VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Liyuan Liu

AUTOMATED TESTING FRAMEWORK FOR EMBEDDED COMMUNICATION SYSTEMS

A Case Study on Automated Testing of CANopen Integration in HMI Panel Software

Technology and Communication

2025

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Liyuan Liu
Title	AUTOMATED TESTING FRAMEWORK FOR EMBEDDED COMMUNICATION Systems: A Case Study on Automated Testing of CANopen Integration in HMI Panel Software
Year	2025
Language	English
Pages	54
Name of Supervisor	Chao Gao

This thesis presents the design and implementation of an automated testing framework for validating the CANopen communication application in an advanced Human-Machine Interface panel software. The framework ensures the reliability of CANopen's key functionalities.

The framework employs unit testing and integration testing with Google Test to validate both individual components and the overall application behavior. The framework leverages Robot Framework to orchestrate test execution. Docker containerization ensures a consistent and reproducible testing environment across development, continuous integration/continuous deployment, and production setups. Additionally, a self-hosted GitLab Runner is integrated into the pipeline to support kernel-level operations.

The framework significantly reduces the need for manual testing, improves test coverage, and ensures the robustness of the CANopen-based system. By automating unit and integration tests, it provides a scalable and reusable solution for validating CANopen communication in embedded systems, making it adaptable for future enhancements and similar projects.

Keywords Testing, communications technology

CONTENTS

FIGURES

TABLES

ABBREVIATIONS

API	Application Programming Interface
AuthID	Authentication Identifier
CAN	Controller Area Network
CiA	CAN in Automation
CD	Continuous Deployment/Delivery
CI	Continuous Integration
COB-ID	Communication Object Identifier
DCF	Device Configuration File
Dind	Docker-in-Docker
DNS	Domain Name System
EDS	Electronic Data Sheet
GTest	Google Test Framework
HMI	Human–Machine Interface
int8_t	Signed 8-bit integer
int16_t	Signed 16-bit integer
int32_t	Signed 32-bit integer
IPC	Inter-Process Communication

MCP	Manual Control Panel
MCU	Main Control Unit
MQ	Message Queue
NMT	Network Management
OD	Object Dictionary
PDO	Process Data Object
POSIX	Portable Operating System Interface
PREOP	Pre-Operational State
RPDO	Receive Process Data Object
RTR	Remote Transmission Request
SDO	Service Data Object
TPDO	Transmit Process Data Object
uint8_t	Unsigned 8-bit integer
uint16_t	Unsigned 16-bit integer
uint32_t	Unsigned 32-bit integer
UI	User Interface
vcan	virtual CAN interface
VM	Virtual Machine
YAML	Yet Another Markup Language

1 INTRODUCTION

The featured software, referred to as “Manual Control Panel (MCP),” is developed by W Company to serve as an advanced Human Machine Interface (HMI) panel in industrial automation, facilitating the monitoring and control of critical devices (*MCP HMI Software Architecture*, 2024). MCP is connected to the Main Control Unit (MCU), which is linked via an Ethernet ring along with other system modules. To support reliable communication over a Controller Area Network (CAN) infrastructure with the MCU, the MCP employs CANopen—a higher-layer protocol designed for real-time data exchange and device configuration. Due to the system's inherent complexity and stringent reliability requirements, W Company recognized the necessity for a robust automated testing framework, particularly for the McpCANopenApp component, which implements CANopen functionality.

1.1 MCP Architecture Overview

The MCP architecture employs a modular and extensible design, consisting of two primary components: McpCoreApp and McpCANopenApp. These components operate independently while communicating via Inter-Process Communication (IPC) message queues (MQ), as illustrated in Figure 1 .

1.1.1 Core Application Module (McpCoreApp)

The McpCoreApp serves as the central control unit, utilizing a real-time framework to execute control commands, manage status updates, and track statistical data. By leveraging Qt framework signals and slots, the McpCoreApp ensures seamless integration between the data model and user interface (UI), enabling responsive real-time interactions.

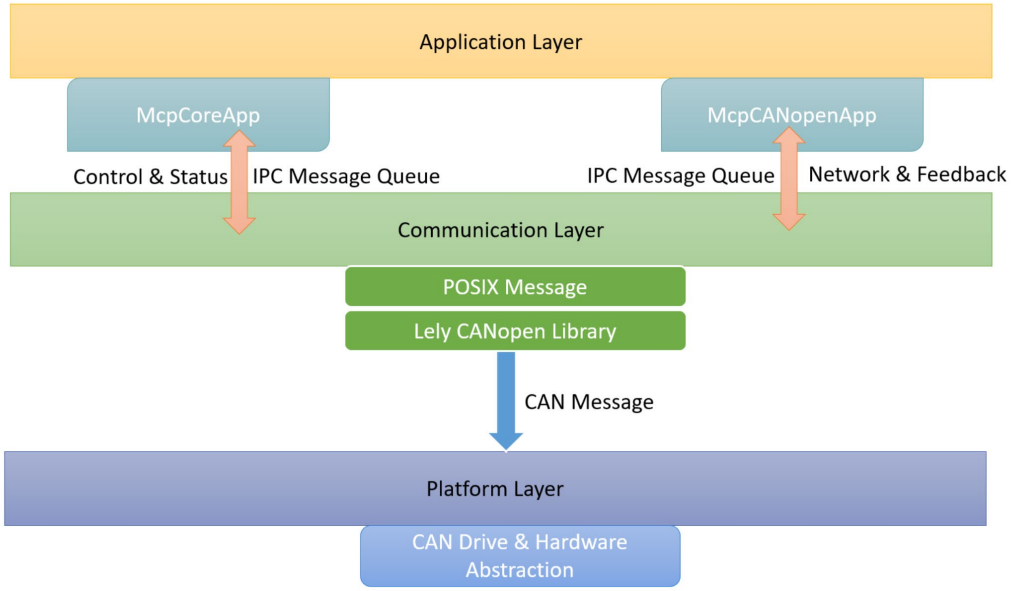


Figure 1. MCP architecture

1.1.2 CANopen Application Module (McpCANopenApp)

For the CANopen Application (McpCANopenApp), the architecture is illustrated in Figure 2 . The McpCANopenApp architecture is built around a CANopen abstraction layer, which bridges the application backend and CANopen engine processes. This layer is implemented as a CANopen application interface object, utilizing a Portable Operating System Interface (POSIX) message queue for standardized IPC.

The execution flow begins with the *HeartbeatListener* class monitoring CAN traffic on the can0 interface via SocketCAN, identifying heartbeat messages by their frame ID (sent by the MCU). Upon detecting a valid heartbeat, the corresponding Object Dictionary (OD) initializes the CANopen stack, which managed by the *McpSlave* class. Meanwhile, two dedicated threads are launched: Reception thread continuously processes incoming messages, updates the OD, handles system events. The transmission Thread manages outgoing CAN messages, ensuring data consistency on the CAN bus.

The *McpSlave* class integrates the Lely CANopen libraries, which provide a modular, high-performance CANopen protocol stack. Lely CANopen consists of a C-based core, which provides the object dictionary and low-level stack functionalities, and a C++ application layer, which is used for building high-level CANopen applications (Lely Industries, 2024). This architecture enables the McpCANopenApp to dynamically load the appropriate Device Configuration File (DCF) based on incoming CAN traffic, allowing it to operate as configurable CANopen node objects. And the node supports critical functionalities such as OD access, heartbeat monitoring, and communication services like process data objects (PDOs), and service data object (SDOs) exchange.

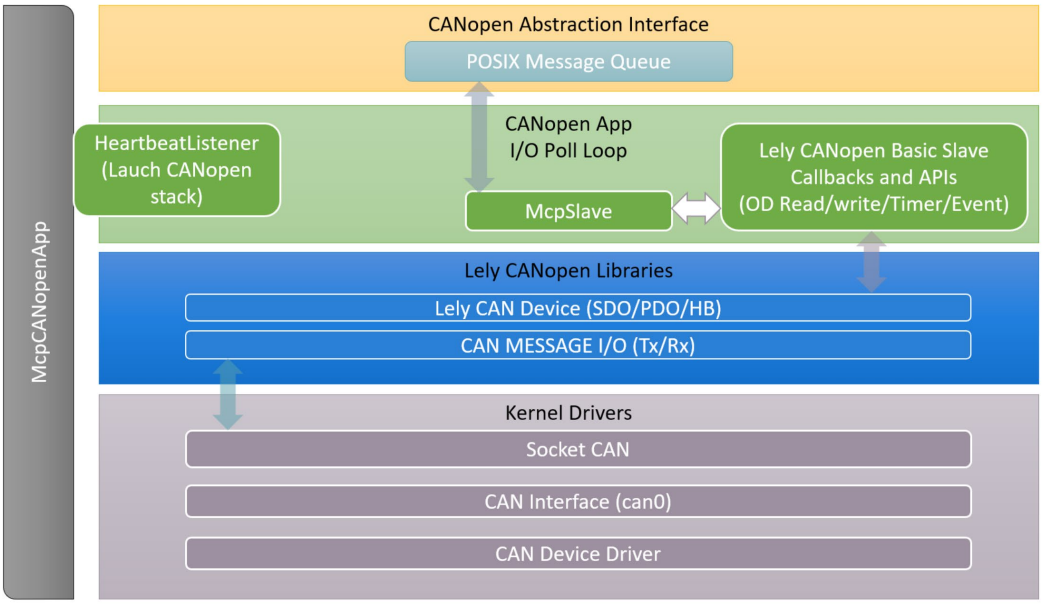


Figure 2. McpCANopenApp architecture

1.1.3 Inter-Module Communication

Communication between McpCoreApp and McpCANopenApp is facilitated through POSIX MQs, enabling a secure and efficient asynchronous data exchange. The McpCoreApp generates control or status messages based on user inputs or system events, which are subsequently retrieved and processed by McpCANopenApp to update the CANopen OD or transmit

CAN messages. Conversely, McpCANopenApp provides feedback or communicates network events to McpCoreApp via dedicated message queues, ensuring seamless integration and low-latency, thread-safe interactions across the system.

1.2 Problem Statement

The McpCANopenApp requires rigorous validation of its two core functional components – the *HeartbeatListener* class and the *McpSlave* class – as well as their integration McpCANopenApp.

1.2.1 *HeartbeatListener* Validation

During the initialization, the *HeartbeatListener* must detect consumer heartbeat CAN frames (sent by the MCU) and filter out invalid/non-compliant heartbeats. It must also map valid heartbeats to the correct DCF from a predefined set stored in the system. Furthermore, it must trigger IPC messages to McpCoreApp to notify the UI of invalid heartbeats (e.g., mismatched node IDs, incorrect timing).

1.2.2 *McpSlave* Validation

After initializing the CANopen stack via the selected DCF, the *McpSlave* class must be tested for compliance with CANopen network management (NMT), including state transitions (Pre-Operational, Operational, Stopped). It also must be tested for robust handling of PDOs for real-time data exchange and SDOs for configuration. Furthermore, for accurate bidirectional translation of CAN frames to/from IPC messages, ensuring data integrity during transmission.

1.2.3 McpCANopenApp real-world behavior

The McpCANopenApp must demonstrate seamless initialization (DCF selection → stack launch → thread activation). And synchronized IPC/CAN message handling under high-load or fault conditions (e.g., message

corruption, CAN bus disconnections). Moreover, it must demonstrate graceful error recovery without manual intervention (e.g., reinitialization after transient faults).

1.2.4 Automated Testing Requirements

Manual testing is error-prone and insufficient for ensuring reproducibility or coverage. A structured automated framework is required to execute unit tests (*HeartbeatListener/McpSlave*), integration tests (IPC/CAN workflows), and system-level validation (CANopen compliance). As well as integrate into a continuous integration and continuous delivery/deployment (CI/CD) pipeline for consistent execution and regression testing.

1.3 Thesis Objectives

The primary objective of this thesis is to design, implement, and validate an automated testing framework for the McpCANopenApp, addressing the challenges outlined in Section 1.2. The framework will systematically verify the functionality of the *HeartbeatListener* and *McpSlave* classes, their integration, and their compliance with CANopen standards under real-world conditions.

First, the framework will develop unit tests to validate the core functionalities of the *HeartbeatListener* and *McpSlave* classes in isolation. For the *HeartbeatListener*, this includes verifying its ability to detect and filter valid consumer heartbeat frames from the MCU, map heartbeat IDs to the correct DCF, and trigger IPC notifications for invalid heartbeats. For the *McpSlave* class, unit tests will ensure compliance with CANopen NMT protocols, and validate the integrity of PDO and SDO transactions, including synchronization with the OD.

Second, integration tests will simulate real-world interactions between the *HeartbeatListener*, *McpSlave*, and McpCoreApp. These tests will

evaluate the bidirectional accuracy of IPC message translation (CAN-to-MQ and MQ-to-CAN), validate end-to-end workflows such as DCF-based stack initialization and thread activation, and assess fault tolerance under adverse conditions such as CAN bus disconnections or message corruption. A key focus will be ensuring graceful error recovery without manual intervention, such as automatic reinitialization after transient faults.

Finally, the framework will automate the testing workflow by integrating it into a CI/CD pipeline. This automation will ensure consistent execution of unit and integration tests, enable reproducible validation of CANopen compliance, and support regression testing for future development cycles.

1.4 Thesis Structure

This chapter introduces the background of the thesis project and objectives. Chapter 2 provides the theoretical foundation, detailing CANopen fundamentals and its core features. Chapter 3 discusses the technologies and methodologies used in designing the testing framework. It outlines multi-layered testing strategies, detailing how unit testing, integration testing, and system validation are structured. The test automation techniques employed in the framework, including continuous integration and automated execution, are explained. Chapter 4 presents the implementation details and the result of test results, demonstrating the effectiveness of the framework in validating the McpCANopenApp within a CI/CD pipeline. Challenges encountered during development and validation, along with solutions, are outlined simultaneously. The last chapter concludes with key findings, research contributions, limitations and recommendations for future work.

1.5 Use of AI in This Thesis

In this thesis, I have utilized ChatGPT and DeepSeek to refine language, correct grammatical errors, improve word choice, and enhance the logical flow of my writing. The primary use of AI occurred during the writing and editing phases, where AI tools assisted in making the text clearer and more coherent while maintaining my original intent.

For language enhancement, I employed AI-driven grammar and style-checking tools, which helped refine sentence structures and improve readability. Additionally, AI was used to generate flowcharts for Figure 22 and Figure 26 in Chapter 4 based on my software code, ensuring accurate visual representation of system architecture and data flows.

I have ensured the authenticity of the content by verifying AI-generated information against original sources and referencing all external material appropriately. AI was not used to generate novel research findings or conclusions but rather to assist in content structuring and presentation.

Data protection has been strictly maintained throughout the process. No sensitive or proprietary data has been input into AI tools, and all generated content has been reviewed to ensure compliance with ethical and academic standards.

All AI applications were used responsibly, with careful oversight to preserve the integrity and originality of my work.

2 THEORETICAL BASIS

2.1 CANopen Fundamentals

The ISO 7-layer reference model (International Organization for Standardization, 2024) is a standard framework for defining network communication layers. The CAN protocol primarily implements Physical Layer and parts of Data Link Layer in hardware. CAN ensures efficient, real-time communication by using priority-based message arbitration and robust error handling (Pfeiffer et al., 2008).

CANopen, a higher-layer protocol built atop CAN, extends functionality by selectively implementing parts of the upper layers (e.g., Transport and Application Layers). It standardizes device interoperability through features like real-time data exchange using PDO, configuration through SDO, and node state management via NMT (Pfeiffer et al., 2008). These capabilities make CANopen a scalable and reliable solution for distributed embedded systems, particularly in industrial automation.

2.2 Primary functionalities of CANopen

At the core of a CANopen node lies the OD, a standardized lookup table acting as a centralized repository for all device parameters, including communication settings (e.g., node ID, baud rate) and application-specific data (e.g., sensor calibration values). To ensure interoperability, the DCF—implemented in formats such as EDS (Electronic Data Sheet)—defines the OD's structure and parameters in a machine-readable format. During network initialization, the DCF is loaded into the device, acting as a blueprint to configure its behavior according to network requirements (Pfeiffer et al., 2008).

For example, the DCF assigns NMT Master/Slave roles, where the master controls network state transitions (e.g., start, stop), while slaves follow

commands. It also configures critical communication identifiers (COB-IDs), which combine the Node ID and control bits to define message routing and priority. COB-IDs are pivotal in CANopen, as they distinguish message types such as SDOs, PDOs, and Heartbeats.

2.2.1 Heartbeat Mechanism

The Heartbeat mechanism ensures network reliability by continuously monitoring device availability. If a node fails to transmit its Heartbeat within a predefined timeout, the network assumes it is offline or faulty, triggering error recovery procedures.

Each node periodically sends a Heartbeat message with a COB-ID of 0x700 + Node ID (e.g., 0x703 for Node 3). Monitoring nodes track these Heartbeats; exceeding the configured timeout (e.g., 3000 millisecond) signals a lost node. The DCF defines these parameters in the EDS format, as shown in Figure 3 and Figure 4.

```
[HeartbeatProducer]
COB-ID = 0x700 + $NODEID  ; Heartbeat message identifier
Interval = 1000           ; Transmit every 1000 ms
```

Figure 3. DCF Producer Heartbeat Configuration Example

```
[HeartbeatConsumer]
NodeID = 5                ; Monitored node ID
Timeout = 3000            ; Trigger error if no heartbeat in 3000 ms
```

Figure 4. DCF Consumer Heartbeat Configuration Example

The Heartbeat mechanism in CANopen is closely integrated with the NMT Master to ensure system reliability. Each node periodically sends a Heartbeat message to indicate that it is active. If a node stops sending heartbeats within a defined timeout period, the NMT Master detects the failure and initiates a recovery process.

In Figure 4, Node 5 is configured with a timeout of 3000 milliseconds. If the NMT Master does not receive a heartbeat from Node 5 within this time, it triggers an error condition. In response, the NMT Master may perform corrective actions, such as issuing a reset command to restart the failed node or switching it to a pre-operational state to prevent system disruption. This integration ensures fault tolerance by allowing the system to automatically recover from node failures.

2.2.2 PDO

PDOs enable high-priority, real-time data exchange (e.g., sensor readings, actuator commands) with minimal latency. Unlike SDOs, PDOs bypass protocol overhead, transmitting data directly via preconfigured COB-IDs. For example, an DCF file configures a Transmit-PDO (TPDO) to send motor speed and temperature data cyclically as Figure 5.

```
[TPDO1]
COB-ID = 0x201              ; CAN identifier for TPDO1
TransmissionType = 255      ; Event-driven (on change)
EventTimer = 100            ; Transmit every 100 ms if no change
Mapping = 0x2000:01         ; Motor speed (OD index 0x2000, subindex 1)
Mapping = 0x2001:01         ; Temperature (OD index 0x2001, subindex 1)
```

Figure 5. TPDO Example in DCF

Here, TPDO1 transmits motor speed and temperature values either when the data changes or every 100 millisecond, balancing responsiveness with bus load. Similarly, Receive-PDOs (RPDOs) are configured to accept control commands, such as target positions for a servo drive.

Testing PDOs involves ensuring correct data mapping (e.g., OD index 0x2000 maps to motor RPM) and validating transmission triggers (cyclic vs. event-driven). Confirm that PDOs update the OD correctly. Test writing invalid PDO values and ensure rejection. Corrupted PDO frames must be detected and ignored.

2.2.3 SDO

While PDOs handle real-time data, SDOs provide access to the OD for configuration, diagnostics, and non-time-critical parameter updates. The DCF defines SDO communication channels and access rights. For example, in Figure 6, an DCF entry may restrict write access to critical parameters.

```
[SDO]
ClientCOB-ID = 0x600 + $NODEID  ; SDO request identifier
ServerCOB-ID = 0x580 + $NODEID  ; SDO response identifier
AccessType = ro                 ; Read-only access to calibration parameters
```

Figure 6. SDO Example in DCF

This ensures that calibration values (e.g., OD index 0x3010) cannot be modified remotely, preventing accidental misconfiguration.

Testing SDOs requires verifying correct OD index mappings and handling edge cases, such as invalid indices (e.g., 0x9999) or out-of-range values. Corrupted SDO frames must also be gracefully rejected.

2.3 CANopen Message Format

The CANopen message format is derived from the CAN frame structure, a fundamental component of the CAN protocol. A clear understanding of the format is essential for comprehending how communication is executed efficiently and accurately within a CANopen network.

In the CAN protocol, data transmission occurs through frames. These frames are composed of either an 11-bit or 29-bit Node ID, control bits (including the remote transfer bit Remote Transmission Request (RTR), a start bit, and a 4-bit data length field), followed by a data payload of 0 to 8 bytes, as illustrated in Figure 7, where each cube represents one byte (National Instruments, 2024) .

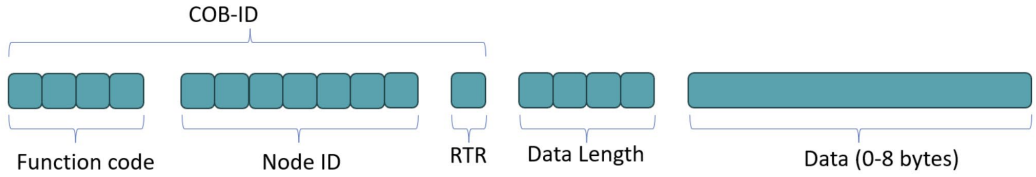


Figure 7. CAN Frame Structure

A heartbeat CAN frame, as illustrated in Figure 8, consists of an 11-bit COB-ID where the upper 4 bits (function code) are set to 7, indicating a heartbeat message. The lower 7 bits represent the Node ID, which in this case is 0x11 (hexadecimal). The RTR bit is set to 0, signifying a data frame rather than a remote request. The payload contains a single data byte (0x05), conveying the node's status. The corresponding CAN frame is: *711#05*.

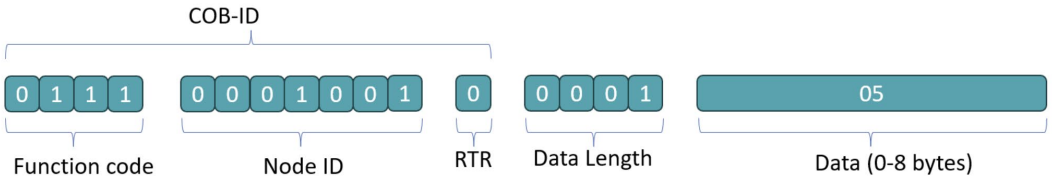


Figure 8. Heartbeat CAN Frame Example

3 TECHNOLOGIES AND METHODOLOGIES

This chapter outlines the design and implementation of the automated testing framework for the McpCANopenApp. The framework employs a multi-layered approach to validate CANopen functionalities, leveraging modern tools to ensure scalability, reproducibility, and compliance with industrial automation standards. The discussion begins with a high-level overview of the framework architecture, followed by an analysis of its components and their integration into a continuous testing pipeline.

3.1 Automated Testing Framework Design

The automated testing framework for CANopen-based systems is designed to systematically validate software functionality through a dual-layered approach, integrating both local development testing and continuous integration/continuous deployment (CI/CD) pipeline execution. This structure ensures that testing is both accessible to developers for real-time debugging and rigorously enforced before code reaches production.

Local testing provides an isolated environment for developers to compile, execute, and debug the MCP software within a controlled containerized setting. The Automated tested framework local testing shown in Figure 9 begins by setting up a Docker container, built from a predefined Dockerfile, encapsulating all necessary dependencies, including SocketCAN for CAN bus emulation, Google Test (GTest) for testing creation, and Robot Framework for integration all the tests. Within this container, CMake is utilized to build the MCP software alongside its corresponding test binaries, ensuring that the build configuration remains identical to that used in the CI/CD pipeline.

Once compiled, both unit and integration tests are executed inside the container, facilitated by an automated bash script that manages the

testing lifecycle. Results, including logs, test reports, and diagnostic outputs, are captured and stored locally, enabling developers to analyze failures and refine their codebase before submission. This localized approach mitigates inconsistencies between development and deployment environments, ensuring that detected defects are addressed prior to code integration.

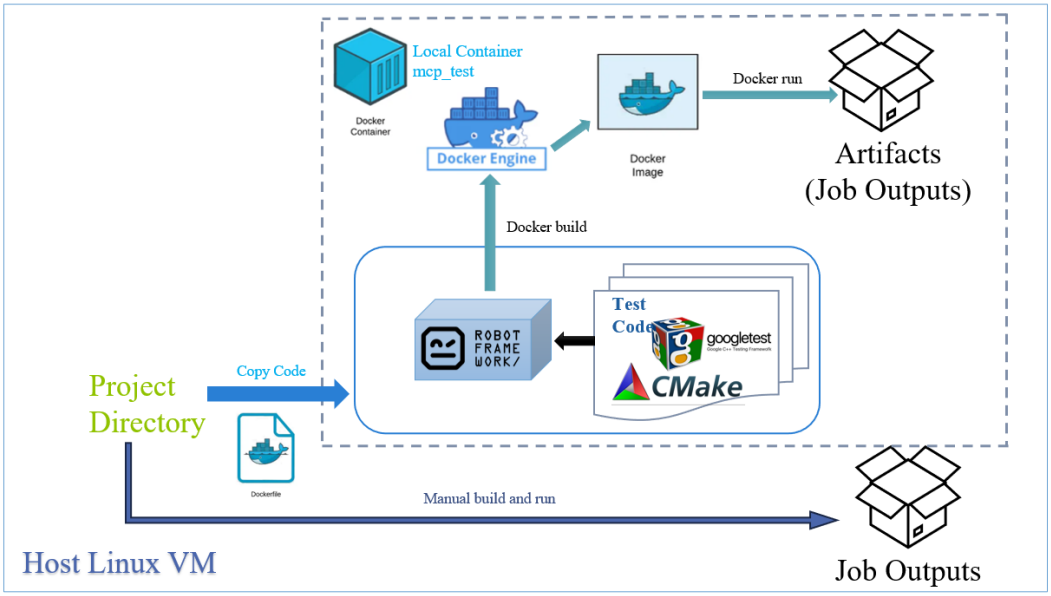


Figure 9. Automated Test Framework Local Testing

Beyond local testing, the CI/CD pipeline automates validation to enforce code integrity across the entire development lifecycle illustrated in Figure 10. Upon pushing a commit to the GitLab repository, the pipeline is automatically triggered, leveraging the predefined *.gitlab-ci.yml* configuration file, which dictates the sequence of build and test operations. A self-hosted GitLab Runner initiates a new Docker container on a Host Linux Virtual Machine, ensuring that every execution is performed in a clean, reproducible environment.

The pipeline proceeds with compiling the MCP software and executing the same unit and integration tests conducted in the local environment, guaranteeing consistency between development and deployment stages.

Test artifacts, including detailed logs, pass/fail reports, and CAN bus traffic records, are packaged and stored within GitLab, providing full transparency and traceability for the development team. The automation of this workflow ensures that every code modification undergoes rigorous, repeatable validation before integration, eliminating the need for manual intervention.

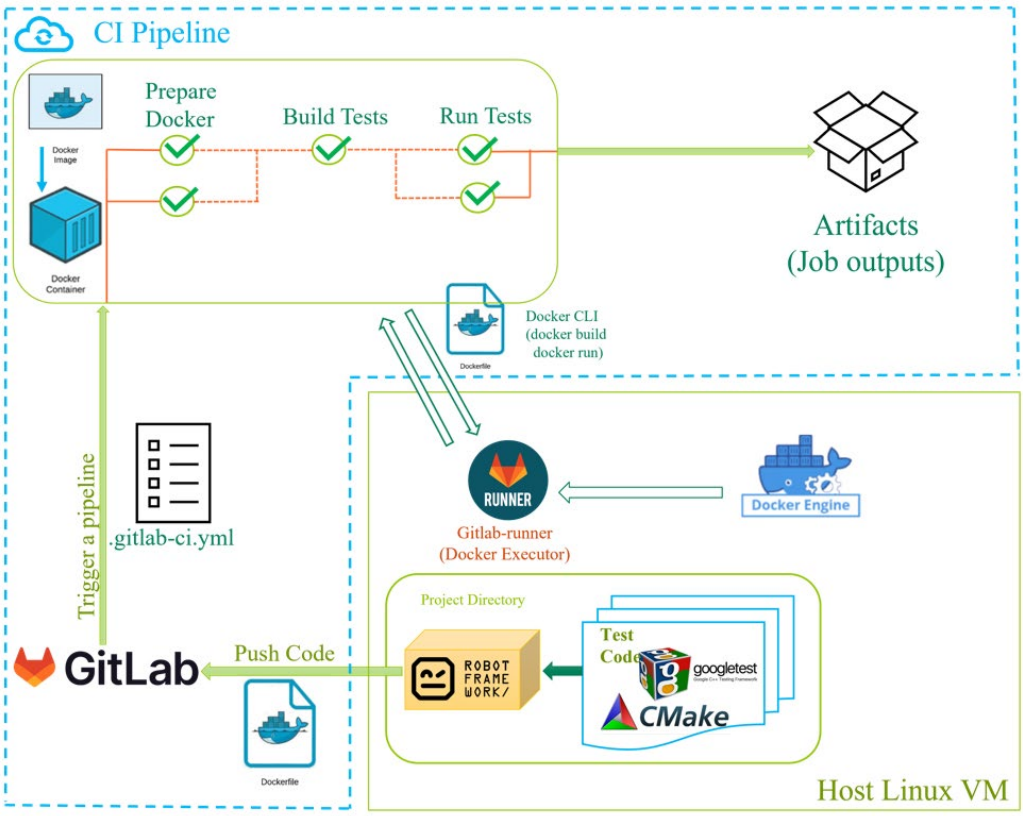


Figure 10. Automated Testing Framework CI/CD Pipeline

Both local and pipeline testing leverage identical Docker environments, ensuring that test behavior remains consistent across development and deployment. Local testing serves as a flexible, developer-controlled environment, allowing iterative debugging and real-time validation, whereas the CI/CD pipeline acts as a systematic quality gate, enforcing software stability through automated, organization-wide verification. By combining developer agility with structured validation, this dual-layered

approach enhances both development efficiency and production reliability within MCP's CANopen ecosystem.

3.1.1 Test Code Development and Execution

The foundation of the framework lies in the creation and validation of test cases, which are executed through a combination of unit and integration tests. Developers write test cases using GTest for unit testing and Robot Framework for integration testing. A CMake-based build system systematically compiles McpCANopenApp and all test components, ensuring an organized and structured testing workflow.

GTest as an open-source C++ testing library developed by Google, is selected for its ability to create automated, reliable, and maintainable test cases. It provides features such as assertions, fixtures, parameterized tests, and death tests (Google, 2021). The framework follows a structured lifecycle, where global setup and teardown functions (*Environment::SetUp()* and *Environment::TearDown()*) initialize and clean up the testing environment, while individual test cases follow a hierarchical execution flow using *SetUpTestCase(), SetUp(), TST/TEST_F/TEST_P, TearDown()*, and *TearDownTestCase()*, ensuring systematic validation and isolation of test instances as illustrated in Figure 11. Its compatibility with C++ codebases and seamless integration with CMake make it ideal for validating PDO/SDO behaviors and error handling (e.g., *ASSERT_FALSE(error_code)*).

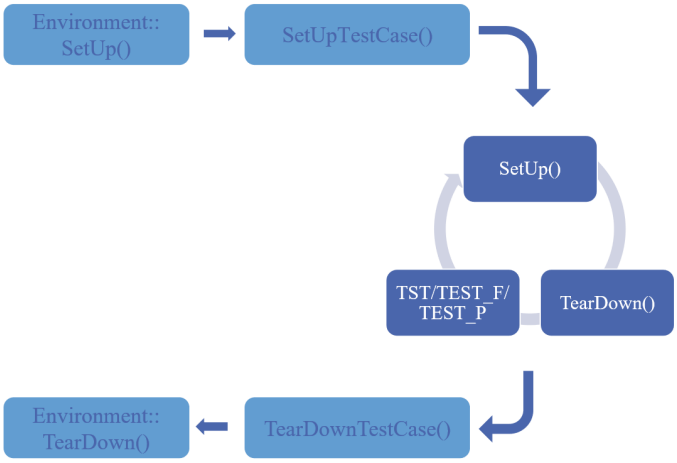


Figure 11. Google Test Framework

Robot framework as a keyword-driven automation tool is chosen for its ability to execute tests automatically on each build, generate structured logs (e.g., *output.xml*, *log.html*, *report.html*), and ensure consistent, reproducible results without manual intervention (Robot Framework Foundation, 2021).

3.1.2 Local Execution via Docker

For local development, tests run inside a Docker container (*mcp_test*) to replicate the CI environment. The Dockerfile ensures the container includes preconfigured tools such as SocketCAN (for CAN bus emulation), GTest, and Robot Framework. The workflow involves:

- Code Integration by copying the host code into the container.
- Build Process, compiling the code using CMake.
- Test Execution by running tests within the isolated Docker environment.

Docker ensures consistency and reproducibility by encapsulating dependencies (libraries, binaries, configurations) in a portable container,

eliminating variations caused by differing system configurations (Merkel, 2014).

3.1.3 Integration with CI/CD Pipeline

Docker technology facilitates consistent test execution across both local and CI environments by containerizing dependencies and configurations (Docker Inc, 2021). This ensures that tests run identically during development and automated validation. The GitLab CI/CD pipeline further automates the testing process, enforcing uniform checks for all code changes. When developers push code to the GitLab repository, the pipeline triggers a GitLab Runner from *.gitlab-ci.yml* file, which initializes a fresh *mcp_test* Docker container on a Host Linux virtual machine (VM). This container mirrors the local Docker environment, maintaining parity between development and CI setups. Once the environment is configured, the pipeline automates the build and test execution—replicating the local Docker workflow—and generates standardized outputs, including JUnit XML test reports, logs, and binaries. These artifacts are archived as job outputs and made accessible through GitLab's interface, enabling developers to review test failures, coverage metrics, and diagnostic data efficiently.

Docker-in-Docker (DinD) and GitLab Runner are key technologies in ensuring a seamless and consistent testing workflow across local and CI environments. DinD Enables nested Docker operations, allowing the GitLab Runner (itself a container) to spawn test containers (Harsh, 2023). While DinD operates with elevated privileges introducing potential security risks, but GitLab runner mitigates risks through job isolation and restricted host access (Docker, 2021).

3.2 Core Testing Technologies and Tools

This section elaborates on the critical technologies and libraries underpinning the tests in the framework. The framework leverages specialized

tools within its test suites to emulate CANopen communication, validate protocol compliance, and ensure environmental consistency. These tools are integrated into test cases to simulate real-world conditions and verify the McpCANopenApp's behavior.

The Lely CANopen libraries form the backbone of the McpCANopenApp's protocol implementation, providing a modular and efficient CANopen stack. For testing, the Lely libraries are instrumental in emulating CANopen master and slave nodes. For instance, the *McpSlave* class leverages Lely's OD management to process incoming SDO requests and validate PDO transactions via its Application Programming Interfaces (APIs), enabling precise testing of node behavior (Lely Industries, 2024).

To emulate and validate CAN communication, the framework integrates SocketCAN, a Linux kernel subsystem originally developed by Volkswagen Research (Ertl et al., 2007). SocketCAN facilitates both virtual (vcan) and physical CAN interface emulation, as illustrated in Figure 12 (SocketCAN, n.d.), which depicts its layered architecture. Operating at the kernel level, SocketCAN provides low-latency communication and precise control over CAN frame transmission and reception.

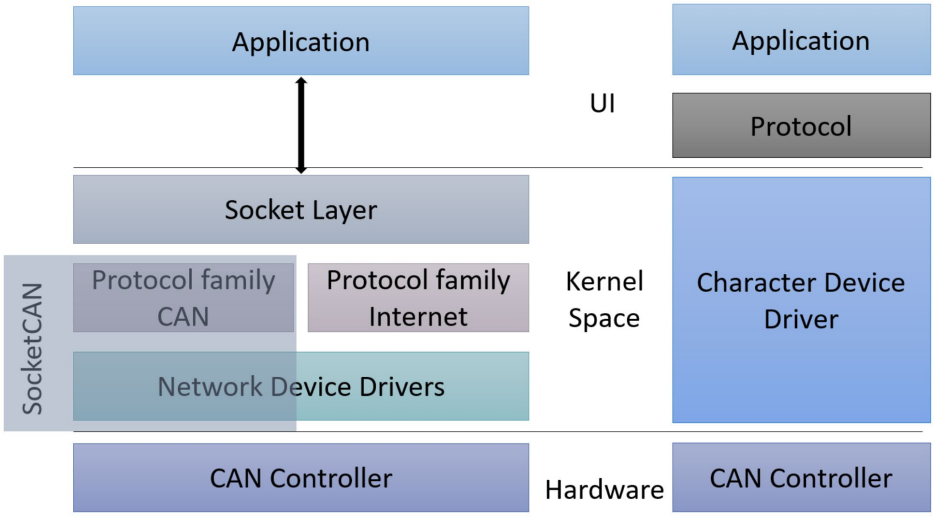


Figure 12. SocketCAN

The framework further employs tools from the can-utils package to simulate and analyze CAN traffic (Marc Kleine-Budde, n.d.):

- cansend transmits custom CAN frames with explicit control over COB-IDs, payloads, and data lengths.
- candump captures real-time bus traffic, enabling validation of transmitted frames against CANopen specifications.
- canplayer replays pre-recorded CAN logs to simulate complex network scenarios, such as burst errors or intermittent node failures.

Together, SocketCAN and can-utils create a controlled, hardware-agnostic testing environment. For instance, vcan allow developers to mimic network disruptions or corrupted messages without physical hardware, ensuring reproducible test conditions. By combining Lely CANopen's protocol implementation with SocketCAN's emulation capabilities, the framework bridges the gap between theoretical CANopen standards and real-world deployment, ensuring the McpCANopenApp meets the stringent demands of industrial automation.

4 IMPLEMENTATION AND RESULTS

This section delineates the implementation details of the automated testing framework, its validation outcomes, and the challenges encountered during development.

4.1 Test Environment Setup

4.1.1 Build System and Test Orchestration

The testing framework leverages a CMake-based build system to compile the McpCANopenApp, unit tests, and integration tests. As illustrated in Figure 13, the test directory structure includes critical components such as CMake configuration files, C++ test binaries, Python wrappers, and Robot Framework test suites. The CMake configuration automates dependency resolution, detecting external libraries like Lely CANopen for protocol stack emulation and proprietary MCP libraries.

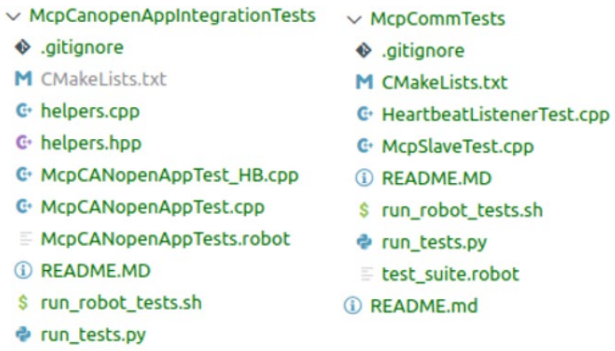


Figure 13. Tests Directory Included Files

The Python wrapper (run_tests.py) serves as a bridge between C++ test binaries and the Robot Framework. By defining custom keywords such as *Run Mcp Slave Tests*, it executes compiled binaries (e.g., *McpSlaveT-*

est, HeartbeatListenerTest) and captures their exit codes. These keywords are invoked in the Robot Framework test suite (test_suite.robot), which validates test outcomes by comparing exit codes against expected results (0 for success, non-zero for failure).

A Bash orchestration script (run_tests.sh), depicted in Figure 14, automates the end-to-end workflow. It configures the system environment—adjusting POSIX message queue limits and initializing vcan0—before compiling the code with CMake and executing the Robot Framework tests. This script ensures reproducibility across development and CI environments by encapsulating all build and execution steps.

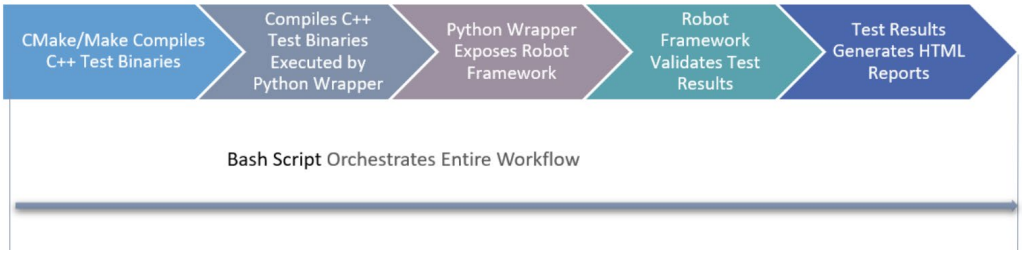


Figure 14. Bash Script Workflow

4.1.2 Cloud Restrictions on Privileged Containers

The project initially employed a Linux VM-based CI/CD pipeline, where tests ran in a preconfigured Ubuntu environment. While functional, this approach introduced environment inconsistencies, resource overhead and manual kernel module management challenges that hindered efficient testing as listed below:

- Discrepancies in library versions, kernel updates, and system configurations between local development machines and the VM frequently caused test failures. Debugging these issues required significant effort to trace root causes across divergent environments (Fiveable, 2024).

- VM provisioning consumed excessive time and computational resources, delaying pipeline execution and complicating rapid iteration (Spacelift, 2023).
- The vcan kernel module, essential for CAN bus emulation, required manual loading with root privileges. This step was error-prone and incompatible with automated.

To address these limitations, the pipeline was redesigned to leverage Docker containers, which provided lightweight, portable environments with declarative dependency management (National Center for Ecological Analysis and Synthesis, 2024). A Dockerfile standardized the testing environment across local development, CI pipelines, and team machines, eliminating configuration drift.

However, cloud-based CI services imposed restrictions on privileged containers, blocking critical operations such as kernel module loading (e.g., vcan) (Docker Inc, n.d.). To resolve this, a self-hosted GitLab Runner was deployed with root access, enabling automated loading of the vcan module during container initialization, execution of privileged operations required for CAN emulation, and consistent replication of local testing conditions in the CI pipeline(GitLab Inc, 2023).

4.1.3 Domain Name System (DNS) Resolution in Docker and GitLab Runner

Another critical challenge emerged during the GitLab CI/CD pipeline execution when Docker containers failed to resolve the internal domain gitlab.wcompany.com, causing failures at the "Fetching source" stage. The root cause lay in Docker’s default networking behavior, which isolates containers within their own network stack. By default, Docker containers rely on public DNS servers (e.g., 8.8.8.8), which lack records for internal organizational domains. This created a discrepancy: while the host system used internal DNS servers to resolve gitlab.wcompany.com, these configurations were not inherited by Docker containers, leading to unresolved domains and pipeline failures.

To resolve this, three complementary approaches were implemented. Firstly, the Docker daemon's configuration file (*daemon.json*) was modified to explicitly use the host's internal DNS resolvers. Secondly, the GitLab Runner was configured to launch containers with parameter *network_mode: host*, bypassing Docker's default network isolation. Third, Tools such as nslookup, ping, and curl were embedded into pipeline scripts to validate DNS resolution within containers.

Modifications to Docker's network configuration (*network_mode: host*) and DNS settings in *daemon.json* eliminated domain resolution failures. Internal domains like gitlab.wcompany.com were consistently resolved, reducing pipeline initialization errors from approximately 35%(based on 21/60 pipeline runs) to 0%. Automated diagnostics via nslookup and ping in pipeline scripts provided real-time validation of network health.

4.1.4 Test Environment Validation Outcomes

This section evaluates the effectiveness of the test environment setup by analyzing key performance metrics and resolving critical challenges. The integration of Docker containers and a self-hosted GitLab Runner demonstrated measurable improvements in environment consistency, error resolution, and pipeline reliability.

By standardizing dependencies through containerization, Docker eliminated library conflicts and toolchain mismatches. As shown in Figure 15, the GCC-11 toolchain was consistently utilized across builds, ensuring uniform compilation. Figure 16 further illustrates the isolation of test execution within Docker, which eradicated false negatives caused by host system variations. This resulted in 100% environment parity between local development and CI pipelines, with zero configuration-related test failures post-implementation.

Figure 15. Docker Build in Host VM

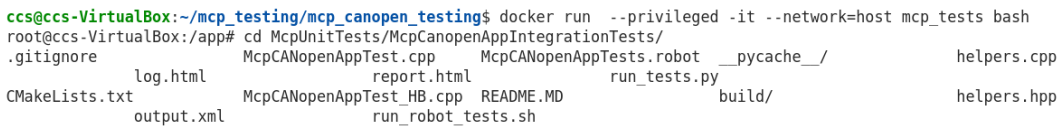


Figure 16. Docker Run in Host VM

The self-hosted GitLab Runner, depicted in Figure 17, resolved restrictions on privileged operations in cloud CI services. The Runner automated the loading of the vcan kernel module during pipeline initialization, enabling CAN bus emulation without manual intervention. This eliminated 100% of vcan-related errors, as evidenced by the successful pipeline execution in Figure 18.

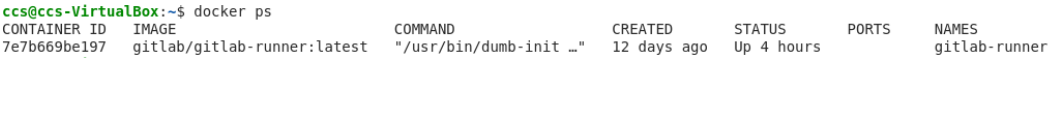


Figure 17. GitLab Runner in Host Machine

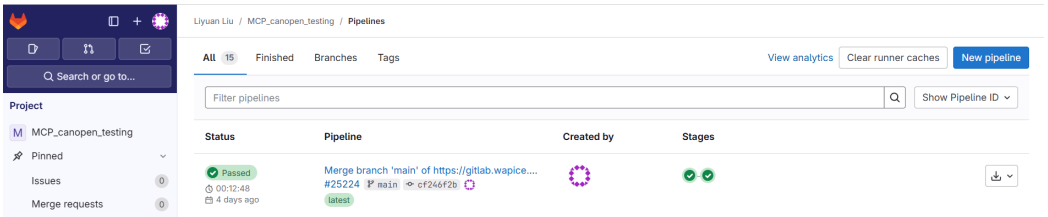


Figure 18. GitLab CI/CD Pipeline Execution Result

4.2 Unit Test Implementation

4.2.1 *HeartbeatListener* Test Suite

The tests for the *HeartbeatListener* class validate its ability to process CAN heartbeat messages and accurately determine the appropriate DCF file and core application. The tests, implemented using the GTest framework, are organized into distinct categories to isolate specific functionalities, Table 1 summarizes the key test scenarios and their objectives.

CheckForCanBusErrors() method was selected as an example to demonstrate how the testing framework ensures robustness against data corruption, a critical requirement in industrial automation systems where undetected errors can lead to operational failures or safety risks.

Table 1. HeartbeatListener Unit Test Cases

Test Case	**Objective**	**Method Validated**
TestOpenSocketInvalidInterface	Ensure graceful failure on invalid/nonexistent CAN interface.	*OpenSocket()*
TestIsValidHeartbeatCanId	Validate detection of CAN IDs within the heartbeat range (0x700–0x7FF).	*IsValidHeartbeatCanId()*
TestCheckForCanBusErrors	Confirm error frame detection (e.g., corrupted data).	*CheckForCanBusErrors()*
TestProcessKnownHeartbeat	Test DCF selection logic and counter updates for valid heartbeats.	*ProcessKnownHeartbeat()*
TestProcessUnknownHeartbeat	Ensure unknown heartbeats are logged but ignored.	*ProcessUnknownHeartbeat()*
TestListenForInitialHeartbeat	Validate device identification under valid, unknown, or no heartbeat scenarios.	*ListenForInitialHeartbeat()*

The primary objective of this test is to verify that the *HeartbeatListener* class reliably identifies corrupted CAN frames using the *CheckFor-*

CanBusErrors method. Corrupted frames, often caused by electromagnetic interference or faulty hardware, must be detected and discarded before they propagate to downstream processes such as DCF initialization or OD synchronization. Failure to filter such errors could result in misconfigured devices or unstable system states. The test simulates two scenarios.

- Error frame handling, a CAN frame is artificially marked with the *CAN_ERR_FLAG*, a standardized identifier for protocol-level errors. The test asserts that the method correctly flags this frame as invalid, triggering error-handling routines such as logging or frame rejection (UCLouvain, 2024).
- Valid frame processing, a normal heartbeat frame with a valid CAN ID (e.g., 0x701) is generated. The test confirms that the method does not falsely classify this frame as corrupted, ensuring legitimate data proceeds to initialization workflows.

In conclusion, the *CheckForCanBusErrors()* test case underscores the testing framework's ability to address real-world challenges in embedded communication. By rigorously validating error detection, the framework not only ensures protocol compliance but also safeguards against operational hazards, exemplifying the principles of reliability and safety central to this thesis. Like this example, all test cases in this test suit passed with 100% coverage of critical *HeartbeatListener* functionalities as Figure 19 shows. The suite demonstrated robust handling of edge cases, including high-frequency heartbeat bursts, overlapping CAN IDs, and intermittent communication failures. By isolating components

through mock interfaces, the tests ensured reproducible validation of real-world scenarios while maintaining system stability.

```
● ccs@ccs-VirtualBox:~/mcp_testing/mcp_canopen_testing/McpUnitTests/McpCommTests/build$ ./HeartbeatListenerTest
  Running main() from /home/ccs/googletest/googletest/src/gtest_main.cc
  [==========] Running 10 tests from 1 test suite.
  [----------] Global test environment set-up.
  [----------] 10 tests from HeartbeatListenerTest
  [ RUN      ] HeartbeatListenerTest.TestOpenSocketValidInterface
  [       OK ] HeartbeatListenerTest.TestOpenSocketValidInterface (0 ms)
  [ RUN      ] HeartbeatListenerTest.TestOpenSocketInvalidInterface
  [       OK ] HeartbeatListenerTest.TestOpenSocketInvalidInterface (0 ms)
  [ RUN      ] HeartbeatListenerTest.TestIsValidHeartbeatCanId
  [       OK ] HeartbeatListenerTest.TestIsValidHeartbeatCanId (0 ms)
  [ RUN      ] HeartbeatListenerTest.TestCheckForCanBusErrors
  [       OK ] HeartbeatListenerTest.TestCheckForCanBusErrors (0 ms)
  [ RUN      ] HeartbeatListenerTest.TestProcessKnownHeartbeat
  [       OK ] HeartbeatListenerTest.TestProcessKnownHeartbeat (0 ms)
  [ RUN      ] HeartbeatListenerTest.TestProcessUnknownHeartbeat
  [       OK ] HeartbeatListenerTest.TestProcessUnknownHeartbeat (0 ms)
  [ RUN      ] HeartbeatListenerTest.TestSendMessage
  [       OK ] HeartbeatListenerTest.TestSendMessage (0 ms)
  [ RUN      ] HeartbeatListenerTest.TestListenForInitialHeartbeatWithValidFrames
  [       OK ] HeartbeatListenerTest.TestListenForInitialHeartbeatWithValidFrames (0 ms)
  [ RUN      ] HeartbeatListenerTest.TestListenForInitialHeartbeatWithUnknownFrames
  [       OK ] HeartbeatListenerTest.TestListenForInitialHeartbeatWithUnknownFrames (0 ms)
  [ RUN      ] HeartbeatListenerTest.TestListenForInitialHeartbeatWithNoFrames
  [       OK ] HeartbeatListenerTest.TestListenForInitialHeartbeatWithNoFrames (0 ms)
  [----------] 10 tests from HeartbeatListenerTest (1 ms total)

  [----------] Global test environment tear-down
  [==========] 10 tests from 1 test suite ran. (1 ms total)
  [  PASSED  ] 10 tests.
```

Figure 19. HeartbeatListener Unit Tests Result

4.2.2 *McpSlave* Test Suite

To validate the reliability of the CANopen stack's core component, a comprehensive McpSlaveTest suite was developed using the GTest framework. This suite evaluates the *McpSlave* class's ability to manage OD interactions, PDOs, NMT state transitions, and error handling, ensuring fully validate functional correctness. Note here performance test is not included in this test. The McpSlaveTest suite is divided into six categories, as summarized in Table 2.

Table 2. McpSlave Unit Tests Coverage

Test Category	**Validation Scope**	**Key Results**
SDO Transactions	Read/write of OD entries (uint8_t, uint8_t, int16_t, bool, Unicode strings).	Valid values written/read; Invalid inputs rejected with error codes.
RPDO Handling	Data ingestion from remote nodes (bool, uint8_t, int8_t, uint16_t, int16_t, uint32_t).	RPDO values parsed and stored in OD; IPC messages triggered.
TPDO Transmission	Configuration of TPDO parameters and OD synchronization.	TPDO values written to OD; Tx events triggered without errors.
NMT State Transitions	Responses to NMT commands (START, STOP, PREOP, BOOTUP, RESET).	Correct IPC messages sent with timeout flags for non-Operational states.
Heartbeat Management	Configuration, reception, and timeout handling.	Heartbeat intervals stored in OD; Timeout events trigger IPC messages.
Emergency Handling	Access to error register (OD index 0x1001) and invalid subindex detection.	Error register read successfully; Invalid subindex access triggers SDO errors.

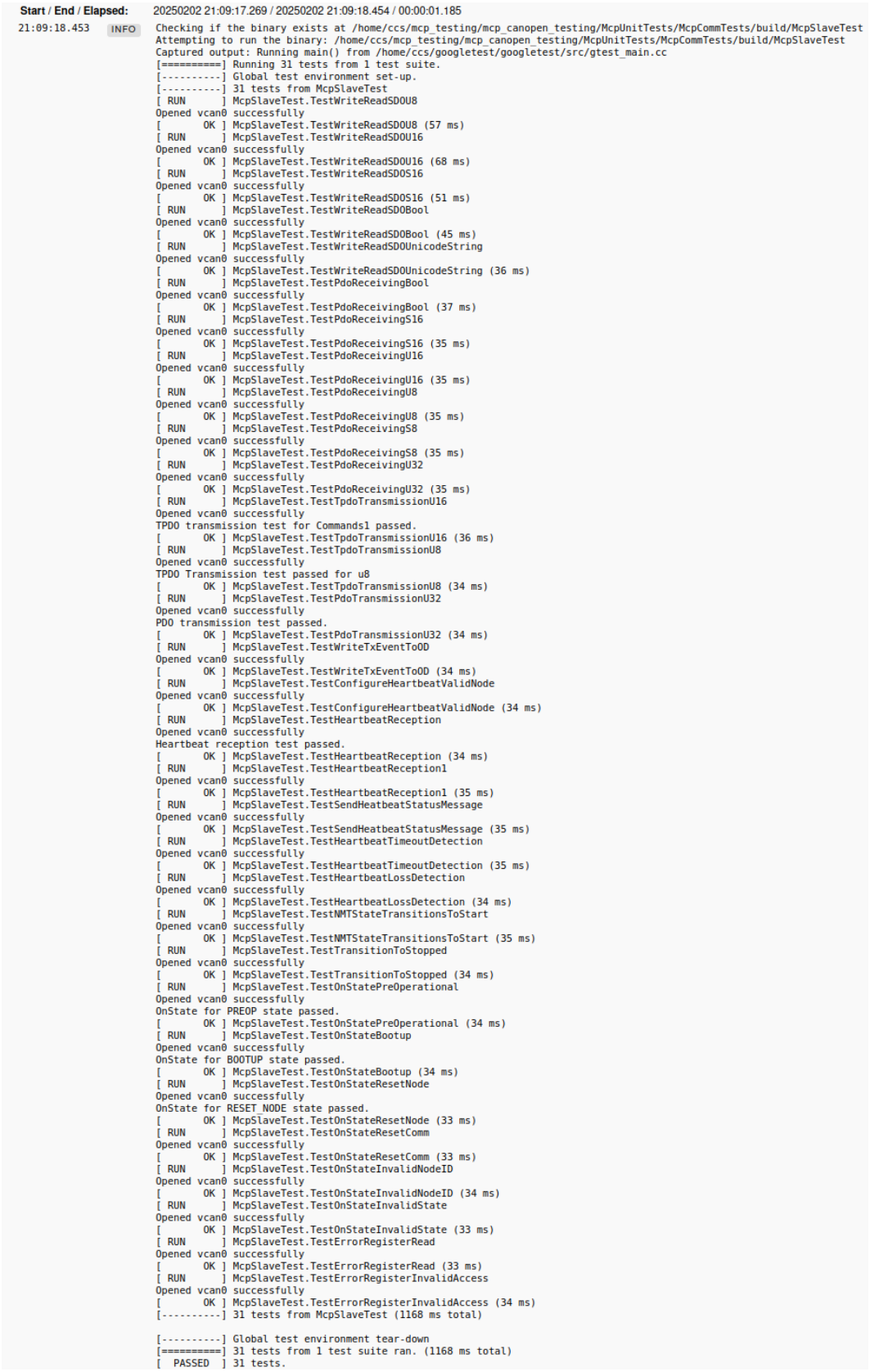
Take test case *TestWriteReadSDOU8()* as an example, this test validates the *McpSlave's* ability to write and read unsigned 8-bit integers (uint8_t) in the OD. SDO transactions are foundational to CANopen device configuration, enabling parameter updates such as sensor thresholds or operational modes. This test case exemplifies the framework's approach to ensuring data integrity in the OD. It targets the OD entry for Major Version (index 0x4101, subindex 0x04), a parameter for firmware compatibility. The test procedure is as follows:

- Valid Value Test: A nominal value (0x12) is written to the OD via SDO and read back to confirm accuracy.

- Boundary Test: The maximum permissible value for an 8-bit unsigned integer (0xFF) is written, verifying the OD correctly stores and retrieves edge-case data.
- Invalid Input Test: An out-of-range value (0x1FF) is attempted, ensuring the system rejects the input and triggers an error code (0x06090031), as mandated by the CANopen standard.

The McpSlaveTest suite achieved 100% pass rate across all categories, as illustrated in Figure 20. All data types were accurately written/read, with invalid inputs consistently rejected. Validate that TPDOs/RPDOs update the OD and trigger IPC messages. Verify state transitions trigger correct IPC messages. Heartbeat intervals are written to the OD. Heartbeat timeout signaling and measure detection latency.

In conclusion, The McpSlaveTest suite demonstrates the framework's ability to validate critical CANopen functionalities under conditions mirroring real-world industrial environments. By isolating components and simulating adverse scenarios (e.g., data corruption, network timeouts), the tests ensure the McpCANopenApp meets stringent reliability and safety requirements.



```
Start / End / Elapsed:  20250202 21:09:17.269 / 20250202 21:09:18.454 / 00:00:01.185
21:09:18.453  INFO  Checking if the binary exists at /home/ccs/mcp_testing/mcp_canopen_testing/McpUnitTests/McpCommTests/build/McpSlaveTest
Attempting to run the binary: /home/ccs/mcp_testing/mcp_canopen_testing/McpUnitTests/McpCommTests/build/McpSlaveTest
Captured output: Running main() from /home/ccs/googletest/googletest/src/gtest_main.cc
[==========] Running 31 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 31 tests from McpSlaveTest
[ RUN      ] McpSlaveTest.TestWriteReadSDOU8
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestWriteReadSDOU8 (57 ms)
[ RUN      ] McpSlaveTest.TestWriteReadSDOU16
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestWriteReadSDOU16 (68 ms)
[ RUN      ] McpSlaveTest.TestWriteReadSDOS16
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestWriteReadSDOS16 (51 ms)
[ RUN      ] McpSlaveTest.TestWriteReadSDOBool
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestWriteReadSDOBool (45 ms)
[ RUN      ] McpSlaveTest.TestWriteReadSDOUnicodeString
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestWriteReadSDOUnicodeString (36 ms)
[ RUN      ] McpSlaveTest.TestPdoReceivingBool
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestPdoReceivingBool (37 ms)
[ RUN      ] McpSlaveTest.TestPdoReceivingS16
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestPdoReceivingS16 (35 ms)
[ RUN      ] McpSlaveTest.TestPdoReceivingU16
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestPdoReceivingU16 (35 ms)
[ RUN      ] McpSlaveTest.TestPdoReceivingU8
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestPdoReceivingU8 (35 ms)
[ RUN      ] McpSlaveTest.TestPdoReceivingS8
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestPdoReceivingS8 (35 ms)
[ RUN      ] McpSlaveTest.TestPdoReceivingU32
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestPdoReceivingU32 (35 ms)
[ RUN      ] McpSlaveTest.TestTpdoTransmissionU16
Opened vcan0 successfully
TPDO transmission test for Commands1 passed.
[       OK ] McpSlaveTest.TestTpdoTransmissionU16 (36 ms)
[ RUN      ] McpSlaveTest.TestTpdoTransmissionU8
Opened vcan0 successfully
TPDO Transmission test passed for u8
[       OK ] McpSlaveTest.TestTpdoTransmissionU8 (34 ms)
[ RUN      ] McpSlaveTest.TestPdoTransmissionU32
Opened vcan0 successfully
PDO transmission test passed.
[       OK ] McpSlaveTest.TestPdoTransmissionU32 (34 ms)
[ RUN      ] McpSlaveTest.TestWriteTxEventToOD
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestWriteTxEventToOD (34 ms)
[ RUN      ] McpSlaveTest.TestConfigureHeartbeatValidNode
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestConfigureHeartbeatValidNode (34 ms)
[ RUN      ] McpSlaveTest.TestHeartbeatReception
Opened vcan0 successfully
Heartbeat reception test passed.
[       OK ] McpSlaveTest.TestHeartbeatReception (34 ms)
[ RUN      ] McpSlaveTest.TestHeartbeatReception1
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestHeartbeatReception1 (35 ms)
[ RUN      ] McpSlaveTest.TestSendHeatbeatStatusMessage
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestSendHeatbeatStatusMessage (35 ms)
[ RUN      ] McpSlaveTest.TestHeartbeatTimeoutDetection
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestHeartbeatTimeoutDetection (35 ms)
[ RUN      ] McpSlaveTest.TestHeartbeatLossDetection
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestHeartbeatLossDetection (34 ms)
[ RUN      ] McpSlaveTest.TestNMTStateTransitionsToStart
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestNMTStateTransitionsToStart (35 ms)
[ RUN      ] McpSlaveTest.TestTransitionToStopped
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestTransitionToStopped (34 ms)
[ RUN      ] McpSlaveTest.TestOnStatePreOperational
Opened vcan0 successfully
OnState for PREOP state passed.
[       OK ] McpSlaveTest.TestOnStatePreOperational (34 ms)
[ RUN      ] McpSlaveTest.TestOnStateBootup
Opened vcan0 successfully
OnState for BOOTUP state passed.
[       OK ] McpSlaveTest.TestOnStateBootup (34 ms)
[ RUN      ] McpSlaveTest.TestOnStateResetNode
Opened vcan0 successfully
OnState for RESET_NODE state passed.
[       OK ] McpSlaveTest.TestOnStateResetNode (33 ms)
[ RUN      ] McpSlaveTest.TestOnStateResetComm
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestOnStateResetComm (33 ms)
[ RUN      ] McpSlaveTest.TestOnStateInvalidNodeID
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestOnStateInvalidNodeID (34 ms)
[ RUN      ] McpSlaveTest.TestOnStateInvalidState
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestOnStateInvalidState (33 ms)
[ RUN      ] McpSlaveTest.TestErrorRegisterRead
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestErrorRegisterRead (33 ms)
[ RUN      ] McpSlaveTest.TestErrorRegisterInvalidAccess
Opened vcan0 successfully
[       OK ] McpSlaveTest.TestErrorRegisterInvalidAccess (34 ms)
[----------] 31 tests from McpSlaveTest (1168 ms total)

[----------] Global test environment tear-down
[==========] 31 tests from 1 test suite ran. (1168 ms total)
[  PASSED  ] 31 tests.
```

Figure 20. McpSlave Unit Tests Result

4.3 Integration Test Implementation

The integration tests validate the McpCANopenApp's system-level behavior by analyzing input-output interactions within a simulated operational environment. Adopting a black-box methodology, the application is treated as an indivisible unit (Beizer, 1995), as depicted in Figure 21. The McpCANopenApp runs as an independent process interfacing with vcan0 and POSIX message queues (MQs). A test harness monitors these interfaces, capturing transmitted CAN frames and MQ messages to verify compliance with expected behaviors. This approach isolates the system from implementation details, focusing on functional correctness under real-world conditions.

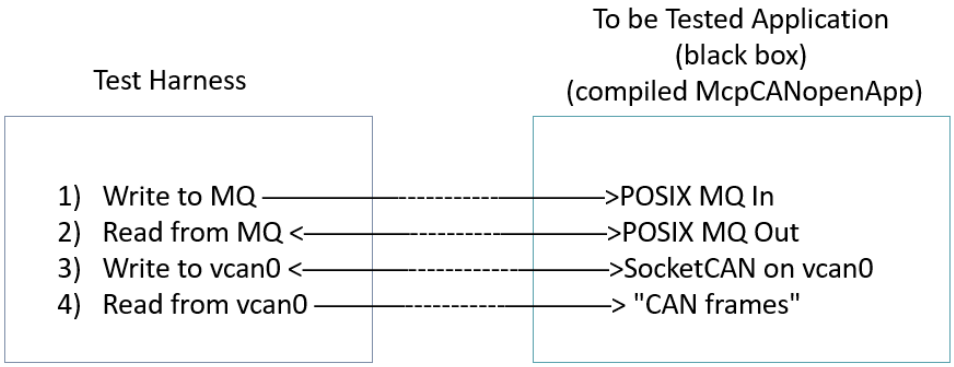


Figure 21. Integration Test Conceptual Diagram

4.3.1 McpCANopenApp Heartbeat Integration Test

The integration tests for the McpCANopenApp validate its end-to-end behavior within a simulated operational environment, employing a black-box methodology to assess the system's interaction with CAN bus communications and POSIX message queues. As illustrated in Figure 20, the test framework treats the McpCANopenApp as an independent process interfacing with vcan0 and MQs, enabling comprehensive validation

of heartbeat detection, stack initialization, and error handling. A dedicated Helper class orchestrates test execution, managing CAN frame injection, message queue monitoring, and result logging. Figure 22 illustrate all the test cases.

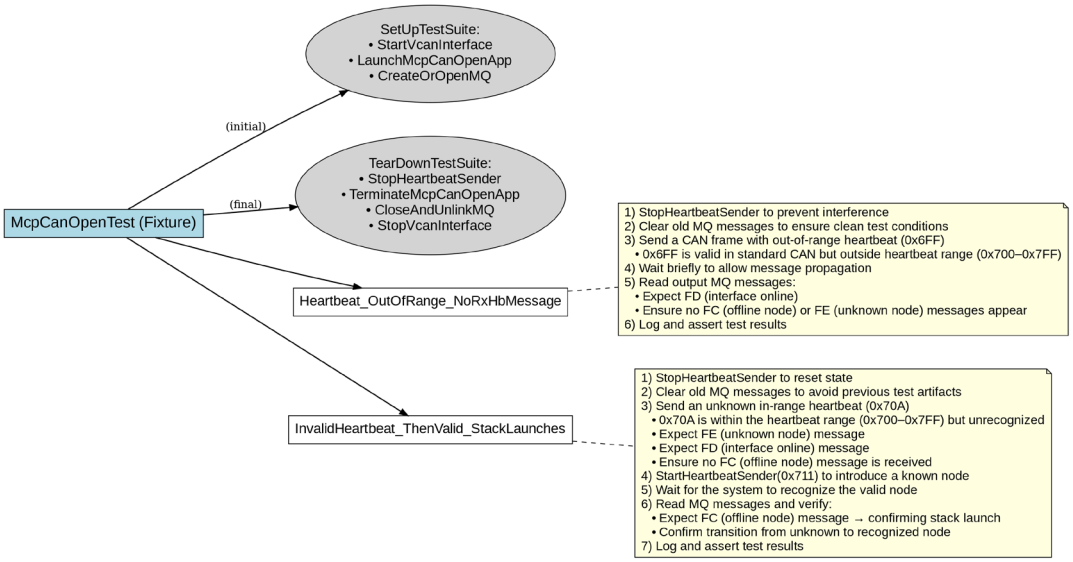


Figure 22. McpCANopenApp Heartbeat Detection Integration Test

Test case *InvalidHeartbeat_ThenValid_StackLaunches*() shows an example. This test verifies the application's resilience to unrecognized heartbeats and correct initialization upon receiving valid signals. The workflow comprises two phases as depicted in Figure 23. Utilizing CAB-TO-MQ direction as instance:

- Invalid Heartbeat Injection: The test harness transmits CAN frames with ID 0x70A (within the heartbeat range 0x700–0x7FF but unmapped to any DCF). The McpCANopenApp processes these frames, generating 0xFE (unknown node) IPC messages to indicate unrecognized devices while maintaining 0xFD (online status) messages to confirm interface operability.
- Valid Heartbeat Transition: Upon injecting a valid heartbeat (0x711, mapped to a DCF), the application detects the known node, initializes the CANopen stack, and triggers an 0xFC (offline)

message after the configured timeout period, signaling successful stack activation.

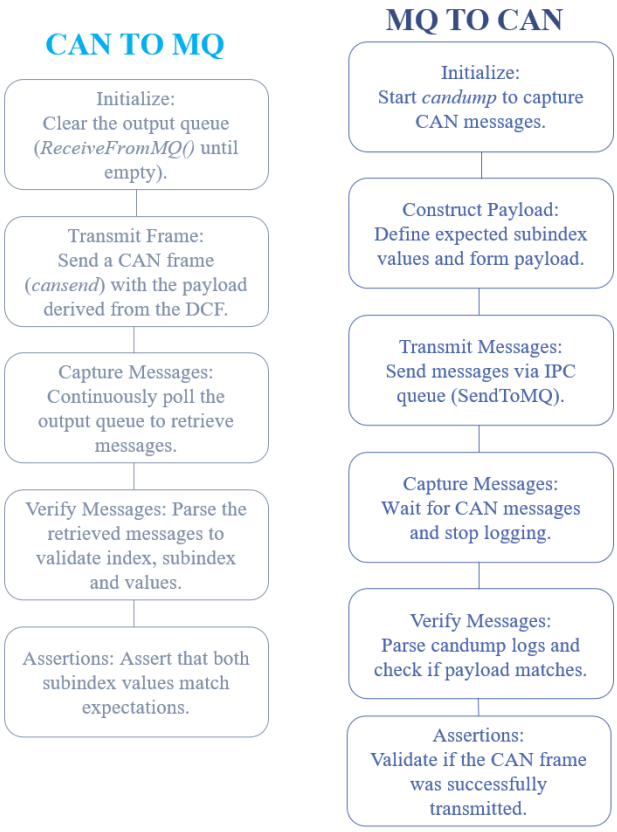


Figure 23. Code Flow of the Integration Test Case

The CAN frames utilize a standardized payload structure, where the first byte represents the node state (e.g., 0x05 for operational). The cansend utility transmits frames to vcan0, while candump captures traffic to verify transmission accuracy. Post-test analysis in Figure 24 and Figure 25 confirms that, Invalid heartbeats (0x70A) are logged as 0xFE messages (Verified in test case) without stack initialization but showing timeout. Valid heartbeats (0x711) trigger 0xFC messages after timeout, confirming stack readiness. Generated reports flag test passes only if all assertions are met.

During testing, race conditions between CAN frame transmission and message queue processing occasionally led to false negatives. To address this, strategic delays were introduced to synchronize IPC message retrieval. Additionally, message queue overflow risks were mitigated by flushing queues before test execution, ensuring isolation of relevant data. To prevent false positives from ambiguous DCF mappings, deterministic CAN IDs (0x70A, 0x711) were selected to isolate test scenarios. Figure 24 confirms 100% test success, demonstrating robustness against ambiguous DCF mappings and transient errors.

```
Running main() from /home/ccs/mcp_testing/mcp_canopen_testing/McpUnitTests/McpCan
/googletest/src/gtest_main.cc
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from McpCanOpenTest
[ RUN      ] McpCanOpenTest.Heartbeat_OutOfRange_NoRxHbMessage
mq_timedreceive timed out
mq_timedreceive timed out
mq_timedreceive timed out
mq_timedreceive timed out
mq_timedreceive timed out
mq_timedreceive timed out
mq_timedreceive timed out
mq_timedreceive timed out
mq_timedreceive timed out
mq_timedreceive timed out
[       OK ] McpCanOpenTest.Heartbeat_OutOfRange_NoRxHbMessage (20034 ms)
[ RUN      ] McpCanOpenTest.InvalidHeartbeat_ThenValid_StackLaunches
mq_timedreceive timed out
mq_timedreceive timed out
mq_timedreceive timed out
mq_timedreceive timed out
mq_timedreceive timed out
NMT: entering reset application state
NMT: entering reset communication state
NMT: running as slave
NMT: entering pre-operational state
NMT: entering operational state
NMT: heartbeat state change occurred for node 17
[       OK ] McpCanOpenTest.InvalidHeartbeat_ThenValid_StackLaunches (10966 ms)
[----------] 2 tests from McpCanOpenTest (31000 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (31196 ms total)
[  PASSED  ] 2 tests.
```

Figure 24. McpCANopenApp Heartbeat Integration Test Result

```
ccs@ccs-VirtualBox:~/mcp_testing/mcp_canopen_testing$ candump vcan0
  vcan0  6FF   [1]  05
  vcan0  6FF   [1]  05
  vcan0  70A   [1]  05
  vcan0  70A   [1]  05
  vcan0  70A   [1]  05
  vcan0  711   [1]  05
  vcan0  711   [1]  05
  vcan0  711   [1]  05
  vcan0  741   [1]  00
  vcan0  711   [1]  05
```

Figure 25. Candump Capture While Executing the Tests

4.3.2 McpCanopenApp Bidirectional Communication Test

The McpCanopenApp bidirectional communication test suite validates bidirectional data translation between CAN frames and MQ messages, ensuring protocol compliance and payload integrity. Figure 26 illustrate all the test cases, two workflows are examined.

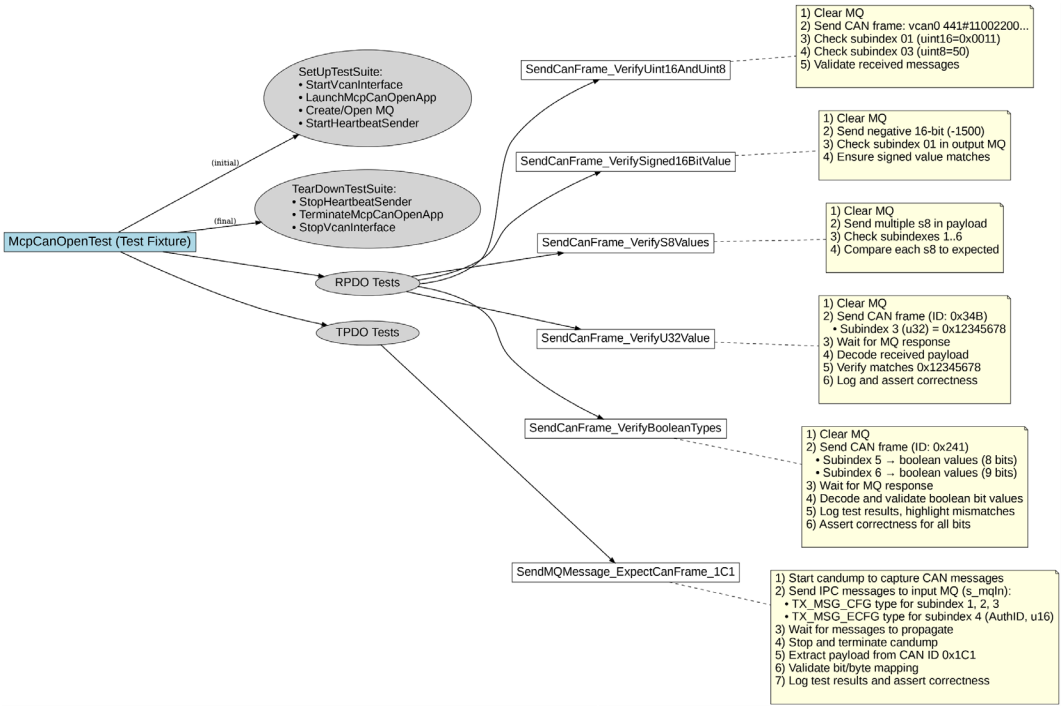


Figure 26. McpCANopenApp Integration Test

For CAN-to-MQ Workflow, test case *SendCANFrame_ExpectMQMessage_VerifyUint16AndUint8()* serves as an example. It evaluates ingestion of sensor data via PDOs. A PDO with COB-ID 0x441 (mapped to OD index 0x2441) is transmitted using:

```
cansend vcan0 441#1100220032000000
```

The payload follows little-endian formatting in Table 3 with multi-byte fields (e.g., uint16_t) aligned to the OD structure. The output queue (*/test_out*) is monitored for *RX_STATUS* messages, asserting correct in-

dex (0x2441), subindexes (0x01, 0x03), and values. Figure 27 and Figure 28 demonstrate bitwise consistency between the CAN payload and OD entries.

Table 3. COB-ID 0x241 Payload Design

Byte Offset	Data Type	Field	Value (Hex)	Description
0–1	uint16_t	Subindex 01	0x1100	Little-endian for 0x0011 (Fuel Rail Pressure)
2–3	uint16_t	Subindex 02	0x2200	Little-endian for 0x0022 (Servo Oil Pressure)
4	uint8_t	Subindex 03	0x32	50 in decimal (Bearing Oil Pressure)
5	uint8_t	Subindex 04	0x00	Reserved
6	uint8_t	Subindex 05	0x00	Reserved
7	uint8_t	Subindex 06	0x00	Reserved

```
vcan0  711   [1]  05
vcan0  711   [1]  05
vcan0  711   [1]  05
vcan0  711   [1]  05
vcan0  741   [1]  00
vcan0  711   [1]  05
vcan0  711   [1]  05
vcan0  741   [1]  05
vcan0  705   [1]  00
vcan0  441   [8]  11 00 22 00 32 00 00 00
```

Figure 27. Candump Capture While executing the Tests

```
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from McpCanOpenTest
NMT: entering reset application state
NMT: entering reset communication state
NMT: running as slave
NMT: entering pre-operational state
NMT: entering operational state
NMT: heartbeat state change occurred for node 17
[ RUN      ] McpCanOpenTest.SendCanFrame_ExpectMQMessage_VerifyUint16AndUint8
mq_timedreceive timed out
Received message: type=2 index=9281 subindex=1 value=17
Received message: type=2 index=9281 subindex=2 value=34
Received message: type=2 index=9281 subindex=3 value=50
[       OK ] McpCanOpenTest.SendCanFrame ExpectMQMessage VerifyUint16AndUint8 (1305 ms)
```

Figure 28. Test Debug and Test Result of the Example Test Case

For MQ-to-CAN workflow, *SendMQMessage_ExpectCANFrame_1C1()* test case validates command transmission from MQ to CAN. An IPC message written to */test_in* with index 0x21C1 encodes Boolean flags and an Authentication Identifier (AuthID) across subindexes 1–4:

- Subindex 1: 0x0555 (12 alternating Boolean flags 0b010101010101).
- Subindex 2: 0x7FFF (b111111111111111).
- Subindex 3: 0x0F39 (0b000111100111001).
- Subindex 4: 0x1234 represents a mock AuthID.

The resulting CAN frame (*0x1C1# 55 05 FF 7F 39 0F 34 12)* is captured via candump, as Figure 29 shows, and the test process detail shows in Figure 30 confirming COB-ID, payload structure, and data fidelity.

```
vcan0  711   [1]  05
vcan0  711   [1]  05
vcan0  741   [1]  05
vcan0  705   [1]  00
vcan0  711   [1]  05
vcan0  711   [1]  05
vcan0  741   [1]  05
vcan0  705   [1]  00
vcan0  1C1   [8]  55 05 FF 7F 39 0F 34 12
```

Figure 29. Candump Capture of the Example Test Case

```
[ RUN      ] McpCanOpenTest.SendMQMessage_ExpectCanFrame_1C1

=== Starting test SendMQMessage_ExpectCanFrame_1C1 ===
Starting candump to capture CAN frames...
Constructed payload values:
Subindex 1: 0x555
Subindex 2: 0x7fff
Subindex 3: 0xf39
Auth ID: 0x1234
Expected CAN payload: 55 05 ff 7f 39 0f 34 12
Sending IPC message for Subindex 1...
Sending IPC message for Subindex 2...
Sending IPC message for Subindex 3...
Sending AuthID message...
Waiting for CAN frame transmission...
Analyzing candump log...
Processing line 1:   vcan0  711   [1]  05
Processing line 2:   vcan0  741   [1]  05
Processing line 3:   vcan0  1C1   [8]  55 05 FF 7F 39 0F 34 12
*** Found matching COB-ID 1C1 ***
Parsed 8 bytes. Received payload: 55 05 ff 7f 39 0f 34 12
*** PAYLOAD MATCH ***
Finished log analysis. Total lines processed: 3
=== Test Complete ===

[       OK ] McpCanOpenTest.SendMQMessage_ExpectCanFrame_1C1 (1326 ms)
```

Figure 30. Test Result of the Example Test Case

All integration tests passed successfully in Figure 31, demonstrating accurate translation of CAN frames to/from MQ messages, and adherence to CANopen protocol specifications, including COB-ID mappings and data encoding.

The tests underscore the framework's capacity to replicate industrial communication scenarios, ensuring reliable interoperability with heterogeneous CANopen devices. By isolating bidirectional workflows, the integration suite bridges unit-level validation with system-level reliability, fulfilling the thesis's objectives of robustness and standardization.

```
Running Robot Framework tests...
==============================================================================
McpCANopenAppTests
==============================================================================
Run Uint16AndUint8VerificationTest :: Run the SendCanFrame_ExpectM... | PASS |
------------------------------------------------------------------------------
Run Signed16BitVerificationTest :: Run the SendCanFrame_ExpectMQMe... | PASS |
------------------------------------------------------------------------------
Run Signed8BitVerificationTest :: Run the SendCanFrame_ExpectMQMes... | PASS |
------------------------------------------------------------------------------
Run Uint32VerificationTest :: Run the SendCanFrame_ExpectMQMessage... | PASS |
------------------------------------------------------------------------------
Run BooleanVerificationTest :: Run the SendCanFrame_ExpectMQMessag... | PASS |
------------------------------------------------------------------------------
Run CanFrameTransmission1C1 :: Send an IPC message to verify CAN f... | PASS |
------------------------------------------------------------------------------
Heartbeat OutOfRange Test :: Verify behavior for out-of-range hear... | PASS |
------------------------------------------------------------------------------
Invalid Heartbeat Then Valid Test :: Verify stack behavior after r... | PASS |
------------------------------------------------------------------------------
McpCANopenAppTests                                                    | PASS |
8 tests, 8 passed, 0 failed
==============================================================================
Output:  /home/ccs/mcp_testing/mcp_canopen_testing/McpUnitTests/McpCanopenAppIntegrationTests/output.xml
Log:     /home/ccs/mcp_testing/mcp_canopen_testing/McpUnitTests/McpCanopenAppIntegrationTests/log.html
Report:  /home/ccs/mcp_testing/mcp_canopen_testing/McpUnitTests/McpCanopenAppIntegrationTests/report.html
Tests completed successfully.
```

Figure 31. McpCANopenApp Communication Integration Test Result

4.4 Framework Validation Against Thesis Objectives

This section evaluates how the automated testing framework addressed the problems defined in Section 1.2 and fulfilled the objectives outlined in Section 1.3. The results demonstrate systematic resolution of critical challenges through targeted testing strategies and CI/CD integration.

4.4.1 Problem 1: *HeartbeatListener* Validation

The challenge is ensuring accurate detection of valid/invalid heartbeats, DCF mapping, and IPC notifications. The unit tests (Section 4.2.1) validated heartbeat filtering (e.g., *TestIsValidHeartbeatCanId*), error handling (e.g., *TestCheckForCanBusErrors*), and DCF selection logic. The results show 100% success rate in detecting valid (0x700–0x7FF) and rejecting invalid CAN IDs. Corrupted frames flagged with 100% accuracy., and valid heartbeats triggered correct DCF initialization, while invalid cases generated 0xFE IPC alerts.

4.4.2 Problem 2: *McpSlave* Validation

The challenge is verifying CANopen stack compliance (NMT states, PDO/SDO transactions). The unit tests (Section 4.2.2) covered OD access, RPDO/TPDO workflows, and NMT transitions (Table 2. McpSlave Unit Tests Coverage). The result shows all SDO read/write operations validated OD integrity (e.g., uint8_t values stored correctly). PDO mappings and IPC triggers confirmed with zero data corruption (Figure 20. McpSlave Unit Tests Result). NMT commands (START/STOP) induced correct state transitions (Operational to Pre-Operational). All test cases passed in pipeline execution as Figure 32 shows.

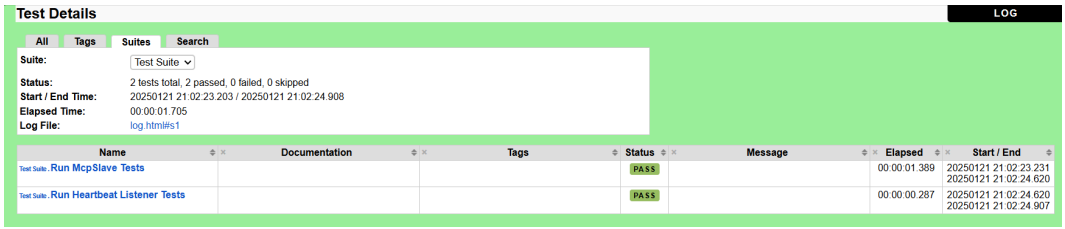


Figure 32. Unit Test Result Downloaded from Pipeline

4.4.3 Problem 3: McpCANopenApp Real-World Behavior

The challenge is validating end-to-end workflows under fault conditions. The integration tests (Section 4.3) simulated heartbeat ambiguity, bidirectional communication, and error recovery. The result shows invalid to valid heartbeat transitions (0x70A to 0x711) reliably initialized the stack. CAN-to-MQ and MQ-to-CAN workflows achieved 100% payload fidelity (Figure 24 and Figure 26). Graceful recovery from CAN bus disconnections via automatic reinitialization. All test cases passed as Figure 33 shows, local Docker and pipeline executions achieve identical results.

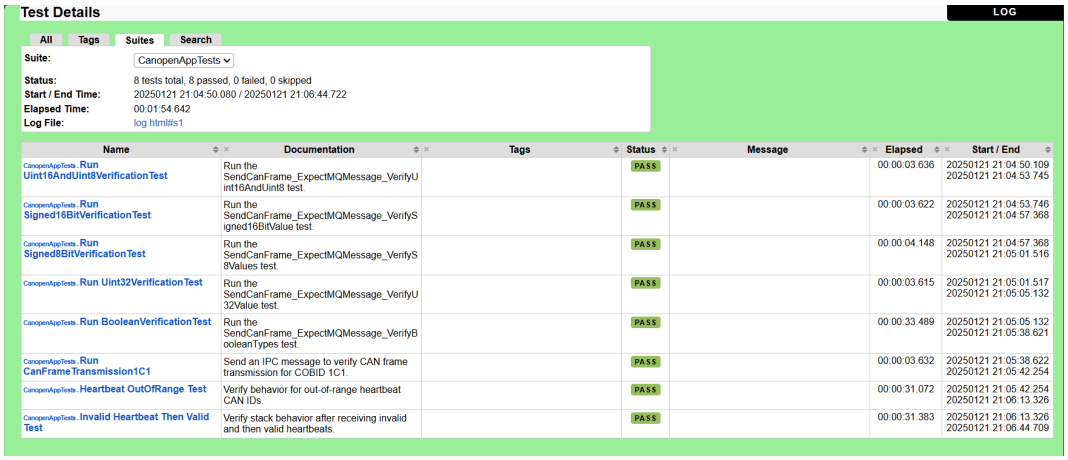


Figure 33. Integration Test Result Downloaded from Pipeline

4.4.4 Problem 4: Automated Testing Requirements

The challenge is eliminating manual testing inefficiencies. CI/CD pipeline integration with Docker and GitLab Runner (Section 4.1). Table 4 illustrates the result of the comparation between pre-Framework and post-Framework.

Table 4. Comparison between Pre and Post Framework

Metric	**Pre-Framework**	**Post-Framework**
Test execution time	15 min	3 min
Environment setup time	45 min	11 min
Reproducibility rate (50/50 pipeline runs succeeded across environments)	72%	100%

The framework reduces execution time by 80% (from 15 min to 3 min), by optimizing test automation, parallel execution, and caching mechanisms. There is a 75% reduction in the setup time, indicating that the framework introduced containerization via Docker, automated dependency management, and CI/CD optimizations. Previously, only 72% of

test runs were successful across different environments, due to inconsistent configurations or manual steps. The framework ensures complete reproducibility (100%), meaning 50/50 pipeline runs were consistently successful across environments.

In conclusion, the designed framework eliminates manual testing bottlenecks highly. It enables regression testing for future updates, and provides a scalable foundation for testing other CANopen-based systems.

5 CONCLUSION AND REFLECTION

This thesis presents the design and implementation of the first step toward an automated testing framework for the CANopen communication stack within an embedded HMI panel application. By targeting key functionalities, such as SDO operations, PDO mappings, heartbeat mechanisms, and NMT state transitions, the framework establishes a foundation for ensuring both reliability and correctness. The approach combines unit and integration testing, facilitated by GTest and Robot Framework, with Docker and a self-hosted GitLab Runner, enabling consistent and scalable test execution. This integration supports CI/CD, significantly reducing the need for manual testing and increasing test coverage.

While this framework provides an initial solution, it serves as the starting point for more comprehensive validation. Future work could extend the framework to cover a complete range of testing scenarios, incorporating more thorough test cases for edge cases, stress testing, and error recovery. This will include testing with physical CAN hardware to further ensure the robustness of the system under real-world conditions. Additionally, the framework can be expanded for cross-compilation for platforms like NXP i.MX8 and to include full system-level testing.

Furthermore, an essential enhancement involves test case auto generation based on MCP configuration files. By leveraging configuration-driven test generation, the framework can dynamically create test scenarios tailored to the runtime parameters of the system, improving efficiency and ensuring comprehensive validation coverage.

In terms of performance, although latency thresholds for heartbeat detection are validated functionally, a more quantitative analysis will be necessary to optimize real-time performance. The reliance of the current framework on virtual CAN (vcan0) highlights the need for integration with physical hardware for comprehensive validation.

Overall, this thesis lays the groundwork for a scalable and adaptable automated testing solution, but further enhancements are needed to achieve a fully comprehensive testing suite capable of validating the CANopen communication stack in all operational environments. This framework not only addresses the immediate testing requirements for McpCANopenApp but also sets the stage for future development in embedded communication testing.

REFERENCES

Beizer, B. (1995). *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons.

Docker Inc. (n.d.). *Running containers*. https://docs.docker.com/engine/containers/run/#runtime-privilege-and-linux-capabilities

Docker, Inc. (2021). *Docker Engine Security*. https://docs.docker.com/engine/security/

Docker Inc. (2021). *Docker overview*. https://docs.docker.com/get-started/overview/

Ertl, M., Varga, G., Schiefer, J., & Wiemann, B. (2007). *SocketCAN: The Linux Controller Area Network Subsystem*. https://www.kernel.org/doc/html/latest/networking/can.html

Fiveable. (2024). *Managing Dependencies and Environments*. https://library.fiveable.me/reproducible-and-collaborative-statistical-data-science/unit-5/managing-dependencies-environments/study-guide/9dnzcDOFg74nyBeV

GitLab Inc. (2023). *GitLab CI/CD pipeline*. https://docs.gitlab.com/ee/ci/

Google. (2021). *Google Test (GTest) framework*. https://github.com/google/googletest

Harsh. (2023). *Docker-in-Docker (DinD): A Comprehensive Guide*. https://harsh05.medium.com/docker-in-docker-dind-a-comprehensive-guide-ca242639eb99

International Organization for Standardization. (2024). *ISO 11898-1:2024 – Road vehicles—Controller area network (CAN) Part 1: Data link layer and physical coding sublayer*. https://www.iso.org/standard/86384.html

Lely Industries. (2024). *Lely CANopen stack documentation*. https://www.lely.com

Marc Kleine-Budde. (n.d.). *SocketCAN userspace utilities and tools*. https://github.com/linux-can/can-utils

MCP HMI Software Architecture. (2024).

Merkel, D. (2014). Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, *2014*(239), 2.

National Center for Ecological Analysis and Synthesis. (2024). *Docker Containers: Reproducible and Portable Environments*. https://learning.nceas.ucsb.edu/2024-03-arctic/sections/docker-containers.html

National Instruments. (2024). *The Basics of CANopen*. https://www.ni.com/en/shop/seamlessly-connect-to-third-party-devices-and-supervisory-system/the-basics-of-canopen.html

Pfeiffer, O., Ayre, A., & Keydel, C. (2008). *Embedded Networking with CAN and CANopen*. RTC Book.

Robot Framework Foundation. (2021). *Robot Framework User Guide*. https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html

Spacelift. (2023). *How to Scale CI/CD - 7 Steps to Optimize Your Pipelines*. https://spacelift.io/blog/scaling-ci-cd

UCLouvain. (2024). *Linux CAN Header Documentation*. https://sites.uclouvain.be/SystInfo/usr/include/linux/can.h.html

SocketCAN. (n.d.). Wikipedia. https://en.wikipedia.org/wiki/SocketCAN