VIETNAM - KOREA UNIVERSITY OF INFORMATION AND COMMUNICATION TECHNOLOGY
FACULTY OF COMPUTER ENGINEERING AND ELECTRONICS

GRADUATION PROJECT

DESIGN AND DEVELOPMENT OF A REAL-TIME DISPLAY INSTRUMENT CLUSTER INTEGRATED WITH AI FOR ADVANCED DRIVER ASSISTANCE SYSTEM

Student's Full Name:	*Tran Van Quoc Dat – 21CE077*
Class:	*21CE2*
Major:	*Computer Engineering Technology*
Specialization	*Computer Engineering Technology*
Supervisor:	*MSc. Nguyen Thi Huyen Trang*

Da Nang – 1/2026

VIETNAM - KOREA UNIVERSITY OF INFORMATION AND COMMUNICATION TECHNOLOGY
FACULTY OF COMPUTER ENGINEERING AND ELECTRONICS

GRADUATION PROJECT

DESIGN AND DEVELOPMENT OF A REAL-TIME DISPLAY INSTRUMENT CLUSTER INTEGRATED WITH AI FOR ADVANCED DRIVER ASSISTANCE SYSTEM

Student's Full Name:	*Tran Van Quoc Dat – 21CE077*
Class:	*21CE2*
Major:	*Computer Engineering Technology*
Specialization	*Computer Engineering Technology*
Supervisor:	*MSc. Nguyen Thi Huyen Trang*

Da Nang – 1/2026

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to the Board of Directors of the Vietnam-Korea University of Information and Communication Technology, and to all the esteemed lecturers of the Faculty of Computer and Electronic Engineering for teaching and imparting valuable knowledge to me throughout my years of study.

In particular, I would like to express my most sincere and profound gratitude to Ms. Nguyen Thi Huyen Trang. Throughout the project, she consistently dedicated her time to caring for, guiding, and providing professional advice. Her enthusiasm and rigorous yet heartfelt feedback helped me overcome many technical challenges to complete the project in the best possible way.

I would also like to express my gratitude to my family, who have always been my emotional support and provided me with the best possible conditions to focus on my studies.

Finally, I would like to thanks my friends who have always stood by me, shared their experiences, and supported me throughout the product development process.

Although I have devoted much effort and dedication to completing this project, due to my limited knowledge and experience, the report is inevitably flawed. I sincerely hope to receive guidance and feedback from the esteemed professors on the committee so that the project can be further improved.

Sincerely,

SUPERVISOR'S EVALUATION

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

Da Nang, .../.../2026
Supervisor

MSc. Nguyen Thi Huyen Trang

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ACK	Acknowledge
ADAS	Advanced Driving Assistance System
AI	Artificial Intelligence
BCN	Body Control Node
CNN	Convolutional Neural Network
CRC	Cyclic Redundancy Check
ECM	Engine Control Module
ECU	Electronic Control Unit
ELU	Exponential Linear Unit
HAL	Hardware Abstraction Layer
PWM	Pulse Width Modulation
QML	Qt Modeling Language
ReLU	Rectified Linear Unit
SoC	System on Chip
SSD	Single Shot Multibox Detector
TFLite	TensorFlow Lite
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
YOLO	You Only Look Once

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

1. Reasons for choosing the topic

The automotive revolution is a transformation from luxury to mass-market transportation, beginning with Karl Benz's invention of the gasoline-powered car (1886), then democratized by Henry Ford with the assembly line (Model T, 1908), and today entering a new era with electric cars, hybrids, autonomous driving technology, and smart connectivity, profoundly changing the economy, society, and environment, moving towards greater sustainability and safety.

In this context, car instrument panels are no longer simply mechanically designed but are gradually shifting to digital screens, integrating digital technology, displaying diverse information, customizable interfaces, and smart connectivity, providing a modern, highly customizable driving experience with much more intuitive information compared to older instrument panels.

2. Research Objectives

This section is provided as a sample for reference only. The primary objective of this project is to successfully design and build a prototype digital instrument panel system capable of stable operation in a real-time environment. Specifically, the system must be able to accurately collect and decode data from sensors via the CAN communication protocol, and then display it on a screen with a smooth and intuitive graphical interface. In addition, the project aims to integrate an artificial intelligence model capable of analyzing camera images to provide intelligent safety warnings, thereby demonstrating the processing capabilities of AI systems on embedded devices in the automotive electronics field

3. Research Subjects and Scope

This research focuses on high-performance embedded system architecture, standard automotive communication protocols such as CAN bus, and optimized graphics libraries for display screens. Additionally, the study delves into computer vision algorithms and machine learning models optimized for resource-constrained hardware. In terms of scope, the research is limited to developing a prototype system operating in a laboratory environment with simulated data from the ECU and dashcam; direct testing on commercial vehicles under harsh real-world traffic conditions has not been conducted.

4. Research Methodology

a. Functional scope

The project will focus on developing the following basic and advanced functionalities:

- Basic function: Collect and display accurate and real-time driving parameters (vehicle signals, engine speed, traffic signs).
- Integrated AI functionality: Uses TFLite and cameras to recognize common traffic signs.
- Connectivity functionality: Focuses on communicating with other vehicle ECUs via the CAN protocol. This project will not delve into other complex protocols such as LIN or FlexRay.

b. Technological scope

- Hardware Platform: Uses microcontrollers (STM32, ESP32) and embedded development boards (Raspberry Pi) as the central hardware.
- Programming Environment: Uses programming languages such as C/C++ for embedded software and Python for AI models.
- Libraries/Frameworks: Uses the Qt graphics library to develop the interface. AI libraries such as TensorFlow Lite and OpenCV will be used to process images and data.

5. Scientific and Practical Significance

Scientifically, this project contributes to the research of integrated multi-service solutions on a single embedded platform, from digital signal processing and real-time graphics to artificial intelligence computing. The research results provide valuable reference material for optimizing AI models on low-power embedded devices. Practically, the project's product offers a modern, highly customizable driver information display solution with integrated intelligent safety features. This serves as a foundation for developing the domestic automotive sector, supporting digital transformation in the transportation industry, and opening up opportunities for widespread application of smart electric vehicles in Vietnam in the future.

Based on this important application, I have chosen the topic "**Design and development of a real-time display instrument cluster integrated with AI for advanced driver assistance system.”**

CHAPTER 1. OVERVIEW AND THEORETICAL FOUNDATION

1.1. Overview of advanced driver assistance system

1.1.1. Definition and history of development

1.1.1.1. Definition

Advanced Driver Assistance Systems (ADAS) are a collection of active safety technologies equipped in automobiles to assist drivers, minimize the risk of accidents, and provide a safer and more comfortable driving experience.

ADAS systems operate through a combination of cameras, radar, ultrasonic sensors, and intelligent control software, enabling the vehicle to detect hazards, issue warnings, and even automatically intervene in the driving process (braking, steering, acceleration, etc.) when necessary.

1.1.1.2. History of development

The history of ADAS is an evolutionary journey from simple mechanical safety mechanisms to today's complex artificial intelligence systems. This process can be divided into the following main stages:

- The early stage (1950s - 1970s): The first steps of ADAS appeared with cruise control systems in the 1950s, allowing vehicles to maintain a stable speed on highways. By the 1970s, anti-lock braking systems (ABS) were introduced, marking the initial electronic intervention in braking control to enhance safety.
- The sensor integration stage (1980s - 1990s): During this stage, sensors began to be used more widely. Traction control and electronic stability control systems were developed, helping vehicles operate safely in slippery road conditions or when cornering sharply.
- The development phase of intelligent sensors (2000s - 2010s): This was the period of explosive growth in modern technologies such as radar and cameras. Features like lane departure warning, lane keeping assist, and automatic emergency braking began to be equipped on high-end vehicles. The system began to be able to "see" and "recognize" its surroundings instead of just reacting to the vehicle's physical parameters.
- The AI and connectivity phase (2020s to present): Currently, ADAS has entered a new era thanks to the integration of artificial intelligence and machine learning. Modern ADAS systems not only provide warnings but also have the ability to

automatically handle complex situations such as automatic parking, advanced autonomous driving assistance, and communication with surrounding vehicles. Integrating AI enables the system to learn from real-world data, thereby improving accuracy and reliability in protecting drivers.

1.1.2. The main functions of ADAS

ADAS systems perform functions ranging from passive warnings to active intervention in the vehicle's control system. These functions can be divided into the following main groups:

1.1.2.1. Warning functions

This is the basic group of functions that uses sound and images to inform the driver about potential hazards:

- Lane Departure Warning: Identifies lane markings and warns when the vehicle shows signs of drifting out of its lane without using a turn signal.
- Forward Collision Warning: Uses radar to measure the distance to the vehicle in front and warns if there is a risk of collision.
- Blind spot warning: Detects vehicles in the blind spot of the rearview mirror and displays a warning signal on the mirror.

1.1.2.2. Assisted control functions

These functions directly intervene in the braking, acceleration, or steering systems to assist the driver:

- Adaptive Cruise Control: Not only maintains a fixed speed but also automatically adjusts speed to maintain a safe distance from the vehicle in front.
- Automatic Emergency Braking: If the driver does not react in time after a collision warning, the system will automatically activate the brakes to minimize damage or avoid an accident.
- Lane Keeping Assist: Actively adjusts the steering wheel to bring the vehicle back to the center of the lane if it tends to drift out of the lane.
- Automatic Parking Assist: Automatically controls the steering wheel, brakes, and acceleration to accurately guide the vehicle into a parking position (parallel or perpendicular).

1.1.2.3. Vision and recognition enhancement function group

Utilizing cameras and AI to process environmental information:

- Traffic sign recognition: Recognizes speed limit signs, no-overtaking signs, etc., and displays them directly on the Instrument Cluster screen to remind the driver.
- Adaptive headlights: Automatically adjusts the light beam to optimize visibility at night without dazzling oncoming vehicles.
- Night vision system: Uses thermal cameras to detect pedestrians or animals in low-light conditions beyond the range of the headlights.

1.1.3. The role of the instrument cluster in the ADAS system

The digital instrument cluster acts as the central visual information display for the driver, transforming complex data from sensors into warnings in the form of images, sounds, and vibrations. In the ADAS ecosystem, it performs the following tasks:

- Sensor data visualization: Converting complex technical data from Radar, LiDar, and Cameras into easily understandable visual signals such as lane simulations, distances to the vehicle ahead, or surrounding obstacles.
- Instant warning transmission: When ADAS functions detect a hazard, the instrument cluster is the first to issue visual warnings through colors and flashing icons, allowing the driver to react quickly before the system intervenes automatically.
- System operating status management: Informs the driver which assistive features are active (e.g., Adaptive Cruise Control is in standby mode or maintaining distance), helping the driver to trust and better control the automated features.
- Optimizes driver focus: By integrating artificial intelligence to prioritize the display of the most important information depending on the driving context, the instrument cluster reduces information overload, ensuring the driver always receives necessary notifications without distraction.
- Intelligent interactive bridge: With AI integration, the instrument cluster displays driver behavior analysis, creating a feedback loop that helps adjust driving habits for safer driving.

1.2. Artificial intelligence and computer vision

1.2.1. Common object recognition models

In the field of computer vision, the object recognition problem goes beyond simply classifying images; it also involves accurately determining the object's position within the frame using bounding boxes. Currently, deep learning-based object recognition models are divided into two main architectures based on their processing procedures: two-stage models and one-stage models.

1.2.1.1. Two-stage model

The two-stage model solves complex problems by breaking them down into two stages for processing:

- Stage 1: The neural network extracts features and suggests regions that may contain objects.
- Stage 2: The suggested regions are fed into a subnet to classify the object and refine the bounding box coordinates.

Typical models in this group include R-CNN, Fast R-CNN, and the most common is Faster R-CNN.

This group of models achieves very high accuracy, especially with small or obscured objects. However, due to the complex structure and sequential computation over two steps, the inference speed is often slow (usually below 10 FPS on common hardware), making it difficult to meet real-time requirements on embedded devices.

1.2.1.2. One-stage model

Unlike two-stage models, one-stage models completely eliminate the object location detection step. Instead, they combine the two steps into a single neural network and process everything in a single input-to-output propagation. This means the model doesn't need to generate a proposal area, doesn't require additional processing stages; the image is fed in, the model extracts features, and returns the envelope and class at the final step.

This group of models is best suited for embedded and real-time tasks due to its streamlined architecture.

- YOLO: This is the most famous model in this group. YOLO divides the input image into an S x S grid. If the center of an object falls into a grid cell, that cell is responsible for detecting the object. Continuously improved versions optimize the balance between speed and accuracy. This model has extremely fast processing speed and a small model size.

- SSD: A model for solving the problem of detecting objects of different sizes by using feature maps at various scales. SSDs are often combined with lightweight feature extraction tools like MobileNet to run efficiently on resource-limited devices.

1.2.2. Introduction to convolutional neural networks

CNNs are one of the most advanced deep learning models, specifically designed to process grid-structured data such as digital images. In advanced driver assistance systems, CNNs act as the visual brain, enabling the system to analyze and understand image data captured from dashcams.

1.2.2.1. Convolutional layer

This is the core component of the network, performing convolutional calculations between the input image data and the filters. This process allows the network to automatically extract features from low to high resolution of the image, from simple lines and edges to complex shapes and entities.

During the convolution process, if the movement "jumps" a certain value S, the output image size is reduced compared to the input image, called Stride = S. At the edges of the image, if there are missing pixels to create a neighborhood, we add P pixels at the edges, called Padding = P.

If we call P the number of pixels added to the edges, S the stride value, m x n the input image size, k x k the mask size, the image size after convolution is:

$$\left(\frac{m-k+2p}{8}+1\right)*\left(\frac{n-k+2p}{s}+1\right)$$

The output of the convolutional layer consists of input image features called feature maps or activation maps. After obtaining K activation maps, they are stacked on top of each other to form the input of the next layer.

1.2.2.2. Activation function layer

Activation functions such as ELU, Leaky ReLU, Tanh, etc., are applied, but the ReLU (Rectified Linear Unit) function is commonly used to introduce nonlinearity into the model, enabling the network to learn and represent more complex data structures.

1.2.2.3. Pooling layer

The concatenation layer is often used between convolutional layers to reduce the spatial size of feature maps after convolution. This process helps reduce the number of computational parameters, thereby reducing the computational load on the embedded system and limiting overfitting.

1.2.2.4. Fully connected layer

The extracted and dimensionally reduced features are then flattened and fed into fully connected layers to perform the final classification task, such as identifying whether the object is a traffic sign, a pedestrian, or another vehicle.

1.2.3. TensorFlow Lite for embedded applications

In the context of deploying deep learning models to edge devices and embedded systems, the biggest challenge lies in the limitations of computing resources, memory capacity, and power requirements. Google developed TFLite as a specialized solution to address these issues, enabling inference execution directly on the device with low latency.

1.2.3.1. Architecture and operating pinciples

TFLite is not used for model training but focuses entirely on optimizing the inference process. The implementation process includes two main components:

- Converts the original TensorFlow model (usually in SavedModel or Keras H5 format using 32-bit floating-point numbers) to the .tflite format. During this process, the converter removes unnecessary operations and optimizes the computation graph.
- Interpreter: This is a lightweight library that runs on the embedded device. The interpreter loads the .tflite model and performs computations based on the device's available hardware, supporting hardware acceleration.

1.2.3.2. Quantization techniques

One of TFLite's most important features for embedded systems is its quantization technique. This technique reduces the precision of weight parameters and enables the transition from 32-bit floating-point numbers to smaller formats such as 8-bit integers (Int8) or 16-bit floating-point numbers (Float16).

Benefits of quantization in ADAS/Instrument Cluster problems:

- Reduced model size: Int8 models are typically four times smaller than Float32 models, saving storage space on memory cards or Flash memory.
- Increased inference speed: Operations on 8-bit integers are significantly faster than on floating-point numbers on ARM CPU architectures (such as Cortex-A on Raspberry Pi), resulting in higher FPS.
- Reduced latency: Suitable for the real-time requirements of driver assistance systems, where processing latency can affect safety.

1.3. Embedded systems and protocol standards

To develop the project effectively, the selection of processing hardware and communication methods is crucial. This section provides an overview of the hardware devices and communication standards used in the system.

1.3.1. STM32F103C8T6 microcontroller

1.3.1.1. Introduction

The STM32F103C8T6 is a 32-bit microcontroller from STMicroelectronics' STM32 family, based on the ARM Cortex-M3 architecture. It's a powerful microcontroller widely used in embedded applications, from consumer electronics to industrial automation, thanks to its good performance, compact size, and low power consumption.

In this project, it acts as a real-time processor, performing the task of collecting values from sensors and buttons.

1.3.1.2. Technical specifications

- CPU Architecture: ARM Cortex-M3, maximum speed 72 MHz.
- Flash: 64KB Flash memory for program storage.
- RAM: 20KB SRAM.
- Operating Voltage: 2.7V to 3.6V.
- GPIO: 37 GPIO pins, configurable for various functions.
- USART/UART: 3 channels.
- SPI: 2 channels.
- I2C: 2 channels.

- CAN: 1 channels (CAN 2.0B).
- USB: 1 channels (USB 2.0 Full-Speed).
- ADC: 2 x 12-bit ADCs with 16 channels.
- PWM: Supported on Timer pins.
- Timer: 4 x 16-bit timers (including PWM).

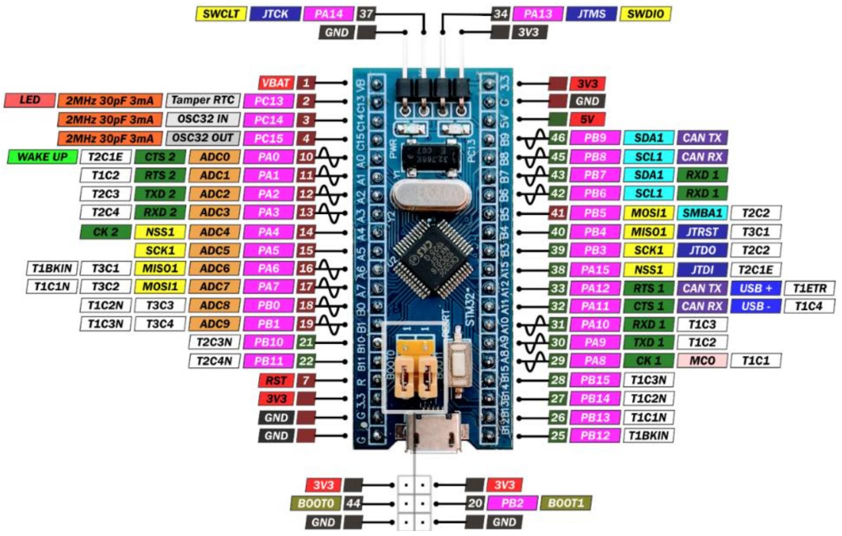

Figure 1.1. STM32F103C8T6 pinout

1.3.2. ESP32 microcontroller

1.3.2.1. Introduction

Besides the STM32 microcontroller handling low-level control functions, the system also integrates the ESP32 module from Espressif Systems. This is a high-performance SoC microcontroller, notable for its built-in communication interfaces, acting as an intermediary node in the system. In this project, the ESP32 is configured as the endpoint of the CAN network, performing the task of collecting and aggregating data from other nodes, and then transmitting this information to the Raspberry Pi via the UART protocol.

1.3.2.2. Technical specifications

- Controller Chip: ESP32
- Number of Pins: 30 pins
- Wireless Connectivity: Wi-Fi and Bluetooth
- Operating Voltage: 3.3V
- Memory: 520KB RAM, Flash memory from 4MB

- GPIO: 34 programmable GPIO pins
- ADC: 12-bit ADC with 18 channels
- DAC: 12-bit ADC with 18 channels
- PWM: Supports PWM for motor and lighting control
- Peripheral Interfaces: SPI, I2C, UART

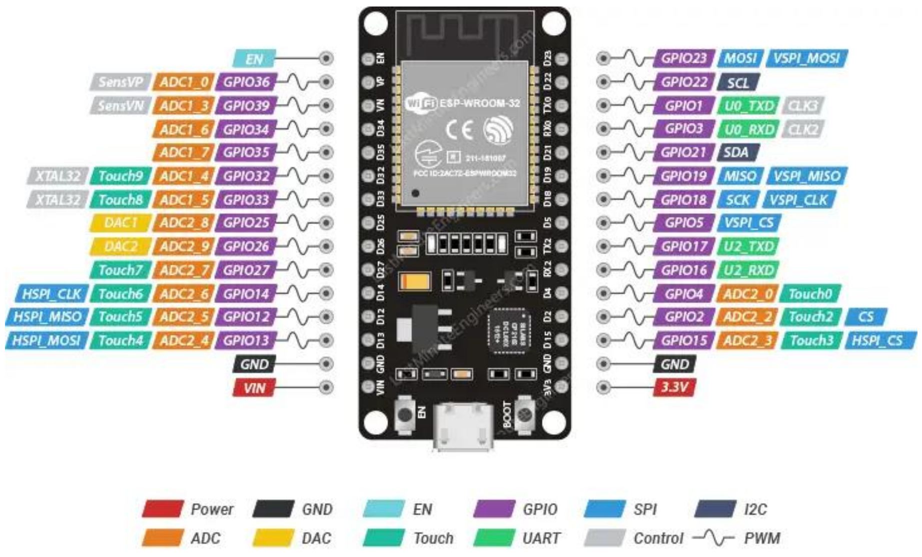

Figure 1.2. ESP32 pinout

1.3.3. Raspberry Pi 4

1.3.3.1. Introduction

The Raspberry Pi 4 is a compact, powerful mini-computer ideal for DIY projects, programming, AI, IoT, web servers, and multimedia centers thanks to its high performance and diverse connectivity options.

In this project, the Raspberry Pi 4 acts as the powerful central processor, responsible for running the operating system, processing images, and displaying the graphical interface.

1.3.3.2. Technical specifications

- Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC 1.8GHz
- 4GB LPDDR4-2400 SDRAM (depending on model)
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE
- Gigabit Ethernet
- 2 USB 3.0 ports / 2 USB 2.0 ports

- Raspberry Pi standard 40 pin GPIO header (fully backwards compatible with previous boards)
- 2 × micro-HDMI ports (up to 4kp60 supported)
- 2-lane MIPI DSI display port
- 2-lane MIPI CSI camera port
- 4-pole stereo audio and composite video port
- H.265 (4kp60 decode), H264 (1080p60 decode, 1080p30 encode)
- OpenGL ES 3.0 graphics
- Micro-SD card slot for loading operating system and data storage

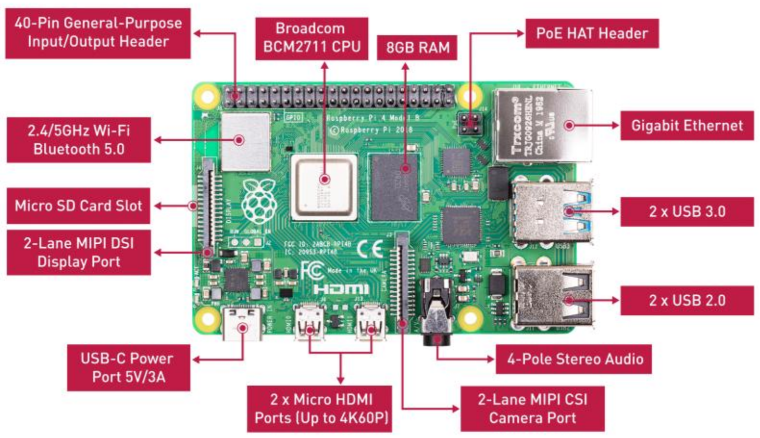
40-Pin General-Purpose Input/Output Header
Broadcom BCM2711 CPU
8GB RAM
PoE HAT Header
2.4/5GHz Wi-Fi Bluetooth 5.0
Gigabit Ethernet
Micro SD Card Slot
2 x USB 3.0
2-Lane MIPI DSI Display Port
2 x USB 2.0
USB-C Power Port 5V/3A
4-Pole Stereo Audio
2 x Micro HDMI Ports (Up to 4K60P)
2-Lane MIPI CSI Camera Port

Figure 1.3. Raspberry Pi 4

1.3.4. CAN protocol

1.3.4.1. Overview

The CAN network in automobiles is a standard network in automotive communication that allows different modules of the car to communicate with the car's ECUs. It is a group of two high-speed signal wires that allow compressed data and commands for vehicle calibration to be transmitted back and forth between different modules.

1.3.4.2. CAN network structure

CAN uses two wires for communication. These wires are called CAN High and CAN Low. CAN connects all components on the network through these two wires. In CAN, there is no distinction between master and slave like in some other protocols. The bus wire terminates with a 120-ohm resistor (minimum 108 ohms and maximum 132 ohms) at each end.

A CAN network is made up of a group of nodes. Each node can communicate with any other node in the network. Communication is done by transmitting and receiving messages. Each type of message in a CAN network is assigned an identifier number depending on the priority of that message. Messages with smaller IDs have higher priority.

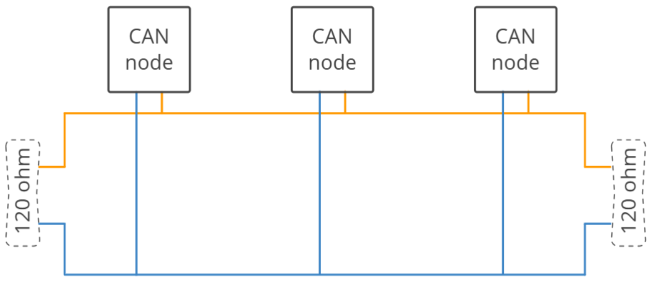

Figure 1.4. CAN bus network connection structure

Unlike some other protocols that use the voltage level on the wire relative to GND to convert to logic bits, CAN uses the voltage difference between the two wires for conversion. These voltage level changes are then translated into logic levels.

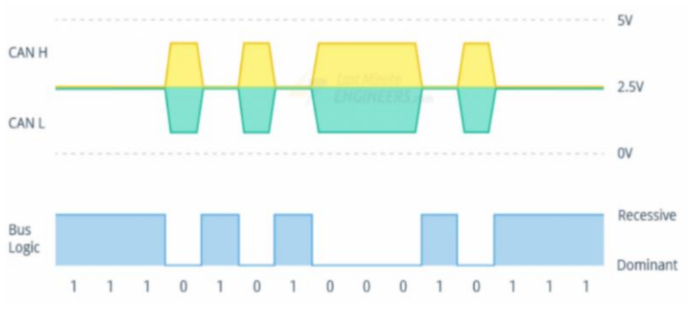

Figure 1.5. Illustration of differential signals

In the CAN protocol, data is transmitted as differential signals on two lines, CAN_H and CAN_L. In the idle state (Recessive – logic 1), the voltages on the CAN_H and CAN_L lines are approximately equal, around 2.5V, so the voltage difference between the two lines is close to 0V.

When the bus is in the Dominant state (logic 0), the voltage on CAN_H increases to approximately 3.5–3.75V, while the voltage on CAN_L decreases to approximately 1.5–1.25V. At that point, the voltage difference between CAN_H and CAN_L reaches approximately 2.0–2.5V. The use of differential signals gives the CAN protocol high noise immunity and stable data transmission in industrial and automotive environments.

Because it uses differential signals, CAN has very good noise immunity. When noise is present, CAN controls the voltage level on both wires to change simultaneously (either increase or decrease simultaneously), so the voltage difference between the two wires remains unchanged.

1.3.4.3. Structure of an ECU node on a CAN

An ECU on a CAN has three basic components: Microcontroller, CAN Controller, and CAN Transceiver, each with different functions:

- Microcontroller: Processes data received from the CAN controller and sends it to another ECU. These two processes can occur independently or simultaneously depending on the ECU's task.
- CAN controller: Transmits serial data from the microcontroller to the bus when the bus is free. When receiving data, it stores the received bits from the bus until the entire transmission frame is complete and notifies the microcontroller.
- CAN transceiver: Converts the voltage level on the CAN into information that the CAN Controller can understand, and conversely, converts the data stream from the CAN Controller into the logic level of the CAN.

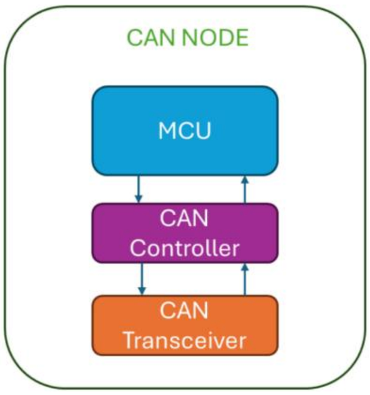

Figure 1.6. Diagram of a single CAN network node

1.3.4.4. Types of CAN frames

Each ECU on the bus has a unique ID that distinguishes it from other ECUs. All ECUs on the bus are connected in parallel to the line, meaning all ECUs receive all data running on the bus at all times. We can configure an ECU to receive packets from only one or more specific ECUs using their ID.

CAN data is transmitted as frames. There are four different types of frames:

- Data Frame: A frame carrying data from a data transmitter to data receivers. This frame has a region to carry data bytes.

- Remote Frame: A frame transmitted from any node to request data from another node. When that other node receives the request, it will transmit back data with an ID matching the ID sent in the Remote Frame.
- Error Frame: A frame transmitted by any node when that node detects an error on the bus.
- Overload Frame: This is a frame sent by a node if it cannot process the data in time or detects an overload condition on the bus. Its purpose is to create additional delay to complete processing. Overload frames do not directly prevent other nodes from sending data; they only create a pause to allow the node sufficient time to process the data.

1.3.4.5. Packet structure during transmission

The ECU's packet structure during transmission includes:

- Bus Idle: The bus is in the idle state.
- Start of frame bit: The position of the first bit in the data frame.
- Arbitration Field:
  + Identifier: The identifier of an ECU. It is used to specify which ECU is sending the packet. The ID also determines the priority level: the lower the ID, the higher the packet's priority. It consists of an 11-bit ID for Standard packets and a 29-bit ID for Extended packets.
  + Remote transmission request: A bit used to identify whether the frame is a Data frame or a Remote frame.
  + Substitute remote request: This bit replaces the Remote transmission request bit in the standard frame, holding a value of 1.
- Control Field:
  + Identifier extension: This bit distinguishes between the standard frame and the extended frame.
  + r, r0, r1: Reserved bits, holding a value of 0.
  + Data length code: Specifies the number of bytes in the Data field (0 to 8 bytes).
- Data Field: Contains the actual data to be transmitted, with a maximum length of 8 bytes.

- Cyclic Redundancy Check Field:
    + Cyclic Redundancy Check Sequence: The encoded sequence to check data integrity.
    + Cyclic Redundancy Check Delimiter: A fixed bit of 1, separating the CRC and ACK.
- Acknowledge Field:
    + Acknowledge Slot: The node receiving the data writes a value of 0 here to confirm successful receipt.
    + Acknowledge Delimiter: A fixed bit of 1, separating the ACK from the rest of the data.
- End Of Frame: 7 fixed bits of 1, signaling the end of the data frame.

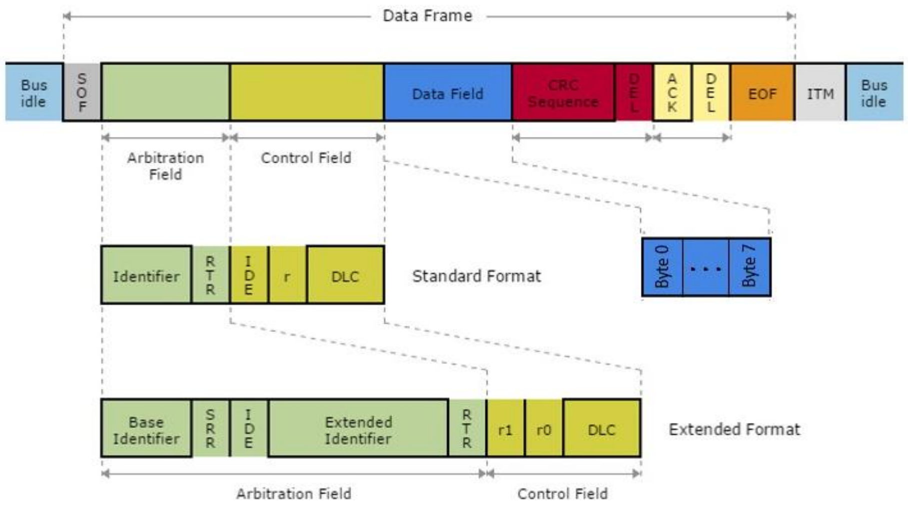

Figure 1.7. Data frame structure

In this project, the CAN protocol is used as the primary communication standard between modules in the embedded system. The CAN bus is responsible for transmitting control and status data between processors, ensuring stable, reliable, and system-compliant information exchange.

Specifically, the CAN protocol is used to transmit data between the central processing module and the control module. The processing module collects and processes data, then packages it into CAN frames and transmits it over the CAN network. The control module receives the CAN data, decodes the message content, and performs corresponding control tasks.

Using the CAN protocol ensures stable system operation in noisy environments, guaranteeing accurate and timely data transmission. Furthermore, CAN allows for system expansion by adding network nodes without significant changes to the hardware structure, contributing to the project's flexibility and practical applicability.

1.3.5. UART protocol

1.3.5.1. Overview

UART is an asynchronous serial communication protocol widely used in embedded systems for data exchange between microcontrollers and peripheral devices. The protocol does not require a shared clock signal; instead, the transmitter and receiver operate based on predefined communication parameters. UART is commonly applied in microcontroller-to-computer communication, system debugging, device configuration, and simple control data exchange.

1.3.5.2. Data transmission principles

Data in a UART is transmitted frame by frame. A standard UART frame includes the following components:

- Start bit: This is the start bit of transmission, with a low logic level, used to signal to the receiver that data transmission is about to begin.
- Data bits: These are the main data bits, usually 5 to 9 bits long, with 8 bits being the most common.
- Parity bit: Used for simple error checking during data transmission.
- Stop bit: This is the end bit of the transmission frame, with a high logic level, which can be 1 or 2 bits.

Each UART frame is transmitted continuously from the start bit to the end bit. The receiver uses the configured baud rate to sample and decode the received data.

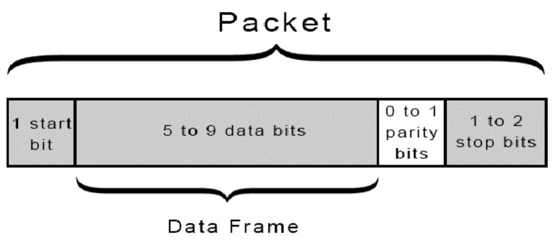

Figure 1.8. UART frame structure

1.3.5.3. Configuration parameters

To ensure accurate data transmission and reception, UART communication devices need to be configured with the following parameters:

- Baud rate: Data transmission speed, in bits per second (bps), e.g., 9600 bps, 115200 bps.
- Data bits: Number of data bits in a transmission frame, usually 8 bits.
- Parity: Error checking mode, including None, Even, or Odd.
- Stop bits: Number of termination bits, usually 1 bit.
- Flow control: Data flow control mechanism, usually not used in simple applications.

In current embedded systems, the most common UART configuration is 115200 baud, 8 data bits, no parity, and 1 stop bit.

1.3.5.4. Connection diagram

UART uses two main signal lines: TX (Transmit) and RX (Receive). When connecting two devices, the TX pin of one device must be connected to the RX pin of the other, and vice versa. Additionally, the devices need to share a common ground point to ensure the same reference voltage level.

UART connections have a simple structure and do not require complex hardware, making them very convenient for system deployment and testing during the development phase.

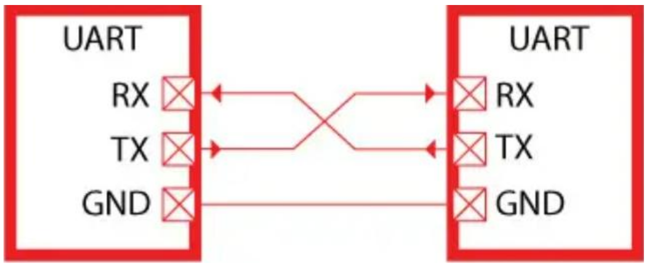

Figure 1.9. UART connection diagram

Within the scope of this project, the UART protocol is used to transmit data between ESP32 and Raspberry Pi. This overcomes hardware limitations, reduces complexity, and facilitates system testing and debugging during development.

1.3.6. UDP protocol

1.3.6.1. Overview

UDP is a communication protocol belonging to the Transport Layer in the TCP/IP model. UDP operates on a connectionless mechanism, where data is sent as independent packets without requiring prior connection establishment between the sender and receiver.

Unlike the TCP protocol, UDP does not guarantee absolute reliability during data transmission, nor does it implement error control, flow control, or retransmission mechanisms in case of packet loss. However, thanks to its simple structure and lack of packet acknowledgment requirements, UDP has low latency and high transmission speeds, making it suitable for real-time applications.

A UDP packet consists of a fixed-length 8-byte header and a data packet. The header contains basic information such as the source port, destination port, packet length, and an error checking field. This simple structure reduces the processing load on the system during data transmission and reception.

1.3.6.2. Characteristics

The main characteristics of the UDP protocol include:

- No prior connection establishment is required before data transmission.
- No guarantee of packet order or integrity.
- Low latency and high transmission speed.
- Easy to implement on embedded systems and embedded computers.

1.3.6.3. Application in the system

In embedded systems and real-time data processing systems, UDP is often used to transmit state data or processing results at high frequency. In this project, the UDP protocol is used to transmit data between software blocks running on a Raspberry Pi, specifically to transmit image processing results from the traffic sign recognition function to the Instrument Cluster interface.

The use of UDP ensures that data is updated quickly to the interface while not interrupting the main processing of the system, meeting the real-time requirements of the project.

CHAPTER 2. SYSTEM ANALYSIS AND DESIGN

2.1. System requirements analysis

This section outlines the system requirements to clearly define the functions to be performed as well as the technical constraints during the design and implementation process. System requirements analysis forms the basis for building the hardware and software architecture and selecting the appropriate communication protocol for the project.

2.1.1. Functional requirements

The system is designed to perform the following main functions:

- The system is capable of acquiring images from a camera and processing them on a Raspberry Pi.
- The system performs traffic sign recognition based on image processing algorithms and a trained deep learning model.
- The system analyzes the recognition results to determine the corresponding control information.
- The system uses communication protocols (CAN, UART) to communicate between nodes.
- Microcontrollers (STM32, ESP32) receive data, decode information, and perform the corresponding functions.
- The system operates continuously and ensures that control requirements are met in real time.

2.1.2. Non-functional requirements

In addition to its primary functions, the system needs to meet the following non-functional requirements:

- Performance requirements
  + The processing time for identification and data transmission must be fast enough to meet control requirements.
  + Communication latency between nodes must be within the permissible limits of a real-time system.
- Reliability requirements

  + The system must operate stably over a long period.
  + Data communication between modules must be accurate and minimize transmission errors.
  + The system must be able to detect and handle errors during communication.
- Communication requirements
  + The communication protocol must be suitable for noisy environments and multi-node systems.
  + Support system expansion by adding more control nodes as needed.
  + Ensure compatibility between devices used in the system.
- Hardware requirements
  + Use common hardware such as Raspberry Pi, STM32, ESP32 for easy deployment and replacement.
  + The hardware system has a clear structure, is easy to install and maintain.
  + Suitable for experimental conditions and research models.
- Cost and feasibility requirements
  + Low deployment cost, components readily available on the market.
  + system is suitable for development into an experimental model for learning and research.
  + Capable of expansion and upgrading in subsequent research.

2.2. System architecture

This section presents the overall architecture of the system, including block diagrams, schematic diagrams of each node, and the operating principles of the system. This clarifies the relationships between nodes and how the system coordinates to perform its intended functions.

2.2.1. System block diagram

The system block diagram describes the overall structure of the system and the relationships between the main functional blocks. The system is designed using a distributed model, where each block performs a separate function to increase flexibility and scalability. The system includes 5 functional nodes connected via two physical transmission lines: CAN Bus and UART:

- Node 1 (Body Control Node): Uses an STM32F103 microcontroller, responsible for collecting control signals from the driver (buttons, joystick).
- Node 2 (Motor Control Node): Uses an STM32F103 microcontroller, responsible for controlling the motor and reading feedback from the encoder.
- Node 3 (Gateway Node): Uses an ESP32 microcontroller. This is an intermediate node acting as a data bridge between the CAN Bus network and the embedded computer.
- Node 4 (Central Unit): Uses a Raspberry Pi, responsible for displaying the user interface, processing images from the camera, and communicating with the Gateway Node via UART.
- Power Supply Unit: This unit provides a stable power supply for the entire system. It utilizes a linear power supply principle with a step-down transformer and bridge rectifier to convert the 220V AC mains voltage to DC voltage. The power supply system provides two separate output voltage levels: 12V and 5V.

Data flows within the system from the nodes to the gateway node for data aggregation, and then it is transmitted to the Raspberry Pi for display on the screen interface.

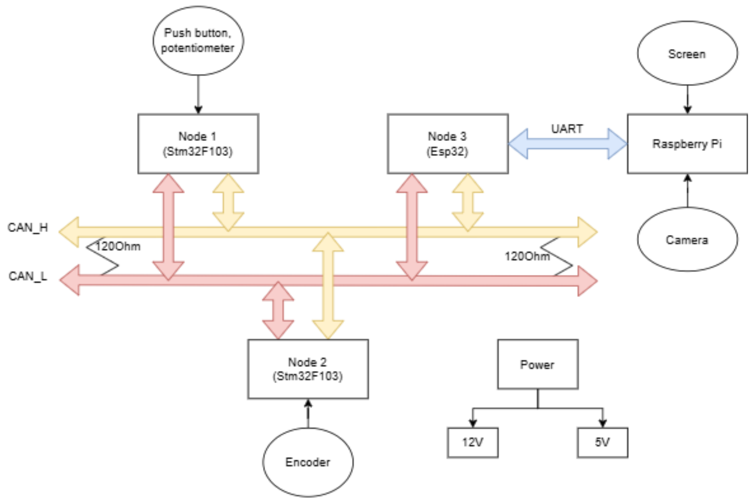

Figure 2.1. System overview diagram

2.2.2. Algorithm flowchart for each node

The hardware system is divided into four separate modules, ensuring modularity and ease of maintenance. The detailed structure and principles of each node are as follows:

2.2.2.1. Body Control node

The Body Control Node simulates the vehicle's central interface, acting as the primary bridge that converts driver inputs into standardized CAN Bus messages. Powered by an STM32F103C8T6 microcontroller, it aggregates signals from function buttons (lights, cruise control), an analog joystick for directional control, and an HC-SR04 ultrasonic sensor for obstacle detection. To ensure signal integrity, digital inputs are stabilized using pull-up resistors, while the joystick's analog signals are processed via the ADC to precisely map voltage levels to speed and steering commands. Network communication is robustly managed by an MCP2515 module via the SPI interface, facilitating the real-time transmission of control commands and system status to other nodes in the network.

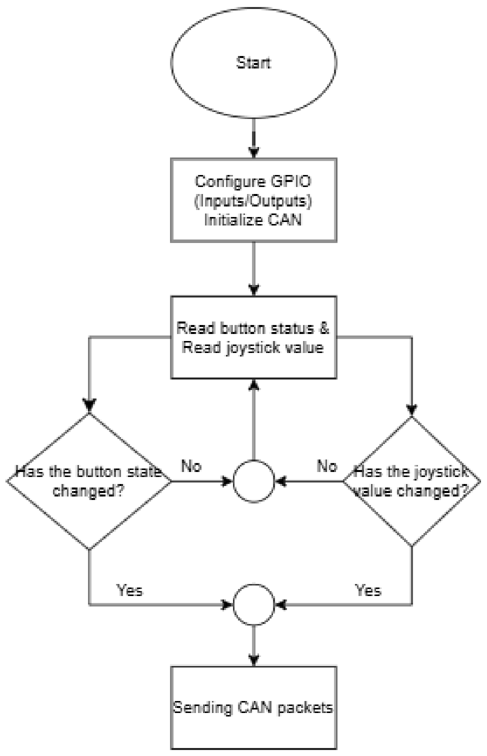
Start
Configure GPIO (Inputs/Outputs) Initialize CAN
Read button status & Read joystick value
Has the button state changed?
No
No
Has the joystick value changed?
Yes
Yes
Sending CAN packets

Figure 2.2. Algorithm flowchart for Body Control node

2.2.2.2. Motor Control node

The Motor Control Node is responsible for controlling the vehicle's operation, including controlling the speed and direction of the motor and providing real-time speed feedback to the CAN Bus network. This node simulates the function of the motor controller in an automotive system.

The central microcontroller uses an STM32F103C8T6, which receives CAN data frames from the network, decodes the control commands sent by the Body Control Node, and controls the motor accordingly. Simultaneously, the microcontroller collects feedback data from the encoder to calculate the speed and transmits it back to the CAN network.

The motor control block uses a module TB6612, receiving PWM pulse signals generated by the microcontroller's timer. By changing the PWM pulse width, the average voltage supplied to the motor is adjusted, thereby changing the motor's speed and direction of rotation.

To measure and provide feedback on speed, the system uses an optical encoder with a 20-pulse encoder disc mounted directly on the wheel axle, combined with a U-shaped optical sensor. The encoder's output pulse signal is fed into the STM32 Timer in External Clock mode. The microcontroller counts the pulses within a fixed time interval (100 ms), thereby calculating the car's actual speed.

Regarding network communication, the Motor Control Node uses the MCP2515 module to connect to the CAN Bus network, allowing the node to receive control commands from other nodes and send back feedback information such as speed and motor status, for monitoring and controlling the entire system.

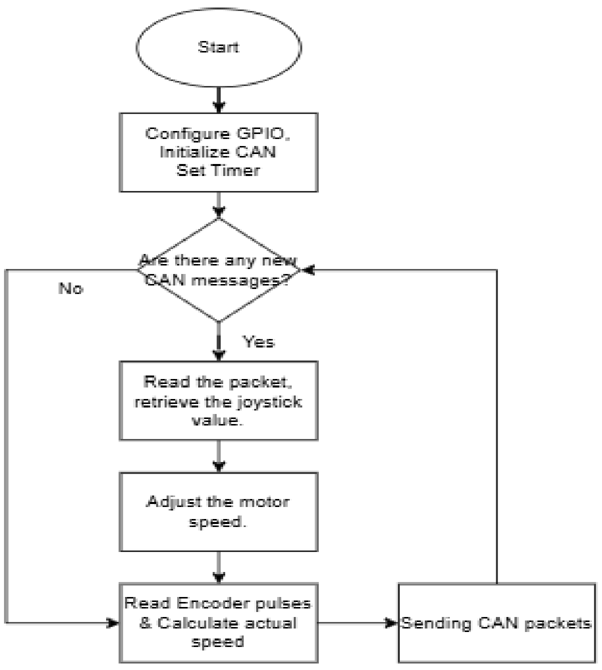
Start
Configure GPIO, Initialize CAN Set Timer
Are there any new CAN messages?
No
Yes
Read the packet, retrieve the joystick value.
Adjust the motor speed.
Read Encoder pulses & Calculate actual speed
Sending CAN packets

Figure 2.3. Algorithm flowchart for Motor Control node

2.2.2.3. Gateway node

The Gateway Node acts as an intermediary node in the system, connecting and converting data between two different transmission environments: CAN Bus and UART. This node separates the real-time control block from the data processing block, contributing to improved system stability and scalability.

The central microcontroller uses the ESP32-WROOM-32 module, which has powerful processing capabilities and supports many peripheral interfaces, making it suitable for the gateway role in a multi-protocol system.

On the CAN network side, the Gateway Node connects to the CAN Bus via the MCP2515 module, allowing the node to listen to and receive data frames transmitted from the Body Control Node and Motor Control Node. The ESP32 reads data from the MCP2515, decodes the CAN ID and the corresponding payload to determine the type of information.

On the UART network side, the Gateway Node uses Serial communication to establish a full-duplex data transmission channel with the Raspberry Pi embedded computer. After processing and conversion, the data is sent via UART for processing tasks.

The main function of the Gateway Node is to perform data parsing. Specifically, the ESP32 extracts raw data from CAN frames with conventional IDs (0x01, 0x02, ...), then converts this data into a character string or byte structure in a unified format, ensuring that the Raspberry Pi can receive and process it accurately and efficiently.

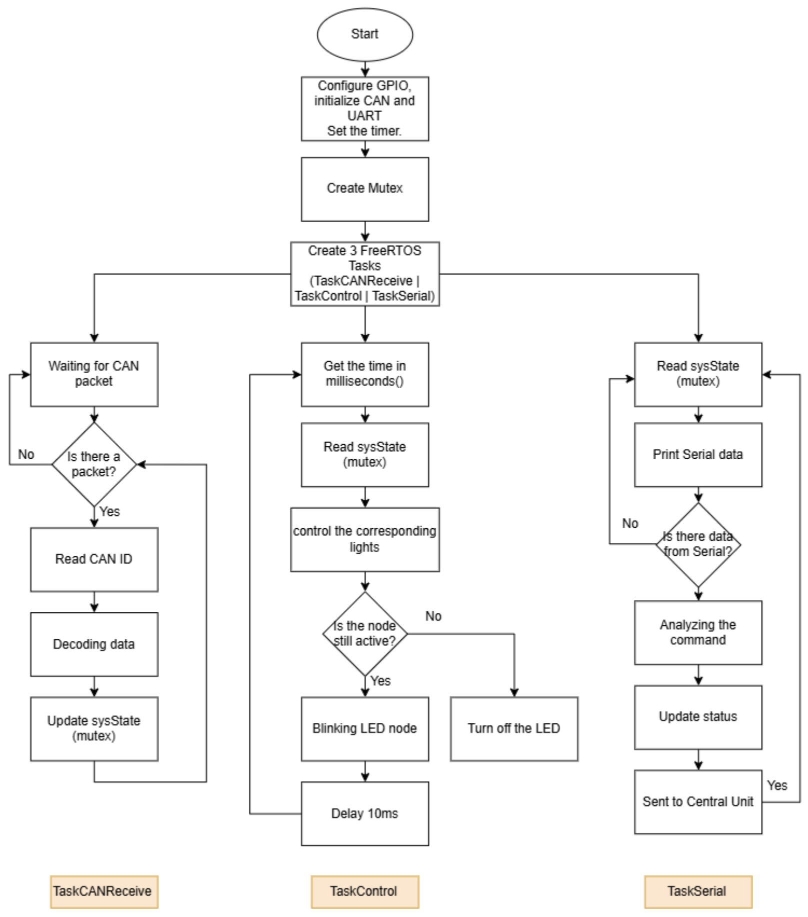

Figure 2.4. Algorithm flowchart for Gateway node

2.2.2.4. Central Unit

The Central Unit acts as the central processing unit of the entire system, handling high-level data processing functions such as image data collection and analysis, operational information display, and control decision support. This node simulates the central controller in modern automotive systems.

The Central Unit's hardware utilizes a Raspberry Pi 4 Model B embedded computer, a powerful processor suitable for resource-intensive tasks such as image processing, display interface, and system data management. The Raspberry Pi connects to an LCD screen via HDMI to display the instrument panel interface, providing operational information such as speed, engine status, and system warnings.

In addition, the Central Unit is equipped with a camera module directly connected to the Raspberry Pi to collect images of the environment in front of the vehicle. The image data is processed using traffic sign recognition algorithms, serving for environmental analysis and control support. The Central Unit receives operational information and system status from the Gateway Node via UART communication, then processes and synchronizes it to the display interface.

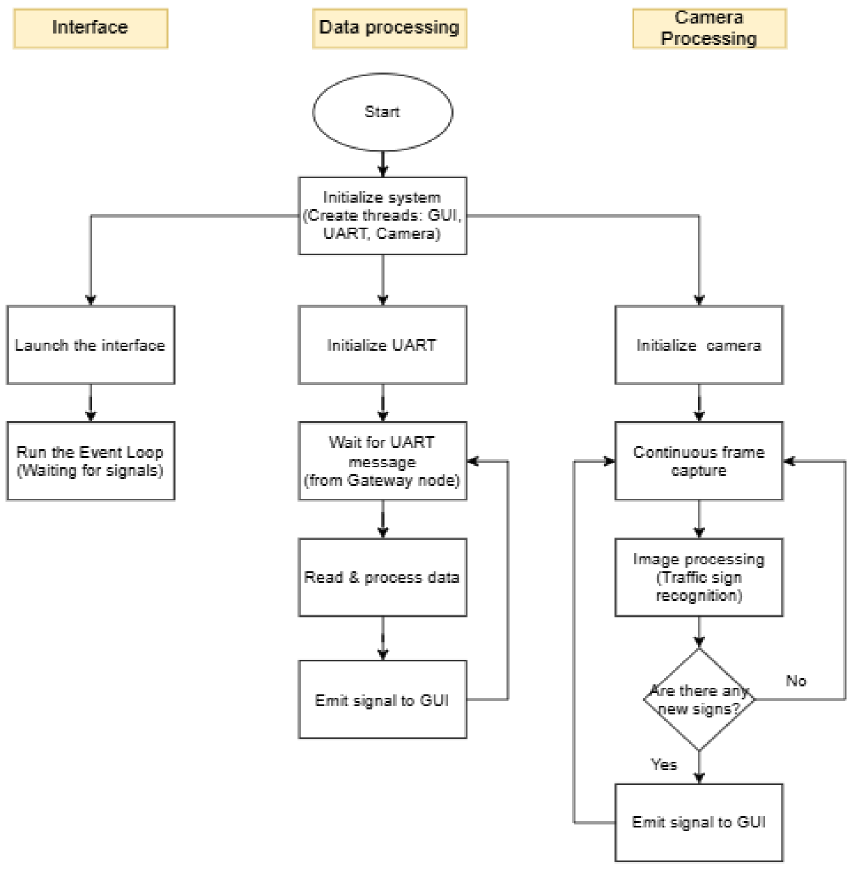

Figure 2.5. Algorithm flowchart for Central Unit

2.3. Hardware design

This section outlines the hardware design process for the system, including selecting components suitable for the project requirements, designing the schematic and printed circuit board, ensuring the system operates stably and is easily expandable.

2.3.1. Component selection

Component selection is based on key criteria including functional capability, stability, scalability, availability, and cost. Components are compared with popular alternatives before a final selection is made for the system.

2.3.1.1. Microcontrollers for control nodes

Table 2.1. Comparing MCUs for control nodes

Criteria	**STM32F103C8T6**	**ATmega328P**	**ESP8266**
Architecture	ARM Cortex-M3 (32-bit)	AVR (8-bit)	Tensilica (32bit)
Operating frequency	72 MHz	16 MHz	80 MHz
Peripheral	Strong, diverse	Limit	Medium
Suitable for real-time control	Excellent	Medium	Medium
Cost	Medium	High	Medium

Select: STM32F103C8T6

Reason: The STM32F103C8T6 has high performance, multiple timers and ADCs, making it suitable for real-time control tasks such as PWM generation, encoder reading, and CAN data processing, thus meeting the project's requirements well.

2.3.1.2. Microcontroller for Gateway node

Table 2.2. Comparing MCUs for Gateway node

Criteria	**ESP32**	**STM32F4**	**Raspberry Pi Pico**
SPI/UART communication interface	Medium	Medium	Medium
Processing capability	High	High	Limit
Multitasking support	Excellent	Medium	Limit
Cost	Medium	High	High

Select: ESP32-WROOM-32

Reason: ESP32 has powerful processing capabilities and supports multiple simultaneous communication interfaces, making it suitable as an intermediary node for data conversion between CAN and UART.

2.3.1.3. Central Unit

Table 2.3. Comparison of embedded computers for the Central Unit

Criteria	**Raspberry Pi 4**	**Raspberry Pi 3**	**Jetson Nano**
CPU Performance	High	Medium	High
Camera Support	Good	Good	Excellent
Display Support	HDMI	HDMI	HDMI
Cost	Medium	Low	High

Select: Raspberry Pi 4 Model B

Reason: The Raspberry Pi 4 performs well in tasks such as image processing, displaying dashboard interfaces, and communicating data with other nodes in the system.

2.3.1.4. CAN communication and sensors

Table 2.4. Comparing devices for CAN communication

Criteria	**MCP2515**	**MCP2551**	**Integrated CAN**
Transmission Standard	CAN 2.0A / 2.0B	CAN 2.0A / 2.0B	CAN
Communication with MCU	SPI	SPI	Internal integration
Anti-interference	Good	Good	Good
Transmission speed	Up to 1 Mbps	Up to 1 Mbps	Up to 1 Mbps
Transmission distance	Long	Long	Long
Compatible with automotive systems	Excellent	Excellent	Excellent

Select: MCP2515 module

Reason: The MCP2515 is a popular solution, acting as a CAN transceiver, converting logic signals to CANH/CANL differential signals. This solution ensures good noise immunity, is suitable for the CAN network model in automotive systems, and is easy to implement and highly stable for this project.

Table 2.5. Comparison of sensor choices

Component	**Alternative**	**Selection**	**Reason**
Distance sensor	IR	HC-SR04	Low cost, easy to use
Velocity measurement	Hall sensor	Optical Encoder	High accuracy, easy to use
Motor driver	L298N	TB6612	Compact, smooth control
Coolant temperature	NTC B3950 10K	DS18B20	High accuracy, stable digital communication

2.3.2. Design the schematic diagram

Based on the overall block diagram and component selection, the hardware system is designed in a modular fashion, where each node performs a separate function. The schematic and printed circuit board designs are created to ensure system stability, ease of assembly, maintenance, and expansion.

2.3.2.1. Body Control node

The BCN schematic centers on the STM32F103C8T6, utilizing GPIOs with pull-up resistors for stable button inputs. An analog joystick connects to ADC1 for control, while the HC-SR04 sensor uses PB7 (Trigger) and PB6 (Echo) for distance measurement. CAN communication is managed by the MCP2515 module via SPI, providing standard CAN_H and CAN_L interfaces. The compact PCB layout isolates signal processing from communication blocks, effectively minimizing interference and facilitating future expansion.

The BCN's printed circuit board is designed with a compact size, and the functional blocks are clearly separated between the signal processing and communication sections, reducing interference and facilitating expansion.

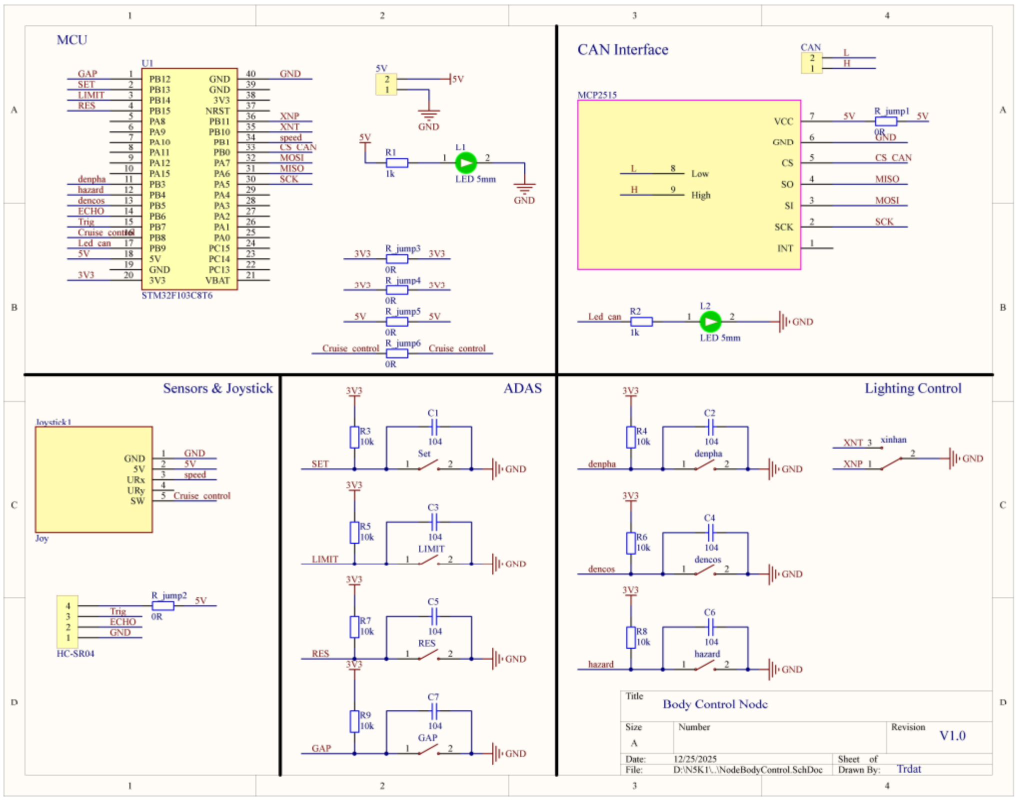

Figure 2.6. Body Control Node schematic diagram

2.3.2.2. Motor Control node

The Motor Control Node is designed to focus on motor control and speed feedback data acquisition. The STM32F103C8T6 microcontroller receives control data from the CAN network and generates PWM signals to control the motor driver.

The motor driver is connected to the microcontroller's PWM pins, regulating the voltage supplied to the 12V DC geared motor. For speed measurement, an optical encoder is connected to the microcontroller's timer in External Clock mode, allowing for accurate pulse counting over a defined period.

Similar to the Body Control Node, the CAN communication block uses the MCP2515, ensuring stable data transmission and reception over the CAN network.

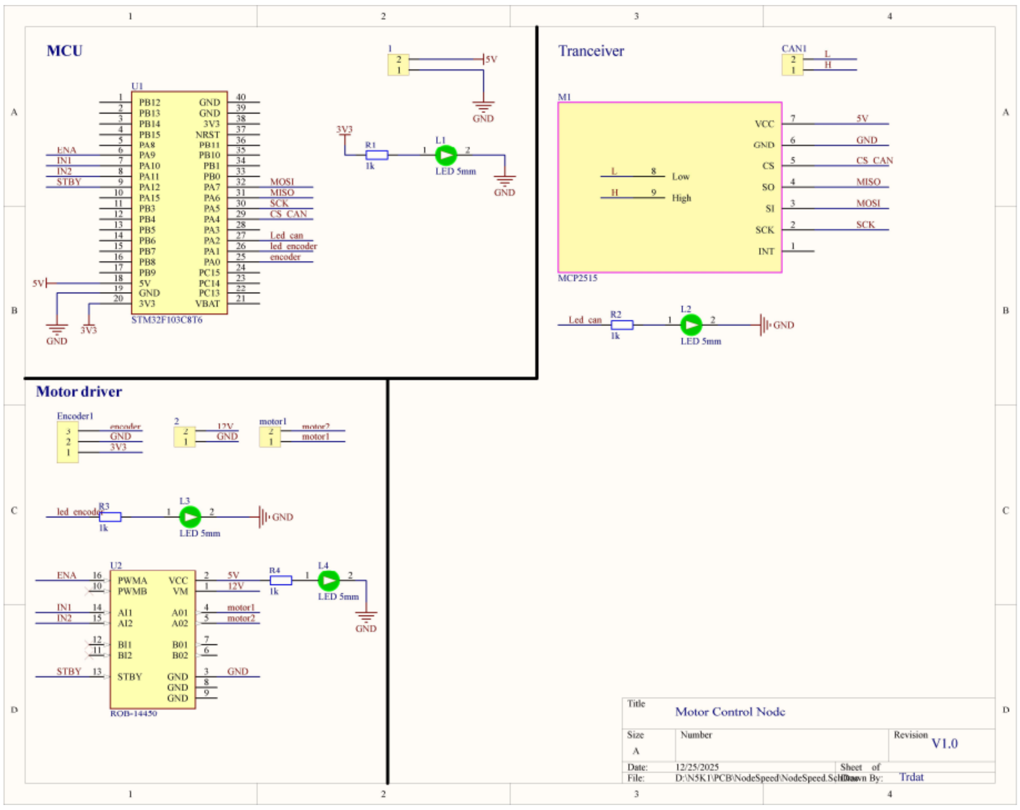

Figure 2.7. Motor Control Node schematic diagram

2.3.2.3. Gateway node

The Gateway Node schematic uses the ESP32-WROOM-32 module as the central microcontroller. The ESP32 communicates with the MCP2515 module via SPI to receive and send data over the CAN Bus network.

Beyond its primary communication duties, the hardware design also incorporates specific interfaces for vehicle function simulation, including control outputs for the lighting system (Headlights, Turn Signals) and a dedicated input for the DS18B20 coolant temperature sensor.

In addition, the Gateway Node uses a UART port to connect to the Central Unit Node, with logic voltage levels designed appropriately to ensure compatibility and safety for the devices.

The Gateway Node's printed circuit board is designed to be compact, prioritizing short SPI and UART connections to minimize electromagnetic interference. Furthermore, the layout clearly displays CAN and UART connectors alongside status LEDs, facilitating convenient system connection and visual troubleshooting during operation.

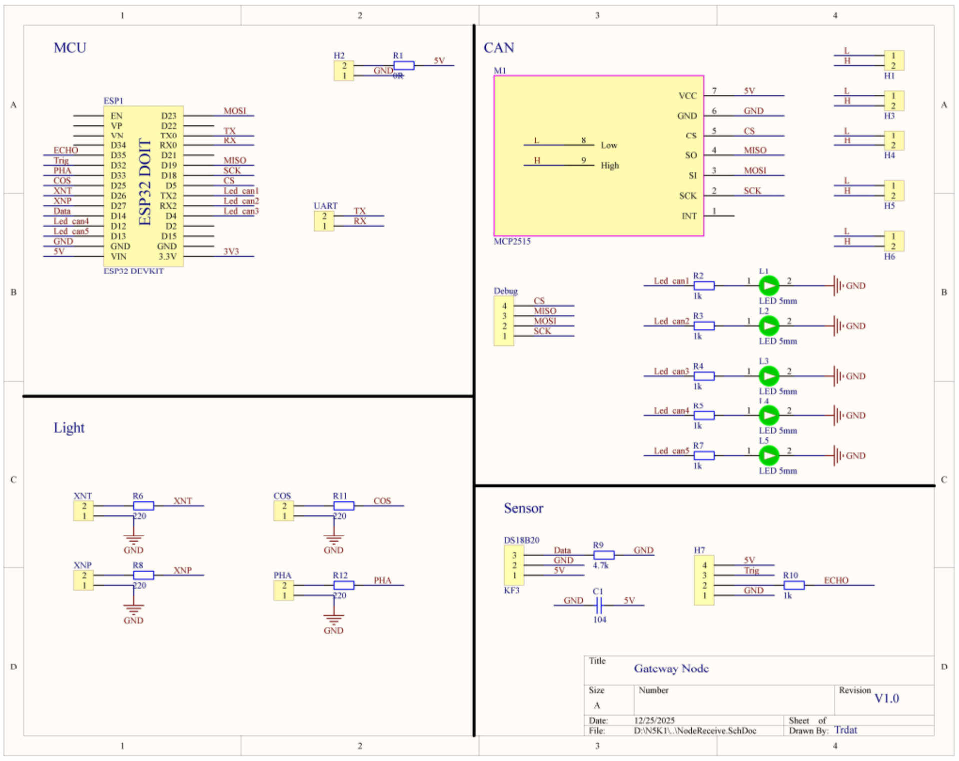

Figure 2.8. Gateway Node schematic diagram

2.3.2.4. Central Unit

The Central Unit uses a Raspberry Pi 4 embedded computer as its central processing unit. Because the Raspberry Pi already has built-in communication ports and a stable power supply, this project does not require a separate printed circuit board design for the Central Unit.

The Raspberry Pi is connected to an LCD screen via HDMI to display the instrument panel interface, and also to a camera to collect images of the front of the vehicle for traffic sign recognition algorithms.

Operational data from other nodes is transmitted to the Central Unit via the Gateway Node using UART communication, allowing the Raspberry Pi to update information on the display interface and process data at a high level.

2.3.2.5. Power supply circuit

The power supply circuit diagram is designed to provide a stable 12V voltage for the motor and 5V for the entire system. The power supply circuit uses a linear power supply structure with an isolation transformer, ensuring electrical safety and reliability during operation.

The AC power after the transformer is rectified full-wave through a diode bridge D1, then filtered by a large capacitance capacitor to create a relatively stable DC voltage. Bypass capacitors are arranged in parallel to reduce high-frequency noise.

The rectified DC voltage is fed into the LM7812 voltage regulator IC to create a 12V power supply for the motor driver. Simultaneously, the LM7805 IC is used to create a 5V power supply for the microcontroller, communication modules, and sensors.

The power supply circuit integrates an input fuse protection, along with LEDs indicating the 12V and 5V power status, making it easy to monitor the system's operating status.

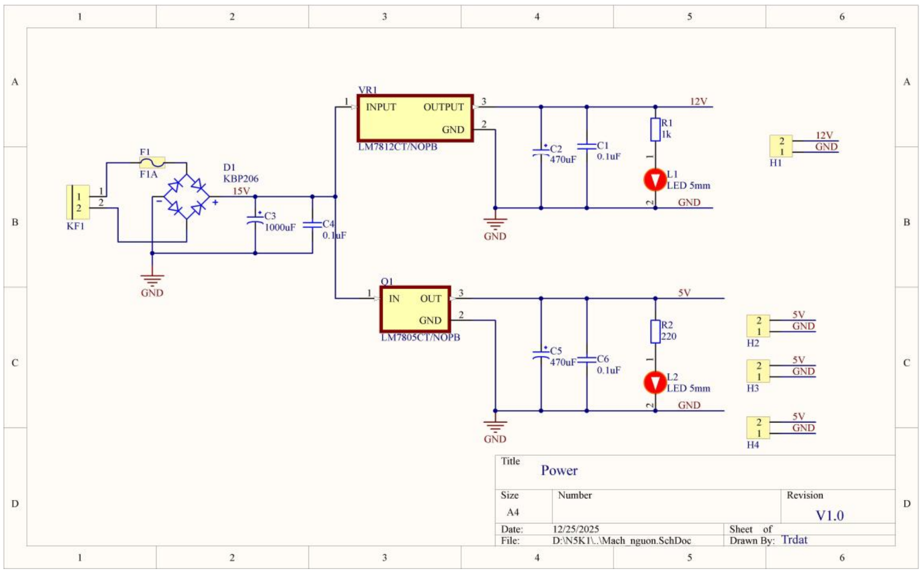

Figure 2.9. Power supply circuit schematic

2.3.3. Printed circuit board design

After completing the schematic diagram, the printed circuit board design process is carried out according to technical standards to ensure machinability and operational stability.

2.3.3.1. Body Control node

This is the control board, housing numerous buttons and input components to simulate driver actions. Its main components include:

- Microcontroller: STM32F103C8T6 module (Blue Pill).
- Communication: MCP2515 CAN module located next to the microcontroller, ensuring the shortest SPI transmission path.
- Joystick: 2-axis joystick assembly used to control the motors.
- Function buttons: Large, clearly defined yellow buttons: Headlights, Low Beam, Turn Signals, and cruise control buttons (SET, RES, GAP, LIMIT).
- Warning: A prominent red button for the hazard lights function.

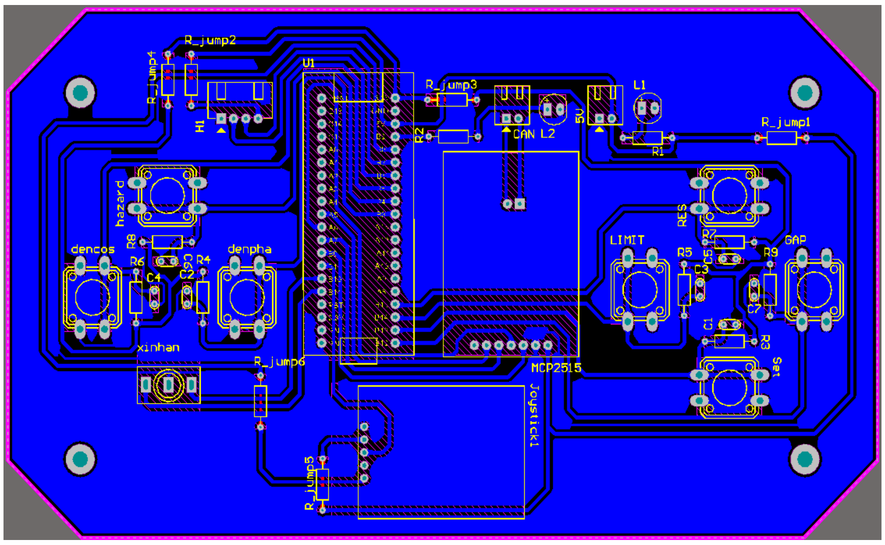

Figure 2.10. Body Control node circuit layout

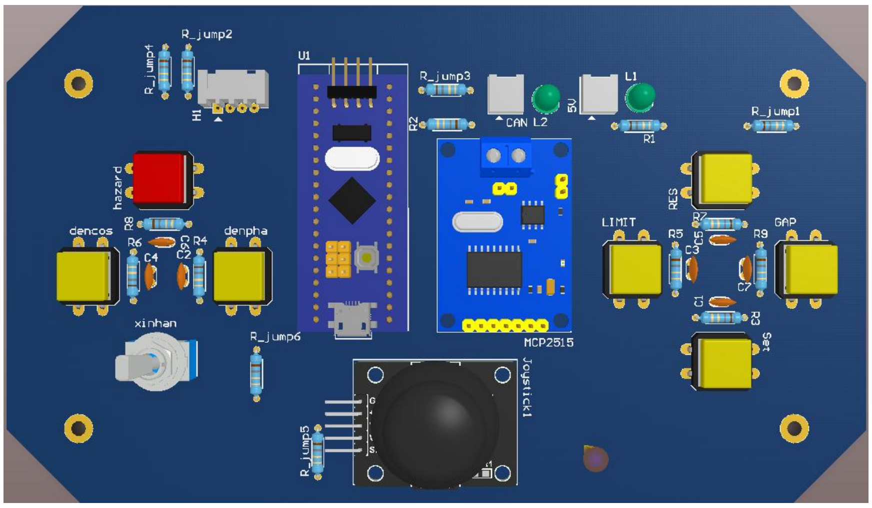

Figure 2.11. 3D simulation of the Body Control node

2.3.3.2. Motor Control node

This circuit controls the motor speed and reads feedback from the encoder. The main components are:

- Microcontroller: STM32F103C8T6 module plugged in.
- Motor driver: TB6612 motor control module, connected to the STM32 via PWM pins.
- Encoder port for reading speed pulses.
- Motor port for powering the motor.
- 12V, 5V, and CAN ports are located nearby for power supply and communication.
- Indicator: Four single LEDs (L1-L4) are integrated to indicate the operating status.

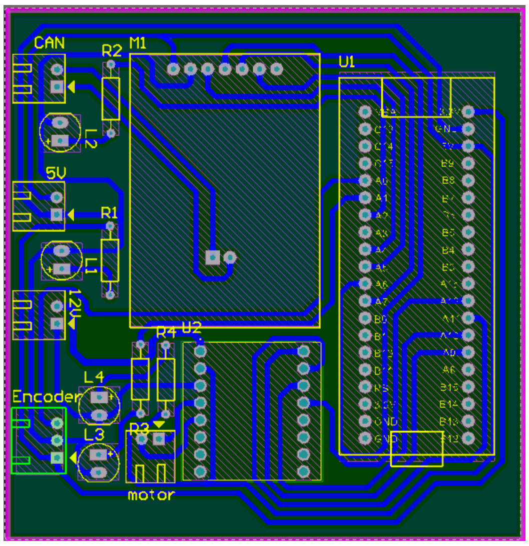

Figure 2.12. Motor Control node circuit layout

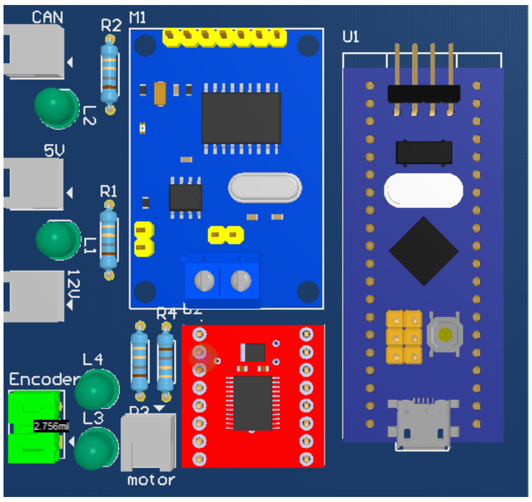

Figure 2.13. 3D simulation of Motor Control node

2.3.3.3. Gateway node

Unlike designs that simply convert data, the Gateway circuit in this model integrates additional functions for controlling the lighting system and monitoring coolant temperature.

- Central processing unit: ESP32-WROOM-32 module.
- CAN communication: MCP2515 module connected to the vehicle's CAN network.
- Light control: The left-hand header connector array is specifically labeled: PHA, COS, XNT (Left turn signal), XNP (Right turn signal). This indicates that the ESP32 directly controls these light clusters based on the received signals.
- Integrated DS18B20 temperature sensor and pre-wired terminal block allow for monitoring of coolant temperature.
- Indicators: A row of 5 LEDs (L1-L5) indicates the CAN connection status.

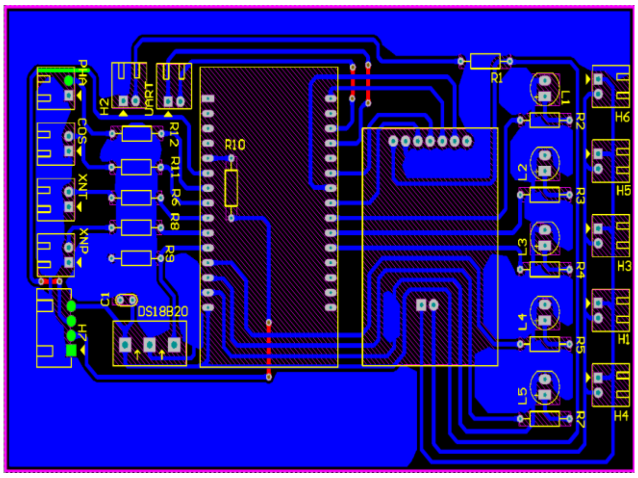

Figure 2.14. Gateway node circuit layout

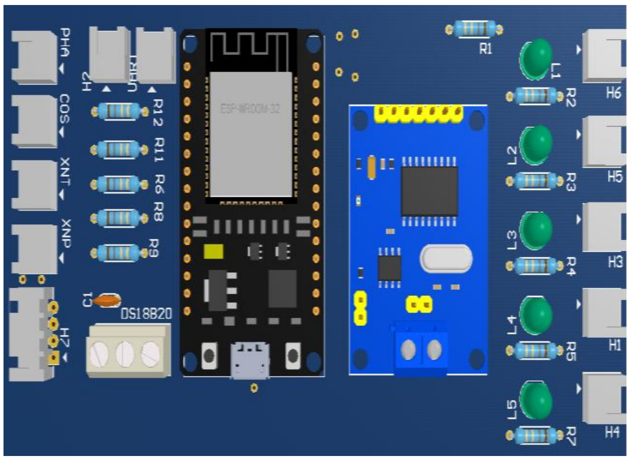

Figure 2.15. 3D simulation of Gateway node

2.3.3.4. Power supply circuit

To ensure the safety of expensive modules, a separate power supply circuit is designed. The circuit incorporates a tubular fuse at the input to interrupt the circuit in case of overcurrent/short circuit. Large-capacity electrolytic capacitors (C2, C3, C5) are used in combination with ceramic capacitors to smooth the voltage. Output ports (H1-H4) distribute stable power to the functional nodes (Motor, Body, Gateway), allowing for neat wiring from a central point.

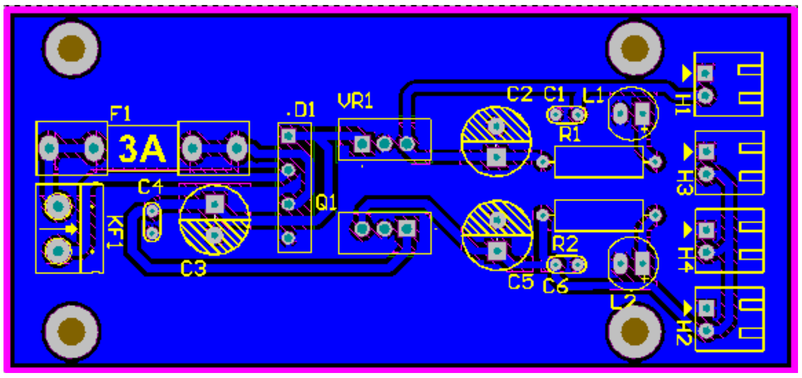

Figure 2.16. Power supply circuit layout

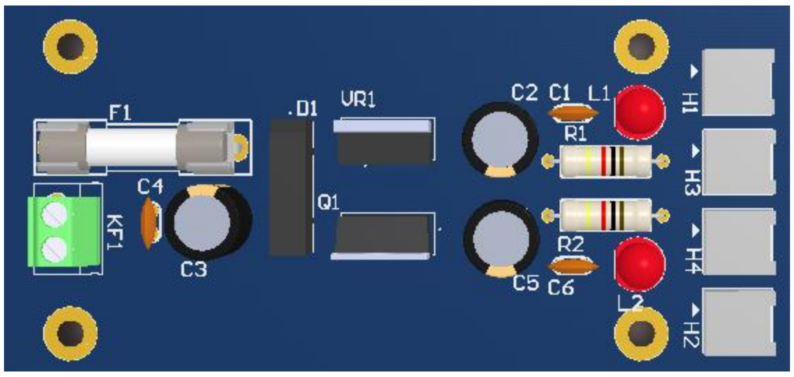

Figure 2.17. 3D simulation of the power supply circuit

2.4. Software design

The system's software is designed with a modular approach, ensuring clarity, scalability, and ease of maintenance. Key functions include image processing, system control, and user interface display.

2.4.1. Image processing function design

The image processing function is implemented in the Central Unit, using a Raspberry Pi 4 Model B embedded computer in conjunction with a camera to collect image data from the environment in front of the vehicle. The goal of the image processing unit is to recognize traffic signs and provide information to the control system and display it on the interface.

2.4.1.1. Develop an identification model

The traffic sign recognition model is built on the MobileNetV2 convolutional neural network architecture. This architecture is optimized for mobile and embedded devices due to its balance between accuracy and processing speed.

To enhance the model's generalization capabilities and avoid overfitting, data augmentation techniques are applied. From the original dataset, new variants are generated through random transformations:

- Rotation: ± 15°.
- Shift: Horizontal and vertical shift by 10%.
- Shear and Zoom: 10%.
- Brightness: Between 0.9 and 1.1

The training process uses Transfer Learning techniques with weights from ImageNet. The network structure was customized by removing the original Fully Connected layers and replacing them with the following layers: GlobalAveragePooling2D, Dense (128 units, ReLU activation), and Dropout (0.3) to minimize overfitting.

- Loss function: Categorical Crossentropy.
- Optimization: Adam Optimizer with a mechanism to reduce learning speed when Loss does not improve.
- Model conversion: After training, the model is converted to TensorFlow Lite (.tflite) format and quantization is applied to reduce size and increase inference speed on Raspberry Pi.

2.4.1.2. Image prepprocessing

Before being fed into the AI model, the images captured from the camera need to undergo a preprocessing step to filter noise and identify the location of potential signs. This process reduces the load on the CPU by removing unnecessary background areas.

Using the Picamera2 library on Raspberry Pi, images are captured in RGB888 format with a resolution of 320x180 pixels, ensuring a high frame rate. The color space is then converted from BGR to HSV. The system applies a color filter to separate red objects (characteristic of prohibition and danger signs). The red color range is defined in two HSV ranges:

- Range 1: H [0, 10], S [100, 255], V [100, 255].
- Range 2: H [160, 180], S [100, 255], V [100, 255].

After applying a color filter to identify the area with red color band, use open morphology with an elliptical (5x5) kernel to remove small noise points on the binary mask. The found outlines will then be checked for geometric conditions to determine if they are signs:

- Area: Area ≥ 1500 pixel.
- Aspect Ratio: $0.7 \leq \frac{width}{height} \leq 1.5$.
- Pixel Ratio: The percentage of red points in the ROI region must be at least 30%.
- Circularity: For Stop signs (octagonal/circular), the system additionally checks for roundness based on the following recipe:

$$Circularity = \frac{4\pi\ x\ arera}{perimeter^2}$$

- This value must be greater than 0.6 to be accepted.

2.4.1.3. Traffic sign recognition

After identifying the region of interest (ROI) containing potential traffic signs, the recognition process proceeds as follows:

- Crop and Normalization: The ROI is cropped from the original frame, resized to the model's input standard (192x192 pixels), and the pixel values are normalized to the range [0, 1].
- Inference: tflite_runtime is used to predict the traffic sign class.
- Result Validation: The prediction result is only accepted when the confidence level is greater than the established threshold (0.85).
- Recognition Stability Algorithm: To avoid "flickering" (inaccurate recognition in a split second), the system requires a traffic sign to be recognized identically in 4 consecutive frames before making a final decision. This increases the reliability of traffic sign recognition, helping the model avoid misidentification even when the sign is partially visible within the frame.

2.4.1.4. Processing results and data transmission

After authentication, the identification results are converted into control commands and transmitted via the UART protocol. Labels are mapped to corresponding control command codes (speed_limit_40 → CMD_LIMIT_40, stop → CMD_STOP,…).

The system sends packets to the Gateway node via the /dev/ttyUSB0 port with a baud rate of 115200. The packet format is: CMD_NAME,Confidence. The system sets a timeout (SIGN_HOLD_TIME = 2s) to prevent the continuous sending of the same control command, helping the vehicle system operate more stably.

2.4.1.5. Synchronization with the display interface

To allow the driver to monitor the status, the image processing system synchronizes data with the user interface. UDP sockets are used to transmit internal data to the interface. The data flow is represented as follows:

- Sending information on the newly identified traffic sign: AI:{Class_ID}.
- Bridging function: The system also reads sensor data from the UART (sent by the vehicle controller) and forwards it via UDP to the display interface, ensuring real-time synchronization between the hardware and the display software.

2.4.2. Control module design

The embedded control module is developed on the STM32CubeIDE platform, using the HAL library to interact with the hardware. The software architecture is designed using an infinite loop model combining functional elements, ensuring simplicity while still meeting real-time requirements.

2.4.2.1. Body Control node algorithm

The main task of this node is to convert physical signals from the driver into digital data to send over the CAN network.

Due to the nature of joystick potentiometers often being susceptible to signal interference, the system applies an Average Filter algorithm. The microcontroller performs 5 consecutive ADC sampling cycles, then calculates the average value to eliminate random noise spikes before use. A dead zone of approximately ±15 units is set around the center value to prevent the vehicle from rolling when the joystick is in the resting position.

The entire button state, joystick value, and distance are packaged into a data frame with the identifier ID 0x01. The data transmission cycle is 50ms to ensure smooth operation.

2.4.2.2. Motor Control node algorithm

This node has the most complex processing logic, including motor control and feedback measurement. Control data (from the joystick or AI command) is mapped to PWM pulse width. Timer 1 is configured to output a 1kHz PWM pulse. The pulse width varies from 0 to 624, corresponding to motor power from 0% to 100%.

The rotation direction control logic is programmed to prioritize safety: In case of sudden reversal, the system will reset the PWM to 0 for a short period to protect the gearbox and driver.

Timer 2 is used in External Clock Mode to count the pulse edges from the encoder. Due to the characteristics of brushed motors which generate significant electromagnetic interference, a Moving Average algorithm with a window size of N=10 is applied. The instantaneous velocity is the average of the 10 most recent measurements.

2.4.2.3. Gateway node algorithm

The Gateway node facilitates communication between two asynchronous communication protocols.

CAN to UART stream: Uses interrupts to capture CAN packets as soon as they appear on the transmission path. The CAN payload data is decoded and reformatted into a JSON string and then sent via the Serial port to the Raspberry Pi.

UART to CAN stream: Listens to the Serial buffer. When it receives a command string from the Raspberry Pi (e.g., from the image processing module: CMD_STOP), the ESP32 parses it to determine the command type.

Based on the command type, it creates a corresponding CAN frame (e.g., ID 0x03 for the emergency braking command) and pushes it down the CAN bus for the Motor Node to execute.

2.4.2.4. Central node algorithm

The Central Node is the central processing unit of the system, responsible for receiving data from peripheral nodes, processing high-level information, and updating operational status in real time. The algorithm at the Central Node is designed using a multi-threaded model, allowing communication, image processing, and interface display tasks to be performed in parallel to ensure system performance and stability.

A dedicated UART communication thread continuously listens for data from the Gateway Node, decoding and analyzing status information to synchronize with the system. Simultaneously, the image processing thread collects frames from cameras and uses the TensorFlow Lite model to recognize traffic signs, thereby generating appropriate control commands. These commands are sent back to the Gateway Node via UART to be transmitted to lower-level control nodes.

Furthermore, the interface thread is responsible for updating data on the dashboard display through a linking mechanism between the Backend and Frontend, allowing the interface to respond instantly to new data without affecting other processing threads.

Thanks to its independent multi-threaded architecture, the Central Node ensures efficient parallel processing, reduces system latency, and meets real-time requirements in intelligent driving assistance applications.

2.4.3. Display interface design

The system's user interface was designed and built using Qt Creator, employing QML. Designing the interface using pure QML programming allows for layout flexibility, easy scalability, and suitability for embedded systems utilizing display screens.

2.4.3.1. Interface architecture

The interface is built on a modular model, where each interface component is packaged into a separate QML file. The main.qml file acts as the main interface, responsible for window initialization, overall layout management, and calling child components. Some of the main interface modules include:

- Speedometer.qml: Displays the speedometer.
- Fuel_gauge.qml: Displays the fuel level.
- Coolant.qml: Displays the coolant temperature.
- CruiseControl.qml: Displays the Cruise Control system status.
- Function.qml: Displays function icons and warnings.

This organization makes the interface clear, easy to edit, and convenient for reusing components.

2.4.3.2. Graphic design and display

The graphical elements of the interface, such as the dashboard background, hands, function icons, and warning icons, are built using a modular approach, where each functional group is designed as a separate QML module. Inside each module, the graphical elements are created from static images and manually loaded via the Image object in QML.

Organizing icons and display elements into independent modules makes the interface easy to manage and convenient for editing individual parts without affecting the overall structure. At the same time, these modules can be reused or extended when adding new display functions.

The interface uses layout properties such as anchors to align positions, scale to adjust the display ratio, and rotation to create rotational movement for dynamic elements such as the clock hands. Display parameters are calculated based on the main window size, allowing the interface to adapt to different screen resolutions while maintaining aesthetics and intuitiveness.

2.4.3.3. Data linking and state updates

To connect data between the central processing unit and the display interface, the system uses an middleware layer built in Python (PyQt5). The backend program is implemented on a Raspberry Pi, responsible for launching the QML interface and receiving and distributing data from other processing units in the system.

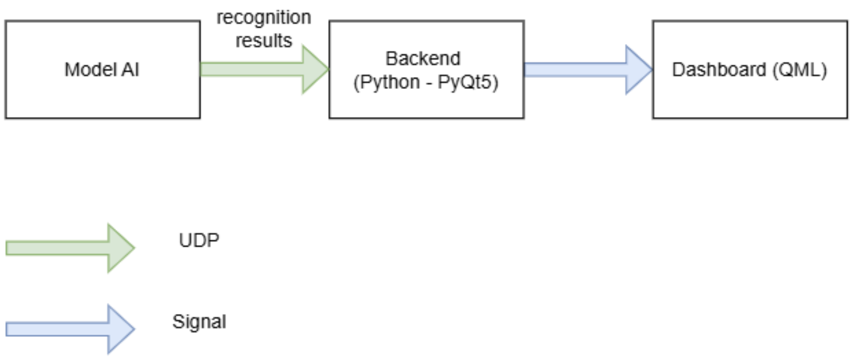

Figure 2.18. Data flow diagram

The interface utilizes QQmlApplicationEngine with main.qml, embedding a Python backend via Context Properties for direct interaction. To minimize latency, a dedicated thread listens for real-time UDP data, which is then processed and emitted via signals to update QML components. This architecture ensures asynchronous UI updates without blocking the main thread, effectively decoupling data processing from visual rendering for enhanced performance and scalability.

2.4.3.4. Effects and interactions

To enhance visual appeal, the interface incorporates motion effects and animations such as:

- Clockwork rotation effect based on velocity value.
- Transition and blurring effects when the display state changes.
- Visual feedback when activating functional modes.

The effects are built using standard QML components such as Behavior, NumberAnimation, and Easing, making the interface lively while maintaining performance.

2.4.3.5. Scalability and integration

The modular QML-based interface design allows for easy integration with other processing units in the system, especially the image processing and control units. The interface can be extended to display additional information such as traffic sign recognition status, operational data, or system warnings without changing the overall structure.

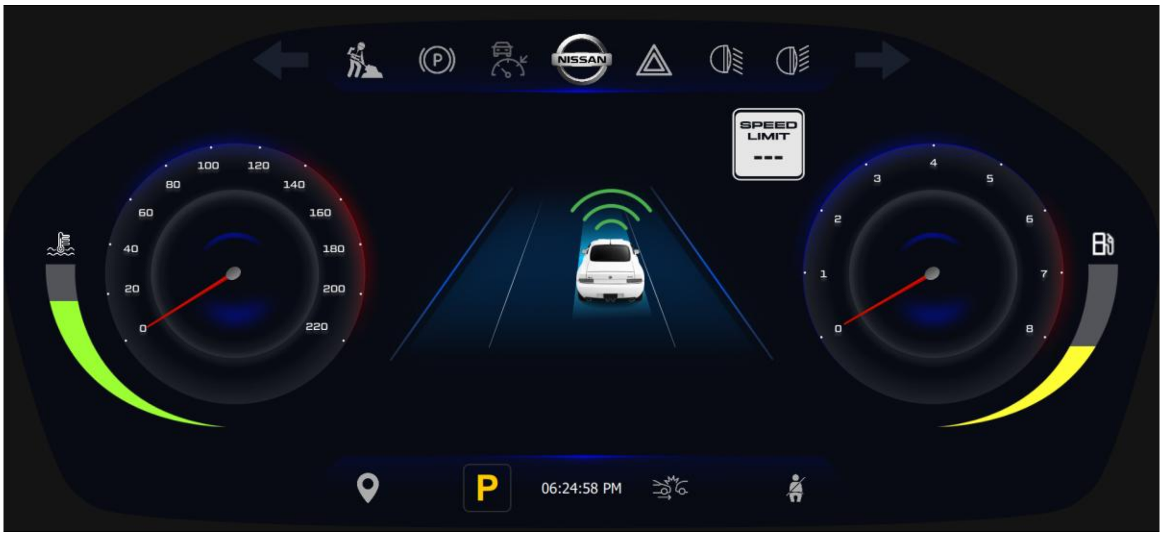

Figure 2.19. Instrument cluster interface

2.5. Configure communication protocols

To ensure accurate and stable data exchange between modules in the system, the CAN, UART, and UDP communication protocols are configured with appropriate data formats and transmission parameters. Specifically, the configuration process focuses on standardizing baud rates, frame structures, and error-checking mechanisms to maintain data integrity across heterogeneous hardware platforms. This approach not only optimizes bandwidth usage but also guarantees that critical control signals and visual feedback are synchronized in real-time. This section details how to format packets and configure communication for each protocol used in this project.

2.5.1. CAN packet format

The CAN protocol is used to transmit control and status data between embedded nodes in the system, configured at a speed of 250kbps, using the Standard Frame 2.0A format. The system defines two main packets for exchanging control and status data.

2.5.1.1. Body Control node packet structure

Sender: Node Body Control (STM32F103)

Receiver: Node Speed (STM32F103), Node Gateway (ESP32)

Sending period: 50m

Table 2.6. Body Control Node data structure

Byte index	Data name	Detailed description
Byte 0	Button Flags	Status of ADAS buttons
		Bit 0: Cruise Activate
		Bit 1: SET/-
		Bit 2: RES/+
		Bit 3: GAP
		Bit 4: LIMIT
Byte 1	Light Flags	Lighting system status
		Bit 0: High Beam
		Bit 1: Low Beam
		Bit 2: Left turn signal
		Bit 3: Right turn signal
		Bit 4: Hazard
Byte 2	Joystick low	The lower 8 bits of the Joystick value (ADC)
Byte 3	Joystick high	The 8 high bits of the Joystick value (ADC)
Byte 4	Distance low	8 low bits of distance
Byte 5	Distance high	8 high bits of distance
Byte 6	Gap level	ACC safe distance level
Byte 7	Target speed	The desired speed is calculated by ADAS.

2.5.1.2. Motor Control node packet structure

Sender: Node Speed (STM32F103)

Receiver: Node Gateway (ESP32)

Sending period: 50ms

Table 2.7. Motor Control Node data structure

Byte index	Data name	Detailed description
Byte 0 - 3	Encoder Freq	Encoder pulse frequency (32-bit integer), used for debugging raw signals.
Byte 4	Speed Low	8-bit high of real speed.
Byte 5	Speed High	8-bit low of real speed.

2.5.2. UART configuration

In the Node Gateway design, the UART protocol acts as the primary gateway for transmitting data collected from the CAN network to the Central Unit. The ESP32 microcontroller is configured using a HardwareSerial (UART0) to ensure maximum stability.

Communication parameters are set with a Baud rate of 115200 bps to meet the requirements for rapidly transmitting large amounts of data from sensors and vehicle status. The frame format is defined as: 8 bits of data, no parity check, and 1 stop bit. Thanks to the ESP32's GPIO Matrix feature, the transmit (TX) and receive (RX) pins are dynamically mapped to the printed circuit board design, and the system also uses a large buffer to prevent data overflow during parallel wireless network processing.

2.5.3. UDP configuration

To ensure image processing performance and display interface smoothness operate independently, the system uses the UDP protocol as an internal communication channel (Local Inter-process Communication). This method completely separates the data processing stream from the display stream, preventing interface freezing when the system handles heavy tasks.

2.5.3.1. Send-Receive architecture model

The system is divided into two independent processes running in parallel on the embedded computer:

- The central processing (Backend - Sender) process is handled by the final.py module. Its function is to read sensor data from the ESP32 microcontroller via the UART port and run the traffic sign recognition algorithm. After processing, the data is encapsulated and sent via the UDP socket to Loopback address 127.0.0.1 on port 5005.
- The display process (Frontend - Receiver): Built on the Qt framework in the Dashboard_main.py module. Its function is to continuously listen on port 5005

on a separate thread to update parameters on the Instrument Cluster screen without interrupting the main graphics rendering thread.

2.5.3.2. Packet structure

Specifically, using a string-based payload avoids the complexity of binary serialization and endianness handling between different processing units. Although binary transmission is theoretically more bandwidth-efficient, the overhead of parsing ASCII strings on a local loopback interface (127.0.0.1) is negligible for modern embedded processors.

This format also facilitates real-time monitoring; developers can simply use a network sniffer tool or print the raw socket data to the terminal to verify the integrity of the sensor values before they are rendered on the Qt interface. The parsing logic on the receiver side utilizes efficient string-splitting algorithms to separate data fields based on predefined delimiters, ensuring that the UI update rate remains synchronized with the backend processing speed.

Table 2.8. Internal UDP communication packet format

Data type	Syntax	Data source	Description
Status	SPD:x, PHA:y, …	ESP32 (UART)	Operational data received from the microcontroller is relayed intact.
AI warning	AI:[Class_ID]	Raspberry Pi	The result of traffic sign recognition (e.g., AI:5 corresponds to the STOP sign).

CHAPTER 3. SYSTEM IMPLEMENTATION AND EVALUATION

3.1. System deployment process

3.1.1. Development tools and environment

To realize the system from design drawings to a real product, the implementation process uses a combination of specialized hardware and software tools for embedded systems and image processing.

3.1.1.1. Software and programming enviroment

Altium Designer: Used to design schematics and print circuit boards for control nodes and gateway node.

STM32CubeMX: A graphical tool used to visually configure the STM32. The software supports setting up clock diagrams, assigning functions to GPIO pins, and configuring parameters for the ADC, UART, and Timer communication modules. After configuration, the tool automatically generates initialization code compatible with the CMSIS standard for porting to the Keil C environment.

Keil µVision 5: The primary integrated development environment for the STM32 microcontroller family. It uses an ARM C/C++ compiler to write low-level control firmware for SPI, ADC, Timer, and GPIO peripherals.

Arduino IDE: Used for developing firmware for Node Gateway. This environment provides powerful libraries supporting FreeRTOS and network protocol stacks, helping to shorten application development time.

Visual Studio Code: A versatile source code editor used for software development on Raspberry Pi.

Qt/QML environment: Supports building user interfaces using the PyQt5 library.

RealVNC Viewer: Allows remote access and control of the Raspberry Pi without a separate monitor.

Putty: A computer terminal used to monitor debug data via the Serial/UART port. Additionally, this software is used to configure the initial functions of the Raspberry Pi.

WinSCP: An open-source file management software using the SFTP/FTP protocol. During development, WinSCP played a crucial role in quickly transferring data, copying source code, and updating AI models from a personal computer to the Raspberry Pi's Linux operating system over the network.

3.1.1.2. Hardware tools and debug loading

ST-Link V2 Programmer: Used for programming and real-time debugging of the STM32F103 microcontroller.

Logic Analyzer: A tool used to check and validate CAN packets. The device connects to CAN_H and CAN_L on the transmitting device to check what data is being sent. Through analysis software on a computer, low-level signals are decoded into CAN data frames and error flags.

3.1.2. System integration process

The hardware deployment process of the system is carried out according to a systematic procedure, including steps from ideation, design, testing to assembly and evaluation, to ensure the correctness, stability and scalability of the project.

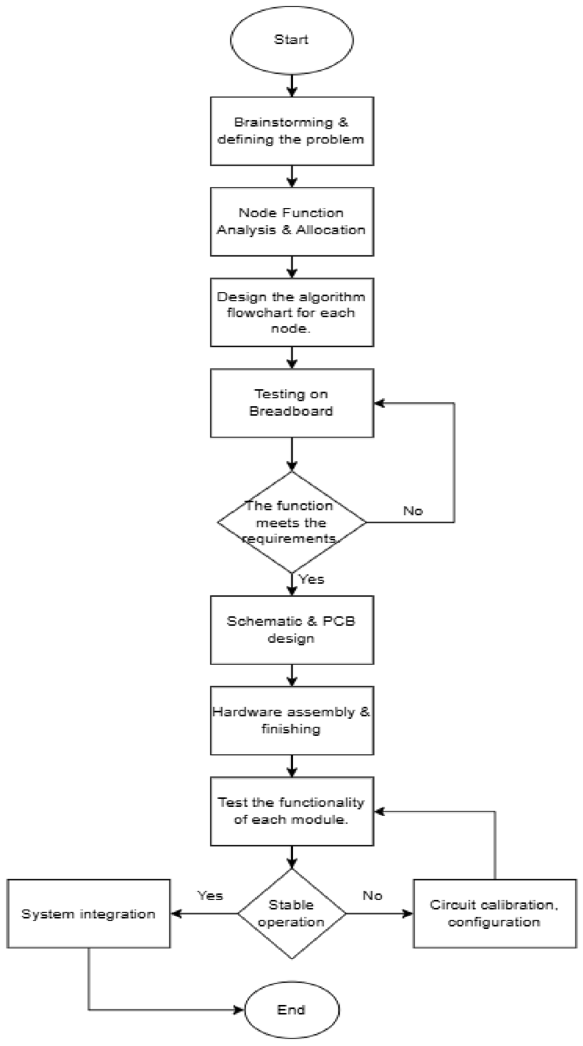
Start
Brainstorming & defining the problem
Node Function Analysis & Allocation
Design the algorithm flowchart for each node.
Testing on Breadboard
The function meets the requirements
No
Yes
Schematic & PCB design
Hardware assembly & finishing
Test the functionality of each module.
Stable operation
Yes
No
System integration
Circuit calibration, configuration
End

Figure 3.1. System deployment process flowchart

Step 1: Ideation and Problem Definition

Based on the requirement to simulate a control and information display system in an automobile, the project is oriented towards building a distributed embedded system where functions are divided into multiple independent nodes, communicating with each other via a CAN Bus network. The system must ensure the ability to control, monitor, and display data in real time.

Step 2: List and Analyze System Functions

Based on the project requirements, the main functions of the system are identified and allocated to each node, including:

- Collecting control signals from the user (Body Control Node).
- Motor control and speed feedback (Motor Control Node).
- Communication and data conversion intermediary (Gateway Node).
- Image processing, sign recognition, and interface display (Central Unit).

This functional division makes the system highly modular, easy to expand, and maintain.

Step 3: Designing Algorithm Flowcharts for Each Node

Based on the defined functions, algorithm flowcharts for each node are constructed to clearly describe the data processing sequence, operating conditions, and communication flow between modules. These flowcharts serve as the basis for deploying embedded software and testing the system's operational logic.

Step 4: Test the principle on a breadboard

Before designing the printed circuit board, the hardware modules are tested on a breadboard. Components such as microcontrollers, sensors, CAN modules, motor drivers, and encoders are temporarily connected to:

- Check the functionality of each component.
- Evaluate the communication capabilities between modules.
- Detect connection errors or signal conflicts early.

Step 5: Design the schematic and printed circuit board:

After successful testing on the breadboard, detailed schematic diagrams of each node were designed using specialized software. Based on these schematics, the printed circuit board was designed to ensure compactness, voltage stability, noise reduction, and ease of assembly.

Step 6: Assembling Components and Completing the Hardware

The components are soldered and assembled onto the printed circuit board according to the designed diagram. The assembly process is carried out carefully to ensure the mechanical and electrical reliability of the system.

Step 7: Testing and Evaluating the Functionality of Each Module

After assembly is complete, each module is tested independently to verify:

- Correct functionality as designed.
- Stable communication via CAN Bus, UART, and related protocols.
- No power failures, interference, or data loss.

3.2. Software deployment

3.2.1. Programming control nodes

The control software for the Node Body and Node Motor was developed on the STM32F103 microcontroller platform, using the C programming language in the Keil µVision 5 integrated development environment. Because the hardware architecture uses the MCP2515 module for CAN bus communication, the software is built based on the SPI communication protocol instead of using the built-in CAN module on the chip.

The processing workflow and software structure are organized into functional subsystems as follows:

3.2.1.1. Peripheral Configuration

The microcontroller's hardware resources are configured via the HAL and STM32CubeMX libraries to serve specific functions:

SPI Communication: Since the system uses a separate CAN controller, the SPI1 block is configured in Master mode with a high baud rate to communicate with the MCP2515 IC. The SCK, MISO, and MOSI signal pins are configured accordingly, along with the Chip Select pin, which is manually controlled by GPIO to ensure compliance with the module's clock pattern.

Timer/PWM: Timer (TIM1/TIM2) is configured in PWM Generation mode. The frequency and duty cycle are calculated to control the motor driver, allowing for smooth changes in DC motor speed through pulse width adjustment.

GPIO Input: Configures the push-button status reading pins (Pull-up/Pull-down) and the ultrasonic feedback signal input pins from the sensor.

GPIO Output: Controls basic peripherals such as status indicator lights (LEDs), trigger pins from ultrasonic sensors, and motor control trigger pins.

3.2.1.2. Building the CAN communication layer

To increase flexibility, code reuse, and ease of maintenance, CAN communication is not written directly in the main program but is packaged into a separate layer. This layer performs the core functions:

- Hardware abstraction: Provides functions such as CANSPI_Initialize, CANSPI_Transmit, and CANSPI_Receive. The application layer only needs to call these functions without having to worry about manipulating SPI registers or complex handshake procedures with the MCP2515 chip.
- Data encapsulation: Control data (speed, light status) is encapsulated in the uCAN_MSG data structure. This structure includes packet ID, data length, and a data array, helping to standardize the message format between nodes.
- Filtering and Masking Mechanism: At the initialization function, the filter registers of the MCP2515 are loaded with values to allow only packets with valid IDs into the receive buffer, reducing the interrupt processing load on the CPU.
- Data Flow Management: Handles the checking of the transmit/receive buffer status, ensuring that data is sent when the transmission line is free and is correctly decoded when data arrives.

3.2.1.3. Application control logic

At the Motor Control Node: The microcontroller decodes the received CAN message to extract the desired speed value, then uses a timer to output PWM pulses to control the H-bridge circuit, and simultaneously reads feedback from the encoder to monitor the actual speed.

At the Body Control Node: The software continuously scans the status of the digital input signals (light switch, turn signal). When a state change occurs, the microcontroller packages the data into a CAN frame and sends it, ensuring minimal signal delay.

3.2.2. Gateway logic implementation

The gateway is the central node responsible for forwarding data between the CAN Bus network and the Raspberry Pi central processing unit. To ensure stability and avoid data bottlenecks, the software for the ESP32 is designed based on the FreeRTOS real-time operating system.

3.2.2.1. Multitasking software architecture

Instead of using traditional sequential infinite loops, the program is broken down into independent tasks, running in parallel using FreeRTOS's scheduler. This allows the system to prioritize processing critical events immediately without being hindered by slower tasks (debug on Serial).

3.2.2.2. Task design

Task 1: CAN Data Collection (TaskCANReceive - Highest Priority)

- Function: Responsible for SPI communication with the MCP2515 module to check and read data from the CAN Bus network.
- Characteristics: Assigned the highest priority to ensure no packets from the system are missed. As soon as data is received, this Task will take over the CPU to process it, update the status in shared memory, and release the CPU immediately afterward.

Task 2: System Logic Processing (TaskControl - Medium Priority)

- Function: Implements local control logic at the Gateway such as: creating blinking rhythms for turn signals/Hazards, controlling the Timeout time of child Nodes to detect connection loss.
- Characteristics: Runs periodically with a short cycle (10ms) to ensure smooth visual feedback.

Task 3: UART Communication (TaskSerial - Lowest Priority

- Function: Periodically read aggregated data from shared memory, package it into a character string, and send it to the Raspberry Pi via the UART port (Baudrate 115200).
- Characteristics: Because sending data via UART takes a long time to process, this task is given the lowest priority to avoid interrupting the CAN message stream.

3.2.2.3. Data synchronization and protection mechanisms

Because multiple tasks access a shared data area, the system uses a Mutex locking mechanism to ensure data integrity and prevent Race Condition situations. For example, if Task CAN is writing speed data and Task Serial is reading that data to send it, it can lead to information errors. To solve this problem, before writing or reading, the task must obtain the "key" (Semaphore Take). If the resource is busy, the task will enter a waiting state until the resource is freed.

3.2.2.4. Data processing procedure

Collection: The CAN task receives packets from the MCP2515, analyzes the ID, and updates the corresponding fields in SystemState (e.g., stateLeft, stateRight, distance, speed).

Processing: The Control task reads SystemState to control the GPIO pins.

Forwarding: The Serial task reads SystemState, formats it into a string (e.g., "COS:ON| Gap:3"), and sends it to the Raspberry Pi to display the instrument cluster.

3.2.3. Deploy the central processing unit

The software on the Raspberry Pi acts as the system's computing brain, responsible for handling computer vision-intensive tasks and data routing. The program is developed in Python, leveraging libraries optimized for the ARM architecture such as TensorFlow Lite Runtime and OpenCV.

3.2.3.1. Building a traffic sign recognition algorithm

To balance accuracy and processing speed on embedded hardware, the system does not apply neural networks to the entire frame but uses a Hybrid Approach consisting of two stages:

Phase 1: Proposing Areas of Interest:

- Using the OpenCV library to convert the color space from BGR to HSV, separating the sign colors (red/blue) from the environment without being significantly affected by brightness.
- Applying morphological filters to remove noise and search for edges.
- Filtering candidates based on geometric features: Minimum area, aspect ratio, and roundness. Only image regions that satisfy the physical conditions of the sign are cropped and moved to the next phase.

Phase 2: AI Classification:

- Potential image regions are normalized in size and fed into a quantized MobileNetV2 SSD model. Using a quantized model significantly reduces model size and increases inference speed compared to the original model.
- Output results are only accepted if the confidence level exceeds 0.85.

3.2.3.2. Signal stabilization technique

To avoid intermittent recognition results due to camera interference or changing lighting conditions, the system applies a sequence validation algorithm. A sign is only validated if the AI classification result remains consistent for 4 consecutive frames (CONSECUTIVE_FRAMES_REQ = 4). This eliminates transient false recognition results and provides a more stable display interface.

3.2.3.3. Data bridge mechanism

Besides image processing, the Python program also acts as a data routing center to link system components. It initializes the Serial port (/dev/ttyUSB0) with a Baud rate of 115200 bps to listen for raw data from the Node Gateway. It uses non-blocking read or buffer checking to ensure that reading vehicle data does not slow down the image processing stream.

All processed data (including license plate ID from AI and vehicle parameters from UART) is packaged into string messages. Using a UDP socket, the data is sent to the Localhost address (127.0.0.1, Port 5005). This method completely separates the central processing stream from the graphics display stream, leveraging the multi-core processing capabilities of the Raspberry Pi processor.

3.2.4. Interface development

Based on the proposed user interface design, the software implementation process was carried out on the Qt Framework. The source code was organized according to a model that separates logic and display to optimize performance on embedded devices.

3.2.4.1. Organization of display source code

To ensure scalability and ease of maintenance, QML source code is not written centrally but is divided into separate components, managed by a main dispatch file:

- Dispatch file (main.qml): Acts as the backbone, responsible for loading image resources and arranging the positions of child modules.

- Functional Components: Complex interface components are separated into separate files (SpeedGauge.qml for speedometers, WarningIcon.qml for warnings, etc.). This organization helps reuse source code and reduce memory load by only loading necessary components.

3.2.4.2. Backed logic integration

The connection between the data processing layer and the interface is made through Qt's Signals & Slots mechanism, ensuring real-time data updates without thread conflicts.

- Engine Initialization: Uses QQmlApplicationEngine to load the interface. The Backend object is directly embedded into the QML context via the identifier "myBackend".
- Data Binding Mechanism: On the QML side, display properties (such as clock hand rotation angle, icon hide/show state) are directly linked to the dataReceived signal. When Python emits a signal, the interface automatically redraws the new state without the need for manual set/get functions.

3.2.4.3. Multithreading

A major challenge when implementing on Raspberry Pi is that listening for network data can cause the graphical interface to freeze. The following technical solutions are applied:

- Separating the UDP thread: A separate thread is created using the threading library to run the udp_listener task. This thread operates in parallel and is responsible for continuously listening on port 5005.
- Asynchronous update mechanism: When receiving a string packet from the AI system or Gateway, the UDP thread decodes it and sends the signal to the main interface thread. This architecture helps the interface maintain smoothness even with large input data traffic.

3.3. Hardware deployment

After completing the schematic design and software programming, the hardware construction process is carried out to convert the design drawings into a complete physical model. This process requires precision in manufacturing printed circuit boards, soldering components, and connecting wiring systems.

3.3.1. Implementation process

The hardware installation process is carried out sequentially from the smallest unit to the entire system integration to minimize the risk of component failure.

Step 1: Convert the drawing and print

From Altium Designer software, the printed circuit board diagram is exported to PDF format in black and white printing mode. Use a printer and specialized paper to print the diagram. Laser printer ink contains carbon powder and resin, which act as a protective layer during the etching process.

Step 2: Heat Transfer and Etching

The copper clad board is cut to the design size, then the surface is cleaned with fine sandpaper and acetone solution to remove oxidation, ensuring the best ink adhesion.

The printed paper is fixed onto the copper clad board and heated with a heat iron. Under the effect of high temperature and pressure, the ink from the paper melts and adheres firmly to the copper surface.

After cooling, the printed paper is gently peeled off, leaving sharp black circuit lines on the copper clad board. Any broken lines are manually re-marked with an oil-based marker.

The copper clad board is then immersed in an iron salt solution. The solution corrodes the bare copper layer not covered by ink, preserving the electrical circuits. The process is shaken to speed up the reaction.

Step 3: Machining

Use a handheld circuit board drill with different drill bits depending on the component pin sizes:

Drill bit 0.8mm: For passive components (resistors, capacitors), headers. Drill bits 1.0mm - 3.0mm: For power components, terminal blocks, and screw holes for circuit board mounting. For the MCP2515 module and microcontroller, use female pins to insert components instead of soldering, making replacement easier in case of failure.

After drilling, the circuit board is lightly sanded with fine sandpaper under running water to remove the protective ink layer and smooth out any burrs.

After cleaning, the circuit board is checked for continuity using a multimeter to ensure no broken or short-circuited traces due to corrosion.

A thin layer of liquid rosin solution is evenly applied to the entire surface of the printed circuit board. Once dry, this coating forms a protective film that prevents copper oxidation and acts as a flux, helping the solder spread evenly, resulting in a shinier and more aesthetically pleasing solder joint during assembly.

Step 4: Soldering Components

The system uses entirely through-hole components to ensure mechanical durability and easy replacement. The soldering process follows the "low first - high last" principle:

- Low-pass components: Soldering resistors, capacitors, and push buttons.
- Sockets: Using sockets for the microcontroller, MCP2515 module, and ESP32 helps prevent component damage due to soldering iron temperature and facilitates disassembly and inspection.
- High-power/power components: Electrolytic capacitors, voltage regulator ICs, and terminal blocks.

After soldering is complete, the circuit is cleaned again to remove burnt flux residue, ensuring an aesthetically pleasing finish.

Step 5: Testing

Before applying power, use a multimeter in continuity test mode to thoroughly check the power lines (VCC - GND) to ensure there are no short circuits caused by solder spillage during the construction process.

3.3.2. Completed hardware modules

After the fabrication and assembly process, the hardware modules have been completed and verified. The physical circuits match the design specifications, ensuring neatness and electrical safety. Below are images of the actual completed modules.

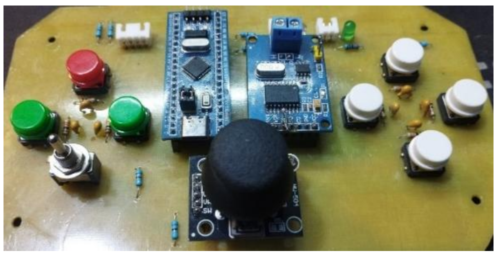

Figure 3.2. Actual Body Control node hardware

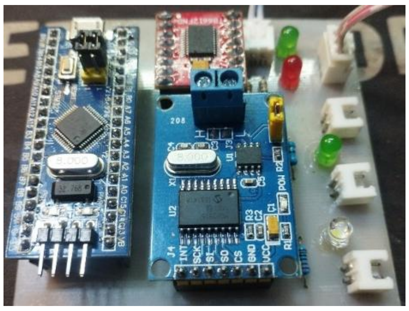
Figure 3.3. Actual Motor Control node hardware

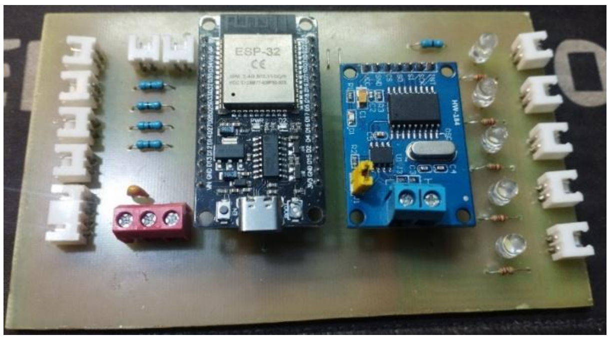
Figure 3.4. Actual Gateway node hardware

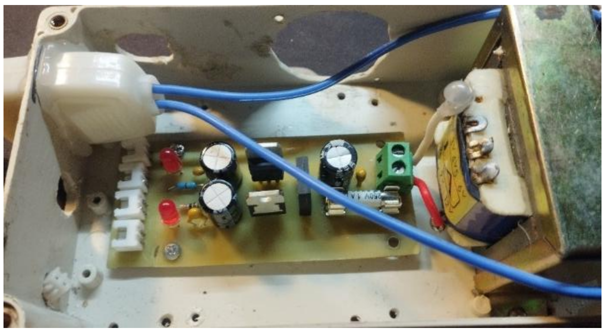
Figure 3.5. Actual power supply circuit hardware

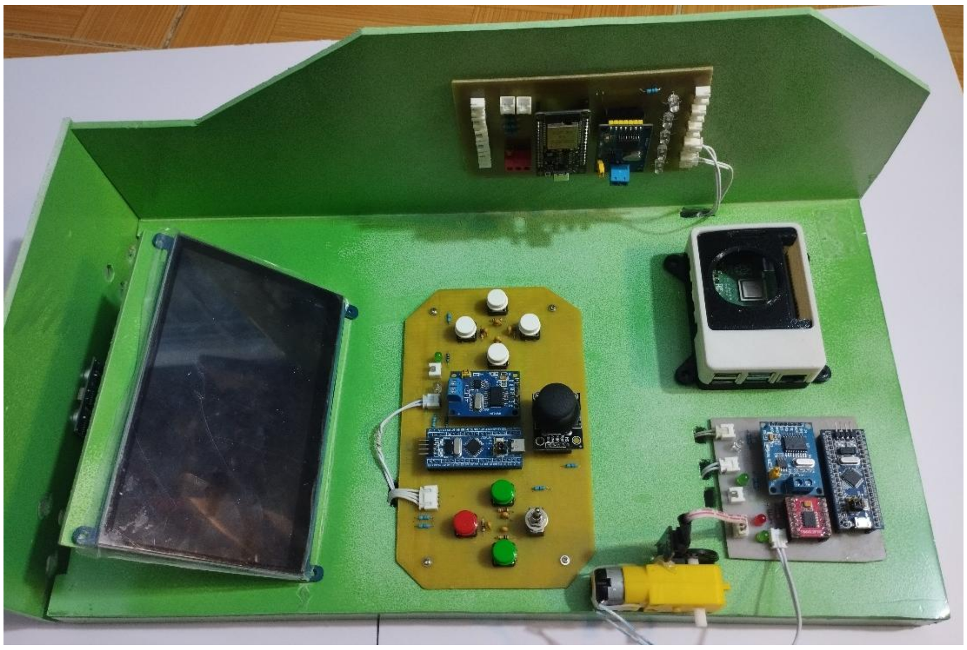
Figure 3.6. Finished product

3.4. Testing and evaluating results

3.4.1. Test scenarios and environments

3.4.1.1. Test environment

Lighting Conditions: Experiments were conducted under laboratory lighting conditions (stable fluorescent lighting) and natural outdoor lighting conditions (daytime) to evaluate the adaptability of the image processing algorithm.

Traffic Simulation Area: A simulated road layout was used, featuring scaled-down standard-sized traffic signs (1:10 ratio), including signs for: Stop and speed limit.

3.4.1.2. Test scenarios

Scenario 1 - CAN Network Test: Continuously send packets from the Body Node to the Gateway Node and vice versa to measure packet loss rate and the stability of the manual CAN Bus network.

Scenario 2 - Control Latency Test: Measure the time from when the user interacts with the control panel (e.g., turning on a light) until the vehicle's actuator responds.

Scenario 3 - Recognition and Automated Driving Test: The vehicle moves at a fixed speed, and the camera scans traffic signs. The system must correctly identify the type of sign and automatically adjust the engine (e.g., stopping completely upon encountering a STOP sign).

3.4.2. Experimental results

3.4.2.1. Communication system evaluation

Measurements using a logic analyzer and system log observation show:

- CAN Bus Network: Operates stably at a Baud rate of 250 kbps.
- Transmission Error Rate: Less than 1% (Very Low).
- Thanks to the use of twisted pair wires and 120Ω end resistors, the signals at the nodes are clear and free from large amplitude interference, even though the circuit is manually fabricated.
- Multitasking Gateway: Mutex mechanism and Priority allocation work effectively. Sending data to UART (Low priority task) does not interrupt receiving CAN messages (High priority task). The system does not freeze when running continuously for 1 hour.

3.4.2.2. Image processing and AI performance evaluation

Frame Rate: Maintains a stable 25 to 30 FPS. This is smooth enough for controlling the vehicle at low and medium speeds.

In indoor lighting conditions: Accuracy is over 90% with a Confidence level of 0.85.

Effective Detection Distance: 5cm to 15cm in front of the vehicle.

Anti-interference Mechanism: The application of 4 consecutive frame authentication completely eliminates transient false positives, resulting in smooth, jerk-free vehicle operation.

3.4.2.3. System interface and latency assessment

Instrument Cluster Interface (Qt/QML): Smooth display, clock hands move naturally thanks to asynchronous update mechanism (Signal & Slot). No interface freezing when data is received in large quantities.

Overall Latency: The time from when the camera sees the sign → AI processing → sending commands via UART/CAN → engine response is approximately 100 ms. This is an acceptable level of latency for a low-speed autonomous vehicle model.

3.4.3. Overall system evaluation

Based on the experimental results, the topic is evaluated overall through the following advantages and limitations:

3.4.3.1. Advantages

Modern distributed architecture: The system successfully applies the industry-standard automotive CAN Bus network model, minimizing wiring and easily expanding functional nodes without changing the central hardware.

Advanced technology application: Effectively combines a real-time operating system (FreeRTOS) on a microcontroller and artificial intelligence on an embedded computer, creating a powerful hybrid system.

Self-reliant hardware: Complete mastery of the manual printed circuit board design and construction process, demonstrating the ability to realize the product from theory. The baseboard design facilitates easy module replacement in case of failure and minimizes the risk of soldering errors in the manual circuit board manufacturing process.

Professional interface: The dashboard is designed with separate backend and frontend, offering high aesthetics and excellent real-time responsiveness.

3.4.3.2. Limitations

Lighting impact: The image processing algorithm is still affected by drastic changes in lighting conditions (backlighting or excessive darkness), due to the use of common cameras with limited dynamic range.

Computational power: Despite model optimization, the Raspberry Pi still gets hot when running AI tasks for extended periods, potentially affecting performance slightly without proper cooling.

Engine noise: In some cases, the motor starts quickly, and the large starting current causes a slight voltage drop on the common power line. Although a voltage regulator is present, the power supply isolation needs improvement

CONCLUSIONS AND FUTURE WORK

1. CONCLUSIONS

Within the framework of this graduation project, the research team successfully designed and implemented an integrated embedded system simulating control and monitoring functions in an automobile. The system is built using a distributed architecture, where functional nodes are connected via the CAN protocol, ensuring stable data exchange and high scalability.

In terms of hardware, the system has been completely designed and constructed, including control nodes using STM32 microcontrollers, gateway nodes using ESP32, and a Raspberry Pi central processing unit. The hardware modules are arranged logically, neatly connected, and operate stably during testing.

In terms of software, the project has developed control programs for each node, successfully implemented a multi-protocol communication mechanism (CAN, UART, UDP), and developed a visual monitoring interface using Qt/QML. In addition, the central processing unit has integrated image processing and traffic sign recognition algorithms, allowing the system to respond and display information in real time.

Experimental results show that the system operates stably, and the main functions meet the initial objectives. This project not only helps students consolidate their knowledge of embedded systems, industrial communication networks, and cross-platform programming, but also has practical significance in researching driver assistance systems and control simulation in automobiles

2. FUTURE WORKS

Despite achieving the stated goals, the system still has some limitations and significant potential for future development. Some potential development directions include:

- Expanding recognition capabilities: Upgrading the deep learning model to recognize additional traffic objects such as traffic lights, pedestrians, or lanes, thereby improving the system's completeness.
- Optimizing system performance: Improving image processing speed and reducing overall system latency by optimizing algorithms, using more powerful hardware, or leveraging hardware acceleration capabilities.

- Improving the user interface: Developing additional configuration, alert, and data logging functions to make the dashboard interface more intuitive and user-friendly.
- Upgrading the communication architecture: Experimenting with new communication standards such as CAN/FD or Ethernet to increase data transmission bandwidth and system scalability.
- Practical applications: Integrating the system into small-scale autonomous vehicle models or for research, teaching, and training purposes on embedded systems in the automotive field.

REFERENCES

Vietnamese:

[1]. Nguyễn Chí Thành, Trần Thế Sơn, Đặng Quang Hiển, Dương Hữu Ái (2023), "Thiết kế, thực nghiệm và đánh giá hoạt động của mạng điều khiển nội bộ CAN trong xe ô-tô".

[2]. Nguyễn Thanh Huyền (9/16/2021), "MobileNets – Mô hình gọn nhẹ cho mobile applications", MobileNets - Mô hình gọn nhẹ cho mobile applications, 27/11/2025.

[3]. Toyota (23/5/2023), "Cruise Control là gì? Chức năng và cách sử dụng trên xe ô tô", Cruise Control là gì? Chức năng và cách sử dụng trên xe ô tô, 10/11/2025.

[4]. Trần Quang Trung B (9/16/2019), "Tìm hiểu về SSD MultiBox Real-Time Object Detection", Tìm hiểu về SSD MultiBox Real-Time Object Detection, 28/11/2025.

[5]. K2 (15/6/2019), "Giải thuật chống nhiễu cho nút nhấn", Giải thuật chống nhiễu cho nút nhấn, 20/11/2025.

[6]. Mesidas, "CAN/CAN Bus là gì? Tổng quan về Control Area Network", CAN/CAN Bus là gì? Tổng quan về Control Area Network, 29/10/2025.

English:

[7]. Andrew G.Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, Hartwig Adam (17/4/2017), "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications".

[8]. Last Minute Engineers, "Create your own CAN network with MCP2515 Modules and Arduino", Create Your Own CAN Network With MCP2515 Modules and Arduino, 2/11/2025.

[9]. Lee Ji-Hoon (2018), "MCP2515 CAN Communication (SPI mode)".

[10]. Mamtaz Alam (2/2/2025), "ESP32 CAN Bus Tutorial | Interfacing MCP2515 CAN Module with ESP32", ESP32 CAN Bus Tutorial | Interfacing MCP2515 CAN Module with ESP32, 2/11/2025.