**Dự Án: Xây Dựng Hệ Thống Cụm Đồng Hồ Kỹ Thuật Số và Bộ Điều Khiển Trung Tâm (VCU)**

**MỤC LỤC**

[1. Tổng quan dự án 1](#_Toc206290688)

[1.1. Giới thiệu 1](#_Toc206290689)

[1.2. Mục tiêu Dự án (Project Objectives) 1](#_Toc206290690)

[1.2.1. Mục tiêu Kỹ thuật 1](#_Toc206290691)

[1.2.2. Mục tiêu chất lượng 1](#_Toc206290692)

[1.3. Phạm vi Dự án (Project Scope) 2](#_Toc206290693)

[1.3.1. Trong phạm vi (In-Scope) 2](#_Toc206290694)

[1.3.2. Ngoài phạm vi (Out-of-Scope) 2](#_Toc206290695)

[2. Tổng quan kiến trúc hệ thống 4](#_Toc206290696)

[2.1. Kiến trúc Zonal (Zonal Architecture) 4](#_Toc206290697)

[2.2. Sơ đồ Khối Hệ thống (System Block Diagram) 4](#_Toc206290698)

[2.3. Luồng Dữ liệu và Tương tác 6](#_Toc206290699)

[3. Thiết kế phần cứng và kết nối 7](#_Toc206290700)

[3.1. Danh sách vật tư (Bill of Materials - BOM) 7](#_Toc206290701)

[3.2. Sơ đồ kết nối 7](#_Toc206290702)

[3.2.1. Sơ đồ kết nối phần ECU nodes 7](#_Toc206290703)

[3.2.2. Kết nối phần VCU 9](#_Toc206290704)

[4. Quy trình phát triển phần mềm 11](#_Toc206290705)

[4.1. Giao tiếp CAN giữa VCU và ECU nodes 11](#_Toc206290706)

[4.1.1. Truyền tin giữa các thành phần 11](#_Toc206290707)

[4.1.2. Kiến trúc linh hoạt dựa trên cấu hình I/O động 11](#_Toc206290708)

[4.1.3. Định nghĩa bảng tin CAN 12](#_Toc206290709)

[4.1.3.1. Định dạng CAN ID 13](#_Toc206290710)

[4.1.3.2. Khung dữ liệu Digital Output 15](#_Toc206290711)

[4.1.3.3. Khung dữ liệu Digital Input 16](#_Toc206290712)

[4.1.3.4. Khung dữ liệu Digital Analog 17](#_Toc206290713)

[4.2. Thiết kế phần mềm VCU 18](#_Toc206290714)

[4.2.1. Sử dụng USB to CAN 18](#_Toc206290715)

[4.2.1.1. Cấu hình USB to CAN 18](#_Toc206290716)

[4.2.1.2. Lập trình SocketCAN trong Linux 18](#_Toc206290717)

[4.2.2. Kết nối Qt/C++ với QML 22](#_Toc206290718)

[4.2.3. Màn hình và giao diện IPC 22](#_Toc206290719)

[4.2.4. Cấu trúc file project Qt 24](#_Toc206290720)

[4.2.5. Cách binding dữ liệu 24](#_Toc206290721)

[4.2.6. Software VCU 25](#_Toc206290722)

[4.3. Firmware cho STM32 (mô phỏng lại các ECU nodes) 27](#_Toc206290723)

[4.3.1. Cấu hình và giao tiếp CAN cơ bản 27](#_Toc206290724)

[4.3.2. Firmware cho ECU nodes (mô phỏng bằng STM32) 31](#_Toc206290725)

[5. Thuật toán xử lý 33](#_Toc206290726)

[5.1.1. Tính toán phần trăm pin 33](#_Toc206290727)

[5.1.2. Tính toán quãng đường còn lại 33](#_Toc206290728)

[6. Kế hoạch tích hợp và kiểm thử 35](#_Toc206290729)

[6.1. Kiểm thử đơn vị (Unit Test) 35](#_Toc206290730)

[6.1.1. Test case cho JSON Config Parser (trên VCU) 35](#_Toc206290731)

[6.1.2. Test case cho CAN Frame Packing/Unpacking (trên ECU và VCU) 36](#_Toc206290732)

[6.2. Kiểm Thử Hệ Thống (System Test) 37](#_Toc206290733)

[6.2.1. Latency End-to-End 37](#_Toc206290734)

[7. Kết luận và hướng phát triển 40](#_Toc206290735)

[7.1. Kết luận 40](#_Toc206290736)

[7.2. Hướng phát triển 40](#_Toc206290737)

[Tài Liệu Tham Khảo 42](#_Toc206290738)

[Phụ lục: Bảng thuật ngữ 44](#_Toc206290739)

**MỤC LỤC HÌNH**

[Hình 1. Kiến trúc Domain (Domain Architecture) và kiến trúc Zonal (Zonal Architecture) 4](#_Toc206290740)

[Hình 2. Sơ đồ khối hệ thống 5](#_Toc206290741)

[Hình 3. Kết nối liên quan STM32F1 9](#_Toc206290742)

[Hình 4. Kết nối toàn bộ phần cứng trong thực tế 10](#_Toc206290743)

[Hình 5. Giao diện màn hình IPC 23](#_Toc206290744)

[Hình 6. Sơ đồ khối mô tả sơ lược logic khối VCU 26](#_Toc206290745)

[Hình 7. Cấu hình CAN controller cho STM32F1 28](#_Toc206290746)

[Hình 8. Cấu hình GPIO của STM32F1 sử dụng CAN 28](#_Toc206290747)

[Hình 9. CAN Bus 29](#_Toc206290748)

[Hình 10. Cấu hình ngắt để nhận khung CAN 31](#_Toc206290749)

[Hình 11. Kết quả kiểm thử khi thay đổi JSON 36](#_Toc206290750)

[Hình 12. Kết quả kiểm thử khi gạt xi nhan trái 36](#_Toc206290751)

[Hình 13. Kết quả đo latency với Busload 0 ~ 5% 37](#_Toc206290752)

[Hình 14. Kết quả đo latency trung bình với Busload 0 ~ 5% 38](#_Toc206290753)

[Hình 15. Kết quả đo latency với Busload 20 ~ 40% 38](#_Toc206290754)

[Hình 16. Kết quả đo latency với Busload ~ 70% 39](#_Toc206290755)

**MỤC LỤC BẢNG**

[Bảng 1. Danh sách vật tư sử dụng trong dự án 7](#_Toc206290756)

[Bảng 2. Sơ đồ kết nối phần ECU nodes (sử dụng STM32F1) 8](#_Toc206290757)

[Bảng 3. Định nghĩa các tín hiệu, phần cứng, và bảng tin CAN 11](#_Toc206290758)

[Bảng 4. Định nghĩa tín hiệu Digital Output Command 15](#_Toc206290759)

[Bảng 5. Định nghĩa tín hiệu Digital Output Response 16](#_Toc206290760)

[Bảng 6. Định nghĩa tín hiệu Digital Input Response 17](#_Toc206290761)

[Bảng 7. Định nghĩa tín hiệu Analog Input Response 17](#_Toc206290762)

# 1. Tổng quan dự án

## 1.1. Giới thiệu

Ngành công nghiệp ô tô đang trải qua một cuộc cách mạng sâu sắc, được thúc đẩy bởi hai xu hướng chính: điện hóa hệ thống truyền động và số hóa buồng lái. Sự chuyển dịch từ động cơ đốt trong sang xe điện (EV) không chỉ thay đổi cách xe vận hành mà còn tái định hình hoàn toàn kiến trúc điện-điện tử (E/E) bên trong. Song song đó, buồng lái kỹ thuật số (digital cockpit) đang thay thế các đồng hồ cơ học truyền thống, biến cụm đồng hồ (instrument cluster) và hệ thống thông tin giải trí (infotainment) thành trung tâm của trải nghiệm người-máy (Human-Machine Interface - HMI). Trong bối cảnh này, HMI không còn đơn thuần là công cụ hiển thị thông tin mà đã trở thành một yếu tố then chốt quyết định sự an toàn, tiện nghi và cảm xúc của người lái.

Dự án này được xây dựng để mô phỏng quy trình phát triển một thành phần cốt lõi trong kiến trúc ô tô hiện đại: một hệ thống tích hợp giữa Cụm Đồng hồ Kỹ thuật số (Instrument Panel Cluster - IPC) và Bộ Điều khiển Xe (Vehicle Control Unit - VCU). Thay vì chỉ là một bài tập lập trình nhúng đơn thuần, dự án đặt mục tiêu tiếp cận vấn đề theo tư duy của các kỹ sư ô tô chuyên nghiệp, áp dụng các tiêu chuẩn ngành, kiến trúc hệ thống thực tế và các nguyên tắc thiết kế HMI tiên tiến. Sản phẩm cuối cùng không chỉ là một màn hình hiển thị, mà là một hệ thống hoàn chỉnh có khả năng nhận thức, xử lý và tương tác với người dùng một cách thông minh và an toàn.

## 1.2. Mục tiêu Dự án (Project Objectives)

Để hiện thực hóa tầm nhìn trên, dự án đề ra các mục tiêu cụ thể, được phân loại thành mục tiêu kỹ thuật và mục tiêu chất lượng.

### 1.2.1. Mục tiêu Kỹ thuật

* Phát triển một hệ thống cụm đồng hồ kỹ thuật số (IPC) hoàn chỉnh, có khả năng hiển thị thông tin xe một cách linh động và theo thời gian thực. Hệ thống sẽ được xây dựng trên nền tảng vi xử lý Raspberry Pi, sử dụng framework Qt/QML để thiết kế giao diện đồ họa.
* Mô phỏng các chức năng của một Bộ Điều khiển Xe (VCU) thực thụ. VCU sẽ là bộ não trung tâm, chịu trách nhiệm quản lý các trạng thái hoạt động của xe (ví dụ: Tắt, Sạc, Lái), xử lý dữ liệu từ các bộ điều khiển điện tử (ECU) được mô phỏng, và ra quyết định điều khiển giao diện HMI.
* Thiết lập một giao thức truyền thông nối tiếp dựa trên mạng CAN (Controller Area Network). Giao thức này sẽ đảm bảo việc truyền dữ liệu có cấu trúc, đáng tin cậy giữa VCU (Raspberry Pi) và các nút ECU (mô phỏng bằng STM32).

### 1.2.2. Mục tiêu chất lượng

* Đạt được hiệu năng cao, với thời gian khởi động hệ thống nhanh và giao diện đồ họa mượt mà, đáp ứng tức thời các tương tác của người dùng.
* Đảm bảo độ tin cậy và tính chính xác tuyệt đối của các dữ liệu quan trọng hiển thị cho người lái, đặc biệt là tốc độ, mức năng lượng và các cảnh báo an toàn.
* Thiết kế HMI tuân thủ các nguyên tắc cốt lõi về an toàn chức năng theo tiêu chuẩn ISO 26262 và sử dụng các biểu tượng (tell-tales) theo tiêu chuẩn quốc tế ISO 2575 để đảm bảo tính quen thuộc và dễ nhận biết.
* Tạo ra một sản phẩm cuối cùng có giao diện trực quan, thẩm mỹ cao, và có khả năng tùy biến linh hoạt theo các ngữ cảnh khác nhau. Đặc biệt dễ dàng scale up và cập nhật cấu hình phần cứng của hệ thống.

## 1.3. Phạm vi Dự án (Project Scope)

Việc xác định rõ ràng phạm vi giúp tập trung nguồn lực và đảm bảo dự án khả thi.

### 1.3.1. Trong phạm vi (In-Scope)

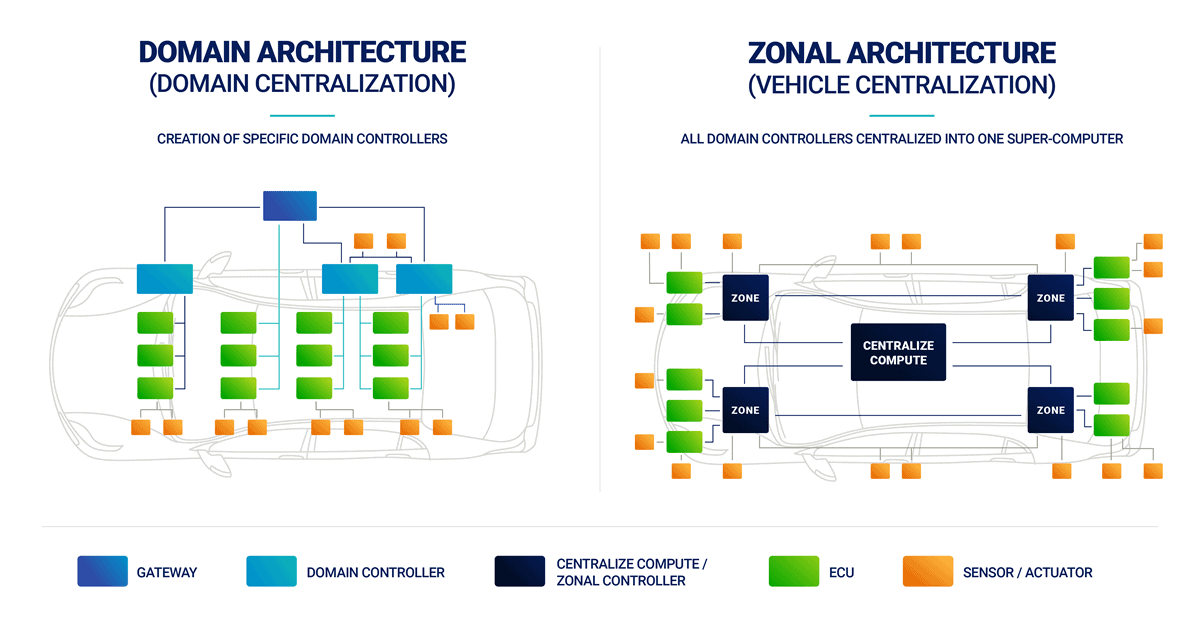
* **Phần cứng:** Thiết kế và lắp ráp một mô hình phần cứng hoàn chỉnh bao gồm VCU (Raspberry Pi), ECU nodes (mô phỏng bằng STM32), các cảm biến mô phỏng (encoder, chiết áp, nút bấm), và màn hình HMI.
* **Phần mềm:** Phát triển phần mềm cho các thành phần chính:
  + Porting driver cho màn hình, CAN trong Linux kernel
  + Phần mềm VCU trên Raspberry Pi sử dụng C++ và Qt/QML.
  + Firmware cho ECU trên STM32.
* **Giao diện HMI:** Phát triển một giao diện hiện đại và trực quan cho người dùng với đầy đủ các chức năng của một màn hình hiển thị bảng đồng hồ kỹ thuật số trên xe, gồm các thông số như tốc độ, pin, nhiệt độ, đèn báo
* **Chức năng VCU:**
  + Nhận và truyền dữ liệu để điều khiển hệ thống trên xe qua CAN bus.
  + Cấu hình IO (digital/analog) linh hoạt qua file JSON.
  + Đảm bảo giao diện hoạt động mượt mà và ổn định trên nền tảng nhúng.
* **Kiểm thử:** Xây dựng kế hoạch và thực hiện các cấp độ kiểm thử khác nhau, từ kiểm thử đơn vị đến kiểm thử hệ thống.

### 1.3.2. Ngoài phạm vi (Out-of-Scope)

* **Chứng nhận Tiêu chuẩn:** Dự án sẽ tuân thủ các nguyên tắc của ISO 26262 nhưng không thực hiện quy trình chứng nhận chính thức vốn rất tốn kém và phức tạp).
* **Hệ thống ADAS:** Không tích hợp các chức năng Hỗ trợ Lái xe Nâng cao (ADAS) phức tạp như ga tự động thích ứng (Adaptive Cruise Control) hay giữ làn đường.
* **Kết nối Mạng:** Không triển khai kết nối không dây (Wi-Fi, Bluetooth) hay các dịch vụ dựa trên đám mây (Cloud services).
* **Hệ thống Chẩn đoán:** Không triển khai các chức năng chẩn đoán lỗi (DTC) hoặc xử lý mã lỗi chẩn đoán.

# 2. Tổng quan kiến trúc hệ thống

## 2.1. Kiến trúc Zonal (Zonal Architecture)

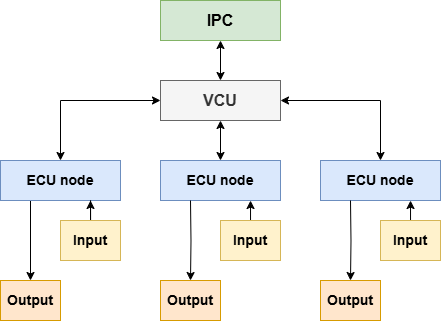


Hình 1. Kiến trúc Domain (Domain Architecture) và kiến trúc Zonal (Zonal Architecture)

Zonal Architecture, hay còn gọi là kiến trúc tập trung theo vùng, là một xu hướng mới trong thiết kế hệ thống điện tử ô tô hiện đại. Thay vì sử dụng các bộ điều khiển trung tâm theo từng miền chức năng như trong kiến trúc Domain, Zonal Architecture tổ chức hệ thống theo từng khu vực địa lý trên xe. Mỗi vùng (zone) có một bộ điều khiển zonal controller để thu thập và quản lý dữ liệu từ các ECU, cảm biến và cơ cấu chấp hành (actuator) gần đó, sau đó truyền dữ liệu đến một máy tính trung tâm (Centralized Compute) để xử lý toàn cục.

Lợi ích của Zonal Architecture bao gồm giảm độ phức tạp dây dẫn, dễ dàng nâng cấp phần cứng/phần mềm, và giúp tối ưu hóa hiệu suất hệ thống tổng thể. Đây là bước tiến quan trọng hướng tới xe điện, xe tự hành, và kiến trúc phần mềm định nghĩa (Software Defined Vehicle – SDV).

## 2.2. Sơ đồ Khối Hệ thống (System Block Diagram)



Hình 2. Sơ đồ khối hệ thống

Sơ đồ khối trên cung cấp mô tả kiến trúc Zonal Architecture đơn giản hoá, thường được ứng dụng trong ngành ô tô hoặc các hệ thống nhúng phân tán. Dưới đây là phần mô tả chi tiết:

Các thành phần trong sơ đồ:

* IPC (Instrument Panel Cluster)
  + Là cụm hiển thị/thông tin như màn hình táp-lô, báo hiệu, trạng thái hệ thống.
  + Kết nối với VCU, nhận thông tin để hiển thị (cảnh báo lỗi, trạng thái I/O, dữ liệu sensor…).
* VCU (Vehicle Control Unit)
  + Là trung tâm điều khiển, giao tiếp 2 chiều:
    - Với IPC: cung cấp dữ liệu để hiển thị và nhận lệnh từ người dùng.
    - Với các ECU node qua CAN.
  + Vai trò:
    - Điều phối dự liệu truyền/nhận với các ECU node.
    - Cấu hình lại hoặc gửi lệnh điều khiển các I/O.
    - Xử lý logic cao cấp, phản hồi từ IPC.
* ECU node
  + Số lượng không giới hạn trong hệ thống
  + Firmware giống nhau
  + Số lượng I/O giống nhau
  + Không biết trước mình đang kết nối với thiết bị gì
  + Cập nhật trạng thái cho VCU và nhận chỉ thị điều khiển từ VCU
* Input
  + Là cảm biến, nút nhấn, công tắc.
  + Đưa dữ liệu vào ECU node.
* Output
  + Là thiết bị điều khiển: relay, đèn, còi.
  + Nhận lệnh từ ECU node (theo chỉ đạo của VCU).

## 2.3. Luồng Dữ liệu và Tương tác

Để làm rõ hơn về cách hệ thống hoạt động, hãy xem xét một chu trình tương tác điển hình khi người lái tăng tốc

1. **Tương tác Vật lý:** Trên phần cứng (ECU), người dùng xoay núm biến trở, mô phỏng việc bánh xe quay nhanh hơn. Khi đó, giá trị điện áp đầu ra thay đổi.
2. **Thu thập Dữ liệu (ECU node):** Firmware trên STM32 đọc giá trị ADC thông qua DMA mỗi 50ms. Giá trị này được lưu vào bộ đệm.
3. **Đóng gói dữ liệu và truyền qua CAN:** Giá trị Analog được đóng gói vào khung dữ liệu CAN (CAN Frame) trong hàm xử lý. Các khung dữ liệu sẽ được sắp xếp vào 1 FIFO queue để truyền ra ngoài CAN bus.
4. **Nhận dữ liệu CAN trên VCU:** Ứng dụng trên VCU nhận khung CAN chứa giá trị analog (tốc độ). Để nhận biết được đâu là dữ liệu của tốc độ trên khung dữ liệu CAN được nhận thì khi startup, VCU sẽ đọc file JSON để tải cấu hình I/O vào bộ nhớ.
5. **Cập nhật UI**

* Giá trị tốc độ được phát qua tín hiệu “speedChanged” từ “CanHandler”.
* Trong QML, khi nhận được tín hiệu, giá trị được thêm vào buffer để xử lý và hiển thị lên giao diện (speedometer).

**Tóm tắt:**

Người dùng tăng tốc → ECU đọc ADC → gửi giá trị qua CAN → VCU nhận CAN frame và tách lấy dữ liệu speed nhờ file cấu hình JSON → cập nhật tốc độ trên UI.

Toàn bộ quá trình này diễn ra liên tục, tạo ra một hệ thống phản hồi theo thời gian thực cho người lái.

# 3. Thiết kế phần cứng và kết nối

## 3.1. Danh sách vật tư (Bill of Materials - BOM)

Bảng dưới đây liệt kê chi tiết các linh kiện phần cứng cần thiết để xây dựng hệ thống trong đề tài này.

|  |  |  |
| --- | --- | --- |
| Mục | Khối | Linh kiện và mô tả |
| 1 | IPC | Màn hình Waveshare 7 inch HDMI Capacitive Touch Screen LCD |
| 2 | VCU | - Raspberry Pi 5 RAM 4GB, CPU 4 nhân 2.4GHz, Wi-Fi 5, BT 5.0, USB 3.0, micro-HDMI kép 4K, GPIO 40 chân, nguồn USB-C 5V/5A.  - USB to CAN (Makerbase CANable v1.0) để connect với CAN bus |
| 3 | ECU Node | - STM32F103C8, MCU ARM Cortex-M3 72 MHz, 64 KB Flash, 20 KB RAM.  - CAN Transceiver SN65HVD230, điện áp hoạt động 3.3V, tốc độ lên đến 1 Mbps. |
| 4 | Input | - Cảm biến nhiệt độ, nút bấm, công tắc. |
| 5 | Output | - Các đèn báo hiệu. |

Bảng 1. Danh sách vật tư sử dụng trong dự án

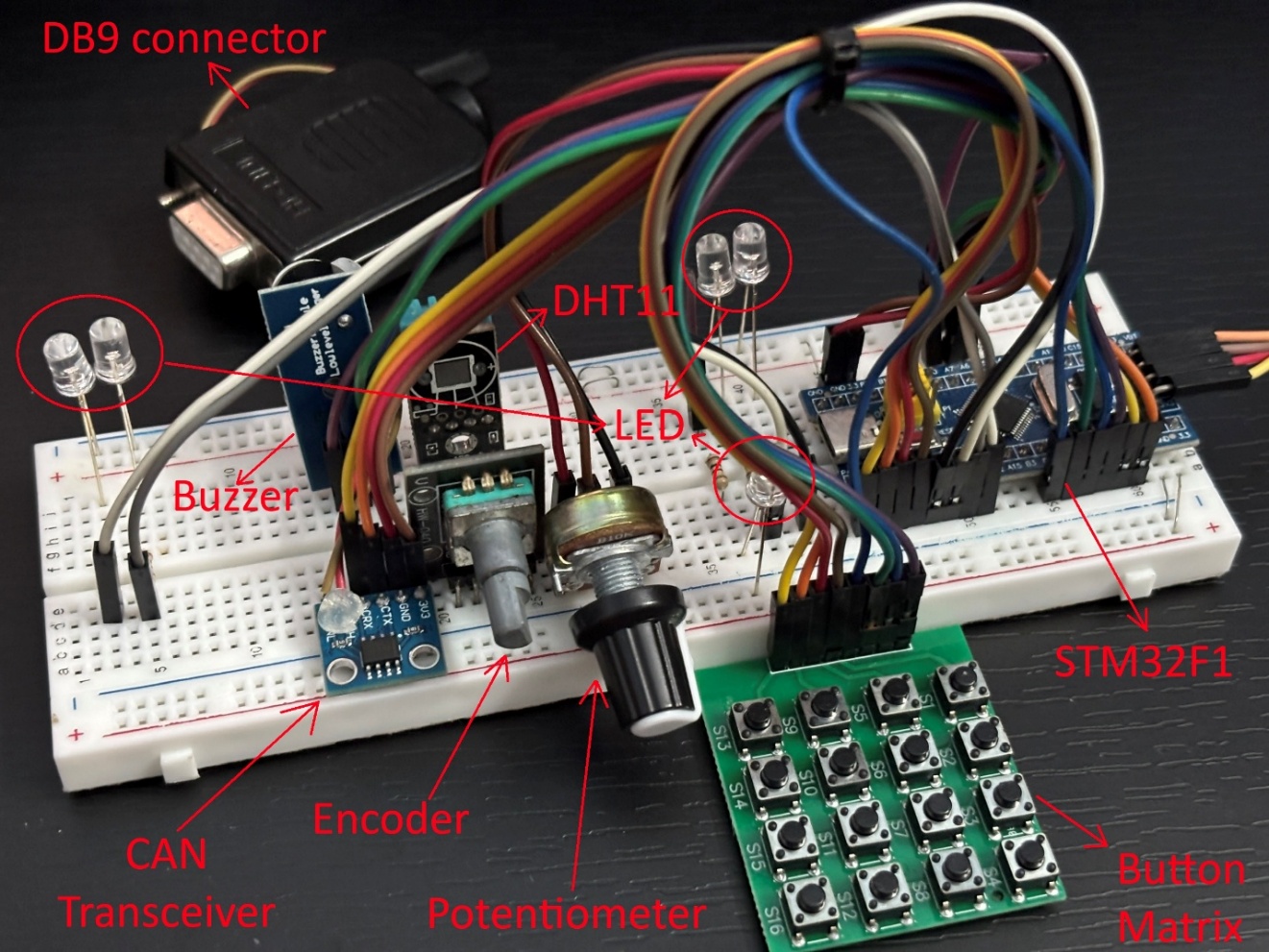
## 3.2. Sơ đồ kết nối

### 3.2.1. Sơ đồ kết nối phần ECU nodes

Bảng dưới đây mô tả sơ đồ kết nối của các nút ECU được mô phỏng bởi STM32F1.

|  |  |  |  |
| --- | --- | --- | --- |
| **STM32 Pin** | **Function** | **Signal Name** | **I/O Connection** |
| PA5 | ADC1\_IN5 | - | Potentiometer |
| PA6 | TIM3\_CH1 | DIGITAL\_OUT\_8 | Buzzer |
| PA7 | TIM3\_CH2 | DIGITAL\_OUT\_12 | Red LED |
| PB0 | TIM3\_CH3 | DIGITAL\_OUT\_22 | High Beam Light |
| PB1 | TIM3\_CH4 | DIGITAL\_OUT\_32 | Low Beam Light |
| PA8 | TIM1\_CH1 | DIGITAL\_OUT\_28 | Right Rear Light |
| PA9 | TIM1\_CH2 | DIGITAL\_OUT\_19 | Right Front Light |
| PA10 | TIM1\_CH3 | DIGITAL\_OUT\_10 | Left Front Light |
| PA11 | TIM1\_CH4 | DIGITAL\_OUT\_1 | Left Rear Light |
| PB12 | GPIO\_Output | BTN\_COL0 | Button matrix (column 1) |
| PB13 | GPIO\_Output | BTN\_COL1 | Button matrix (column 2) |
| PB14 | GPIO\_Output | BTN\_COL2 | Button matrix (column 3) |
| PB15 | GPIO\_Output | BTN\_COL3 | Button matrix (column 4) |
| PB4 | GPIO\_EXTI4 | BTN\_ROW0 | Button matrix (row 1) |
| PB5 | GPIO\_EXTI5 | BTN\_ROW1 | Button matrix (row 2) |
| PB6 | GPIO\_EXTI6 | BTN\_ROW2 | Button matrix (row 3) |
| PB7 | GPIO\_EXTI7 | BTN\_ROW3 | Button matrix (row 4) |
| PB8 | CAN\_RX | - | CAN transceiver (CRX) |
| PB9 | CAN\_TX | - | CAN transceiver (CTX) |

Bảng 2. Sơ đồ kết nối phần ECU nodes (sử dụng STM32F1)



Hình 3. Kết nối liên quan STM32F1

### 3.2.2. Kết nối phần VCU

VCU (Raspberry Pi) kết nối với hệ thống qua USB to CAN để giao tiếp dữ liệu xe và sử dụng màn hình HDMI (kết nối bằng cáp micro HDMI to HDMI) để hiển thị giao diện bảng đồng hồ kỹ thuật số.



Hình 4. Kết nối toàn bộ phần cứng trong thực tế

Tham khảo hình ảnh setup của dự án và file 3D tại đường link:

[VCU\_Cluster\_System/hardware at main · Tdieney/VCU\_Cluster\_System · GitHub](https://github.com/Tdieney/VCU_Cluster_System/tree/main/hardware)

# 4. Quy trình phát triển phần mềm

## 4.1. Giao tiếp CAN giữa VCU và ECU nodes

### 4.1.1. Truyền tin giữa các thành phần

Trong hệ thống này, VCU không trực tiếp nắm mọi thông tin, mà thu thập từ các ECU khác qua cùng 1 bus CAN. Bảng dưới đây mô tả sơ lược về sự kết nối và truyền tin được sử dụng trong đề tài này.

|  |  |  |
| --- | --- | --- |
| Thông tin giao diện | Kết nối mô phỏng | Dạng CAN message |
| Tốc độ xe | Input: Biến trở | ECU → VCU: Định kỳ, 50ms |
| Xi-nhan trái/phải | Input: Công tắc  Output: LED | ECU → VCU: Định kỳ, 50 ms  VCU → ECU: Định kỳ, 50 ms |
| Đèn chiếu xa, chiếu gần, đèn hazard | Input: Công tắc  Output: LED | ECU → VCU: Định kỳ, 50 ms  VCU → ECU: Định kỳ, 50 ms |
| Đèn báo hiệu đèn chiếu gần | Input: Công tắc  Output: LED | ECU → VCU: Định kỳ, 50 ms  VCU → ECU: Định kỳ, 50 ms |
| Đèn báo hiệu đèn chiếu xa | Input: Công tắc  Output: LED | ECU → VCU: Định kỳ, 50 ms  VCU → ECU: Định kỳ, 50 ms |
| Đèn sương mù sau bật | Input: Công tắc  Output: LED | ECU → VCU: Định kỳ, 50 ms  VCU → ECU: Định kỳ, 50 ms |
| Quãng đường đã đi | Input: Tổng hợp từ Encoder | ECU → VCU: Định kỳ, 50 ms |
| Cần số P/R/N/D | Input: Công tắc | ECU → VCU: Định kỳ, 50 ms |
| Nhiệt độ ngoài trời | Input: Cảm biến nhiệt độ | ECU → VCU: Định kỳ, 50 ms |

Bảng 3. Định nghĩa các tín hiệu, phần cứng, và bảng tin CAN

### 4.1.2. Kiến trúc linh hoạt dựa trên cấu hình I/O động

Như đã đề cập ở phần trước, trong hệ thống này, tất cả các ECU node đều được thiết kế với số lượng chân I/O giống nhau, nhưng không bị ràng buộc cố định vào vai trò cụ thể nào (như đèn, nút, cảm biến…). Điều này có nghĩa là mỗi node không cần biết trước nó đang điều khiển cái gì, mà toàn bộ hành vi sẽ được cấu hình động thông qua file cấu hình mà VCU sẽ đọc.

Dữ liệu cấu hình I/O sẽ chứa trong file .json ở dạng như sau:

{

"digital\_inputs": {

"ignition": 0,

"turn\_left\_switch": 9,

"turn\_right\_switch": 18,

"hazard\_switch": 27,

"high\_beam\_switch": 1,

"low\_beam\_switch": 10,

"parking\_lights\_switch": 19

},

"analog\_inputs": {

"speed": 0

},

"digital\_outputs": {

"left\_front\_light": 0,

"left\_rear\_light": 9,

"right\_front\_light": 18,

"right\_rear\_light": 27

}

}

* **digital\_inputs**:
  + Ánh xạ tên tín hiệu số đầu vào (ví dụ: công tắc, nút nhấn) với vị trí của nó trong hệ thống tương ứng.
  + Ví dụ: `"ignition": 0` nghĩa là tín hiệu "ignition" được gán index 0.
* **analog\_inputs**:
  + Ánh xạ các tín hiệu analog đầu vào (ví dụ: cảm biến tốc độ) với vị trí của nó trong hệ thống tương ứng.
* **digital\_outputs**:
  + Ánh xạ tên tín hiệu số đầu ra (ví dụ: đèn, relay) với vị trí của nó trong hệ thống tương ứng.

- Điều này giúp phần mềm dễ dàng cấu hình, mở rộng hoặc thay đổi logic I/O mà không cần sửa mã nguồn.

- Đảm bảo tính linh hoạt khi tích hợp với nhiều loại phần cứng hoặc thay đổi sơ đồ kết nối.

### 4.1.3. Định nghĩa bảng tin CAN

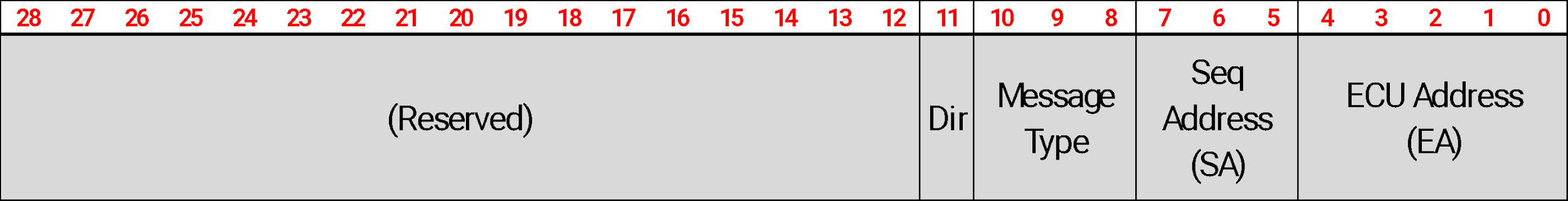
Dựa theo kiến trúc đã trình bày, việc trao đổi dữ liệu giữa các ECU node và VCU cần được thực hiện một cách thống nhất và linh hoạt. Vì vậy, các bảng tin CAN (CAN frames) sẽ tuân theo một cấu trúc cố định, nhưng có nội dung hoàn toàn cấu hình được thông qua file mà VCU đọc.

#### 4.1.3.1. Định dạng CAN ID

Trong đề tài này, có 2 loại message chính liên quan đến phần điều khiển I/O thông qua các ECU nodes:

* Digital Output
* Digital Input

Vì vậy mà ta cần định nghĩa format cho CAN ID để chúng nhận biết được Data của CAN Frame đó mang ý nghĩa gì. Dưới đây là bảng định nghĩa format CAN ID (chỉ sử dụng CAN ID 29-bit cho phần này)



* ECU Address: Trường này cho biết Frame này dùng để tương tác với ECU nào trong hệ thống
  + 00000: ECU1
  + 00001: ECU2
  + 00010: ECU3
  + 00011: ECU4
  + 00100: ECU5
  + 00101: ECU6
  + 00110: ECU7
  + 00111: ECU8
  + 01000: ECU9
  + 01001: ECU10
  + 01010: ECU11
  + 01011: ECU12
  + 01100: ECU13
  + 01101: ECU14
  + 01110: ECU15
  + 01111: ECU16
  + 10000: ECU17
  + 10001: ECU18
  + 10010: ECU19
  + 10011: ECU20
  + 10100: ECU21
  + 10101: ECU22
  + 10110: ECU23
  + 10111: ECU24
  + 11000: ECU25
  + 11001: ECU26
  + 11010: ECU27
  + 11011: ECU28
  + 11100: ECU29
  + 11101: ECU30
  + 11110: ECU31
  + 11111: ECU32
* Sequene Address: Trường này đánh dấu thứ tự Frame
  + 000: Dành cho signal trên frame thứ 1
  + 001: Dành cho signal trên frame thứ 2
  + 010: Dành cho signal trên frame thứ 3
  + 011: Dành cho signal trên frame thứ 4
  + 100: Dành cho signal trên frame thứ 5
  + 101: Dành cho signal trên frame thứ 6
  + 110: Dành cho signal trên frame thứ 7
  + 111: Dành cho signal trên frame thứ 8
* Message Type: Trường này dùng để phân loại message
  + 000: Digital Output message
  + 001: Digital Output config
  + 010: Digital Input message
  + 010 – 111: Reserved
* Dir (Direction): Hướng của message
  + 0: VCU → ECU
  + 1: ECU → VCU

#### 4.1.3.2. Khung dữ liệu Digital Output

**VCU → ECU**

VCU truyền dữ liệu gồm các tín hiệu điều khiển 1 output cụ thể được kết nối với ECU cụ thể trong hệ thống. Hình sau thể hiện tín hiệu để điều khiển 1 Digital Output cụ thể.

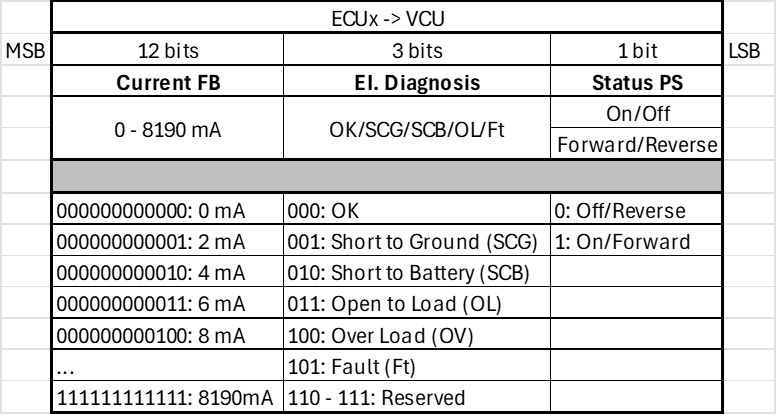


Bảng 4. Định nghĩa tín hiệu Digital Output Command

* Giải thích các trường trong 1 tín hiệu của hình trên:
  + Duty Cycle [7 bits]: Mức xung PWM từ 0 đến 100%
  + Switch Command [1 bit]: Bật/tắt hoặc Đảo chiều
* Theo hình trên, 1 tín hiệu gồm 8-bit. Với 1 Frame CAN có 8-byte (tức là 64-bit dữ liệu), số lượng tín hiệu tối đa có thể chứa được: 64 / 8 = **8 tín hiệu**
* Ta có tối đa là 8 giá trị để đánh thứ tự cho các frame (dựa theo trường **Sequence Address** ở mục **4.1.3.1**). Vậy đối với 1 ECUx cụ thể, ta có thể truyền lệnh và điều khiển lên đến: 8 x 8 = **64 output**

**ECU → VCU**

ECU truyền dữ liệu định kỳ hoặc theo sự kiện đến cho VCU để cập nhật trạng thái của các Digital Output kết nối với chúng. Hình sau thể hiện data của tín hiệu của 1 Digital Output được truyền từ ECU đến VCU.



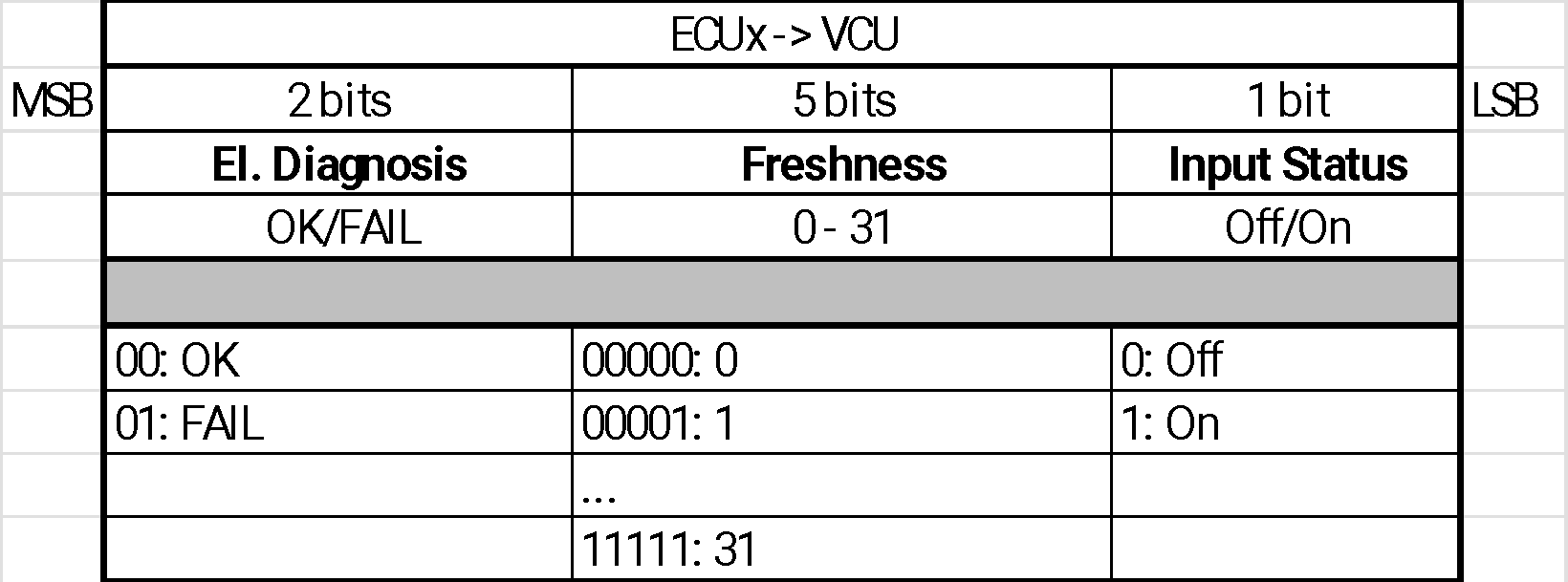
Bảng 5. Định nghĩa tín hiệu Digital Output Response

* Giải thích các trường trong 1 tín hiệu của hình trên:
  + Current FB [12 bits]: Dòng điện phản hồi từ ECU, đơn vị mA (0 – 8190 mA).
  + El. Diagnosis [3 bits]: Trạng thái chẩn đoán điện như OK, ngắn mạch, hở mạch...
  + Status PS [1 bit]: Trạng thái logic hoặc điều khiển: On/Off hoặc Forward/Reverse.
* Theo hình trên, 1 tín hiệu gồm 16-bit. Với 1 Frame CAN có 8-byte (tức là 64-bit dữ liệu), số lượng tín hiệu tối đa có thể chứa được: 64 / 16 = **4 tín hiệu**
* Ta có tối đa là 8 giá trị để đánh thứ tự cho các frame (dựa theo trường **Sequence Address** ở mục **4.1.3.1**). Vậy đối với 1 ECUx cụ thể, ta có thể nhận tín hiệu điều khiển trả về lên đến: 4 x 8 = **32 output**

#### 4.1.3.3. Khung dữ liệu Digital Input

**ECU → VCU**

ECU truyền dữ liệu định kỳ hoặc theo sự kiện đến cho VCU để cập nhật trạng thái của các Digital Input kết nối với chúng. Hình sau thể hiện data của tín hiệu của 1 Digital Input được truyền từ ECU đến VCU.



Bảng 6. Định nghĩa tín hiệu Digital Input Response

* Giải thích các trường trong 1 tín hiệu của hình trên:
  + El Diagnosis [2 bits]: Chẩn đoán điện (OK / FAIL)
  + Freshness [5 bits]: Chỉ số làm mới (0 – 31) để xác nhận dữ liệu còn hợp lệ. Tăng 1 đơn vị mỗi lần gửi
  + Input Status [1 bit]: Trạng thái tín hiệu đầu vào (Off / On)
* Theo hình trên, 1 tín hiệu gồm 8-bit. Với 1 Frame CAN có 8-byte (tức là 64-bit dữ liệu), số lượng tín hiệu tối đa có thể chứa được: 64 / 8 = **8 tín hiệu**
* Ta có tối đa là 8 giá trị để đánh thứ tự cho các frame (dựa theo trường **Sequence Address** ở mục **4.1.3.1**). Vậy đối với 1 ECUx cụ thể, ta có thể nhận tín hiệu điều khiển trả về lên đến: 8 x 8 = **64 input**

#### 4.1.3.4. Khung dữ liệu Digital Analog

**ECU → VCU**

ECU truyền dữ liệu định kỳ hoặc theo sự kiện đến cho VCU để cập nhật trạng thái của các Analog Input kết nối với chúng. Hình sau thể hiện data của tín hiệu của 1 Analog Input được truyền từ ECU đến VCU.



Bảng 7. Định nghĩa tín hiệu Analog Input Response

* Giải thích các trường trong 1 tín hiệu của hình trên:
  + El Diagnosis [2 bits]: Chẩn đoán điện (OK / FAIL)
  + Analog Value [14 bits]: Giá trị analog với độ phần giải 2 mV/đơn vị
* Theo hình trên, 1 tín hiệu gồm 16-bit. Với 1 Frame CAN có 8-byte (tức là 64-bit dữ liệu), số lượng tín hiệu tối đa có thể chứa được: 64 / 16 = **4 tín hiệu**
* Ta có tối đa là 8 giá trị để đánh thứ tự cho các frame (dựa theo trường **Sequence Address** ở mục **4.1.3.1**). Vậy đối với 1 ECUx cụ thể, ta có thể nhận tín hiệu điều khiển trả về lên đến: 8 x 4 = **32 input**

## 4.2. Thiết kế phần mềm VCU

### 4.2.1. Sử dụng USB to CAN

Raspberry Pi 5 không có sẵn CAN nên việc sử dụng USB-to-CAN là lựa chọn nhanh chóng và đơn giản để kết nối với mạng CAN bus.

SocketCAN là một phần của nhân Linux, cung cấp giao diện lập trình socket tương tự như mạng TCP/IP cho giao tiếp với CAN.

#### 4.2.1.1. Cấu hình USB to CAN

Sử công cụ can-utils sau khi đã kết nối thiết bị USB-to-CAN và cấu hình interface.

**Bật CAN interface**

Ví dụ bật can0 với tốc độ 1 Mbps:

**sudo ip link set can0 up type can bitrate 1000000**

Kiểm tra trạng thái:

**ip -details -statistics link show can0**

**Tắt interface khi không dùng**

**sudo ip link set can0 down**

**Ngoài ra**, nếu muốn test CAN loopback mode:

**sudo ip link set can0 down**

**sudo ip link set can0 up type can bitrate 1000000 loopback on**

#### 4.2.1.2. Lập trình SocketCAN trong Linux

SocketCAN là một phần mở rộng của ngăn xếp socket trong Linux để hỗ trợ giao tiếp với mạng CAN (Controller Area Network) theo cách tương tự như TCP/UDP. Không như các API độc quyền của nhà sản xuất thiết bị CAN, SocketCAN cung cấp một giao diện socket thuần túy, đồng nhất và độc lập phần cứng, rất phù hợp cho các ứng dụng nhúng hoặc firmware chạy trên Linux.

SocketCAN sử dụng:

* PF\_CAN: Protocol family cho CAN
* AF\_CAN: Address family cho CAN
* SOCK\_RAW / SOCK\_DGRAM: Loại socket
* Protocol CAN\_RAW: Cho phép truy cập CAN raw frame (CAN 2.0A/B)

**Lưu ý:** Bạn không cần phải cài “can-utils” để sử dụng socketCAN vì socketCAN là một phần của nhân Linux (kernel)

**Kiến trúc SocketCAN trong C**

SocketCAN không yêu cầu thư viện ngoài – chỉ cần các API chuẩn POSIX:

**#include <linux/can.h>**

**#include <linux/can/raw.h>**

**#include <sys/socket.h>**

**#include <sys/ioctl.h>**

**#include <net/if.h>**

**#include <unistd.h>**

**#include <string.h>**

**Các bước khởi tạo socket CAN trong C**

Bước 1: Tạo socket:

**int s = socket(PF\_CAN, SOCK\_RAW, CAN\_RAW);**

Bước 2: Lấy chỉ số giao diện can0, slcan0,...:

**struct ifreq ifr;**

**strcpy(ifr.ifr\_name, "can0");**

**ioctl(s, SIOCGIFINDEX, &ifr);**

Bước 3: Gắn socket vào giao diện:

**struct sockaddr\_can addr = {**

**.can\_family = AF\_CAN,**

**.can\_ifindex = ifr.ifr\_ifindex**

**};**

**bind(s, (struct sockaddr \*)&addr, sizeof(addr));**

**Gửi và nhận CAN frame**

Gửi một khung CAN:

**struct can\_frame frame;**

**frame.can\_id = 0x123; // 11-bit CAN ID**

**frame.can\_dlc = 8; // số byte dữ liệu**

**memcpy(frame.data, "\x11\x22\x33\x44\x55\x66\x77\x88", 8);**

**write(s, &frame, sizeof(frame));**

Nhận một khung CAN:

**read(s, &frame, sizeof(frame));**

**printf("ID: 0x%X DLC: %d\n", frame.can\_id, frame.can\_dlc);**

* Tất cả hoạt động đều giống như làm việc với file descriptor thông thường (có thể dùng poll(), select(), epoll()...).

**CAN ID và DLC**

can\_id: 11-bit hoặc 29-bit ID. Nếu muốn dùng extended ID (CAN 2.0B), gắn cờ **CAN\_EFF\_FLAG**:

can\_dlc: Số byte dữ liệu từ 0 đến 8 (CAN 2.0).

**frame.can\_id = 0x1FFFFFFF | CAN\_EFF\_FLAG;**

**frame.can\_dlc = 2;**

**Sử dụng CAN filter (lọc ID)**

Đặt filter trước khi bind():

**struct can\_filter rfilter[1];**

**rfilter[0].can\_id = 0x123;**

**rfilter[0].can\_mask = CAN\_SFF\_MASK; // So khớp toàn bộ 11 bit**

**setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_FILTER, &rfilter, sizeof(rfilter));**

**Ví dụ tổng hợp:**

**#include <stdio.h>**

**#include <string.h>**

**#include <unistd.h>**

**#include <net/if.h>**

**#include <sys/socket.h>**

**#include <sys/ioctl.h>**

**#include <linux/can.h>**

**#include <linux/can/raw.h>**

**int main() {**

**int s;**

**struct ifreq ifr;**

**struct sockaddr\_can addr;**

**struct can\_frame frame;**

**// Tạo socket**

**s = socket(PF\_CAN, SOCK\_RAW, CAN\_RAW);**

**// Gán vào can0**

**strcpy(ifr.ifr\_name, "can0");**

**ioctl(s, SIOCGIFINDEX, &ifr);**

**addr.can\_family = AF\_CAN;**

**addr.can\_ifindex = ifr.ifr\_ifindex;**

**bind(s, (struct sockaddr \*)&addr, sizeof(addr));**

**// Gửi frame**

**frame.can\_id = 0x321;**

**frame.can\_dlc = 2;**

**frame.data[0] = 0xAB;**

**frame.data[1] = 0xCD;**

**write(s, &frame, sizeof(frame));**

**// Nhận frame**

**read(s, &frame, sizeof(frame));**

**printf("ID: 0x%03X DLC: %d DATA:", frame.can\_id, frame.can\_dlc);**

**for (int i = 0; i < frame.can\_dlc; i++)**

**printf(" %02X", frame.data[i]);**

**printf("\n");**

**close(s);**

**return 0;**

**}**

### 4.2.2. Kết nối Qt/C++ với QML

Trong dự án này, việc xây dựng giao diện người dùng (HMI) hiện đại cho hệ thống điều khiển xe điện được thực hiện bằng Qt/QML, trong khi các logic xử lý nghiệp vụ, giao tiếp phần cứng (CAN, I/O), và quản lý trạng thái được triển khai bằng C++. Để hai phần này có thể "nói chuyện" với nhau một cách mượt mà, Qt cung cấp cơ chế **context property** và **QObject**.

Cụ thể, đối tượng C++ như *CanHandler* được khởi tạo trong file *main.cpp* và được đăng ký với QML thông qua dòng lệnh:

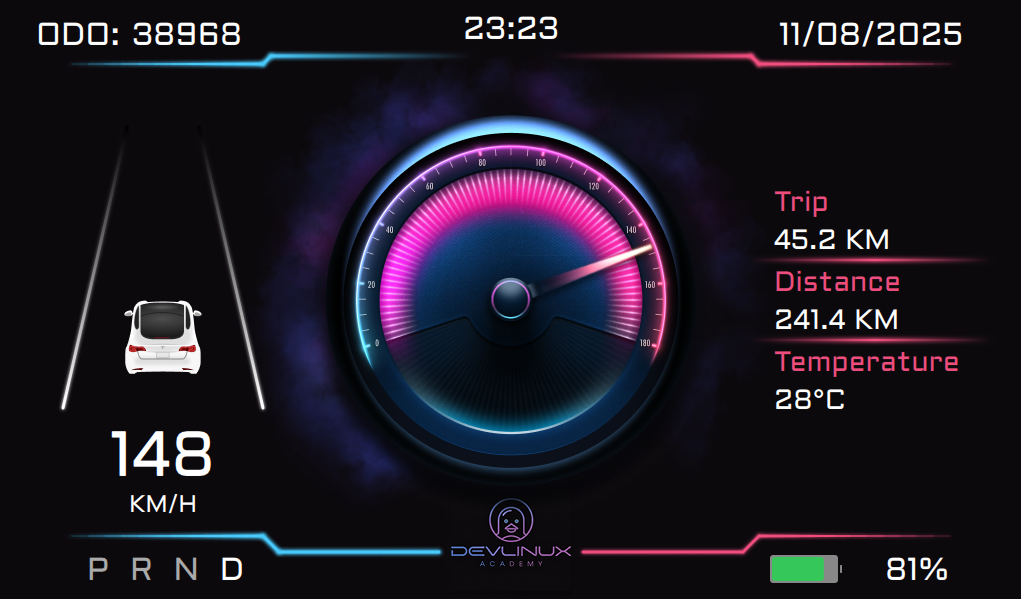
engine.rootContext()->setContextProperty("canHandler", &canHandler);

Nhờ đó, toàn bộ các file QML trong dự án đều có thể truy cập trực tiếp các thuộc tính, phương thức, và tín hiệu của đối tượng canHandler bằng tên canHandler. Điều này mở ra khả năng tương tác hai chiều: QML gọi hàm C++ để gửi lệnh xuống ECU, đồng thời C++ phát tín hiệu (signal) để cập nhật trạng thái lên giao diện.

Ví dụ, khi nhận được dữ liệu tốc độ mới từ CAN, CanHandler phát tín hiệu speedChanged, QML sẽ tự động cập nhật giao diện tốc độ xe mà không cần polling hay xử lý phức tạp.

### 4.2.3. Màn hình và giao diện IPC

Dưới đây là hình ảnh và giải thích về bố cục của giao diện HMI:



Hình 5. Giao diện màn hình IPC

Giao diện này được bố trí một cách trực quan và hiện đại, với các yếu tố sắp xếp hợp lý để tối ưu hóa khả năng đọc và sử dụng:

* **ODO: 38968**: Số kilomet tổng cộng mà xe đã đi được (odometer).
* **23:23**: Thời gian hiện tại.
* **11/08/2025**: Ngày hiện tại.
* **148 KM/H**: Tốc độ hiện tại của xe (kilomet mỗi giờ).
* **P R N D**: Chế độ số hiện tại (Park, Reverse, Neutral, Drive), với "D" (Drive) được chọn, cho thấy xe đang ở chế độ lái.
* **Trip 45.2 KM**: Khoảng cách đã đi trong chuyến đi hiện tại.
* **Distance 241.4 KM**: Khoảng cách còn lại đến khi hết pin.
* **Temperature 28°C**: Nhiệt độ môi trường bên ngoài.
* **81%**: Mức pin còn lại của xe.

Ngoài ra, sẽ còn có các đèn báo hiệu, xi-nhan khi người dùng tác động lên các cảm biến, công tắc trên xe.

Video demo:

<https://drive.google.com/file/d/1FQPo_EUh4a_-FMsMQpHUV6WIgGDGTdTw/view?usp=sharing>

### 4.2.4. Cấu trúc file project Qt

Cấu trúc project Qt cụ thể như sau:

.

├── main.cpp

├── main.qml

├── qtapp.pro

├── communication/

│ ├── canhandler.cpp

│ └── canhandler.h

├── fonts/

│ └── Aldrich-Regular.ttf

├── images/

│ ├── arrow.png

│ ├── background.png

│ ├── centre.png

│ ├── hazard.png

│ ├── high\_beam.png

│ ├── left\_arrow.png

│ ├── low\_beam.png

│ ├── parking\_lights.png

│ └── right\_arrow.png

├── io\_configs/

│ └── io\_config.json

├── \*.qrc

**Giải thích từng phần**

* **main.cpp:** Khởi tạo ứng dụng Qt, kết nối các thành phần backend với giao diện QML.
* **main.qml:** Định nghĩa giao diện người dùng, xử lý logic hiển thị các thông số xe.
* **qtapp.pro:** File cấu hình dự án Qt, quản lý tài nguyên và build.
* **communication/:** Chứa mã nguồn xử lý giao tiếp CAN bus (nhận, truyền dữ liệu xe).
* **fonts/:** Lưu trữ font chữ sử dụng cho giao diện.
* **images/:** Chứa các hình ảnh cho UI như kim đồng hồ, đèn báo, nền...
* **io\_configs/:** File cấu hình IO dạng JSON, cho phép tùy chỉnh mapping tín hiệu.
* **\*.qrc:** Các file Qt resource quản lý tài nguyên (QML, hình ảnh, font, cấu hình).

### 4.2.5. Cách binding dữ liệu

Binding dữ liệu là cốt lõi của giao diện động. Trong Qt, binding được thực hiện thông qua các **property** (Q\_PROPERTY) và **signal-slot** của QObject. Khi một property thay đổi, QML sẽ tự động cập nhật mọi thành phần liên quan trên giao diện.

Ví dụ, thuộc tính visible của icon xi nhan trái, QML chỉ cần:

Rectangle {

    id: turnLeftRec

    width: 40

    height: 40

    x: 39

    y: 77

    color: "transparent"

    Image {

        id: turnLeftImage

        source: "qrc:/images/left\_arrow.png"

        visible: false

        anchors.fill: parent

        fillMode: Image.PreserveAspectFit

    }

}

Khi giá trị thay đổi ở backend, giao diện sẽ tự động cập nhật mà không cần code thêm.

Đối với các sự kiện (event), QML sử dụng Connections để lắng nghe các signal từ C++:

Connections {

    target: canHandler

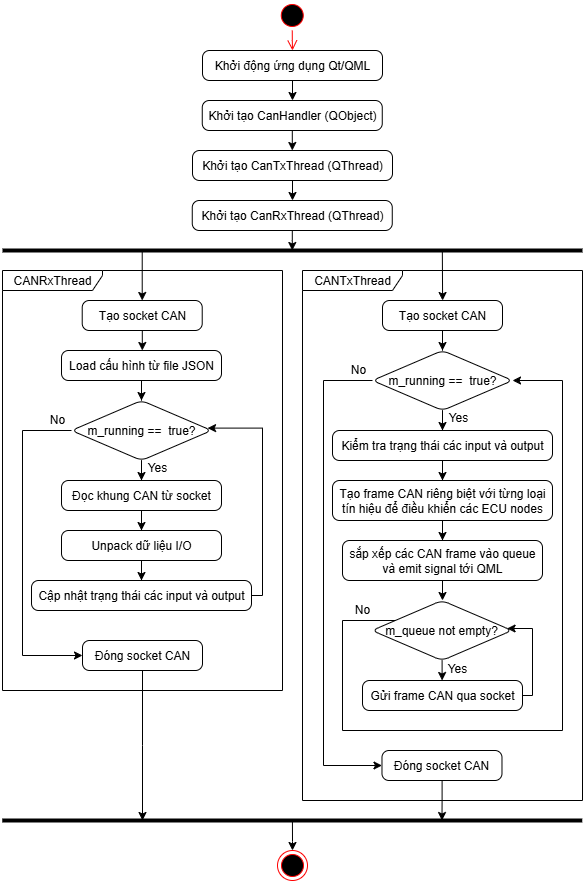
    onLeftLightChanged: turnLeftImage.visible = leftLight

}

Nhờ cơ chế này, mọi thay đổi dữ liệu từ backend C++ sẽ được phản ánh tức thời lên giao diện QML, giúp xây dựng HMI động, trực quan, dễ bảo trì và mở rộng. Việc binding còn giúp giảm thiểu lỗi đồng bộ dữ liệu, tăng hiệu năng và trải nghiệm người dùng.

### 4.2.6. Software VCU

VCU (sử dụng Raspberry Pi 5) là thành phần trung tâm, chịu trách nhiệm nhận dữ liệu từ các ECU qua CAN, xử lý và cập nhật cho giao diện Qt/QML.



Hình 6. Sơ đồ khối mô tả sơ lược logic khối VCU

Toàn bộ mã nguồn liên quan được lưu trữ tại thư mục “software/” trong đường dẫn GitHub sau:

[Tdieney/VCU\_Cluster\_System: Building a Vehicle Control Unit (VCU) with digital instrument cluster for electric vehicles.](https://github.com/Tdieney/VCU_Cluster_System)

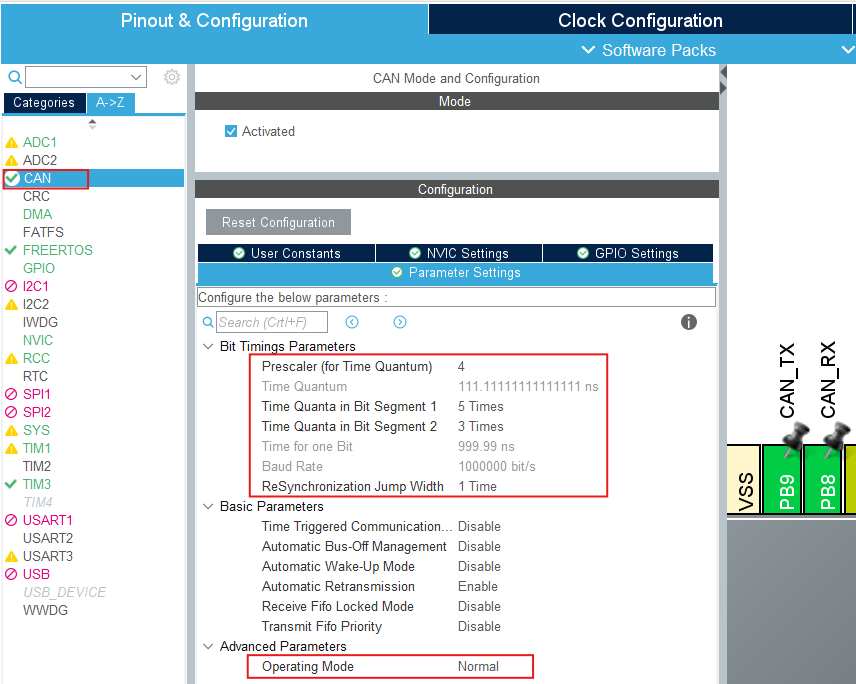
Bên cạnh đó, trong folder “docs/” còn có các bài viết liên quan đến Yocto để từng bước build và run software VCU trên Raspberry Pi 5:

* [Build image for Raspberry Pi 5](https://github.com/Tdieney/VCU_Cluster_System/blob/main/docs/yocto/1.%20Build%20image%20for%20Raspberry%20Pi%205.md)
* [CAN Communication](https://github.com/Tdieney/VCU_Cluster_System/blob/main/docs/yocto/2.%20CAN%20Communication.md)
* [Systemd startup script](https://github.com/Tdieney/VCU_Cluster_System/blob/main/docs/yocto/3.%20Systemd%20startup%20script.md)
* [Integrating Qt into a Yocto Project](https://github.com/Tdieney/VCU_Cluster_System/blob/main/docs/yocto/4.%20Integrating%20Qt%20into%20a%20Yocto%20Project.md)
* [Auto-launch Qt application](https://github.com/Tdieney/VCU_Cluster_System/blob/main/docs/yocto/5.%20Auto-launch%20Qt%20application.md)

## 4.3. Firmware cho STM32 (mô phỏng lại các ECU nodes)

### 4.3.1. Cấu hình và giao tiếp CAN cơ bản

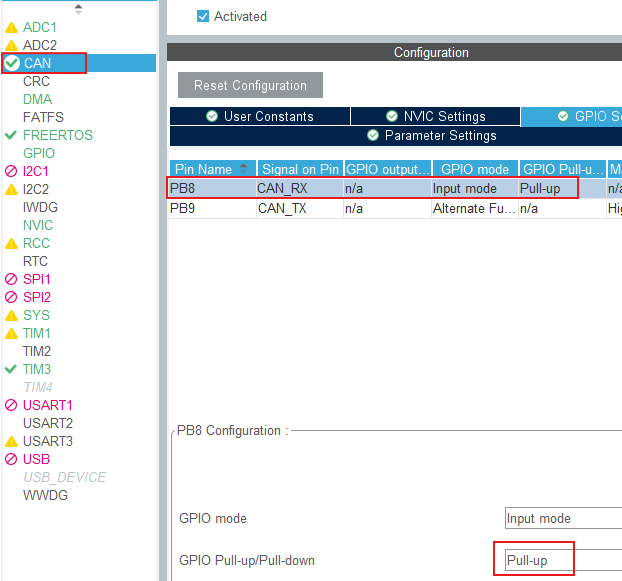
Cấu hình CubeMX được hiển thị như bên dưới.



Hình 7. Cấu hình CAN controller cho STM32F1

* Tốc độ truyền (BAUD RATE) được đặt là 1 Mbps. Bạn có thể thử các tổ hợp khác nhau cho Prescalar và Time Quanta để đạt được giá trị này.
* Chế độ hoạt động là Chế độ Bình thường (NORMAL Mode).
* Chân PA11 và PA12 được cấu hình lần lượt là CAN\_RX và CAN\_TX.

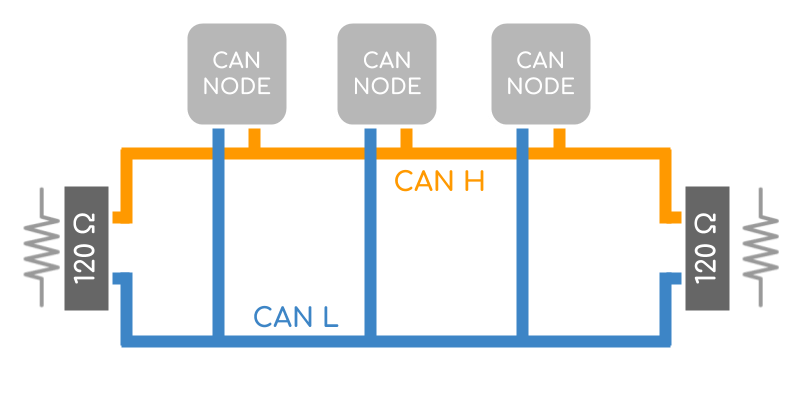
Chúng ta cũng cần kích hoạt pull-up cho chân RX như được hiển thị bên dưới.



Hình 8. Cấu hình GPIO của STM32F1 sử dụng CAN

Kết nối giữa VCU và ECU node:

* Kết nối CANH và CANL của USB-to-CAN và CAN transceiver (đã connect với STM32F1) với nhau
* Cần phải có điện trở 120 Ohm ở đầu và cuối đường dây



Hình 9. CAN Bus

**Truyền dữ liệu CAN**

Sau khi generate code, định nghĩa một số biến để chứa header và data

**CAN\_TxHeaderTypeDef TxHeader;**

**uint8\_t TxData[8];**

**uint32\_t TxMailbox;**

* TxHeader sẽ được sử dụng để lưu thông tin tiêu đề, như RTR, DLC, v.v. Đây là kiểu dữ liệu CAN\_TxHeaderTypeDef.
* TxData được dùng để lưu dữ liệu mà chúng ta sẽ truyền qua bus CAN.
* TxMailbox là hộp thư, sẽ được gửi đến bus CAN.

Lưu dữ liệu cần thiết vào **TxHeader** và **TxData**

**TxHeader.IDE = CAN\_ID\_STD;**

**TxHeader.StdId = 0x446;**

**TxHeader.RTR = CAN\_RTR\_DATA;**

**TxHeader.DLC = 2;**

**TxData[0] = 50;**

**TxData[1] = 0xAA;**

* CAN\_ID\_STD có nghĩa là chúng ta đang sử dụng Standard CAN ID (11-bit)
* 0x446 là ID.
* CAN\_RTR\_DATA chỉ ra rằng chúng ta đang gửi một khung dữ liệu.
* DLC là độ dài của các byte dữ liệu
* Sau đó lưu 2 byte dữ liệu vào mảng TxData.

Truyền dữ liệu trên CAN bus

**if (HAL\_CAN\_AddTxMessage(&hcan, &TxHeader, TxData, &TxMailbox) != HAL\_OK)**

**{**

**Error\_Handler ();**

**}**

Sử dụng hàm HAL\_CAN\_AddTxMessage với các tham số sau:

* hcan là phiên bản của CAN mà chúng ta đang sử dụng.
* TxHeader là tiêu đề của thông điệp.
* TxData là trường dữ liệu.
* TxMailbox là hộp thư, sẽ chứa tiêu đề và thông điệp dữ liệu.

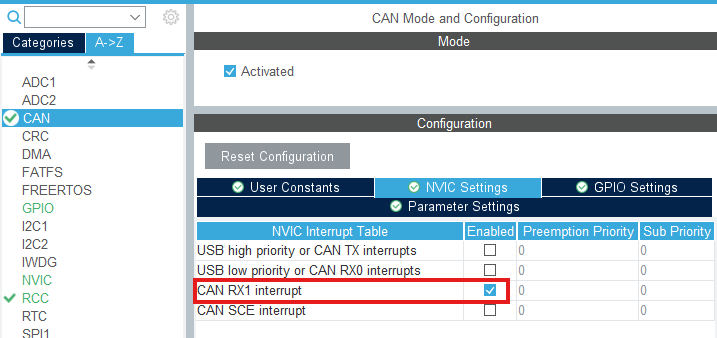


Bạn có thể thấy dữ liệu trên CAN bus như hình trên.

* Mã định danh là 0x446, ID Chuẩn (STD ID) của bộ truyền.
* Trường điều khiển là 0x2, chứa DLC, RTR, IDE.
* 2 byte của trường dữ liệu.
* Cuối cùng là giá trị CRC, được thêm bởi HAL.

**Nhận dữ liệu CAN**

Cho phép ngắt trên CAN



Hình 10. Cấu hình ngắt để nhận khung CAN

**Activate the Notification** để nhận CAN frame

**If (HAL\_CAN\_ActivateNotification(&hcan1, CAN\_IT\_RX\_FIFO0\_MSG\_PENDING) != HAL\_OK)**

**{**

**Error\_Handler();**

**}**

* Chọn CAN\_IT\_RX\_FIFO0\_MSG\_PENDING. Điều này sẽ kích hoạt ngắt mỗi khi có thông điệp đang chờ xử lý trong RX\_FIFO 0. Khi ngắt được kích hoạt, một hàm gọi lại (callback function) sẽ được gọi. Trong trường hợp này, đó sẽ là HAL\_CAN\_RxFifo0MsgPendingCallback.

**CAN\_RxHeaderTypeDef RxHeader;**

**uint8\_t RxData[8];**

**void HAL\_CAN\_RxFifo0MsgPendingCallback(CAN\_HandleTypeDef \*hcan)**

**{**

**if (HAL\_CAN\_GetRxMessage(hcan, CAN\_RX\_FIFO0, &RxHeader, RxData) != HAL\_OK)**

**{**

**Error\_Handler();**

**}**

**if ((RxHeader.StdId == 0x103))**

**{**

**datacheck = 1;**

**}**

**}**

* Chúng ta sẽ nhận thông điệp từ RX\_FIFO 0.
* Tiêu đề thông điệp sẽ được lưu trong RxHeader, và dữ liệu sẽ được lưu trong RxData.
* Chúng ta có thể thực hiện các kiểm tra bổ sung, chẳng hạn như nếu thông điệp được nhận từ ID 0x103, thì cờ datacheck sẽ được đặt.
* Sau đó, trong vòng lặp while, chúng ta có thể thực hiện một số hành động dựa trên cờ này.

### 4.3.2. Firmware cho ECU nodes (mô phỏng bằng STM32)

Node ECU được mô phỏng bằng vi điều khiển STM32. Toàn bộ mã nguồn firmware liên quan được lưu trữ tại thư mục “ecu\_nodes/” trong đường dẫn GitHub sau:

[Tdieney/VCU\_Cluster\_System: Building a Vehicle Control Unit (VCU) with digital instrument cluster for electric vehicles.](https://github.com/Tdieney/VCU_Cluster_System)

* Ghi chú: Firmware mô phỏng các chức năng cơ bản như ADC, CAN, PWM,... và tương tác với hệ thống chính thông qua giao tiếp CAN. Vui lòng xem README trong repository để biết chi tiết.

# 5. Thuật toán xử lý

## 5.1.1. Tính toán phần trăm pin

Trong file main.qml, quá trình tính toán pin (battery) được thực hiện như sau:

1. **Tính lượng điện năng tiêu thụ tức thời**

* Dựa vào tốc độ xe (speed), hàm consumptionFromSpeed(speed) tính ra mức tiêu thụ điện năng tức thời (kWh/100km).

1. **Tính điện năng tiêu thụ trong mỗi chu kỳ**

* energy\_used\_kWh\_this\_tick =

*(với “delta\_distance\_km” là quãng đường đi được trong chu kỳ Timer).*

1. **Cộng dồn điện năng đã sử dụng**

* cumulative\_energy\_used\_kWh += energy\_used\_kWh\_this\_tick

1. **Tính điện năng còn lại**

* energy\_remaining\_kWh

= battery\_capacity\_kWh × () - cumulative\_energy\_used\_kWh

1. **Tính phần trăm pin**

* battery = 100

Tóm lại:

Pin được tính dựa trên tổng dung lượng ban đầu trừ đi điện năng đã sử dụng, sau đó quy đổi ra phần trăm so với dung lượng pin gốc.

## 5.1.2. Tính toán quãng đường còn lại

Tính toán quãng đường còn lại (DTE) là một trong những thách thức lớn nhất đối với xe điện, vì nó ảnh hưởng trực tiếp đến “nỗi lo hết pin” (range anxiety) của người lái. Một thuật toán DTE tốt cần đưa ra con số ổn định và đáng tin cậy, thay vì thay đổi đột ngột theo điều kiện lái tức thời.

Cách làm đơn giản nhất — lấy năng lượng còn lại chia cho mức tiêu thụ hiện tại — tuy dễ triển khai nhưng thường tạo ra trải nghiệm người dùng kém, do giá trị DTE dao động mạnh khi xe tăng/giảm tốc, thay đổi đường hoặc bật/tắt điều hòa.

Trong thực tế, các hệ thống hiện đại sử dụng thuật toán pha trộn (blending), kết hợp cả dữ liệu lịch sử và dữ liệu hiện tại. Nguyên tắc cơ bản:

* Thói quen lái trong quá khứ
* Điều kiện chuyến đi hiện tại.

Cách tiếp cận:

* Dữ liệu dài hạn (ví dụ: mức tiêu thụ trung bình trong 300 km gần nhất) đại diện cho phong cách lái và môi trường quen thuộc của người dùng.
* Dữ liệu ngắn hạn (ví dụ: mức tiêu thụ trung bình trong 10–20 km gần nhất) phản ánh điều kiện của chuyến đi hiện tại, như chạy cao tốc hay trong phố.

Khi mới bắt đầu một chuyến đi, DTE sẽ chủ yếu dựa vào dữ liệu dài hạn. Khi quãng đường tăng lên, trọng số dữ liệu ngắn hạn tăng dần, giúp DTE chính xác hơn trong bối cảnh hiện tại.

Các biến sử dụng:

* energy\_remaining\_kWh: Năng lượng còn lại trong pin (kWh), tính từ SoC (%).
* avg\_consumption\_long\_term: Mức tiêu thụ trung bình dài hạn (kWh/100km), lưu trong bộ nhớ non-volatile.
* avg\_consumption\_short\_term: Mức tiêu thụ trung bình ngắn hạn (kWh/100km), tính và reset mỗi chuyến đi.
* trip\_distance\_km: Quãng đường đã đi trong chuyến hiện tại.

Thuật toán:

1. Tính hệ số pha trộn (tăng tuyến tính theo quãng đường, bão hòa ở 1.0 sau 50 km):

blending\_factor =

1. Pha trộn mức tiêu thụ trung bình:

final\_avg\_consumption = (blending\_factor avg\_consumption\_short\_term) + ((1 - blending\_factor) × avg\_consumption\_long\_term)

1. Giới hạn giá trị tiêu thụ để tránh chia cho 0 hoặc tạo DTE quá lớn:

final\_avg\_consumption = max(final\_avg\_consumption, 0.1) (đơn vị kWh/100km)

1. Tính DTE (đơn vị km):

DTE\_km = × 100

# 6. Kế hoạch tích hợp và kiểm thử

Để đảm bảo tính tin cậy, hiệu năng và khả năng mở rộng của hệ thống, dự án áp dụng một kế hoạch kiểm thử toàn diện, tập trung vào các mục tiêu kỹ thuật và chất lượng đã đề ra ở phần 1.2.. Kiểm thử được thực hiện ở nhiều cấp độ, từ đơn vị đến hệ thống, với trọng tâm là đánh giá performance – cụ thể là latency giữa input và output. Điều này đặc biệt quan trọng vì hệ thống nhấn mạnh tính scale up (khả năng mở rộng số lượng ECU nodes và traffic dữ liệu) và configurable dễ dàng (qua file JSON). Dựa trên kết quả latency, chúng ta có thể tối ưu hóa hệ thống, ví dụ: điều chỉnh buffer size, ưu tiên gói tin CAN, hoặc cải thiện xử lý DMA trên STM32.

Kiểm thử sử dụng các công cụ sau:

* **Phần cứng:** Oscilloscope (đo thời gian từ input vật lý đến output LED), USB-to-CAN analyzer (như CANalyzer hoặc PCAN-View) để theo dõi traffic CAN.
* **Phần mềm:** dùng qDebug() cho unit test (C++ trên Raspberry Pi), STM32CubeIDE cho firmware STM32, và debug mode trong Qt Creator cho integration với QML.
* **Môi trường:** Thực hiện trên prototype thực tế (Raspberry Pi 5 + STM32 nodes), với dữ liệu mô phỏng (tăng traffic bằng script gửi CAN frame định kỳ).

Các kiểm thử được thực hiện theo nguyên tắc ISO 26262 (ASIL B cho các thành phần quan trọng như hiển thị tốc độ), bao gồm kiểm tra failure modes (ví dụ: mất gói tin CAN) và coverage code >80%.

## 6.1. Kiểm thử đơn vị (Unit Test)

Kiểm thử tập trung vào các module riêng lẻ để đảm bảo tính đúng đắn và configurable.

### 6.1.1. Test case cho JSON Config Parser (trên VCU)

* **Mô tả:** Kiểm tra khả năng đọc và ánh xạ I/O từ file JSON mà không cần rebuild code.
* **Input:** File JSON mẫu với thay đổi (ví dụ: chuyển "speed" từ analog\_input 0 sang 1).
* **Expected Output:** Ánh xạ đúng trong bộ nhớ (kiểm tra bằng assert trên map<string, int>).
* **Test Steps:**

1. Thêm QElapsedTimer vào để lưu lại thời gian và in ra console thời gian sau khi file cấu hình được load xong

QElapsedTimer timer;

timer.start();

// Code to load JSON file is putted here

qDebug() << "Loaded IO configuration in" << static\_cast<float>(timer.nsecsElapsed())/1000 << "us";

1. Load JSON với cấu hình mặc định.
2. Thay đổi file JSON (thêm/exclude input/output).
3. Reload parser và assert mapping mới.

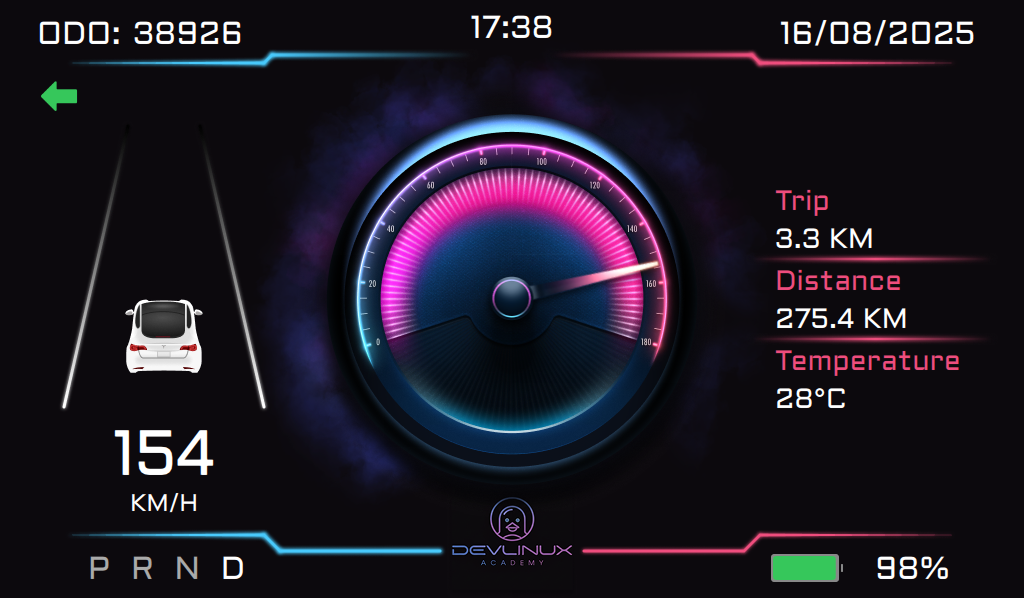
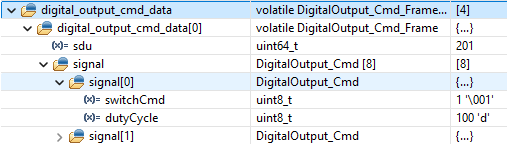
* **Kết quả mong đợi:** Pass nếu parser xử lý được 100% thay đổi mà không crash. Thời gian parse < 1ms.
* **Kết quả thực tế:** Luôn luôn load được config và thời gian trong khoảng từ 40 micro giây đến 120 micro giây

Hình 11. Kết quả kiểm thử khi thay đổi JSON

### 6.1.2. Test case cho CAN Frame Packing/Unpacking (trên ECU và VCU)

* **Mô tả:** Kiểm tra đóng gói dữ liệu Output Command từ VCU theo định dạng ở 4.1.3.
* **Input:** Giá trị mẫu của xi nhan trái (ví dụ: Switch Command = 1, Duty Cycle = 100).
* **Expected Output:** CAN frame đúng (kiểm tra byte-by-byte).
* **Test Steps:** Sử dụng mock CAN library để simulate send/receive.
* **Kết quả mong đợi**: 100% frame đúng, phân tách dữ liệu đúng như yêu cầu.
* **Kết quả thực tế:** 100% frame đúng, phân tách dữ liệu đúng như yêu cầu.



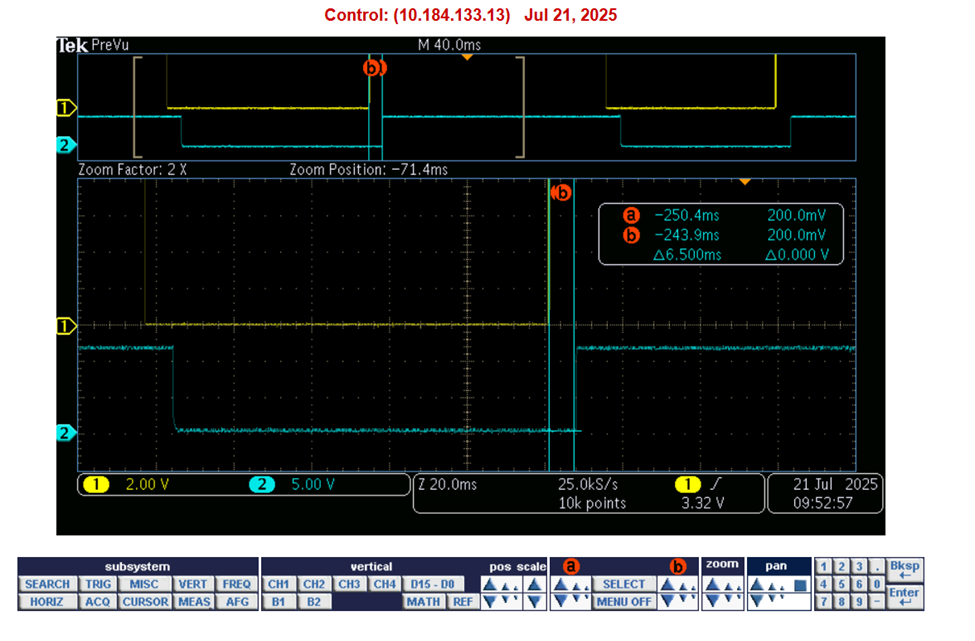
Hình 12. Kết quả kiểm thử khi gạt xi nhan trái

## 6.2. Kiểm Thử Hệ Thống (System Test)

System test đánh giá toàn bộ pipeline từ input vật lý (chiết áp, công tắc) đến output (HMI update, LED), bao quát các tương tác tích hợp (ECU ↔ VCU ↔ HMI). Test tập trung vào latency end-to-end. Các kết quả từ kiểm thử tích hợp được tích hợp để phân tích bottleneck.

## 6.2.1. Latency End-to-End

* **Mục đích:** Đo thời gian từ input vật lý đến output (HMI update hoặc LED bật), đảm bảo latency thấp, đáp ứng yêu cầu HMI thời gian thực (ASIL B).
* **Test Case:**
  + Input: Gạt công tắc xi nhan trái.
  + Output mong đợi: HMI hiển thị icon xi nhan trái nhấp nháy và LED chớp tắt với tần số 1Hz.
* **Test Steps:**
  + Dùng oscilloscope đo từ GPIO change (input) đến UI refresh/LED on (output).
  + Log timestamp tại các thời điểm
  + Lặp lại với Busload khác nhau
* **Với Busload = 0 ~ 5%:** Độ trễ trung bình của Input (màu vàng) đến Output (màu xanh) là khoảng 6,5ms



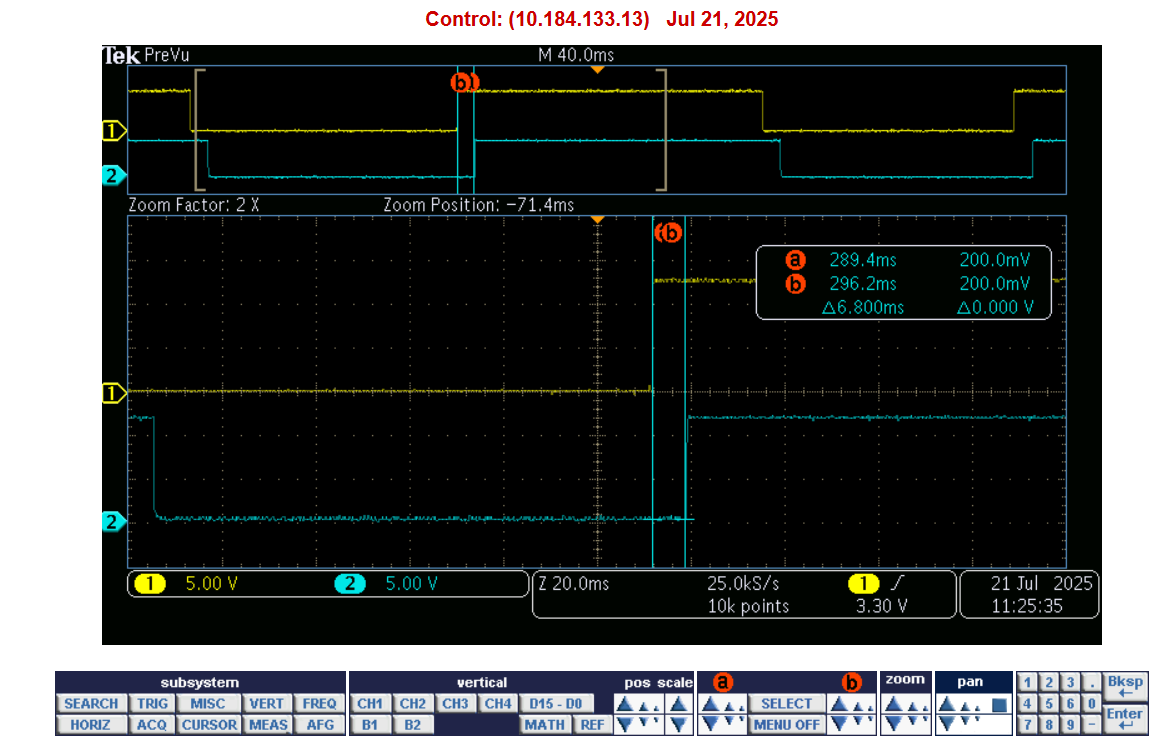
Hình 13. Kết quả đo latency với Busload 0 ~ 5%



Hình 14. Kết quả đo latency trung bình với Busload 0 ~ 5%

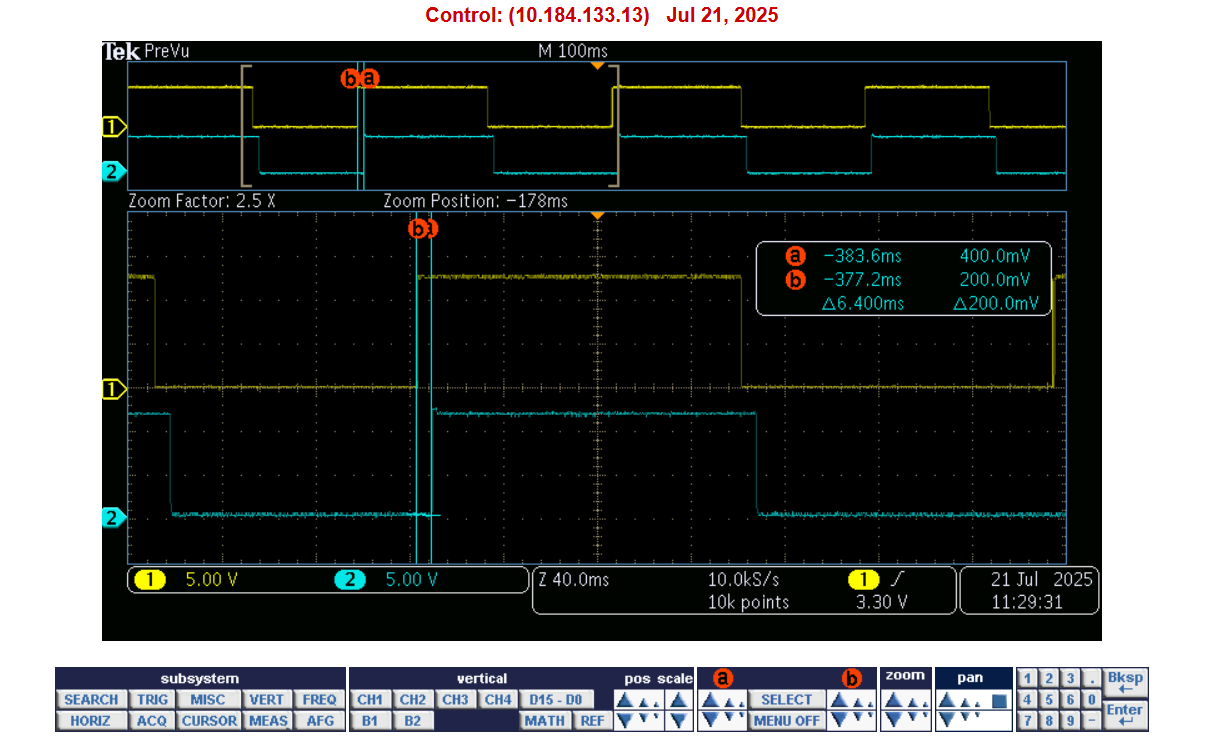
Giá trị ghi lại được nằm trong khoảng từ 6,1ms đến 7,4ms.

* **Với Busload = 20 ~ 40%:** Độ trễ trung bình của Input (màu vàng) đến Output (màu xanh) là khoảng 6,8ms



Hình 15. Kết quả đo latency với Busload 20 ~ 40%

* **Với Busload = ~ 70%:** Độ trễ trung bình của Input (màu vàng) đến Output (màu xanh) là khoảng 6,8ms



Hình 16. Kết quả đo latency với Busload ~ 70%

# 7. Kết luận và hướng phát triển

## 7.1. Kết luận

Dự án đã hoàn thành mục tiêu thiết kế và triển khai một hệ thống mô phỏng kiến trúc zonal trên nền tảng xe điện, với sự tích hợp giữa cụm đồng hồ số (IPC) và bộ điều khiển trung tâm (VCU). Hệ thống sử dụng Raspberry Pi cho VCU, STM32 làm các ECU node, Qt/QML cho HMI, và giao tiếp qua CAN.

Các kết quả chính:

* **Latency**: Thời gian phản hồi end-to-end trung bình ~ 7ms, dưới mức mục tiêu 10 ms. Hệ thống đáp ứng yêu cầu real-time của HMI (tốc độ, pin, đèn báo) và phù hợp tiêu chuẩn ISO 26262 (ASIL B).
* **Khả năng mở rộng**: Hoạt động ổn định với nhiều ECU node và I/O. Kiến trúc cho phép tăng node mà không cần thay đổi thiết kế, với bit error rate <0.1% ở mức tải cao.
* **Tính cấu hình**: File JSON cho phép cập nhật lại cấu hình I/O, thời gian tải cấu hình nhanh, ánh xạ đúng 100%, hỗ trợ thêm phần cứng mới mà không cần chỉnh sửa mã nguồn.
* **Tính thực tiễn**: Prototype đã được chạy thử nghiệm thực tế, có hình ảnh/video demo, mã nguồn public. Kết quả cho thấy hệ thống khả thi và có thể áp dụng trong nghiên cứu phát triển tiếp theo.

Một số điểm đáng chú ý: giao tiếp CAN là bottleneck chính (chiếm ~40% latency), và việc tách phần cấu hình ra file JSON giúp giảm đáng kể effort khi thay đổi phần cứng.

## 7.2. Hướng phát triển

Các cải tiến khả thi trong giai đoạn tiếp theo:

* **Giảm latency**: Sử dụng CANFD để đơn giản hóa phần CAN ID và tối ưu về tốc độ cũng như lựng dữ liệu trên 1 khung CAN, bổ sung cơ chế ưu tiên frame quan trọng (speed, warning), tối ưu DMA để giảm thời gian đọc ADC xuống.
* **Mở rộng hệ thống**: thêm auto-discovery cho ECU node, thử nghiệm với 32 node (CAN ID 29-bit), stress test với 10k frames/s. Chuyển sang CAN-FD hoặc Ethernet khi số node vượt 16.
* **Tăng tính cấu hình**: phát triển giao diện web để chỉnh JSON trực quan, thêm validation tự động (schema check, checksum).
* **Tích hợp tính năng nâng cao**: thử nghiệm các chức năng ADAS đơn giản (lane departure warning), kết nối Bluetooth/Wi-Fi để hiển thị thông báo điện thoại, dùng machine learning để dự đoán DTE dựa trên dữ liệu lái xe.
* **Kiểm thử và chứng nhận**: chạy test môi trường (nhiệt độ, rung động) và tiến hành quy trình chứng nhận ISO 26262 (ASIL B/C) cho các module quan trọng.

Ngoài các cải tiến về hiệu năng và mở rộng, một hướng quan trọng là **cơ chế cập nhật phần mềm và cấu hình** cho VCU:

* **Cập nhật phần mềm qua OVA**: phát triển tính năng cho phép update toàn bộ image VCU dưới dạng OVA (Open Virtual Appliance). Cách này đảm bảo đồng bộ môi trường, giảm rủi ro lệch phiên bản, và thuận tiện khi triển khai ở nhiều hệ thống thử nghiệm.
* **Cập nhật file cấu hình động**: bổ sung khả năng tải và áp dụng file JSON mới thông qua giao diện (web/USB/Ethernet). Hệ thống có thể reload cấu hình mà không cần reboot VCU, giúp rút ngắn thời gian tích hợp và thử nghiệm.
* **Cơ chế rollback**: khi update thất bại, VCU cần có khả năng tự động quay lại phiên bản ổn định trước đó để tránh downtime.
* **Bảo mật update**: áp dụng chữ ký số hoặc checksum để đảm bảo chỉ có gói phần mềm/cấu hình hợp lệ mới được chấp nhận.

Cơ chế update này sẽ giúp VCU trở thành nền tảng linh hoạt, dễ bảo trì và phù hợp với quy trình OTA (Over-The-Air) trong ngành ô tô. Những bổ sung sẽ giúp dự án này tạo nền tảng để phát triển thành hệ thống điều khiển tổng thể cho EV hoặc xe tự hành trong các nghiên cứu sau.

# Tài Liệu Tham Khảo

[1] International Organization for Standardization, *ISO 26262: Road vehicles – Functional safety*. Geneva, Switzerland: ISO, 2018.

[2] International Organization for Standardization, *ISO 2575: Road vehicles – Symbols for controls, indicators and tell-tales*. Geneva, Switzerland: ISO, 2024.

[3] United Nations Economic Commission for Europe, *UN Regulation No. 121: Uniform provisions concerning the approval of vehicles with regard to the location and identification of hand controls, tell-tales and indicators*. UNECE, 2018.

[4] Broadcom Inc., *BCM2712 Datasheet (Raspberry Pi 5 SoC)*, 2023.

[5] STMicroelectronics, *STM32F103C8T6 Datasheet*, 2020.

[6] Texas Instruments, *SN65HVD230 CAN Bus Transceiver Datasheet*, 2018.

[7] The Qt Company, *Qt Documentation*. [Online]. Available: <https://doc.qt.io>

[8] Raspberry Pi Foundation, *Raspberry Pi System Reference Manual*. Cambridge, UK: Raspberry Pi Trading, 2023.

[9] STMicroelectronics, *RM0008 Reference Manual: STM32F103x8/B microcontrollers*, Rev. 21, 2018.

[10] A. Amditis *et al.*, “Towards the Automotive HMI of the Future: Overview of the AIDE-Integrated Project Results,” *IEEE Trans. Intell. Transp. Syst.*, vol. 11, no. 3, pp. 567–578, Sep. 2010.

[11] The Yocto Project, *Welcome to the Yocto Project Documentation — Version 5.2.999*. [Online]. Available: <https://docs.yoctoproject.org>

[12] OpenEmbedded, *OpenEmbedded Project*. [Online]. Available: <https://www.openembedded.org>

[13] L. Anavi, *Building a Basic GNU/Linux Distribution for Raspberry Pi 5 with the Yocto Project LTS Release Scarthgap*, Mobile & Embedded Conference, 2023. Avaiblable: <https://anavi.org/article/298/>

[14] The Linux Kernel Organization, *SocketCAN – Controller Area Network Documentation*. [Online]. Available: <https://docs.kernel.org/networking/can.html>

[15] CSDN Blog, *Linux SocketCAN 编程 (C++，启用多线程接收)*. [Online]. Available: <https://blog.csdn.net/Flag_ing/article/details/126387114>

[16] Molex, “Zonal Architecture vs. Domain Architecture,” Technical Article, 2022. Available: <https://www.molex.com/en-us/blog/zonal-architecture-vs-domain-architecture-modular-automotive-infrastructure-face-off>

[17] ControllersTech, *CAN Protocol in STM32 – Tutorial*. [Online]. Available: <https://controllerstech.com/can-protocol-in-stm32/>

[18] STMicroelectronics, *FreeRTOS on STM32 v2 – CMSIS API v2.0 Documentation*. [Online]. Available: <https://www.st.com/content/st_com/en/support/learning/stm32-education/stm32-moocs/freertos-common-microcontroller-software-interface-standard-osv2.html>

# Phụ lục: Bảng thuật ngữ

* **ADC (Analog-to-Digital Converter):** Bộ chuyển đổi tín hiệu tương tự sang kỹ thuật số, được sử dụng để đọc giá trị từ cảm biến như biến trở mô phỏng tốc độ xe.
* **ASIL (Automotive Safety Integrity Level):** Mức độ toàn vẹn an toàn ô tô, dùng để phân loại rủi ro theo tiêu chuẩn ISO 26262, đặc biệt cho các chức năng quan trọng như hiển thị tốc độ và cảnh báo.
* **BOM (Bill of Materials):** Danh sách vật tư, liệt kê các linh kiện phần cứng cần thiết cho dự án, như Raspberry Pi, STM32 và cảm biến.
* **CAN (Controller Area Network):** Giao thức mạng truyền thông nối tiếp, dùng để trao đổi dữ liệu giữa VCU và các ECU nodes trong hệ thống.
* **DMA (Direct Memory Access):** Truy cập bộ nhớ trực tiếp, giúp xử lý dữ liệu nhanh chóng mà không cần CPU can thiệp, ví dụ đọc ADC trên STM32.
* **Domain Architecture:** Kiến trúc miền chức năng, tổ chức hệ thống điện tử ô tô theo từng miền như truyền động, an toàn, giải trí.
* **DTE (Distance-to-Empty):** Khoảng cách ước tính xe có thể đi được với lượng năng lượng còn lại, được tính toán dựa trên dữ liệu tiêu thụ điện và lịch sử lái xe.
* **ECU (Electronic Control Unit):** Bộ điều khiển điện tử, là các nút mô phỏng bằng STM32 để quản lý input/output như cảm biến và đèn LED.
* **EV (Electric Vehicle):** Xe điện, là bối cảnh chính của dự án với sự chuyển dịch từ động cơ đốt trong sang hệ thống điện hóa.
* **FIFO (First In First Out):** Hàng đợi dữ liệu, được sử dụng để sắp xếp khung CAN trước khi truyền.
* **GPIO (General Purpose Input/Output):** Chân vào/ra đa năng trên vi điều khiển, dùng để kết nối cảm biến, nút bấm và đèn báo.
* **HMI (Human-Machine Interface):** Giao diện người-máy, bao gồm cụm đồng hồ kỹ thuật số và tương tác với người lái qua màn hình IPC.
* **IPC (Instrument Panel Cluster):** Cụm đồng hồ kỹ thuật số trên bảng điều khiển, hiển thị thông tin như tốc độ, pin và cảnh báo.
* **ISO 26262:** Tiêu chuẩn quốc tế về an toàn chức năng cho xe hơi, được áp dụng để thiết kế HMI an toàn và đáng tin cậy.
* **ISO 2575:** Tiêu chuẩn quốc tế về biểu tượng cho các nút điều khiển, đèn báo và chỉ thị trên xe, đảm bảo tính quen thuộc và dễ nhận biết.
* **JSON (JavaScript Object Notation):** Định dạng dữ liệu dùng để cấu hình I/O động, cho phép thay đổi ánh xạ input/output mà không cần sửa mã nguồn.
* **Odometer (ODO):** Đồng hồ đo tổng quãng đường đã đi của xe, được hiển thị trên giao diện HMI.
* **PWM (Pulse Width Modulation):** Điều chế độ rộng xung, dùng để điều khiển output như đèn LED hoặc còi với mức duty cycle.
* **QML (Qt Modeling Language):** Ngôn ngữ khai báo dựa trên JavaScript trong framework Qt, dùng để thiết kế giao diện đồ họa cho HMI.
* **SDV (Software Defined Vehicle):** Xe được định nghĩa bởi phần mềm, nhấn mạnh khả năng cập nhật và cấu hình linh hoạt qua phần mềm.
* **SoC (State of Charge):** Trạng thái sạc của pin, biểu thị bằng phần trăm (%), dùng để tính toán pin còn lại và DTE.
* **SocketCAN:** Giao diện lập trình socket trong Linux để giao tiếp với CAN bus, được sử dụng trên VCU (Raspberry Pi).
* **VCU (Vehicle Control Unit):** Bộ điều khiển xe trung tâm, quản lý trạng thái xe, xử lý dữ liệu từ ECU và cập nhật HMI.
* **Zonal Architecture:** Kiến trúc zonal, tổ chức hệ thống điện tử ô tô theo vùng địa lý, giúp giảm phức tạp dây dẫn và dễ nâng cấp.