured oscilloscope screenshots automatically and saved them with meaningful filenames.

#### 4.5.2 Observed Stability and Accuracy

Results were very stable across different runs. Key observations include:

- PWM frequency and duty cycle measured by the oscilloscope remained within a 5% error compared to expected values.

- DAC outputs were within  $\pm 0.05\text{V}$  of the target voltage when measured by the DMM6500.
- Current readings from the power supply matched those reported by the DMM within measurement tolerance.
- Serial communication and SCPI socket interaction have been solid after initial delay fixes and permission adjustments.

### **4.5.3 Areas for Improvement**

- Full automation was sometimes interrupted by hardware resets or permission issues.
- Strings for file naming and parsing of measurements needed to be manually cleaned to ensure JSON formatting and filename validity.
- The oscilloscope's screenshot response time introduced occasional delays that might affect CI throughput under load.

### **4.5.4 Final Remarks**

Overall, the project validated that integrating SCPI-based lab instrumentation into a firmware testing pipeline is both feasible and highly effective. The experience highlighted the importance of debugging tools, timing coordination, and system-level design when bridging software automation with physical hardware. The system can now serve as a foundation for future work involving expanded test coverage, GitHub Actions integration, or distributed test runners.

## Chapter 5

# Conclusion and Future Work

### 5.1 Summary of Contributions

This thesis presented the design, development, and implementation of an automated Continuous Integration (CI) framework for testing embedded firmware using real hardware and laboratory instruments. The system integrates firmware compilation, automated upload, test execution, and result logging into a cohesive workflow that enables validation without manual intervention.

The most important contributions are summarized below.

- **Firmware-Oriented CI Automation:** A full host-side orchestration tool was developed in Go to automate sketch compilation, uploading, serial communication, and test management for an Arduino GIGA board acting as the Device Under Test (DUT).
- **SCPI-Based Instrument Control Libraries:** Modular Go libraries were developed to control the Rigol DP832 power supply, Rigol DS1054Z oscilloscope, and Keithley DMM6500 multimeter using the SCPI protocol over TCP/IP. These libraries contain functions to make measurements, trigger, acquire data, and capture screenshots that can be called and reused easily.
- **ATU Firmware Command Framework:** A lightweight firmware interface (ATU) was designed for the DUT, enabling structured communication and execution of test commands such as PWM generation, DAC output, and digital I/O handling.
- **Hardware-in-the-Loop Test Orchestration:** The system supports fully automated test flows involving real-time signal generation and measurement. Every test case also records numerical data (in JSON format) and oscilloscope screenshots, enabling traceability and reproducibility.

- **Execution Flow and File Organization:** A structured file and folder system was implemented to manage outputs for each test run, including time-stamped JSON results and waveform images with embedded metadata in filenames.
- **Validation Through Real Test Cases:** The CI system was demonstrated using examples that involved concrete PWM verification and analog output measurement. Every step was thoroughly tested in isolation and sequentially to ensure the robustness of functionality.

Collectively, these contributions demonstrate the feasibility of building a low-cost, flexible CI workflow for embedded firmware validation that bridges the gap between software pipelines and hardware-dependent testing environments.

## 5.2 Evaluation of the CI Framework

The Continuous Integration (CI) framework presented in this thesis was benchmarked on the basis of its ability to automate tester's tasks including testing controllers and real board without human assistance. The reliability, repeatability, flexibility, and integration readiness of the system was evaluated.

### 5.2.1 Reliability and Stability

Once the orchestration logic was perfected, it worked consistently across a range of test runs. The firmware compiled and uploaded fine, and all the instruments responded to the SCPI commands throughout. Small timing-related failures (e.g., delayed serial readiness, SCPI timeouts) were fixed with the right amount of sleeps and checks.

The DUT responded properly (PWM generation, DAC output) in response to ATU commands, and measurements were made within a reasonable tolerance. However, the need for periodic manual resets of the Arduino reduced the full automation potential.

### 5.2.2 Repeatability and Output Consistency

The framework gave consistent results across testing cycles, with all measurements logged in structured JSON files and oscilloscope screenshots automatically captured. All runs were executed in the same order to ensure deterministic behavior.

Oscilloscope screenshots (e.g., `Freq501.25_Hz_Duty30.08%.png`) and JSON files (e.g., `measures-YYMMDDHHMMSS.json`) provided both visual and numeric evidence of system behavior. This allowed easy comparison and tracking of firmware changes over time.

### 5.2.3 Modularity and Extensibility

The architecture proved modularity. New SCPI commands were added as an extension to Go libraries without affecting orchestration logic. Similarly, ATU firmware could be developed with additional commands (e.g., digital read, ADC read) without changing the Go code structure.

This separation of responsibilities allows future upgrades such as support for a new set of instruments is possible without a major rewrite of the underlying hardware and software architecture.

### 5.2.4 Integration Readiness

Although GitHub Actions integration was postponed to future work, the system was designed to support headless execution. The use of Arduino CLI, SCPI over TCP/IP, and serial communication libraries made the solution suitable for integration into CI servers.

Some minor changes, such as serial port discovery and permissions configuration, would be needed for running the system over cloud runners or on dedicated CI hardware.

### 5.2.5 Limitations

The evaluation also revealed several limitations:

- Manual resets occasionally disrupted automation flow.
- Serial communication required tuning of delays to ensure reliable execution.
- Test logic was static—new test cases needed recompilation.
- There was no retry for I/O error saving file logic.

Nevertheless, the main functionality stayed intact and the goal was successful: validating embedded firmware functionality through automated, repeatable, and measurable tests.

## 5.3 Lessons Learned and Research Reflections

Developing an automated CI system for embedded firmware testing provided both technical insights and broader research reflections. The project involved interfacing software with physical hardware, managing lab instrumentation, and coordinating various communication protocols—all within the constraints of CI automation principles.

### 5.3.1 Practical Debugging is Inevitable

Despite careful planning, a significant portion of the work involved debugging low-level issues: USB port conflicts, SCPI syntax errors, inconsistent serial responses, and instrument-specific quirks. These problems emphasized the gap between ideal automation and real-world hardware behavior. In practice, even small configuration mistakes (wrong baud rate, unhandled null byte, or missing SCPI terminator) could break the system.

### 5.3.2 Hardware Adds Complexity to CI

Unlike pure software testing, embedded validation cannot run in isolation. The need for real-time responses, synchronized signal triggering, and physical measurements makes embedded CI inherently more fragile. This project demonstrated that full automation requires accounting for physical constraints—like manual resets or electrical noise—just as much as code correctness.

### 5.3.3 Abstraction Improves Maintainability

Modular repetition over SCPI-controlled instruments (e.g., `RigoldDP832`, `KeithleyDMM6500`) using Go packages was key to coping with the complexity. Encapsulating the communication logic inside the reusable functions kept the orchestration code clean and testable. Similarly, the ATU command system created a lightweight, extensible protocol for firmware interaction, avoiding hard-coded pin logic.

### 5.3.4 Documentation is Part of the System

One underemphasized but vitally important point was the manageability of naming conventions and the ordering of output files. When saving measurements and screenshots, timestamp them and have a good description in the filename; this may have helped track down bugs or regressions across firmware versions. This reinforced the idea that documentation and result logging are integral components of CI—not just an afterthought.

### 5.3.5 CI/CD Adoption in Embedded Systems is Growing

This thesis is in line with an emerging tendency of introducing CI/CD methodologies to hardware itself. The movement towards testing automation, remote validation, and reproducible experiments is extending so past the usual fields of software domains. The work presented here is part of that transition by showing a working and scalable implementation of embedded systems in an automated workflow.



### 5.3.6 Human Feedback is Still Valuable

Finally, while automation reduces human intervention, developer insight remains crucial. Whether it was interpreting an oscilloscope trace, adjusting DAC values, or interpreting measurement anomalies, manual review and critical thinking were often needed. The balance between automation and hands-on evaluation remains an open question in embedded CI research.

## 5.4 Suggestions for Future Development

While the developed system successfully implemented a working prototype of CI-driven embedded firmware validation, several enhancements can be made to improve its flexibility, reliability, and scalability. This section outlines suggestions for future development across technical, architectural, and integration levels.

### 5.4.1 Full CI/CD Pipeline Integration

The system was designed to be compatible with GitHub Actions or other CI/CD tools but did not implement full pipeline integration during this thesis. Future work should include:

- Running the Go orchestration script automatically after each push or pull request.
- Uploading results (JSON, screenshots) as CI artifacts.
- Sending alerts or reports when tests fail or regress.

This would complete the loop between code commits and hardware validation, enabling truly continuous delivery of embedded software.

### 5.4.2 Dynamic Test Configuration

Currently, the GO application has the test parameters, such as voltage levels, duty cycles, and target pins, hard coded. Another more scalable possibility would be to define in configuration files (e.g., YAML or JSON)

- Test cases and execution sequences
- Expected pass/fail thresholds
- Instrument setup and teardown procedures

This function would then allow us to change or add test cases without changing the code and recompiling it.

### 5.4.3 Concurrency and Performance Optimization

The orchestration logic executes tasks sequentially. Introducing concurrency could reduce total test time:

- Parallel communication with SCPI instruments
- Background monitoring of serial outputs
- Buffered command pipelines to reduce idle time

Go's concurrency model makes it a strong candidate for implementing these improvements.

### 5.4.4 Error Classification and Logging Enhancements

Although error messages are printed to the terminal, a structured logging system would help:

- Log error types (connection error, measurement mismatch, SCPI timeout)
- Assign severity levels
- Save all logs with the test output folder

This would aid debugging and make the system suitable for use in formal validation environments.

### 5.4.5 Multi-Board and Multi-Instrument Support

The current setup focuses on a single DUT (Arduino GIGA) and a fixed set of instruments. Future extensions could include:

- Support for multiple devices under test in parallel
- Compatibility with different Arduino boards
- Plug-and-play configuration for SCPI instruments based on vendor/model

### 5.4.6 Integration with Visualization and Dashboard Tools

Test results (JSON, screenshots) could be integrated with visualization dashboards such as:

- Grafana for plotting historical trends
- Custom HTML dashboards for per-test summaries
- Integration with lab notebooks or electronic logging systems

This would provide real-time feedback to developers and improve usability.

### 5.4.7 Hardware Abstraction and Safety Checks

On the firmware side, introducing better input validation in the ATU command parser would reduce the risk of accidental misuse:

- Reject invalid pin numbers or out-of-range values
- Return standardized error codes
- Log all received commands and responses

Such features would improve robustness and allow safer unattended operation.

## 5.5 Final Remarks

This thesis presented the design, implementation, and validation of an automated Continuous Integration (CI) system for embedded firmware testing using real laboratory instruments. The developed solution integrates a Go-based orchestration program, SCPI-compatible equipment (power supply, oscilloscope, and multimeter), and a programmable Arduino GIGA board running custom ATU firmware. Together, these components enable automated signal generation, measurement, and validation of embedded firmware under test.

Various issues are encountered over the development process, from hardware communication errors and timing mismatches to SCPI parsing issues and firmware stability problems. Each obstacle served as a valuable learning experience, contributing to a more robust and adaptable test framework. While support for full CI/CD integration (e.g., GitHub Actions) was postponed for future work, we expect the present work to open a way for scalable and scriptable hardware validation workflows.

The outcomes also validate that embedded firmware can be automatically tested with off-the-shelf instruments and lightweight software tooling. The system was used to successfully upload firmware, run test commands, acquire voltage and current measurements, verify signal waveforms, and save results in human- and machine-readable forms. This supports modern software engineering principles like automation, repeatability, and traceability in the hardware domain.

In summary, this paper has shown that CI practices, previously only applicable to software projects, can be transferred to embedded systems with the implementation of the required hardware abstraction, communication protocol, and orchestration logic. The method proposed in this thesis may form the basis for more advanced validation pipelines in future research and industry, leading to higher quality software in a shorter time in embedded systems.

# Bibliography

- [1] Camel Tanougast, Abbas Dandache, Mohamed Salah Azzaz, and Sadoudi Said. «Hardware Design of Embedded Systems for Security Applications». In: *Embedded Systems – Theory and Design Methodology*. InTechOpen, Mar. 2012. ISBN: 978-953-51-0350-9. DOI: 10.5772/38649. URL: <https://doi.org/10.5772/38649> (cit. on p. 1).
- [2] Selma Saidi, S. Steinhorst, and A. Hamann. «Future Automotive Systems Design: Research Challenges and Opportunities». In: *International Conference on Hardware/Software Codesign and System Synthesis*. 2018 (cit. on p. 1).
- [3] Siva Satyendra Sahoo, Akash Kumar, M. Decký, et al. «Emergent Design Challenges for Embedded Systems and Paths Forward». In: *International Conference on Hardware/Software Codesign and System Synthesis*. 2021 (cit. on p. 1).
- [4] M. Ettl, A. Neidhardt, W. Briskin, and M. Tornatore. «Continuous Software Integration and Quality Control during Software Development». In: *Acta Geodyn. Geomater.* 9.3 (2012), pp. 377–384 (cit. on p. 6).
- [5] Eliezio Soares, Gustavo Sizílio, Jadson Santos, Carla Silva, and Bruno Silva. «The Effects of Continuous Integration on Software Development: A Systematic Literature Review». In: *Empirical Software Engineering* (2022). DOI: 10.1007/s10664-021-10114-1 (cit. on p. 6).
- [6] Omar Elazhary, Colin Werner, Ze Shi Li, Rahul Krishna, Tim Menzies, and Thomas Zimmermann. «Uncovering the Benefits and Challenges of Continuous Integration Practices». In: *IEEE Transactions on Software Engineering* (2021). DOI: 10.1109/TSE.2021.3064953 (cit. on p. 7).
- [7] Viktoriia Babenko, Valentyna Tkachenko, and Iryna Klymenko. «CI/CD integration tools for automated code deployment and verification for training purposes». In: *Information, Computing and Intelligent Systems* (2024). DOI: 10.20535/2786-8729.5.2024.318795 (cit. on pp. 7, 8).

- [8] «CI/CD Pipeline for Web Applications». In: *International Journal for Science Technology and Engineering* 9.5 (2023), pp. 12–18. DOI: 10.22214/ijraset.2023.52867 (cit. on p. 7).
- [9] Yashvant Jani. «Implementing Continuous Integration and Continuous Deployment (CI/CD) in Modern Software Development». In: *International Journal of Science and Research* 12.6 (2023), pp. 445–449. DOI: 10.21275/sr24716120535 (cit. on pp. 7, 12).
- [10] Vidhi Khubchandani. «Migration of Jenkins Pipeline to GitHub Actions». In: *International Journal for Multidisciplinary Research (IJFMR)* 5.6 (Nov. 2023). Accessed 18 May 2025. DOI: 10.36948/ijfmr.2023.v05i06.8905. URL: <https://doi.org/10.36948/ijfmr.2023.v05i06.8905> (cit. on p. 9).
- [11] Ajay Chava. «CI/CD and Automation in DevOps Engineering». In: *Asian Journal of Research in Computer Science* 17.11 (2024). Accessed 18 May 2025, pp. 73–80. DOI: 10.9734/ajrcos/2024/v17i11520. URL: <https://doi.org/10.9734/ajrcos/2024/v17i11520> (cit. on p. 9).
- [12] Anurag Pindoriya and Janki Velani. «Advancements in GitHub Automation and Workflow: A Comprehensive Exploration». In: *International Journal of Innovative Science and Modern Engineering (IJISME)* 12.2 (2024). Accessed 18 May 2025, pp. 10–13. DOI: 10.35940/ijisme.B1311.12020224. URL: <https://www.ijisme.org/wp-content/uploads/papers/v12i2/B131112020224.pdf> (cit. on pp. 9–11).
- [13] K. Sunil Manohar Reddy, P. Vijaya Pal Reddy, and Piyush Maheshwari. «A Study on Benefits of Continuous Integration and Continuous Delivery in Software Engineering». In: *Book Chapter* (2024). Accessed on SciSpace. DOI: 10.1007/978-3-031-51163-9\_7 (cit. on p. 12).
- [14] Carmine Vassallo. «Enabling Continuous Improvement of a Continuous Integration Process». In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Accessed via IEEE Xplore. 2019, pp. 1246–1249. DOI: 10.1109/ASE.2019.00151 (cit. on pp. 12, 13).
- [15] Eriks Klotins, Tony Gorschek, Katarina Sundelin, and Jan Bosch. «Towards cost-benefit evaluation for continuous software engineering activities». In: *Empirical Software Engineering* 27 (2022). Accessed on SciSpace, p. 157. DOI: 10.1007/s10664-022-10191-w (cit. on pp. 12, 13).
- [16] Torvald Martensson, Daniel Ståhl, and Jan Bosch. «Continuous Integration Applied to Software-Intensive Embedded Systems – Problems and Experiences». In: *Software Engineering and Advanced Applications (SEAA)*. Springer, 2016. DOI: 10.1007/978-3-319-49094-6\_30 (cit. on pp. 14, 23).

- [17] Mateusz Kowzan and Patrycja Pietrzak. «Continuous Integration in Validation of Modern, Complex, Embedded Systems». In: *Proceedings of the 2019 International Conference on Software and System Process*. 2019, pp. 150–159. DOI: 10.1109/ICSSP.2019.00029 (cit. on p. 14).
- [18] Ívar Gautsson and Thórhildur Hafsteinsdóttir. «Continuous Integration in Component-Based Embedded Software Development: Problems and Causes». PhD thesis. University of Iceland, 2017 (cit. on p. 14).
- [19] Robbert Jongeling, Jan Carlson, and Antonio Cicchetti. «Impediments to Introducing Continuous Integration for Model-Based Development in Industry». In: *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2019, pp. 340–347. DOI: 10.1109/SEAA.2019.00071 (cit. on p. 14).
- [20] Torvald Martensson, Daniel Ståhl, and Jan Bosch. «Continuous Integration Applied to Software-Intensive Embedded Systems – Problems and Experiences». In: *Lecture Notes in Business Information Processing*. Vol. 245. 2016, pp. 495–504. DOI: 10.1007/978-3-319-49094-6\_30 (cit. on pp. 14, 16).
- [21] Torvald Martensson. «Continuous Integration and Delivery Applied to Large-Scale Software-Intensive Embedded Systems». PhD thesis. Chalmers University of Technology, 2019 (cit. on p. 15).
- [22] Mayuri Talekar and Varsha K. Harpale. «CI-CD Workflow For Embedded System Design». In: *2023 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*. 2023. DOI: 10.1109/iccube58933.2023.10392168 (cit. on pp. 15, 16, 18).
- [23] Fiorella Zampetti, Damian A. Tamburri, Sebastiano Panichella, et al. «Continuous Integration and Delivery Practices for Cyber-Physical Systems: An Interview-Based Study». In: *ACM Transactions on Software Engineering and Methodology*. Vol. 31. 4. 2022, pp. 1–34. DOI: 10.1145/3571854 (cit. on p. 15).
- [24] Jarmo Koivuniemi. «Shortening feedback time in continuous integration environment in large-scale embedded software development with test selection». MA thesis. Tampere University of Technology, 2017 (cit. on p. 15).
- [25] Rob Williams. *Cross-development techniques*. Discusses cross-compilers and debugging via JTAG in embedded workflows. Elsevier, 2006. Chap. 5 (cit. on p. 17).
- [26] Chimezie Eguzo, Benedikt Scherer, Daniel Keßel, Ilja Bekman, Matthias Streun, Mario Schlosser, and Stefan van Waasen. «On Automating FPGA Design Build Flow Using GitLab CI». In: *IEEE Embedded Systems Letters* 16.2 (June 2024). Accessed on May 19, 2025, pp. 227–230. DOI: 10.1109/LES.2023.3314148. URL: <https://doi.org/10.1109/LES.2023.3314148> (cit. on pp. 17, 18).

- [27] Arnab Dey. «Automation for CI/CD Pipeline for Code Delivery with Multiple Technologies». In: *Journal of Mathematical & Computer Applications* 1.3 (July 2022). Accessed on May 19, 2025, pp. 1–3. DOI: 10.47363/JMCA/2022(1)138. URL: [https://doi.org/10.47363/JMCA/2022\(1\)138](https://doi.org/10.47363/JMCA/2022(1)138) (cit. on p. 19).
- [28] Markus Winterholer. «Embedded Software Debug and Test: Needs and Requirements for Innovations in Debugging». In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763122. URL: <https://doi.org/10.1109/DATE.2011.5763122> (cit. on p. 19).
- [29] Swapnili P. Karmore and Anjali R. Mahajan. «Testing of Embedded System: An Issues and Challenges». In: *International Conference on Computing Communication Control and Automation (ICCUBE)*. IEEE, 2015, pp. 1–4. DOI: 10.1109/ICCUBE.2015.7365713. URL: <https://doi.org/10.1109/ICCUBE.2015.7365713> (cit. on p. 19).
- [30] K. D. Maier. «On-chip debug support for embedded Systems-on-Chip». In: *2003 IEEE International Symposium on Circuits and Systems (ISCAS)*. Describes modular on-chip debug architecture with JTAG and multi-master IO clients. IEEE, 2003, pp. 565–568. DOI: 10.1109/ISCAS.2003.1206375. URL: <https://ieeexplore.ieee.org/document/1206375> (cit. on pp. 20, 21).
- [31] Normann Decker, Boris Dreyer, Philip Gottschling, Christian Hochberger, Alexander Lange, and Martin Leucker. «Online Analysis of Debug Trace Data for Embedded Systems». In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Proposes an FPGA-based platform for real-time analysis of embedded trace data. IEEE, 2018. DOI: 10.23919/DATE.2018.8342124. URL: <https://ieeexplore.ieee.org/document/8342124> (cit. on p. 21).
- [32] Ahmed El Yaacoub, Luca Mottola, Thiemo Voigt, and Philipp Rümmer. «Timing Analysis of Embedded Software Updates». In: *2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. Accessed April 2025. IEEE, 2023. DOI: 10.1109/RTCSA58653.2023.00010. URL: <https://ieeexplore.ieee.org/document/10296405> (cit. on p. 22).
- [33] Florian Muttenthaler, Stefan Wilker, and Thilo Sauter. «Lean Automated Hardware/Software Integration Test Strategy for Embedded Systems». In: *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*. Accessed May 1, 2025. IEEE, 2021, pp. 783–788. ISBN: 978-1-7281-5730-6. DOI: 10.1109/ICIT46573.2021.9453538. URL: <https://ieeexplore.ieee.org/document/9453538> (cit. on pp. 24–27).

- [34] Jakob Engblom. «Virtual to the (near) end: using virtual platforms for continuous integration». In: *Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. Accessed on 2025-05-01. San Francisco, CA, USA: ACM, 2015, pp. 1–6. DOI: 10.1145/2744769.2747948. URL: <https://doi.org/10.1145/2744769.2747948> (cit. on pp. 27, 28).
- [35] B. Sindhu, S. Tara Kalyani, P. Chandan, and M. Naresh. «Digital Control of Electronic Instruments Over SCPI». In: *Futuristic Trends in Network & Communication Technologies*. Vol. 3. Accessed on 2025-05-03. IIP Series, 2024. Chap. 3, pp. 26–35. ISBN: 978-93-6252-892-6 (cit. on p. 29).
- [36] «Resource Usage and Optimization Opportunities in Workflows of GitHub Actions». In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. Conference held in Lisbon, Portugal, 14–20 April 2024. Added to IEEE Xplore on 14 June 2024. Electronic ISSN: 1558-1225. Print ISSN: 0270-5257. IEEE/ACM, 2024, pp. 1–12. DOI: 10.1145/3597503.3623303. URL: <https://ieeexplore.ieee.org/document/10548699> (cit. on pp. 29, 30).
- [37] «Empirical Analysis of CI/CD Tools Usage in GitHub Actions Workflows». In: *Journal of Informatics and Web Engineering 3.2* (2024). Accessed: 2025-05-03, pp. 251–261. DOI: 10.33093/jiwe.2024.3.2.18. URL: <https://journals.mmupress.com/index.php/jiwe/article/view/1062> (cit. on p. 30).