الجمهورية الجزائرية الديموقراطية الشعبية
People's Democratic Republic of Algeria

وزارة التعليم العالي والبحث العلمي
Ministry of Higher Education and Scientific Research

المدرسة الوطنية العليا للتكنولوجيات المتقدمة
National Higher School of Advanced Technologies

Département Génie électrique Et Informatique Industrielle

Final Year Project to Obtain the Diploma of

Engineering

-Field -

Electronics

-Specialty-

Embedded Systems

- Subject -

Test and Validation of a motorcycle dashboard

Realized by

ELABASSI Mohammed Yassine

KHOULA Omar

Presented publicly, the 23/06/2025

Members of The Jury:

First LAST NAME	University	Grade	Quality
Mme.BOUCHAMA Samira	ENSTA	MCB	President
Mme.CHOUAF Selma	ENSTA	MAA	Examinator
Mr.CHETTAT Hicham	Externe	MCB	Examinator
Mme.ZELLAT Khadija	ENSTA	MCA	Supervisor
Mr.MAOUCHE Idris	Béjaïa University	Engineer	Supervisor

2024/2025

Abstract

It’s a presentation of the design and implementation of a Hardware-in-the-Loop (HIL) bench dedicated to testing a motorcycle dashboard. This project aims to simulate various motorcycle conditions by generating realistic analog and digital signals to verify the proper functioning of the dashboard in a controlled environment using the actual tools of Vector Informatik that are made for automotive test and validation. The bench consists of a microcontroller-based system (ESP32 and Arduino Nano), DAC modules, relays, and signal conditioning circuits, all coordinated together by custom firmware. This work highlights the challenges of embedded systems integration, real-time signal emulation and the importance of verification tools in the development lifecycle of automotive-grade electronics.

Keywords: Hardware In the Loop (HIL), motorcycle dashboard, test bench, CAN communication, test and validation

Resumé

Il s'agit d'une présentation de la conception et de la mise en œuvre d'un banc Hardware-in-the-Loop (HIL) dédié au test d'un tableau de bord de moto. Ce projet vise à simuler diverses conditions de conduite d'une moto en générant des signaux analogiques et numériques réalistes afin de vérifier le bon fonctionnement du tableau de bord dans un environnement contrôlé, à l'aide des outils professionnels, conçus pour les tests et la validation automobiles. Le banc est composé d'un système à base de microcontrôleurs (ESP32 et Arduino Nano), de modules DAC, de relais et de circuits de conditionnement de signaux, le tout coordonné par des firmwares personnalisés. Ce travail met en évidence les défis de l'intégration de systèmes embarqués, de l'émulation de signaux en temps réel et, bien sûr, l'importance des outils de vérification dans le cycle de développement de l'électronique automobile.

Mots-clés: Hardware In the Loop (HIL), tableau de bord moto, banc d'essai, communication CAN, test et validation.

ملخص

هذا لمشروع لتصميم وتنفيذ منصة مخصصة لاختبار لوحة قيادة دراجة نارية. يهدف هذا المشروع إلى محاكاة ظروف قيادة مختلفة للدراجة النارية من خلال توليد إشارات تناظرية ورقمية واقعية للتحقق من صحة تشغيل لوحة القيادة في بيئة مُتحكم بها، باستخدام أدوات احترافية في مجال السيارات، مُصممة لاختبار السيارات والتحقق من صحتها. تتكون المنصة من نظام قائم على وحدات تحكم دقيقة للترحيل ودوائر معالجة الإشارات، جميعها مُنسقة بواسطة برامج ثابتة مُخصصة. يُسلط هذا العمل الضوء على تحديات تكامل الأنظمة المُدمجة، ومحاكاة الإشارات في الوقت الفعلي، وبالطبع، أهمية أدوات التحقق في دورة تطوير إلكترونيات السيارات.

الكلمات المفتاحية. الأجهزة في الحلقة، لوحة قيادة الدراجة النارية، منصة الاختبار،شبكة ، الاختبار والتحقق

Acknowledgments

First of all, we would like to express our deepest gratitude to both of our parents for their unconditional love, endless support and of course the constant encouragement throughout this journey. Their belief in us has been the foundation of all my accomplishments in my life literally.

We would also like to sincerely thank our friends who stood by us during difficult times and celebrated every happy moment along the way.

Our heartfelt thanks go to Mr. Idris Maouche, our supervisor at Fibonova, for his exceptional guidance, valuable insights, and technical mentorship throughout the development of this project we are so grateful for the patience and professionalism he demonstrated during the entire journey.

A special thank you to Madame Zellat Khadija, our academic advisor at ENSTA for her continuous support, constructive feedback, and for trusting us even when we had delays in communication, her role was vital in shaping the academic quality of this work.

We also want to extend our sincere appreciation to Mr. Tahar Bensadoun, the founder of Fibonova whose vision, encouragement and personal involvement have made a profound impact on this project. His support throughout the journey has been both inspiring and also motivating.

To all of you, thank you for being part of this experience.

Content

List of tables

List of Figures

List of abbreviations

µs	Microsecond
ABS	Anti-lock Braking System
ADAS	Advanced Driver-Assistance Systems
ANSYS	Analysis System (software company)
ASIL	Automotive Safety Integrity Level
BMS	Battery Management System
CAN	Controller Area Network
CAN FD	CAN Flexible Data Rate
CANoe	CAN Open Environment
CAPL	Communication Access Programming Language
CI/CT	Continuous Integration/Continuous Testing
DAC	Digital to Analog Converter
DAC MCP4725	I2C DAC Module
DBC	Database CAN
DIA	Development Interface Agreement
DLC	Data Length Code (CAN frame length)
DUT	Device Under Test
ECU	Electronic Control Unit
ESP32	Espressif Systems 32-bit Microcontroller
FPGA	Field-Programmable Gate Array
GND	Ground
GPIO	General Purpose Input/Output
HARA	Hazard and Risk Assessment

HIL	Hardware-in-the-Loop
Hz	Hertz
I/O	Input/Output
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
ISO 26262	International Standard for Road Vehicles – Functional Safety
LCD	Liquid Crystal Display
LIN	Local Interconnect Network
LM324	Quad Operational Amplifier
MCP2515	Microchip CAN Controller
ms	Millisecond
NI	National Instruments
OEM	Original Equipment Manufacturer
OPC DA	OLE for Process Control Data Access
PCB	Printed Circuit Board
PWM	Pulse Width Modulation
RAM	Random Access Memory
RPM	Revolutions Per Minute
SIL	Software-in-the-Loop
SPI	Serial Peripheral Interface
TFT	Thin Film Transistor
VCC	Voltage at the Common Collector
VCU	Vehicle Control Unit
V-Model	Validation and Verification Model

GENERAL INTRODUCTION

General Introduction

In the automotive industry, the continuous evolution of embedded systems and the increasing complexity of Electronic Control Units (ECUs) necessitated robust methods for validation and testing. Traditional methods that rely solely on real-world testing are often time-consuming, expensive and may pose safety concerns which is particularly important when it comes to automotives… especially during early development phases. This also critical in the context of two-wheeled vehicles like motorcycles, where testing conditions can be even more variable and constrained.

> **Problematic:**
> *How can we validate and test the behaviour of a modern motorcycle dashboard effectively without direct access to the real vehicle, while ensuring high reliability, safety and cost-efficiency during development?*

To address this challenge, this thesis proposes the development and implementation of a Hardware-in-the-Loop (HIL) simulation bench specifically designed for a motorcycle dashboard. By mimicking real sensor signals and CAN bus communication using microcontrollers: the ESP32 and Arduino Nano, the objective is to enable full testing of dashboard functionalities so it can eventually get approved for manufacturing without requiring the physical motorcycle itself.

Finally, to provide a clear understanding of the project, this document is organized into three chapters. The first chapter outlines the general context, defines key concepts such as Hardware-in-the-Loop (HIL), and presents related works. The second chapter introduces the tools and technologies employed, including microcontrollers, communication protocols, and testing software. The third chapter describes the design and implementation of the HIL bench, covering system architecture, signal simulation, and test strategy. Finally, the report concludes with a general conclusion that summarizes the outcomes, addresses encountered challenges, and proposes future improvements and perspectives.

CHAPTER I:
Context and State of the Art

1.1 Introduction

This chapter lays the foundational understandings of the project by introducing its context, objectives and the concept of Hardware-in-the-Loop (HIL), we also provide insights into the architecture of the motorcycle dashboards and have a general view on the dashboard provided by Fibonova and the role of embedded systems in modern vehicles and to ensure our work is well-positioned, we included similar existing projects that we got the inspiration from to make our own prototype.

Fibonova is a startup based in Algeria, founded in 2023 by a team of engineers passionate about electronics and embedded systems. The company has a core team of four engineers and three co-founders.

Fibonova focuses on prototyping and developing embedded systems, particularly in the automotive and IoT domains. One of its notable prototypes includes a motorcycle digital dashboard designed to interpret sensor inputs and display real-time vehicle data.

1.2 General Context of the Project

Embedded systems play a crucial role in modern vehicles, enabling complex functionalities like real-time data acquisition, communication between components and intelligent decision-making. As motorcycles in particular become more advanced, the integration of electronic dashboards has become a standard replacing traditional analog meters with digital displays that provide enhanced information and interactivity to the rider.

However, testing such embedded systems poses several challenges. It is often impractical, costly, or unsafe to test real dashboards directly on a motorcycle during development. To address this, engineers rely on Hardware-in-the-Loop (HIL) simulation, a method that replicates the real-world environment by generating the same electrical signals that a dashboard would receive from an actual motorcycle.

This project was developed within this context of building a custom HIL test bench using low-cost components specifically designed to simulate a motorcycle's behaviour for testing a digital dashboard. The goal was to provide a reliable, reproducible, and safe environment for validating dashboard functionalities without relying on physical vehicle components.

This work was carried out as part of our final-year internship, a startup working on embedded solutions for the automotive sector.

1.3 Objectives of the HIL Bench

The primary objective of this project was to design and implement a Hardware-in-the-Loop (HIL) bench that could simulate the different signals typically received by a motorcycle dashboard. The purpose of this bench is to test and validate the functionality of the dashboard independently from the motorcycle.

Specifically, the HIL bench was designed to:

- Generate digital pulse signals that simulate wheel speed using a Hall-effect sensor model.
- Produce analog signals using DACs to replicate parameters such as fuel level and outdoor temperature.

- Transmit CAN messages to simulate real-time vehicle data such as RPM, battery voltage, and warning indicators.

- Create a safe and controlled testing environment where engineers can trigger specific spontaneous scenarios and observe the dashboard's response in real time.

- Reduce development time by providing a repeatable and adjustable signal source without needing physical movement or a live vehicle.

In summary, the bench acts as a substitute for the motorcycle, enabling efficient testing, validation, and demonstration of the dashboard under various conditions.

1.4 The concept of a HIL Bench

A Hardware-in-the-Loop (HIL) bench is a testing platform used to simulate real-world inputs for embedded systems, allowing developers to validate the behaviour of a system without needing the entire physical environment.

Key components of a HIL system include:

- Signal Generators for producing Analog, digital signals.
- Processing Unit which is typically a microcontroller or an embedded system (ESP32 and Arduino nano in this project).
- Test Management Software to control scenarios and monitor outputs (CANoe).
- Real Device Under Test (DUT): In this case, the actual motorcycle dashboard we worked with.

HIL testing is widely used in automotive, aerospace and industrial sectors due to its ability to reduce costs, accelerate development and enhance safety by eliminating the need for early physical prototypes or dangerous test scenarios.

1.5 Overview of Motorcycle Dashboards

Automotive Dashboards and motorcycles dashboards specifically have changed significantly over the two last decades from analog gauges to advanced digital displays as it is shown in the Fig.1.1. These dashboards, which are considered the primary interface for the rider, provide him with real-time data of his overall vehicle state and conditions such as speed, engine RPM, fuel level, temperature, and warning indicators (left and right indicators, headlights and RGB lights).

Traditionally, motorcycles used analog dials for speed and RPM, combined with small LED indicator lights for functions like turn signals, oil pressure, and high beam. However, with the integration of microcontrollers and sensors, modern motorcycles increasingly adopt digital dashboards or TFT (Thin Film Transistor) displays, which offer greater flexibility, better visibility, and additional features like phone dialing for example through bluetooth connectivity. The internal layout of a digital dashboard is almost the same on every model which is shown in the diagram below in Fig.1.2.

Advanced dashboards now include:

- Digital speedometers
- Fuel level and engine temperature displays
- CAN-based communication with the ECU (Engine Control Unit)
- GPS integration and turn-by-turn navigation
- Connectivity with smartphones for calls, music, and diagnostics
- Diagnostic warning systems and service reminders

Fig 1.1 Traditional analog dashboards and modern digital dashboards.

- On the upper photo we have a view of a classic analog dashboard (with mechanical speedometer).
- On the lower photo we have a view of a modern digital dashboard (TFT screen or LCD with multiple data fields).

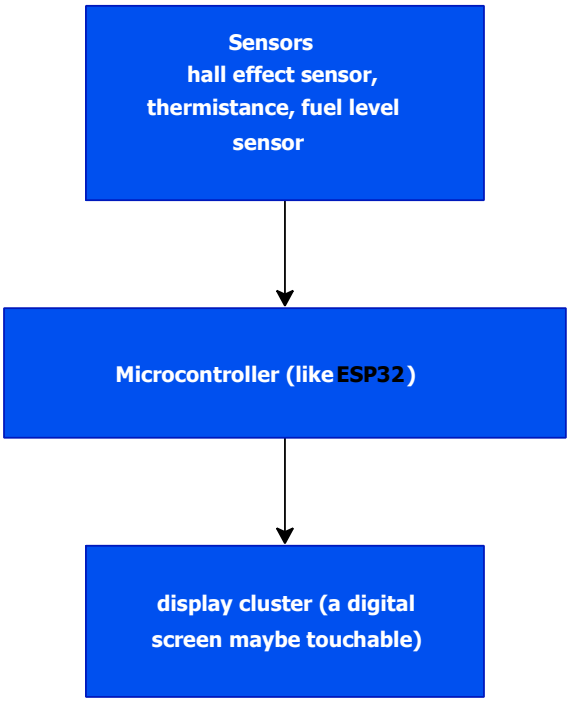

Fig.1.2. The internal functional layout of a digital dashboard.

As you can see in the diagram above, we can identify:

- Inputs (first case): Hall-effect sensors (speed), thermistors (temperature), CAN bus (ECU data).
- Processing (middle case): Microcontroller (ESP32, STM32, etc.).
- Outputs (last case): LCD/TFT screen showing processed data.

The change toward digital dashboards also enables the integration of custom firmware, which allows manufacturers to develop new dashboards that can be updated or personalized via their software. This opens the door for companies to design modular, software-driven displays made for specific requests by the customers.

In our project, we worked with a digital dashboard prototype, designed to read various types of input signals (digital, analog, and CAN) and then display them on a user-friendly screen. Dashboards like these require a reliable testing environment to validate its behaviour before deploying it on an actual motorcycle and then be labelled as approved for manufacturing which is the primary motivation for building our Hardware-in-the-Loop (HIL) bench.

1.6 Embedded Systems Overview

An embedded system is a dedicated computing system designed to perform a specific function, often within a larger system. Unlike general-purpose computers, embedded systems are made for real-time tasks, constrained environments, and direct interaction with a hardware.

In the context of motorcycles, embedded systems are integrated into critical systems like:

- The Engine Control Unit (ECU).
- Anti-lock Braking Systems (ABS).
- Lighting and signaling systems.
- Digital dashboards.

These systems typically include:

- A microcontroller or microprocessor (like the ESP32 used in our dashboard and our HIL bench).
- Memory components (Flash, RAM).
- Communication Interfaces: enable the exchange of data like (GPIO, ADC, PWM, UART, SPI, I2C, CAN).
- Sensors.
- Embedded software or firmware programmed to respond to inputs in real time manner.

The motorcycle dashboard developed and used in this project is itself a complete embedded system, responsible for:

- Reading input signals (speed pulses, fuel levels, engine temperature, lights and direction indicators).
- Processing data and translating raw sensor values (in voltages) into human-readable output.
- Managing and updating the display in real time.
- Handling interrupts, multitasking (with FreeRTOS), and communication protocols.

Also we can add that modern embedded systems are often characterized by:

- Low power consumption.
- Real-time performance.
- High reliability and fault tolerance.
- Compact size.
- cost-efficiency.

In the dashboard, it must perform accurately even when subjected to simulated inputs, just like it would on an actual motorcycle. That’s why it was necessary to build another embedded system called Hardware-in-the-Loop (HIL) test bench__ to detect any faulty inefficient results

and eventually be able to validate the behavior of this embedded system under realistic, and controlled conditions.

Furthermore, embedded systems development requires a careful coordination of hardware and software design which involves:

- Selecting suitable fast microcontrollers.
- Designing the circuit (power supply, connectors, protections).
- Writing and debugging real-time embedded code.
- Integrating communication protocols such as CAN, which is widely used in automotive systems due to its robustness and fault tolerance and rapid response rate.

In this project, the ESP32 serves as the heart of our test bench, compiling and running the logic that simulates inputs, receives CAN frames, and drives the display.

our HIL device emphasizes the importance of real-time signal handling, communication protocol support (such as in this case CAN) and a good embedded software architecture.

1.7 Similar Works and Safety Protocol

Hardware-in-the-Loop (HIL) testing is a necessary methodology in the validation of embedded systems, especially within the automotive sector. HIL test benches enable engineers to simulate real-world conditions by replacing actual hardware components with virtual models, which allows for the comprehensive testing of embedded controllers and dashboards without the need for a complete physical prototype [1]. This approach is widely adopted in the automotive industry for its ability to accelerate and organize development cycles, reduce costs and ensure system reliability and functional safety, particularly in terms of the safety protocol ISO 26262 [2] [3].

Commercial HIL benches, like those provided by Vector and STEP Lab, offer modular and high effective platforms capable of simulating complex vehicle behaviours and integrating multiple communication protocols especially the one used widely in automotives which is CAN [4] [5]. These systems are highly flexible, supporting the expansion of I/O interfaces and the automation of test scenarios, which is essential for the validation of advanced features in modern vehicles. For example, Link Engineering's HIL system can recreate dynamic vehicle environments, allowing for the real-time testing of brake systems and advanced driver-assist systems (ADAS) by integrating both physical hardware and simulated vehicle dynamics [6].

The academic research so far has also demonstrated the versatility of HIL test benches. Viennet et al. (2024) developed a HIL test bench for e-bike ABS validation, which highlights the importance of test simulation (such as rider mass and tire grip) to the specific application, which closely aligns with the needs of our motorcycle dashboard testing [7]. These studies emphasize that the level of model complexity must be balanced to avoid overloading the simulator while still achieving accurate system test and validation.

Compared to these established systems, our HIL test bench developed for the motorcycle dashboard project focuses on simulating key motorcycle sensor signals such as speed, fuel level, temperature and indicator lights. While commercial and high-end HIL benches take years to come to light and often target four-wheeler automotive platforms and utilize expensive proprietary hardware, our version in the other hand prioritizes cost-effective components and is also specifically adapted for two-wheeler applications (but can also be adapted for four-wheeled with further work) [2] [7]. makes our test bench particularly valuable for automotive startups and small manufacturers seeking affordable, flexible and rapid validation tools. Furthermore, our design supports modular signal expansion and real-time closed-loop testing, which shows loyalty to the core principles of HIL methodology while addressing the unique requirements of motorcycle dashboard systems [1] [3] [8].

1.7.1 The ISO 26262 protocol

We talked earlier about the ISO 26262 protocol but we didn't really explain what it really is and why is it so important in automotive test and validation so ISO 26262 is an international standard governing the functional safety of electrical and electronic (E/E) systems in road vehicles (excluding mopeds). It provides a risk-based framework to mitigate hazards caused by E/E system failures, ensuring safety across a vehicle's lifecycle**.** The Key Components of ISO 26262 are:

a). Safety Lifecycle

The standard defines three phases for safety-critical system development**:**

Phase	**Activities**	**Output**
Concept Phase	- Item definition (system boundaries) - Hazard and Risk Assessment (HARA) - Safety goal definition	Safety goals, ASIL classification
Product Development	- System/hardware/software design - Safety requirement decomposition - Verification/validation	Safety case, validated system
Production & Operation	- Manufacturing controls - Field monitoring - Decommissioning	Safety reports, incident management

Tab.1.1. The different phases of the safety lifecycle

Source: our own work starting from the link : "The lifecycle ensures safety is embedded from concept to decommissioning, with rigorous documentation and traceability [9] [10].

b). Automotive Safety Integrity Level (ASIL)

ASIL classifies risks based on three factors:

- Severity (S0–S3): Injury severity (e.g., S3 = life-threatening).
- Exposure (E0–E4): Likelihood of operational scenarios (e.g., E4 = frequent).
- Controllability (C0–C3): Driver's ability to avoid harm (e.g., C3 = difficult).

ASIL ratings (A–D) dictate safety measures:

- ASIL D: Highest integrity (like braking systems). Requires redundant architectures, formal methods, and fault injection [11] [12].
- ASIL C/B: Moderate (like power steering, BMS).
- ASIL A/QM: Low risk (like infotainment).

Our motorcycle dashboard's speed sensor might be ASIL B (moderate severity, rare exposure, controllable) [11][13].

c). Hazard and Risk Assessment (HARA)

HARA identifies hazards and defines safety goals:

- Item Definition: Describe system functions and boundaries (e.g., dashboard ECU).
- Hazard Identification: Brainstorm failure modes (incorrect speed display).
- Risk Classification: Assign ASIL using severity, exposure, controllability.
- Safety Goals: Top-level requirements (e.g "Speed signal deviation ≤ 5%") [9] [14].

d). V-Model Development Process

ISO 26262 mandates a V-Model as you can see in Fig.1.3 which we widely adopted in automotive industries:

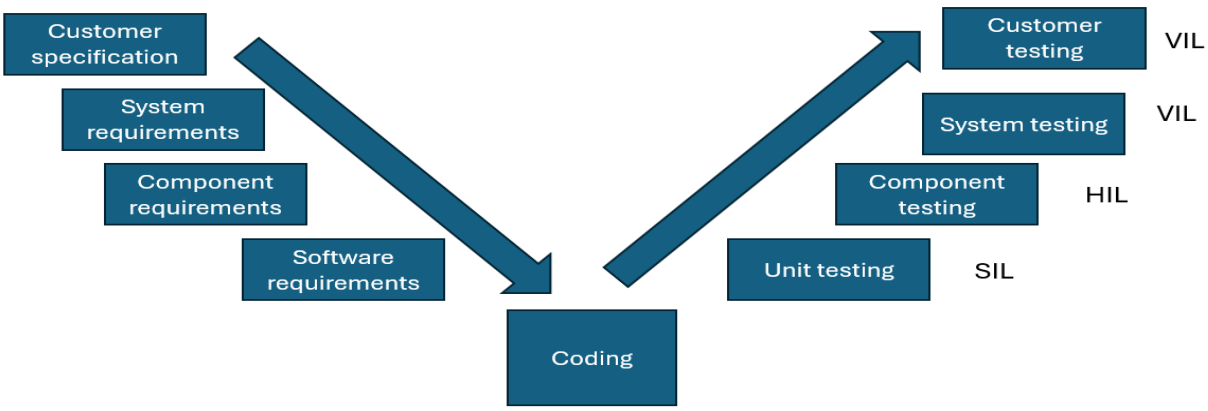

Fig.1.3. Diagram of the V cycle

- Left Side: Decomposing safety goals into technical requirements.
- Right Side: Validating each level (unit, integration, system) [10] [11].

Note: For ASIL D, semi-formal modelling (Simulink®) and simulation are required [12] [14].

e). Functional Safety Management

- Safety Plan: Outlines roles, milestones, and confirmation measures (e.g., audits) [13] [15].
- Development Interface Agreement (DIA): Defines responsibilities between suppliers and OEMs 14].
- Fault Injection: Simulates hardware/software failures (e.g., sensor disconnection) to validate robustness [10] [13].

1.7.2. Alignment of the HIL bench with ISO 26262

Furthermore, our motorcycle dashboard HIL bench aligns as well with ISO 26262 principles:

- **Validation**: Simulates sensor signals (speed, fuel) to validate dashboard responses, mirroring ISO 26262's verification requirements [13] [16].
- **Fault Testing**: By injecting edge-case signals (out-of-range temperatures and power supply exaggerated levels) we addressed ASIL B/C-level safety goals [10].
- **Documentation**: Documenting the test cases and results ensures compliance with the standard's lifecycle phases [14] [15].

1.8. Conclusion

In summary, this chapter established the foundational context of the project by highlighting the growing complexity of embedded systems in motorcycles and the need for robust testing methodologies such as Hardware-in-the-Loop (HIL). It introduced the main challenges in validating modern dashboards, presented the objectives of the proposed HIL bench, and reviewed relevant literature and standards like ISO 26262 that guided our approach. These insights justify the adoption of a custom HIL solution tailored to the needs of the dashboard. The next chapter builds upon this foundation by presenting the tools and technologies selected to implement the bench, including the microcontrollers, signal generation components, and testing platforms used throughout the project.

CHAPTER II:

Tools and Technologies Used

2.1 Introduction

In this chapter, we explore the main tools and technologies employed in developing the HIL test bench. Each component, whether hardware or software plays a very critical role in ensuring the accurate simulation and validation of dashboard signals. This includes microcontrollers, communication protocols, DAC modules, and professional automotive testing tools.

2.2 System Under Test General Description

To understand how our HIL bench work we first need to have a general view on our SUT (System Under Test] which in our case is the motorcycle dashboard. This latter serves as a compact embedded system designed to display critical information to the rider, the prototype is entirely signal-driven and uses direct analog and digital inputs for operation.

The hardware architecture includes:

- **ESP32-S3 Microcontroller:** is the central processing unit of the dashboard. It reads all incoming signals and controls the display logic. the ESP32-S3 offers advantages like built-in Wi-Fi and Bluetooth (reserved for future expansion), and dual-core processing power.
- **TFT Display Screen:** The dashboard includes a color screen for visualizing speed, indicators, fuel level, temperature, and other parameters. The display is managed directly by the ESP32-S3 card using SPI or another parallel interface.
- **Voltage Dividers:** Since the signals originating from the motorcycle are typically at 12V, voltage divider circuits are used to step them down to safe levels (usually 3.3V) compatible with the ESP32's GPIOs.
- **Demultiplexer Circuit:** Used to route multiple incoming signals to the limited number of GPIO pins present in the card which allows for efficient use of the ESP32's available I/O and adds more inputs than the microcontroller natively provides.
- **Direct Analog and Digital Inputs:** The card reads sensor data (like fuel level or outdoor temperature) using the ESP32's ADC pins, while pulse-based inputs (like speed from a Hall effect sensor) are handled using interrupt-capable digital pins.

For testing and development of this dashboard, a Hardware-in-the-Loop (HIL) bench was designed by us to simulate these inputs replicating a real motorcycle environment through a software/hardware-controlled signal generation. Below in Fig.2.1 we can see the main general diagram of the System Under Test.

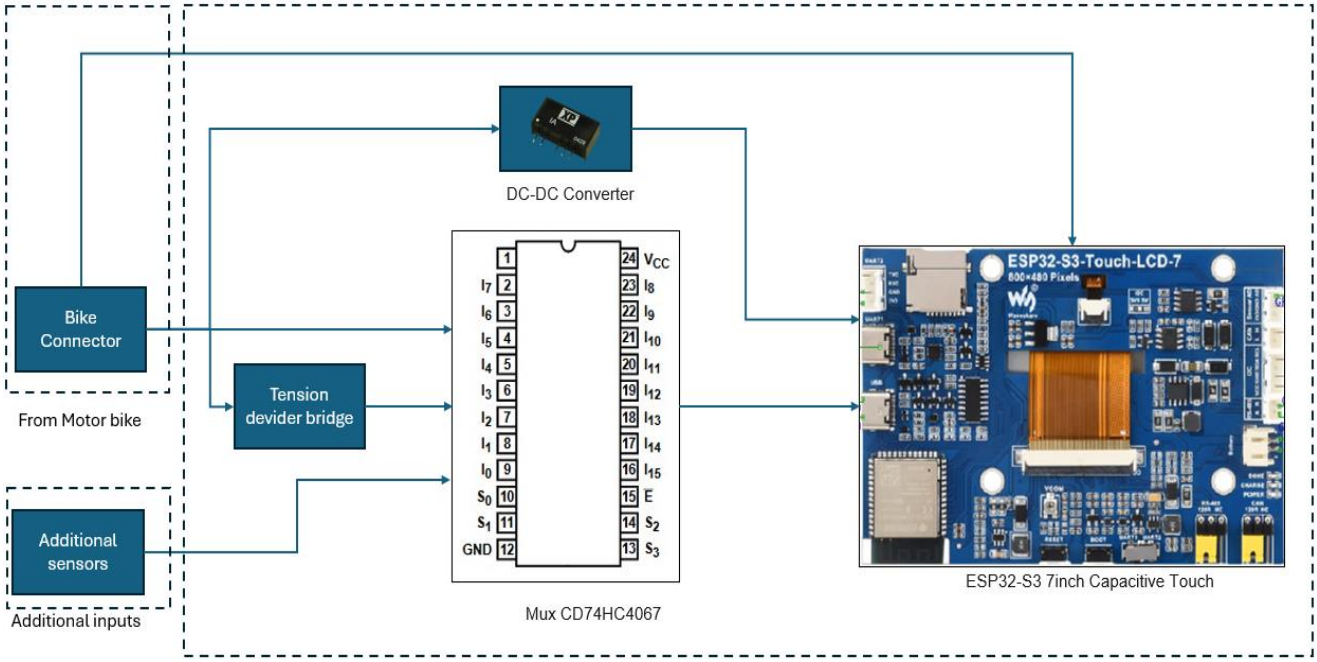

Fig.2.1. Diagram of the System Under Test

2.3 Presentation of ESP32

The ESP32 is a low-cost, low-power microcontroller series developed by Espressif Systems, it is widely adopted in both academic and industrial embedded systems projects. It has gained a lot of popularity due to its high performance with a processing speed up to 240Mz, rich peripheral set and integrated wireless capabilities, which makes it ideal for Internet of Things (IoT), real-time systems and also embedded applications like the ones implemented here in this project.

2.3.1 Core Architecture

The ESP32 is based on a dual-core Tensilica Xtensa LX6 or LX7 processor (depending on the model) capable of running up to 240 MHz with support for:

- Real-Time Operating Systems (RTOS).
- Floating point operations.
- Multitasking.
- On-chip memory (RAM and Flash).
- Power-saving modes.

2.3.2 Connectivity Features

A big advantage of the ESP32 family is its integrated wireless communication modules:

- Wi-Fi 802.11 b/g/n.
- Bluetooth 4.2 / 5.0 (Classic and BLE).

These allow the ESP32 to interface with remote services, mobile apps, or cloud platforms which makes it ideal for smart systems.

2.3.3 Peripheral Interfaces

The ESP32 includes a wide array of peripherals and interfaces as shown below Tab.2.1:

Interface	Description
GPIO	33 general-purpose I/O pins
ADC	Up to 18 channels of 12-bit ADC
DAC	2 × 8-bit integrated DAC outputs
SPI/I2C/I2S	SPI, I2C and I2S interfaces for communication
UART	Up to 3 UART interfaces
PWM	Pulse Width Modulation on any GPIO

Tab.2.1. Key specs of ESP32

In this project, two ESP32 boards are used: one in the motorcycle dashboard and another in the Hardware-in-the-Loop (HIL) bench, each fulfilling different roles.

2.3.4 Development E-ecosystem

The ESP32 benefits from a robust open-source community and tooling:

- Arduino IDE and PlatformIO for simple C++-based development.
- Espressif IoT Development Framework (ESP-IDF) for advanced development.
- Compatible with many third-party libraries like FreeRTOS, LovyanGFX and LittlevGL (LVGL).

2.3.5 Usage in The Project

The usage of this microcontroller can be summarized in:

Dashboard	Acquires signals (voltage levels, indicators), processes logic, and renders real-time data on a touchscreen.
HIL Bench	Simulates motorcycle behavior by generating digital, analog and receive CAN signals to validate dashboard functionality.

2.3.6 Advantages

ESP32 has many advantages, we can highlight the following:

- All-in-one solution: CPU, RAM, wireless, and peripherals on one chip.
- Cost-effective for both prototyping and production.
- Excellent documentation and community support.
- Flexible enough to support multitasking through his dual cores, signal processing and generating user interface rendering simultaneously.

2.4 Arduino Nano Presentation

The Arduino Nano is a compact and breadboard-friendly microcontroller board based on the ATmega328P microcontroller. It offers the same functionality as the Arduino Uno but in a much smaller form with lesser pins and a different shape, making it ideal for embedded systems and space-constrained designs like the one that is used in our motorcycle dashboard HIL bench.

2.4.1 Key Specs

The key specs of the ARDUINO Nano are:

- **Microcontroller**: ATmega328P.
- **Operating Voltage**: 5V.
- **Input Voltage (recommended)**: 7–12V.
- **Digital I/O Pins**: 14 (6 PWM outputs).
- **Analog Input Pins**: 8.
- **Clock Speed**: 16 MHz.
- **Flash Memory**: 32 KB (2 KB used by bootloader).
- **SRAM**: 2 KB.
- **EEPROM**: 1 KB.
- **USB Interface**: Mini-USB.

2.4.2 Arduino Nano Utility

In our project, the Arduino Nano was selected specifically to manage the power supply level signal via a dedicated DAC module connected through the I²C interface. Initially, all analog signals were planned to be generated by a single ESP32 but bandwidth limitations and the external DAC instability over the I²C bus caused fluctuation in the other signal output. To overcome this, the analog voltage generation for the power supply level was offloaded to a separate controller which is the Arduino Nano thereby ensuring more stable and isolated signal output to ensure maximu accuracy.

2.4.3 Role in the HIL Bench

the role of ARDUINO Nano can be summarized in:

- Receives CAN messages via the MCP2515 CAN module.
- Extracts the power supply voltage level from the CAN frame.
- Generates a corresponding analog voltage (0–5V) using a connected DAC module (MCP4725).
- Ensures signal stability by avoiding I²C interference with the ESP32 signals.
- Runs a dedicated firmware that receives the corresponding CAN message.

2.5 CAN Bus: Principles and Operation

The Controller Area Network (CAN) bus is well known because it is a robust, efficient and widely adopted communication protocol used in automotive and industrial systems. Originally developed by Bosch in the 1980s, CAN was designed to allow microcontrollers and devices to communicate with each other without the need for a host computer using a multi-master, message-based protocol.

Although the motorcycle dashboard does not originally use CAN, the HIL (Hardware-in-the-Loop) bench developed in this project employs the CAN protocol to simulate realistic in-vehicle communication using the CANOE software. Understanding the working principles of CAN is therefore essential for interpreting and generating test signals.

2.5.1 Communication Mode

CAN uses a multi-master architecture, meaning multiple devices (nodes) can initiate communication on the bus. The type of communication here is message-based rather than address-based like other protocols (I2C, SPI…). This allows every node to listen to all messages and react only to the ones relevant to them, identified by a unique message ID.

2.5.2 Electrical and Physical features

The electrical and physical features of a CAN bus are:

- **Bus Type**: Differential two-wire system (CAN_H and CAN_L) they need to be both present for the message to be sent.
- **Speed**: Common speeds include 125 kbps, 250 kbps and 500 kbps and up to 1 Mbps max for CAN 2.0 (CAN FD can go even faster).
- **Termination**: Requires sometimes 120-ohm resistors at both ends of the bus for signal integrity and to avoid fluctuations.
- **Voltage Levels**: it uses a differential signal to resist noise in automotive environments.

2.5.3 CAN Frame Structure

A typical **CAN 2.0 frame** (standard) includes as shown below in Tab.2.2:

Field	Description
Start of Frame (SOF)	Indicates the beginning of a message
Identifier (11 or 29 bits)	Determines message priority and content
Control Field	Includes data length
Data Field	Actual data (0 to 8 bytes for standard CAN)
CRC	Cyclic Redundancy Check for error detection
ACK	Acknowledge slot (receiver confirms message)
End of Frame (EOF)	Marks the end of the transmission

Tab.2.2. CAN data frame structure

A Lower ID = higher priority during collisions.

Below in Fig.2.2 is a representation of what a data CAN message may look like

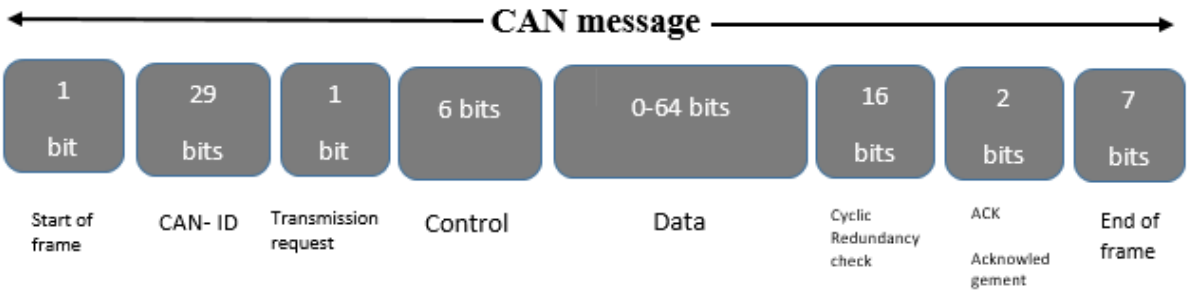

Fig 2.2: CAN message frame

2.5.4 CAN in the HIL Bench

Although the actual motorcycle dashboard is not based on CAN, the HIL test bench uses a CAN bus setup to:

- Send predefined frames representing engine RPM, speed, fuel level, or warning lights.
- Verify the dashboard's behaviour and visual outputs in response to specific values of the CAN messages.

The CAN messages were generated using a Vector CANoe and sent to the esp32 through a CAN case and a CAN module connected to the ESP32.

2.5.5 An example CAN message

A typical message to simulate Bike Speed looks like this:

- **ID**: `0x0CFF0500`
- **Data**: `03 4B 00 00 00 00 00 00` → Interpreted as 833 RPM
- **Periodicity**: 10 ms (real-time emulation)

This data is visualized on the real dashboard connected to the HIL bench, which provides feedback about system accuracy and responsiveness.

2.4.6 The main advantages of using a CAN communication protocol in automotives

The main advantages of using a CAN in the automotive industry are:

- **Reliability**: Real-time error detection with CRC and acknowledgment.
- **Scalability**: Easily integrates multiple nodes and test cases.
- **Noise Immunity**: Well-suited for electrically noisy environments (vehicles).
- **Non-destructive arbitration**: Messages don't collide; higher priority wins without interference.

2.6 DACs and Analog Signals

Digital-to-Analog Converters (DACs) play a crucial role in connecting digital microcontrollers with Analog hardware components. Many real-world devices like sensors, actuators and even automotive dashboards either produce or require Analog signals, and since most microcontrollers including the ESP32, operate using digital logic (0s and 1s), DACs are really necessary to transform digital outputs into corresponding voltage levels.

2.6.1 Principle of Operation

A DAC receives a digital value (typically an integer) and converts it into an Analog voltage that is proportional to that value. For example, an 8-bit DAC (like the one present in the ESP32) can take values from 0 to 255 and output a voltage between 0 V and a reference voltage provided by an engineer (e.g., 3.3 V). The output voltage V_{out} is given by:

$$V_{out} = \frac{D}{2^n - 1} \times V_{ref}$$

Where:

- D is the digital input value,
- n is the DAC resolution (typically 8, 10 or 12 bits),
- V_{ref} is the reference voltage (which is typically 3.3 V or 5 V).

2.6.2 Use of DACs in the HIL Bench

In the Hardware-in-the-Loop (HIL) bench developed for this project, DACs are used to simulate Analog inputs that would normally come from real sensors on a motorcycle. This includes the signals:

- **Fuel level.**
- **Temperature sensor output.**
- **Power Supply signal.**

To simulate these signals accurately, three DACs must provide stable and noise-free voltage levels. So in total there were two types of DACs used:

- **MCP4725 External DAC**: This is a 12-bit DAC module that communicates with the Arduino Nano via the I2C protocol (SDA and SCL). It provides high resolution and stable output for a signal like Power Supply Level, suitable for simulating analog signals required by the dashboard. As you can see below in Fig.2.3 is an illustration of what a MCP4725 DAC module may look like.

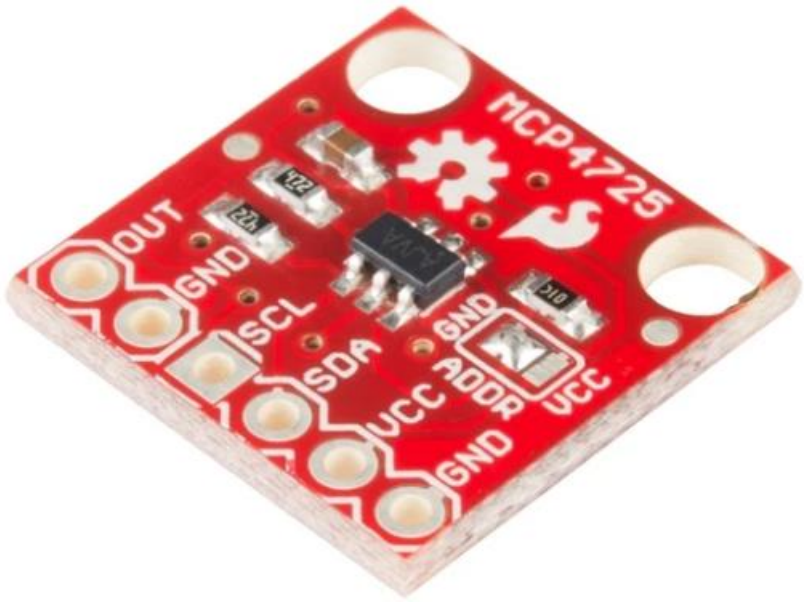

Fig.2.3.The MCP4725 DAC Module

- **Built-in ESP32 DACs**: The ESP32 microcontroller includes two 8-bit DACs on pins GPIO25 and GPIO26. These were used for less critical signals or because we can say simplicity was favoured over resolution for temperature signal and fuel tank level signal.

2.6.3 Signal Filtering and Smoothing

DAC output is usually passed through a low-pass filter (typically an RC circuit) to eliminate digital noise and to smooth the overall Analog waveform. This is especially important when the Analog voltage is used as an accurate reference to display logical values in the dashboard.

2.6.4 Calibration and Scaling

To ensure accuracy, each Analog output must be scaled to match the expected voltage range of the motorcycle dashboard. For instance, if the dashboard expects a fuel signal between 0.5 V (empty) and 4.5 V (full), the DAC output must be calibrated accordingly. This was achieved either through software scaling or with voltage dividers and operational amplifiers.

To fully understand how software scaling works which is the main course we followed to output the right values from our DACs here is a quick example of generating a 1V analog signal through an ESP32 and an external MCP4725 DAC module as shown in Fig.2.4 below.

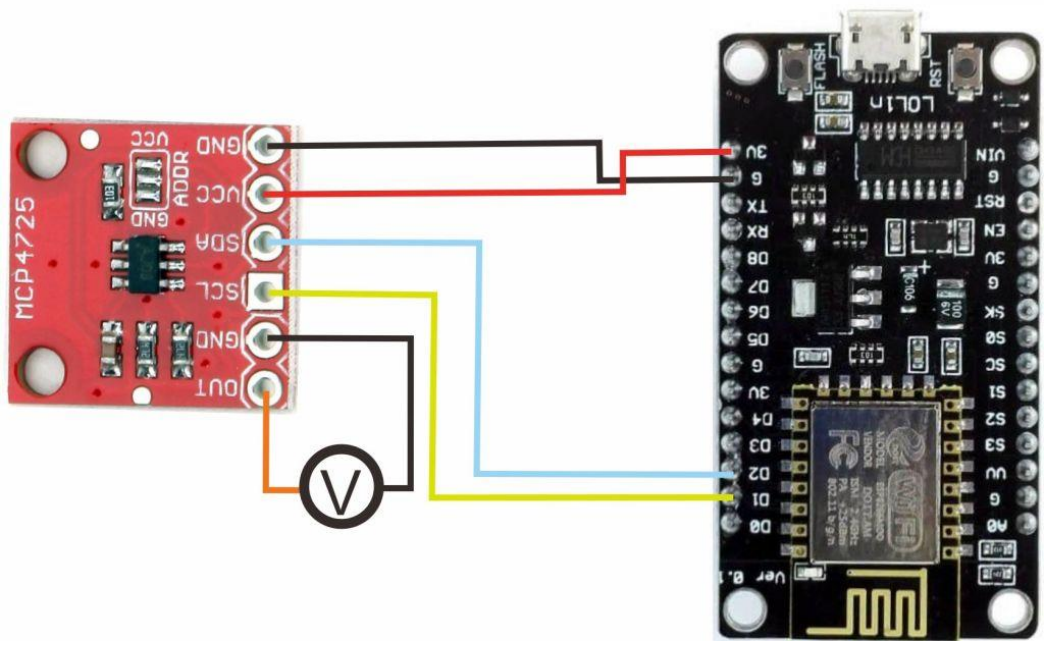

Fig.2.4. An example of a ESP32 connected to MC4725 through an I2C bus

Here is a simple code that generates a 1V through the DAC

```
1.  #include <Wire.h>
2.  #include <Adafruit_MCP4725.h>
3.
4.  Adafruit_MCP4725 dac;
5.
6.  // Set this value to 9, 8, 7, 6 or 5 to adjust the resolution
7.  #define DAC_RESOLUTION    (9);
8.
9.  void setup() {
10.   Serial.begin(9600);
11.   Serial.println("Hello!");
12.   dac.begin(0x60);
13. }
14.
15. void loop() {
16.   // put your main code here, to run repeatedly:
17.   dac.setVoltage(1 * 4096 / 3.3, false);
18. }
```

`#include <Wire.h>`: Includes the I2C communication library. The MCP4725 communicates over I2C.

`#include <Adafruit_MCP4725.h>`: This is the official Adafruit library for the MCP4725 DAC module.

`Adafruit_MCP4725 dac`: Creates an instance of the DAC.

Line 12: Initializes the MCP4725 DAC using its I2C address (default is `0x60`).

Starts communication between the ESP32 and the DAC.

Line 17: `setVoltage(value, persist);` sends a digital value to the DAC, which then converts it into an analog voltage.

1 * 4096 / 3.3: This line attempts to convert 1V into a digital value for a 12-bit DAC (range: 0–4095).

So, if we want 1 V out, and the reference voltage is 3.3 V:

$\frac{1}{3.3} \times 4096 \approx 1240$ This means the DAC will output a voltage close to 1V.

`false` as the second argument tells the DAC not to store the value in EEPROM (it will be gone on power off).

2.7 CANoe Software

CANoe (Controller Area Network Overall Environment) is a comprehensive development and testing tool by Vector Informatik for validating embedded systems, particularly automotive ECUs and networks. It supports simulation, analysis, and testing of CAN, LIN, FlexRay, Ethernet, and other protocols. Below is a breakdown of its core components and functionalities:

2.7.1 DBC File

A DBC file (CAN database) defines the structure of CAN bus communication, including:

- Messages (CAN frames with IDs and data).
- Signals (data fields within messages, e.g., speed, temperature).
- Nodes (ECUs transmitting/receiving messages).
- Encoding rules (scaling, offset, byte order).

The two most important elements in a DBC file are messages and signals.

a) Messages

A message represents a single CAN data frame transmitted on the bus. It is the main container of an information or multiple informations present in one or more signals.

- **Syntax:** Each message in **CAPL** is defined by a line starting with BO_.
- **Components:**
  - **CAN ID:** A unique identifier for the message, written in hexadecimal.
  - **Message Name:** A unique name for the message (1–32 characters).
  - **Length:** it's the total number of data bytes in the message (DLC).
  - **Sender:** The node (ECU) that transmits the message.
- **Example:**

```
BO_ 123 SPEEDM: 8 Vector__XXX
```

This line defines a message named "`SPEEDM`" with CAN ID `123`, length 8 bytes, sent by the node "`Vector__XXX`".

b) Signals

signals are the individual pieces of data contained within a message. Each signal represents a physical quantity present within a vehicule, such as speed, temperature, or status flags for example.

- **Syntax:** each signal is defined by a line starting with `SG_` and is always placed under a message definition.
- **Components:**
  - **Signal Name:** unique identifier for the signal.
  - **Bit Start:** the starting bit position of the signal within the message byte.
  - **Bit Length:** the number of bits the signal occupies.
  - **The endianness:** Byte order (`@1` for little-endian, `@0` for big-endian).
  - **The signedness:** Whether the signal is signed (-) or unsigned (+).
  - **Scaling and Offset:** Used to convert the raw value into a physical value (raw * scale + offset).
  - **Minimum/Maximum:** the range of the signal.
  - **Unit:** Physical unit ("km/h", "°C").
  - **Receiver:** Node(s) that use this signal.
- **Example:**

```
SG_ VehSpd: 39|12@0+ (0.05,0) [0|127.96875] 'm/s' InstrumentCluster
```

This defines a signal "`VehSpd`" starting at bit 39, 12 bits long, little-endian, unsigned with a scale of 0.05, offset 0, min 0, max 127.96875, unit "`m/s`" received by "`InstrumentCluster`".

CANoe uses DBC files to decode raw CAN data into human-readable signals for simulation, testing, and analysis.

In addition to defining messages and signals directly in the DBC file, you can also create and manage them easily using the CANdb++ Editor which is included with Vector CANoe, this graphical tool allows you to visually add, edit and organize all elements of our CAN database without manually editing the text files. While the best practices when creating a DBC file are as follows:

- Naming Conventions: Use descriptive names (Brake_Pressure instead of something like Sig 1).
- Signal Groups: Group related signals (Light_Indicators for turn signals, headlights).
- Validation: Use CANoe's Database → Validate to check for errors.

2.7.2 CAPL (Communication Access Programming Language)

CAPL is a C-like scripting language for automating tests, simulating ECUs and manipulating the bus traffic.

The Key CAPL Features are:

- Event-driven execution: Triggers on bus events (message reception).
- Signal manipulation: Modify signals in real time (inject faulty speed values).
- Test automation: Execute test sequences and generate reports.

2.7.3 ECU Simulation and Testing

CANoe simulates ECUs and network behaviors for validation and there are mainly to types of Testing Workflows:

- SIL (Software-in-the-Loop): Validate ECU software in a virtual environment.
- HIL (Hardware-in-the-Loop): Test physical ECUs with simulated sensors/actuators.

the motorcycle dashboard ECU can be tested by simulating CAN messages for speed, fuel level and also fault codes.

2.7.4 CANoe Interface and Modules

CANoe has several essential components that are used of the creation of test cases.

a) Core Interface Components

- Simulation Setup: configure ECUs, networks and I/O channels.
- Trace Window: to monitor raw and decoded bus traffic.
- Graphics Panel: create virtual dashboards for signal visualization .
- Test Module: to design automated test cases using CAPL

b) Hardware Integration

- It only supports Vector interfaces like the can case VN1630 for real-time HIL testing.
- FPGA-based models (like electric motor simulation) which enables high-fidelity testing.

2.7.5 Advanced Features

CANoe has some advanced features that are highlighted as follows:

a) Fault Injection

- Simulate bus errors, signal outliers or when an ECU disconnects.
- Validates error-handling logic in ECUs.

b) Physical Models

- Prebuilt models (Vehicle Dynamics, Electric Motor) simulate mechanical systems.
- Example: Simulate PWM signals for a motorcycle’s fuel gauge.

By understanding CANoe's architecture, we can then contextualize our work within the industry standards while highlighting its elite innovation.

2.8 Vector VN1630: Professional CAN Interface

The CAN case VN1630 made by VECTOR Informatik is an essential tool to ensure communication between the HIL bench and the CANoe software. It's a sophisticated equipment that has several features and capabilities.

2.8.1 Hardware Specifications

The Vector VN1630 is a professional-grade CAN interface device from the VN1600 family, it is designed for industrial and automotive testing applications. It features 4 channels specifically made for flexibility and I/O support which makes it suitable for complex testing environments. The device connects to a host computer via USB, drawing power directly from the USB connection, which greatly simplifies setup in laboratory environments.

The VN1630 supports multiple CAN protocols including standard CAN, CAN FD (Flexible Data-rate), and in newer versions, CAN XL, allowing for communication speeds up to 5 Mbit/s for CAN FD this makes it suitable for modern automotive applications requiring high-speed data transfer.

2.8.2 Key Features and Capabilities

The key features of the VN1630 are:

- **Multi-Application Support**: it allows multiple software applications to access the same CAN channel simultaneously.
- **Synchronized Channels**: provides minimal latency times with high timestamp accuracy (within one device: 1µs).
- **Hardware-Based Flash Routine**: it enables fast CAN flashing for ECU programming.
- **Digital/Analog I/O**:it includes dedicated D-B9 connector (CH5) for digital-analog input/output tasks.

2.8.3 Software Integration

The VN1630 is designed to work seamlessly with Vector's software suite but these softwares can never be used for testing without the CAN Case provided by Vector, these softwars are:

- **CANoe**: For comprehensive network design, simulation, and testing.
- **CANalyzer**: For bus monitoring and analysis.
- **CANape**: For ECU calibration and diagnostics.

This integration creates a complete ecosystem for automotive development and testing, particularly valuable in Hardware-in-the-Loop (HIL) testing environments especially with CANoe. As you can see below in Fig.2.5 is an illustration of what the VN1630 CAN case may look like.

Fig.2.5.The VN1630 CAN Case by Vector Informatik

2.9 MCP2515: Embedded CAN Controller

The MCP2515 is a stand-alone CAN controller manufactured by Microchip Technology that implements CAN network.

The MCP2515 operates at speeds up to 1 Mbit/s and communicates with microcontrollers via a high-speed SPI interface (up to 10 MHz). It usually operates on a voltage range of 2.7V to 5.5V with a typical active current consumption of only 5mA and standby current of 1 μA in sleep mode. It's compatible with both Arduino and ESP32.

2.9.1 Key Features and Capabilities

Despite its lower cost compared to professional solutions, the MCP2515 offers several valuable features:

- **One-Shot Mode**: it makes sure message transmission is attempted only once.
- **Interrupt Support**: it provides a configurable interrupt output pins for event notification.
- **Clock Features**: it usually includes a clock out pin with a programmable prescaler that can serve as a clock source for other devices.

2.9.2 Integration with and ESP32 Arduino and DIY Projects

The MCP2515 is widely used in Arduino-based CAN bus projects due to its affordability and ease of integratio. Typical Arduino CAN modules combine the MCP2515 controller with a transceiver (often the TJA1050) on a single board with SPI pins for connection to microcontrollers.

This configuration allows Arduino boards as well as ESP32 microcontrollers to communicate with automotive CAN networks or other CAN-enabled device.

2.9.3 System Architecture Implementation

In our HIL test bench the communication flow follows a very sophisticated chain that makes a bridge of professional automotive testing tools with cost-effective embedded systems. The system architecture of the HIL bench implements a three-stage communication process: Starting with CANoe that simulates generated CAN messages then:

- the Vector VN1630 interface transmits these messages through the DB9 connector.
- the MCP2515 CAN controller receives and processes the data (as you can see below in Fig.2.6 is an illustration of what the MCP2515 CAN module may look like).
- Finally the ESP32 and the Arduino Nano microcontrollers manipulate the received signals for test simulation.

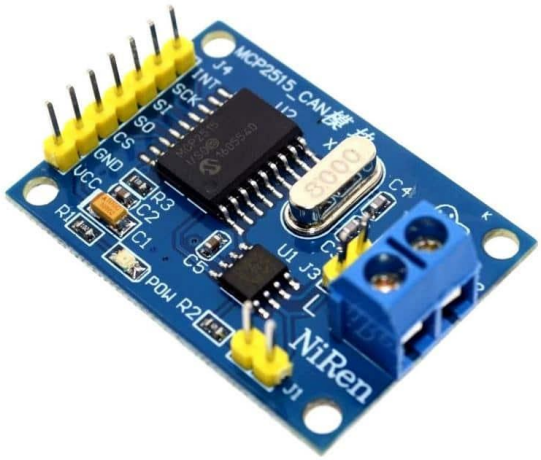

Fig.2.6 The MCP2515 CAN Module

2.9.5 ESP32 Integration and Data Manipulation

The ESP32 integration in our system is explained as follows:

a) SPI Communication Interface

The MCP2515 communication with the ESP32 microcontroller through the high-speed SPI interface enables efficient transfer of received CAN messages for further processing. The typical connection configuration uses the ESP32's hardware SPI pins for optimal performance and below in Tab.2.3 is the complete pinout of the MCP2515 and ESP32 used in our project:

MCP2515 Pin	ESP32 Pin	Function
CS	GPIO5	Chip Select
SI	GPIO23	SPI MOSI (Master Out, Slave In)
SO	GPIO19	SPI MISO (Master In, Slave Out)
SCK	GPIO18	SPI Clock

MCP2515 Pin	ESP32 Pin	Function
INT	GPIO4	Interrupt signal for message reception

Tab.2.3. Pinout of MCP2515 with ESP32

b) Message Processing and Manipulation

Once the ESP32 receives CAN messages from the MCP2515, our firmware can extract and manipulate the data according to our HIL test requirements. The ESP32's dual-core architecture provides sufficient processing power to handle the real-time CAN message processing while simultaneously generating appropriate analog and digital signals for the motorcycle dashboard. This manipulation capability allows us to simulate various motorcycle operating conditions, fault scenarios, and edge cases that would be difficult or dangerous to reproduce with the actual vehicle hardware.

2.9.6 Arduino Nano Integration and Data Manipulation

It’s basically the same with ESP32 but with different pinout as you can see below in Tab.2.4:

MCP2515 Pin	Arduino Nano Pin	Function
CS	D10	Chip Select
SI	D11	SPI MOSI (Master Out, Slave In)
SO	D12	SPI MISO (Master In, Slave Out)
SCK	D13	SPI Clock
INT	D2	Interrupt signal for message reception

Tab.2.4. Pinout of MCP2515 with Arduino Nano

The Arduino Nano’s role in this HIL setup is to act as a dedicated CAN-to-analog converter for the specific signal of power supply level. therefore, it also needs its second separate CAN module.

2.10 DB9 Connector: The Physical Interface Standard

The DB9 (D-SUB 9) connector serves as the industry-standard physical interface for CAN bus communication in professional automotive testing environments. This 9-pin male connector follows standardized pinout configurations that ensure compatibility across different manufacturers and testing platforms, the connector's robust design provides a reliable signal integrity and electromagnetic compatibility which makes it ideal for automotive and industrial applications where the signal quality is critical.

2.10.1 Standard DB9 CAN Pinout Configuration

The DB9 connector in the VN1630 follows the CiA 303-1 standard pinout, which is widely adopted in industrial and automotive testing applications as follows in Tab.2.5:

Pin Number	Signal	Description
1	NC	No Connection
2	CAN_L	CAN Low differential signal line
3	GND	ground
4	NC	No Connection
5	SHIELD	Optional cable shield connection
6	GND	ground
7	CAN_H	CAN High differential signal line
8	NC	No Connection
9	VB+	Optional external power supply (+9V to +30V)

Tab.2.5. Pin mapping of the DB-9 connector

And below in Fig.2.7 is schematic of the entire pin mapping

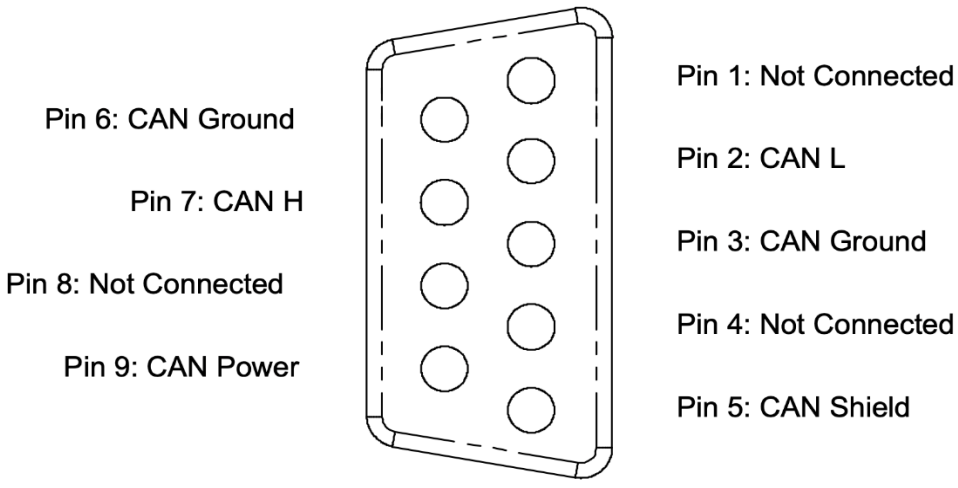

Fig.2.7. A schematic of the Pin Mapping of the DB-9 connector

This pinout configuration can ensure a proper differential signalling between CAN_H (pin 7) and CAN_L (pin 2), which is essential for noise immunity and reliable data transmission over

extended distances. The dual ground connections (pins 3 and 6) provide enhanced signal integrity and electromagnetic compatibility.

2.10.2 Vector VN1630 CAN Message Transmission

The VN1630 connects must use the DB9 connector to ensure CAN transmissions successfully.

a) VN1630 Output Methode

The Vector VN1630 interface generates CAN messages from CANoe simulation and outputs them through its DB9 connectors with professional-grade signal conditioning. The device supports CAN 2.0B protocol at speeds up to 1 Mbit/s, with CAN FD capabilities extending to 5 Mbit/s depending on network configuration. The VN1630's built-in CAN transceiver provides the necessary voltage levels and current drive capabilities to ensure reliable signal transmission through the DB9 interface. As you can see below in Fig.2.8 how the DB-9 connects to our CAN case through a 120 Ω resisor to ensure signal stability.

b) Signal Conditioning and Output Characteristics

The VN1630 uses high-speed CAN transceivers that convert the digital CAN protocol data into differential analog signals suitable for transmission over the physical CAN bus. These transceivers provide the necessary signal conditioning including voltage level conversion and protection against electrical faults such as short circuits and overvoltage conditions. The device maintains precise timing characteristics with timestamp accuracy within 1µs, ensuring deterministic message transmission which is critical for HIL testing applications.

Fig.2.8. A picture of a DB-9 connector connected to VN1630 CAN Case

2.10.4 MCP2515 CAN Controller Integration

The integration of the MCP2515 CAN module in the system is explained as follows:

a) MCP2515 Message Reception

The MCP2515 CAN controller in our test bench serves as the bridge between the professional VN1630 interface and ESP32 microcontroller. When connected to the VN1630's DB9 output, the MCP2515 receives the differential CAN signals. The MCP2515 implements the complete CAN 2.0B protocol stack handling message filtering, error detection and CAN Id reading.

b) Hardware Connection Configuration

The physical connection between the VN1630 DB9 output and the MCP2515 module requires proper signal routing and termination:

- **CAN_H (DB9 pin 7)** connects to the CAN transceiver's CANH input.
- **CAN_L (DB9 pin 2)** connects to the CAN transceiver's CANL input.
- **CAN_GND (DB9 pins 3/6)** provides the reference ground for the CAN signals.
- **120Ω termination resistor** between CAN_H and CAN_L ensures proper signal integrity and avoids fluctuations.

The MCP2515's message filtering capabilities allow selective reception of specific CAN IDs relevant to our motorcycle dashboard simulation, which reduces processing overhead and also improves system efficiency.

2.11 Additional Components

In addition to the technologies we explained in the previous sections we used three other much less sophisticated basic components but rather essential to ensure the full functioning of our HIL bench, the 8-relay module, the bike connector and the LM324 amplifier IC.

a) A brief explanation of the 8-Channel Relay Module

An **8-channel relay module** as shown in Fig.2.9 allows a microcontroller like the ESP32 to **amplify simple ON and OFF signals (**3.3V or 5V **indicators or headlights)** to a higher voltage depending on the power supplied to the relay switches.

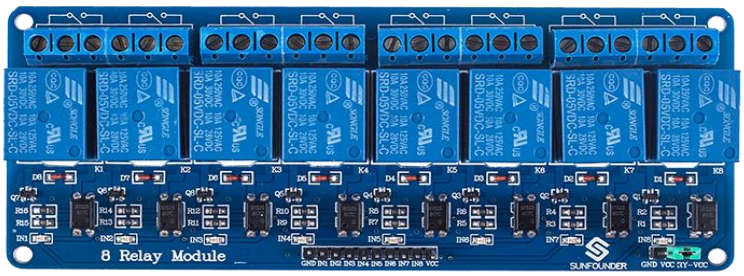

Fig.2.9. An image of the 8 relays module

Each relay on the module is an electromagnetically-operated switch. When the ESP32 sends a HIGH or LOW signal to one of the relay input pins(each input pin correspond to each relay IN1

corresponds to D1 and etc..), the corresponding relay opens or closes its contacts allowing 12V from an external power supply to reach the target device (in this case, dashboard signal lines).

Each relay typically supports Normally Open (NO) and Normally Closed (NC) terminals and can handle 10A at 12V DC or 220V AC and also has optocouplers for isolating microcontroller logic from the high-voltage side.

Four out of eight relays are used in this module and they are controlled by pins 33, 27, 32 and 15 of the ESP32. Used to amplify ON/OFF signals (Digital HIGH/LOW) to actual 12V levels.

The signals that control the relay are:

- Right Indicator.
- Left Indicator.
- Headlights.
- Key Contact.

We used relays because the dashboard expects real-world 12V signals, and ESP32 can only output 3.3V logic which is not enough to simulate the actual bike dashboard conditions so relays basically act as the bridge between simulation logic and real voltage levels.

b) A brief explanation of the 9-pin Connector

This is a **male connector** (9-pin) as shown below in Fig.2.10 that groups and routes all the output signals from our HIL bench to the motorcycle dashboard. It ensures easy plug-in and plug-out with its female twin during tests. We used one because:

- It makes the system user-friendly.
- It prevents wiring errors.
- Mimics the **actual connector used between the dashboard and the motorcycle** for signal interfacing.

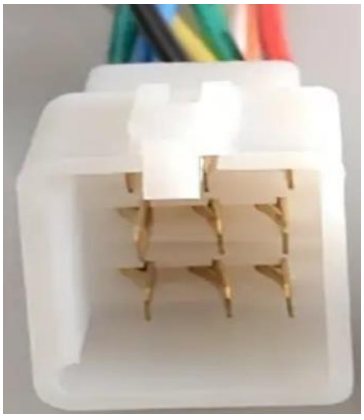

Fig.2.10. An image of the male 9 pins bike connector

Pin Mapping of the bike connector

The signals meant for simulation are carefully assigned to each pin with a specific number by following a Pin Mapping shared between both the male bike connector and its female encounter as follows in Tab.2.6 :

Pin	Signal Name	Description
1	Ground	Common reference ground
2	Right Indicator	12V digital ON/OFF signal (via relay)
3	Left Indicator	12V digital ON/OFF signal (via relay)
4	Headlights	12V ON/OFF signal (via relay)
5	Key Contact	12V ON/OFF signal (via relay)
6	Fuel Tank Level	0-12V analog signal (amplified from DAC)
7	Outdoor Temperature	0–3.3V analog signal from ESP32 DAC
8	Bike Speed Signal	0–10.8V digital pulse (amplified from ESP32)
9	Supply level	To indicate the battery level on the screen as well as power the entire dashboard

Tab.2.6. Pin Mapping of the Bike Connector

2.11.3 A brief explanation of LM324 Operational Amplifier

The LM324 is a quad op-amp IC, it contains four independent operational amplifiers inside one integrated circuit.

we used one as to:

- To amplify analog signals generated by DACs (ESP32 or MCP4725) to higher voltage levels, 15V 12V and 10.8V.
- used for:
  - Fuel Tank Level → Amplified to 12V
  - Bike Speed Pulse → Amplified to ~10.8V
  - Power Supply level → Amplified to ~15V

I**mportant Note:**

Since LM324 is not rail-to-rail, we had to power it with 16V to ensure it could output a clean 12V without distortion or clipping.

2.12 Conclusion

By understanding the tools and technologies involved in the development of this HIL bench we have built a strong technical foundation for our system. Each element was selected based on compatibility and precision. This chapter provides the technical prerequisites necessary for understanding the architecture and design choices detailed in Chapter 3.

CHAPTER III:
Design of the HIL Bench

3.1 Introduction

This chapter details the design, implementation, and integration of all components within the HIL bench. It explains the methods used for simulating analog and digital signals, the firmware structure, and how CAN frames were generated and interpreted using CANoe. Additionally, the test strategy and mechanical integration are presented.

3.2 System Overview

The Hardware-in-the-Loop HIL bench developed in our project aims to simulate **8** real-time motorcycle signals that in an actual motorcycle come from different sensors in a controlled, reproducible environment. The main purpose of this is to test and validate the behavior of the motorcycle dashboard designed by the team without the need for a physical motorcycle. Below in Fig.3.1 is a general diagram that illustrates the principle functioning of our HIL bench.

This concept is rather necessary in the automotive manufacturing word as it is adapted by every car company in the world as we have already seen in chapter 1.

The bench inside-architecture is composed of several connected modules, each of them responsible for emulating a specific kind of inputs that are typically present in a motorcycle:

- **Digital Signals** (speed pulses): Generated using the ESP32 to simulate the output of the Hall-effect proximity sensor used for speed detection.
- **Analog Signals** (fuel level, outdoor temperature and: Produced via DAC modules (MCP4725 and the **ESP32**'s internal DAC) to mimic varying voltage levels coming from the sensors.
- **Power supply level signal :** this particular signal was produced using an extra microcontroller which is the famous ARDUINO NANO.

The bench also adaptes the CAN network for simulation and testing using the hardware and software provided by **Vector** company:

- **CAN Messages**: Sent using the MCP2515 module and coming essentially from the Vector CAN Case VN1630, these messages are all sent via the CANoe Software.
- **Testing Interface**: CANoe software is used as the testing interface on the PC, which allows real-time visualization and manipulation of CAN messages which are later transferred to analog and digital signals through our HIL bench.

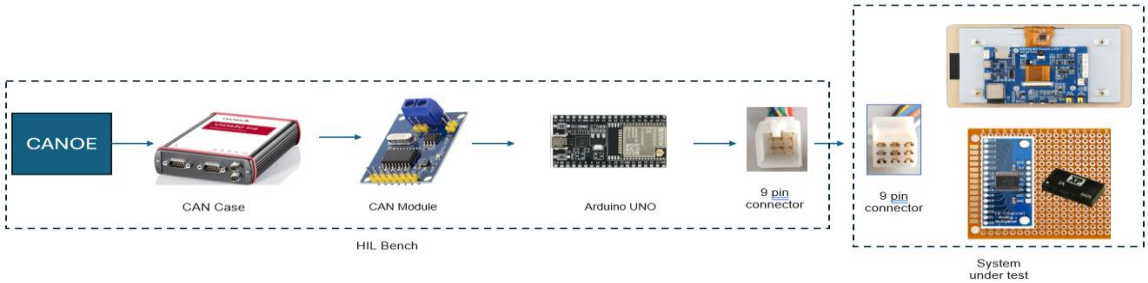

Fig.3.1. The HIL bench principle

3.3 Signal Simulation

The eight real-time motorcycle signals that we mentioned above are the main variables of our HIL test bench that need to be carefully replicated to provide a safe testing environment for the dashboard. These output signals from our HIL bench are finally grouped through a male 9 pins bike connector which will be then directly inserted to its female twin which is attached to the dashboard (the System Under Test). These main signals that exist on actual motorcycle and were then mimicked in the HIL test bench are as follows:

3.3.1. Right & Left Indicators

a) On an actual motorcycle

1. The rider flips a mechanical switch for left or right turn.
2. This switch activates a flasher relay, which turns 12V on and off periodically in a form of a square wave.
3. The indicator bulbs flash accordingly.
4. The dashboard simply detects this 12V blinking signal after reducing it to 5v through internal voltage dividers, and then displays them on the screen.

b) HIL Mimicking

1. CANoe sends a message (ID 0x111) indicating whether the left/right indicator is on or off.
2. Our ESP32 toggles a GPIO pin HIGH/LOW at approximately 1Hz.
3. A relay module takes that 3.3V signal and outputs a real 12V blinking signal periodically.
4. the signal is then plugged into the male bike connector.

• We don’t need to mimic the physical switch or relay but only the resulting 12V blinking output.

3.3.2. Headlights

a) On an actual motorcycle

1. Activated by a toggle switch.
2. Sends a constant 12V to the headlights and the dashboard.

b) HIL Mimicking

1. ESP32 receives the ON/OFF state over CAN (ID 0x111).
2. GPIO 27 is set HIGH or LOW.
3. Relay module amplifies that to real 12V.

3.3.3. Key Contact / Ignition ON

a) On an actual motorcycle

1. When you insert and turn the ignition key, it connects the battery to the rest of the bike.
2. This sends 12V to the dashboard to indicate power is on, the dashboard reduces it to 5v before indicating it on the screen.

b) HIL Mimicking

1. ESP32 receives the key contact state via CAN.
2. GPIO 15 controls a relay to output a 12V ignition signal.

3.3.4. Fuel Tank Level

a) On an actual motorcycle

1. A float inside the fuel tank is connected to a variable resistor (potentiometer).
2. As fuel level drops, the float moves changing resistance.
3. This is wired in a voltage divider, outputting a voltage between 0–12V.
4. The dashboard reads this analog voltage after reducing it again to 5v to estimate how full the tank is.

b) HIL Mimicking

1. CANoe sends fuel level in litters via CAN (ID 0x103).
2. ESP32 maps this to a DAC value and outputs it on pin 25 (0–3.3V).
3. A non-reversible amplifier using one of LM324 four op-amps amplifies this voltage to 0–12V using a calculated gain with the formula:

$$V_{out} = V_{in} \times (1 + \frac{R_f}{R_i})$$

So, the final gain is:

$$G = 1 + \frac{R_f}{R_i}$$

So, a gain from 3.3v to 12v is 3.63 therefore we need $R_f = 26.3\boldsymbol{k}\Omega$ and $R_i = 10\boldsymbol{k}\Omega$.

Below in Fig.3.2 we can see a schematic of the non-reversible amplifier made using the open source software KiCad.

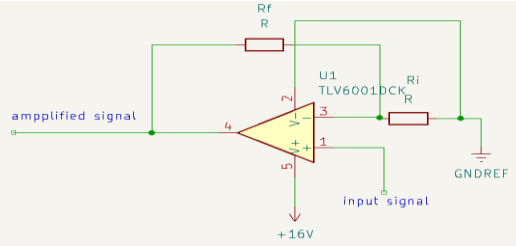

Fig.3.2. A non-reversible amplifier

3.3.5. Outdoor Temperature

a) On an actual motorcycle

1. Uses an NTC thermistor (Negative Temperature Coefficient).
2. As the temperature increases the resistance decreases.
3. Placed in a voltage divider → outputs voltage that drops as temperature rises.
4. The dashboard converts this analog voltage to temperature.

b) HIL Mimicking

1. CANoe sends temperature value via CAN (ID 0x101).
2. ESP32 maps this to voltage and outputs it on DAC pin 26.

- We don't need amplification because the dashboard expects 3.3V max.

3.3.6. Bike Speed Sensor

a) On an actual motorcycle

Uses a Hall effect sensor mounted on the wheel, this is how a hall effect sensor work:

1. A magnet passes the sensor (which stays in HIGH state) on every rotation therefore produces a digital LOW pulse.
2. The dashboard counts pulse frequency (how many pulses are there per second) to determine speed.

b) HIL Mimicking

1. CANoe sends speed in km/h (CAN ID 0x104).
2. ESP32 calculates the frequency f:

$$f = \frac{speed\ aqcuired\ by\ CANoe\ (in\ m/s)}{circumference\ of\ the\ bike\ wheel\ (in\ m)} = \frac{v}{1.29}$$

It then converts this into microsecond delays which toggles GPIO 14 at the right frequency.

by doing:

$$f_{micro} = \frac{10^6}{f}$$

The time in which a single pulse stays in LOW state corresponds exactly to the time period it takes the actual magnet to completely pass the sensor which is approximately:

$$LOWtime = \frac{22000}{speed\ aqcuired\ by\ CANoe\ (in\ m/s)}$$

There for the period which is $\frac{1}{f}$ is equal to the sum of $LOWtime$ and $HIGHtime$.

- Output is amplified with another voltage divider across an LM324 op-amp to ~10.8V to match the sensor levels, using the same formula used above, the proper gain is:

  $$G \approx 3.27$$

  Which means we will use resistors $R_f = 33k\Omega$ and $R_i = 10k\Omega$.

3.3.7 Power Supply Level

a) On an actual motorcycle

1. The battery voltage is monitored by the dashboard.
2. Normally 12V–15V.
3. It is usually sensed via a resistive voltage divider or an ADC in the dashboard.

b) HIL Mimicking

1. Arduino Nano receives this value via CAN (ID 0x107).
2. Uses an I²C bus MCP4725 DAC to output 0–5V.
3. It's then amplified through the LM324 op-amp to 0-15v using a gain of 3.00.
4. That's interpreted as 0–15V by the dashboard (with internal scaling again).

We have chosen to use another microcontroller for this signal because of:

- The limited number of hardware DACs on the ESP32 (only 2).
- Signal fluctuation and delay when an I2C DAC was used on the ESP32 due to I2C bus speed bottlenecks.

3.3.8 Summary

The summary of all the signals here in Tab.3.1:

Signal	Sensor Type	Real Output	Bench Output (Mimic)	Generated By	Amplified
Right Indicator	Switch + flasher	Pulsed 12V	Relay from GPIO 33	ESP32	Yes
Left Indicator	Switch + flasher	Pulsed 12V	Relay from GPIO 32	ESP32	Yes
Headlights	Toggle switch	Constant 12V	Relay from GPIO 27	ESP32	Yes
Key Contact	Key switch	Constant 12V	Relay from GPIO 15	ESP32	Yes
Fuel Level	Float + resistor	Analog 0–12V	DAC + LM324 from GPIO25	ESP32	Yes
Temperature	NTC thermistor	Analog 0–3.3V	DAC directly on GPIO26	ESP32	No
Speed Sensor	Hall effect	Pulsed 12V	GPIO14 + LM324	ESP32	Yes
Supply Voltage	Battery voltage	Analog 0–15V	DAC via MCP4725	Arduino Nano	Yes

Tab.3.1. Summary of the different signals meant for simulation

3.4 Hardware Design

The Hardware-in-the-Loop (HIL) bench designed in this project aims to test and validate a motorcycle dashboard under realistic operational conditions by simulating both digital and analog signals corresponding to real-world sensor inputs. The goal is to ensure the dashboard reacts accurately to all signals as if it were mounted on an actual motorcycle. This specific section provides an exhaustive explanation of all the design decisions, components and interactions.

The complete HIL system architecture includes the following subsystems:

- **CAN Communication Bridge** using a CAN Case device VN1630.
- **A dual Microcontroller System:**
  - **ESP32**: Handles real-time signal generation and decoding for most dashboard signals.
  - **Arduino Nano**: Dedicated to the output for the power supply level signal.
- **Digital and Analog Signal Generation** for simulating bike speed, fuel level, temperature and power supply level as well as key contact, left and right indicators and headlights.
- **Signal Amplification** using an LM324 Op-Amp.
- **8 Relay Module** for ON and OFF digital signals.

The complete schematic of the HIL test bench using KiCad is shown down below in Fig.3.3 with all the different components used and their interconnections.

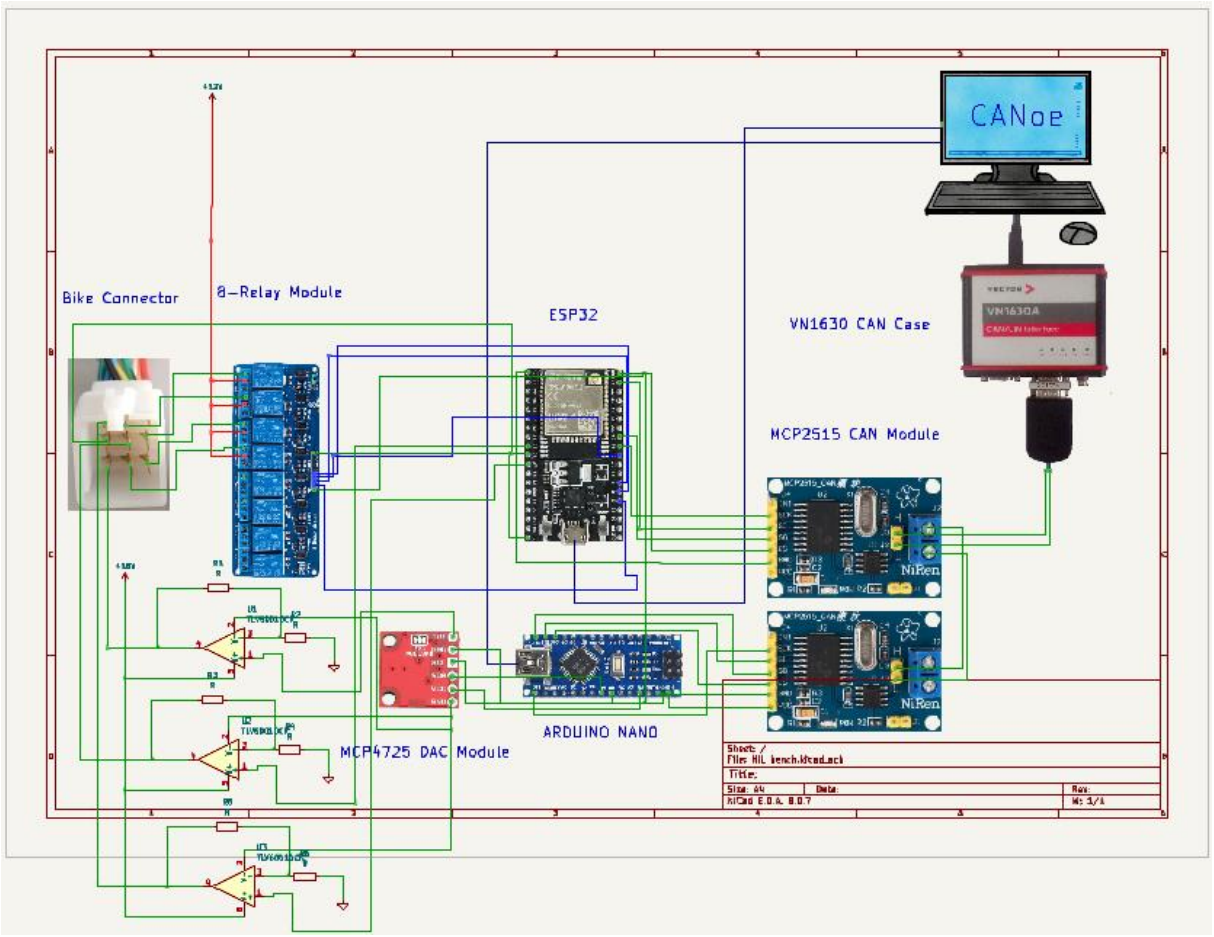

Fig.3.3. The complete schematic of the HIL test bench using KiCad

The CAN Communication Setup is doing via the VN1630 CAN Case injects CAN frames coming from the CANoe software through two MCP2515 CAN transceivers:

- One connected to ESP32, responsible for:
    - Fuel level (analog via DAC).
    - Outdoor temperature (analog via DAC).
    - Bike speed (digital pulses).
    - Indicators and headlights (digital high/low).
- One connected to Arduino Nano, solely for the power supply level signal.

3.4.1 Signal Interface with Dashboard

The following signals are connected via a 9-pin male connector as shown in Tab.3.2:

Pin	Signal	Source Pin	Type	Amplification	Final Voltage
1	Ground	-	Reference	No	0V
2	Fuel Level	ESP32 DAC 25	Analog	Yes	0–12V
3	Outdoor Temp	ESP32 DAC 26	Analog	No	0–3.3V
4	Bike Speed	ESP32 Pin 14	Digital PWM	Yes	~10.81V
5	Power Supply Level	DAC (Nano)	Analog	Yes	0–15V
6	Right Indicator	ESP32 Pin 33	Digital	Yes (Relay)	0/12V
7	Left Indicator	ESP32 Pin 32	Digital	Yes (Relay)	0/12V
8	Headlights	ESP32 Pin 27	Digital	Yes (Relay)	0/12V
9	Key Contact	ESP32 Pin 15	Digital	Yes (Relay)	0/12V

Tab.3.2. Pinout of the bike connector with the simulated signals

3.5 Code walkthrough

This walkthrough begins with the ESP32 firmware, which manages CAN communication, digital speed pulse generation, indicator blinking logic, temperature and fuel level output via its internal DACs, and the control of relays for on/off signals like headlights and key contact. Then, the Arduino Nano code is explained focusing briefly on its role in receiving the power supply level via CAN and converting it to a stable analog voltage using an external DAC over I2C.

Each part of the code will be explained briefly, clarifying how the real-world signals of a motorcycle are mimicked in our HIL bench.

3.5.1 ESP32 firmware

The code blocks for the ESP32 firmware are each explained as follows:

a) Included Libraires

```
#include <SPI.h>
#include <mcp_can.h>
#include <Wire.h>
```

- `SPI.h`: For communication with the MCP2515 CAN module over SPI.
- `mcp_can.h`: Library for interfacing with the MCP2515 CAN controller.
- `Wire.h`: For I2C communication (not directly used in this ESP32 code, may be used in future extensions).

b) Global Constants and Variables

```
const int pulsePin = 14;
const float Circumference = 1.29;
float speed_kmh = 0;
float pulseWidth = 0;
float pulseInterval = 0;
float highTime = 0;
```

- **pulsePin (14)**: Digital output simulating the Hall Effect speed signal.
- **Circumference (1.29 m)**: Tire circumference to calculate rotation/speed.
- Other variables are used to determine the frequency of pulses representing speed.

c) Indicator States

```
bool rightIndicatorOn = false;
bool leftIndicatorOn  = false;

bool rightBlinkState = false;
bool leftBlinkState  = false;

unsigned long lastBlinkTime = 0;
const unsigned long blinkInterval = 1000;
```

- these control blinking behavior for left and right indicators.

d) Pulse Generator State Variables

```
unsigned long lastPulseTime = 0;
bool pulseState = false;
```

- Used in the `simulatePulse()` function to generate a square wave on `pulsePin`.

e) CAN Bus Setup

```
#define CAN_CS_PIN 5
MCP_CAN CAN(CAN_CS_PIN);
```

- Defines **CS pin (GPIO 5)** to connect with the MCP2515 CAN module**.**

f) setup() Function

```
void setup() {
  Serial.begin(115200);
  pinMode(pulsePin, OUTPUT);
  pinMode(27, OUTPUT);  // Headlights
  pinMode(15, OUTPUT);  // Key Contact
  pinMode(33, OUTPUT);  // Right Indicator
  pinMode(32, OUTPUT);  // Left Indicator
```

- Initializes serial communication and configures the GPIOs used for digital outputs.

```
  if (CAN.begin(MCP_ANY, CAN_500KBPS, MCP_8MHZ) == CAN_OK) {
    Serial.println("CAN Bus initialized successfully!");
  } else {
    Serial.println("CAN Bus initialization failed!");
    while (1);
  }
  CAN.setMode(MCP_NORMAL);
}
```

- Initializes the MCP2515 CAN controller to listen to any ID at 500 Kbps using an 8 MHz oscillator.
- Switches the MCP2515 to normal mode to receive actual traffic.

g) loop() Function

```
if (CAN_MSGAVAIL == CAN.checkReceive()) {
  long unsigned int canId;
  unsigned char len = 0;
  unsigned char buf[8];

  CAN.readMsgBuf(&canId, &len, buf);
```

- Waits for incoming CAN messages.
- Reads the message buffer into `buf`.

h) ID: 0x103 – Fuel Level

```
if (canId == 0x103) {
  float fuelLevelRaw = buf[0];
  float liters = fuelLevelRaw * 0.01;
  Serial.print("Fuel level: ");
  Serial.print(liters);
  Serial.println(" L");

  float fuelRatio = liters / 2.5;
  uint8_t fuelDACValue = constrain(fuelRatio * 255.0, 0, 255);
  dacWrite(25, fuelDACValue);
}
```

- Converts fuel level from raw value to liters.
- Normalizes this to a 0–255 DAC value.
- Outputs analog voltage on **GPIO 25**, which goes to the fuel signal amplifier.

i) ID: 0x104 – Vehicle Speed

```
if (canId == 0x104) {
  speed_kmh = buf[0];
  Serial.print("Received speed: ");
  Serial.print(speed_kmh);
  Serial.println(" km/h");

  float speed_Mps = speed_kmh / 3.6;

  if (speed_Mps > 0) {
    float pulsePerSecond = speed_Mps / Circumference;
    pulseInterval = 1000000.0 / pulsePerSecond;
    lowTime = 22000.0 / speed_Mps;
    Serial.print("Target frequency: ");
    Serial.print(pulsePerSecond);
    Serial.println(" Hz");
  } else {
    pulseInterval = 0;
  }
}
```

- Extracts speed and converts to meters per second.
- Calculates how often to send pulses (`pulseInterval`) and how long to keep the signal high (`highTime`).
- These values are used in `simulatePulse()` to generate a square wave simulating a Hall sensor.

j) ID: 0x101 – Temperature

```
if (canId == 0x101) {
  int16_t rawTemp = (int16_t)buf[0];
  if (rawTemp >= 100) rawTemp -= 128;
  float temp_c = rawTemp;
  Serial.print("Temperature: ");
  Serial.print(temp_c);
  Serial.println(" °C");

  float tempRatio = constrain((temp_c + 10.0) / 65.0, 0.0, 1.0);
  uint8_t tempValue = tempRatio * 255;
  dacWrite(26, tempValue);
}
```

- Decodes temperature from signed 8-bit format.
- Normalizes and sends it to **GPIO 26** to simulate temperature sensor output (no amplification needed).

k) ID: 0x111 – Digital ON/OFF signals

```
if (canId == 0x111) {
  uint8_t statusByte = buf[0];

  rightIndicatorOn = statusByte & 0x01;
  leftIndicatorOn  = statusByte & 0x02;
  bool headLights  = statusByte & 0x04;
  bool keyContact  = statusByte & 0x08;

  digitalWrite(27, headLights);
  digitalWrite(15, keyContact);

  Serial.print("Right Indicator: ");
  Serial.println(rightIndicatorOn ? "ON" : "OFF");

  Serial.print("Left Indicator: ");
  Serial.println(leftIndicatorOn ? "ON" : "OFF");

  Serial.print("Head Lights: ");
  Serial.println(headLights ? "ON" : "OFF");

  Serial.print("Key Contact: ");
  Serial.println(keyContact ? "ON" : "OFF");
}
```

- Reads individual bits to detect corresponding signals in the Digital ON/OFF message for right/left indicators, headlights, and key contact.
- Outputs high/low values accordingly on GPIOs (which are later amplified to 12V via relays).

l) Blink Logic

```
unsigned long currentMillis = millis();
if (currentMillis - lastBlinkTime >= blinkInterval) {
  lastBlinkTime = currentMillis;

  if (rightIndicatorOn) {
    rightBlinkState = !rightBlinkState;
    digitalWrite(33, rightBlinkState);
  } else {
    digitalWrite(33, LOW);
    rightBlinkState = false;
  }

  if (leftIndicatorOn) {
    leftBlinkState = !leftBlinkState;
    digitalWrite(32, leftBlinkState);
  } else {
    digitalWrite(32, LOW);
    leftBlinkState = false;
  }
}
```

- Flashes indicator outputs on GPIO 33 (right) and 32 (left) every 1 second when activated to simulate blinking effect.

m) simulatePulse()

```
void simulatePulse() {
  if (pulseInterval <= 0) {
    digitalWrite(pulsePin, LOW);
    return;
  }

  unsigned long now = micros();

  if (!pulseState && now - lastPulseTime >= pulseInterval) {
    pulseState = true;
    lastPulseTime = now;
    digitalWrite(pulsePin, HIGH);
  }

  if (pulseState && now - lastPulseTime >= highTime) {
    pulseState = false;
    digitalWrite(pulsePin, LOW);
  }
}
```

- Generates a square wave on GPIO 14 simulating a Hall effect speed sensor.
- Alternates between HIGH and LOW based on pulse timing and `lowTime`.

3.5.2 ARDUINO Nano firmware

The code blocks for the ARDUINO Nano firmware are each explained as follows:

a) Library Inclusions

```
#include <SPI.h>
#include <Wire.h>
#include <Adafruit_MCP4725.h>
#include <mcp_can.h>
```

- `Wire.h`: Handles I2C communication between the Nano and the MCP4725 DAC.
- `Adafruit_MCP4725.h`: A driver library provided by Adafruit to simplify controlling the MCP4725 DAC.
- `mcp_can.h` and `SPI.h`: Required for interfacing with the MCP2515 CAN controller over SPI.

b) Object Instantiations and Pin Definitions

```
Adafruit_MCP4725 dac;
#define CAN_CS_PIN 10
MCP_CAN CAN(CAN_CS_PIN);
```

- `dac` is the object that communicates with the DAC module.
- `SPI_CS_PIN`: The Chip Select pin for the MCP2515 CAN module. It's usually connected to D10 on Arduino Nano.
- `CAN_INT_PIN`: Can be used if you want to use interrupts (optional, not used in this code).
- `MCP_CAN`: Object that handles CAN communication using MCP2515.

c) Setup Function

```
void setup() {
  Wire.begin();
  dac.begin(0x60);
  CAN.begin(MCP_ANY, CAN_500KBPS, MCP_8MHZ);
  CAN.setMode(MCP_NORMAL);
}
```

- Initializes the serial monitor for debugging.
- Initializes the I2C communication (used for the DAC).
- Starts the DAC at address 0x60, which is the default for most MCP4725 modules.

```
  if (CAN.begin(MCP_ANY, CAN_500KBPS, MCP_8MHZ) == CAN_OK) {
    Serial.println("CAN initialized successfully.");
  } else {
    Serial.println("CAN init failed.");
    while (1);
  }
  CAN.setMode(MCP_NORMAL);   // Set to normal mode
}
```

- Initializes the CAN module at 500 kbps speed and 8 MHz oscillator.
- If initialization fails, the Nano enters an infinite loop.
- Sets the MCP2515 to Normal Mode, meaning it will both send and receive messages.

d) Main Loop (Listening for CAN Messages)

```
void loop() {
  if (CAN_MSGAVAIL == CAN.checkReceive()) {
    long unsigned int canId;
    unsigned char len = 0;
    unsigned char buf[8];

    CAN.readMsgBuf(&canId, &len, buf);
```

- Constantly checks if a new CAN message is available.
- If so, it reads the message into canId (the identifier) and buf (the data buffer).

e) Handling the Specific Power supply level Message ID (0x105)

```
if (canId == 0x105) {  // ID for power supply level
  uint8_t voltageRaw = buf[0];  // Example: value from 0 to 150
  float voltage15v = voltageRaw * 0.1;  // Simulate 0.0V to 15.0V range
```

- Only processes messages with a specific ID (e.g., 0x105 = Power Supply Level).
- Extracts the first byte buf[0] and interprets it as a raw value (e.g., 0–150).
- Multiplies by 0.1 to convert it into a real voltage value (0.0V to 15.0V).

f) Mapping to 0–5V for the DAC

```
// Map 0-15V to 0-5V output for DAC
float voltage5v = voltage15v * (5.0 / 15.0);
uint16_t dacValue = (uint16_t)((voltage5v / 5.0) * 4095);  // 12-bit DAC
```

- Since the DAC can only generate a 0–5V analog output, we scale the 15V value down proportionally.
- We calculate the 12-bit DAC value corresponding to that scaled voltage (0–4095).

g) Output Voltage via DAC

```
    dac.setVoltage(dacValue, false);

    Serial.print("Received voltage: ");
    Serial.print(voltage15v);
    Serial.print("V, DAC Output: ");
    Serial.println(voltage5v);
  }
 }
}
```

- The calculated DAC value is sent to the MCP4725, generating a precise analog voltage.
- A debug message is printed showing both the interpreted voltage and its corresponding DAC output.

3.6 CAN Frame generation

The CAN frames generation were made possible by first creating the DBC file then attaching it to the CANoe interface followed by adding the environment variables then lastly writing the CAPL script and wiring it to each environment variable that respectfully corresponds to each signal concerned by the simulation.

3.6.1. Creating the CAN Database (DBC File)

The DBC (Database CAN) file describes the structure of CAN messages. It includes message IDs, signal names, data lengths, and bit positions.

The Steps to creating a DBC file are:

1. Open Vector Database Editor (`CANdb++ Editor`) from CANoe.
    - Create a new DBC file and define Messages:
        - `0x103`: Fuel Tank Level.
        - `0x104`: Speed.
        - `0x105`: Power Supply Level.
        - `0x101`: Outdoor Temperature.
        - `0x111`: Status Flags (this one has 4 different signals, indicator right, indicator left, headlights, supply_key_contact).
2. For each message, we define the signals:

    - Name (`FuelLevel`).
    - Type: Unsigned / Signed / Float.
    - Start Bit, Length, Byte Order.
    - Factor and offset (for scaling raw values).
    - Minimum and Maximum.

3. We save the DBC file and import it into our CANoe simulation setup. As you can see the different signals are all mentioned below in Fig.3.4.

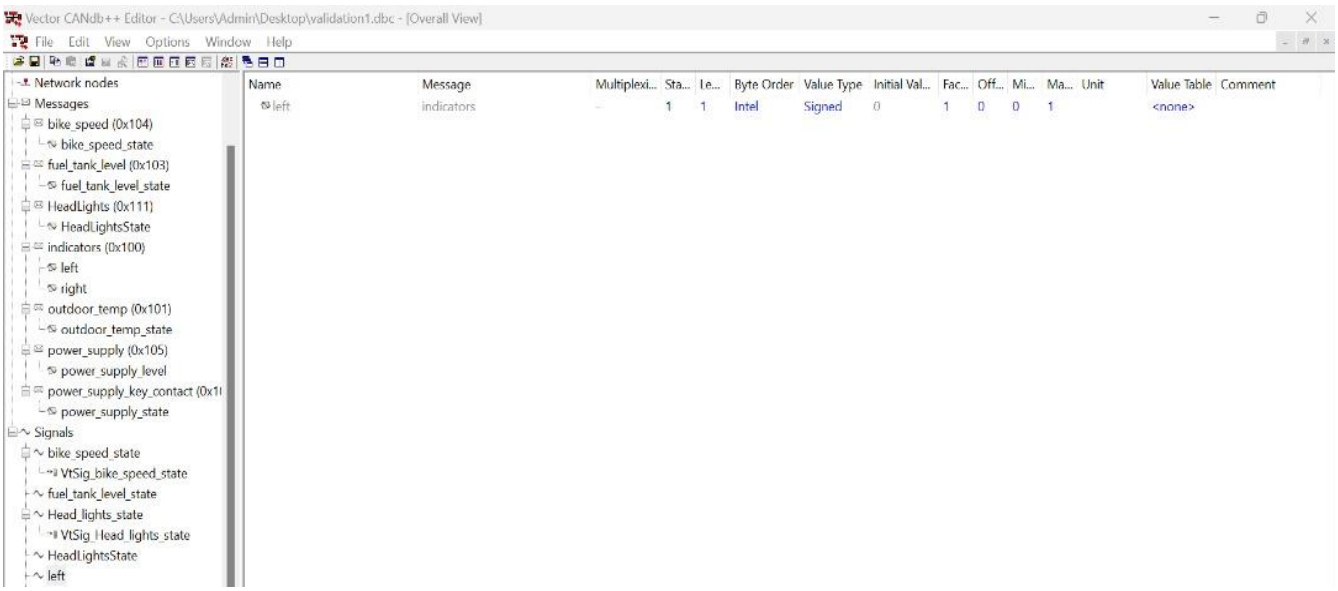

Fig.3.4. Snippet of the different signals and messages present in our DBC file

3.6.2. Attaching the DBC File in CANoe

1. Open CANoe.
2. Go to Simulation Setup → CAN Configuration.
3. Attach the DBC file to the appropriate CAN channel (`CAN1`).

- This allows CAPL scripts and panels to reference signal names directly.

3.6.3. Adding environment Variables in CANoe

Environment variables in CANoe act as global placeholders that allow different parts of the simulation such as CAPL scripts, panels or measurement setup, to communicate and synchronize data. They are essential for simulating sensor values and controlling message behavior during a test scenario. In the context of a Hardware-in-the-Loop (HIL) bench, environment variables are used to model physical inputs such as speed, fuel level or temperature. Each signal in a CAN frame is typically linked to a corresponding environment variable in the CAPL script as it's explained more in the next section, allowing dynamic control of message content (signals) based on user input or test logic. This linking ensures that the simulated signals behave like real-world values providing a realistic and flexible test environment for ECUs like dashboards.

To add new variables, you can do it through a CAPL script or simply navigate through the CANoe gui to system variables in the Environment tab, then right click on env and click add a new variable after that, you can simply enter your variable details which are usually the same ones you entered for your corresponding signals in the DBC file (it's better to give them the same names as your signals to avoid confusion), as shown below in Fig.3.5:

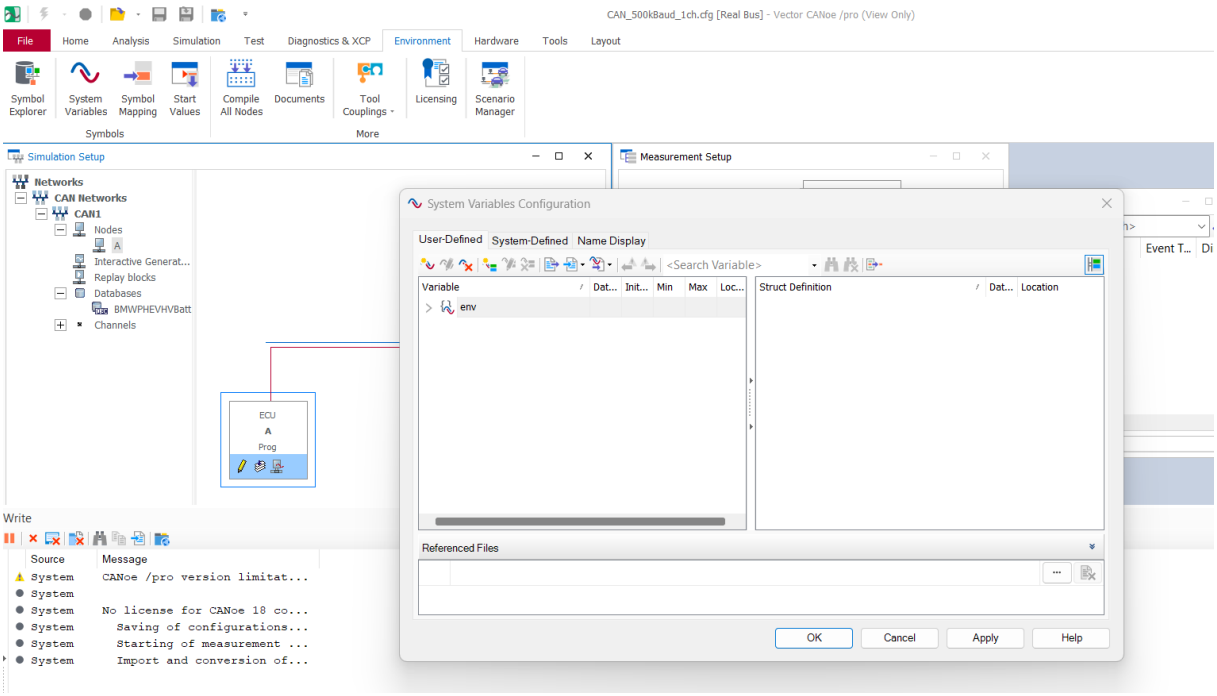

Fig.3.5. Snippet of the Environment Variables window inside CANoe

3.6.4. Writing CAPL Scripts to Simulate Sensor Signals

CAPL (CAN Access Programming Language) is used to generate dynamic CAN messages during simulation.

Our CAPL script simulates a motorcycle dashboard using periodic CAN messages. It fetches environment variable values which are created then link them to each corresponding signal in our DBC representing various sensor states (like fuel level, speed, indicators, etc.) at regular intervals. And this is a detailed explanation of the script:

a) General Structure

```
variables
{
  msTimer TIMER_fuel_tank_level;
  message CAN::fuel_tank_level AA_fuel_tank_level;
  msTimer TIMER_bike_speed;
  message CAN::bike_speed AA_bike_speed;
  msTimer TIMER_power_supply;
  message CAN::power_supply AA_power_supply;
   msTimer TIMER_HeadLights;
  message CAN::HeadLights AA_HeadLights;

  msTimer TIMER_indicators;
  message CAN::indicators AA_indicators;
  msTimer TIMER_power_supply_key_contact;
  message CAN::power_supply_key_contact AA_power_supply_key_contact;
  msTimer TIMER_indicators_display;
  message CAN::outdoor_temp AA_outdoor_temp;
  msTimer TIMER_outdoor_temp;

}
```

All timers and CAN messages are declared here:

- `msTimer`: A 1 millisecond timer used to trigger events at regular intervals.
- `message CAN::X`: Refers to a CAN message defined in the DBC file under the node `CAN`.

b) On start Block

```
on start
{
  setTimer (TIMER_fuel_tank_level,100);
  setTimer (TIMER_bike_speed,100);
  setTimer ( TIMER_power_supply,100);

  setTimer ( TIMER_indicators,1000);
  setTimer ( TIMER_power_supply_key_contact,1000);
  setTimer (TIMER_indicators_display,1000);
  setTimer (TIMER_outdoor_temp,1000);
  setTimer (TIMER_HeadLights,100);
}
```

Each timer triggers a corresponding on timer block periodically (every 100ms or 1000ms).

c) Individual Timer Logic

- **on timer TIMER_fuel_tank_level**

```
on timer TIMER_fuel_tank_level
{
  if(@sysvar::env::fuel_tank_level <= 0.5) @sysvar::env::fuel_tank_level_low=1;
  if(@sysvar::env::fuel_tank_level > 0.5) @sysvar::env::fuel_tank_level_low=0;
  AA_fuel_tank_level.fuel_tank_level_state = ((@sysvar::env::fuel_tank_level-0)/0.01);
  output(AA_fuel_tank_level);
  setTimer (TIMER_fuel_tank_level,100);
 }
```

- Reads the fuel level from the environment variable.
- Sets a second env variable `fuel_tank_level_low` to indicate low fuel (<50%).
- Converts the float fuel level into an integer for the CAN signal using the scaling from the DBC:
  `Signal = (Value - Offset) / Factor.`
- Sends the `fuel_tank_level` CAN message.

- **on timer TIMER_bike_speed**

```
 on timer TIMER_bike_speed
{
  AA_bike_speed.bike_speed_state = ((@sysvar::env::bike_speed-0)/1);
 output(AA_bike_speed);
  setTimer (TIMER_bike_speed,100);
 }
```

- Reads bike speed from env variable.
- Sends it as a CAN message.
- No scaling is applied (factor = 1).

- **on timer TIMER_outdoor_temp, TIMER_power_supply and TIMER_HeadLights**

```
on timer TIMER_outdoor_temp
{
  AA_outdoor_temp.outdoor_temp_state = ((@sysvar::env::outdoor_temp-0)/1);
 output(AA_outdoor_temp);
  setTimer (TIMER_outdoor_temp,100);
 }
on timer TIMER_power_supply
{
  AA_power_supply.power_supply_level = ((@sysvar::env::power_supply_level-0)/0.01);
 output(AA_power_supply);
  setTimer (TIMER_power_supply,100);
 }
on timer TIMER_HeadLights
{
  AA_HeadLights.HeadLightsState = ((@sysvar::env::HeadLights-0)/1);
 output(AA_HeadLights);
  setTimer (TIMER_HeadLights,100);
 }
```

- **Temperature**:

  Sends the temperature value directly from environment variable.

- **Power Supply:**

  1. Converts a float power supply voltage to an integer (e.g., 12.34V → 1234 if factor = 0.01).
  2. Sends the power_supply message.

- **HeadLights:**

  Sends 1 or 0 to represent whether headlights are on or off.

- **on timer TIMER_indicators**

```
on timer  TIMER_indicators
{
  AA_indicators.right = ((@sysvar::env::indicator_right-0)/1);
  AA_indicators.left = ((@sysvar::env::indicator_left-0)/1);
 output(AA_indicators);
  setTimer (TIMER_indicators,1000);
 }
```

Sends a message with two boolean values representing left and right indicators.

- **on timer TIMER_indicators_display**

```
on timer  TIMER_indicators_display
{
  if(@sysvar::env::indicator_right==1) @sysvar::env::indicator_right_on_off = !@sysvar::env::indicator_right_on_off;
  if(@sysvar::env::indicator_right==0) @sysvar::env::indicator_right_on_off = 0;
  if(@sysvar::env::indicator_left==1) @sysvar::env::indicator_left_on_off = !@sysvar::env::indicator_left_on_off;
  if(@sysvar::env::indicator_left==0) @sysvar::env::indicator_left_on_off = 0;
  setTimer (TIMER_indicators_display,1000);
 }
```

  - When the indicator is activated, it toggles a second variable on/off every 1 second.
  - This second variable can be used in a panel to show a blinking light.

- **Looping:**

  All timers reset themselves at the end of their block using:

```
setTimer(TIMER_X, interval);
```

d) Summary

This CAPL script is responsible for simulating the CAN messages for out test bench which is meant to test the motorcycle dashboard in a Hardware-in-the-Loop (HIL). It periodically reads environment variables such as fuel level, speed, power supply, temperature, and indicator states and encodes them into CAN messages defined in the database (DBC file). Each timer triggers at a fixed interval to send the corresponding message onto the CAN bus. Additionally, some logic is included to simulate conditions like low fuel and blinking indicators, ensuring realistic dashboard responses during testing.

3.6.5 Creating the Simulation Panel

In CANoe, the simulation panel provides a visual way to control and observe system behavior during testing. Each widget (slider, switch, LED, etc.) (which are all installed by default in the panel editor) represents an environment variable linked to its correspondent signal. This link ensures that changes made through the panel directly affect the CAPL script logic allowing real-time generation of CAN messages based on user input. It makes testing easier, interactive and more intuitive.

Creating the panel is done through the panel editor inside CANoe as shown below in Fig.3.6.

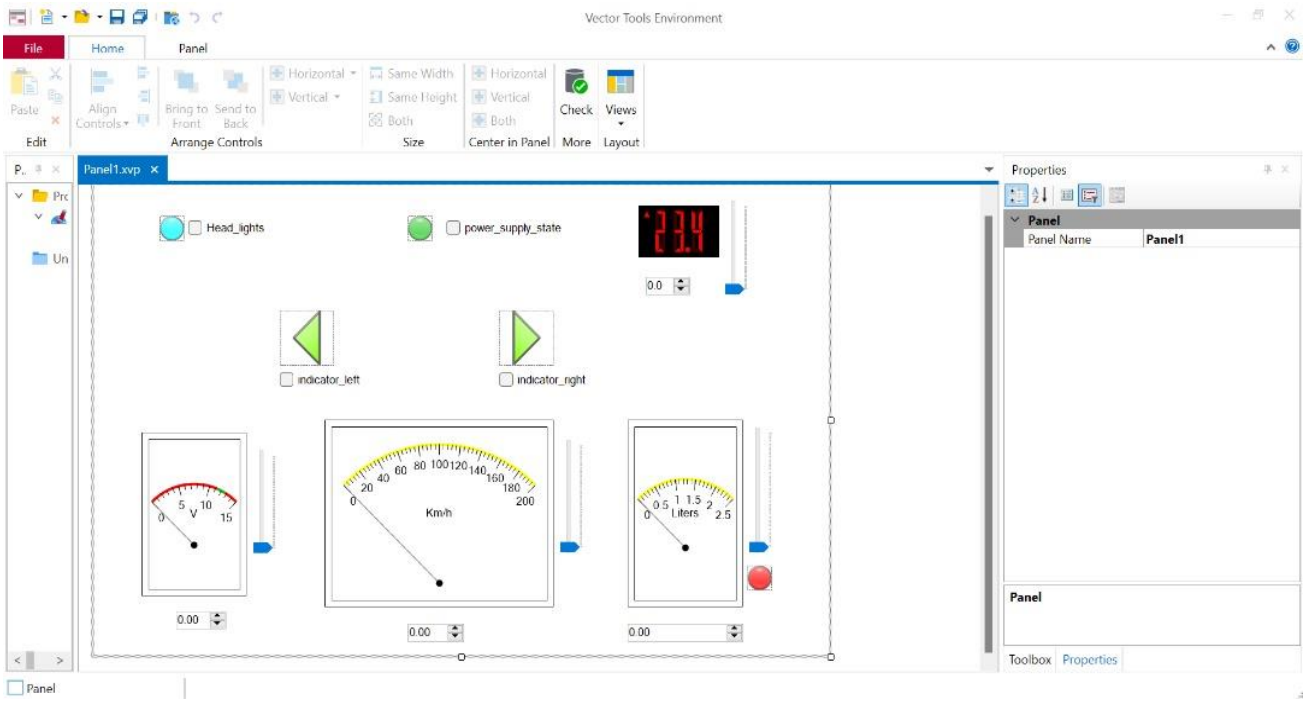

Fig.3.6. Snippet of the Panel Editor inside CANoe.

3.7 Test Cases and Verification

In order to validate the functionality of our HIL (Hardware-in-the-Loop) test bench we defined a series of test cases corresponding to real-world motorcycle scenarios. Each test case involves simulating a specific signal via CANoe using environment variables controlled by the panel and observing the physical response on the actual dashboard. For example, to test the speedometer, we varied the bike_speed environment variable and verified the digital pulse signal on pin 14 of the ESP32 while also checking that the dashboard needle responded correctly. Similarly, adjusting the fuel_tank_level variable produced an analog voltage on DAC pin 25, amplified to 12V, which the dashboard interpreted as varying fuel levels. Indicator lights and key contact were tested using binary CAN messages, with visual confirmation of correct blinking or activation. The signals were measured and verified using a multimeter and oscilloscope where needed to ensure correctness in voltage levels and frequencies. Through this rigorous process, we confirmed that the simulated signals successfully mimic real sensor outputs, verifying the reliability of the HIL system.

3.8 Results and Observations

After the successful integration and testing of the Hardware-in-the-Loop bench, several key results and observations were recorded. First, all eight output signals were accurately generated and interpreted by the dashboard, confirming the effectiveness of our simulation approach. Also the analog voltages for the fuel tank level and power supply level were successfully adjusted using both the ESP32's internal DAC and the MCP4725 module controlled by the Arduino Nano. However, we observed signal fluctuation when both DACs were operated on the ESP32 via I2C, likely due to I2C bus latency or bandwidth limitations. This led us to offload the power

supply level generation to another dedicated microcontroller, which resolved the issue completely. Additionally, the LM324 amplifier proved effective in boosting DAC signals to the required 12V or 10.8V levels, although special care was taken to ensure the input reference voltage exceeded the expected output to avoid clipping, since the LM324 is not rail-to-rail. Digital signals such as speed pulses and binary states (indicators, headlights, key contact) showed accurate behavior and frequency modulation effectively mimicking real-world sensor data. Additionally the use of CANoe allowed for precise control over signal injection through a user-friendly simulation panel. Overall, the HIL bench functioned as intended by demonstrating stable and reproducible performance across all test cases, and confirming the feasibility of using such a system for dashboard validation and production line testing in the future.

Below in Fig.3.7 is a snippet that shows the final Simulation Panel on the left besides the trace window which allows us to trace in real-time the CAN messages sent to our VN1630 CAN case.

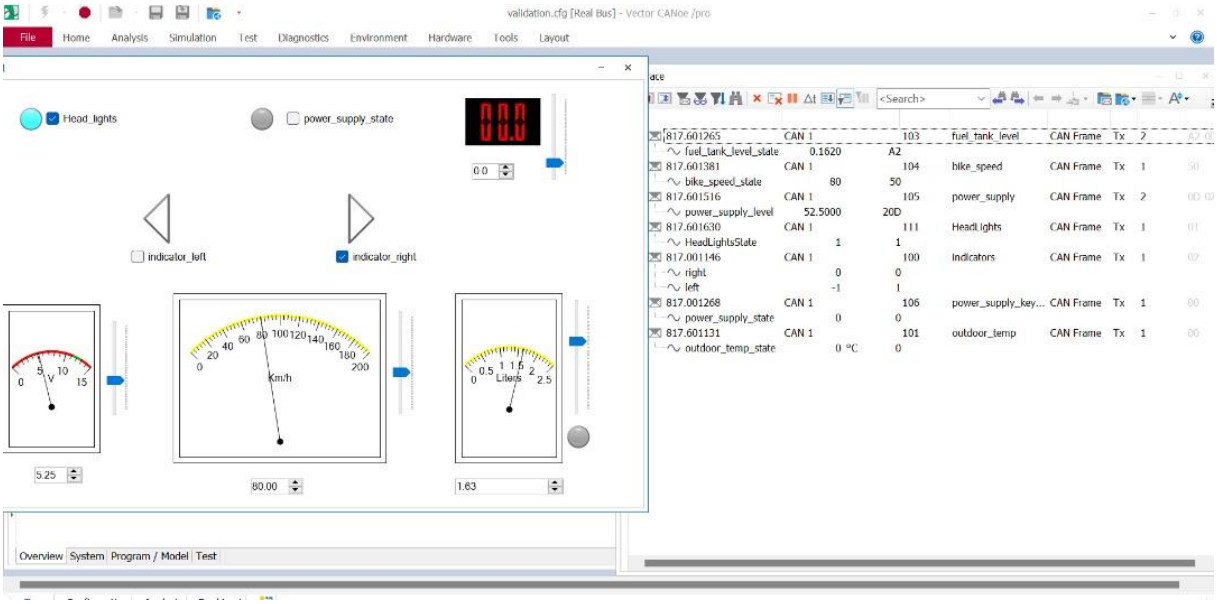

Fig.3.7. The final Simulation Panel alongside the trace window

3.9 Challenges Encountered

Throughout the development of the HIL bench several challenges emerged that required thoughtful troubleshooting. One of the main issues was the signal interference caused by I2C bus congestion when attempting to control multiple DACs from the ESP32. This led to fluctuations in output voltages, especially when trying to generate three analog signals simultaneously and to resolve this, we put the power supply simulation to an Arduino Nano ensuring a smooth, isolated DAC output for that signal.

Another challenge was amplifying DAC outputs to match the real dashboard's required voltage levels. The LM324 op-amp, while cost-effective and easy to implement. it is not rail-to-rail, which caused clipping issues when the supply voltage was too close to the output target. This

was mitigated by using a higher 16V power supply and carefully choosing gain resistor values to obtain a clean 12V or 10.8V output.

Additionally, precise CAN message timing and synchronization between CANoe and the embedded systems required careful tuning of timers and message structures. Debugging CAN traffic was also occasionally hindered by frame collisions and misread signal values which were later solved by refining the CAPL script and properly linking environment variables.

3.10 Enclosure Design and Assembly of the HIL Test Bench

To ensure both safety and a professional presentation, all the electronic components of the HIL test bench were finally installed inside a robust metallic enclosure enclosure as you can see in Fig.3.8 below, this approach we took provides organized cable management for the system. the process followed several best practices that we found across the web (all links are found in the refences chapter) for mounting electronics in metal alimunium project boxes:

Fig.3.8. The different components used for the enclosure of the HIL test bench

3.10.1. The choice a metallic enclosure

The choice of a metallic enclosure was picked mainly for:

- **Enclosure Selection:** A sturdy red aluminum enclosure was chosen for its durability and ability to dissipate heat efficiently, which is crucial for embedded systems
- **Ventilation:** The enclosure includes several ventilation holes to prevent overheating of the internal electronics, as it is recommended for safe operation of embedded projects.
- **Planning Cutouts:** Before the assembly, all required cutouts were carefully planned and marked on the enclosure surface for the different connectors, supply cables and also for the output signals going into the bike connector.

3.10.2. Drilling and Cable Management

The drilling techniques used in the making of the enclosure are:

- **Custom Drilling:** Using a precision drill, we made holes for:

    1. Output signal connectors.
    2. The VN1630 CAN interface DB9 connector.
    3. Power supply cables for both the ESP32 and Arduino Nano.
    4. Separate entries for the 16V and 12V power supplies.

- **Cable Protection:** All cable entry points were fitted with rubber grommets to prevent abrasion and reduce the risk of short circuits, following standard safety guidelines.

3.10.3. Mounting and Isolation

Isolation is necessary to avoid any electrical short circuits and it was realized using the following materials:

- **Standoffs and Insulation:** All the PCBs (VN1630, MCP2515, ESP32, Arduino Nano) were fixed using nylon standoffs as you can see in the lower half of Fig.3.8 above, elevating them above the metal surface. This prevents accidental contact and short circuits which is definitely a critical step when working with conductive enclosures.
- **Screw Fixing:** Each board was secured with screws through those standoffs which mainly ensures mechanical stability even during transport or exhibition use.

3.10.4. Power Supply Integration

Power supply ports were carefully secured as we had to make ports for two power supplies:

- **Dual Power Inputs:** The enclosure was well drilled to take both 16V and 12V power supply inputs each with a dedicated entry points and clear labelling to avoid confusion during the setup and operation.
- **Grounding:** The metal enclosure was properly grounded to enhance safety and reduce electromagnetic interference thus biding by industrial standards.

3.10.5. Final Assembly and Testing

Before closing everything, system checks and labelling had to be done to ensure good continuity and insulation as well a clear organized overall structure:

- **System Check:** Before sealing the enclosure, all connections were tested for continuity and insulation. The system was powered up to ensure there were no accidental shorts and that all signals were correctly routed through their respective connectors.
- **Labeling:** All external connectors and switches were clearly labeled for ease of use and troubleshooting during demonstrations or maintenance.

3.10.6. Summary

The use of a metallic enclosure and careful mechanical design, with adherence to best practices for the mounting and isolating electronics ensured both the safety and professional appearance of the HIL test bench, this successful demonstration at this major industry event further attests to the project's quality and relevance.

3.11 Conclusion

On this final chapter it is explained that The HIL test bench was successfully designed and implemented using ESP32 and Arduino Nano microcontrollers, supported by DACs, relays and CAN interfaces. The generated signals accurately emulate real-world conditions and the setup was enclosed in a professionally assembled metal casing. This comprehensive design enables precise and repeatable testing of motorcycle dashboards

GENERAL CONCLUSION

This project aimed to design, implement, and validate a Hardware-in-the-Loop (HIL) test bench dedicated to testing a motorcycle dashboard under conditions that closely simulate real-world operation. The primary goal was to emulate the analog and digital signals that the dashboard would normally receive on an actual vehicle, allowing for functional validation in a controlled and repeatable environment.

The adopted methodology began with an analysis of the dashboard's behaviour, identification of required signals and the selection of appropriate hardware components. A modular system architecture was then designed and developed to dynamically generate signals based on incoming CAN messages. Special care was given to signal accuracy, voltage amplification and also system protection through isolation and proper ground referencing.

Key challenges included the ESP32's limitation, instability with the I2C bus, and voltage supply issues needed to achieve realistic analog outputs. These were overcome by introducing a second microcontroller, using properly configured operational amplifiers, and powering them with a dedicated 16V supply to prevent signal clipping.

The results demonstrate that a reliable, low-cost HIL test bench can be achieved, offering accurate simulation of real signals and enabling effective validation of the dashboard. This work represents a crucial step toward implementing testing and validation practices that align with automotive industry standards, especially the ISO 26262 standard, which governs functional safety for road vehicles.

The added value of this project lies not only in its technical achievements but in its contribution to bringing automotive testing and validation culture into a local context. In this regard, it opens new possibilities for the future development of a national embedded systems testing and validation ecosystem which is a necessary foundation if Algeria ever intends to move toward manufacturing smart vehicles or electronic automotive components.

The integration of professional tools for test and validation was also necessary in this process to approach automotive-grade quality, traceability, and repeatability in testing processes.

Looking ahead, this project can evolve into a complete HIL system with closed-loop feedback, integration of Software-in-the-Loop (SIL) environments to simulate virtual ECUs, and the implementation of automated test case generation. It could also serve as a training platform for students and engineers, encouraging the growth of a national workforce skilled in embedded systems and functional validation.

In summary, this project not only allowed the application of advanced technical skills but also addressed a real-world engineering challenge. It stands as a concrete step toward fostering a culture of rigorous and high-quality testing, an essential requirement for any serious industrial ambition in the automotive field.

References

[1] Groupe 6NAPSE, “HIL Test and Simulation Bench,” 6NAPSE. [Online]. Available: https://6-napse.com/en/technical-means/hil-test-simulation-bench/ [Accessed: May 26, 2025].

[2] Allion Labs, “HIL Solution and ISO 26262 Compliance,” Allion Labs. [Online]. Available: https://www.allion.com/test-lab/hardware_in_the_loop/ [Accessed: May 27, 2025].

[3] LHP Engineering Solutions, “HIL Testing and Functional Safety,” LHP Engineering Solutions. [Online]. Available: https://www.lhpes.com/blog/what-is-hil-testing [Accessed: May 29, 2025].

[4] STEP Lab, “HIL Testing Systems,” STEP Lab. [Online]. Available: https://step-lab.com/hardware-in-the-loop-systems/ [Accessed: May 30, 2025].

[5] Vector Informatik, “HIL Test Systems,” Vector. [Online]. Available: https://www.vector.com/se/en/products/products-a-z/hardware/hil-test-systems/ [Accessed: Jun. 1, 2025].

[6] Link Engineering, “Dynamic Brake Simulation HIL System,” The Brake Report. [Online]. Available: https://thebrakereport.com/links-new-hil-dynamic-brake-simulation/ [Accessed: Jun. 2, 2025].

[7] P. Viennet et al., “Development of a Hardware-in-the-Loop Test Bench for E-bike ABS,” ARODES, 2024. [Online]. Available: https://arodes.hesso.ch/record/14539/files/Viennet_2024_development_hardware-in-the-loop_test_bench.pdf [Accessed: Jun. 4, 2025].

[8] Test & Measurement World, “SIL vs. HIL in Embedded Systems,” Test & Measurement World. [Online]. Available: https://www.test-and-measurement-world.com/articles/embedded-systems/sil-vs-hil-testing-comparison [Accessed: Jun. 5, 2025].

[9] LEADVENT Group, “Safety Lifecycle in ISO 26262,” LEADVENT Group Blog. [Online]. Available: https://www.leadventgrp.com/blog/safety-lifecycle-in-iso-26262 [Accessed: Jun. 6, 2025].

[10] International Organization for Standardization, ISO 26262-1:2018 – Road vehicles — Functional safety — Part 1: Vocabulary, ISO Standard. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en [Accessed: Jun. 7, 2025].

[11] Y. Dubrov, “Key Concepts in ISO 26262,” LinkedIn. [Online]. Available: https://www.linkedin.com/pulse/key-concepts-iso-26262-yakov-dubrov-8n10e/ [Accessed: Jun. 8, 2025].

[12] National Instruments, “What is the ISO 26262 Functional Safety Standard,” NI White Paper. [Online]. Available: https://www.ni.com/en/solutions/transportation/what-is-the-iso-26262-functional-safety-standard-.html [Accessed: Jun. 10, 2025].

[13] Ansys, “What is ISO 26262,” Ansys. [Online]. Available: https://www.ansys.com/simulation-topics/what-is-iso-26262 [Accessed: Jun. 11, 2025].

[14] BYHON, “A Brief Introduction to ISO 26262,” BYHON. [Online]. Available: https://www.byhon.it/a-brief-introduction-to-iso-26262/ [Accessed: Jun. 13, 2025].

[15] EMBITEL, “Functional Safety for Automotive ECU Development,” EMBITEL. [Online]. Available: https://www.embitel.com/product-engineering-2/iso-26262-functional-safety [Accessed: Jun. 15, 2025].

[16] M. Kruszynska, “ISO 26262: The Complete Guide,” Spyrosoft Blog. [Online]. Available: https://spyro-soft.com/blog/automotive/iso-26262 [Accessed: Jun. 17, 2025].