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

[1. Tổng quan dự án 4](#_Toc204205274)

[1.1. Giới thiệu 4](#_Toc204205275)

[1.2. Mục tiêu Dự án (Project Objectives) 4](#_Toc204205276)

[1.2.1. Mục tiêu Kỹ thuật 4](#_Toc204205277)

[1.2.2. Mục tiêu Chất lượng 4](#_Toc204205278)

[1.3. Phạm vi Dự án (Project Scope) 5](#_Toc204205279)

[1.3.1. Trong phạm vi (In-Scope) 5](#_Toc204205280)

[1.3.2. Ngoài phạm vi (Out-of-Scope) 5](#_Toc204205281)

[2. Tổng quan kiến trúc hệ thống 6](#_Toc204205282)

[2.1. Kiến trúc Zonal (Zonal Architecture) 6](#_Toc204205283)

[2.2. Sơ đồ Khối Hệ thống (System Block Diagram) 6](#_Toc204205284)

[2.3. Luồng Dữ liệu và Tương tác 8](#_Toc204205285)

[3. Thiết kế phần cứng và kết nối 8](#_Toc204205286)

[3.1. Danh sách vật tư (Bill of Materials - BOM) 8](#_Toc204205287)

[3.2. Sơ đồ Kết nối 9](#_Toc204205288)

[4. Quy trình phát triển phần mềm 9](#_Toc204205289)

[4.1. Giao tiếp CAN giữa VCU và ECU nodes 9](#_Toc204205290)

[4.1.1. Truyền tin giữa các thành phần 9](#_Toc204205291)

[4.1.2. Kiến trúc linh hoạt dựa trên cấu hình I/O động 10](#_Toc204205292)

[4.1.3. Định nghĩa bảng tin CAN 10](#_Toc204205293)

[4.1.3.1. Định dạng CAN ID 11](#_Toc204205294)

[4.1.3.2. Khung dữ liệu Digital Output 12](#_Toc204205295)

[4.1.3.3. Khung dữ liệu Digital Input 13](#_Toc204205296)

[4.2. Thiết kế phần mềm VCU 14](#_Toc204205297)

[4.2.1. Sử dụng USB to CAN 14](#_Toc204205298)

[4.2.1.1. Test nhanh giao tiếp CAN 14](#_Toc204205299)

[4.2.1.2. Sử dụng SocketCAN để viết firmware cho VCU 15](#_Toc204205300)

[4.2.2. Kết nối Qt/C++ với QML 18](#_Toc204205301)

[4.2.3. State machine 18](#_Toc204205302)

[4.2. Thiết kế giao diện IPC 18](#_Toc204205303)

[4.2.1. Màn hình và giao diện QML 18](#_Toc204205304)

[4.2.2. Cấu trúc file project Qt 19](#_Toc204205305)

[4.2.3. Cách binding dữ liệu 19](#_Toc204205306)

[4.3. Firmware cho STM32 (mô phỏng lại các ECU nodes) 19](#_Toc204205307)

[4.3.1. Cấu hình và giao tiếp CAN cơ bản 19](#_Toc204205308)

[4.3.2. Firmware cho ECU nodes (mô phỏng bằng STM32) 24](#_Toc204205309)

[5. Thuật toán xử lý 24](#_Toc204205310)

[5.1.1. Tính tốc độ (encoder) 25](#_Toc204205311)

[5.1.2. Quản lý chế độ lái 25](#_Toc204205312)

[5.1.3. Thuật toán tính năng chẩn đoán cơ bản 25](#_Toc204205313)

[6. Kết luận và hướng phát triển 25](#_Toc204205314)

[Tài Liệu Tham Khảo 25](#_Toc204205315)

[Phụ lục: Bảng thuật ngữ 25](#_Toc204205316)

# 

# 

# 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 (dưới 2 giây) 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, điển hình là việc thay đổi giao diện theo chế độ lái (Eco, Normal, Sport).

## 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 người dùng đầy đủ chức năng với các màn hình riêng biệt cho các chế độ hoạt động chính của xe: Lái xe (với các biến thể Eco/Normal/Sport) và Sạc pin.
* **Chức năng VCU:** Mô phỏng các chức năng VCU cốt lõi, bao gồm: quản lý máy trạng thái xe, xử lý và lưu trữ mã lỗi chẩn đoán (DTC) cơ bản, và quản lý các chế độ lái.
* **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.

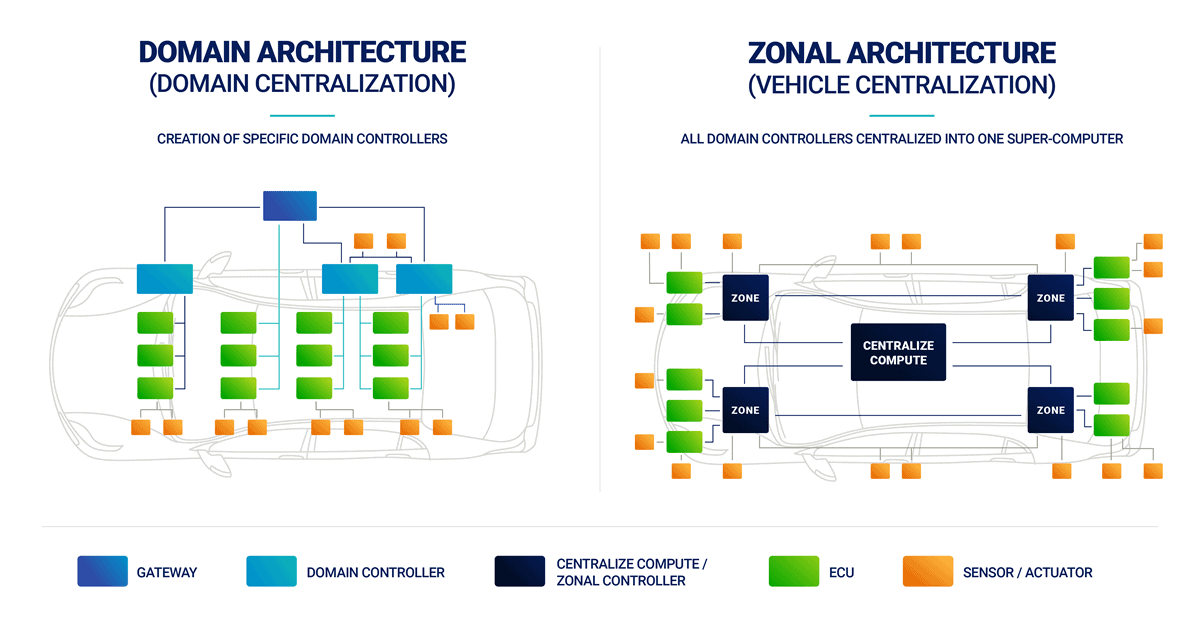
### 1.3.2. Ngoài phạm vi (Out-of-Scope)

* **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 Vật lý:** Dự án chỉ mô phỏng các tín hiệu từ các hệ thống cơ khí và điện tử công suất (phanh, động cơ, pin thật). Không phát triển các hệ thống vật lý này.

# 

# 2. Tổng quan kiến trúc hệ thống

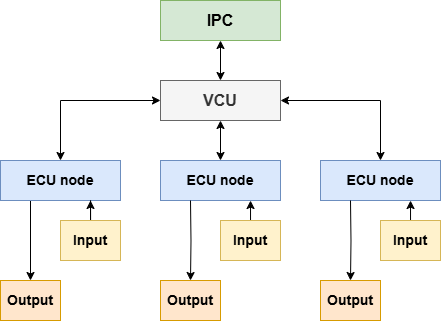
## 2.1. Kiến trúc Zonal (Zonal Architecture)



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)



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: người lái tăng tốc.

1. **Tương tác Vật lý:** Người dùng xoay núm encoder, mô phỏng việc bánh xe quay nhanh hơn.
2. **Thu thập Dữ liệu (ECU node):** Firmware trên STM32 đếm số xung encoder trong một khoảng thời gian cố định (ví dụ: 100ms).
3. **Xử lý Sơ bộ (ECU Nút):** Dựa trên số xung đếm được và các thông số đã biết (số xung/vòng, chu vi bánh xe), STM32 tính toán giá trị tốc độ tạm thời.
4. **Đóng gói Dữ liệu (ECU Nút):** Giá trị tốc độ này được đóng gói vào một khung tin CAN đã định.
5. **Truyền Dữ liệu (Bus):** Khung tin được gửi đi từ của STM32 và đến Raspberry Pi.
6. **Nhận và Giải mã (VCU):** Raspberry Pi nhận được một chuỗi byte. Nếu hợp lệ, nó trích xuất dữ liệu từ tin nhắn.
7. **Cập nhật Mô hình Dữ liệu (VCU):** SerialManager thông báo cho lớp VehicleModel rằng có dữ liệu tốc độ mới. VehicleModel cập nhật thuộc tính speed của nó và phát ra một tín hiệu (signal) trong môi trường Qt.
8. **Cập nhật Giao diện (HMI):** Giao diện QML, vốn đã "binding" (liên kết) với thuộc tính speed của VehicleModel, sẽ tự động nhận được tín hiệu thay đổi. Engine của Qt Quick sẽ vẽ lại chỉ thành phần bị ảnh hưởng (đồng hồ tốc độ) với giá trị mới, đảm bảo hiệu suất tối ưu.

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 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. |

# 

# 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 |

### 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

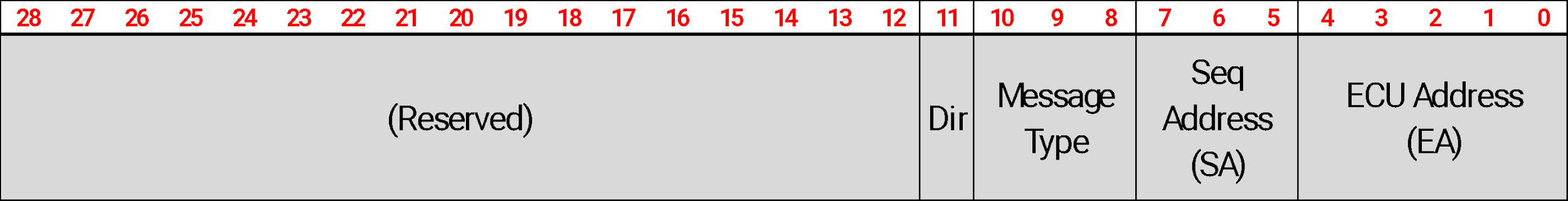
Dựa theo kiến trúc đã trình bày, việc trao đổi dữ liệu giữa các ECU node và VCU cần được thực hiện một cách thống nhất và linh hoạt. Vì vậy, các bảng tin CAN (CAN frames) sẽ tuân theo một cấu trúc cố định, nhưng có nội dung hoàn toàn cấu hình được thông qua file mà VCU đọc.

#### 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**

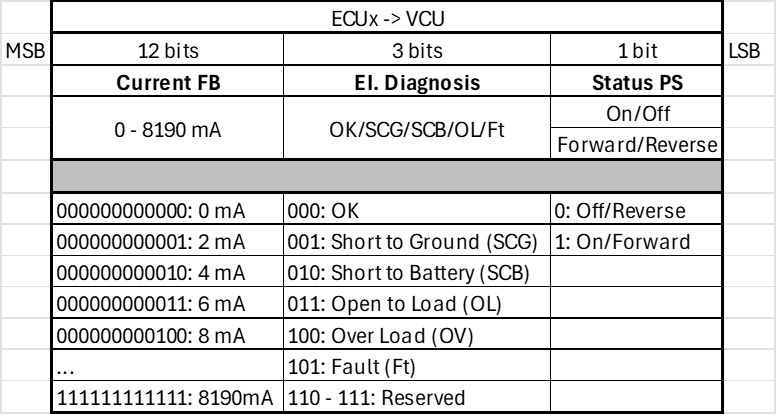
VCU truyền dữ liệu gồm các tín hiệu điều khiển 1 output cụ thể được kết nối với ECU cụ thể trong hệ thống. Hình sau thể hiện tín hiệu để điều khiển 1 Digital Output cụ thể.



* 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.

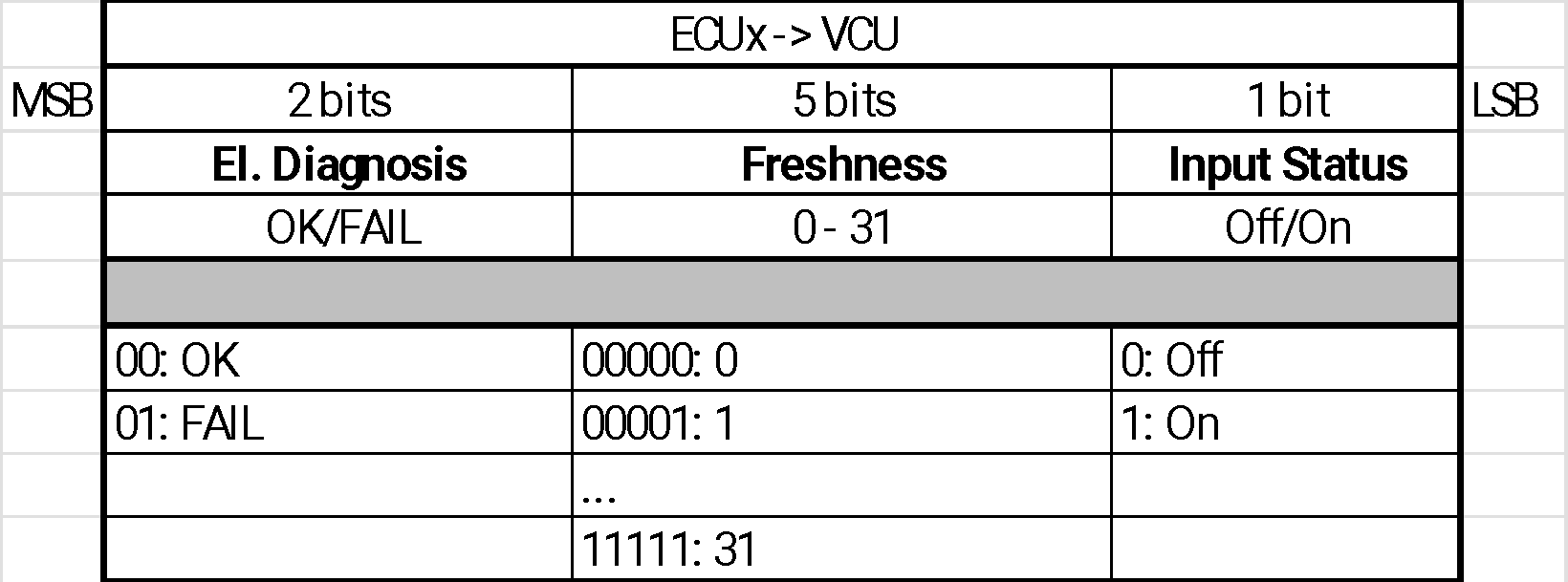


* Giải thích các trường trong 1 tín hiệu của hình trên:
  + Current FB [12 bits]: Dòng điện phản hồi từ ECU, đơn vị mA (0 – 8190 mA).
  + El. Diagnosis [3 bits]: Trạng thái chẩn đoán điện như OK, ngắn mạch, hở mạch...
  + Status PS [1 bit]: Trạng thái logic hoặc điều khiển: On/Off hoặc Forward/Reverse.
* 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.



* 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**

## 4.2. 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. Test nhanh giao tiếp CAN

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

**Bước 1**: Cài gói can-utils

Cài thông qua apt:

**sudo apt install can-utils**

Thêm gói trong Yocto project:

**IMAGE\_INSTALL:append = “ can-utils”**

**Bước 2**: Bật CAN interface

Ví dụ bật can0 với tốc độ 666kbps:

**sudo ip link set can0 up type can bitrate 666666**

Kiểm tra trạng thái:

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

**Bước 3**: Gửi và nhận khung CAN

Để nhận CAN frame:

**candump can0**

Để gửi dữ liệu:

**cansend can0 123#1122**

**cansend can0 1ABCDEEF#010203**

* 123: CAN ID (11-bit)
* #1122: 2 byte dữ liệu hex

**Bước 4**: 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 666666 loopback on**

**candump can0 & # Terminal 1**

**cansend can0 123#AABBCCDD # Terminal 2**

#### 4.2.1.2. Sử dụng SocketCAN để viết firmware cho VCU

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;**

**}**

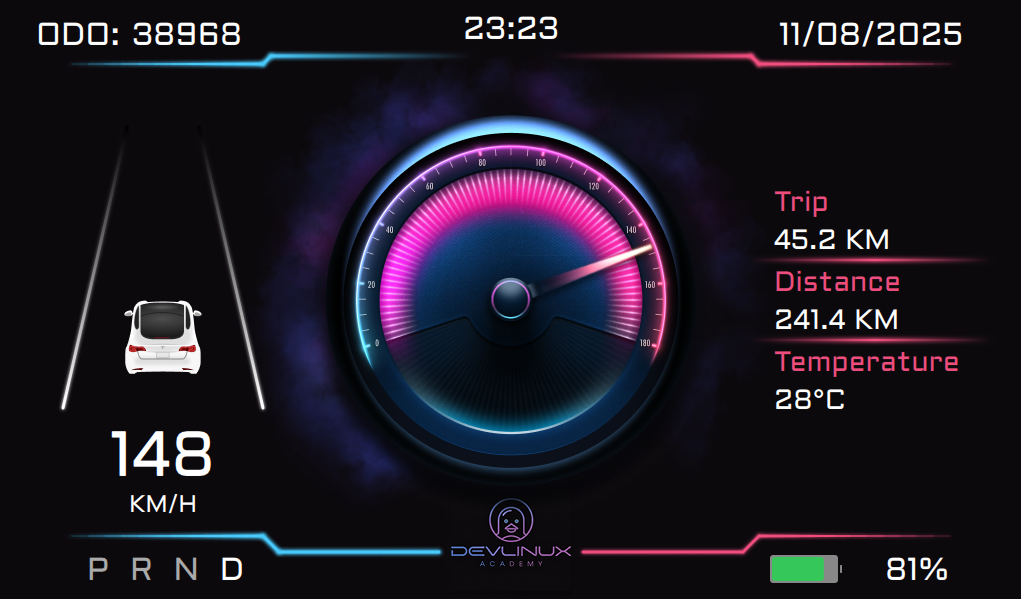
### 4.2.2. Kết nối Qt/C++ với QML

### 4.2.3. State machine

## 4.2. Thiết kế giao diện IPC

### 4.2.1. Màn hình và giao diện QML

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



Hình ảnh giao diện HMI

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>

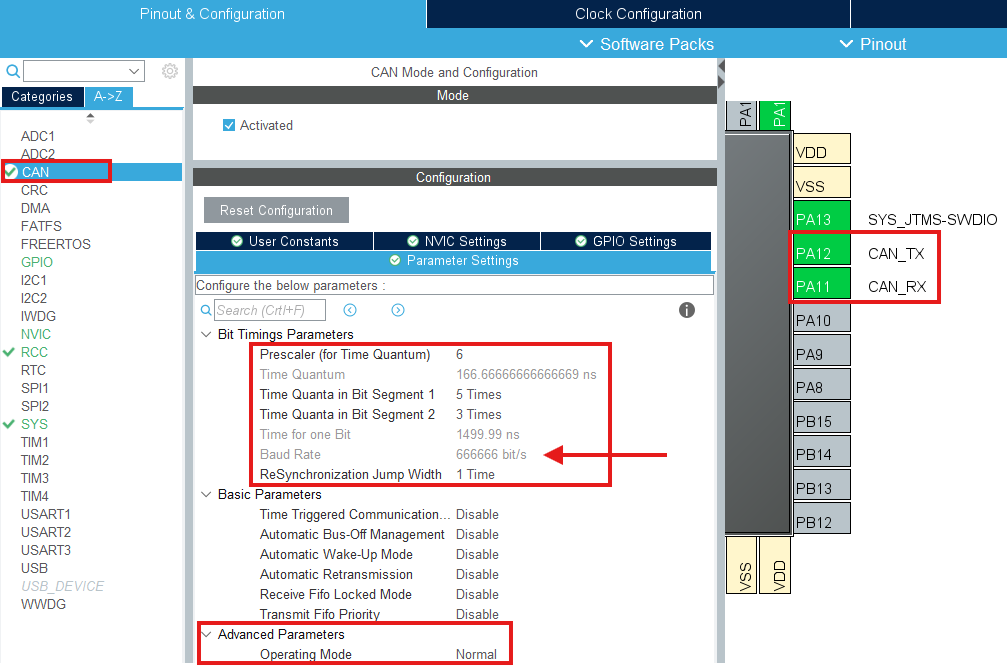
### 4.2.2. Cấu trúc file project Qt

### 4.2.3. Cách binding dữ liệu

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

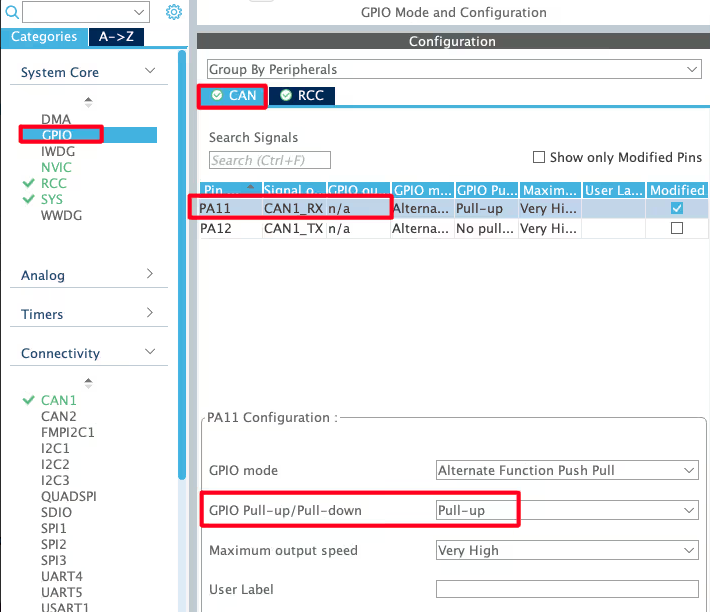
### 4.3.1. Cấu hình và giao tiếp CAN cơ bản

Cấu hình CubeMX được hiển thị như bên dưới.



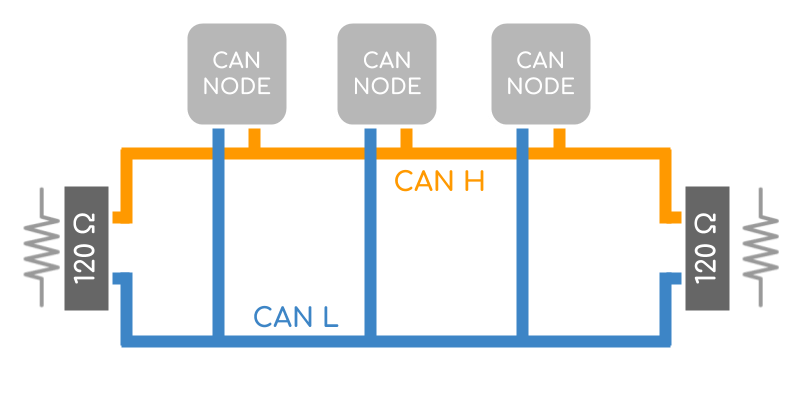
* Tốc độ truyền (BAUD RATE) được đặt là 666666 bps. Bạn có thể thử các tổ hợp khác nhau cho Prescalar và Time Quanta để đạt được giá trị này.
* Chế độ hoạt động là Chế độ Bình thường (NORMAL Mode).
* Chân PA11 và PA12 được cấu hình lần lượt là CAN\_RX và CAN\_TX.

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



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



**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.

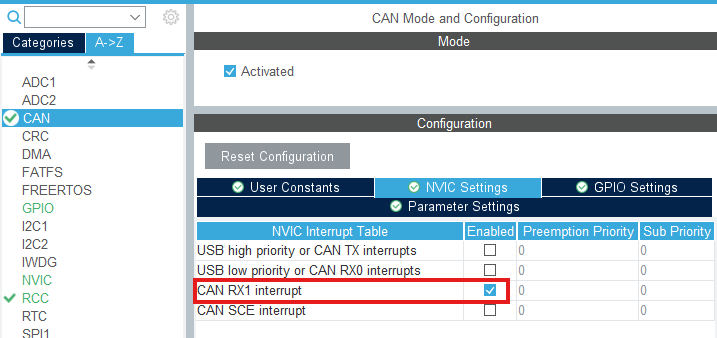


Bạn có thể thấy dữ liệu trên CAN bus như hình trên.

* 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



**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.

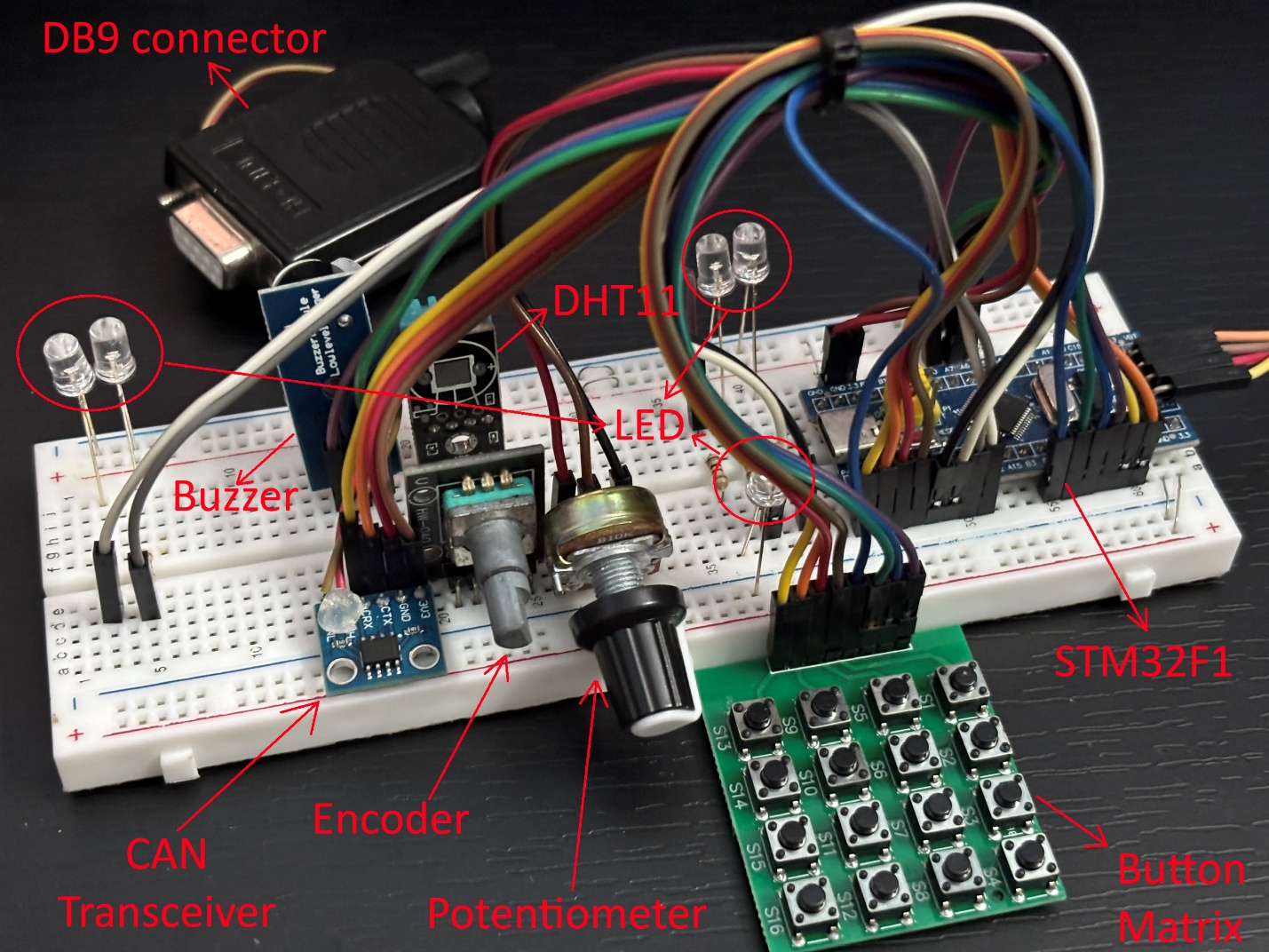
### 4.3.2. Firmware cho ECU nodes (mô phỏng bằng STM32)

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.

Đây là hình ảnh setup:



# 5. Thuật toán xử lý

## 5.1.1. Tính tốc độ (encoder)

## 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 = min(trip\_distance\_km / 50.0, 1.0)

1. Pha trộn mức tiêu thụ trung bình:

final\_avg\_consumption = (blending\_factor \* avg\_consumption\_short\_term) + ((1.0 - 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) # kWh/100km

1. Tính DTE (đơn vị km):

DTE\_km = (energy\_remaining\_kWh / final\_avg\_consumption) \* 100.0

## 5.1.3. Thuật toán tính năng chẩn đoán cơ bản

# 6. Kết luận và hướng phát triển

# Tài Liệu Tham Khảo

[The Yocto Project](https://www.yoctoproject.org/)

[Welcome to the Yocto Project Documentation — The Yocto Project ® 5.2.999 documentation](https://docs.yoctoproject.org/)

[Openembedded.org](https://www.openembedded.org/wiki/Main_Page)

[Leon Anavi - Mobile & Embedded - Building a Basic GNU Linux Distribution for Raspberry Pi 5 with the Yocto Project LTS Release Scarthgap](https://anavi.org/article/298/)

[SocketCAN - Controller Area Network — The Linux Kernel documentation](https://docs.kernel.org/networking/can.html)

[CAN Protocol in STM32 → ControllersTech®](https://controllerstech.com/can-protocol-in-stm32/)

[Linux SocketCAN 编程（C++，启用多线程接收）\_cansocket-CSDN博客](https://blog.csdn.net/Flag_ing/article/details/126387114)

# Phụ lục: Bảng thuật ngữ

* **ASIL (Automotive Safety Integrity Level):** Mức độ toàn vẹn an toàn ô tô, một hệ thống phân loại rủi ro theo tiêu chuẩn ISO 26262.
* **CAN (Controller Area Network):** Một giao thức mạng truyền thông nối tiếp mạnh mẽ, được sử dụng rộng rãi trong ngành ô tô.
* **DTC (Diagnostic Trouble Code):** Mã lỗi chẩn đoán, được các ECU tạo ra khi phát hiện sự cố.
* **DTE (Distance-to-Empty):** Khoảng cách ước tính xe có thể đi được với lượng năng lượng còn lại.
* **ECU (Electronic Control Unit):** Bộ điều khiển điện tử, một máy tính nhúng điều khiển một hoặc nhiều hệ thống con trên xe.
* **HMI (Human-Machine Interface):** Giao diện người-máy, là điểm tương tác giữa người dùng và máy móc.
* **IPC (Instrument Panel Cluster):** Cụm đồng hồ trên bảng điều khiển.
* **ISO (International Organization for Standardization):** Tổ chức Tiêu chuẩn hóa Quốc tế.
* **MIL (Malfunction Indicator Lamp):** Đèn báo lỗi, thường được gọi là đèn "Check Engine".
* **Odometer:** Đồng hồ đo tổng quãng đường đã đi của xe.
* **QML (Qt Modeling Language):** Một ngôn ngữ khai báo dựa trên JavaScript để thiết kế giao diện người dùng trong framework Qt.
* **SoC (State of Charge):** Trạng thái sạc của pin, thường được biểu thị bằng phần trăm (%).
* **VCU (Vehicle Control Unit):** Bộ điều khiển xe, là ECU trung tâm điều phối hoạt động của toàn bộ chiếc xe.