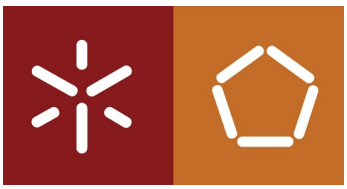


University of Minho
School of Engineering

AllWays Safe

An Intelligent Traffic System

Group 4

Master's in Industrial Electronics and Computers

André Martins — PG60192
Mariana Martins — PG60211

Project Supervised by
Professor Adriano Tavares

January 2026

Contents

List of Figures

List of Tables

Acronyms

ADC Analog to Digital Converter

ABI Application Binary Interface

API Application Programming Interface

AP Access Point

COTS Commercial Off-the-Shelf

CRUD Create, Read, Update and Delete

DDS Data Distribution Service

ERD Entity Relationship Diagram

FK Foreign Key

GPIO General Purpose Input/Output

I2C Inter-Integrated Circuit

IPC Inter-Process Communication

PK Primary Key

PWM Pulse Width Modulation

RAII Resource Acquisition Is Initialization

RFID Radio-Frequency Identification

REST Representational State Transfer

RTPS Real-Time Publish–Subscribe

SPI Serial Peripheral Interface

SSID Service Set Identifier

UDP User Datagram Protocol

UML Unified Modelling Language

URL Uniform Resource Locator

V2I Vehicle-To-Infrastructure

V2X Vehicle-To-Everything

I2P Infrastructure-To-Pedestrian

Chapter 1

1.1 Introduction

Studies from Texas A&M Transportation Institute have suggested that commuters in the United States waste an average of 54 hours a year stalled in traffic, which is almost equivalent to a whole week of work. Beyond the loss of time, prolonged congestion contributes to driver fatigue, which in turn increases the likelihood of traffic accidents. This scenario reflects a broader global challenge, as growing urban populations and the rise in vehicle numbers continue to put pressure on road infrastructure.

Figure 1.1: Traffic environment

In the recent years, new and innovative technologies such as V2I (Vehicle-To-Infrastructure) and V2X (Vehicle-To-Everything) and its subsets (such as I2P) are starting to lead the future world of autonomous vehicles. Vehicle-to-Infrastructure (V2I) Communication is a technology that enables vehicles to exchange information with traffic infrastructure, such as traffic lights, road signs, and roadside sensors. The goal of V2I is to improve road safety, optimize traffic flow, and support smart transportation systems. For example, V2I can allow traffic signals to prioritize emergency vehicles, warn drivers of upcoming hazards, or provide real-time traffic updates to reduce congestion.

1.2 Problem statement

Traffic management is essential for ensuring the safety and efficiency of both drivers and pedestrians. However, conventional traffic systems lack real-time adaptability and effective communication between road users and infrastructure.

To address this limitation, an intelligent traffic management system is required. The proposed system is based on Vehicle-to-Infrastructure (V2I) communication and incorporates Infrastructure-to-Pedestrian (I2P) interaction to enhance pedestrian safety and

accessibility. The system should enable real-time communication between traffic lights, emergency vehicles, and pedestrians, allowing early alerts and proactive traffic signal adjustments.

By prioritizing emergency vehicles such as ambulances, the system reduces response times and minimizes traffic congestion. In addition, it should improve pedestrian accessibility by dynamically adjusting crossing durations to accommodate individuals with reduced mobility. Tracking pedestrian movement further enhances safety and provides support in situations where pedestrians may become disoriented or lost.

Figure 1.2: Vehicle-To-Infrastructure and Infrastructure-to-Pedestrian proposed technologies

1.3 Motivation

In modern cities, traffic management plays a vital role in ensuring safety, efficiency, and accessibility. However, current systems often struggle to adapt to unexpected situations, such as emergencies, where delays or lack of communication can put both drivers and pedestrians at risk. At the same time, they frequently fail to address the needs of people with reduced mobility, who require additional time or support to cross safely. With growing urban populations and an increasing number of vehicles, the limitations of traditional traffic systems become even more evident. Traffic congestion, emergency delays, and accessibility gaps highlight the urgent need to rethink how urban mobility is managed to create safer and more inclusive environments for all road users.

1.4 Requirements

In every project, a list of requirements is mandatory as it shapes the project's development direction. These can be divided into two sections: functional and non-functional.

1.4.1 Functional requirements

- Command Semaphores based on time and external alerts;
- Track the path done by people with disabilities;

• Communicate with nearby ambulances;
• Alert pedestrians in case of an emergency situation;

1.4.2 Non-Functional requirements

• Provide a friendly User Interface;
• Durable;
• Low Market Price.

1.5 Constraints

Similarly, addressing the constraints is equally important. These can be either technical or non-technical.

1.5.1 Technical Constraints

• Use the Raspberry Pi;
• Use Buildroot's embedded Linux image;
• Use C/C++ language (OOP).
• Implement at least one Linux device driver;
• Use UML for the analysis (OOD);
• Use PThreads.

1.5.2 Non-Technical Constraints

• Project must be developed in groups with no more than 2 elements;
• Project deadline at the end of the semester;
• Limited budget.

1.6 System Overview

Considering the project's requirements and constraints mentioned before, a high level system overview was conceived (Figure 1.3). This representation outlines the most relevant sub systems and how they interact.

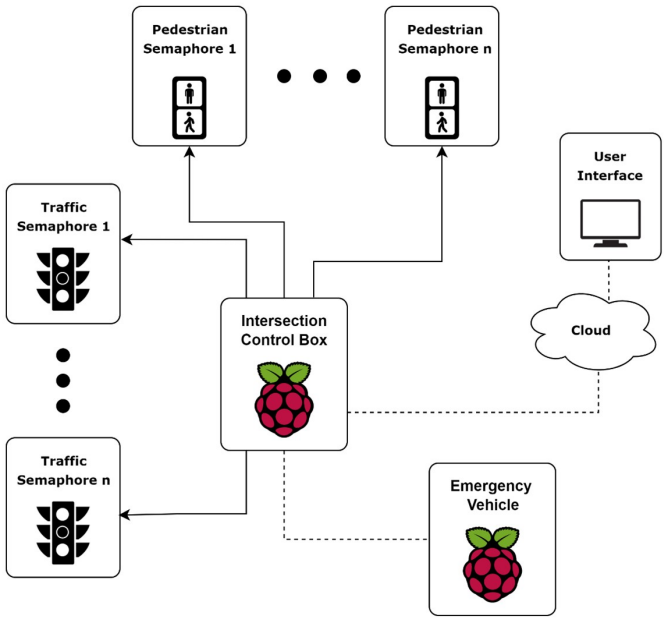


Figure 1.3: AllWays Safe system schematic

1.7 Gantt diagram

Figure 1.4 below illustrates the expected project timeline along with the key milestones to be achieved.

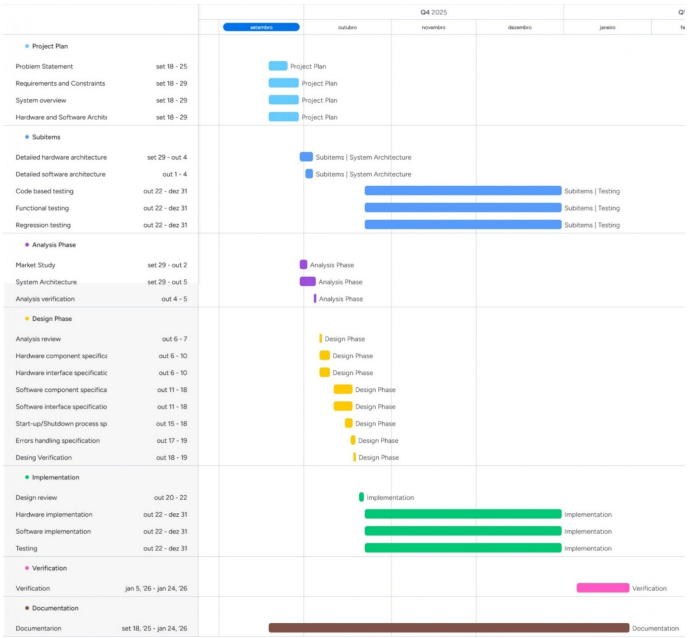


Figure 1.4: Gantt diagram

Chapter 2

Analysis

In the previous chapter, a general overview and a high-level analysis of the system were presented. Now, to more accurately identify certain characteristics that were previously overlooked or not considered, a deeper analysis of the problem and the requirements for its solution must be carried out.

2.1 Market Study

The global intelligent traffic management system market was valued between $12–15 billion in 2024 and is projected to reach $28–47 billion by 2030–2034, reflecting a strong annual growth rate (about 10–15%) due to accelerating adoption of AI, IoT, and real-time data analytics in urban traffic control, according to Precedence Research (Figure 2.1). The market features a range of products already in real- world use, particularly those integrating real-time adaptation for emergency vehicles and enhanced accessibility for pedestrians.

Figure 2.1: Global intelligent traffic management system market

2.1.1 Similar products

Sinowatcher Adaptive Traffic Light Controller
- Manufacturer: Sinowatcher Technology Co., Ltd (Shenzhen, China);
- Price: Approximately US$ 3,500 per unit (minimum order 5 pieces);
- Features:
  - Modular and extendable design, stable and reliable hardware with embedded Linux OS;
  - Supports coordinated control network with command centre integration;
  - Multi-period control includes manual mode, yellow flashing and over-voltage protection;
  - Algorithms to improve traffic capacity and flow;
  - Supports pedestrian crossing trigger control with 8 pedestrian inputs (dry contact or

transistor output);

• Vehicle input detection interface;

• Signal output: 56 channels, extendable to 112 channels;

• Sensors and detection hardware (vehicle loops, pedestrian push-buttons) must be purchased separately;

• Centralized control software, which can be integrated optionally with host computer systems.

Miovision

- Manufacturer: Miovision (Canada);
- Product: Miovision Adaptive;
- Price: Custom pricing based on deployment scale;
- Features:

• Real-time adaptive traffic signal control using AI algorithms;

• Continuously analyses real-time traffic data to dynamically adjust signal timings every second, reducing delays, stops, and emissions;

• Multi-modal optimization including vehicles, pedestrians, bicycles, transit, and connected/autonomous vehicles;

• Decentralized and scalable system designed for complex traffic scenarios beyond simple intersections;

• Uses advanced sensing technologies including cameras, radar, and connected vehicle data feeding AI for accurate traffic state estimation;

• Emergency vehicle prioritization and transit priority integrated;

• Centralized dashboard and cloud-based management for visualization, reporting, and remote control.

2.1.2 AllWays Safe on the Market

AllWays Safe stands out from existing adaptive traffic control systems by prioritizing accessibility, affordability, and inclusivity without compromising real-world performance. Unlike costly commercial solutions such as Sinowatcher and Miovision Adaptive, which depend on complex infrastructure and centralized management, AllWays Safe offers intelligent traffic control at a fixed, budget-friendly price. The system features an emergency response mechanism that automatically prioritizes emergency vehicles, adjusts traffic lights, and alerts pedestrians with distinct audible signals, achieving responsiveness comparable to premium systems but without reliance on AI cloud processing or external sensors. AllWays Safe also enhances pedestrian safety through innovations like:

• RFID-based accessibility, extending crossing time for users with reduced mobility;

• A simple web and cloud interface providing real-time intersection data while ensuring privacy.

By focusing on essential smart functions over costly analytics, AllWays Safe delivers a modular, reliable, and inclusive solution—making it the most practical and affordable intelligent traffic system for cities seeking sustainable modernization.

2.2 System Architecture

To properly understand the structure and behaviour of a system, it is essential to define its architecture. A system's architecture provides a high-level description of the roles of its components and the way they interact with each other and with the external environment.

2.2.1 Hardware Architecture

Intersection Control Box

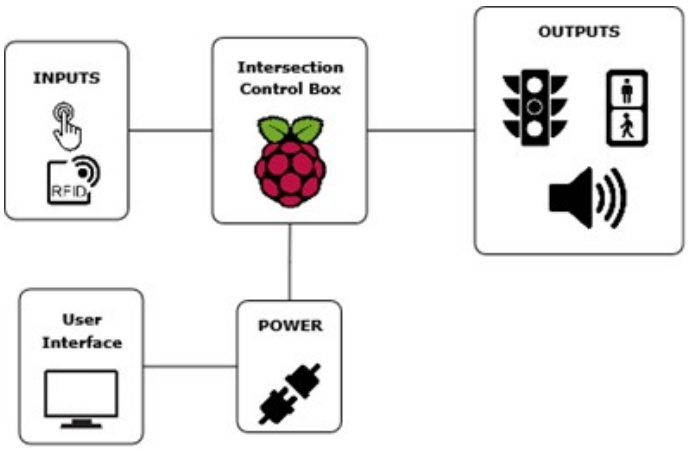


Figure 2.2: Intersection Control Box's Hardware Architecture

• **Power Supply**: the system is powered directly from the main electrical grid, ensuring continuous and stable operation of all connected components.
• **Raspberry Pi 4**: serves as the central processing unit of the system, responsible for managing inputs and outputs. It runs the control algorithms and ensures synchronization of all components.
• **Traffic Semaphore**: manage vehicle flow at the intersection by controlling stop-and-go signals, ensuring traffic safety and efficiency.
• **Pedestrian Semaphore**: provide visual signals (red/green) for pedestrians to cross the intersection safely.
• **Push-Button**: allows pedestrians to request a crossing. The system considers both the number of button presses and the remaining time in the current traffic light cycle. The more people that press the button and the longer the wait time, the higher the probability that the pedestrian light will switch to green sooner. However, if the signal is already close to changing, pressing the button will not force an immediate switch.
• **RFID card reader module**: designed for pedestrians with disabilities. By tapping an RFID card, their presence is registered in the system. When the pedestrian signal turns green, the system automatically extends the green phase to provide additional crossing time for those in need.
• **Buzzer**: provides auditory signals for pedestrians. It emits sounds to indicate when it is safe to cross, and in emergency situations, it changes its frequency to alert pedestrians of approaching hazards.
• **User Interface**: displays the current and historical status of all traffic lights and pedestrian signals at the intersection. It also provides logs of crossings by registered disabled pedestrians and shows pedestrian flow over time, based on button presses. This interface helps monitor system performance and traffic demand.

Emergency Vehicle

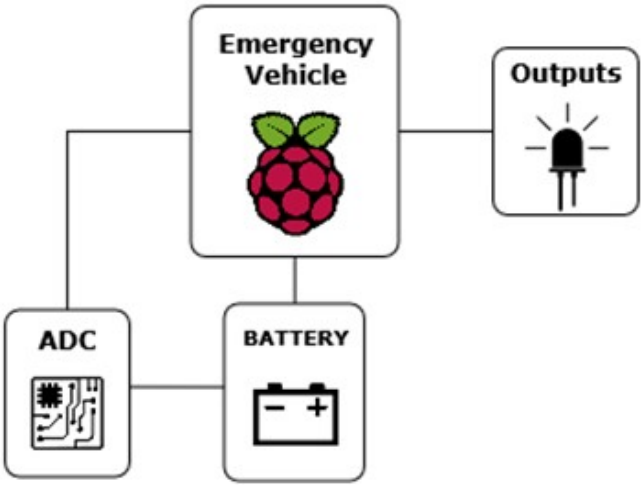


Figure 2.3: Emergency Vehicle's Hardware Architecture

• **Battery Power**: provides portable energy for the Raspberry Pi, allowing the system to be mobile while ensuring sufficient and stable power for proper operation.
• **Raspberry Pi 4**: acts as the main controller of the ambulance simulation system. It is responsible for simulating the approach of an ambulance to an intersection managed by the Intersection Control Box, enabling the testing of emergency scenarios where traffic signals must adapt to give priority.
• **Analog to Digital Converter**: monitors the battery voltage level and converts the analog signal into digital data, allowing the Raspberry Pi to track the battery's charge status accurately.
• **Blink Red LED**: serves as a visual indicator of low battery charge, providing immediate feedback to the user.

2.2.2 Software Architecture

The software architecture is a high-level representation of how different software systems and their components interact with each other to produce a functional and cohesive system. It defines the structure, relationships among the system's modules and layers. It can be divided into three main layers:

• **Application Layer**: Defines the main functionalities of the system, contains user interfaces and services that directly meet user requirements.

• **Middleware Layer**: Acts as an intermediary between the application and the underlying operating system. It manages communication, data exchange, and service coordination. This layer may include APIs, libraries, message brokers, or frameworks that simplify integration and scalability.

• **Linux Layer**: Represents the lowest layer of the architecture, where the operating system kernel and device drivers reside.

Intersection Control Box

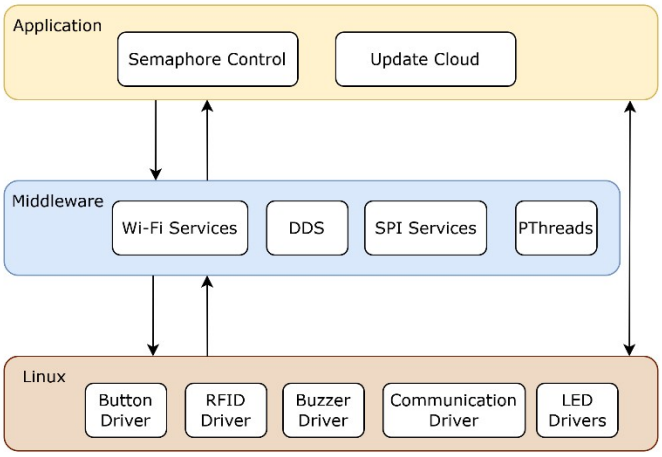


Figure 2.4: Intersection Control Box's Software Architecture

• **Semaphore Control**: This module is responsible for managing the state of all traffic lights within the system. It ensures that each semaphore follows the correct timing sequence based on predefined logic or dynamic inputs from other modules.
• **Update Cloud**: Handles data synchronization with the cloud. It periodically uploads key system information such as pedestrian button activity, RFID card detections from disabled users, and traffic light status changes.
• **Wi-fi Services**: Provide the communication interface between the local traffic controller and external systems. They are essential for connecting to the cloud (to send and receive updates) and for enabling DDS (Data Distribution Service) communication between such as other traffic controllers or emergency vehicle modules. This ensures reliable, real-time data exchange across the network.
• **DDS**: Used for real-time communication and coordination between different traffic lights and emergency vehicles. In the Intersection Control Box enables message subscription on a specific topic allowing the system to react quickly to high-priority events and adjust traffic behaviour accordingly.
• **SPI Services**: Handle low-level communication between the main controller the RFID module. Through SPI, the system can read the card ID of users with disabilities, ensuring accurate and fast data transfer to trigger adaptive traffic light behaviour.
• **PThreads**: Used to implement real-time multitasking within the system. This ensures concurrent task execution with minimal latency and high reliability.
• **Button Driver**: Records and analyses the number of times the pedestrian button is pressed, allowing the system to anticipate green lights at crossings by evaluating pedestrian demand (via button activity) and the time remaining until the next green light switch.
• **RFID Driver**: To allow other system layers to determine how to adjust traffic when a disabled person swipes a card on the RFID module.
• **Buzzer Driver**: For driving the buzzer that alerts pedestrians when it's safe to cross the traffic light, and also activates in case of an emergency, such as an ambulance passing by, to warn people to clear the crosswalk.
• **Communication Drivers**: Responsible for receiving and processing messages published on DDS topics, as well as, SPI communication for the RFID module.
• **LED Drivers**: Directly controls the physical LEDs that represent the traffic light colours. It receives instructions from the Semaphore Control module to illuminate the correct signals at the correct times.

Emergency Vehicle

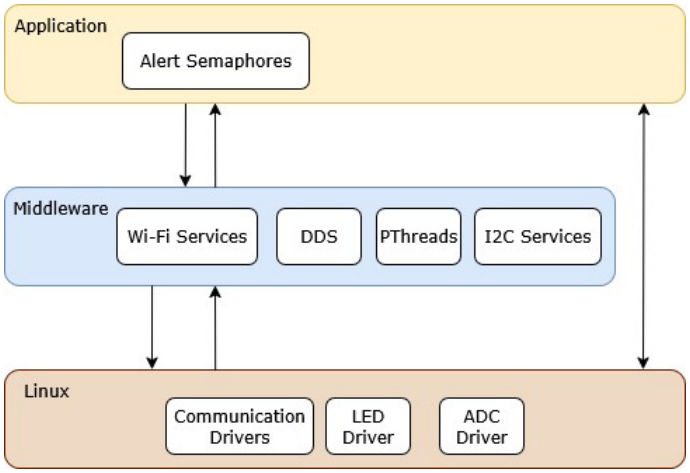


Figure 2.5: Emergency Vehicle's Software Architecture

• **Alert Semaphore**: responsible for notifying nearby traffic lights when an emergency vehicle is approaching.
• **Wi-fi Services**: Provides connectivity for cloud synchronization and facilitate data exchange with other layers.
• **DDS**: Used for real-time communication and coordination between different traffic lights and emergency vehicles. The Emergency Vehicle is configured to publish data to a specific topic.
• **PThreads**: Used to implement real-time multitasking within the system. This ensures concurrent task execution with minimal latency and high reliability.
• **I2C Services**: Manage the communication between the battery monitoring ADC module and the Raspberry Pi controller.
• **Communication Drivers**: Serves as a general communication interface within the system. They handle both high-level message exchange (such as DDS-based updates) and low-level hardware communication (through I2C). By processing messages and data from multiple sources, the driver coordinates appropriate system responses.
• **LED Driver**: Controls the visual indicator that display the battery's charge status. When the battery voltage falls below the threshold required to power the Raspberry Pi and its peripherals, the LED driver activates a warning light to alert maintenance actions.
• **ADC Driver**: For monitoring the system's battery condition. It continuously measures the battery voltage and converts the analog readings into digital data.

Package Diagram

A package diagram is a type of structural diagram used in Unified Modeling Language (UML) to organize and group related elements of a system. It provides a high-level view of the system architecture by showing how different packages are structured and how they depend on one another.

Packages are used to reduce complexity by dividing a large system into smaller, manageable parts. Each package can contain classes, interfaces, or even other packages. The relationships between packages, such as dependencies, help illustrate how different parts of the system interact.

Figure 2.6 presents a high-level representation of the system's package diagram.

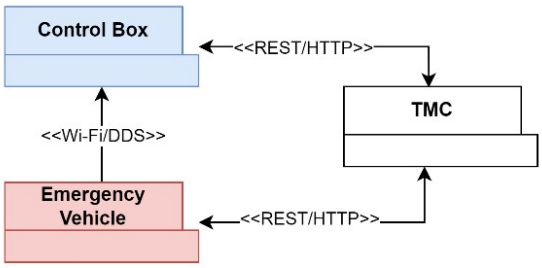


Figure 2.6: Package Diagram overview

2.3 System Analysis

The system analysis aims to clarify the interaction between the different parts of the system, both internal and external, as well as the system's response on various events.

2.3.1 System Events

This section details the main system events that drive the behaviour of both the intersection box Control and the emergency vehicle subsystems. Each event is characterized by its trigger, the entity responsible, synchronization type, and the system's corresponding response. By identifying these events, the system's dynamic operation and its ability to react to both routine and exceptional scenarios are clarified.

Intersection Box Control

Event	Trigger/Source	Type	System Response
Power On	User	Async	Initialize System
Emergency Vehicle Approximation	Emergency Vehicle	Async	Interpret message from topic
End of colour of Traffic semaphore	Timer	Sync	Change to next colour
End of colour of Pedestrian Semaphore	Timer	Sync	Change to next colour
Card passed on RFID reader	User	Async	Adapt time passage
Button pressed in Pedestrian Semaphore	User	Async	Check if Traffic Semaphore can change to allow passage

Table 2.1: Intersection Control Box's Events

Emergency Vehicle

Event	Trigger/Source	Type	System Response
Power On	User	Async	Initialize System
Low Battery Level	System	Async	Blink LED
Battery Level Acquisition	Interrupt	Async	Save Battery Level data
Traffic Semaphore Approximation	System	Async	Publish warning message to topic

Table 2.2: Emergency Vehicle's Events

2.4 Use Case Diagrams

The Use Case Diagram is a type of UML diagram that represents the interaction between actors (which may be users or external systems which interact with the system) and a system under consideration. When presenting these relations, it specifies whether certain actions are compulsory or not, by identifying them with `<<include>>` and `<<extended>>`

keywords. Thus, it provides a high-level view of the system's functionality by illustrating the various ways users can interact with it.

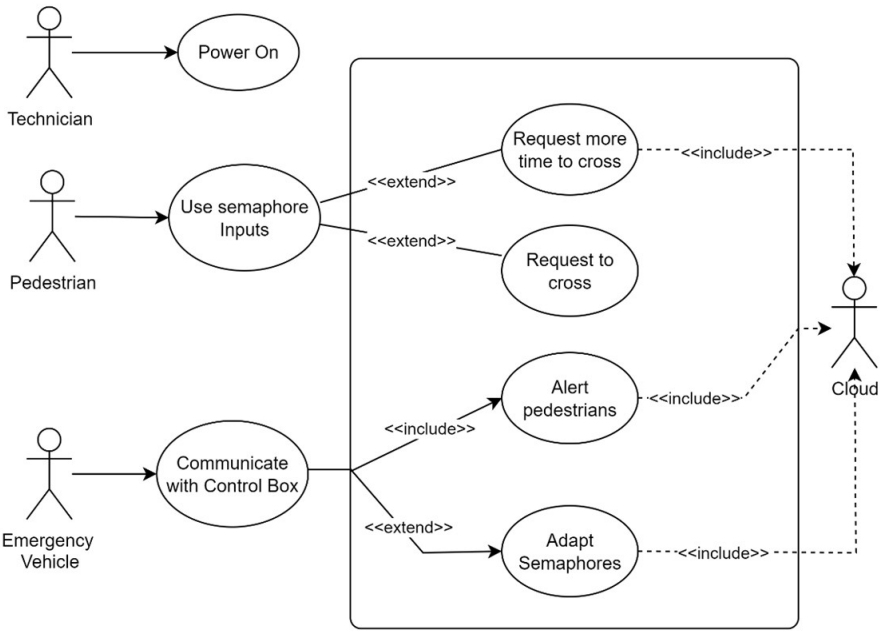


Figure 2.7: System Use Case Diagram

By analysing the Use Case Diagram (Figure 2.7), we can identify the actors "Technician", "Monitor", "Pedestrian" and "Emergency Vehicle" and the system itself which controls the whole intersection and keep data in the Cloud, which is, therefore, an intervenient in the system. The system is powered on by the technician who is responsible for setting it up. Then, it can be used both by pedestrians, who will physically interact with it, and emergency vehicles which establish wireless communication.

The monitor will be a remote person who is in charge of monitoring the data sent to the database, as it is perceivable in the Use Case Diagram, in Figure 2.8.

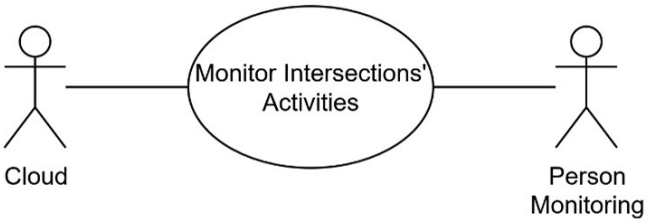


Figure 2.8: Cloud Use Case Diagram

2.5 Entity Relationship Diagram

In Figure 2.9, the Conceptual ERD illustrates how information about intersections, semaphores, push-buttons, and identification cards for people with disabilities are organized and connected in the cloud database, which will be updated by the Intersection Control Box.

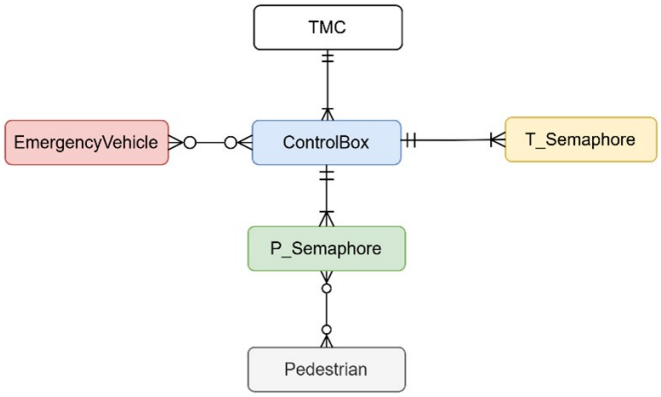


Figure 2.9: Cloud Conceptual Entity Relationship Diagram

The main purpose of this diagram is to ensure a clear and efficient organization of data, allowing:

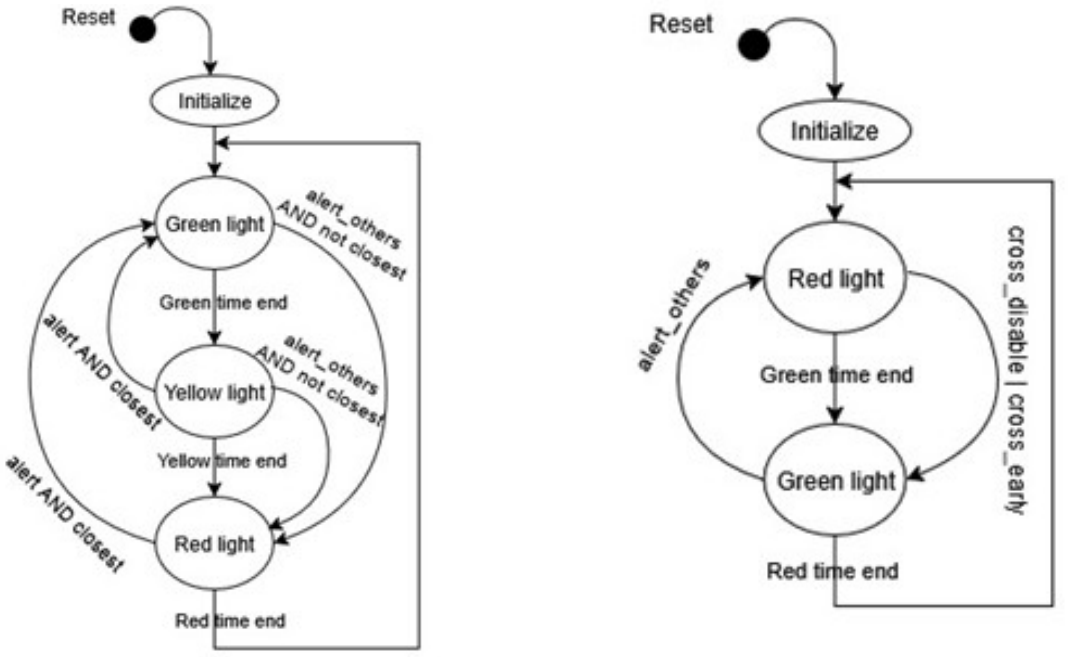
- the registration and monitoring of traffic and pedestrian semaphores within each intersection;
- the association of push-buttons and identification cards with specific semaphores;
- and the collection of real-time information about traffic flow, user activity, and semaphore status.

2.6 State Diagrams

To more clearly specify the sequence of system execution, a state machine diagram is extremely helpful. Due to its inherently sequential nature, a state machine diagram makes it straightforward to understand and predict how the system will behave in response to various events, illustrating state transitions and system reactions in an intuitive and organized manner.

Intersection Box Control

In Figure 2.10b and Figure 2.10b, the state diagrams for both the traffic lights and pedestrian lights are illustrated, showcasing their sequential and event-driven behaviours. These diagrams demonstrate the standard flow of signal states under normal operation, following a preset sequence when no external events occur to interrupt it. However, the diagrams also highlight how certain external events can lead to state transitions outside the regular pattern, for example, when a pedestrian presses the crossing button, a disabled person activates an RFID module, or an emergency vehicle sends an alert message. These responsive transitions ensure that the system can adapt to real-time needs, prioritizing safety and accessibility for all users when special situations arise.

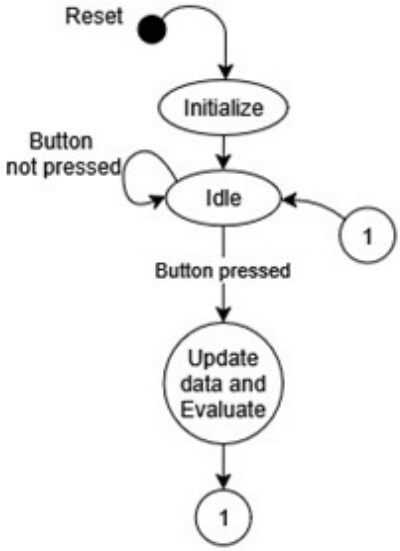


(a) Traffic Semaphores state diagram

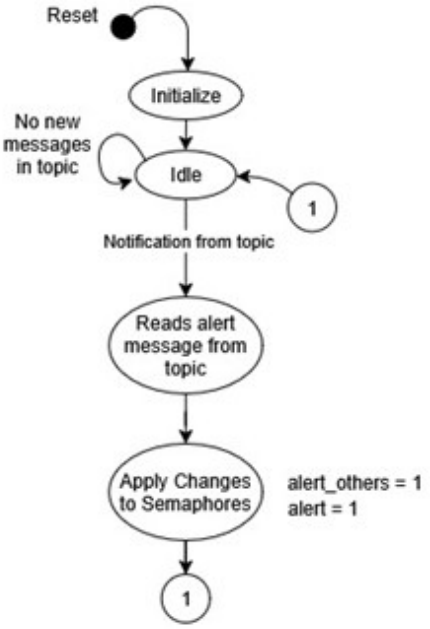
(b) Pedestrian Semaphores state diagram

Figure 2.10: State diagrams of traffic and pedestrian semaphores

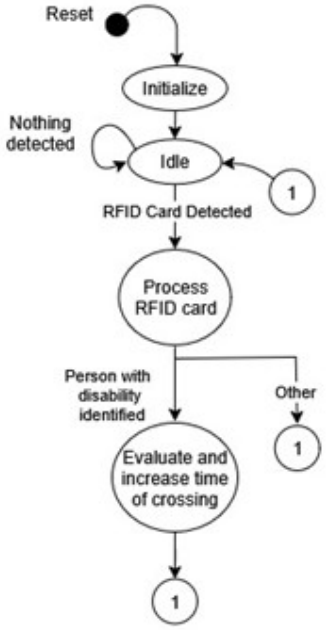
These additional state machine diagrams illustrate how the system responds to the three types of external events that may interrupt the standard signal sequence. The first diagram shows how the controller reacts to alert messages, such as from emergency vehicles, where notifications trigger immediate updates to the traffic signals to prioritize urgent passage and synchronize semaphores for safety. The second diagram details the sequence activated when an RFID card is detected, identifying a person with disabilities; the controller processes this input and adjusts the crossing time to ensure safe and accessible passage across the street. The third diagram depicts the action taken when a pedestrian presses the crossing button. The system moves from idle to evaluating the request, updating data to allow crossing at the next safe opportunity. Together, these state machines exemplify the adaptivity traffic control, allowing immediate response to different circumstances.



(a) Push-Button state diagram



(b) Alert message reception state diagram

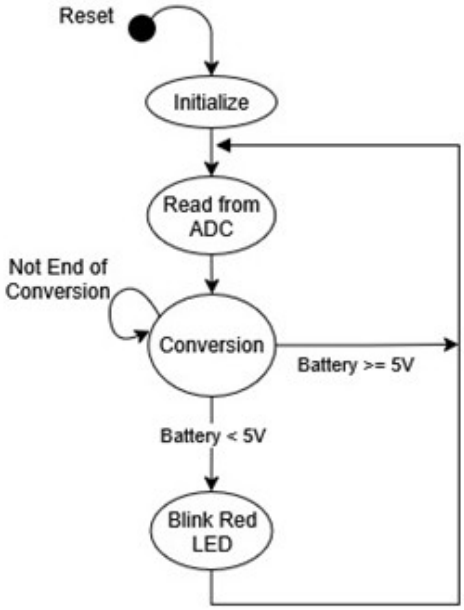


(c) RFID module state diagram

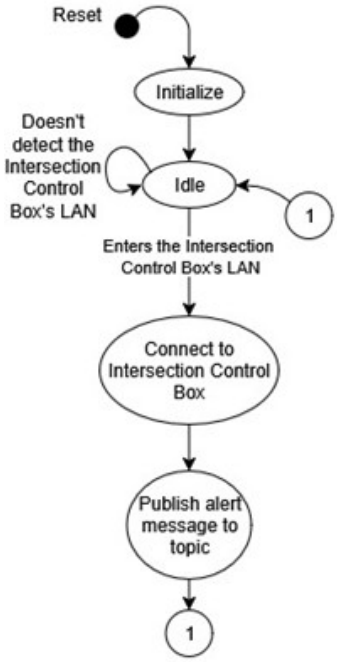
Figure 2.11: State diagrams of the push-button, alert reception, and RFID modules

Emergency Vehicle

The emergency vehicle system relies on two interconnected state diagrams to ensure reliable operation and fast response. The first state machine (Figure 2.12a) is dedicated to continuously monitoring the battery voltage using an ADC. It repeatedly reads ADC data, and whenever the battery voltage drops below a critical threshold, the system triggers a blinking red LED as a warning indicator. The second state machine (Figure 2.12b) manages the alert protocol itself. When an emergency vehicle enters the intersection and its system connects to the control box, this state machine automatically publishes an alert message, signalling the emergency and enabling quick adjustments in the traffic control infrastructure to grant priority to the emergency vehicle.



(a) Battery acquisition state diagram



(b) Alert message system state diagram

Figure 2.12: State diagrams of battery acquisition and alert message system

2.7 Sequence Diagram

The sequence diagram is an extremely relevant component of UML, since it enables the visualization of the interaction between several objects of the system in sequential order, for a specific application. It illustrates the flow of messages exchanged between objects, emphasizing the order and timing of these interactions. Here the dashed lines with an empty arrow indicate an asynchronous event, while the others indicate a synchronous event, which is triggered by the former.

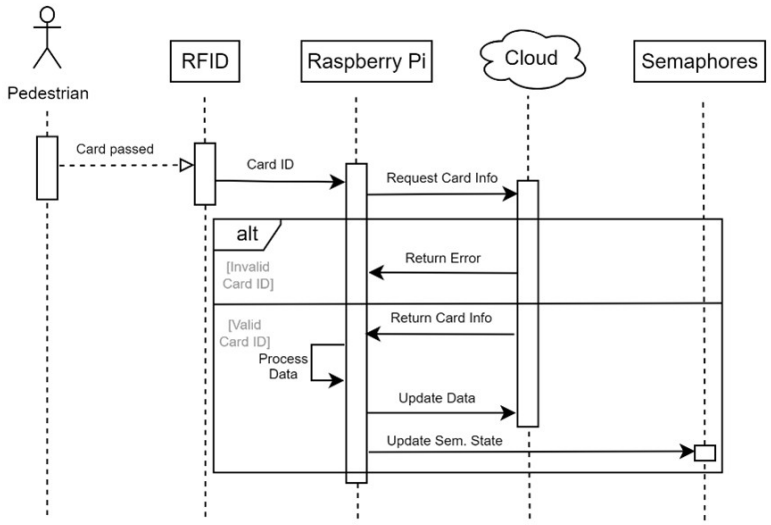


Figure 2.13: Sequence Diagram – RFID Card Passed

In Figure 2.13, the flow of the RFID read is depicted. Here we can see that upon a RFID card has been detected, its information is requested to the cloud, where there can be

two possible outputs. The first is "Error", which indicates that the RFID card passed isn't a valid one, and, therefore, the control box takes no action. The second is a confirmation, where the requested card information is received and then, processed in order to possibly act on the semaphores' state.

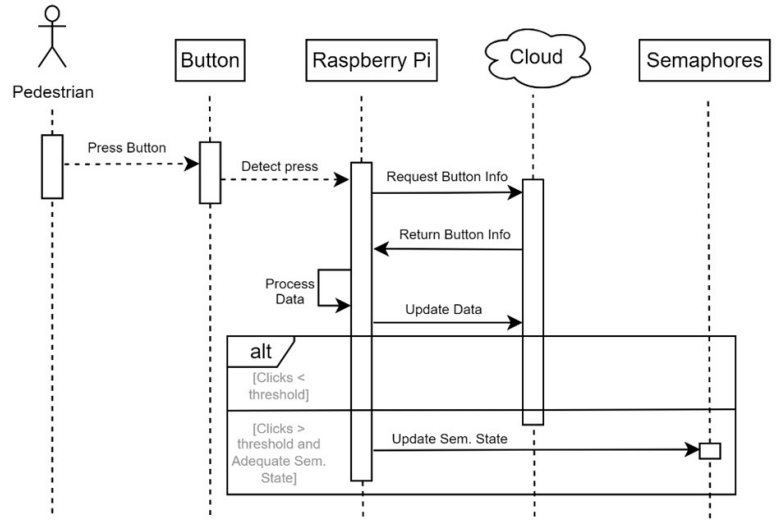


Figure 2.14: Sequence Diagram – Button Pressed

The sequence of actions for the Button press in Figure 2.14 is equivalent to the RFID process previously mentioned, with the only difference being the processing stage, in which the button press is characterized by always having a valid "Button Info".

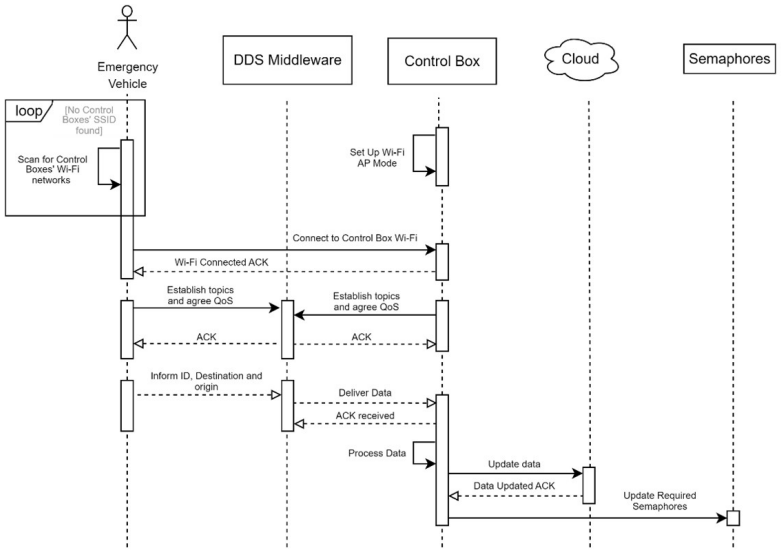


Figure 2.15: Sequence Diagram - Communication Between Emergency Vehicle and Control Box

In Figure 2.15, the communication process between the Emergency Vehicle and the Control Box is described. Initially, the Control Box (Raspberry Pi) must set itself as Access Point (AP), and the Emergency Vehicle searches for the Control Boxes' Wi-Fi Network. Upon finding a Control Box, the connection is established. Afterwards, the DDS's options are set up and an agreement is reached between the Control Box and the Emergency Vehicle. From here, communication between the two is enabled and the Emergency Vehicle informs its approximation, with the publication of its ID, destination and origin. The

data is correctly acknowledged and then processed, so as to update the database and the required semaphores.

Intersection Control Box

Item	**Quantity**	**Unit Price**
Raspberry Pi 4 Model B	1	52.00€
Push-Button	2	0.63€
Buzzer	2	0.48€
Protoboard	10	0.25€
RFID Module	2	1.87€
5mm LEDs	15	0.02€
Structural Components	1	25.00€
Total	**33**	**85.76€**

Emergency Vehicle

Item	**Quantity**	**Unit Price**
Raspberry Pi 4 Model B	1	52.00€
5mm LEDs	1	0.02€
Battery	1	15.00€
Analog to Digital Converter	1	2.24€
Voltage Regulator	1	5.90€
Structural Components	1	5.00€
Total	**6**	**80.16€**

Chapter 3

Design

The design phase begins with a review of the analysis phase. After that, we aim to determine how the system will meet the requirements, taking its constraints into account. To do this, we provide a more detailed description and use of the components mentioned earlier during the Hardware and Software Architecture stages – how they will interact, what needs to be built and what already exists, as well as what scenarios the system will face and how it should respond to them.

3.1 Hardware Specification

The hardware specification section provides a detailed description of all the physical components used in the system, including their characteristics, connections, and 3D design elements. Clearly defining the hardware setup is essential to ensure that the system functions as intended, supports all required operations, and maintains reliability and performance. This section outlines the components used, their technical features, and how they interact within the system. Proper documentation of hardware specifications not only helps in replicating and troubleshooting the setup but also ensures compatibility between different modules and provides a solid foundation for future upgrades or improvements.

3.1.1 Raspberry Pi 4 Model B

Within our project, and as previously mentioned, the Raspberry Pi 4 plays a critical role. One unit will be deployed as the Intersection Control Box, responsible for managing and synchronizing traffic control. A second Raspberry Pi will operate in Emergency Vehicle mode dedicated to communicating with the Intersection Control Box. The Raspberry Pi 4 Model B is built around the Broadcom BCM2711 system-on-chip, featuring a quad-core ARM Cortex-A72 (ARM v8) 64-bit processor running at 1.5–1.8GHz. This platform supports advanced computing, I/O expansion, and connectivity requirements essential for smart traffic infrastructure. Some key features of the Raspberry Pi 4 relevant to our project include:

- 40-Pin GPIO Header;
- 2.4GHz and 5.0GHz WiFi connection compliant with the IEEE 802.11b/g/n/ac standard;
- 1x Gigabit Ethernet port (supports PoE with add-on PoE HAT);
- 1x Raspberry Pi display port (2-lane MIPI DSI);
- MicroSD Card for loading the operating system and saving persistent data;
- Up to 6x I2C;
- Up to 5x SPI;
- Up to 2x PWM channels;
- ARMv8 Instruction Set;
- Recent Linux kernel support;

- Many drivers up streamed.

Electrical specifications:

- Output high voltage (V_{OH}) - 3.3V
- Output low voltage (V_{OL}) – 0.4V
- Output high current (I_{HL}) – 8mA
- Output low current (I_{OL}) – 7mA

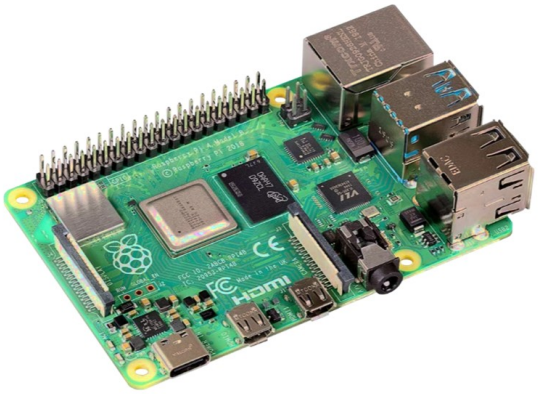


Figure 3.1: RaspberryPi 4 Model B

3.1.2 LED

For clear visual feedback during demonstrations and testing, our setup includes dedicated LED indicators controlled directly from the Raspberry Pi GPIO pins. Each traffic light state — red, green and yellow — is represented using 3 LEDs to increase visibility and make the current phase easy to perceive at a glance, even from different viewing angles.

To drive these LEDs safely and consistently, we designed simple current-limiting circuits and selected different resistor values for each colour to account for their distinct forward voltages. This ensures uniform brightness and reliable operation while providing an immediate visual reference of the system status.

Figure 3.2: LED colours that will be used

Colour	**Forward Voltage (V)**
Red	1.8
Green	2.3
Yellow	1.8

Table 3.1: Forward voltage for different LED colours

Since the default output current of a Raspberry Pi GPIO pin is approximately 8 mA, we used Equation 3.1 to calculate the appropriate series resistor for each LED branch. For the red and green LEDs we targeted a forward voltage of about 3 V, while for the yellow LED

we allowed a slightly higher forward voltage to compensate for its lower luminous intensity. This resulted in a smaller resistor value for the yellow branch, ensuring a comparable brightness across all three colours.

$$\mathbf{R} = \frac{3.3 - 3}{\frac{8\times10^{-3}}{3}}\,\Omega \tag{3.1}$$

Therefore, in order to connect the LEDs to the Raspberry Pi, the following circuits will be implemented in protoboard.

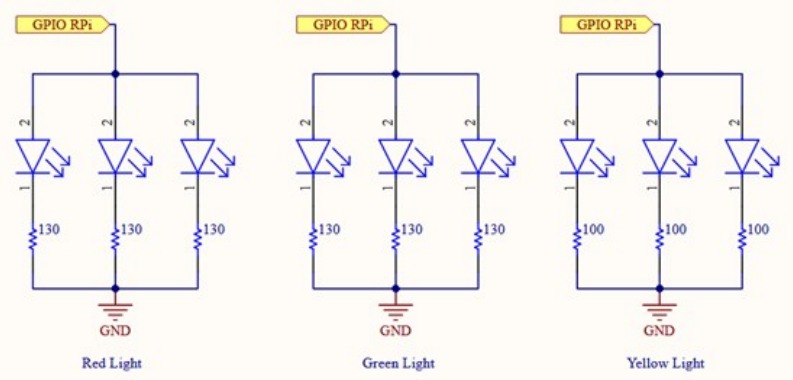


Figure 3.3: LED hardware interface

3.1.3 Push-Button

The push-button is a simple button that will be used for user interface in the pedestrian semaphores. In order to connect it to the Raspberry Pi, it will be necessary to design a circuit with pull-down configuration (Figure 3.4),which will be implemented in protoboard.

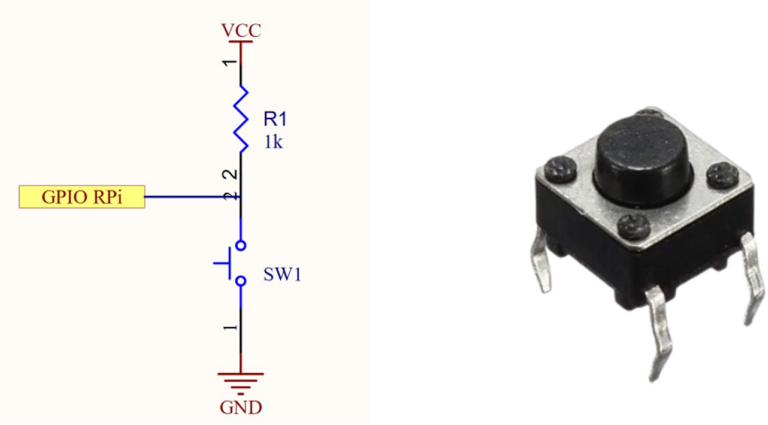


Figure 3.4: Push button and respective hardware interface

3.1.4 RC522 Module

The RC522 Module is a RFID (Radio Frequency Identification) Module (Figure 3.5) based on the NXP's MFRC522 chip, which will be used to read the cards or tags of disabled people on the semaphores, from a distance of up to 5cm. Other relevant characteristics are:

- Operating Voltage: 2.5V – 3.3V.
- Operating / Standby Current: 13–26mA / 10–13mA.

Peak Current: < 30 mV

- Operating Frequency: 13.56MHz.
- Communication Interface: SPI bus, up to 400 kBd (Fast mode) / 3.4 MBd (High-speed mode).

- Supported Card Types: MIFARE 1K (S50), MIFARE 4K (S70), MIFARE UltraLight, MIFARE Pro, MIFARE DESFire.
- Data Rate: Up to 424 kbit/s.
- Compatible Standards: ISO/IEC 14443A.

Figure 3.5: RC522 module

3.1.5 Buzzer

In order to warn pedestrians of emergency situations and indicate when it is safe for them to cross, a buzzer module is used (Figure 3.6). It is an active buzzer and, therefore, it only requires a DC input in order to produce sound. It specifications are the following:
- Rated Voltage: 3.3V – 5V
- Current Draw: $\leq$ 30 mV
- Sound Output: $\geq$ 85 mV

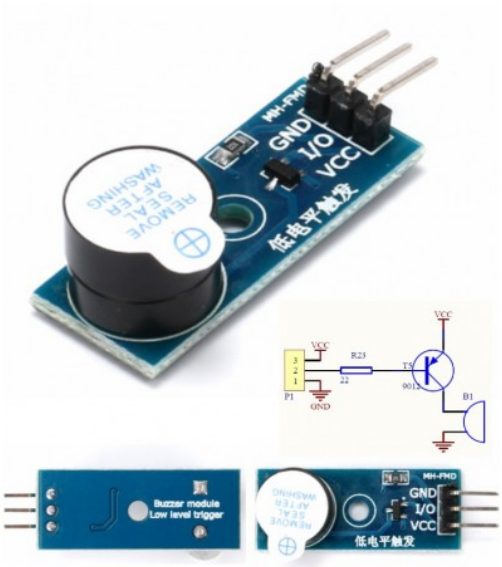


Figure 3.6: Buzzer Module

3.1.6 Power Supply Unit

This unit will be used to power the Raspberry Pi and other electronic components in the intersection control box, ensuring stable and efficient operation for all critical traffic management functions.

This power supply unit, model LM50-23B05R2, is a compact industrial-grade module designed for reliable operation. It supports a wide input voltage range (80–305VAC or 100–430VDC), delivers 50W output at 5V and up to 10A, with high efficiency of 86%, which is perfect for its purpose.

Figure 3.7: Power unit- LM50-23B05R2

3.1.7 Emergency Vehicle Battery

The battery provides flexibility for the Raspberry Pi's movement while simulating an emergency vehicle navigating through different traffic lights. To achieve this, we use two 3.7V lithium-ion batteries connected in series, providing a nominal voltage of 7.4V. Lithium-ion batteries are chosen for their high energy density, lightweight design, low self-discharge, and ability to deliver high currents, making them ideal for mobile applications.

Figure 3.8: Lithium-ion battery

3.1.8 Step-Down

Because the Raspberry Pi requires a stable 5 V / 3 A supply, the design uses a high-efficiency synchronous buck (step-down) converter to regulate the battery voltage to 5 V. The converter accepts a wide 7–24 V DC input range and provides a regulated 5 V output capable of 3 A continuous current (4 A peak), making it suitable for powering the Raspberry Pi with margin for transient loads. The converter features up to 96% efficiency, low ripple ($\leq$ 30 mV), precise load and line regulation, and includes short- circuit protection for safety.

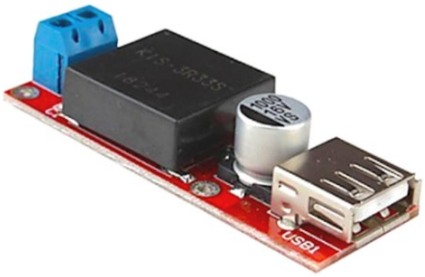


Figure 3.9: Step-Down Converter

Electrical specifications:
- Topology: Synchronous buck converter
- Input voltage: 7 – 24 V DC
- Output voltage: 5 V DC
- Output current: 3 A continuous, 4 A peak
- Efficiency: up to 96%
- Switching frequency: 340 kHz
- Output ripple:$\leq$ 30 mV
- Load regulation: $\pm$ 0.5%
- Voltage regulation: $\pm$ 2.5%
- Protection: 5 A short-circuit protection
- Operating temperature: $-40\,^{\circ}$C to $+85\,^{\circ}$C

3.1.9 ADC Module- ADS1115

This module will be used to monitor the battery charge and ensure that the voltage is sufficient to power the Raspberry Pi under its nominal conditions. It features 4 analog inputs with 16-bit resolution and supports up to 860 samples per second, significantly improving the precision compared to typical microcontroller ADCs, which usually offer 10– 12 bits. Input voltages from 2V to 5.5V are supported, and a programmable gain amplifier (PGA) allows the signal to be amplified up to 16× for more accurate readings of low-voltage signals. All of this is achieved using only the I2C communication bus. For accurate battery measurement, since the analog input can only handle up to 6.144 V (as specified in the datasheet), a voltage divider must be implemented to ensure the input voltage remains within the ADC's measurable range, according to the following equation.

$$\mathbf{Vout} = \mathbf{Vin} * \frac{\mathbf{R2}}{\mathbf{R1} + \mathbf{R2}} \tag{3.2}$$

Figure 3.10: Analog to digital converter- ADS1115

3.2 Hardware COTS

Most of the hardware introduced in the previous section consists of COTS components, which we are using to avoid losing time by designing and manufacturing these parts ourselves. Therefore, we are only developing the interfaces for the LEDs and button. The COTS components being used are the ADC, Step-Down converter, Power Supply unit, Emergency vehicle battery, RFID module and Buzzer module.

3.3 Hardware connections

This section describes the hardware connections implemented for both the Intersection Control Box and the Emergency Vehicle Unit. It details how the different components are interconnected, including the specific Raspberry Pi 4 GPIO pins used, the power supply configuration, and the communication interfaces between modules. The purpose of this section is to provide a clear understanding of the physical setup that enables data exchange and system control. Proper hardware integration ensures reliable operation, stable power distribution, and accurate signal transmission between all units involved in the intersection management system.

Intersection Control Box

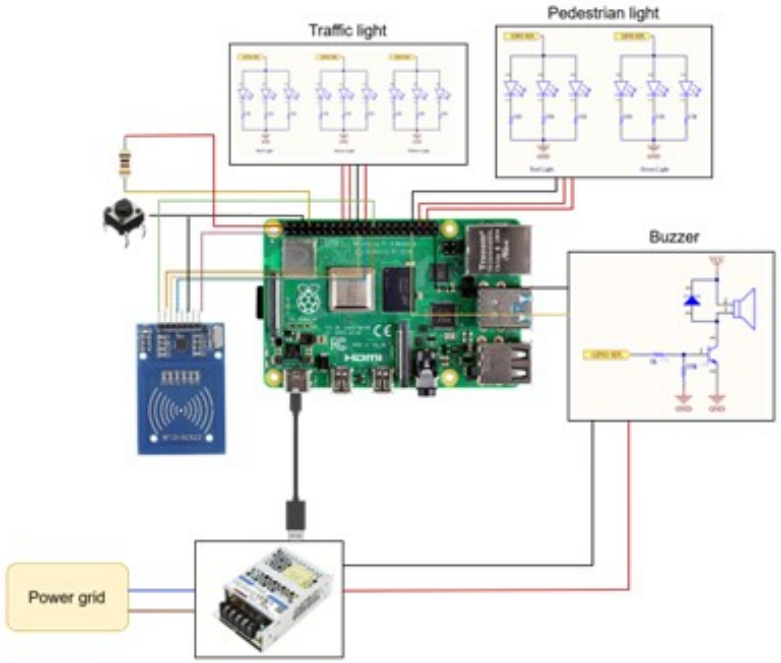


Figure 3.11: Intersection Control Box Hardware connections

GPIO	Description
10	MOSI for SPI communication - RFID module
9	MISO for SPI communication - RFID module
11	SCLK for SPI communication - RFID module
13	PWM1 to control Buzzer's frequency
16	Output pin to turn on/off green pedestrian light
20	Output pin to turn on/off red pedestrian light
8	CE0 for SPI communication - RFID module
25	Output pin to turn on/off red traffic light
24	Output pin to turn on/off green traffic light
23	Output pin to turn on/off yellow traffic light
14	Input pin for button read

Table 3.2: Used GPIOs for Intersection Control Box

Emergency Vehicle

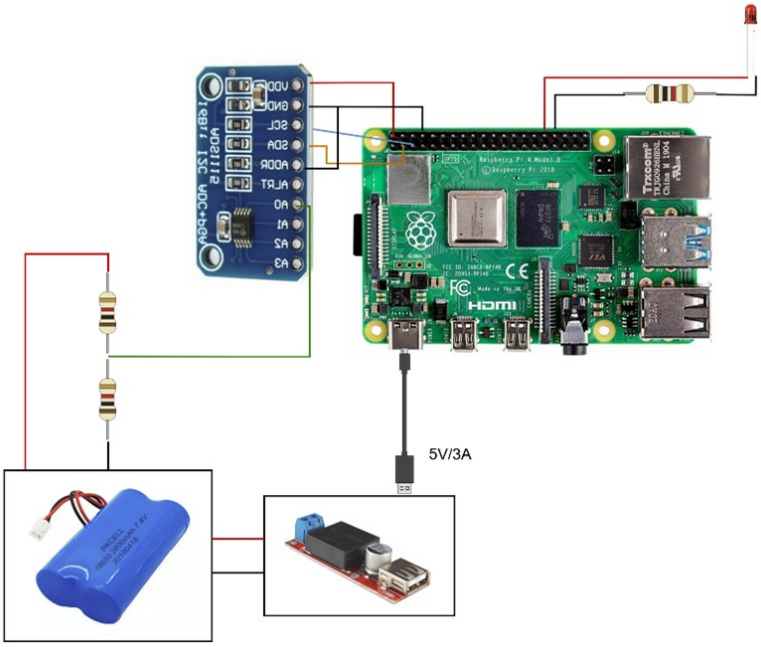


Figure 3.12: Emergency vehicle hardware connections

GPIO	Description
2	SDA for I2C communication - ADC module
3	SCL for I2C communication - ADC module
12	PWM0 to control blinking LED frequency

Table 3.3: GPIOs used for Emergency Vehicle module

3.4 3D Printed Components

In order to simulate a real-life situation and simplify the project's showcasing, 3D designs (Figures 3.13 and 3.14) were designed for the traffic and pedestrian semaphores, as well as for the emergency vehicle.

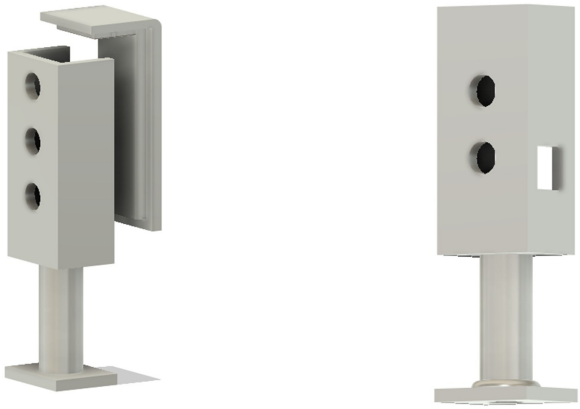


Figure 3.13: 3D of traffic and pedestrian semaphores

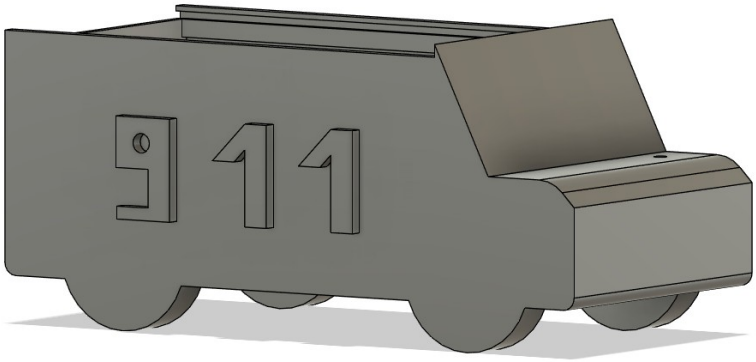


Figure 3.14: 3D of emergency vehicle

3.5 Software Components

3.5.1 Software Tools

Below is a list of the tools that will be used during the project, along with a brief description of each:

- **Buildroot** : A build system used to generate a custom, minimal Linux distribution for the Raspberry Pi 4. It allows configuration and compilation of the kernel, bootloader, root filesystem, and required packages into a deployable image.
- **Draw.io** : A diagramming tool used to create flowcharts, architecture diagrams, UML class diagrams, and entity relationship diagrams for software and hardware documentation.
- **Altium Designer** : A professional-grade PCB CAD tool used to design the schematics and PCB layouts for custom boards and prototyping shields used in the project.
- **Git** : A distributed version control system used to track changes across all project assets such as source code, firmware, hardware design files, diagrams, and documentation.
- **CLion** : Primary IDE for C/C++ and scripting.
- **Logic analyser software** : To capture and visualize digital signals, such as GPIO, I2C, timing from ADC, and more.

- **Figma** : Software dedicated to UI design.
- **Fusion 360** : A 3D design software to create 3D components of the emergency vehicle and the traffic and pedestrian semaphores.
- **Notion** : Software used for scheduling and task management between team members, providing a friendly user interface and simple organization methods.
- **Supabase** : Backend-as-a-Service platform used to manage databases, authentication, and real-time features for project applications.
- **VS Code** : Lightweight and extensible code editor used for editing, debugging, and managing source code.
- **React + Vitae** : Frontend framework and UI component library used to build interactive and responsive user interfaces.

3.5.2 Software COTS

We leverage COTS software to reduce development time, ensure reliability, and take advantage of existing tested solutions. COTS components provide essential functionalities that would otherwise require significant effort to develop from scratch, allowing us to focus on system integration and customization for our specific requirements:

- **Pthreads** : Provides multi-tasking mechanisms, which are especially important given the asynchronous nature of the system.
- **SPI kernel driver** : Enables communication with SPI-connected peripherals through a standardized and well-tested Linux kernel interface.
- **I2C kernel driver** : Facilitates interaction with I2C sensors and devices by abstracting bus access and ensuring consistent data exchange.
- **GPIO kernel driver** : Native Linux kernel support for controlling general-purpose input/output pins.
- **libgpiod** : A modern user-space C library (with accompanying command-line tools) providing access to GPIOs via the character device interface.
- **FastDDS** : Open-source Data Distribution Service (DDS) implementation designed for real-time, reliable, and scalable communication between distributed systems.
- **Wi-Fi kernel driver** : Provides wireless connectivity by enabling the Broadcom Wi-Fi interface.
- **PostgreSQL** : PostgreSQL is a powerful, open-source relational database management system (RDBMS) known for its reliability, extensibility, and support for advanced SQL features. It is widely used in both small and large-scale applications and provides robust support for JSON, complex queries, and data integrity.

3.6 Cloud Specification

In order to correctly specify the relationships between different entities, their attributes and Primary and Foreign Keys (PK and FK) in the Relational Database, a Cloud Specification based on a more advanced ERD than the one on Figure 2.9 is required.

3.6.1 Cloud Database Specification

In Figure 3.15, we can see the different entities, specified by the different colours, as well as the cardinality between them, already previously mentioned. However, now it is possible to outline the attributes and their PK and FK. These will be crucial in the organization of the database structure.

The cardinality of the associations between EmergencyVehicle and ControlBox, and between P_Semaphore and Pedestrian, is many-to-many and optional on both sides. Consequently, these relationships must be implemented using junction tables, which allow each entity to be linked to multiple instances of the other without redundancy or violation of normalization rules (Figure 3.15– entities in orange).

The intermediary tables establish a connection between the entities, without requiring a compulsory connection between them: the Control Box doesn't necessarily have Emergency Vehicles passing through it, but if there are, it must register its properties. Also, an Emergency Vehicle does not necessarily belong to only one Control Box. The same applies to the relationship between P_Semaphore and Pedestrian. Therefore, the intermediary table helps with this management.

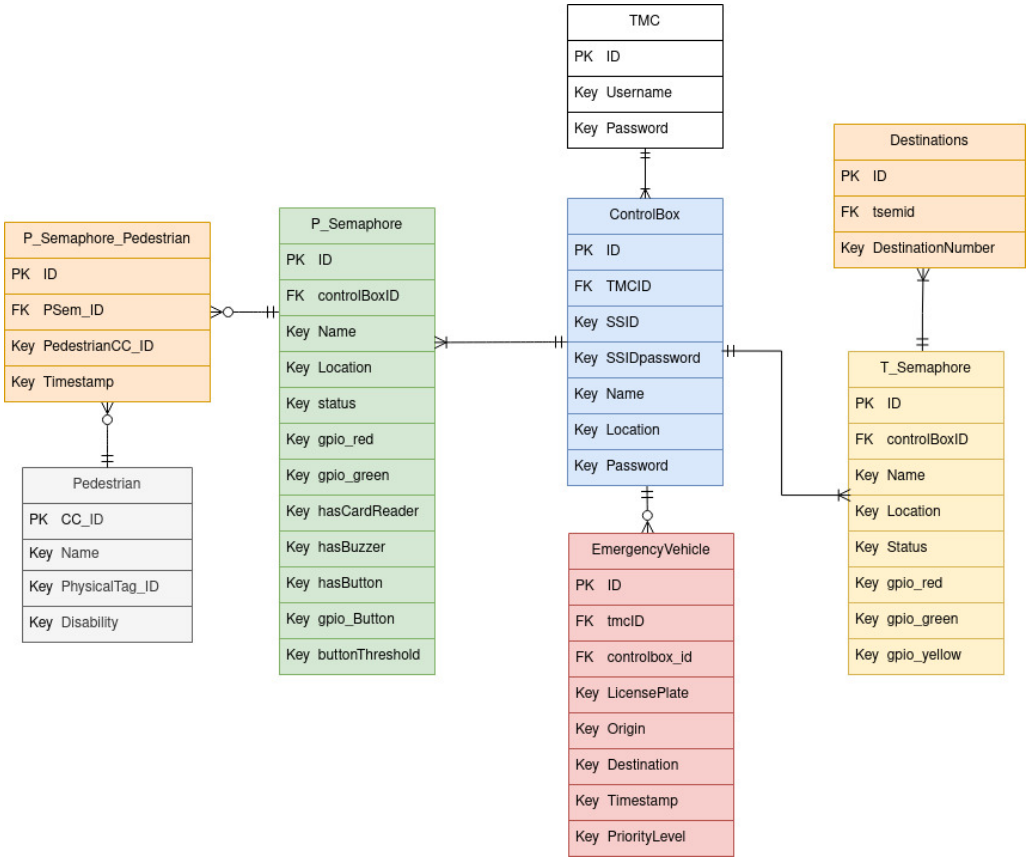


Figure 3.15: ERD with junction tables

A brief description of the entities, their PKs, FKs and attributes.

TMC (Traffic Management Center)

TMC entity.
- ID: Internal identification of the Traffic Management Center;
- Username: Traffic Management Center username to log in;
- Password: Traffic Management Center password to log in.

ControlBox

Control Box entity.
- ID: Internal identification of a Control Box (identification of the intersection);
- TMCID: Foreign key referencing the TMC to which this Control Box belongs;
- SSID: Network's name;
- SSIDpassword: Network's password;
- Name: User defined name for the Control Box;
- Location: Specification of the address where the Control Box is located;
- Password: Control Box password for authentication.

Destinations

Destinations entity.
- ID: Internal identification of the destination definition;
- tsemID: Foreign key referencing the Traffic Semaphore associated with this destination;
- DestinationNumber: Number of the destination.

T_Semaphore

Traffic Semaphore entity.
- ID: Internal identification of a Traffic Semaphore;
- controlBoxID: Foreign key referencing the Control Box to which it belongs;
- Name: User defined name for Traffic Semaphore;
- Location: Location in the intersection;
- Status: Current colour (Red, Yellow or Green);
- gpio_red: GPIO pin number for red light;
- gpio_green: GPIO pin number for green light;
- gpio_yellow: GPIO pin number for yellow light.

P_Semaphore

Pedestrian Semaphore entity.
- ID: Internal identification of a Pedestrian Semaphore;
- controlBoxID: Foreign key referencing the Control Box to which it belongs;
- Name: User defined name for Pedestrian Semaphore;
- Location: Location in the intersection;
- status: Current colour (Red or Green);
- gpio_red: GPIO pin number for red light;
- gpio_green: GPIO pin number for green light;
- hasCardReader: Boolean indicating if the semaphore has a card reader;
- hasBuzzer: Boolean indicating if the semaphore has a buzzer;
- hasButton: Boolean indicating if the semaphore has a button;
- gpio_Button: GPIO pin number for the button;
- buttonThreshold: Threshold value for button activation.

Pedestrian

Pedestrian entity.
- CC_ID: Citizen Card ID (therefore, unique for every person);
- Name: Name of the Pedestrian;
- PhysicalTag_ID: Attributed Card ID;
- Disability: Degree of priority in disability.

P_Semaphore_Pedestrian

P_Semaphore_Pedestrian relationship entity.
- ID: Internal identification of the P_Semaphore_Pedestrian interaction;
- PSem_ID: Foreign key referencing the P_Semaphore present in this interaction;
- PedestrianCC_ID: Foreign key referencing the Pedestrian present in this interaction;
- Timestamp: When the interaction occurred.

EmergencyVehicle

Emergency Vehicle entity.
- ID: Internal identification of an Emergency Vehicle interaction;
- tmcID: Foreign key referencing the TMC;
- controlbox_id: Foreign key referencing the ControlBox present in this interaction;
- LicensePlate: License Plate as identification of the vehicle that sent an emergency message;
- Origin: From where it is coming;
- Destination: Where it is going;
- Timestamp: When the interaction occurred;
- PriorityLevel: Priority of the emergency.

3.6.2 Cloud Interface

To enable secure and controlled communication between the system's endpoints, an intermediary service layer must be established. This layer will serve as the central access point, ensuring safety, providing a means for managing interactions, and enabling system extensibility. To accomplish this, a RESTful API should be implemented, with communication carried out through standard HTTP requests.

Cloud API Endpoints

A well-defined Cloud API endpoint structure is fundamental to the system architecture, as it establishes the interaction model between different entities and clarifies data exchange processes. The API follows RESTful principles, utilizing standard HTTP methods (GET, POST, PATCH) to perform operations on resources, and returns appropriate status codes (200 for success, 400 for client errors, 404 for not found, 5xx for server errors) to indicate the outcome of each request.

The endpoint design encompasses several key functional areas: traffic management control (TMC), control box operations, pedestrian semaphore management, and emergency vehicle coordination. Each endpoint is designed with specific input parameters and structured response formats to ensure consistent and reliable communication across the distributed system.

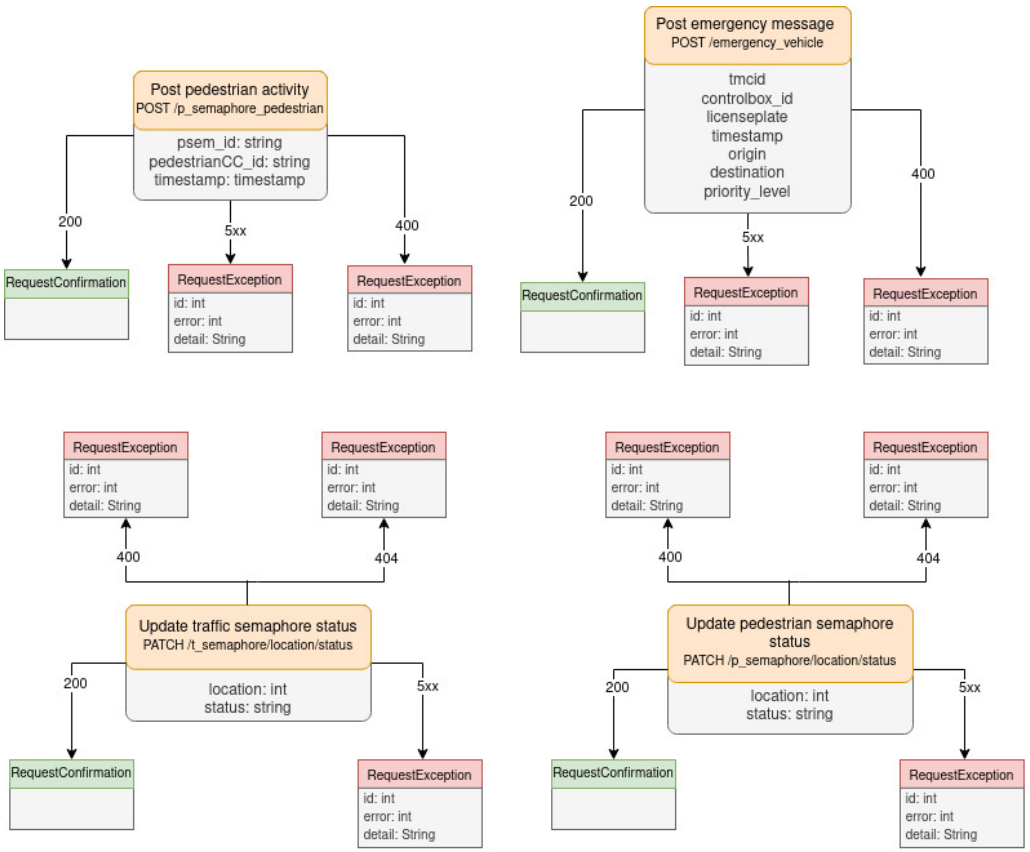


Figure 3.16: POST and PATCH endpoints for data creation and modification operations

POST and PATCH Endpoints

The POST and PATCH endpoints handle data creation and modification operations respectively, as shown in Figure 3.16. These endpoints are crucial for:

- **Activity logging**: Recording pedestrian activities at semaphores with timestamps and location data, enabling comprehensive tracking of system usage patterns.
- **Emergency messaging**: Posting emergency vehicle messages with priority levels, origins, destinations, and license plate information to coordinate traffic flow during critical situations.
- **Status updates**: Modifying the operational status of both traffic and pedestrian semaphores through PATCH operations, allowing dynamic control of the traffic management system.

These modification endpoints return either a 200 status code confirming successful operation, or error responses (400, 5xx) with detailed exception information to facilitate debugging and error handling in client applications.

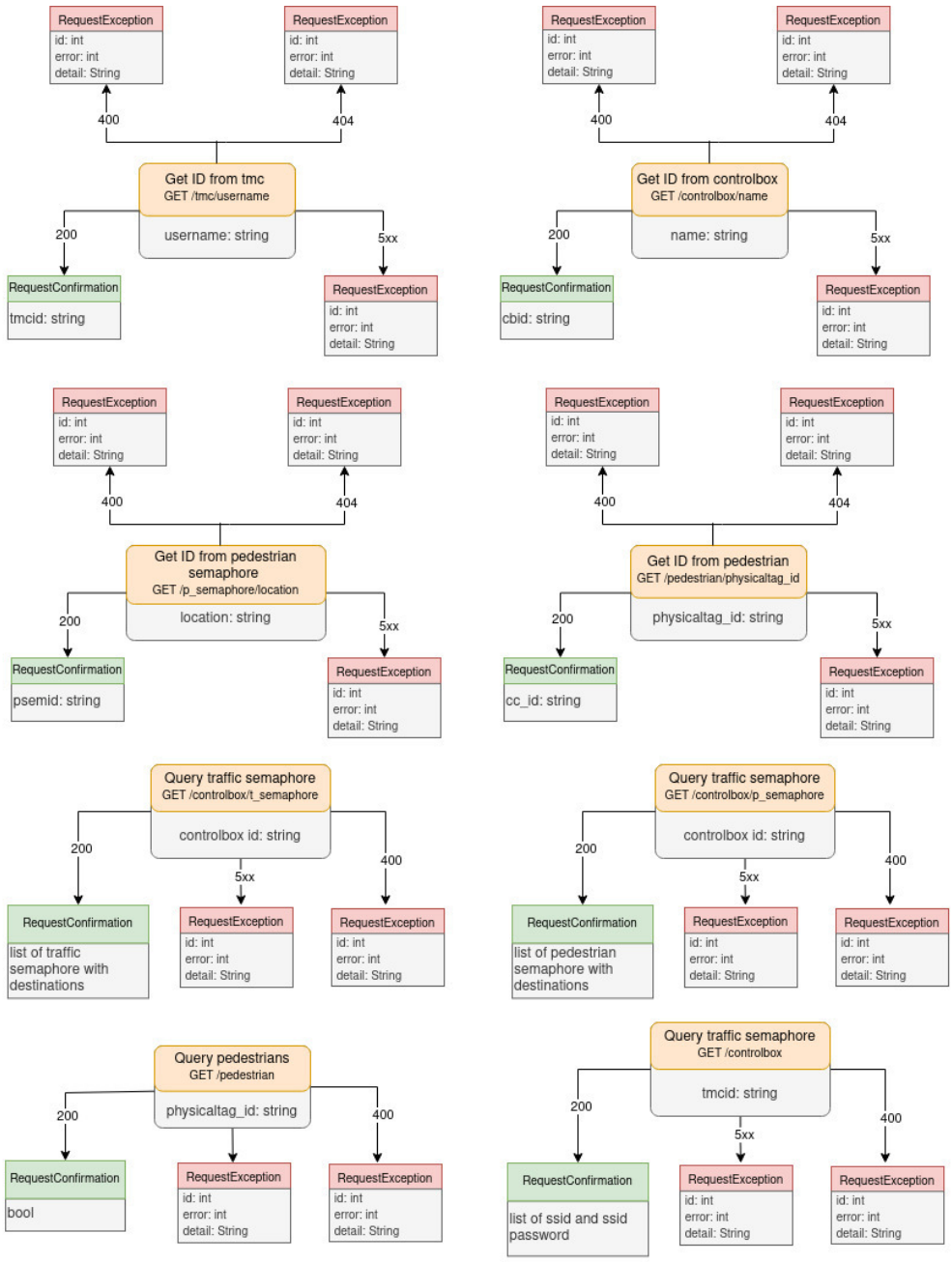


Figure 3.17: GET endpoints for querying system resources and state information

GET Endpoints

The GET endpoints provide read-only access to system resources and are designed to retrieve current state information without modifying data. As illustrated in Figure 3.17, these endpoints support various query operations:

- **Identity retrieval endpoints**: Fetch unique identifiers for TMC users, control boxes, and pedestrian semaphores based on their respective attributes (username, name, location, or physical tag ID).
- **Traffic semaphore queries**: Retrieve real-time traffic semaphore information filtered by control box ID or specific semaphore location, returning comprehensive traffic data including current states and destinations.

- **Pedestrian semaphore queries**: Access pedestrian semaphore data either by physical tag ID or by querying all semaphores, supporting both targeted and bulk data retrieval operations.

All GET endpoints return either a 200 status code with the requested data, or appropriate error codes (400, 404, 5xx) with structured exception details when resources cannot be found or request parameters are invalid.

The endpoint architecture ensures separation of concerns, with clear delineation between read and write operations, and provides a robust foundation for building scalable and maintainable traffic management applications.

Expected JSON output

For system setup and configuration, a **RESTful API** is used to initialize and manage the control boxes and their associated peripherals. During the setup phase, the API is expected to receive a JSON payload describing the hardware configuration and operational parameters of each control unit.

The API expects data in a structured format, where each object represents a control box or peripheral node. Initially, the configuration template contains uninitialized fields, which are later populated with concrete values provided by the system administrator or a configuration service.

Listing 3.1: Expected configuration template for system setup

```
[
  {
    "controlboxid": "",
    "destinations": [],
    "gpio_green": null,
    "gpio_red": null,
    "gpio_yellow": null,
    "id": "",
    "location": "",
    "name": "",
    "status": null
  }
]

[
  {
    "buttonclicks": null,
    "controlboxid": "",
    "gpio_green": null,
    "gpio_red": null,
    "id": "",
    "location": null,
    "name": "",
    "status": null,
    "hasButton": null,
    "hasCardReader": null,
    "hasBuzzer": null,
    "buttonThreshold": null,
    "gpio_button": null
  }
]
```

Once the configuration data is submitted, the API processes the request and stores a fully populated configuration object. Each field in the configuration object defines a specific aspect of the system, such as GPIO pin assignments, peripheral availability, physical location, and operational thresholds. By using a RESTful interface and a standardized JSON structure, the system allows flexible remote configuration, simplifies deployment, and enables seamless integration with external management tools.

This approach ensures that all hardware parameters are explicitly defined during initialization, reducing configuration errors and improving system reliability.

3.7 Fast-DDS Specification

To enable reliable and time-critical communication between the Emergency Vehicle and the Intersection Control Box, the system adopts the Fast DDS middleware as its primary communication framework. In this architecture, the Emergency Vehicle acts as the Publisher, responsible for broadcasting alert notifications about its presence and priority, while the Control Box acts as the Subscriber, responsible for receiving those alerts, adapting traffic light behaviour accordingly, and updating the cloud with status information. Communication between both entities occurs over a Wi-Fi network created by the Control Box, which serves as the access point, that will be explained in the next section. When the Emergency Vehicle approaches the intersection, its onboard communication unit scans for available Wi-Fi networks and connects to the Control Box's access point.

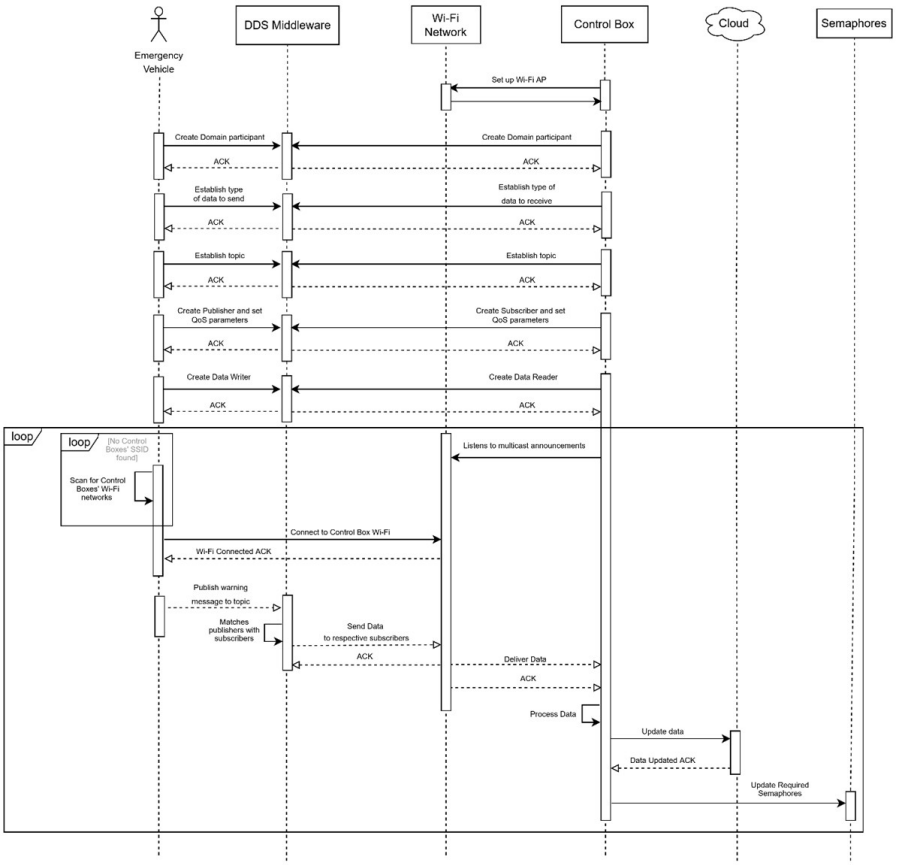


Figure 3.18: Stages of FastDDS communication

Once connected, both systems join the same DDS Domain, which represents a logical communication space where publishers and subscribers can interact. Within the same domain, the Fast DDS middleware automatically initiates the discovery process using the RTPS protocol. RTPS operates primarily over UDP multicast, allowing all participants to

broadcast and listen to discovery messages without requiring a centralized broker. During this process, each participant advertises:
• Its unique identifier (GUID Prefix);
• The list of Topics it publishes or subscribes to;
• The associated Quality of Service (QoS) parameters.

This decentralized discovery ensures that once the Emergency Vehicle's DDS participant announces its intention to publish on a specific topic, the Control Box's subscriber middleware will detect this and automatically establish communication if their topic names, data types, and QoS profiles are compatible.

In this project, a topic named "EmergencyAlert" is created to carry messages from the Emergency Vehicle to the Control Box. The corresponding data type is defined in an IDL (Interface Definition Language) file. Both the publisher and subscriber register this data type in their respective DDS domain participants. This step ensures that when the middleware compares messages, it can confirm that the data structures and serialization formats are compatible, preventing runtime communication errors. When the Emergency Vehicle detects that it is approaching an intersection, it constructs an EmergencyAlert message and uses its DDS DataWriter to publish the data to the "EmergencyAlert" topic.

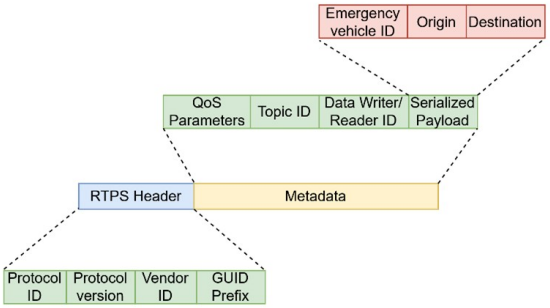


Figure 3.19: Frame created in DDS middleware

At this point, the DDS middleware takes over and performs several important internal operations before the message is transmitted, as we can confirm in Figure 3.19. It serializes the structured data (using CDR – Common Data Representation) into a byte stream suitable for transmission. Encapsulates the serialized data within an RTPS message.

Afterwards is adds transport-level headers according to the UDP protocol stack. Broadcasts the message to the network, using UDP multicast or unicast depending on the configured QoS and discovery mode. This encapsulation process is crucial: it allows the DDS middleware to include all metadata required for the automatic matching between publishers and subscribers, without the application having to manage IP addresses or sockets manually.

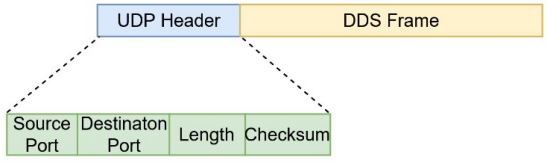


Figure 3.20: Frame sent through UDP

On the Control Box side, the DDS DataReader continuously listens for incoming RTPS messages within the same DDS domain. When a packet is received, the middleware automatically verifies its metadata: It checks whether the Topic ID and data type correspond to any existing subscriptions. It evaluates the QoS policies to confirm compatibility between

publisher and subscriber (e.g., reliability mode, liveliness, durability). Once compatibility is confirmed, a matching is established between the DataWriter and DataReader. The middleware then deserializes the payload and passes the reconstructed data structure to the application layer. From this point onward, the subscriber application can directly access the alert message content and react accordingly. Upon receiving the emergency alert, the Control Box interprets the data and adjusts the traffic light cycles to create a clear path for the emergency vehicle, prioritizing its direction of travel. After executing this action, the Control Box also transmits an update to the cloud system, logging the event and providing real-time monitoring of intersection status and emergency vehicle movement. This ensures both local reaction (for immediate safety) and centralized awareness (for city-wide traffic management and analytics).

3.8 Wi-fi Specification

The communication between the Emergency Vehicle and the Intersection Control Box relies on Wi-Fi (IEEE 802.11) as the underlying wireless communication technology. Wi-Fi provides the physical and link layers of the network, enabling both devices to connect to a shared local network and exchange IP packets without the need for wired connections. In this system, the Intersection Control Box acts as the Wi-Fi Access Point, broadcasting its network SSID and handling the connection of nearby devices. The Emergency Vehicle connects to this Wi-Fi network as a client, obtaining a local IP address through DHCP or static configuration. Once both devices are connected to the same network, they can communicate directly at the IP level. The Wi-Fi standard operates using radio frequencies (typically 2.4 GHz or 5 GHz bands) and provides sufficient bandwidth and range to support low-latency data transmission. It serves purely as the transmission medium, carrying packets generated by higher-level protocols such as IP, UDP, and ultimately DDS/RTPS.

The Data Distribution Service (DDS) middleware used in this project runs on top of the IP and UDP layers. When the Emergency Vehicle publishes an emergency alert message, the DDS middleware encapsulates the message using the RTPS protocol, which is then transported by UDP. UDP packets are subsequently encapsulated in IP packets and finally transmitted over Wi-Fi as radio signals. This hierarchical communication stack, based on the OSI Model, can be represented as follows:

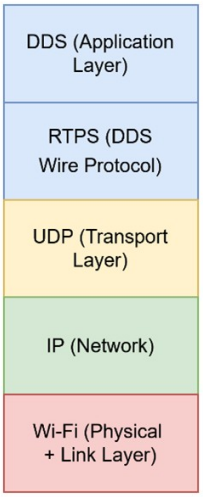


Figure 3.21: Communication stack based on OSI model

Through this structure, Wi-Fi provides the physical channel over which the DDS-based communication occurs. It ensures network connectivity and packet delivery at the radio level, while the DDS middleware (via UDP/RTPS) ensures that messages are properly formatted,

discovered, and delivered between participants. Because Wi-Fi supports broadcast and multicast communication, it is especially well suited for DDS, which uses UDP multicast for automatic discovery of publishers and subscribers within the same domain. This allows the Emergency Vehicle to be discovered by the Control Box dynamically as soon as it connects to the Wi-Fi network, without requiring any manual configuration of IP addresses. As shown in the sequence diagram below (Figure 3.22), the process represents how the Emergency Vehicle connects to the Wi-Fi Access Point (AP) of the Intersection Control Box. The Access Point continuously broadcasts beacon frames containing its SSID (network name) to announce the availability of the network to nearby devices.

Once the Emergency Vehicle detects the network and successfully authenticates, both devices are connected at Layer 2 (link layer). At this stage, the physical and data-link connection is established, but the vehicle does not yet have an IP address.

After completing the DHCP process, the Emergency Vehicle receives its network configuration and now has an IP address, allowing it to send and receive data over the Wi-Fi connection.

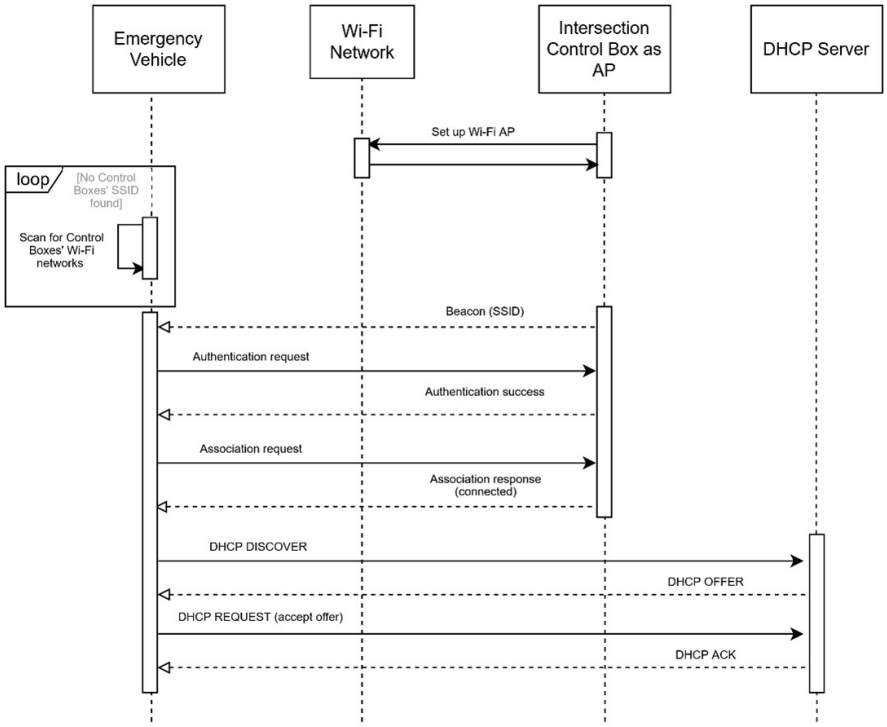


Figure 3.22: IP acquisition process for Emergency Vehicle

3.9 Software Specification

3.9.1 Package Diagram

The package identifies which are the responsibilities of the system and indicate which components work together in order to fulfil each responsibility. The Control Box represents the embedded controller of the physical traffic system. It has its own hardware, software and responsibility. The Emergency Vehicle Unit is the embedded system inside the emergency vehicle, operating independently and sending alerts. The Cloud System makes up the monitoring station includes the backend server storing data and the API for doing so and the user interface for city operators. It runs separately from the main system itself.

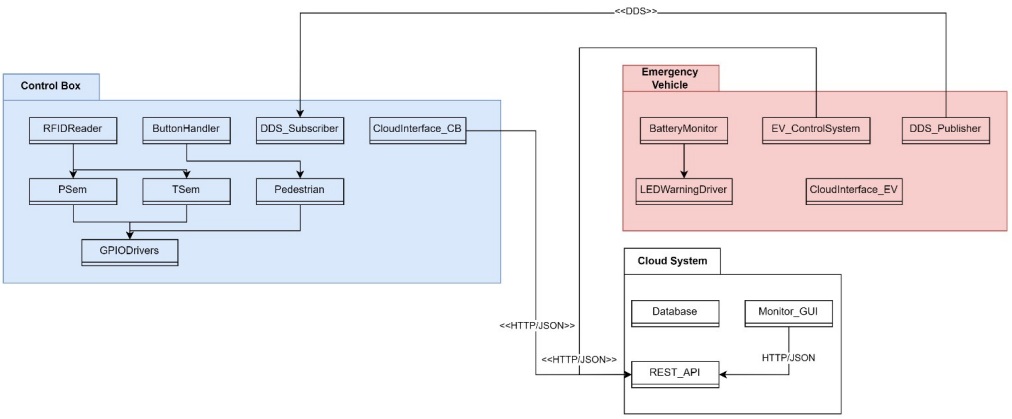


Figure 3.23: Specified Package Diagram

3.9.2 Class Diagrams and Patterns

In the class diagrams shown in Figure 3.25 and Figure 3.27, we can observe the classes that compose the two different systems — the Intersection Control Box and the Emergency Vehicle, respectively.

Taking into consideration that `pthreads` will be used and it does not comply with RAII programming technique, a C++ wrapper must be created so as to enable that the lifetime of a resource is bound to the lifetime of an object, avoiding memory leaks and bugs.

During the design process, the SOLID principles were carefully applied to ensure maintainability, scalability, and a clear separation of responsibilities. In addition, the use of design patterns was deliberately planned to standardize code modularization and define well-structured inter-class relationships.

In the Control Box class diagrams, each class has a single responsibility within the system (S) and implements a structure which makes it easily open to extension (O). Moreover, with the "Semaphore" interface, subclasses can replace their base class without altering the system behaviour (L) and helps promoting the implementation of smaller and more specific interfaces instead of general one (I). Additionally, it adds dependency on abstraction rather than on concrete implementations (D), considering the application for which it is designed. The emergency vehicle also partially implements this principle, since it requires adaptation for the specific purpose of the application it was designed for.

The Emergency Vehicle Control System (Figure 3.27) complements this approach by enforcing a strict separation between hardware-level interfacing and application logic. The Single Responsibility Principle (S) is evident in the isolation of the `ADS1115` and `PWM_DeviceDriver` classes, which manage low-level sensor and signal data independently of the `Battery` monitoring logic. The system demonstrates Dependency Inversion (D) and Open/Closed (O) principles through the use of the `PWM_DeviceDriver` wrapper; this abstraction ensures that the battery management logic is decoupled from the specific Linux kernel driver (`pwm_dd`) implementation. Furthermore, by utilizing specialized wrappers such as `CloudInterface` and `DDSPublisher`, the system adheres to Interface Segregation (I) and Liskov Substitution (L), ensuring that the high-level `EVControlSystem` interacts only with well-defined data contracts and communication protocols without being exposed to unnecessary implementation complexities.

C++ Wrapper

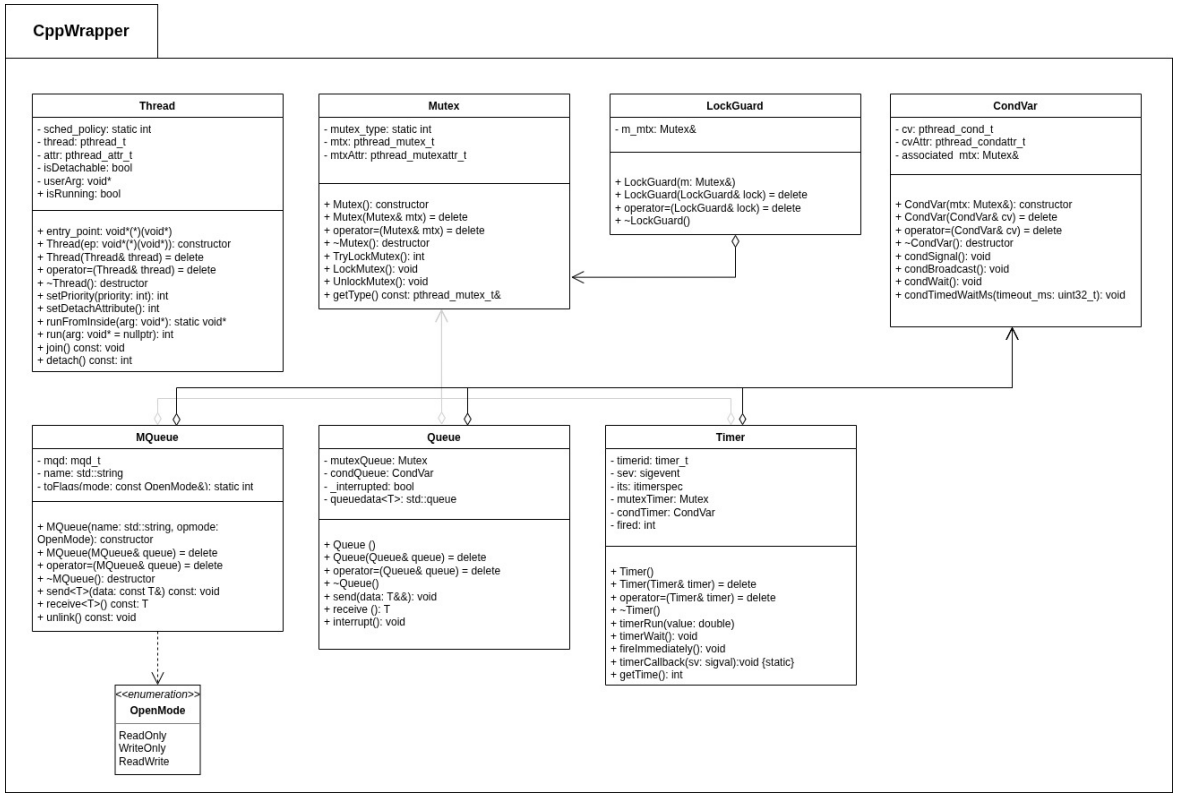


Figure 3.24: C++ namespace

C++ Wrapper

Foundation

The C++ Wrapper (`CppWrapper`) focuses on the application of RAII to POSIX threads, POSIX timers and POSIX IPC relevant mechanisms and APIs for the project. It also aims for a new design on the C++ already existent `queue`, enabling thread-safe mechanisms.

The POSIX IPC (in specific, the mutex and the condition variable) were chosen over the C++ already existing mutex (`std::mutex`) and condition variable (`std::condition_variable`) in order to explore the POSIX offerred APIs, even though there is no significant advantage for each considering the application.

The wrapper of the Message Queue (`MQueue`) aims to explore the POSIX message queue mechanism, which provides interprocess communication. However, the C++ `std::queue`-based implementation offers more flexibility regarding the types of data that can be sent, since it does not require the stored objects to be trivially copyable, unlike POSIX message queues, which operate on raw byte buffers and impose this restriction at the ABI level.

In order to avoid conflicts between the already existing APIs, the namespace CppWrapper is required.

Classes Overview

Thread A wrapper for `pthread` functionality. It manages the lifecycle of a system thread, including configuration (attributes, priority), execution (`run`), and termination (`join`, `detach`).

Mutex Encapsulates a `pthread_mutex_t`. It provides fundamental synchronization primitives for mutual exclusion, supporting locking, unlocking, and non-blocking `TryLock` attempts.

LockGuard Implements the RAII (Resource Acquisition Is Initialization) pattern for the `Mutex` class. It automatically acquires a mutex lock upon construction and releases it upon destruction to ensure exception safety.

CondVar Wraps `pthread_cond_t` to provide condition variable signaling. It allows threads to wait for specific conditions or broadcast signals to multiple waiting threads, working in conjunction with a Mutex.

MQueue A wrapper for POSIX message queues (`mqd_t`). It facilitates inter-process or inter-thread communication through a named messaging system with specific **OpenMode** permissions.

Queue A thread-safe, t**emplate-based producer-consumer** queue. It uses an internal Mutex and Condition Variable (CondVar) to synchronize data exchange between threads. The condition variable ensures threads are notified only when necessary, avoiding *spurious wake-ups* and maintaining system consistency.

Timer Manages POSIX timers (`timer_t`). It allows for scheduling events or callbacks based on time intervals, utilizing `sigevent` structures for notification.

Intersection Control Box

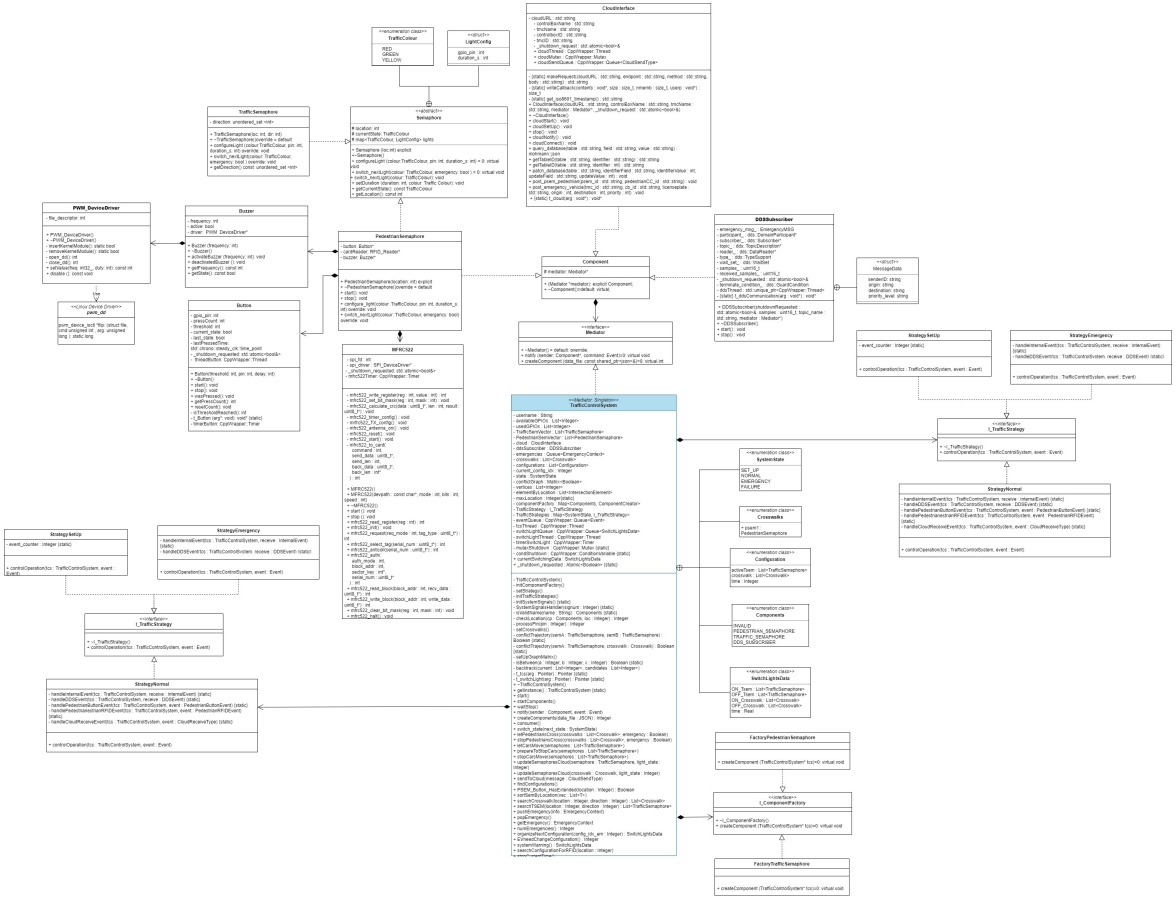


Figure 3.25: Intersection Control Box Class Diagram

In the Intersection Control Box system, each class is responsible for a well-defined aspect of traffic management. Through their collaboration, the system enables both local and remote control of vehicle and pedestrian signals, while supporting system-wide data exchange and event propagation.

TrafficControlSystem The central orchestrator of the traffic control architecture. It coordinates all traffic and pedestrian semaphores, manages system operation modes, and handles interaction with pedestrians through RFID cards and physical buttons. The system integrates cloud connectivity via the `CloudInterface` and distributed communication through a DDS messaging layer, enabling both local autonomy and remote supervision.

TrafficSemaphore Represents a vehicle traffic light associated with a specific intersection location and a set of directions. It provides an interface for controlling light state transitions (red, yellow, green) and their timing parameters, as well as mechanisms for overriding normal behavior in special traffic conditions.

PedestrianSemaphore Controls a pedestrian traffic signal with visual and audio feedback. It manages the pedestrian request button, regulates buzzer activation and frequency, interfaces with the RFID card reader (`MFRC522`), and supports dynamic overrides of signal duration and state based on current traffic conditions.

Button Models a physical push button connected to a GPIO pin. It monitors press events, applies a configurable activation threshold, tracks press counts, and determines when the threshold is reached in order to trigger corresponding system actions.

Buzzer Represents a physical buzzer associated with a pedestrian semaphore. The buzzer is activated when pedestrian crossing is permitted and is driven through a dedicated PWM interface implemented by the `PWM_DeviceDriver`.

PWM_DeviceDriver Provides low-level control of the PWM hardware used by the buzzer. It allows configuration of frequency and duty cycle and interfaces directly with the hardware through the `ioctl` system call.

MFRC522 Interfaces with the RFID card reader hardware via SPI communication. It reads RFID card identifiers and maintains a record of the most recently scanned card.

CloudInterface Manages communication with a remote cloud service using HTTP. It establishes and maintains connectivity, performs entity write and query operations, and continuously verifies the connection status.

DDSSubscriber Implements a DDS subscriber for distributed messaging. It joins a DDS domain with a specific participant identifier, subscribes to configured topics, receives and tokenizes messages through a data reader, manages message queues, and configures Quality of Service (QoS) parameters. This component is responsible for disseminating emergency notifications and triggering system-wide adaptations.

Supporting Structures

The `TrafficControlSystem` relies on a set of internal supporting structures that encapsulate configuration data, represent physical intersection elements, and enable coordination logic. These structures are defined within the system scope and function as internal data carriers rather than independent components.

Enumerations

SystemState
Defines the high-level operational modes of the system and drives control strategy selection. The supported states are:

- `SET_UP` – initial system initialization and configuration,
- `NORMAL` – standard traffic operation,
- `EMERGENCY` – priority handling for emergency situations.

Components
Used during dynamic component creation and validation. This enumeration provides a controlled vocabulary for supported component types, including traffic semaphores, pedestrian semaphores, and communication subscribers.

Structural Data Types

Crosswalk
Represents a pedestrian crossing by aggregating two pedestrian semaphore instances. This abstraction allows the system to manage a crossing as a single logical entity while internally coordinating multiple signaling elements.

Configuration
Models a complete traffic configuration for the intersection. It groups the set of active traffic semaphores, their associated crosswalks, and a timing parameter defining the switching duration. Configurations are precomputed and stored to enable rapid transitions between traffic patterns.

Switching Support Structures

SwitchLightsData
Encapsulates all information required to execute a light-switching operation. It explicitly separates elements to be activated from those to be deactivated for both traffic semaphores and pedestrian crosswalks, along with the associated timing value. This structure is used for queued execution and coordination of asynchronous transitions.

Intersection Element Abstraction
To support extensibility and generic handling of intersection entities, the system defines an abstract `IntersectionElement` type. This abstraction allows traffic semaphores and crosswalks to be treated uniformly during organization, scheduling, and conflict analysis.

Conflict Representation
The system maintains an internal conflict model based on an adjacency matrix that encodes trajectory conflicts between intersection elements, enabling constant-time conflict checks. A complementary virtual vertex list maps physical elements to logical graph nodes.

Role in the System Architecture
All supporting structures are owned by the `TrafficControlSystem` and are not exposed as standalone components. Their primary purpose is to simplify internal state representation, enable efficient configuration switching, support strategy-based control decisions, and facilitate reliable conflict detection and resolution. Together, they form the foundational data layer upon which the system's coordination and control logic is built.

Design Patterns in `Intersection Control Box`

Given the system requirements, the need for a design of a model which was able of efficiently and clearly managing the intersection was evident. Therefore, in the Intersection Control Box system, creational, behavioral and structural design patterns (Section 3.16.19) are applied as follows:

- Singleton
- Mediator
- Strategy
- Factory
- Facade

Singleton The Singleton design pattern was chosen for the `TrafficControlSystem` class, where it was imperative to ensure only one `TrafficControlSystem` existed in the system, since no more than one would make sense. For this, the **Meyers Singleton** variation was chosen, given its simplicuty, thread-safety (since C++11) and immunity to static initialization order bugs.

Mediator In order to ensure clear communication between the classes of the system, a "manager" is required. This "manager" must be a class who knows about all participants in the system and ensures they are not thightly coupled, therefore ensuring the single responsibility principle. The Mediator design pattern enables this behavior, by providing a "manager" — the Mediator — who mediates the communication between all system participants — Components — by offering an interface they can use to notify events which may reflect in a need for the system to alter its behavior. Thus, in this system, the interface classes `Mediator` and `Component` are proposed in order to follow the textbook Mediator design pattern design, and the class `TrafficControlSystem` implements the actual Mediator class. Here, the system components are the `PedestrianSemaphore`, `CloudInterface` and `DDSSubscriber`, because the events produced by these may reflect in actual changes in the system's behavior.

Strategy Given the different system states in which the system may be (described by the `SystemState` enumeration in 3.9.2), the system must implement different approaches of handling events. Thus, taking into considering this problem, the Strategy design pattern offers a standart way of handling these situations. A `I_TrafficStrategy` interface is provided with three different strategies (`StrategySetUp`, `StrategyEmergency`, `StrategyNormal`), which reflect on the three existing system states, providing, also, an easy expansion. Given this, for any current state, the `TrafficControlSystem` class should directly call the function which implements the handling strategy to control operation.

Factory Considering that the system should be able of automatically creating its components and their attributes, by mapping them onto physical pins and internal information, the system requires an efficent way of creating these components *on-the-fly*. The Factory design pattern, provided by the `TrafficControlSystem`, enables such structure by providing a dedicated interface for components creation, which acts differently basde on the component.

Facade When interfacing with a system, simplicity is a pivotal. Therefore, an interface that hides all the program details is preferred. This is exactly provided by the Facade design pattern, where thorough details are hidden by a simple interface. In the Intersection Control Box, both the `TrafficControlSystem` class and the `PedestrianSemaphore` class implement the Facade design pattern, by offerring the *start* and *stop* APIs to interacts with all the system's components.

Software Architectural Patterns in `Intersection Control Box`

The usage of the Mediator design pattern can be combined with an **Event-Driven Architecture** — EDA — (software architectural pattern- (Section 3.16.20)), where components communicate by emitting and reacting to events rather than invoking each other directly, enables a traffic management system that is both modular and highly responsive, allowing real-time adaptation to changing traffic conditions while maintaining loose coupling between different system modules. Thus, the implemented Mediator pattern can be called **Event-Driven Mediator** or **Asynchronous Mediator**, as it not only centralizes communication between components but also allows them to react to events asynchronously, combining the coordination benefits of the Mediator pattern with the flexibility and scalability of an Event-Driven Architecture.

Concurrency Patterns in `Intersection Control Box`

Since the system has more than one thread (Section 3.16.21) which produces events, a simple **Producer-Consumer pattern** should be implemented to enable effective data transfer via a shared element — a queue. This design ensures that event-producing threads (producers) can generate data independently of the threads responsible for processing these events (consumers), allowing the system to handle varying workloads without blocking or losing data. Proper synchronization mechanisms, such as thread-safe queues, must be used to avoid race conditions and deadlocks.

System Events

Given the Event-Driven Mediator Architecture used, the system events must be defined. Figure 3.26 showcases the various possible events.

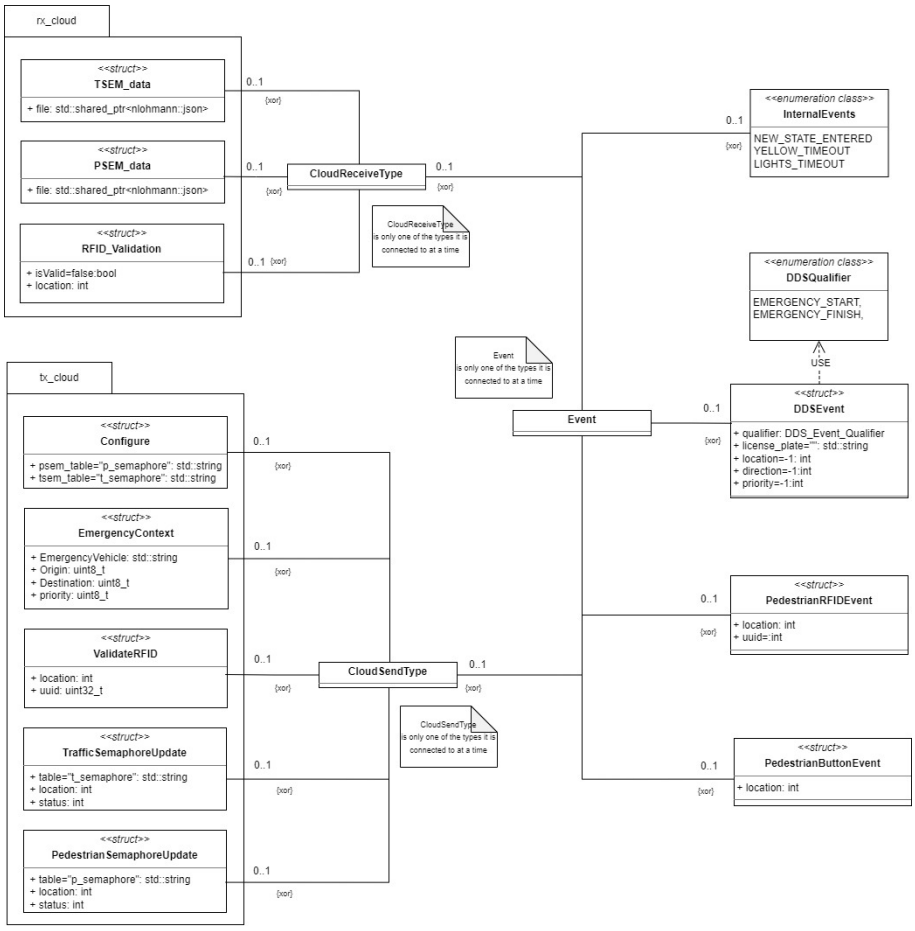


Figure 3.26: Intersection Control Box Events

Emergency Vehicle

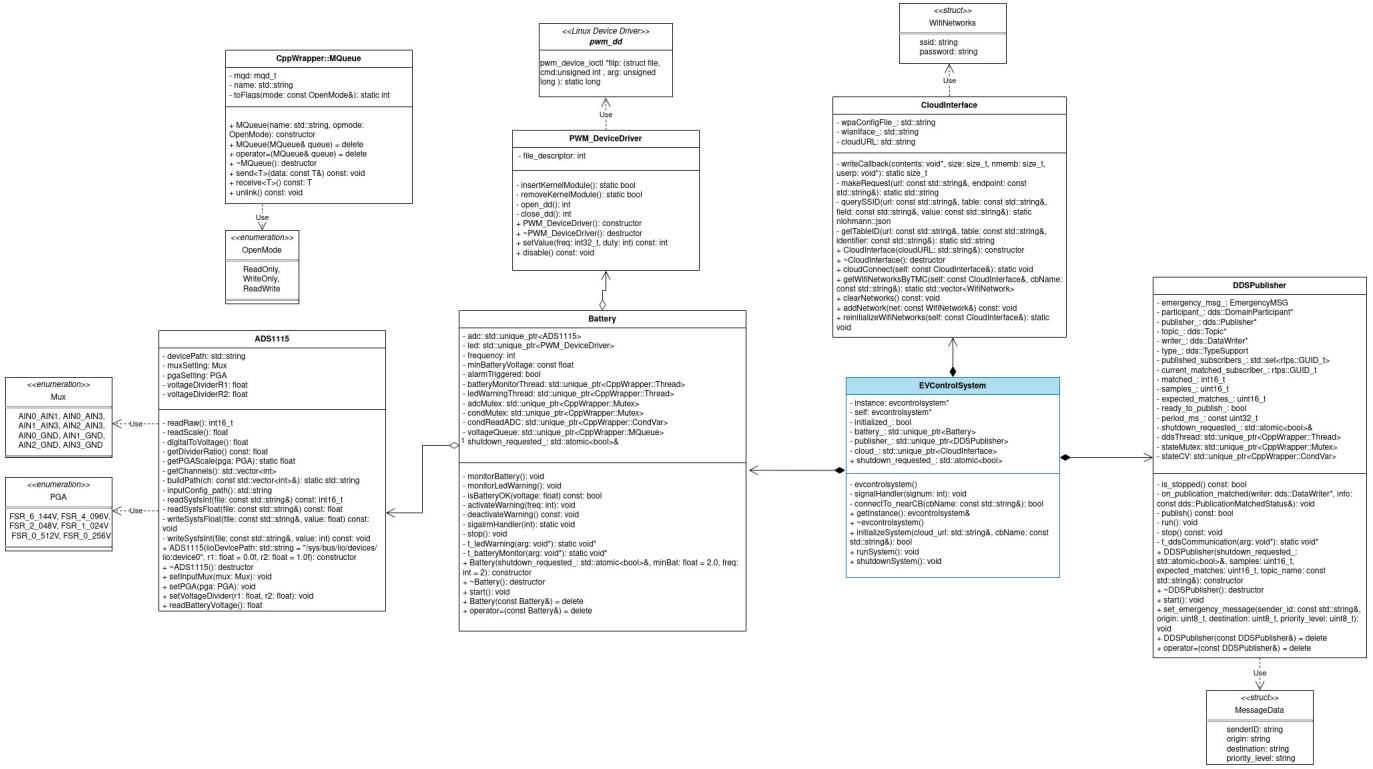


Figure 3.27: Emergency Vehicle Class Diagram

EVControlSystem: Controls the entire EV system. Manages battery, DDS communication, and cloud connectivity. Provides system initialization, execution, shutdown, and connection to the nearest Control Box.

DDSPublisher: Publishes emergency messages via DDS. Creates and configures DDS participants, publishers, topics, and data writers. Ensures reliable delivery and maintains publishing state.

Battery: Monitors battery status via ADS1115, controls warning LEDs, and activates/deactivates low-voltage alerts. Runs continuous monitoring threads.

ADS1115: Reads analog battery values via I2C, converts digital readings to actual voltage, and manages gain and multiplexer configuration.

CloudInterface: Manages cloud communication to acquire ID, query available Wi-Fi networks, and configure SSID lists.

PWM_DeviceDriver: Controls PWM hardware for the battery LED. Allows setting frequency and duty-cycle.

Design Patterns in `evcontrolsystem`

In the emergency vehicle control box, two main design patterns were applied: the **Singleton** and the **Facade**. These patterns were chosen to better organize access to critical resources and to simplify the system interface.

Singleton

The class `evcontrolsystem` implements the *Singleton* pattern, using the *Meyers Singleton* variation. This ensures that there is only one instance of `evcontrolsystem` during the program execution. This approach is coherent because the vehicle control system should have a single centralized control point, avoiding inconsistencies that could arise if multiple instances were manipulating the same hardware resources, such as the battery (`Battery`) or the DDS module (`DDSPublisher`).

The *Singleton* provides:

- Global control of the system through `getInstance()`.
- Single initialization of critical components (battery, DDS communication, and cloud interface).
- Easy and orderly shutdown, preventing concurrency issues or multiple resource destruction.

Facade

Additionally, `evcontrolsystem` acts as a *Facade* for the rest of the system. It centralizes access to the classes `Battery`, `DDSPublisher`, `CloudInterface` and, indirectly, to hardware wrappers such as `ADS1115`, `PWM_DeviceDriver`, and the threading and synchronization classes in `CppWrapper`.

The *Facade* pattern provides:

- A simplified interface for complex operations, such as system initialization, main loop execution, and shutdown.
- Hides the internal complexity of dependencies (e.g., battery monitoring logic, DDS message sending, and Wi-Fi network management).
- Makes future system modifications easier without directly impacting clients that use `evcontrolsystem`.

3.9.3 Task Overview

To guarantee that the system operates efficiently and meets real-time constraints, the project is organized into multiple concurrent tasks (threads). Each task is designed to handle a specific function within the overall system. In order to achieve smooth coordination among these tasks, several inter-process communication (IPC) and synchronization mechanisms are used (Figure 3.28 and Figure 3.29). Shared data is protected through mutexes, while message queues and signals are employed to exchange information between threads safely and efficiently. These mechanisms ensure that all components interact consistently without data corruption or timing conflicts.

Intersection Control Box

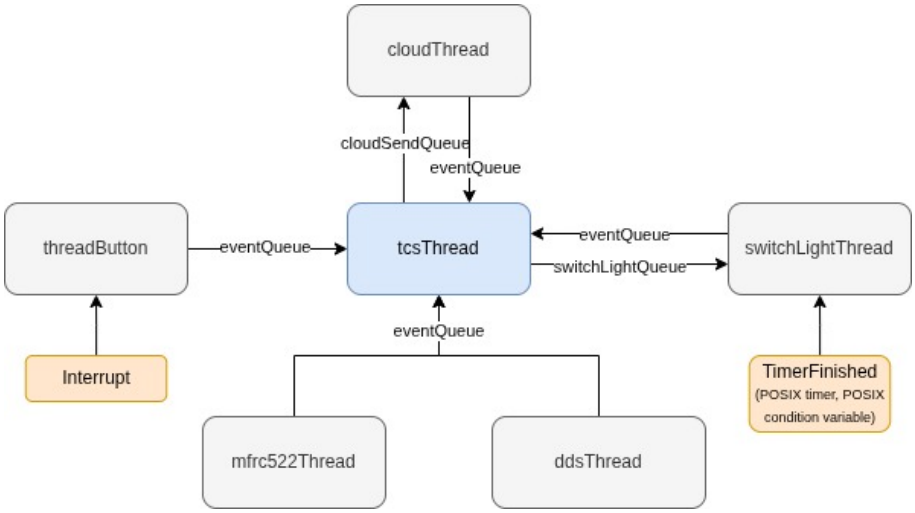


Figure 3.28: Intersection Control Box Task overview

In Figure 3.28, a high level overview of the Intersection Control Box tasks and IPC mechanisms can be observed. In it are described which tasks the Intersection Control Box will run, as well as how they will communicate between them. An overview of the individual tasks and their responsibilities is given below.

- **ddsThread**: Handles communication with the emergency vehicle system via DDS. This thread is responsible for receiving alerts and events from emergency vehicles and other DDS-enabled components. It remains blocked while waiting for incoming DDS messages and awakens upon reception, immediately forwarding relevant events to the Mediator logic to ensure timely system reactions.
- **tcsThread**: Acts as the central controller and Mediator of the system. It receives events from other threads through IPC mechanisms (primarily message queues), processes them according to internal configurations, and determines the appropriate system response. Based on DDS events, pedestrian requests, RFID detections, or traffic light timeouts, it generates new traffic light configurations and sends control commands to the `t_switchLight` thread.
- **cloudThread**: Manages all data exchange with the cloud infrastructure. This thread sends system data for logging and monitoring purposes and retrieves configuration updates when required. Communication is based on blocking methods, meaning the thread waits for cloud responses before continuing execution. Data to be sent to the cloud is received through a dedicated message queue.
- **switchLightThread**: Controls the physical traffic light behavior according to the commands received from the Mediator. It applies configuration changes, manages

semaphore state transitions, and handles timing signals. Upon light timeouts or state changes, it notifies the Mediator so that new configurations can be computed if necessary.

- **mfrc522Thread**: Implements the logic for RFID card detection using the MFRC522 module. This thread detects RFID tags used to identify disabled pedestrians at intersections and sends the corresponding events to the Mediator for further processing and prioritization.
- **threadButton**: Monitors the pedestrian crossing button. The thread is normally blocked and is awakened by an interrupt generated by a button press. It processes button click events, counts presses, and, when predefined conditions are satisfied, sends a request to the Mediator to grant early pedestrian passage.

Inter-thread communication is achieved through IPC mechanisms based on the C++ queue wrapper designed. The `eventQueue` carries heterogeneous event types generated by system components or by the Mediator itself. The `cloudSendQueue` is dedicated to data intended for cloud transmission, while the `switchLightQueue` transports commands related to traffic light state transitions.

The data each queue holds is specified in table 3.4.

Queue	**Data Type**
eventQueue	Event
switchLightQueue	SwitchLightsData
cloudSendQueue	CloudSendType

Table 3.4: Queues and their corresponding data types

Emergency Vehicle

The Emergency Vehicle system is composed of several concurrent tasks that ensure reliable communication, system monitoring, and visual feedback during operation.

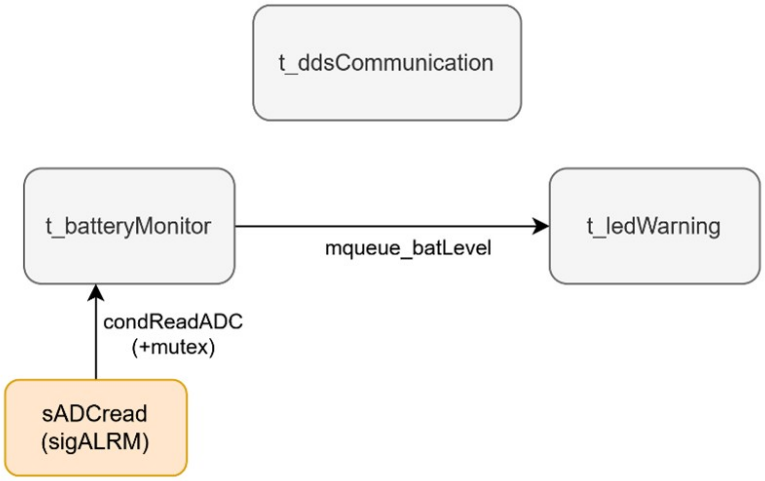


Figure 3.29: Emergency vehicle task overview

- t_ddsCommunication: This task acts as the publisher responsible for sending alert messages to the Control Box using the DDS communication protocol. When the emergency vehicle is approaching an intersection, this task transmits the signal that allows the traffic management system to prepare and ensure a clear passage for the vehicle.

- t_batteryMonitor: Monitors the battery voltage of the Raspberry Pi or the embedded system. It continuously checks the power level through ADC readings and ensures that the system does not experience unexpected shutdowns due to low voltage. This monitoring is essential for maintaining the reliability of the communication and control processes.
- t_ledWarning: Provides visual feedback to the user by controlling an LED that indicates the battery status. When the battery voltage drops below a critical threshold, this task activates the warning LED to alert the operator that the system needs charging or maintenance.

3.9.4 Task Priority

The threads are assigned different priority levels based on their importance and timing requirements. High-priority tasks handle time-critical operations — such as emergency signal detection — while lower-priority threads manage background processes. This priority-based scheduling allows the operating system to allocate CPU time effectively and maintain reliable real-time performance.

Intersection Control Box

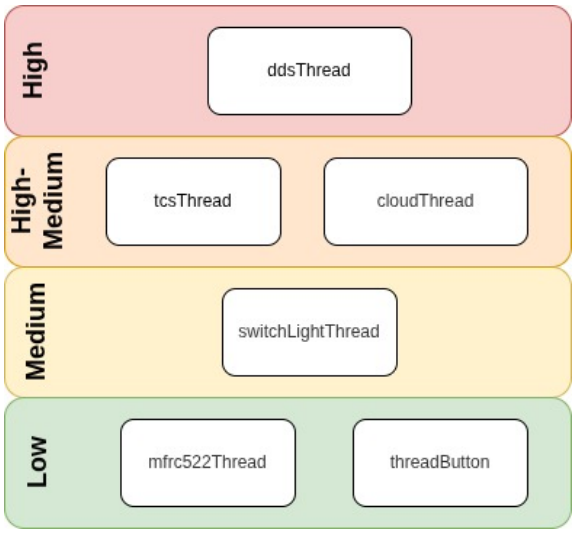


Figure 3.30: Intersection Control Box task priorities

In Figure 3.30, we can find a diagram with the priorities of each of the Intersection Control Box tasks. The task priorities were defined based on the criticality and time constraints of each process.

- **ddsThread**: Assigned the highest priority, as it must process emergency vehicle alerts in real time. Since emergency vehicles may approach the intersection at high speed, the system is required to react immediately in order to guarantee safe and efficient passage.
- **tcsThread** and **cloudThread**: Configured with high–medium priority. The Mediator thread must rapidly interpret and coordinate information received from all other threads, triggering time-critical actions such as traffic light (re)configuration. The cloud communication thread, while less time-critical than DDS communication, must ensure reliable synchronization with the cloud to maintain accurate system data and configuration updates.

- **switchLightThread**: Runs at medium priority, as it executes the commands issued by the Mediator and controls the physical traffic light transitions. Although it must respond promptly, its timing constraints are less strict than those of DDS communication or system management tasks.
- **mfrc522Thread** and **threadButton**: Assigned low priority, since their operations are primarily event-driven and less time-sensitive. These threads remain responsive to pedestrian or vehicle interactions but do not require immediate CPU attention unless awakened by an interrupt.

Emergency Vehicle

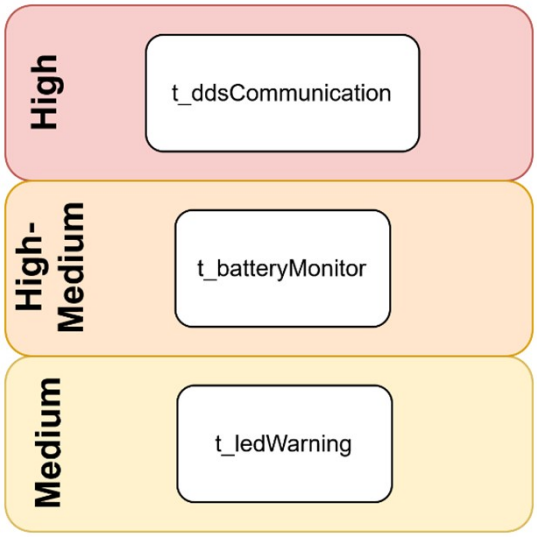


Figure 3.31: Emergency Vehicle task priorities

The task priorities were defined based on the criticality of each operation and the timing requirements for maintaining reliable emergency communication. t_ddsCommunication has the highest priority because it is responsible for sending emergency alerts in real time. Any delay in publishing these messages could compromise the coordination with the Control Box and delay the system's reaction to an approaching emergency vehicle. The t_batteryMonitor requires a relatively high priority to ensure the power status is constantly verified. A power failure could result in a complete system shutdown, preventing DDS messages from being sent — which is critical in an emergency context. t_ledWarning runs at a medium priority level. While it provides an important visual alert to the operator, its function is less time-sensitive than the DDS or battery monitoring tasks. Its main role is to inform the user promptly when the battery is low without interfering with the higher-priority operations.

3.9.5 Flowcharts

The flowcharts provide a structured and comprehensive visualization of the program's logical flow, offering a clear overview of how individual processes are executed.

Every system must have a pair of Start-Up and Shut-Down APIs, which are responsible for starting and stopping the system's relevant elements. The flowcharts in Figure 3.32 indicate the base sequence for the designed `start()` and `stop()` APIs.

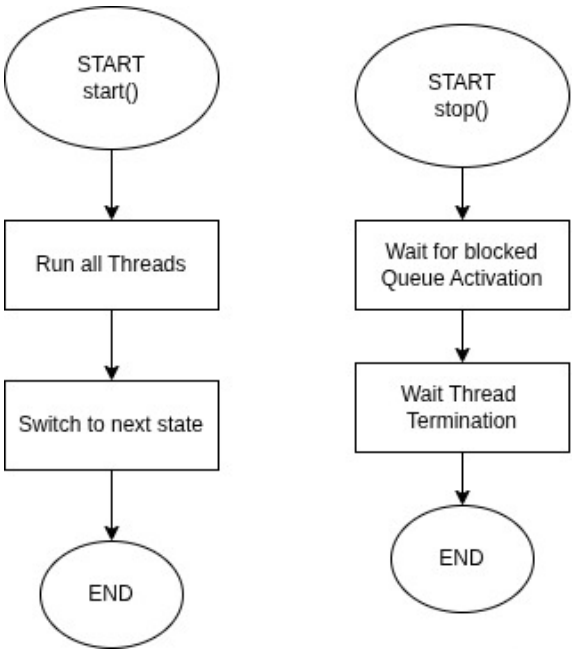


Figure 3.32: Start-up and shut-down APIs

Figure 3.33 indicates, more thoroughly, the start-up and shut-down processes. However, these only serve for visual understanding, since the mentioned sequence should be encapsulated in each class logic, considering their respective objects and methods.

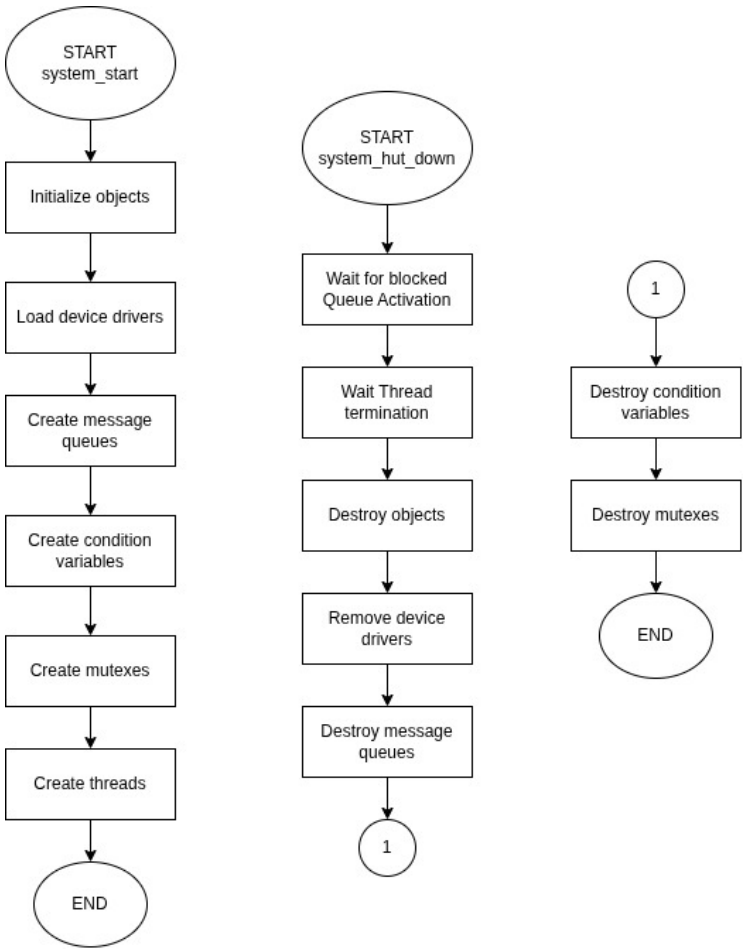
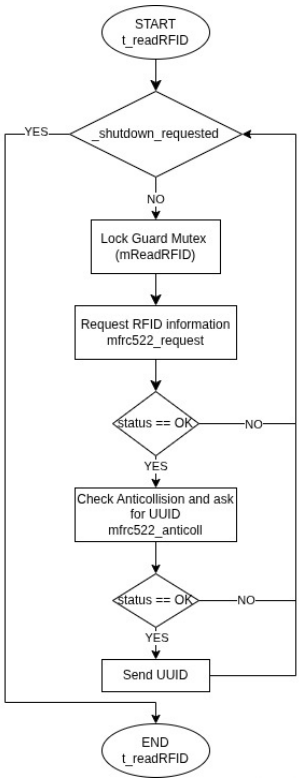


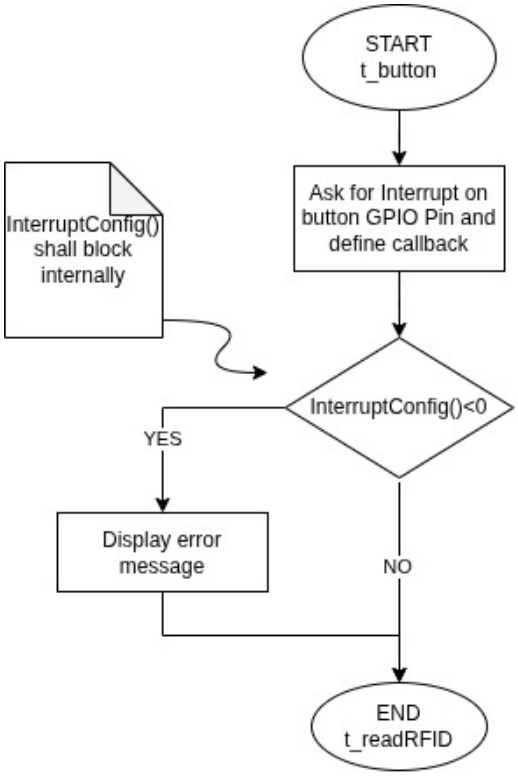
Figure 3.33: System start-up and shut-down sequence

Intersection Control Box

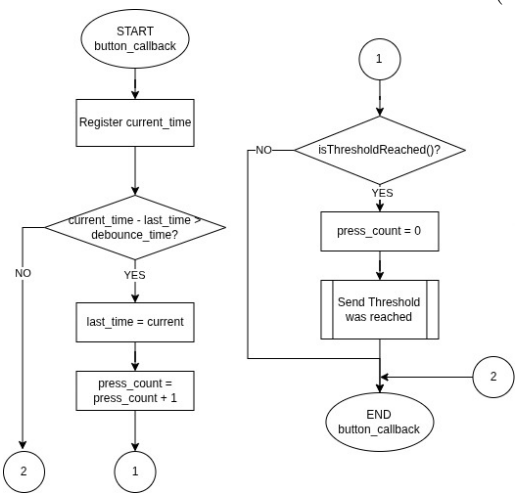
Additionally, each system has its own threads which execute in a *superloop*. Therefore, upon a system signal which may require system shutdown, the system must evaluate the situation and stop the threads and other active mechanisms to ensure correct destruction and shutdown. Thefore, dependency injection with a shutdown signal (`_shutdown_requested`) must exist between classes and be evaluated in each loop.



(a) `t_readRFID`

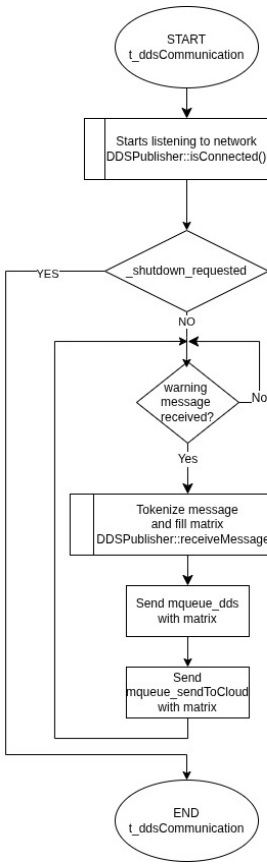


(b) `t_button`

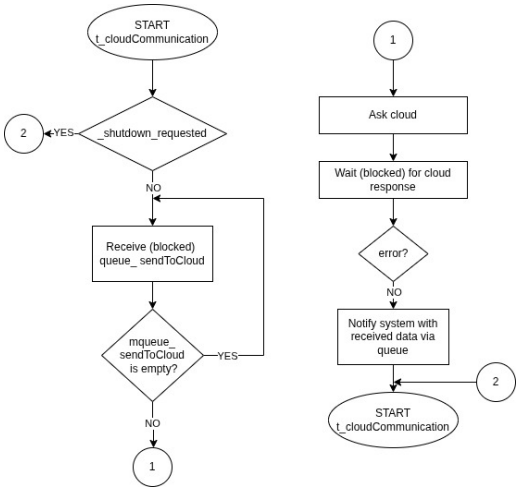


(c) `debounce`

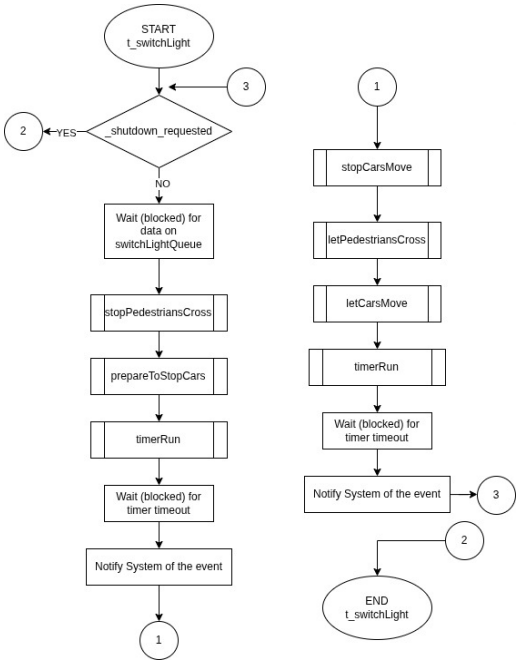
Figure 3.34: Flowcharts of Intersection Control Box system threads and support functions — 1



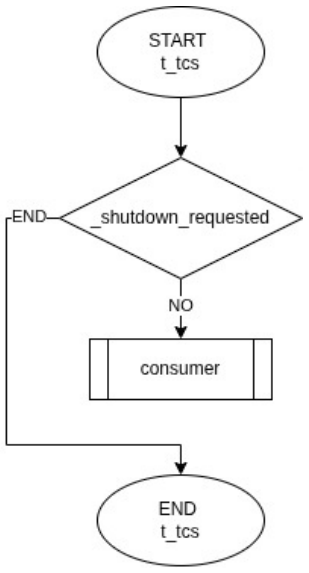
(a) `t_ddsCommunication`



(b) `t_cloudCommunication`



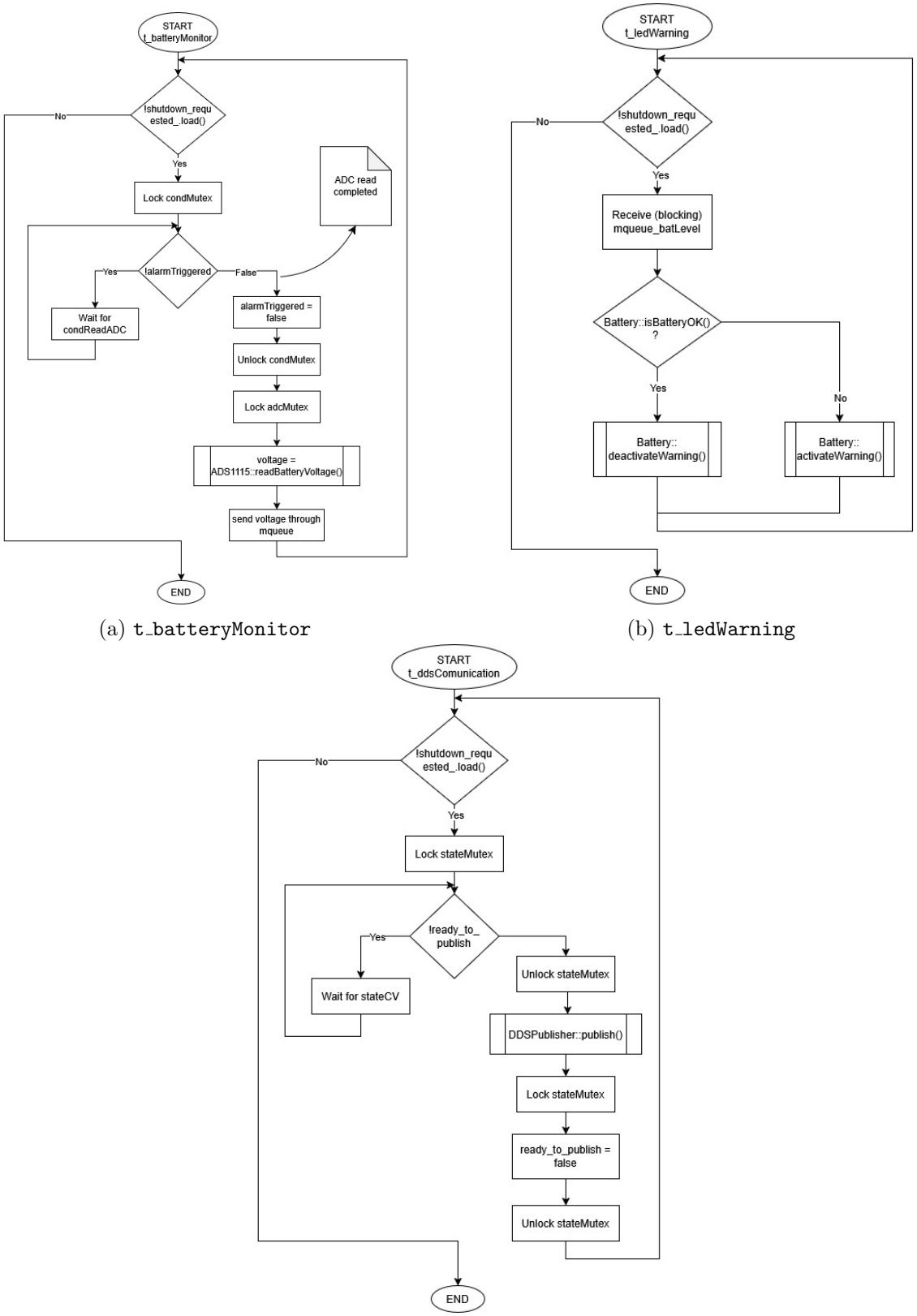
(c) `t_switchLights`



(d) `t_tcs`

Figure 3.35: Flowcharts of Intersection Control Box system threads — 2

Emergency Vehicle



(a) `t_batteryMonitor` (b) `t_ledWarning`

(c) `t_ddsCommunication`

Figure 3.36: Flowcharts of emergency vehicle system threads

3.10 PWM Device Driver

Device drivers (section 3.16.15) are software components that establish communication between the operating system and the hardware devices. In this project, a PWM device driver is developed for Raspberry Pi 4 Model B, which aims to control PWM in one or more pins.

In the Intersection Control Box, the device driver will control the pedestrian semaphore buzzer using PWM. In the emergency vehicle, when the battery voltage reaches a threshold which is considered critical (too close to the lower limit of the Raspberry Pi power voltage), an LED will blink.

The register configuration is specified in the remaining of this subsetion.

3.10.1 Memory Addressing

Physical Address Map

The driver interfaces with three primary hardware register blocks on the BCM2711 SoC:

Peripheral	Physical Address	Description
Base Address	`0xFE000000`	BCM2711 peripheral base
GPIO Registers	`0xFE200000`	GPIO function selection
PWM Registers	`0xFE20C000`	PWM control and data
Clock Manager	`0xFE1010A0`	PWM clock configuration

Table 3.5: Hardware Register Base Addresses

Address Calculation

$$\begin{aligned} \text{GPIO\_BASE\_ADDR} &= \text{BASE\_ADDR} + \text{GPIO\_OFFSET} \\ &= \text{0xFE000000} + \text{0x00200000} = \text{0xFE200000} \\ \text{PWM\_CONFIG\_ADDR} &= \text{GPIO\_BASE\_ADDR} + \text{PWM\_OFFSET} \\ &= \text{0xFE200000} + \text{0x0000C000} = \text{0xFE20C000} \\ \text{CLK\_CONFIG\_ADDR} &= \text{BASE\_ADDR} + \text{CLK\_OFFSET} \\ &= \text{0xFE000000} + \text{0x001010A0} = \text{0xFE1010A0} \end{aligned}$$

3.10.2 Register Structures

GPIO Function Select Register (GPFSEL)

Controls the function mode of GPIO pins (input, output, or alternate functions).

Register	Offset	Purpose
GPFSEL[0-5]	0x00-0x14	6 registers, 3 bits per pin

Table 3.6: GPIO Register Structure

Calculation for pin configuration:

- Register index: $\text{GPFSELindex} = \lfloor \text{pin\_number}/10 \rfloor$
- Bit position: $\text{startBit} = (\text{pin\_number} \mod 10) \times 3$
- Bit mask: $\text{mask} = 0b111 \ll \text{startBit}$

PWM Control Registers

Register	Offset	Description
CTL	0x00	Control register (enable, mode, polarity)
STA	0x04	Status register
DMAC	0x08	DMA configuration
RNG1	0x10	Channel 1 range (period)
DAT1	0x14	Channel 1 data (duty cycle)
FIF1	0x18	Channel 1 FIFO input
RNG2	0x20	Channel 2 range (period)
DAT2	0x24	Channel 2 data (duty cycle)

Table 3.7: PWM Register Map

PWM Control Register Bits

Bit	Channel 1	Bit	Channel 2
0	PWEN0 (Enable)	8	PWEN1 (Enable)
1	MODE0 (PWM/Serializer)	9	MODE1
2	RPTL0 (Repeat last)	10	RPTL1
3	SBIT0 (Silence bit)	11	SBIT1
4	POLA0 (Polarity)	12	POLA1
5	USEF0 (Use FIFO)	13	USEF1
6	CLRF0 (Clear FIFO)	14	CLRF1
7	MSEN0 (M/S Enable)	15	MSEN1

Table 3.8: PWM CTL Register Bit Definitions

Clock Manager Registers

Register	Address	Description
CM_PWMCTL	0xFE1010A0	Clock control (source, enable)
CM_PWMDIV	0xFE1010A4	Clock divider (integer and fractional)

Table 3.9: Clock Manager Registers

Clock Configuration:

- Password required: `0x5A000000` for all writes
- Primary source: PLLD (500 MHz)
- Divider: 4 (considering frequency 187.5 MHz)
- Fallback: Oscillator (19.2 MHz)

3.10.3 PWM Pin Configuration

GPIO Pin	Alt Function	PWM Chip	Channel	Enum ID
12	ALT0	CHIP0	CH_0	GPIO12
13	ALT0	CHIP0	CH_1	GPIO13
18	ALT5	CHIP0	CH_0	GPIO18
19	ALT5	CHIP0	CH_1	GPIO19
40	ALT0	CHIP1	CH_0	GPIO40
41	ALT0	CHIP1	CH_1	GPIO41
45	ALT0	CHIP0	CH_1	GPIO45

Table 3.10: Supported PWM GPIO Pins

3.10.4 Device Driver Architecture

File Operations

The driver implements the standard Linux character device file operations:

Operation	Function	Description
open	`pwm_device_open`	Opens device, stores register pointer
release	`pwm_device_close`	Closes device, clears private data
read	`pwm_device_read`	Currently not implemented
write	`pwm_device_write`	Text-based interface: "freq duty"
unlocked_ioctl	`pwm_device_ioctl`	Binary interface for configuration

Table 3.11: File Operations

IOCTL Commands

Command	Code	Direction	Description
PWM_SET_CONFIG	1	Write	Set frequency and duty cycle
PWM_DISABLE	3	None	Disable PWM output

Table 3.12: IOCTL Command Set

3.10.5 Register Configuration Procedures

CM_PWMCTL Register - Clock Control

CM_PWMCTL (0xFE1010A0)						
31-24	23-8	7	6-5	4	3-0	
PASSWD	Reserved	BUSY	Reserved	ENAB	SRC	
0x5A	-	R	-	R/W	R/W	

Field Descriptions:

- **PASSWD [31:24]:** Must be 0x5A for any write operation
- **BUSY [7]:** Read-only. 1 = Clock generator running, 0 = Stopped
- **ENAB [4]:** 1 = Enable clock generator, 0 = Disable

- **SRC [3:0]:** Clock source selection
  - 0 = GND (off)
  - 1 = Oscillator (19.2 MHz)
  - 6 = PLLD (500 MHz) - *Primary choice*

Configuration Sequence:

1. Write 0x5A000000 to disable (SRC=0, ENAB=0)
2. Wait for BUSY=0
3. Write 0x5A000006 to set source (SRC=6, ENAB=0)
4. Write 0x5A000016 to enable (SRC=6, ENAB=1)
5. Verify BUSY=1

CM_PWMDIV Register - Clock Divider

CM_PWMDIV (0xFE1010A4)		
31-24	23-12	11-0
PASSWD	DIVI	DIVF
0x5A	R/W	R/W

Field Descriptions:

- **PASSWD [31:24]:** Must be 0x5A for any write
- **DIVI [23:12]:** Integer part of divisor (1-4095)
- **DIVF [11:0]:** Fractional part of divisor

Divider Calculation:

$$f_{\text{output}} = \frac{f_{\text{source}}}{\text{DIVI} + \frac{\text{DIVF}}{4096}}$$

Example Configuration:

- Source: PLLD = 500 MHz
- Desired output: 125 MHz
- DIVI = 4, DIVF = 0
- Register value: 0x5A004000
- Result: $\frac{500}{4} = 125$ MHz (experimentally: 187.5 MHz)

GPFSEL Registers - GPIO Function Select

GPFSELn (n=0 to 5)				
31-30	...	[3p+2:3p]	...	2-0
Reserved	...	FSELp	...	FSEL0
-	...	R/W	...	R/W

Each pin uses 3 bits:

Value	Function
000	Input
001	Output
010	ALT5
011	ALT4
100	ALT0
101	ALT1
110	ALT2
111	ALT3

Pin Mapping:

- GPFSEL0: GPIO 0-9
- GPFSEL1: GPIO 10-19
- GPFSEL2: GPIO 20-29
- etc.

Example: Configure GPIO12 to ALT0 (PWM)

- Register: GPFSEL1 (GPIO 10-19)
- Pin position: GPIO12 = position 2 in register
- Bit range: [8:6] (position 2 × 3)
- Value: 100 (ALT0)
- Operation: Clear bits [8:6], set to 0b100

PWM CTL Register - Control

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MSEN1	CLRF1	USEF1	POLA1	SBIT1	RPTL1	MODE1	PWEN1	MSEN0	CLRF0	USEF0	POLA0	SBIT0	RPTL0	MODE0	PWEN0

Key Bit Definitions:

- **PWENx:** Channel enable (0=disabled, 1=enabled)
- **MODEx:** Mode selection (0=PWM, 1=Serializer)
- **MSENx:** Mark/Space enable (0=PWM algorithm, 1=M/S mode)
- **POLAx:** Output polarity (0=normal, 1=inverted)
- **USEFx:** Use FIFO (0=data register, 1=FIFO)

Standard Configuration for M/S PWM:

Configuration	Channel 0	Channel 1
Enable (PWENx)	Bit 0 = 1	Bit 8 = 1
M/S Mode (MSENx)	Bit 7 = 1	Bit 15 = 1
PWM Mode (MODEx)	Bit 1 = 0	Bit 9 = 0
All other bits	0	0
Result Value	`0x00000081`	`0x00008100`

PWM RNG Registers - Range (Period)

RNG1 / RNG2 (32-bit)
Range Value (0x00000000 - 0xFFFFFFFF)

Purpose: Defines the PWM period in clock cycles.

Calculation:

$$\text{RNG} = \frac{f_{\text{clock}}}{f_{\text{desired}}}$$

Examples:

Frequency	Calculation	RNG Value
1 kHz	$\frac{187,500,000}{1,000}$	187,500
10 kHz	$\frac{187,500,000}{10,000}$	18,750
100 kHz	$\frac{187,500,000}{100,000}$	1,875
1 MHz	$\frac{187,500,000}{1,000,000}$	187

PWM DAT Registers - Data (Duty Cycle)

DAT1 / DAT2 (32-bit)
Data Value (0x00000000 - RNG value)

Purpose: Defines the number of clock cycles the output is HIGH.

Calculation:

$$\text{DAT} = \frac{\text{RNG} \times \text{duty\_cycle}}{100}$$

Examples for 1 kHz (RNG=187,500):

Duty Cycle	Calculation	DAT Value
0%	$\frac{187,500\times0}{100}$	0
25%	$\frac{187,500\times25}{100}$	46,875
50%	$\frac{187,500\times50}{100}$	93,750
75%	$\frac{187,500\times75}{100}$	140,625
100%	$\frac{187,500\times100}{100}$	187,500

Complete PWM Configuration Summary

To configure 2 kHz PWM at 60% duty cycle on GPIO12 (Channel 0):

Step	Register	Value
1. Set clock source	CM_PWMCTL	0x5A000006
2. Set clock divider	CM_PWMDIV	0x5A004000
3. Enable clock	CM_PWMCTL	0x5A000016
4. Set GPIO12 to ALT0	GPFSEL1[8:6]	100 (0x4)
5. Disable channel	CTL[0]	0
6. Set range	RNG1	93,750
7. Set duty	DAT1	56,250
8. Enable M/S mode	CTL[7,0]	0x81

Table 3.13: Complete Configuration Sequence

Verification Calculations:

$$\text{Range} = \frac{187,500,000}{2,000} = 93,750 \text{ clock cycles}$$
$$\text{Data} = \frac{93,750 \times 60}{100} = 56,250 \text{ cycles HIGH}$$
$$\text{Actual Frequency} = \frac{187,500,000}{93,750} = 2,000 \text{ Hz}$$
$$\text{Actual Duty Cycle} = \frac{56,250}{93,750} \times 100 = 60\%$$

3.10.6 Core Functions

GPIO Function Configuration

Function: `void SetGPIOFunction(GPIORegister *gpio, PWM_GPIO pin, FSELx func, CLKRegister *clk)`

Purpose: Configure GPIO pin to PWM alternate function.

Steps:

1. Calculate register index and bit position
2. Create bit mask: `mask = 0b111 << startBit`
3. Read current register value
4. Clear function bits: `value &  mask`
5. Set new function: `value | (func << startBit)`
6. Write back to register

PWM Configuration

Function: `void SetPWM(PWMRegister *pwm, PWM_GPIO gpio, int freq, int duty, CLKRegister *clk)`

Purpose: Configure PWM frequency and duty cycle.

Calculations:

$$\text{range} = \frac{\text{CLK\_PWM}}{\text{frequency}}$$
$$\text{data} = \frac{\text{range} \times \text{duty\_cycle}}{100}$$

Where:

- CLK_PWM = 187,500,000 Hz (should first be experimentally determined)
- range = total clock ticks per PWM period
- data = number of ticks the output is high

Configuration sequence:

1. Disable channel (clear PWENx bit)
2. Calculate range and data values
3. Write to RNGx and DATx registers
4. Configure CTL register (enable M/S mode)
5. Enable channel (set PWENx and MSENx bits)

3.10.7 Key Design Decisions

Mark/Space Mode

The driver uses Mark/Space (M/S) mode rather than standard PWM mode:

- Provides more predictable duty cycle control
- Enables by setting MSENx bit in CTL register
- Ensures output high time = (DATx / RNGx) × period

Channel Disable During Update

Before modifying PWM parameters:

- Channel is disabled to prevent glitches
- Registers are updated while disabled
- Channel is re-enabled with new configuration

3.10.8 Init and Close APIs

pwm_init() Function - Part 1: Device Registration

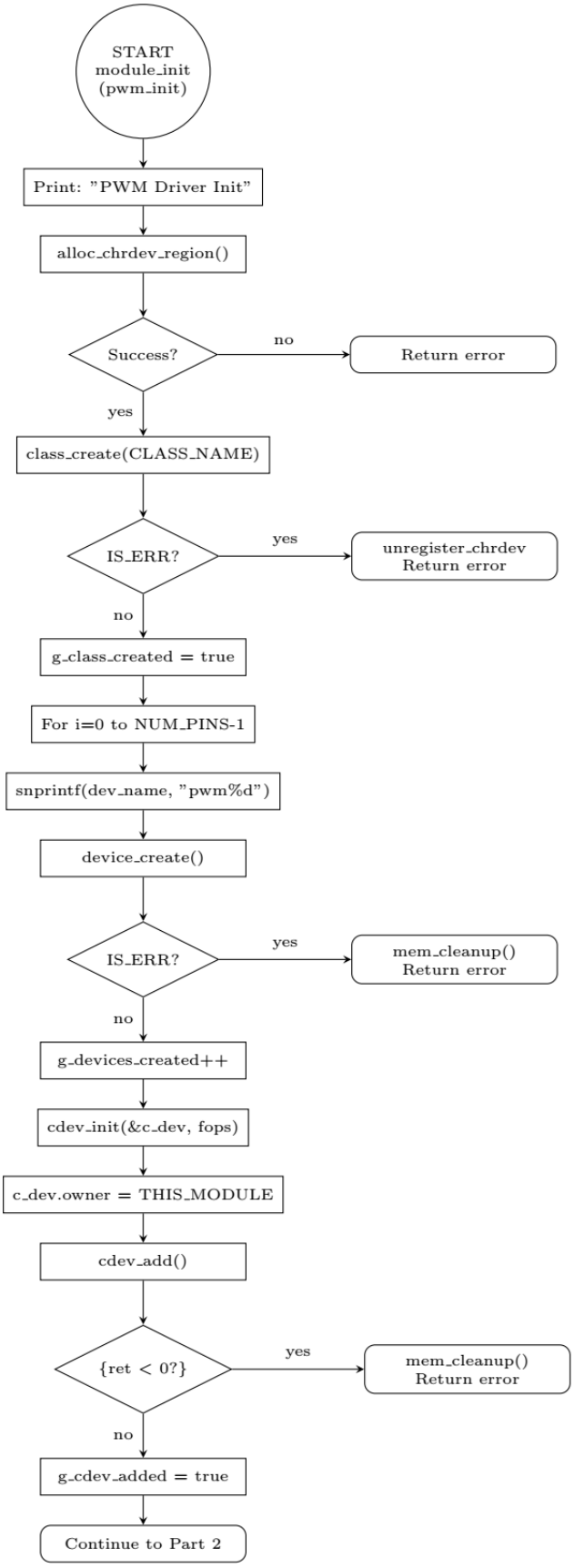


Figure 3.37: pwm_init() - Device Registration Phase

pwm_init() Function - Part 2: Hardware Initialization

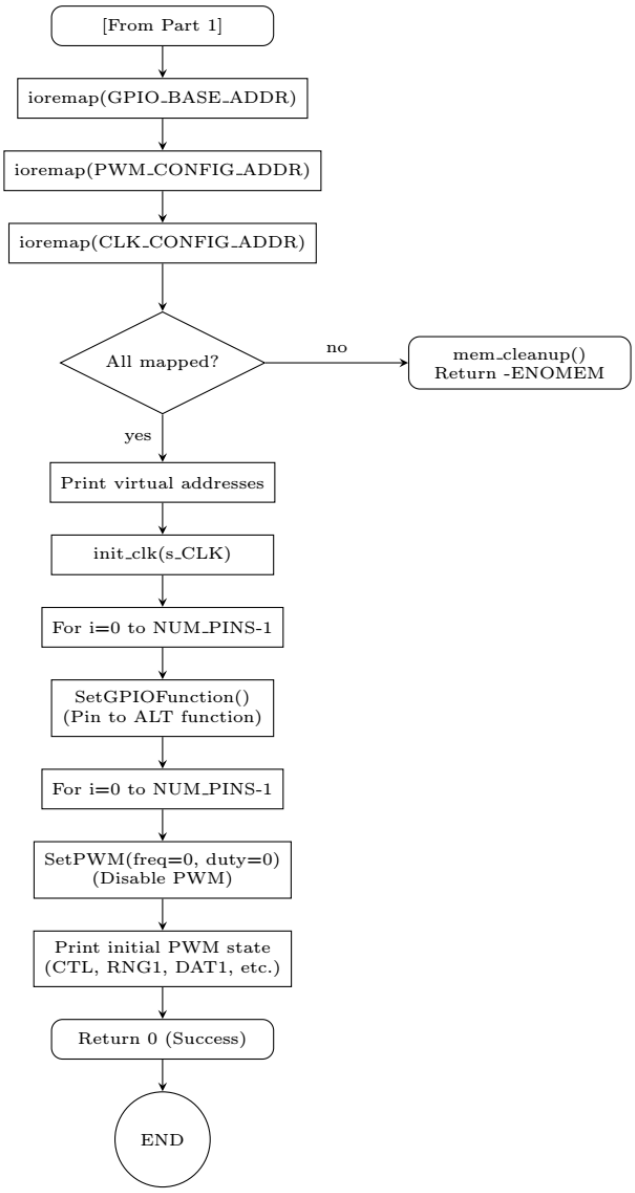


Figure 3.38: pwm_init() - Hardware Setup Phase

cleanup() Function

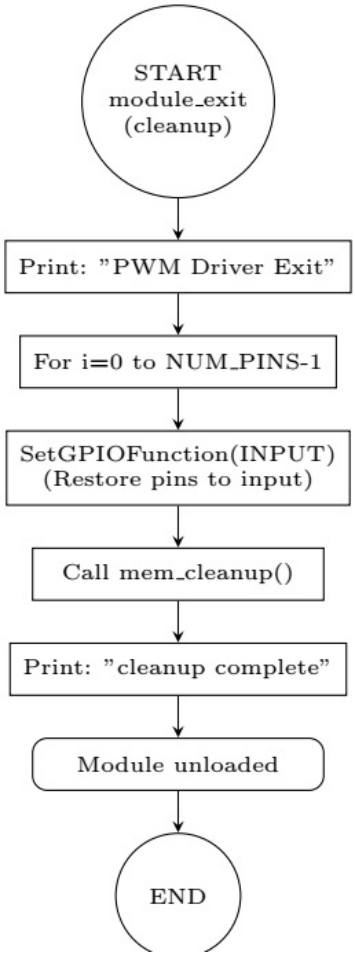


Figure 3.39: cleanup() Module Exit Function

mem_cleanup() Helper Function

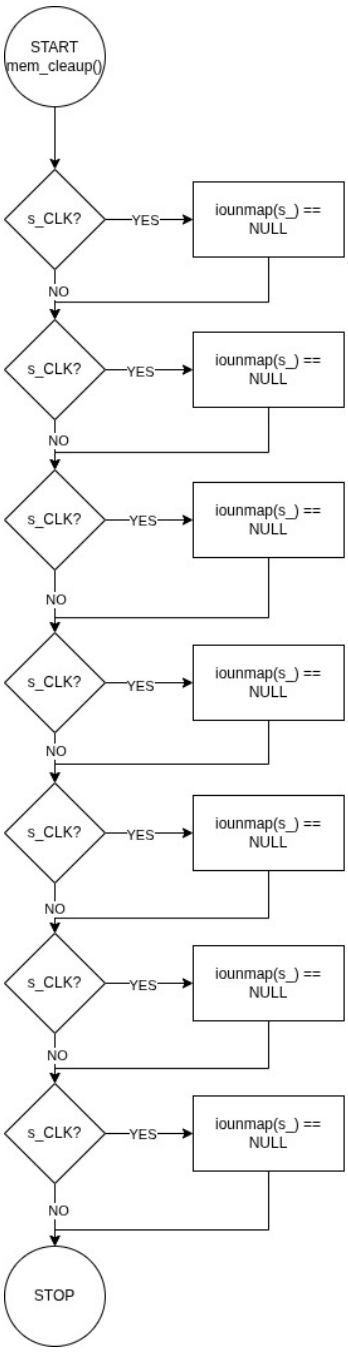
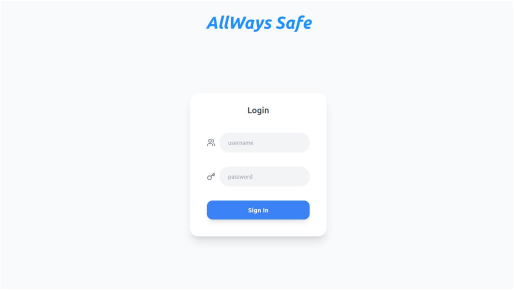


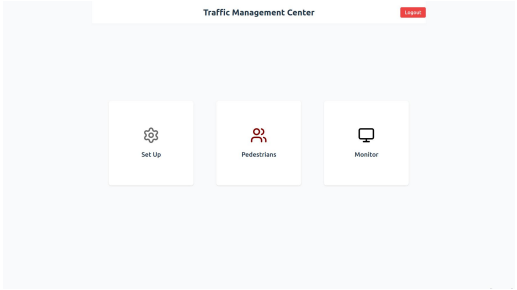
Figure 3.40: mem cleanup() Resource Deallocation

3.11 GUI interface

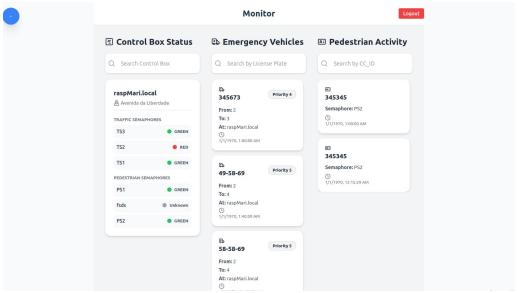
Since there will be a monitoring interface through a remote station, and user interface must be designed in order to display the options available to the monitor in charge. In Figure 3.41a, we can that the system's login interface, which then goes to the main menu in Figure 3.41b. In the main menu, it is possible to set up the system, which allows for the creation of Control Boxes, Traffic Semaphores and Pedestrian Semaphores, as Figure 3.41e, 3.41f, 3.41g illustrate. Then, in the Pedestrians menu in Figure 3.41d, we can also add pedestrians to the database and. Lastly, in the Monitor menu in Figure 3.41c it is possible to keep track of general occurrences.



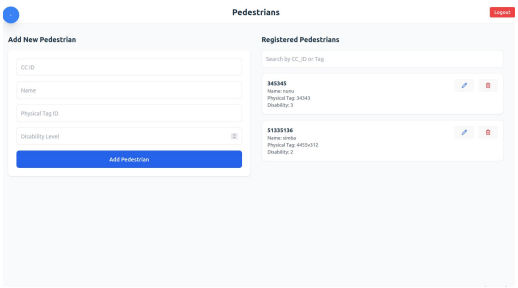
(a) LoginPage



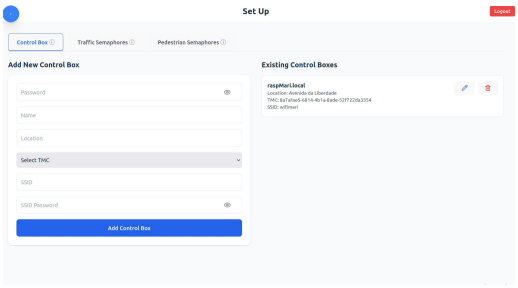
(b) Traffic Management Center



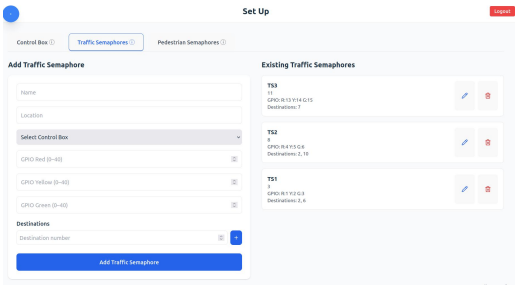
(c) Monitor



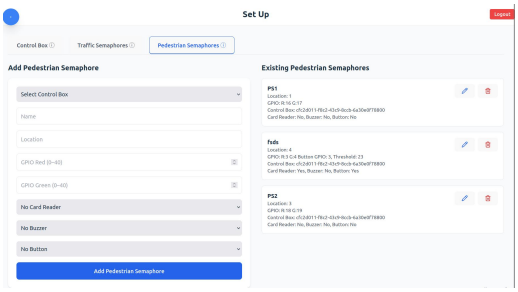
(d) Pedestrians Register



(e) Control Box Set Up



(f) Traffic Semaphore Set Up



(g) Pedestrian Semaphore Set Up

Figure 3.41: Graphic User Interface

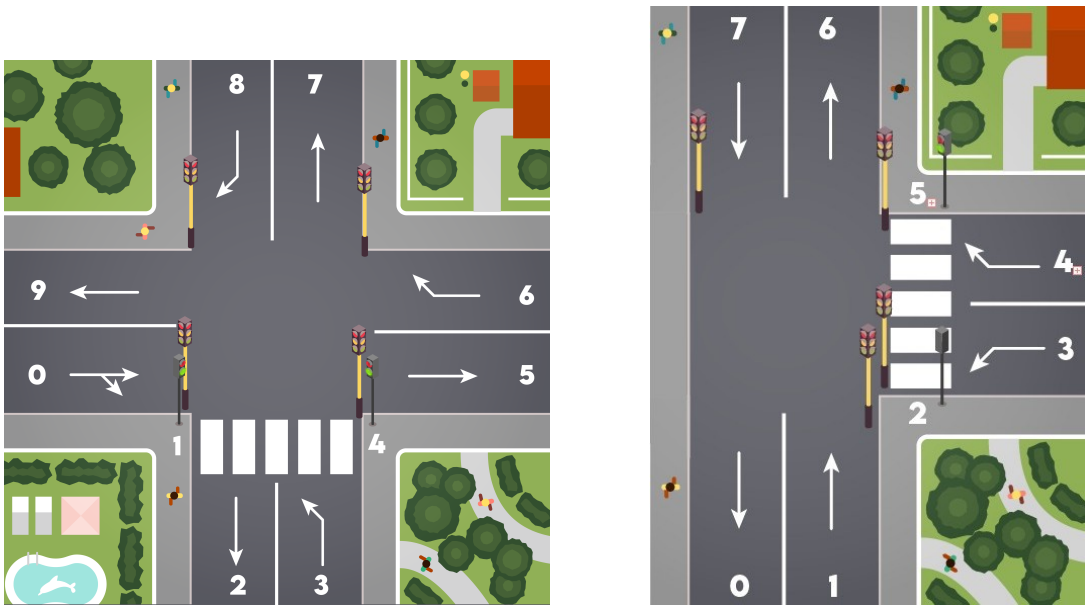
3.12 Traffic Control Algorithm

Taking into consideration that we desire a platform which is able of, automatically, on *boot*, loading the configurations from the Cloud and configure the system, it must be able to adapt to any semaphore configuration. Therefore, a Traffic Control Algorithm must be established.

This algorithm aims to finds the system configurations (i.e., which semaphores will be on, and which semaphores will be off simultaneously, allowing that each of them is, at least, once turned on) from a Cloud configuration performed for that specific Intersection Control Box, by the Traffic Manager on the GUI. The code must ensure configurations feasibility: thus, it must verify all configurations received.

Given the statement, the most suitable approach for the Traffic Control Algorithm can be explained through **Graph Theory**, using the **Maximal Independent Set Problem**.

3.12.1 Labelling, Trajectory Conflict and Used Terminology



(a) Intersection Scenario 1 (b) Intersection Scenario 2

Figure 3.42: Possible Intersection scenarios

First of all, there must be a standardized labelling of all routes in each intersection, which is performed taking into consideration the existance of **locations** and **directions**. **Locations** are integer numbers which identify a position of any existing route in an intersection. Therefore, they can represent any relevant element in an intersection (in this case, they can represent either a Traffic Semaphore or a Pedestrian Semaphore — i.e., where they are located or the origin from where the vehicles or pedestrians who pass there come from). Moreover, **destinations** can be defined for each Traffic Semaphore: these are the locations, in the intersection, to where vehicles can go from each traffic semaphore (thus, they are also integers).

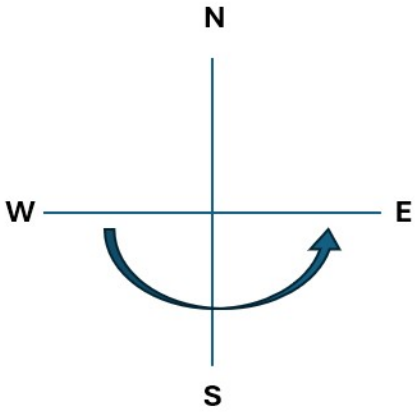


Figure 3.43: Location labelling criteria

Taking this into account, the starting point and direction criteria to start the labelling still needs to be addressed. For this, the criteria of **starting from West to East in**

counter-clockwise direction was chosen (Figure 3.43).

So, Traffic Semaphores are placed under a specific location and can have one or many destinations. Pedestrian Semaphores are placed under a location and are directly related to the Pedestrian Semaphore placed in the next location in which a Pedestrian Semaphore exists. Table 3.14 ilustrates this reasoning, taking into consideration the Intersection Scenario 1, depicted in Figure 3.42a.

Semaphore	**Location (L)**	**Destination (D)**
TSEM0	0	5, 2
TSEM3	3	9
TSEM6	6	7
TSEM8	8	9
PSEM1	1	–
PSEM4	4	–

Table 3.14: Traffic Semaphore (TSEM) and Pedestrian Semaphore (PSEM) data

Considering criteria mentioned above, any intersection configuration can be established.Figure 3.42a and Figure 3.42b illustrate the labelling algorithm for different intersection scenarios.

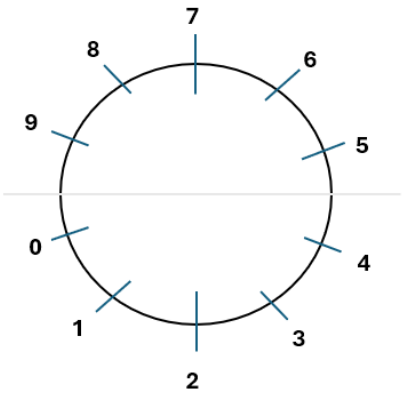


Figure 3.44: Circular Approach to Scenario depicted in Figure 3.42a

The only remaining criteria to establish is the algorithm that ensures if there is **trajectory conflict** or not. A **trajectory** is a possible path that can be done from a location to another location (in the Pedestrian Semaphores' case) and from a location to any of the possible directions (in the Traffic Semaphores' case). There is **Trajectory Conflict** when any trajectory from one semaphore crosses another trajectory from another semaphore.

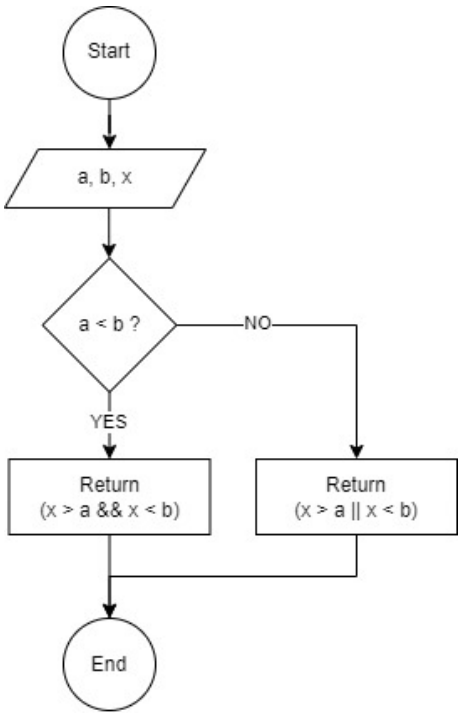
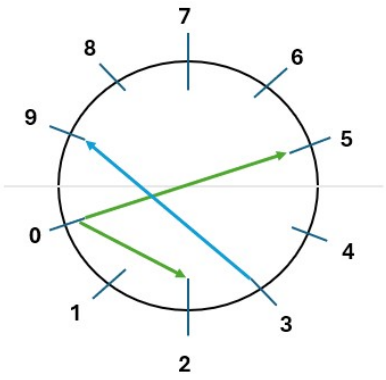
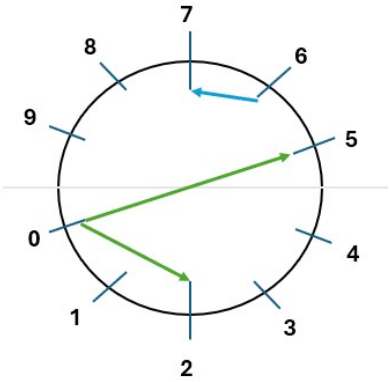


Figure 3.45: Flowchart for Trajectory Conflict Verification — argument `a` and `b` represent respectively location and direction of the same semaphore, argument `x` represents either the location or direction of other semaphore

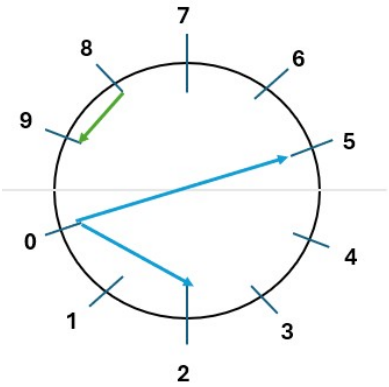
For this, a circular approach is considered (Figures 3.44 and 3.45): all locations in the intersection form a circular path, and, for each semaphore and its destination point (Pedestrian Semaphore: location — Next Pedestrian Semaphore location; Traffic Semaphore: location — destination) a line is drawn.



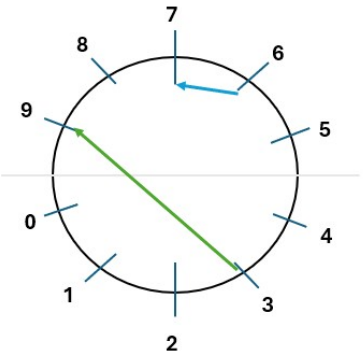
(a) Validation between Semaphore at Location 0 and Semaphore at Location 3: trajectories conflict



(b) Validation between Semaphore at Location 0 and Semaphore at Location 6: trajectories do not conflict



(c) Validation between Semaphore at Location 0 and Semaphore at Location 8: trajectories do not conflict



(d) Validation between Semaphore at Location 3 and Semaphore at Location 6: trajectories do not conflict

Figure 3.46: Trajectory Conflict Algorithm for Figure 3.42a— Visual Dry Run for a small set of cases

Visually, the validation of the algorithm follows the reasoning on 3.46.
Non-conflicting trajectories mean that those semaphores can be ON simultaneously.

3.12.2 Graph Theory and Undirected Graphs

Graph theory is the study of graphs, which are structures composed of vertices connected by edges. These structures are widely used to model a variety of mathematical and real-world problems.

In this case, we can model semaphores' locations as the vertices of our graph, and the edges between vertices as a conflict between the semaphores on those locations. This problem can be represented as an **undirected graph**, since the influence between vertices is symmetrical. Furthermore, those vertices and edges can be modelled using an adjacency matrix (also known as a conflict graph), where each vertex corresponds to both a row and a column. An entry shall be marked with 1 if there is a conflict between the corresponding vertices, and 0 otherwise.

3.12.3 Maximal Independent Set

Considering that, as mentioned, the system must the able of analysing which semaphores can be ON simultaneously for any configuration (that follows the convention labelling established), there are relevant characteristics with which the desired algorithm must comply: it must generate the mininum number of configurations with the maximal number of semaphores possible. This is, each configuration should have the most semaphores, since

their trajectories don't collide. This problem is formulated under graph theory as the **Maximal Independent Set**.

The Maximal Independent Set (Figure 3.47) problem is described in graph theory as a set of vertices where no two are connected, and you can't add any other vertex from the graph to the set without breaking this rule — this formulates exactly the system's requirements.

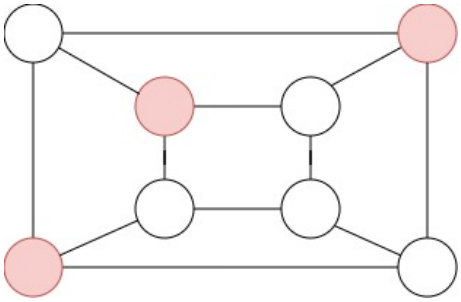


Figure 3.47: Example of Maximal Independent Set Graph — Circles represente Vertices, Lines represent Edges — Red Circles are conflicting vertices.

Transposing this algorithm to the project, the conflicting vertices (red) are the semaphores whose trajectories conflict, whereas the others (white) don't have conflicting trajectories. Considering the intersection configuration on Figure 3.42a, the following graph can be obtained.

This reasoning is iterated through all semaphores, and if an intersection is detected, then that semaphore is ignored for the current configuration and that branch is ignored. If there is no conflict, the algorithm should proceed to evaluate if there a viable configuration for that set of chosen semaphores (recursive iteration). This approach is called **Backtracking with prunning**, whose best case complexity is $O(N)$ and worst case is $O(2^N)$. Therefore, is presents a better approach than brute-force backtracking.

3.12.4 Algorithm

The algorithm that implements the Backtracking with prunning is in picture 3.48.

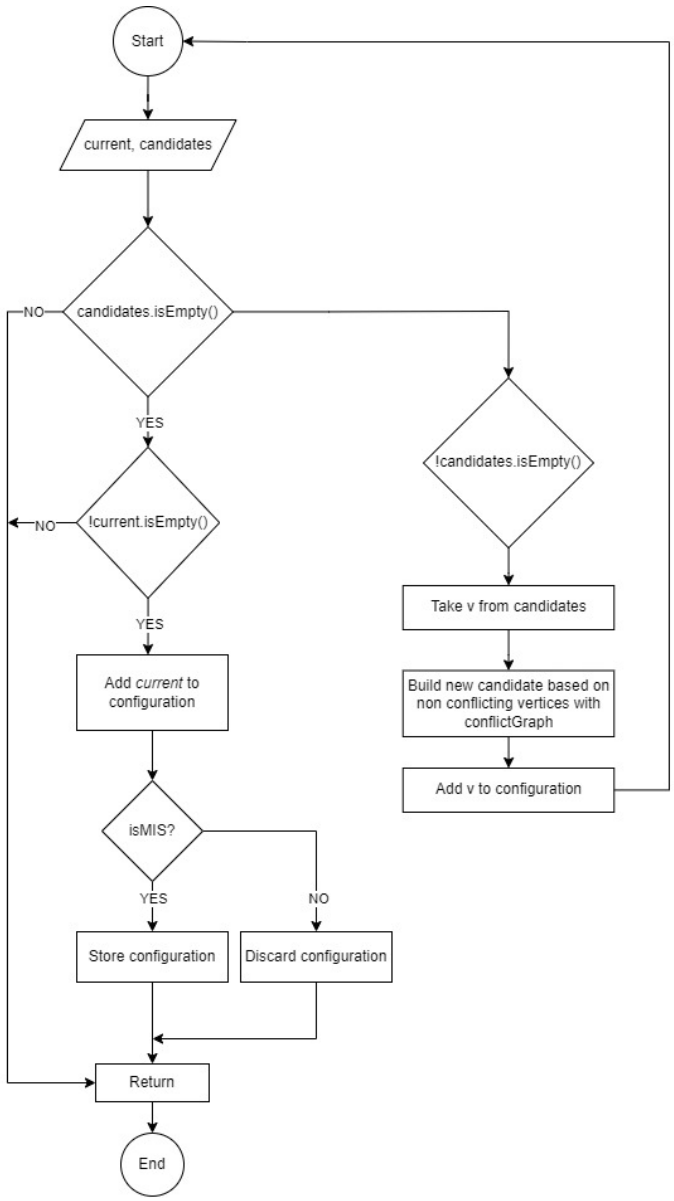


Figure 3.48: Backtracking with prunning Algorithm

3.13 Test Cases

The AlwaysSafe testing strategy has been carefully designed to verify that all hardware and software components meet the required standards of performance, reliability, and safety expected in a veterinary clinical environment. This section presents the set of test cases developed to validate the system's functionality at different levels.

The testing process includes unit tests to evaluate individual hardware components, integration tests to ensure proper communication between hardware and software modules, and system tests to confirm overall performance, responsiveness, and fault tolerance. Table 3.15 to Table 3.17 detail the Intersection Control Box, Emergency Vehicle and Cloud test cases.

Finally, given that AlwaysSafe operates as a real-time system, the test cases listed in Table 3.18 acesses its real-time responsiveness and stability.

Intersection Control Box

Test Case	Test Description	Expected Result	Real Result
Login in cloud	Verify authentication with valid credentials	System returns success, user authenticated and connected to Cloud	
Logout off cloud	Disconnect control box from cloud system	Session terminated successfully, Cloud disconnected	
Button Responsiveness	Test if buttons respond to user input	Press event detected (getPressCount incremented) and sent to system	
Semaphore State Change	Change semaphore state from one colour to another	Traffic/Pedestrian Light changes to another colour, duration respected, state updated	
RFID Card Reading	Bring valid RFID card close to reader	RFID_Reader detects card, reads CardID and returns data correctly	
Buzzer Activation	Pedestrian semaphore turns GREEN	BuzzerDriver activates buzzer, audible sound emitted (with a specified frequency rate, ioctl commands executed successfully)	
Traffic/Pedestrian Light Duration	Configure specific duration for green light (e.g., 30s)	Semaphore maintains GREEN for exactly 30 seconds before changing	
Invalid Login Attempt	Attempt login with invalid credentials	System rejects authentication, returns error, access denied	
DDS Responsiveness	Receive message from DDS communication	Receive message with emergency vehicle ID and direction	
Emergency Mode Activation	Activate System Mode EMERGENCY_ACTIVE	Semaphores change to emergency state, normal operation suspended	
DDS Subscriber creation	Access to DDS topic	Receive message from correct topic	
Extended Passage	Person with disability passed card on RFID	Extend GREEN pedestrian light and extend RED traffic light	
Early Passage	Count of press buttons greater than threshold in a few seconds	Pedestrian light turns GREEN before its time	
Pin Association through cloud	Cloud attributes a pin for semaphore light	Pin attributed is available for the effect and is configured with the right settings	
Invalid RFID cardID	Person without disability scans card on RFID module	GREEN light is not extended	

Table 3.15: Intersection Control Box test cases

Emergency Vehicle

Test Case	Test Description	Expected Result	Real Result
Login in cloud	Verify authentication with valid credentials	System returns success, user authenticated and connected to Cloud	
Logout off cloud	Disconnect user from cloud system	Session terminated successfully, Cloud disconnected	
ADS1115 Warning Activation	Battery voltage drops below threshold	Blinking LED turns on	
ADS1115 Warning Deactivation	Battery voltage returns to normal levels	Blinking LED turns off	
ADS1115 Voltage Reading	Read battery voltage from ADS1115 sensor	Correct voltage reading	
Network Connection to Control Box AP check	Verify network connectivity status	Emergency Vehicle has IP address	
DDS Message Publishing	Publish MessageData through DDSPublisher	Message published to topic successfully, timestamp and senderID correct	
LED Blink Control	Control LED blinking through LedBlinkDriver	LED blinks at specified rate, ioctl commands executed successfully	

Table 3.16: Emergency Vehicle test cases

Cloud

Test Case	Test Description	Expected Result	Real Result
Connection Test	Check connection to cloud	Can connect to cloud	
Data Insertion	Check write to cloud	Can insert items in cloud	
Data Retrieval	Check read from cloud	Can read items from cloud	
Data Alteration	Check altered items in cloud	Can alter items in cloud	
Data Consistency	Verify that updates made locally are reflected in the cloud database and vice versa	Data remains consistent between local and cloud databases	
Unauthorized access attempt detected	Try access cloud with an invalid login	Access not granted	

Table 3.17: Cloud test cases

Real-time performance and responsiveness

Test Case	Test Description	Expected Result	Real Result
Sensor Data Read Latency	Measure the time from the start of the sensor data request to the data being available for processing	Data is available within 500 ms of request	
Periodic Sensor Read Consistency	Check if periodic sensor reads occur precisely at the defined interval (e.g., every 2 minutes)	Sensor data is consistently read every 2 min	
Thread Synchronization Integrity	Test for race conditions or deadlocks when multiple threads access shared resources simultaneously	No deadlocks or race conditions occur	

Table 3.18: Real-time performance and responsiveness test cases

3.14 Error Handling

Every system is susceptible to failure. With this in mind, it is essential to implement robust mechanisms capable of detecting, managing, and mitigating errors effectively. Given the multiple components that make up the AlwaysSafe system, various error-handling strategies must be carefully designed and integrated to ensure consistent and safe operation. As an embedded system, AlwaysSafe requires a high degree of reliability and safety. While

it is engineered for stable and continuous performance, anticipating potential faults and implementing appropriate recovery measures are crucial to maintaining system integrity and availability. Table 3.19to Table 3.21 present an overview of the most relevant error types and scenarios within each system, along with the corresponding responses designed to minimize disruption and prevent critical failures.

Intersection Control Box

Error	**System Response**	**Critical**
Invalid login credentials	Reject authentication, return error message, deny access to Cloud	No
Cloud connection lost	Attempt reconnection until success	Yes
RFID reader hardware failure	Disable RFID functionality, alert message	Yes
Button GPIO not responding	Attempt GPIO reinitialization	No
Traffic light stuck in one state	Error message and blink yellow light	Yes
Invalid semaphore duration value	Use default duration value, notify operator	Yes
Emergency mode activation failed	Activate all warning signals	Yes
DDS subscriber connection failed	Retry connection	Yes
Buzzer driver malfunction	Continue operation without audio alerts, notify maintenance	No
DDS subscriber initialization failed	Retry initialization	Yes

Table 3.19: Intersection Control Box error handling

Emergency Vehicle

Error	**System Response**	**Critical**
ADS1115 sensor communication failure	Use last known voltage value	Yes
Network connection lost	Attempt reconnection periodically	Yes
Cloud authentication failed	Retry authentication	Yes
DDS publisher initialization failed	Retry initialization	Yes
LED blink driver not responding	Notify maintenance	Yes
Invalid voltage reading from ADC	Discard reading, use previous valid value	Yes
Message publishing failed	Queue message for retry, check DDS connection status	Yes
Battery warning cannot be deactivated	Force warning reset	No

Table 3.20: Emergency Vehicle error handling

Cloud

Error	System Response	Critical
Communication Lost	Try to restore connection, if not possible show error message in screen	Yes
Data upload failed (partial transmission)	Rollback partial upload, retry complete transmission	Yes
Query response malformed/corrupted	Discard response, retry query, validate JSON schema	Yes
Data synchronization conflict	Notify system	Yes
Invalid session token	Request new token, re-authenticate user, clear invalid session data	Yes
Cloud HTTP connection refused	Retry connection until success	Yes

Table 3.21: Cloud error handling

3.15 Dry Run

A dry run is a valuable analytical technique used to simulate and evaluate a system's behaviour without actually executing its code. This process enables system designers and developers to manually trace algorithms step by step, verifying their logic, data flow, and expected outcomes. By doing so, they can detect potential issues early, ensure compliance with system requirements, and validate the correctness of the design before implementation. Conducting a dry run not only helps in minimizing errors during the development phase but also enhances the overall reliability and efficiency of the final system. In this case, the dry run will focus on analysing whether pedestrians are granted early passage based on the number of button clicks registered before half of the red-light duration has elapsed. If the clicks occur after this midpoint, triggering a signal change would no longer be meaningful, since the remaining waiting time would already be short. THe following table will therefore demonstrate how the system reacts to different input scenarios, allowing observation of its logic and timing under various pedestrian interactions.

Line	Button Pressed?	Button clicks greater than Threshold?	Pedestrian light green, Traffic light red	Pedestrian light red, Traffic light green	Half of the time of red pedestrian light gone?	Wait until red pedestrian light finishes
1	0	0	1	0	0	0
2	1	0	0	1	0	1
3	1	0	0	1	1	1
4	1	1	0	1	0	0
5	1	1	0	1	1	1
6	1	0	1	0	x	0

Line	Change red pedestrian light to green before timeout	Buzzer activates to inform pedestrian passage	Traffic light switches to red
1	0	1	0
2	0	0	0
3	0	0	0
4	1	1	1
5	0	0	0
6	0	1	0

3.16 Theoretical Introduction

3.16.1 I2C

I2C, often called “I two C”, stands for Inter-Integrated Circuit. It is a synchronous communication protocol which is used in various devices from many different product families. I2C requires two lines for communication: SCL (Serial Clock) and SDA (Serial Data). I2C can connect to multiple devices on the bus with only the two lines. The controller device can communicate with any target device through an unique I2Caddress sent through the serial data line and, therefore, removes the potential problem of bus contention. I2C is a half-duplex communication, meaning that only a single controller or a single target can send data at the same time.

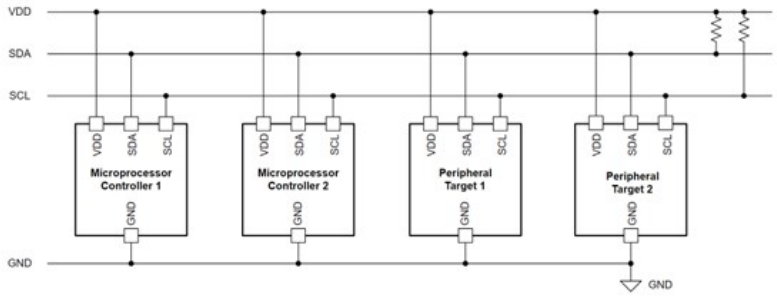


Figure 3.49: I2C implementation

In I²C communication, the SCL line is provided by the controller. Data transmission begins when the SDA line transitions from HIGH to LOW while SCL is HIGH (Start condition) and ends when SDA transitions from LOW to HIGH while SCL is HIGH (Stop condition). After the start condition, the controller sends an address frame followed by a read/write (R/!W) bit. The target then sends an acknowledgement bit (ACK) – SDA LOW – to confirm it has received the address, and if it didn’t, a NACK is received – SDA HIGH. Next, the controller sends one or more data frames, each followed by another ACK bit from the receiver. Data on the SDA line must be stable while SCL is High. The transmitter (Master or Slave) transitions the data level during the Low period of the SCL line, typically triggered by the falling edge.

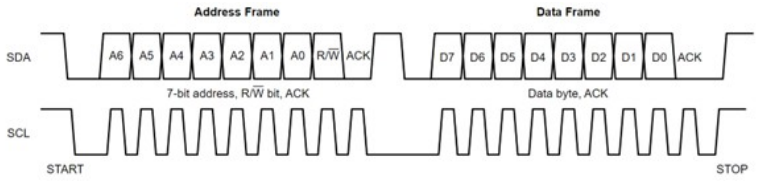


Figure 3.50: I2C Address and Data Frames

3.16.2 Radio Frequency Identification

Radio Frequency Identification, also known as RFID, is form of wireless communication that uses waves in the radio frequency portion of the electromagnetic spectrum read and transmit data from tags/cards to a reader, allowing for identification without direct contact. All RFID systems consist of three main components:
- Antenna: emits radio waves to communicate with RFID tags. Responsible for creating the electromagnetic field which “wakes up” the transponders in its range;
- Transceiver: the electronic unit that controls the antenna. It send the signal to tha transponder and receives the signals it sends back. Together with the antenna, it forms the RFID reader.
- Transponder: the tag or card which contains a microchip that stores information and an

antenna. It responds with its own data.

The different types of RFID systems are described in the Table 16.

RFID System Types	**Frequency Band**	**Average Frequency**	**Distance**
Low-Frequency RFID	30 kHz to 500 kHz	125 kHz and 134.2 kHz	¡10 cm
High-Frequency RFID	3 MHz to 30 MHz	13.56 MHz	1 m
UHF (Ultra High Frequency) RFID	300 MHz to 960 MHz	856 MHz and 960 MHz	12 m

Table 3.22: RFID system types

For our application, both High-Frequency RFID (Figure 3.5) and UHF RFID are applicable, however High-Frequency RFID was chosen due to cost issues.

3.16.3 SPI

SPI stands for Serial Peripheral Interface. It is a high-speed synchronous serial input/output port that allows a serial bit stream of programmed length (2 to 16 bits) to be shifted into and out of the device at a programmed bit-transfer rate. The SPI is normally used for communication between the device and external peripheral.

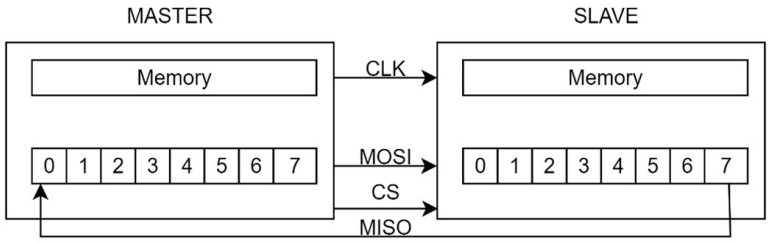


Figure 3.51: SPI communication

Similarly to I2C, implements a clock line (SCK) which serves as base for data transmission. It implements always one Master/controller (most commonly the microcontroller) and can admit several Slaves/peripherals. When data is sent from the Master to the Slave, it's sent through the data line, which is called MOSI (Master Output/Slave Input). If the Slave needs to send a response back to the Master, the Master will continue to generate a prearranged number of clock cycles, and the Slave will put the data onto a third data line called MISO (Master Input/Slave Output). Lastly, there is another line which is CS - Chip Select. This line is responsible for telling the peripheral whether it should wake up or not. This is, if the Master wants to communicate with that Slave, the CS line switches state (normally from HIGH to LOW) which means that that Slave will have to read/write data from/to the Master. This is especially advantageous when there are several peripherals connected which, therefore, requires that there is a CS line for each slave, and thus proves to be a limitation of SPI due to limited board GPIO pins. CLK, MOSI and MISO lines are common to every Slave.

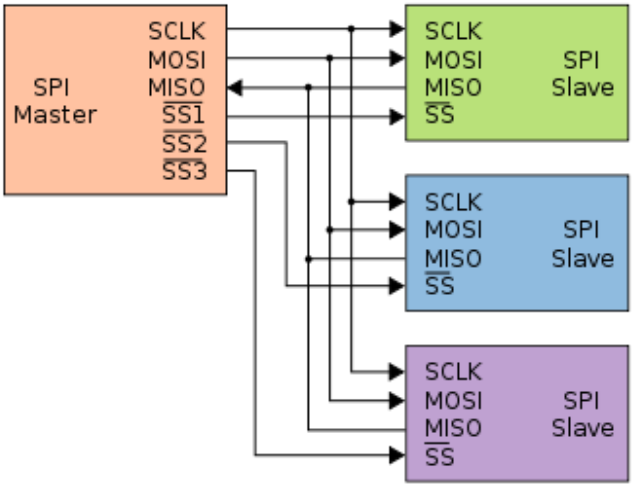


Figure 3.52: SPI communication between one master and many slaves

3.16.4 SOLID

The SOLID principles are a set of five guidelines used in object-oriented software design to create systems that are easier to understand, maintain, and extend. They encourage developers to structure code in a modular and flexible way, reducing complexity and avoiding tight coupling between components. The five principles are:

- Single Responsibility: states that each class should have only one responsibility;
- Open/Closed: meaning software should be open to extension but closed to modification;
- Liskov Substitution: ensures that subclasses can replace their base classes without altering system behavior;
- Interface Segregation: which promotes using smaller and more specific interfaces instead of large, general ones;
- Dependency Inversion: suggests depending on abstractions rather than concrete implementations.

3.16.5 Relational Database

A Relational Database, also known as RD, implements data organization in the form of tables, columns and rows. It has the ability of establishing relationships between information by joining tables, which makes it easy to understand and gain insights about the relationship between various data points. The various "main parts" of the system are called entities and can relate to one another. These can have several attributes (characteristics), which are pieces of data that relate to each instance of the entity. Therefore, transposing to the database organization, an entity is a table, which has several columns (the attributes) and several rows (data about one single entity). These databases also implement Primary Keys (PK), which uniquely identify each row in a relation – they must contain unique values –, and Foreign Keys (FK) which in an attribute in one relation that refers to the Primary Key of another relation: FK are used for fixed and permanent association between entities. The possible number of occurrences in one entity which is associated with the number of occurrences in another in defined by cardinality, where the most common types are One-to-One, One-to- Many and Many-to-Many.

3.16.6 Entity Relationship Diagram (ERD)

ERD, which stands for Entity Relationship Diagram, is a type of structural diagram used for database specification. It aims to specify the different entities within the scope, their properties and how they relate. An ERD can be in the form of different data models:
- Conceptual Model: aims to provide an overview of the system's relationships, by indicating the entities and their relationship from an overall perspective.
- Logical Model: enriches the the Conceptual model, by adding the entities' attributes and, thus, making up the columns of the database.
- Physical Model: implements the design blueprint of the actual database: it not only includes everything that the Logical model presents, but it also adds more specifications, such as, the primary and foreign keys. In order to implement many-to-many cardinality, an intermediary table (log table) is required to store intermediary data, registering events.

3.16.7 API

From a general perspective, an API (Application Programming Interface) is a tool that enables interaction between an application and a service without requiring direct or manual communication between them. The API acts as an intermediary layer, simplifying how the application accesses the service by providing a standardized and consistent way to request, access and exchange data.

3.16.8 HTTP

The Hypertext Transfer Protocol (HTTP) is the foundational protocol of the World Wide Web, defining a standart for data transfer over the web and creating the necessary foundations for the conection between a client and a server. It implements methods such as GET, POST, PUT, and DELETE.

3.16.9 Endpoints

An endpoint is a specific URL within an API that exposes an operation on a resource. It is defined by an HTTP method and a path, which together indicate what action will be performed and which data will be accessed or manipulated.

3.16.10 RESTful API

REST (Representational State Transfer) is an architectural style or set of principles for designing networked applications, focusing on scalable, stateless, and uniform web services. The REST API (Representational State Transfer Application Programming Interface), commonly referred to as RESTful API or RESTful Web API is an interface that allows different software systems to communicate over the web, by following the REST architectural style. It is the most common communication standard between computers over the internet. Restful APIs typically use HTTP as the transport protocol, in order to perform CRUD operations. They also name endpoints under the REST conventions.

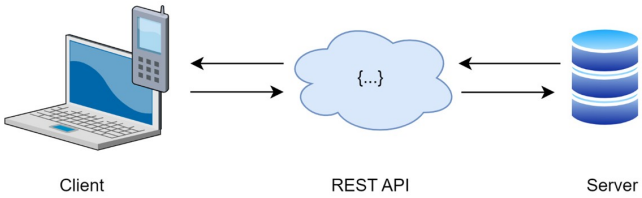


Figure 3.53: REST API interface

A REST API abstracts direct access to system resources by exposing controlled endpoints, which can be accessed using standard HTTP methods. In order for an API to be considered RESTful it must implement the following characteristics:
- Client–Server Architecture made up of clients, servers and resources, with requests managed by HTTP;
- Stateless client–server communication: no client information is stored between GET resquests, and each request is separate and unconnected;
- Uniform services: consistent and standardized way for clients to interact with resources – same HTTP methods are used consistently, responses follow in a standart format and errors are reported in consistent manner;
- Cacheable: allows for responses to be cacheable in order to improve efficiency and reduce server load – upon repeated requests of the same resource, the responses can be served locally from a cache (closer to the client), reducing latency;
- Layered System: REST architecture allows the system to be composed of hierarchical layers (e.g., client, proxy, gateway, server) without the client needing to know the underlying details: improves scalability and security.

3.16.11 DDS (Data Distribution Service)

DDS is a middleware communication protocol standard designed to facilitate scalable and reliable real-time data exchange in distributed systems. Its core operation is based on a data-centric publish/subscribe model, where data producers (publishers) share information on specific topics and data consumers (subscribers) receives updates by expressing interest in these topics. The information flow is regulated by QoS policies established between the entities in charge of the data exchange, and only entities belonging to a same domain can interact. This approach ensures loose coupling between

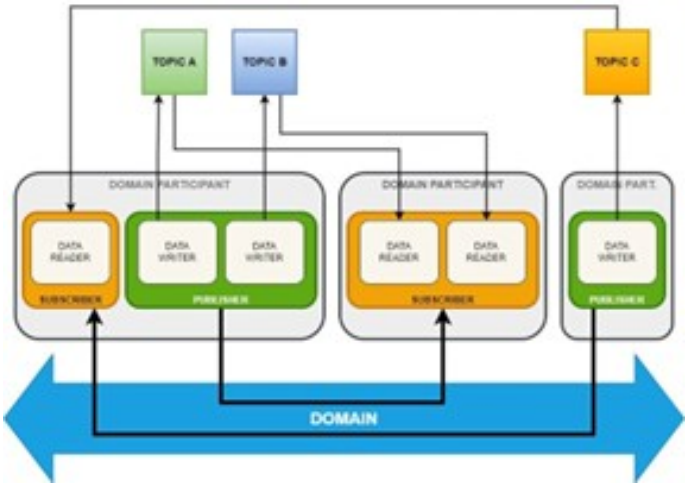


Figure 3.54: DDS communication

components, allowing seamless communication regardless of their direct dependencies

or network location. DDS manages the complexity of network communications through a middleware layer that abstracts the underlying transport layer. It supports automatic discovery of participants, dynamic reconfiguration, and robust fault-tolerance, making it ideal for large-scale applications. Quality of Service (QoS) parameters—like reliability, durability, latency, and security—are highly configurable, allowing developers to tailor data delivery behaviours to meet stringent real-time requirements. DDS systems operate across different platforms, programming languages, and vendors, providing vendor, network, and language independence for wide interoperability.

3.16.12 Fast DDS

Fast DDS is a specific implementation of the DDS standard written in C++. It places special emphasis on performance, resource efficiency, and flexibility, supporting real-time features and advanced transport configuration. Fast DDS is the default middleware for ROS 2 and provides both a high-level DDS-compliant API and a low-level RTPS API for fine-grained control. It offers robust security options, built-in discovery mechanisms, and is completely free and open source, making it an excellent choice for embedded and robotics applications.

3.16.13 UDP

The User Datagram Protocol (UDP) is a transport layer protocol defined by the Internet Protocol (IP) suite. It provides a connectionless and unreliable method for sending datagrams between networked devices. Unlike TCP, UDP does not establish a connection before data transfer and does not guarantee that packets will arrive in order, without duplication, or even at all. UDP's simplicity makes it fast and efficient, as it eliminates the overhead of connection management, acknowledgments, and retransmissions. Each message, known as a datagram, is sent independently and contains just enough information for delivery, including the source port, destination port, length, and a checksum for basic error detection. Because of its low overhead, UDP is commonly used in applications where speed and timing are more important than reliability, such as video streaming, voice communication (VoIP), online gaming, and sensor or control systems. It also supports broadcast and multicast, allowing one sender to reach multiple receivers simultaneously — something that is not supported by TCP, which only allows point-to-point communication.

3.16.14 I/O system

An I/O system is composed by the I/O devices, the device drivers, the I/O subsystem, the operating system's kernel and the application and aims to provide a uniform method to access I/O devices and to abstract the application developer from the hardware's specific details. Figure 3.55 illustrates a representation of an I/O subsystem. To provide a better understanding of the referred system, some of its integrant parts will be explained in greater detail in the following sections.

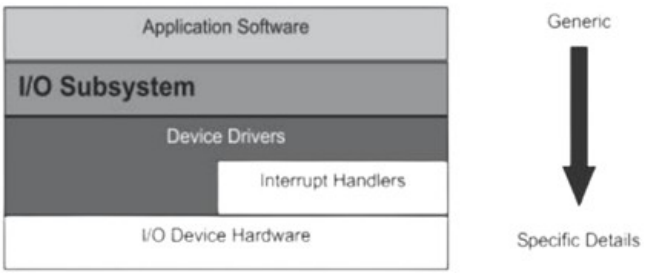


Figure 3.55: : I/O subsystem

3.16.15 Device Drivers

Device drivers are software components that act as an interface between the I/O subsystem and hardware devices, ultimately to make a uniform communication between the application and the hardware possible. These can be divided into two categories:
• Character device drivers: Character device drivers receive a single character of data, making them easier to implement and are used for a wide array of components, typically devices with unpredictable data arrival patterns, such as user input devices. Keyboards, serial ports and terminals are all examples of devices for which character device drivers are typically implemented;
• Block device drivers: Block device drivers, as the name indicates, establish communication between the operating system and devices that manage data in fixed-size blocks (the block size is usually imposed by the underlying hardware) such as hard drives, for example. The transferring of data in blocks proves to be a more efficient approach compared to the one used in the block device driver's single character receiving counterpart. Even though the referred device drivers are more complex due to the used data transfer mechanism and the consequent need for buffering, these prove to be more adequate for devices that need to transfer large amounts of data at a time.

3.16.16 I/O subsystem

Each device driver can provide a specific set of APIs to the applications, which would force the latter to take in account the nature and restrictions imposed by the I/O device. This would mean that the application would then be dependent on the several connected devices as well as the implementation of their drivers, which is not feasible in more complex systems. In order to reduce this dependence, I/O subsystem are implemented by defining a standard set of functions for I/O operations, providing the applications a unified I/O access to several devices while also abstracting them from their specific implementations.

3.16.17 RAII

RAII (Resource Acquisition Is Initialization) is a C++ idiom which states that the lifetime of a resource is bound to the lifetime of an object. This ensures that acquisition happens in the constructor and release happens in the destructor, giving automatic, exception-safe resource management. Thus, RAII removes the need for manual `delete`, `close`, `unlock`, etc., in most cases, reducing leaks and bugs.

3.16.18 Signals

Signals are software interrupts that can be sent to a process or thread. They are a fundamental concept in Linux systems, used for a variety of purposes such as inter-process communication, error handling, and process management. Each signal has a default action associated with it; however, this mechanism is highly flexible. Developers can't only use predefined signals but also define and send custom signals, which can be particularly useful in specialized application scenarios. These custom signals are typically assigned numbers outside the range of predefined ones. To handle signals, processes can define signal handlers, which specify how a process should respond when a particular signal is received. Some of the most used signals in Linux include:
• SIGINT – Generated when the user sends an interrupt signal.
• SIGKILL – Forces the process to immediately terminate.
• SIGALRM – Triggered when a timer set by the alarm function expires.
• SIGSTOP – Suspends a process, pausing its execution.
• SIGCONT – Resumes a process that was previously stopped.

• SIGTERM – A software termination signal, allowing a process to shut down gracefully.

3.16.19 Design Patterns

Design patterns can be described as typical solutions to commonly occurring problems in software design. They provide well-defined templates for structuring code, managing responsibilities, and promoting best practices without prescribing a specific implementation. They are divided into three categories (creational, structural and behavioral) and in each there are several patterns. Each addresses a different issua by providing a standart way of solving a common problem. Examples include the Singleton pattern, Mediator pattern and Visitor pattern. The main goal of design patterns is that by applying them, code becomes cleaner, more maintainable, scalable and modular.

3.16.20 Software Architectural Patterns

Software architectural patterns provide proven templates for organizing the components and interactions of a system. They define high-level structures, responsibilities, and communication flows, helping architects design systems that are maintainable, scalable, and extensible. Common examples include Layered Architecture, Event-Driven Architecture (EDA), and Microservices Architecture, each addressing different system requirements such as modularity, performance, and loose coupling. These patterns serve as blueprints that guide design decisions and ensure consistency across large, complex systems.

3.16.21 Concurrency Patterns

Concurrency patterns are design approaches for managing multiple threads or processes that execute simultaneously, ensuring that shared resources are accessed safely and efficiently. They provide solutions to common problems such as race conditions, deadlocks, and synchronization overhead. Examples include the Producer-Consumer pattern, Read-Write Locks, and Thread Pools. By applying these patterns, software can achieve parallel execution, improved responsiveness, and better resource utilization, which is particularly important in real-time or high-throughput systems.

Chapter 4

Implementation

4.1 Buildroot Image Configuration

The embedded system for the AllWays Safe project is built on a custom Linux image generated with Buildroot. This approach enables precise control over the software stack, ensuring that only the components required for system operation are included.

The Buildroot configuration was carefully tailored to balance minimal footprint with robustness and performance. Core libraries, dependencies, and APIs were selected to support the system's functional requirements while keeping resource usage low. Emphasis was placed on reliability, fast boot times, and long-term maintainability.

The following subsections detail the principal Buildroot configuration decisions and component selections, explaining how each contributes to a streamlined, efficient, and optimized embedded Linux platform.

4.1.1 Intersection ControlBox

Access Point Configuration

To enable emergency vehicles to connect directly to an intersection control box, the Raspberry Pi acting as a control box is configured to operate as a Wi-Fi access point. This allows vehicles to join a local LAN and transmit emergency messages in real time. To achieve this, the Buildroot system includes the following packages: `hostapd`, `dnsmasq`, and `brcmfmac-sdio-firmware-rpi-wifi`.

The `brcmfmac-sdio-firmware-rpi-wifi` package provides the required firmware for the Broadcom Wi-Fi chipset used by the Raspberry Pi. Although the `brcmfmac` driver is included in the Linux kernel, it depends on external firmware files to initialize the wireless hardware. Without this firmware, the `wlan0` interface would not be created, preventing both access point and client Wi-Fi operation. The firmware is explicitly loaded during system startup to ensure that the wireless interface is available before network services are started.

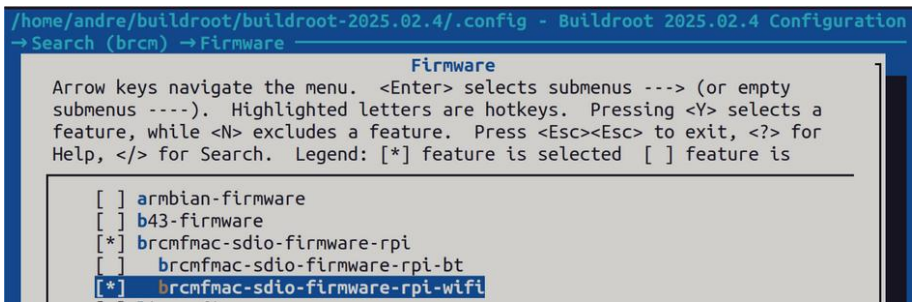


Figure 4.1: Wireless interface configuration for the Raspberry Pi access point

`hostapd` is responsible for creating the access point and managing authentication, encryption, and wireless network settings. Its configuration file (`/etc/hostapd.conf`) defines key parameters such as:

- `interface=wlan0`: specifies the wireless interface.
- `ssid=wifimari`: sets the network name.
- `hw_mode=g`, `channel=6`: define Wi-Fi mode and channel.
- `wpa=2`, `wpa_passphrase=wifimari`, `wpa_key_mgmt=WPA-PSK`, `wpa_pairwise=TKIP`, `rsn_pairwise=CCMP` configure WPA2 security for encrypted communication.

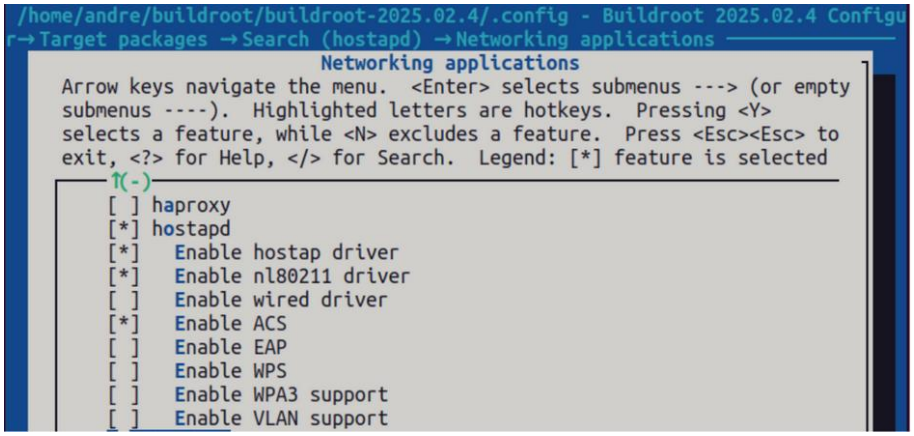


Figure 4.2: `hostapd` configuration in Buildroot for the access point

DHCP is provided by `dnsmasq` provides and DNS services for devices connecting to the access point. Its configuration (`/etc/dnsmasq.conf`) specifies:

- `interface=wlan0`: serve DHCP requests on the wireless interface.
- `dhcp-range=192.168.4.2,192.168.4.20,255.255.255.0,24h`: defines the IP address range for connected clients.
- `address=/gw.wlan/192.168.4.1`: sets the gateway for the network.

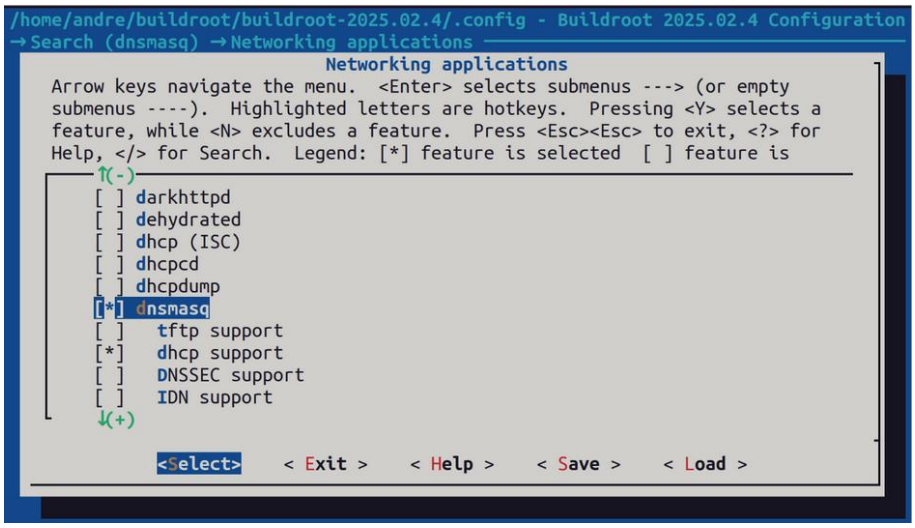


Figure 4.3: `dnsmasq` configuration providing DHCP for connected emergency vehicles

The startup script (`/init.d/S45accesspoint`) automates the initialization:

- Loads the wireless driver (`modprobe brcmfmac`) and brings `wlan0` up.
- Enables IP forwarding if routing is needed.
- Starts `hostapd` and `dnsmasq` as background processes.
- Provides a clean stop procedure to terminate services and reset the interface.

By configuring the intersection control box as an access point, emergency vehicles can seamlessly connect via Wi-Fi, join the local LAN, and send emergency messages using Fast DDS. The combination of `hostapd` for AP management, `dnsmasq` for DHCP/DNS, and the Raspberry Pi Wi-Fi firmware ensures a stable and secure wireless network for real-time communications.

SPI Configuration for RFID Reader

The intersection control box rely on the SPI bus to communicate with the RFID reader, which is used to detect cards presented by disabled pedestrians. These cards allow the system to extend the crossing time at intersections, providing additional safety for pedestrians with mobility challenges.

To enable SPI on the Raspberry Pi within the Buildroot environment, the device tree overlay is configured as follows:

```
1 dtoverlay=spi0-1cs
```

Additionally, it was necessary to modify the kernel configuration via `Kernel make menuconfig` to compile SPI support as a *builtin* rather than a module. This ensures that the SPI bus and associated drivers are initialized early during the boot process, which is critical for the RFID reader to function correctly as soon as the system starts.

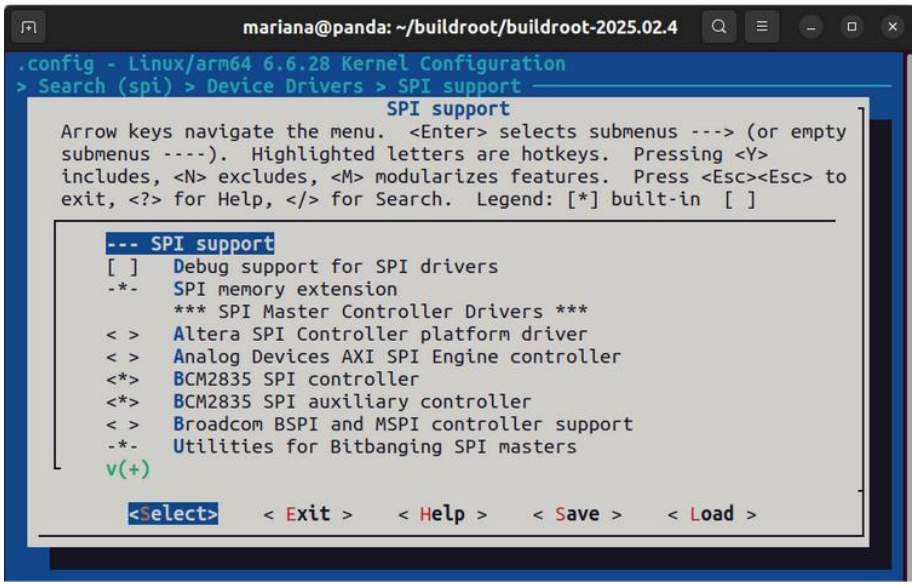


Figure 4.4: `spi` configuration in Buildroot

The SPI interface provides a fast and reliable communication channel between the Raspberry Pi and the RFID module. By reading the cards presented by pedestrians, the system can dynamically adjust the crossing timing, guaranteeing that users requiring extra time can safely traverse the intersection.

NTP Configuration

To ensure accurate timestamps for all events and communications, the intersection control box use the Network Time Protocol (NTP) to synchronize its system clock. Correct date, hour, and year settings are critical for logging and auditing purposes, especially when emergency vehicles pass through intersections and transmit messages to the cloud. Accurate timestamps allow the Traffic Management Center (TMC) to reconstruct events reliably and maintain an accurate historical record of system activity.

NTP is a standard protocol designed to synchronize the clocks of computer systems over packet-switched, variable-latency networks. It operates by exchanging timestamped messages between the client (in this case, the Raspberry Pi) and remote NTP servers, which are considered accurate reference clocks. The protocol compensates for network delays and clock drift, ensuring that the system time remains precise.

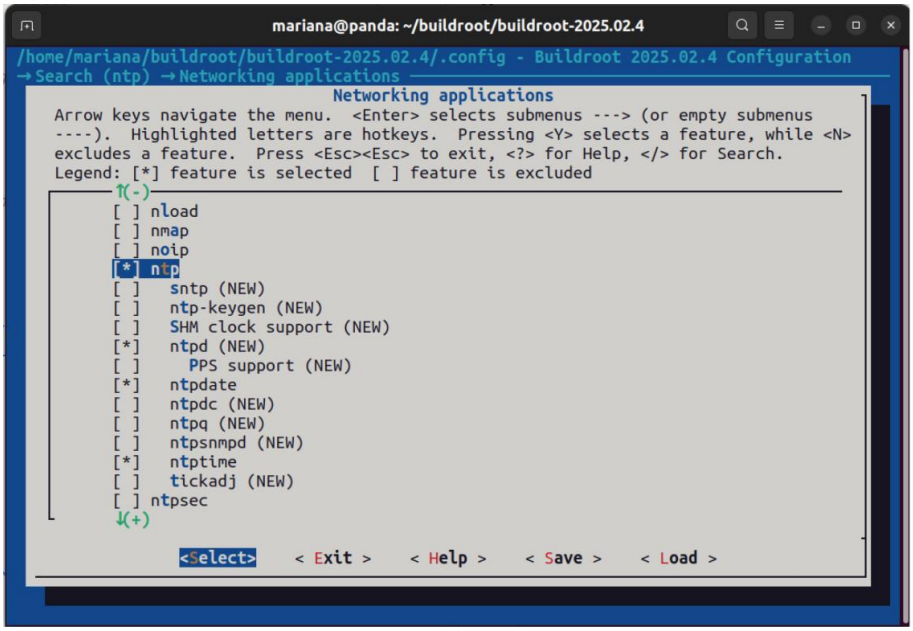


Figure 4.5: NTP service configuration in Buildroot

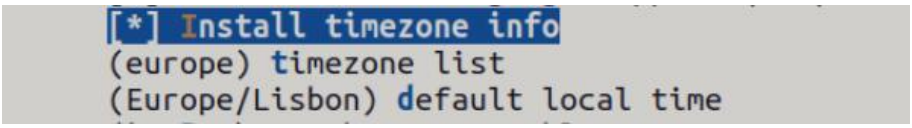


Figure 4.6: NTP client configuration file for system time synchronization

In this project, NTP is configured within Buildroot and enabled on system startup. Once active, it continuously synchronizes the system clock with configured NTP servers, guaranteeing that all outgoing messages from the Raspberry Pi, including those sent by emergency vehicles, carry the correct timestamp. This synchronization is essential for post-event analysis, performance monitoring, and coordination with other nodes in the traffic management system.

GPIO Control with `libgpio`

The `libgpio` library is included in the Buildroot configuration to provide a simple and reliable interface for controlling the General Purpose Input/Output (GPIO) pins on the Raspberry Pi. In the AllWays Safe system, `libgpio` is used to manage the traffic and pedestrian semaphores, allowing the embedded software to set and clear individual pins to control lights and signals.

GPIO pins provide a direct interface between the Raspberry Pi and external hardware components. By using `libgpio`, the software can interact with these pins at a higher abstraction level, without dealing directly with low-level register access. This simplifies application development, reduces potential errors, and ensures consistent behavior across different Raspberry Pi models.

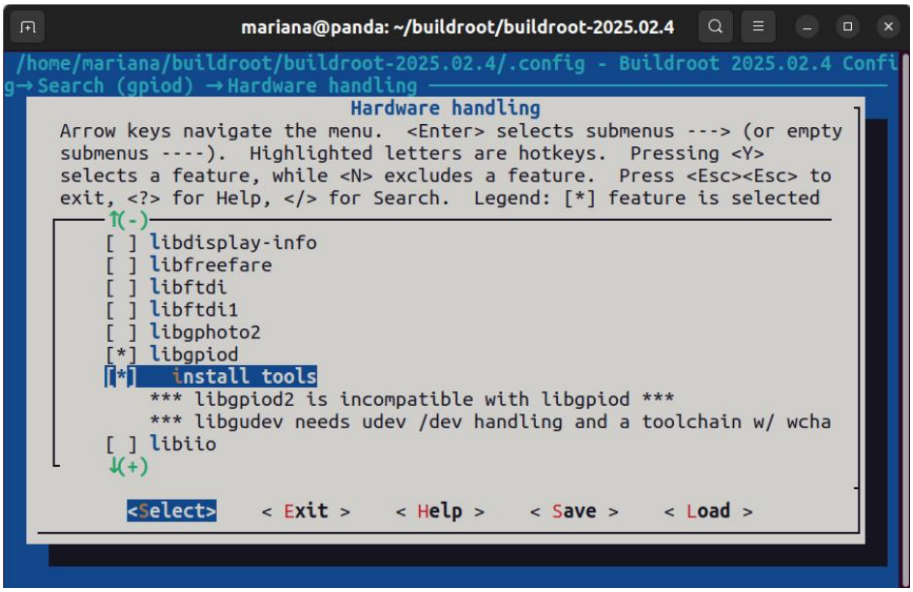


Figure 4.7: `libgpio` configuration in Buildroot

In practice, `libgpio` allows the system to:

- Set a pin high or low to turn on or off traffic lights and pedestrian signals.
- Read the state of input pins, if necessary, for monitoring external switches or sensors.
- Provide a thread-safe and maintainable API for semaphore control, which is essential for ensuring timely and reliable signal changes.

By integrating `libgpio` into the Buildroot environment, the system ensures that semaphore control is both efficient and easy to maintain, while remaining compatible with the rest of the embedded Linux software stack.

HTTP-Based Communication for Cloud Integration

To enable reliable communication between the Intersection Control Box and the cloud infrastructure, both `libcurl` and `curlpp` were included in the Buildroot configuration. These libraries provide comprehensive support for data exchange over network protocols, which is essential for interacting with the cloud through the RESTful API already mentioned.

`libcurl` is a widely used, low-level client-side URL transfer library that supports HTTP and HTTPS protocols. Within the AllWays Safe system, `libcurl` is responsible for handling secure data transmission, including sending light status updates and alerts to the cloud, as well as receiving configuration parameters for creating the traffic and pedestrian semaphores. Its robustness, protocol support, and efficient resource usage make it well suited for embedded Linux environments.

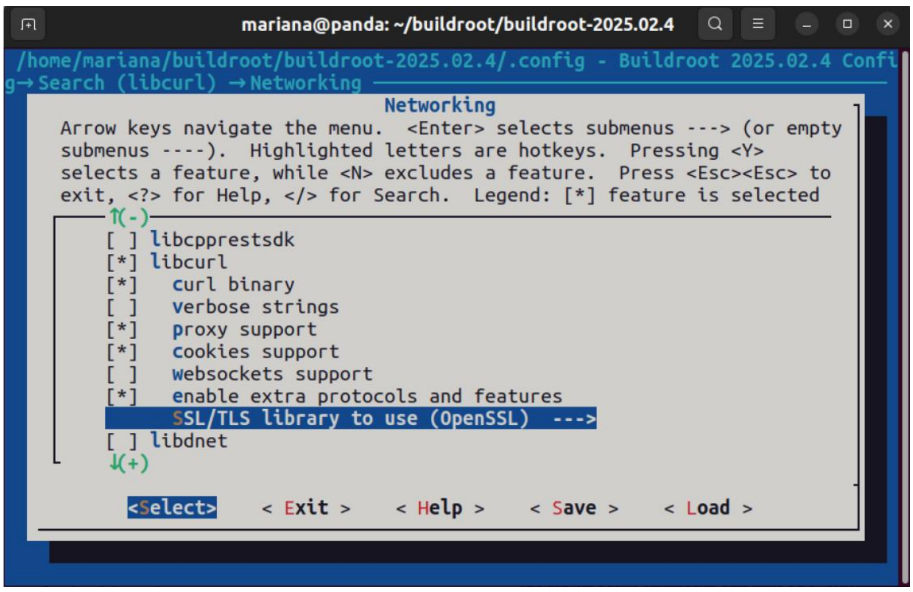


Figure 4.8: `libcurl` configuration in Buildroot

`curlpp` is a C++ wrapper built on top of `libcurl` that provides a higher-level, object-oriented interface. By using `curlpp`, application development was simplified through improved code readability, easier error handling, and better integration with C++-based system components. This abstraction reduces development complexity while still leveraging the performance and reliability of `libcurl`.

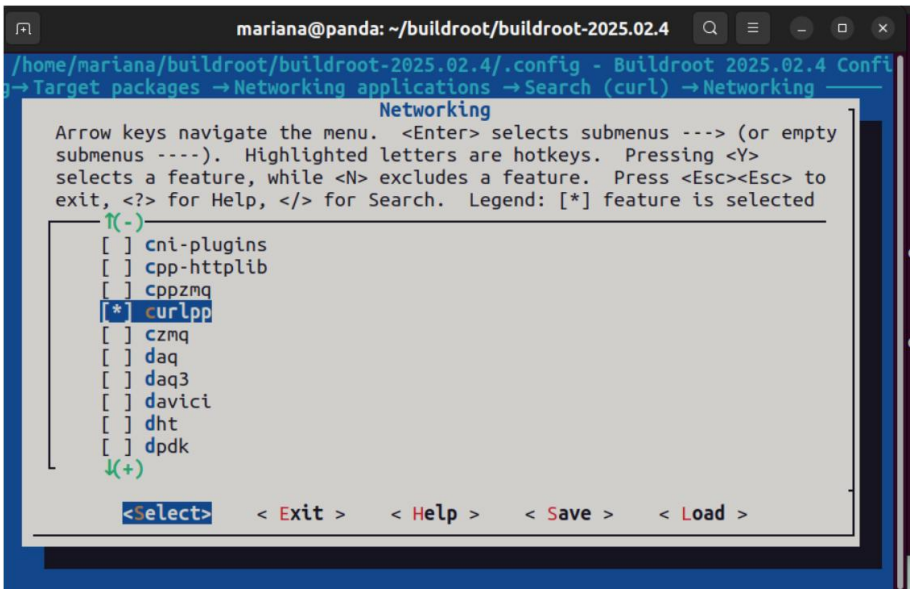


Figure 4.9: `curlpp` configuration in Buildroot

To complement these libraries, the `json-modern-cpp` library was integrated into the Buildroot configuration to handle JSON-formatted data, which is the primary data format used in RESTful API communications. This library provides a modern, intuitive, and efficient C++ interface for parsing incoming JSON responses from the cloud and constructing outgoing JSON requests. For example, it is used to extract configuration parameters such as SSIDs, passwords, and traffic signal settings, as well as to format status updates and alert messages before sending them to the cloud. By using `json-modern-cpp`, the system ensures that data exchanged with the cloud is structured, consistent, and easy to process within the C++ applications.

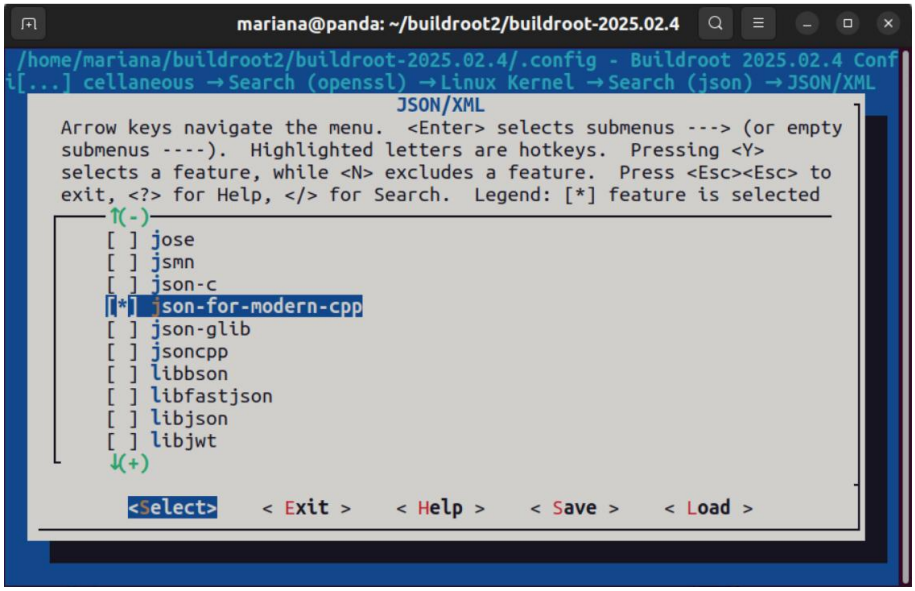


Figure 4.10: `json-modern-cpp` configuration in Buildroot

The inclusion of `libcurl`, `curlpp`, and `json-modern-cpp` allows the embedded applications to implement RESTful API communication in a secure, structured, and maintainable manner. Together, these libraries enable the Raspberry Pi to efficiently exchange JSON-formatted data with cloud services, while maintaining a lightweight software stack that aligns with the system's performance and reliability requirements.

Fast DDS Configuration and Dependencies

The intersection control box acts as a *subscriber* in the Fast DDS communication framework, enabling it to receive messages from emergency vehicles or other system nodes in real time. To integrate Fast DDS into the Buildroot-based embedded system, several additional packages and dependencies were required, including `asio`, `Fast CDR`, `Fast DDS`, and `foonathan_memory`. These packages were imported from GitHub and integrated into Buildroot by creating the necessary `Config.in` and `*.mk` files, after which they were added as new packages. Running `make` subsequently built the libraries and made them available for application development.

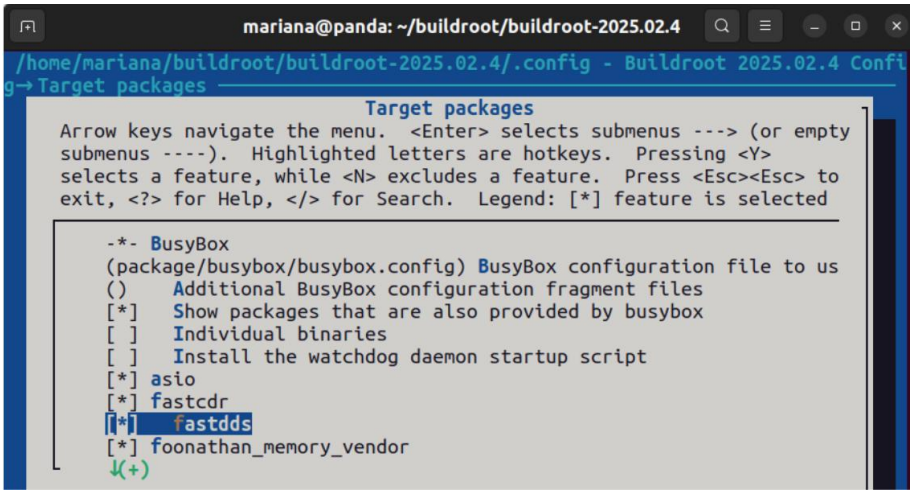


Figure 4.11: Fast DDS libraries built in Buildroot

Each of these packages serves a specific purpose for Fast DDS operation:

- `asio`: Provides asynchronous input/output capabilities for network communication, which Fast DDS relies on for efficient message transmission and reception without blocking the main application thread.
- `Fast CDR`: Implements the Common Data Representation (CDR) serialization format required by Fast DDS, allowing structured data types to be encoded and decoded consistently across different nodes in the system.
- `Fast DDS`: The main middleware library implementing the DDS (Data Distribution Service) protocol, handling publish-subscribe communication, message routing, and discovery of participants on the network.
- `foonathan_memory`: Provides efficient memory management utilities, which Fast DDS uses for dynamic allocation of message buffers and other internal structures, ensuring high performance and low latency communication.

In addition, the `TinyXML` library, which is already included in Buildroot, is used in this context to parse XML-based configuration files that define QoS (Quality of Service) policies for Fast DDS. QoS rules govern message reliability, delivery deadlines, and resource limits, ensuring that the subscriber in the intersection control box receives emergency messages with the appropriate priority and timing guarantees. Using `TinyXML` allows the system to load and interpret these configurations at runtime in a lightweight and consistent manner.

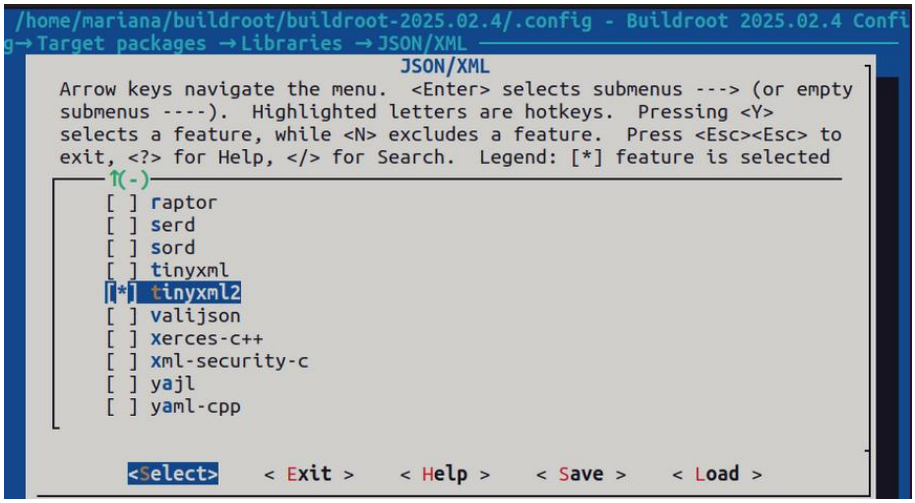


Figure 4.12: `TinyXML` library for QoS configuration

Together, these dependencies provide a complete and efficient environment for Fast DDS to operate on the embedded platform, enabling the intersection control box to participate as a subscriber in the system's real-time communication network.

4.1.2 Emergency Vehicle Control Box

ADC Configuration for battery monitoring

The Texas Instruments `ADS1115` analog-to-digital converter (ADC) is included in the embedded system to monitor the battery voltage of the emergency vehicle control box. The `ADS1115` is configured via the `Config.txt` file with the following device tree overlay:

```
1 dtoverlay=ads1115,addr=0x48
```

This configuration ensures that the ADC is correctly recognized by the Raspberry Pi and assigned the I2C address `0x48`.

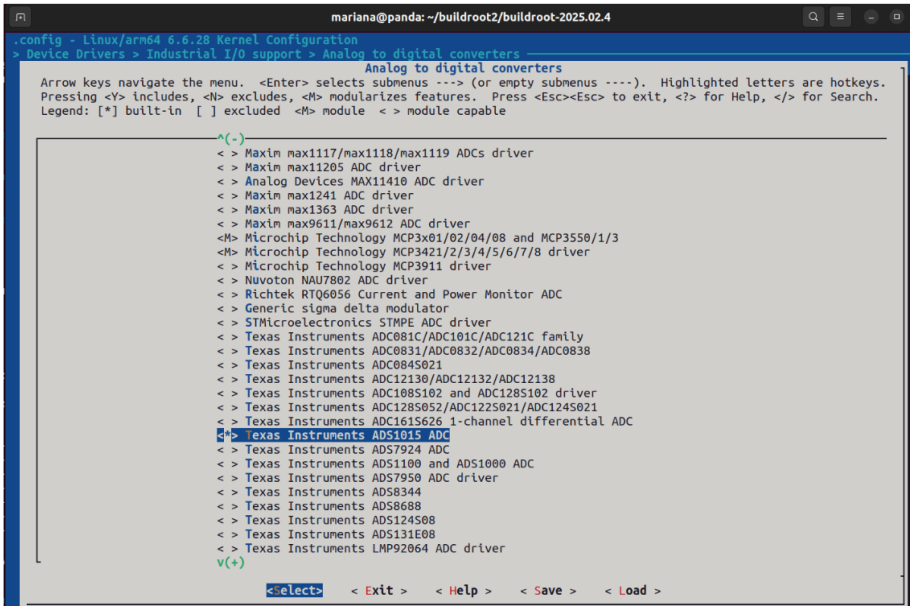


Figure 4.13: `ADS1115S` configuration in Buildroot

Voltage readings can be accessed directly through the Linux IIO (Industrial I/O) interface, without the need for a custom API. For example, the raw battery voltage can be read from:

```
/sys/bus/iio/devices/iio:device0/in_voltage_raw0
```

Similarly, other channels can be accessed by reading the corresponding `in_voltage_rawX` files, allowing flexible monitoring of multiple inputs. In addition, information such as the ADC's sampling frequency and scale can also be obtained from the IIO subsystem, providing insight into how the measurements are being taken and allowing proper interpretation of the raw values. This approach provides a simple, reliable, and low-overhead method for obtaining precise voltage measurements, which is essential for monitoring the battery that powers the emergency vehicle control box.

Wireless Network Configuration with `wpa_supplicant` and `iw`

To enable the emergency vehicle Raspberry Pi to connect to the local wireless network created by nearby control boxes, `wpa_supplicant` is included in the Buildroot configuration. `wpa_supplicant` is a widely used Linux daemon responsible for managing wireless connections, authenticating with Wi-Fi access points, and negotiating security protocols such as WPA and WPA2. It allows the Raspberry Pi to automatically associate with and maintain a secure connection to a specified access point, which is essential for ensuring that emergency messages can be transmitted through the local LAN using Fast DDS.

The configuration file `wpa_supplicant.conf` defines the wireless network credentials and security parameters:

```
ctrl_interface=/var/run/wpa_supplicant
update_config=1
country=PT

network={
  ssid="SSID_NAME"
  psk="password"
  key_mgmt=WPA-PSK
  priority=1
}
```

The `network` block is critical for enabling the Raspberry Pi to join a specific Wi-Fi network. The `ssid` specifies the network name to connect to, `psk` defines the pre-shared key (password), and `key_mgmt` indicates the authentication protocol, in this case WPA-PSK. The `priority` parameter allows multiple networks to be managed, giving precedence to the most important connection. By providing these parameters, `wpa_supplicant` can automatically detect, authenticate, and maintain a connection to the desired access point.

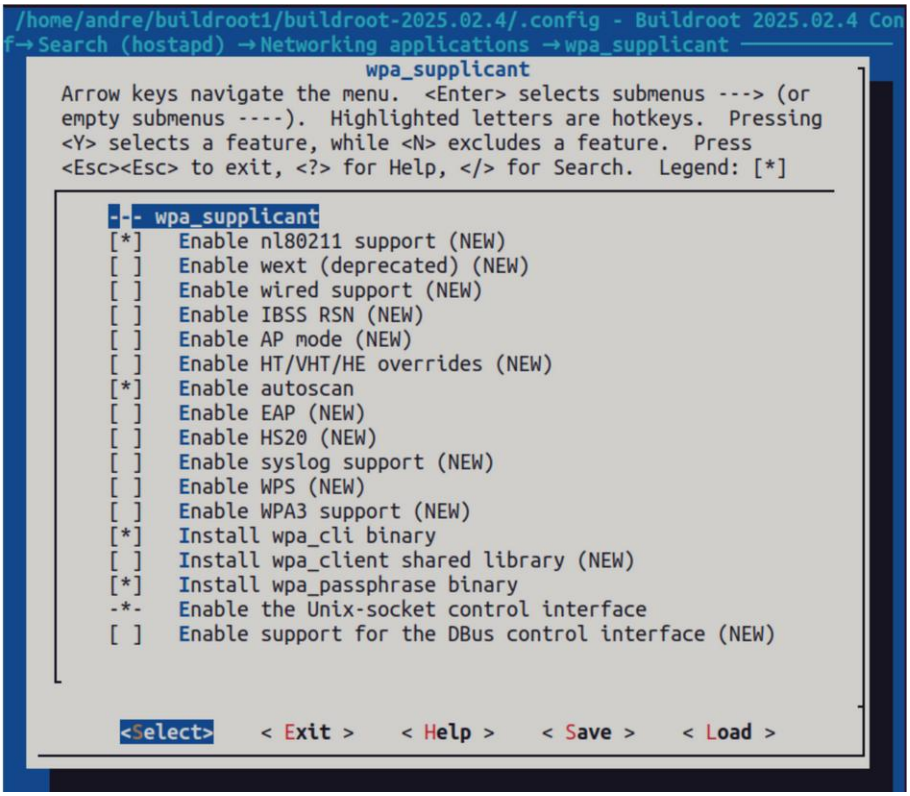


Figure 4.14: `wpa_supplicant` configuration in buildroot

The `network/interfaces` file complements this configuration by defining the network interfaces and their behavior:

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp
  pre-up /etc/network/nfs_check
  wait-delay 15
  hostname raspAndre

auto wlan0
iface wlan0 inet dhcp
  post-down killall -q wpa_supplicant
  hostname raspAndre
```

Here, `eth0` (Ethernet) and `wlan0` (wireless) are configured to obtain IP addresses via DHCP. The `pre-up` and `post-down` commands ensure that required services are started or stopped in coordination with the interface, maintaining network stability. This configuration allows the Raspberry Pi to seamlessly integrate into the LAN established by nearby control boxes, enabling the Fast DDS middleware to transmit emergency messages reliably.

Additionally, the `iw` utility is included in the Buildroot configuration to provide command-line control and monitoring of wireless interfaces. It allows the system to scan for available access points, query signal quality, and perform other diagnostic or management

tasks. While `wpa_supplicant` handles automatic connection and authentication, `iw` is valuable for troubleshooting and ensuring that the emergency vehicle Raspberry Pi can detect and join the correct network.

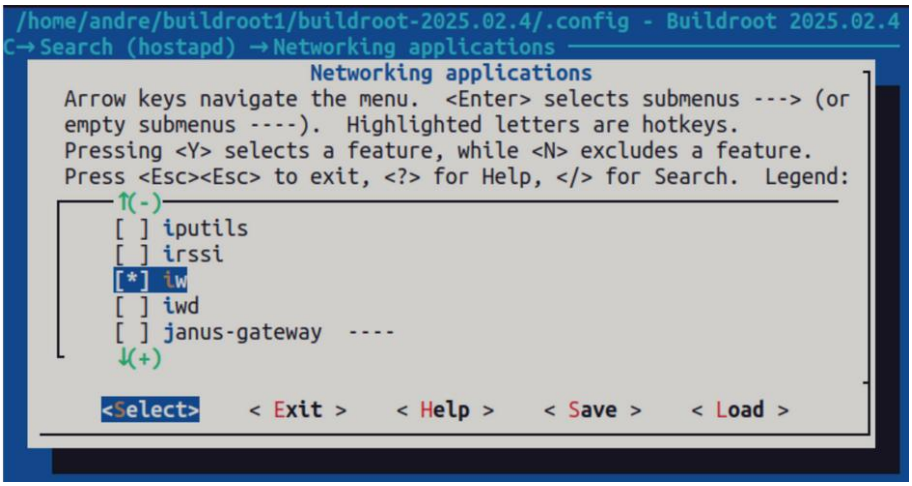


Figure 4.15: `iw` configuration in Buildroot

HTTP-Based Communication for Cloud Integration

For the control box installed on emergency vehicles, the same network and HTTP communication configurations described in section 4.1.1 were reused. These configurations were adapted specifically to support dynamic network access to nearby control boxes during emergency operations.

In this scenario, the emergency vehicle queries the Traffic Management Center (TMC) through the cloud using the RESTful API to retrieve the SSID and corresponding password of control boxes located in its vicinity. The `json-modern-cpp` library is used to parse the JSON-formatted responses received from the TMC, enabling the system to efficiently extract network credentials and other relevant configuration parameters. This mechanism enables the vehicle to automatically connect to the local wireless network of a nearby control box as it approaches the area.

Once connected to the same network, the emergency vehicle can directly transmit the emergency message to the control box, ensuring timely and reliable dissemination of priority signals. The underlying communication principles, protocols, and security mechanisms remain unchanged from those previously described, as the system continues to rely on HTTP-based RESTful communication, JSON-formatted data exchange, and cloud-mediated coordination.

This design minimizes configuration complexity by reusing an established communication framework while extending its functionality to support emergency vehicle prioritization.

Fast DDS Configuration and Dependencies

The emergency vehicle Raspberry Pi uses the Fast DDS middleware to send emergency messages to nearby intersection control boxes when approaching an intersection. Unlike the control boxes, which primarily act as subscribers, the emergency vehicle functions as a *publisher*, leveraging the same Fast DDS configuration and dependencies (`asio`, `Fast CDR`, `Fast DDS`, and `foonathan_memory`) for compatibility and interoperability.

Fast DDS enables the emergency vehicle to automatically discover nearby control boxes and transmit messages with the appropriate priority. QoS rules, managed via `TinyXML`, ensure reliable and timely delivery. Integration with the existing network stack, including Wi-Fi connectivity through `wpa_supplicant` and `iw`, allows the vehicle to join the local LAN of a control box and publish emergency messages efficiently.

4.2 Shared Software Components

4.2.1 PWM Device Driver

The PWM Device Driver was implemented with the goal of being able of controlling a PWM's frequency and *duty-cycle* on onr or many GPIO pins, known *à priori* of the board's boot.

In order to interface with the kernel-space created device driver, a user-space interface was created (Listing 4.1), which used the `ioctl` API to bridge both spaces.

```
class PWM_DeviceDriver
{
    private:
        int file_descriptor;
        static bool insertKernelModule();
        static bool removeKernelModule();
        int open_dd();
        int close_dd();
    public:
        PWM_DeviceDriver();
        ~PWM_DeviceDriver();
        int setValue(int32_t freq, int duty) const;
        void disable() const;
};
```

Listing 4.1: PWM_DeviceDriver Class

And for it to know the provided API functions the include on Listing 4.2 was necessary.

```
extern "C"{
#include "LinuxDeviceDriver/pwm_dd.h"
}
```

Listing 4.2: Include on PWM_DeviceDriver class

The mentioned `open_dd()` and `close_dd()` methods are responsible for performing the "insert module" and "remove module" from kernel operations on program start-up and on program shutdown.

Considering the kernel-space code, in order to implement all the `write/read` and `ioctl` operations, the file operations must first be specified.

```
static struct file_operations pwmDevice_fops = {
   .owner = THIS_MODULE,
   .write = pwm_device_write,            // terminal
   .read = pwm_device_read,
   .unlocked_ioctl = pwm_device_ioctl,   // user-applications
   .release = pwm_device_close,
   .open = pwm_device_open,
};
```

Listing 4.3: File Operations for the PWM Device Driver

This structure is used for the device driver initialization (Listing 4.4 has the overview of the `pwm_init` implementation code) upon kernel insertion instruction has been specified.

```
static int __init pwm_init(void)

   /* Allocate device numbers */
   alloc_chrdev_region(&DEV_major_minor, 0, NUM_PINS, DEVICE_NAME)
   /* Create class */
   pwm_class = class_create(CLASS_NAME)
   /* Create device nodes for each minor */
   for (int i = 0; i < NUM_PINS; i++) {
```

```
9         char dev_name[16];
10        snprintf(dev_name, sizeof(dev_name), "pwm%d", i);
11        device_create(pwm_class, NULL, MKDEV(MAJOR(DEV_major_minor),
              i), NULL, dev_name));
12    }
13    /* Initialize character device - specify fops */
14    cdev_init(&c_dev, &pwmDevice_fops);
15    c_dev.owner = THIS_MODULE;
16    /* Add character device to system */
17    cdev_add(&c_dev, DEV_major_minor, NUM_PINS)
18    /* Map hardware registers */
19    s_GPIO  = (GPIORegister *)  ioremap(GPIO_BASE_ADDR, sizeof(
          GPIORegister));
20    s_PWM   = (PWMRegister *)   ioremap(PWM_CONFIG_ADDR, sizeof(
          PWMRegister));
21    s_CLK   = (CLKRegister *)   ioremap(CLK_CONFIG_ADDR, sizeof(
          CLKRegister));
22    /* Set up PWM clock first */
23    init_clk(s_CLK);
24    /* Configure GPIO pins for PWM respective alternate function */
25    for (int i = 0; i < NUM_PINS; i++)
26        SetGPIOFunction(s_GPIO, Pins[i], PWMconfig[Pins[i]].FSELx,
              s_CLK);
27
28    for (int i = 0; i < NUM_PINS; i++)
29        SetPWM(s_PWM, Pins[i], 0, 0, s_CLK);
30
31    return 0;
```

Listing 4.4: Device Driver Initialization

When module is removed from kernel the `cleanup()` function is called (Listing 4.5), where the memory cleanup function is called in order to clean all allocated resources (Listing 4.6) — this function is also called whenever any resource allocation fails or the program returns an error in an operation.

```
1  static void __exit cleanup(void)
2  {
3      pr_alert("PWM Driver Exit function called\n");
4
5      /* Disable PWM by setting pins back to default state (input) */
6      for (int i = 0; i < NUM_PINS; i++)
7          SetGPIOFunction(s_GPIO, Pins[i], INPUT, s_CLK);
8
9      /* Cleanup resources */
10     mem_cleanup ();
11
12     pr_alert("PWM Driver cleanup complete\n");
13 }
```

Listing 4.5: Device Driver Cleanup

```
1  static void mem_cleanup (void){
2      if (s_CLK) {
3          iounmap(s_CLK);
4          s_CLK = NULL;
5      }
6      if (s_PWM) {
7          iounmap(s_PWM);
8          s_PWM = NULL;
9      }
```

```
    if (s_GPIO) {
        iounmap(s_GPIO);
        s_GPIO = NULL;
    }

    if (g_cdev_added)
        cdev_del(&c_dev);
    g_cdev_added = false;

    if (g_devices_created > 0) {
        int i;
        for (i = 0; i < g_devices_created; i++)
            device_destroy(pwm_class, MKDEV(MAJOR(DEV_major_minor),
                i));
    }
    g_devices_created = 0;

    if (g_class_created)
        class_destroy(pwm_class);
    g_class_created = false;

    if (DEV_major_minor)
        unregister_chrdev_region(DEV_major_minor, NUM_PINS);
}
```

Listing 4.6: Device Driver Memory Clean

The defined `ioctl` operations are described in Listing 4.7, and can be called as per Listing 4.8. These are handled under the `pwm_device_ioctl` function, indicated in the file operations (Listing 4.3).

```
#define PWM_SET_CONFIG  _IOW(PWM_IOC_MAGIC, 1, ioctl_pwm_config_t)
#define PWM_DISABLE     _IO(PWM_IOC_MAGIC, 3)
```

Listing 4.7: `ioctl` PWM defined operations

```
ioctl (file_descriptor, PWM_SET_CONFIG, &conf);  //update value
ioctl (file_descriptor, PWM_DISABLE, &conf);
```

Listing 4.8: `ioctl` PWM defined operations user-space call

4.2.2 C++ Wrapper

The developed C++ Wrapper simply enables RAII mechanisms on POSIX threads, timers and IPC relevant mechanisms for the project. It also enables thread-safe mechanisms on the C++ `std::queue`. In Listing 4.9 it is possible to see the implementation of the class diagram in Figure 4.16, which was designed in the design phase.

```
1  namespace CppWrapper
2  {
3      class Thread
4      {
5          static int sched_policy;
6          pthread_t thread;
7          pthread_attr_t attr;
8          bool isDetachable;
9          void* userArg;
10
11     public:
12         bool isRunning;
13
14         void*(*entry_point)(void *);
15
16         explicit Thread(void*(*ep)(
               void *));
17         Thread(const Thread&)=delete;
18         Thread& operator=(const
               Thread&)=delete;
19
20         ~Thread();
21
22         int setPriority(int priority)
               ;
23         int setDetachAttribute();
24
25         static void* runFromInside(
               void* arg);
26         int run(void* arg = nullptr);
27         void join() const;
28         [[nodiscard]] int detach();
29     };
30 }
```

Listing 4.9: POSIX Thread C++ Wrapper implementation

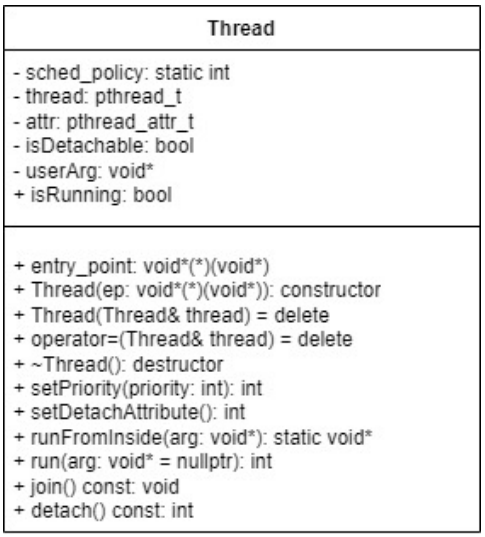


Figure 4.16: Class Diagram of POSIX Thread Wrapper

In Listing 4.10 the implementation of the thread-safe mechanisms, using a mutex, lock guard mechanism and a condition variable, on the C++ `std::queue` is depicted, alonside the respective class diagram (Figura 4.17).

```cpp
namespace CppWrapper
{
 // Queue allows to have non-trivially copiable data, contrary to MQueue
    template <typename T>
    class Queue
    {
        Mutex mutexQueue;
        CondVar condQueue;

        bool _interrupted;

        std::queue<T> queueData;
    public:
        Queue(): condQueue(mutexQueue), _interrupted(false){};
        Queue(const Queue& queue) = delete;
        Queue& operator= (Queue& queue) = delete;
        ~Queue();

        void send (T&& data)
        {/*...*/}

        T receive ()
        {/*...*/}

        void interrupt ()
        {/*...*/}
    };
}
```

Listing 4.10: POSIX Thread C++ Wrapper implementation

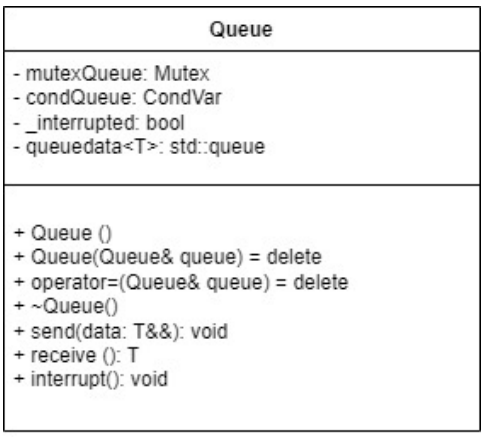


Figure 4.17: Class Diagram of C++ Queue Wrapper

4.2.3 Supabase Database Integration

Supabase is an open-source backend-as-a-service (BaaS) platform built on top of PostgreSQL. It provides a fully managed relational database together with additional services such as authentication, storage, and real-time data synchronization. In this project, Supabase is used as the central database system, responsible for storing configuration parameters, system metadata, and relational data shared between the control boxes, the RESTful API, and the graphical user interface (GUI).

One of the main advantages of using Supabase is its reliance on a standard PostgreSQL database, which allows the database schema to be defined using conventional SQL. This approach provides full control over the data model while maintaining compatibility with widely adopted relational database principles.

Table Creation and Schema Definition

The database tables used in the project were created directly in Supabase using SQL `CREATE TABLE` statements. Listing 4.11 presents an example of a table creation statement used to define the `Control Box` table. This table is responsible for storing identification and configuration information related to each intersection control box used in the system.

Each table uses a universally unique identifier (UUID) as its primary key. UUIDs are generated automatically using the `uuid_generate_v4()` function, ensuring global

uniqueness across distributed components. This design choice is particularly suitable for systems where multiple devices, such as Raspberry Pi–based control boxes, interact with a shared database.

```
CREATE TABLE IF NOT EXISTS ControlBox (
    ID UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    TMCID UUID NOT NULL,
    Password TEXT NOT NULL,
    Name TEXT,
    Location TEXT,
    SSID TEXT NOT NULL,
    SSIDPassword TEXT NOT NULL,
    FOREIGN KEY (TMCID) REFERENCES TMC(ID) ON DELETE CASCADE
);
```

Listing 4.11: Example of a table creation statement in Supabase using SQL

Several attributes stored in the `ControlBox` table include network credentials, location information, and authentication data. Mandatory fields are explicitly marked as `NOT NULL`, ensuring that incomplete or invalid records cannot be inserted into the database.

Figure 4.18 shows an overview of the tables created in Supabase to support interaction with both control boxes implemented in the project. Together, these tables form the foundation of the system's persistent data layer.

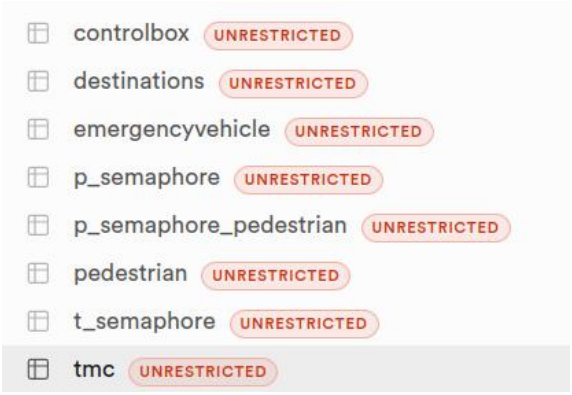


Figure 4.18: Tables created to interact with both Control boxes

To enforce data integrity and maintain consistency across the database, relationships between tables are defined using foreign key constraints. Foreign keys ensure that referenced records exist in their parent tables and prevent invalid associations from being created.

In the example shown in Listing 4.11, the `TMCID` attribute is defined as a foreign key that references the primary key of the `TMC` table. This relationship enforces a logical connection between control boxes and their corresponding traffic management centers.

Cascade delete actions (`ON DELETE CASCADE`) are used to automatically remove dependent records when a referenced parent record is deleted. This mechanism simplifies database maintenance and prevents the creation of orphaned records, which could otherwise lead to inconsistencies or unexpected system behavior. In a distributed system with multiple interacting components, such automated integrity management is essential.

An example of the contents and structure of one of the database tables is shown in Figure 4.19. This figure presents the `p_semaphore` table as displayed in the Supabase interface, including its attributes and stored entries.

id uuid	controlboxId uuid	name text	location int4	status PedestrianColor	gpio_red int4	gpio_gr... i...	hasCardReader int2
3fcb3faf-662f-45bf	cfc2d011-f8c2-43c9-...	PS1	1	GREEN	16	17	NULL
a626ee3b-5dc3-41	cfc2d011-f8c2-43c9-...	fsds	4	NULL	3	4	1
ba54d2f3-bfd3-495	cfc2d011-f8c2-43c9-...	PS2	3	GREEN	18	19	NULL

hasButt... i...	gpio_butt... i...	buttonThreshold int4	+
NULL	NULL	NULL	
1	3	23	
NULL	NULL	NULL	

Figure 4.19: Overview of the `p_semaphore` table, including its attributes and entries, in Supabase

The table structure clearly illustrates how individual records are organized and how each attribute contributes to representing the system state. By using a relational model, the database allows efficient querying, filtering, and updating of data, which is required for real-time interaction with both the GUI and the control boxes.

The complete database structure and the relationships between tables are summarized in the entity–relationship diagram (ERD) shown in Figure 4.20 created by Supabase, based on the tables created, which we can see that is the same ERD we made on the design phase.

Figure 4.20: Entity–relationship diagram (ERD) of the Supabase database

Within the overall system architecture, Supabase acts as a centralized data repository accessed by multiple components. The GUI interacts with Supabase in a bidirectional manner: configuration parameters can be created or modified through the GUI and stored in the database, while changes made directly in Supabase can be reflected back in the GUI interface. This bidirectional interaction allows flexible system configuration and monitoring.

In addition to the GUI, Supabase is connected to the RESTful API, which acts as an intermediary between the database and the Raspberry Pi–based control boxes. Through the API, both control boxes are able to retrieve configuration data from Supabase and transmit operational data back to the database. This architecture enables centralized data management while maintaining a clear separation between the user interface, backend logic,

and embedded hardware devices.

Further details regarding the GUI interaction and RESTful API implementation are presented in section 4.2.4 and 4.2.5 of this report.

4.2.4 RESTful API

To enable controlled and secure access to the cloud database from the embedded devices, a custom RESTful API was developed specifically to meet the needs of this project. Rather than exposing the Supabase database directly to the control boxes, the API acts as an intermediary layer between the Raspberry Pi–based control boxes and the Supabase backend. This design improves security, centralizes access logic, and ensures that only the required operations are available to the embedded devices.

The RESTful API is used by both control boxes to access cloud-stored data, retrieve configuration parameters, and update system state. It also serves as the integration point between the embedded layer and the database used by the GUI, enabling a coherent and synchronized system architecture.

API Initialization and Supabase Connection

The API is implemented using **Node.js** with the **Express** framework. Listing 4.12 shows the initialization of the Express server and the creation of the Supabase client. The Supabase client is configured using environment variables, including a service role key, allowing the API to perform privileged database operations on behalf of the control boxes.

```
const express = require('express');
const { createClient } = require('@supabase/supabase-js');
require('dotenv').config();

const app = express();
app.use(express.json());

// Initialize Supabase client
const supabase = createClient(
  process.env.SUPABASE_URL,
  process.env.SUPABASE_SERVICE_KEY
);
```

Listing 4.12: Express server initialization and Supabase client configuration for the RESTful API

By centralizing the Supabase connection within the API, the control boxes do not require direct database credentials, reducing exposure of sensitive keys and enforcing a clear separation of responsibilities.

API Connectivity

The control boxes communicate with the RESTful API over HTTP using standard networking libraries. Before issuing requests, connectivity to the API server is verified, as shown in Listing 4.13. This step ensures that the cloud services are reachable before attempting database interactions.

```
std::string url = "http://192.168.0.138:3000";
curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
curl_easy_setopt(curl, CURLOPT_NOBODY, 1L); // HEAD request
curl_easy_setopt(curl, CURLOPT_TIMEOUT, 5L);

const CURLcode res = curl_easy_perform(curl);
```

Listing 4.13: Connectivity check from the control box to the RESTful API server

Once connectivity is established, the control boxes use the API to retrieve identifiers, update semaphore states, and synchronize system data with the cloud database. These updates are subsequently propagated to the GUI through Supabase Realtime, completing the data flow between the embedded devices, the API, the database, and the user interface.

Implemented Endpoints and Design Constraints

As defined during the design phase, only the endpoints strictly required for system operation were implemented. The API supports the `GET`, `POST`, and `PATCH` HTTP methods. These methods are sufficient for retrieving data, creating new records, and updating existing entries. Notably, the `DELETE` method was intentionally not implemented. As a result, the control boxes do not have the ability to remove records from the database, preventing accidental or unauthorized data loss and preserving database integrity.

This restricted API design reflects the principle of least privilege, ensuring that embedded devices can only perform actions necessary for their operation.

Query implementation using GET Requests

The RESTful API implements a flexible data retrieval mechanism using `GET` requests. This mechanism allows the control boxes and other system components to query database tables dynamically, either retrieving full tables or filtered subsets of data, depending on the request parameters.

The implementation of this functionality is shown in Listing 4.14. Two routes are defined: one for retrieving all records from a given table and another for retrieving records filtered by a specific field–value pair. Both routes are handled by a single request handler, simplifying the API structure while maintaining flexibility.

```
module.exports = (supabase) => {
    const router = express.Router();

    router.get("/data/:table", handler);
    router.get("/data/:table/:field/:value", handler);

    return router;

    async function handler(req, res) {
        try {
            const { table, field, value } = req.params;

            if (table === "t_semaphore") {

                let query = supabase
                    .from("t_semaphore")
                    .select("*");

                if (field && value) {
                    query = query.eq(field, value);
                }

                const { data: tsemData } = await query;

                return res.status(200).json(/* result */);
            }

            if (table === "pedestrian" && field === "physicaltag_id
                ") {

                const { data } = await supabase
                    .from("pedestrian")
```

```
                    .select("cc_id")
                    .eq("physicaltag_id", value)
                    .limit(1);

                return res.status(200).json(/* { found: boolean } */);
            }

            let query = supabase
                .from(table)
                .select("*");

            if (field && value) {
                query = query.eq(field, value);
            }

            const { data } = await query;

            return res.status(200).json(data);

        } catch (err) {
            return res.status(500).json(/* erro */);
        }
    }
};
```

Listing 4.14: Generic GET endpoint for querying Supabase tables with optional filtering

The endpoint follows the structure `/data/:table` to retrieve all records from a given table, and `/data/:table/:field/:value` to retrieve only the records that match a specific condition. Internally, the handler dynamically constructs a Supabase query based on the parameters provided in the request.

Special handling is implemented for certain tables to meet project-specific requirements. For the `t_semaphore` table, the API allows retrieval of all traffic semaphore records or filtered queries based on a specified attribute. This functionality is primarily used by the monitoring components to obtain the current state of traffic semaphores.

For the `pedestrian` table, a dedicated query is implemented to check whether a pedestrian exists based on a physical tag identifier. In this case, only the corresponding control card identifier (`cc_id`) is retrieved. This targeted query minimizes data transfer and avoids exposing unnecessary database fields.

For all other tables, a generic query mechanism is used, allowing full table retrieval or filtered access using the specified field and value. This design provides a balance between flexibility and simplicity, while still enforcing controlled access to the database.

The following example illustrates how the GET endpoint is used by a control box to retrieve data from the RESTful API.

Example Request

```
json result_tsem = query_database("t_semaphore", "location","3");
```

Example Response

```
[
  {
    "controlboxid": "cfc2d011-f8c2-43c9-8ccb-6a30e0f78800",
    "destinations": [
        2,
        5
```

```
    ],
    "gpio_green": 5,
    "gpio_red": 6,
    "gpio_yellow": 7,
    "id": "cd8c3528-aa7e-4676-8e84-6c0a85cd7021",
    "location": 3,
    "name": "TS2",
    "status": null
    },
]
```

In this example, the control box requests all traffic semaphore records associated with a specific location. The API queries the Supabase database, retrieves the matching records, and returns them in JSON format. The returned data can then be used by the control box to update its internal state or trigger further actions.

This unified GET-based query mechanism simplifies communication between the embedded devices and the cloud database, while ensuring that all access is mediated through the RESTful API layer.

Test Cases

Test Case	Test Description	Expected Result	Real Result
Connection Test	Check connection to cloud	Can connect to cloud	Raspberry-Pi connected to cloud
Data Insertion	Check write to cloud	Can insert items in cloud	Items inserted in cloud
Data Retrieval	Check read from cloud	Can read items from cloud	Items read from cloud used to configure semaphores or verify data
Data Alteration	Check altered items in cloud	Can alter items in cloud	Items altered in cloud through patch and post

Table 4.1: Cloud-RESTful API test cases

4.2.5 GUI

As outlined in the system design phase, the GUI was developed to provide intuitive access to the system's functionality, including user authentication, monitoring, configuration of pedestrian and traffic semaphores, and management of control boxes. The interface is structured around several pages, namely a login page, a home page for monitoring, forms for registering pedestrian semaphores, and configuration pages for the traffic semaphores and control boxes (Figure 3.41). The overall structure of the GUI and its connection to the Supabase backend is shown in Listing 4.15, which illustrates the main components and the initialization of the Supabase client.

```
1  import React, { useState, useEffect } from 'react';
2  import {
3    Users, Key, Settings, Eye, EyeOff, Monitor, ArrowLeft,
4    Clock, Plus, Trash2, Ambulance, IdCard, Search,
5    MapPinned, SquareChartGantt, Pencil
6  } from 'lucide-react';
7  import { createClient } from '@supabase/supabase-js';
8
9  // Supabase configuration
10 const supabaseUrl =
11   import.meta.env.VITE_SUPABASE_URL;
12 const supabaseKey =
```

```
  import.meta.env.VITE_SUPABASE_ANON_KEY;

const supabase = createClient(supabaseUrl, supabaseKey);

// ============================
// COMPONENTS
// ============================
const LoginPage = ({ onLogin }) => { /* ... */ };
const Dashboard = ({ onLogout }) => { /* ... */ };
const HomePage = ({ setCurrentPage }) => { /* ... */ };
const SetupPage = () => { /* ... */ };
const ControlBoxForm = () => { /* ... */ };
const TrafficSemaphoreForm = ({ /* ... */ }) => { /* ... */ };
const PedestrianSemaphoreForm = ({ /* ... */ }) => { /* ... */ };
const MonitorPage = () => { /* ... */ };

// ============================
// APPLICATION ROOT
// ============================
export default function App() {
  const [user, setUser] = useState(null);

  return (
    <>
      {!user ? (
        <LoginPage onLogin={setUser} />
      ) : (
        <Dashboard onLogout={() => setUser(null)} />
      )}
    </>
  );
}
```

Listing 4.15: React-based GUI structure and Supabase client initialization, including publications and Realtime configuration

The frontend is implemented using **React**, leveraging **Vite** as a build tool for fast development and hot-reloading. The Supabase client is initialized with the project's URL and anonymous API key, enabling interaction with the PostgreSQL database. The GUI can be accessed through a standard web browser (http://172.20.10.2:5173/), providing an interface to visualize and update the state of semaphores and other system elements in real time (Figure 4.21).

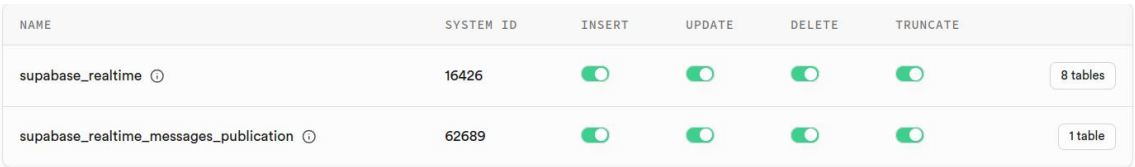


Figure 4.21: Supabase Realtime connection and GUI page view, illustrating the live synchronization with the database.

Realtime Updates and Database Synchronization

The GUI maintains a live connection to the Supabase database using `Realtime` channels. Instead of directly updating the database, the GUI listens for changes that occur in specific tables and updates its internal state accordingly. In this project, the `t_semaphore` table is monitored to reflect the current state of traffic semaphores.

Listing 4.16 shows the logic used to synchronize traffic semaphore data in real time. The

`updateTrafficSemaphore` function receives an updated semaphore record and updates the corresponding entry in the React state by iterating over the list of control boxes and their associated traffic semaphores. This approach ensures that only the modified semaphore is updated, while the remaining state is preserved.

```
const updateTrafficSemaphore = (updated) => {
    setControlBoxes((prev) =>
      prev.map((cb) => ({
        ...cb,
        t_semaphore: cb.t_semaphore?.map((ts) =>
          ts.id === updated.id ? { ...ts, ...updated } : ts
        ),
      }))
    );
  };

/*
    ********************************************************************
    */

    const channel = supabase
       .channel('monitor-realtime')
       .on('postgres_changes', { event: '*', schema: 'public', table
          : 't_semaphore' }, (payload) =>
         updateTrafficSemaphore(payload.new)
       )
       .subscribe();

     return () => supabase.removeChannel(channel);
  }, []);
```

Listing 4.16: Realtime synchronization of traffic semaphore data in the GUI using Supabase Realtime channels

A Supabase Realtime channel is created to subscribe to database changes using the `on('postgres_changes')` listener. The subscription listens to all events occurring on the `t_semaphore` table. Whenever a change is detected, the updated row data contained in the event payload is passed to the `updateTrafficSemaphore` function. As a result, any modification in the database is immediately reflected in the GUI without requiring a page refresh.

The source of these database changes is the Raspberry Pi–based intersection control box. When a traffic semaphore state is modified on the Raspberry Pi, a `patch` function is invoked to update the corresponding record in the Supabase database. This database update then triggers the Realtime event, which propagates the change to the GUI. An example of such a database update call executed on the Raspberry Pi is shown in Listing 4.17.

```
patch/_database(cloud/_, "t/_semaphore", "location", 3, "status", "
    GREEN");
```

Listing 4.17: Example of a database update issued by the Intersection Control Box (TMC), triggering a Realtime update in the GUI

When the GUI component is unmounted, the Realtime channel subscription is explicitly removed using `supabase.removeChannel`, preventing unnecessary network usage and potential memory leaks.

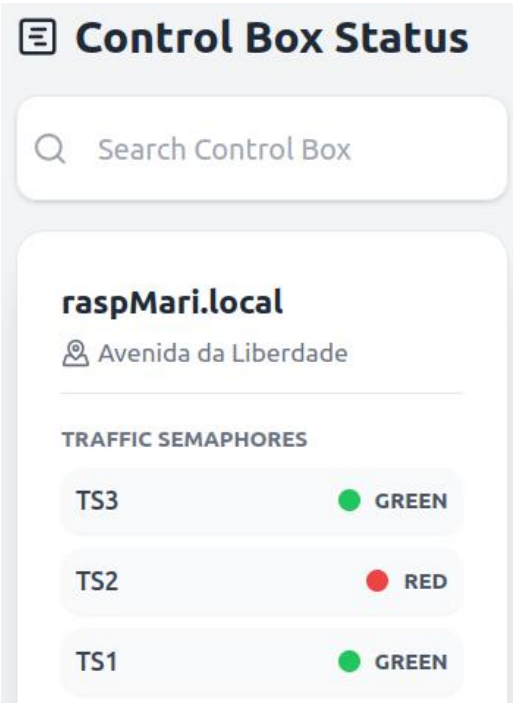
Figures 4.22(a) and 4.22(b) illustrate the traffic semaphore before and after an update triggered via the GUI. Similarly, Figures 4.22(c) and 4.22(d) show the corresponding changes in the monitoring page. These visualizations demonstrate the effectiveness of the Realtime synchronization mechanism, ensuring that updates are consistently reflected throughout the interface.

name text	status TrafficColor
TS1	GREEN
TS2	RED
TS3	GREEN

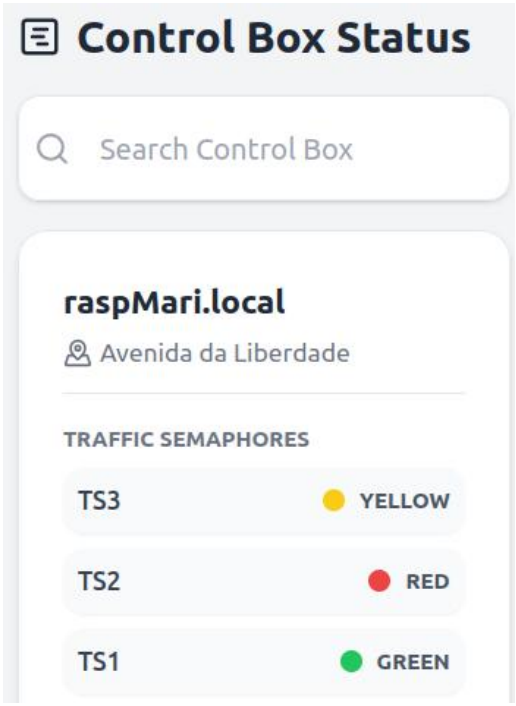
(a) Traffic semaphore before update

name text	status TrafficColor
TS1	GREEN
TS2	RED
TS3	YELLOW

(b) Traffic semaphore after update



(c) Monitor before update



(d) Monitor after update

Figure 4.22: Changes in traffic semaphores and monitoring interface before and after updates, illustrating the effect of GUI-driven and Realtime database synchronization.

Test Cases

Test Case	Test Description	Expected Result	Real Result
Connection Test	Check connection to cloud	Can connect to cloud	GUI connected to the cloud
Data Insertion	Check write to cloud	Can insert items in cloud	Items inserted in cloud
Data Retrieval	Check read from cloud	Can read items from cloud	Items read from cloud and displayed in GUI
Data Alteration	Check altered items in cloud	Can alter items in cloud	Items altered in cloud through GUI
Data Consistency	Verify that updates made locally are reflected in the cloud database and vice versa	Data remains consistent between local and cloud databases	Updates are real-time and consistent
Unauthorized access attempt detected	Try access cloud with an invalid login	Access not granted	Invalid username and/or password - access not granted

Table 4.2: Cloud-GUI test cases

4.3 Intersection Control Box

The Intersection Control Box is a system designed for efficient traffic management at intersections. It supports cloud communication, enabling *on-the-fly* updates of intersection

status as well as dynamic configuration and command execution based on cloud-defined policies. In addition, it provides DDS-based communication with emergency vehicles to promptly update and adapt the intersection state. The system also supports pedestrian interaction, receiving inputs through both push buttons and an RFID reader.

4.3.1 System Initialization

The main of this program simply contains the content of Listing 4.18.

```
1
2  int main()
3  {
4      try
5      {
6          TrafficControlSystem& tcs = TrafficControlSystem::
               getInstance();
7          tcs.start();
8          tcs.waitStop();
9      }catch (const std::runtime_error& e) {
10         std::cerr << "Runtime error: " << e.what() << '\n';
11     }
12     return 0;
13 }
```

Listing 4.18: ICB System initialization in main.cpp

An instance of `TrafficControlSystem` is retrieved via the `TrafficControlSystem::getInstance()` method, thus implementing the Meyers Singleton (Listing 4.19).

```
1
2  TrafficControlSystem& TrafficControlSystem::getInstance()
3  {
4      static TrafficControlSystem instance;
5      return instance;
6  }
```

Listing 4.19: `TrafficControlSystem::getInstance()` method, Meyers Singleton

The constructor of the `TrafficControlSystem` class initializes all required variables, as well as the Components Factory, Traffic Strategies, and system signal handlers to ensure proper shutdown. The initialization of traffic strategies follows the **Strategy** design pattern. The application of the **Factory** design pattern, however, does not strictly follow the textbook convention; instead, a variation of the Factory Method pattern is implemented using lambda functions to instantiate components.

```
1
2  TrafficControlSystem::TrafficControlSystem():
3      username("controlBox1.local"),
4      cloud("http://192.168.1.185:3000", username, "tmc1", this,
           _shutdown_requested),
5      tcsThread(t_tcs),
6      switchLightThread(t_switchLight),
7      ddsSubscriber(_shutdown_requested, 0, "EmergencyAlert", this)
8  {
9      state = SystemState::SET_UP;
10     current_config_idx = 0;
11     availableGPIOs = {1, 2, 3, 4, 5, 6, 7, 13, 14, 15, 16, 17, 18,
           19, 20, 21, 22, 23, 24, 25, 26, 27};//22 total
12
13     initComponentFactory();
14     initTrafficStrategies();
15     initSystemSignals();
```

```
}
```

Listing 4.20: Implementation of `TrafficControlSystem` constructor

Some variables in the `TrafficControlSystem` class require initialization beyond the constructor. To encapsulate these details and enforce a controlled startup sequence, a `start()` method is provided. This approach aligns with safe initialization practices and helps prevent errors or exceptions by ensuring that all dependent components are properly configured before the system becomes operational (this, thus, implements the Facade design pattern).

```

void TrafficControlSystem::start()
{
    // Start threads
    tcsThread.run(this);
    switchLightThread.run(this);

    cloud.cloudStart();
    ddsSubscriber.start();

    switch_state(state);
}
```

Listing 4.21: Implementation of `TrafficControlSystem start()` method

4.3.2 System Shutdown

Since the system has a `start()` method, it must also have a `stop()` method. The `waitStop()` function was implemented for that purpose (Listing 4.22).

```

void TrafficControlSystem::waitStop()
{
    switchLightQueue.interrupt();
    eventQueue.interrupt();

    ddsSubscriber.stop();
    cloud.stop();

    for (auto& psem: PedestrianSemVector)
        psem->stop();

    switchLightThread.join();
    tcsThread.join();
}
```

Listing 4.22: Implementation of `TrafficControlSystem waitStop()` method

4.3.3 Mediator and Components

In the Mediator and Component classes, which implement the Mediator design pattern, the methods are slightly adapated to the purpose of the code, as mentioned in the design phase. Listing ilustrates the implementation of this design pattern.

```

class Mediator;

class Component
{
```

```
protected:
    Mediator *mediator;
public:
    explicit Component(Mediator *mediator);
    virtual ~Component()=default;
    // void setMediator (Mediator* mediator); // Mediator is set on
        creation and never changed
};

class Mediator
{
public:
    virtual ~Mediator() = default;
    virtual void notify (Component* sender, Event command)=0;
    virtual int createComponents (const std::shared_ptr<json>&
       data_file)=0;
};
```

Listing 4.23: Implementation of `TrafficControlSystem waitStop()` method

All the system components inherit from the Component class like Listings 4.24, 4.25 and 4.26 indicate. Additionally, the TrafficControlSystem class inherits from the Mediator class, in order to implement its functionalities and to mediate the system's events the in EDA — Listing 4.27.

```
class PedestrianSemaphore :
    public Semaphore, public
    Component {
    /*...*/
};
```

Listing 4.24: `PedestrianSemaphore` component

```
class CloudInterface: public
    Component {
    /*...*/
};
```

Listing 4.25: `CloudInterface` component

```
class DDSSubscriber: public dds
    ::DataReaderListener, public
    Component {
    /*...*/
};
```

Listing 4.26: `DDSSubscriber` component

```
class TrafficControlSystem:
    public Mediator {
    /*...*/
};
```

Listing 4.27: `TrafficControlSystem` Mediator

The notification function is implemented as mentioned in Listing 4.28 . This method is used by all components when they want to notification the system about an event — `mediator->notify(this, event)`. Since it only sends an event through the `eventQueue`, then there must be a consumer function in a thread waiting to consume that data (implementation of the Producer-Consumer pattern) — Listings 4.29 and 4.30.

```

void TrafficControlSystem::notify (Component* sender, Event event)
{
    std::cout << "TrafficSystem this: " << this << std::endl;
    eventQueue.send(std::move(event));
}
```

Listing 4.28: Mediator Notification function

```
void TrafficControlSystem::consumer()
{
```

```
    Event data = eventQueue.receive();
    TrafficStrategy->controlOperation(this, data);
}
```

Listing 4.29: Consumer function

```
void* TrafficControlSystem::t_tcs(void* arg)
{
    const auto self = static_cast<TrafficControlSystem*>(arg);

    while (!_shutdown_requested.load())
    {
        self->consumer();
    }
    return arg;
}
```

Listing 4.30: Thread t_tcs

4.3.4 Strategies

The Strategy Design Pattern is used to confer the system different strategies to deal with the system events, based on the system's current state (Listing 4.31). Listing 4.34 depicts the strategy for system on emergency state. As observable, the strategy only cares about certain events, dismissing all the others, since they aren't relevant for that state.

```

void TrafficControlSystem::setStrategy()
{
    TrafficStrategy = TrafficStrategies[state].get();
}
```

Listing 4.31: Attribution of the strategy based on the current state

```

/* Emergency Strategy
 *  - Identify all the currently ON semaphores which interfere with
     the EV passage
 *  - Switch them off, and warn of the situation where it should be
     warned
 */
void StrategyEmergency::controlOperation(TrafficControlSystem* tcs,
    Event& event)
{

    auto visitor = [tcs](auto&& receive)
    {
        using T = std::decay_t<decltype(receive)>;

        if constexpr (std::is_same_v<T, InternalEvent>)
            handleInternalEvent(tcs, receive);
        else if constexpr (std::is_same_v<T, DDSEvent>)
            handleDDSEvent(tcs, receive);
    };

    std::visit(visitor, event);
}
```

Listing 4.32: Implementation of the Emergency Strategy

Here, as in other parts of the code, the `std::visitor` was applied. This acts as a variant of the textbook Visitor design pattern, and enables efficient handling when various types are being used and each requires a different approach.

4.3.5 Event Processing

In each strategy, events must be evaluated because the `Event` type is a variant capable of representing multiple subtypes. Efficiently distinguishing between different event variants within an operation state is therefore critical.

One common approach is to use `std::holds_alternative<T>()` in combination with `std::get<T>()`. This allows type-safe access to the underlying event without unnecessary copies or dynamic casts. Both operations execute in $O(1)$ time since they rely on the variant's internal type index for direct access. However, this approach requires explicitly checking and handling each variant type, which can become verbose if the number of variants is large.

An alternative, often more concise and equally efficient approach, is to use `std::visit` with a visitor. `std::visit` dispatches the correct callable for the currently held variant type in $O(1)$ ime, providing both type safety and automatic handling of all variant cases in a single, unified expression. This reduces boilerplate and makes the code more maintainable, especially when many variant types require distinct handling. While `std::visit` may introduce slight overhead due to the visitor invocation mechanism, in practice this cost is minimal, and the approach scales better for complex variants than multiple `std::holds_alternative/std::get` checks.

Therefore, `std::holds_alternative<T>()` and `std::get<T>()` are used when few events need to be treated, since it is visually simpler (Listing 4.33). When a state may handle many events, `std::visit` is used (Listing 4.34).

```
1  // Sets up the System: Based only on CloudReceiveType
2  void StrategySetUp::controlOperation (TrafficControlSystem* tcs,
       Event& event)
3  {
4      if (std::holds_alternative<InternalEvent>(event))
5      {
6          const auto& receive = std::get<InternalEvent>(event);
7          if (receive == InternalEvent::NEW_STATE_ENTERED)
8          {
9              tx_cloud::Configure configuration;
10             tcs->sendToCloud(configuration);
11         }
12     }
13
14     if (std::holds_alternative<CloudReceiveType>(event))
15     {
16         const auto& receive = std::get<CloudReceiveType>(event);
17
18         /*...*/
19     }
20         /*...*/
21 }
```

Listing 4.33: Strategy Set-Up — `std::holds_alternative<T>()` and `std::get<T>()`

4.3.6 Traffic Configuration Generation

Given the cloud configurations, the Intersection Control Box should be able of, on *boot*, get the data from the cloud an configure the system: the GPIO pins and arrange the configurations. As mentioned in the design phase, the used algorithm was the **Backtracking**

with prunning, considering a **Maximal Independent Set**, formulated under graph theory. Appendice A.2 contains the algorithm to perform these operations.

```
void TrafficControlSystem::findConfigurations()
{
    size_t totalSize = maxLocation + 1;
    conflictGraph.resize(totalSize);
    for (auto &row : conflictGraph)
        row.resize(totalSize, false);

    elementByLocation.resize(totalSize);

    setUpGraphMatrix();

    std::vector<int> current;

    for (auto& tsem : TrafficSemVector)
        vertices.push_back(tsem->getLocation());

    for (auto& cross : crosswalks) {
        vertices.push_back(cross->psem1->getLocation());
        vertices.push_back(cross->psem2->getLocation());
    }
    std::vector<int> candidates = vertices;

    std::sort(candidates.begin(), candidates.end());

    backtrack(current, candidates);
}
```

Listing 4.34: Implementation of the Emergency Strategy

The code which implements the trajectory collision which is used in the conflict graph matrix set up (Appendice A.1 — `conflictTrajectory`) is in Listing 4.35.

```
/* Considers a Circular Approach to return whether trajectories
   intersect or not
 * Args:
 *  a and b shall be Location and Destination of a single semaphore
 *  x is the location or destination of the other semaphore
 */
bool TrafficControlSystem::isBetween(const int a, const int b,
   const int x) {
    if (a < b)
        return x > a && x < b;
    return x > a || x < b;
}
```

Listing 4.35: Implementation of the Trajectory Collison Algorithm

4.3.7 Switch Light Thread

The overall software architecture of the threads is the same: they wait for events and produce events. Bellow, in Listing 4.36, is the implementation of the thread that commands the semaphores. It waits on the `switchLightQueue` to receive data and then uses that data to know which semaphores should be turned ON or OFF. Moreover, it notifies the system everytime a state change happens on the semaphores(i.e., when the timer which counts the time for that configurations timeouts).

```

void* TrafficControlSystem::t_switchLight(void* arg)
```

```
{
    auto self = static_cast<TrafficControlSystem*>(arg);

    SwitchLightsData&  switchingData = self->currentSwitchingData;
    while (!_shutdown_requested.load())
    {
        // Wait for switching data to be ready
        switchingData = self->switchLightQueue.receive();

        // Change Semaphores
        std::cerr << "PSEM OFF \n";
        self->stopPedestriansCross(switchingData.OFF_Crosswalk,
           false);

        std::cerr << "YELLOW \n";
        self->prepareToStopCars(switchingData.OFF_Tsem);

        self->timerSwitchLight.timerRun(YELLOW_DURATION);
        self->timerSwitchLight.timerWait();

        self->notify(nullptr, InternalEvent::YELLOW_TIMEOUT);

        self->stopCarsMove(switchingData.OFF_Tsem);

        self->letPedestriansCross(switchingData.ON_Crosswalk, false
           );
        self->letCarsMove(switchingData.ON_Tsem);

        std::cerr<<"GREEN: config "<< self->current_config_idx <<"
            \n";

        self->timerSwitchLight.timerRun(switchingData.time);
        self->timerSwitchLight.timerWait();

        // Notify the system itself
        self->notify(nullptr, InternalEvent::LIGHTS_TIMEOUT);
        std::cerr<<"RED\n";
    }
    return arg;
}
```

Listing 4.36: Switch Light Thread

4.3.8 Hardware Implementation

The images below illustrate the final implementation of a replica of the system design proposed in the previous sections.

Figure 4.23: Hardware deployment of an Intersection managed by a Intersection Control Box

4.3.9 Test Cases

Test Case	Test Description	Expected Result	Real Result
Login in cloud	Verify authentication with valid credentials	System returns success, user authenticated and connected to Cloud	Success
Button Responsiveness	Test if buttons respond to user input	Press event detected (pressCount incremented) and sent to system	Success
Semaphore State Change	Change semaphore state from one colour to another	Traffic/Pedestrian Light changes to another colour, duration respected, state updated	Success
RFID Card Reading	Bring valid RFID card close to reader	RFID_Reader detects card, reads CardID and returns data correctly	Success
Buzzer Activation	Pedestrian semaphore turns GREEN	BuzzerDriver activates buzzer, audible sound emitted (with a specified frequency rate, ioctl commands executed successfully)	Success
Traffic/Pedestrian Light Duration	Configure specific duration for green light (e.g., 30s)	Semaphore maintains GREEN for exactly 30 seconds before changing	Success
Invalid Login Attempt	Attempt login with invalid credentials	System rejects authentication, returns error, access denied	Success
DDS Responsiveness	Receive message from DDS communication	Receive message with emergency vehicle ID and direction	Success
Emergency Mode Activation	Activate System Mode EMERGENCY_ACTIVE	Semaphores change to emergency state, normal operation suspended	Success
DDS Subscriber creation	Access to DDS topic	Receive message from correct topic	Success
Extended Passage	Person with disability passed card on RFID	Extend time on that configuration, on the next time it happens	Success
Early Passage	Count of press buttons greater than threshold in a few seconds	Pedestrian light turns GREEN before its time	Success
Pin Association through cloud	Cloud attributes a pin for semaphore light	Pin attributed is available for the effect and is configured with the right settings	Success
Invalid RFID cardID	Person without disability scans card on RFID module	GREEN light is not extended	Success
Configuration Set Up on boot	Get data from cloud and form the correct configurations	Data is retrieved and configurations are performed effectively	Success

Table 4.3: Intersection Control Box test cases

4.4 Emergency Vehicle Control Box

The Emergency Vehicle Control Box is an embedded system designed for emergency vehicles that require priority passage through traffic intersections. When in an emergency state, the system broadcasts warning messages (publishes) to semaphores using Fast DDS, enabling real-time communication with intersection control boxes. The system establishes communication by connecting to access points provided by intersection control boxes and transmits critical information including the vehicle's license plate, origin, destination and priority of the emergency.

The control box integrates several critical subsystems: a battery monitoring system that continuously reads voltage levels through an ADC, a visual warning system implemented via a Linux device driver that controls an LED to blink when battery levels are low, and a cloud interface that manages WiFi network configurations to ensure connectivity with intersection infrastructure. This multi-threaded, real-time system employs synchronization primitives and design patterns to ensure reliable operation under critical conditions.

4.4.1 System Initialization

The system initialization follows a structured approach that instantiates all necessary components and establishes their dependencies. The initialization process is orchestrated through a singleton pattern implementation in the `evcontrolsystem` class, ensuring a single point of control for the entire system.

As shown in Listing 4.37, the main function obtains the singleton instance and configures the system with the cloud URL and Traffic Management Center (TMC) identifier. The initialization parameters include the cloud server endpoint and the specific TMC name that identifies which intersection control boxes the vehicle can communicate with.

```
evcontrolsystem *evSystem = &evcontrolsystem::getInstance();
std::string cloud_url = "http://172.26.200.215:3000";
std::string tmcName = "tmc1";
evSystem->initializeSystem(cloud_url, tmcName);
```

Listing 4.37: System initialization in main.cpp

The `initializeSystem` method, presented in Listing 4.38, performs several critical operations. First, it instantiates the battery monitoring subsystem, the Fast DDS publisher for emergency messages, and the cloud interface. The method then establishes a connection to the cloud server and retrieves the WiFi network credentials (SSIDs) associated with the specified TMC. Finally, it registers signal handlers for graceful shutdown on SIGINT, SIGTERM, and SIGHUP signals. This initialization sequence ensures that all components are properly configured before the system begins operation.

```
bool evcontrolsystem::initializeSystem(std::string& cloud_url,
    const std::string& tmcName)
{
    if (initialized_) return true;

    cloud_ = CloudInterface(cloud_url);
    cloud_.cloudConnect();

    battery_ = new Battery(shutdown_requested_);
    publisher_ = new eprosima::fastdds::examples::emergencyMSG::
        DDSPublisher(shutdown_requested_, 1, 1, "EmergencyAlert");

    if (!receive_SSIDs(tmcName))
        return false;

    signal(SIGINT, evcontrolsystem::signalHandler);
    signal(SIGTERM, evcontrolsystem::signalHandler);
```

```
16      signal(SIGHUP, evcontrolsystem::signalHandler);
17
18      initialized_ = true;
19      return true;
20  }
```

Listing 4.38: System initialization method in evcontrolsystem.cpp

4.4.2 Design Pattern Implementation

The system architecture employs two fundamental design patterns: Singleton and Facade. These patterns provide structure, maintainability, and controlled access to system resources.

Singleton Pattern

The Singleton pattern ensures that only one instance of the `evcontrolsystem` class exists throughout the application lifecycle. This is critical for maintaining consistent system state and preventing resource conflicts. As demonstrated in Listing 4.39, the implementation uses a static instance within the `getInstance` method, which is thread-safe in C++11 and later due to guaranteed initialization of static local variables. The private constructor prevents direct instantiation, enforcing access through the singleton interface. The manner in which this singleton instance is accessed and utilized within the system is described in Section 4.4.1.

```
1   class evcontrolsystem {
2   private:
3    // Private constructor
4       evcontrolsystem();
5
6   public:
7       static evcontrolsystem& getInstance
            ()
8       {
9           static evcontrolsystem instance;
10          return instance;
11      }
12
13      ~evcontrolsystem();
14  };
```

Listing 4.39: Singleton pattern implementation

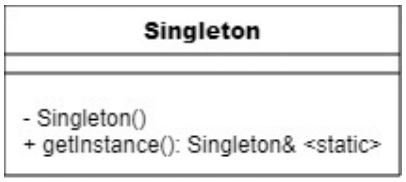


Figure 4.24: Default Singleton implementation

Facade Pattern

The Facade pattern is implemented through the `evcontrolsystem` class, which provides a simplified interface to the complex subsystems of battery monitoring, DDS publishing, and cloud communication. As shown in Listing 4.40, the facade exposes high-level methods such as `initializeSystem`, `runSystem`, and `shutdownSystem`, hiding the intricate details of component initialization, thread management, and inter-component communication. This abstraction allows clients to interact with the entire system through a cohesive interface without needing to understand the implementation details of each subsystem.

```
1   class evcontrolsystem {
2   private:
3       Battery* battery_;
4       eprosima::fastdds::examples::emergencyMSG::DDSPublisher*
            publisher_;
5       CloudInterface cloud_;
```

```
6       bool initialized_;
7
8   public:
9       bool initializeSystem(std::string& cloud_url, const std::string
            & tmcName);
10      void runSystem() const;
11      void shutdownSystem();
12  };
13
14  void evcontrolsystem::runSystem() const
15  {
16      if (!initialized_) return;
17      battery_->start();
18      publisher_->start();
19  }
```

Listing 4.40: Facade pattern providing simplified system interface

4.4.3 Thread Coordination and Execution Flow

The system employs multiple threads to handle concurrent operations (has show in the design phase in section 3.9.5), with particular emphasis on real-time battery monitoring and LED warning management. The battery monitoring subsystem creates two primary threads: `batteryMonitorThread` for voltage measurements and `ledWarningThread` for visual warnings, as shown in Listing 4.41.

```
1   void Battery::start()
2   {
3       signal(SIGALRM, Battery::sigalrmHandler);
4
5       batteryMonitorThread = std::make_unique<CppWrapper::Thread>(
            t_batteryMonitor);
6       batteryMonitorThread->setPriority(60);
7       batteryMonitorThread->run(this);
8
9       ledWarningThread = std::make_unique<CppWrapper::Thread>(
            t_ledWarning);
10      ledWarningThread->setPriority(30);
11      ledWarningThread->run(this);
12
13      itimerval timer{};
14      timer.it_value.tv_sec = time_between_adcREAD;
15      timer.it_interval.tv_sec = time_between_adcREAD;
16      setitimer(ITIMER_REAL, &timer, nullptr);
17  }
```

Listing 4.41: Thread creation and priority assignment in Battery.cpp

The battery monitoring thread demonstrates sophisticated coordination using POSIX mutexes, and condition variables wrapped in the custom CppWrapper library. The `monitorBattery` method, presented in Listing 4.42, implements a producer pattern where voltage readings are triggered periodically by SIGALRM signals. The thread waits on a condition variable until the `alarmTriggered` flag is set by the signal handler, ensuring synchronization between the timer interrupt and the monitoring loop.

```cpp
void Battery::monitorBattery()
{
    while (!shutdown_requested_.load())
    {
        condMutex->LockMutex();
        while (!alarmTriggered)
            condReadADC->condWait();

        alarmTriggered = false;
        condMutex->UnlockMutex();

        float voltage;
        adcMutex->LockMutex();
        voltage = adc->readBatteryVoltage();
        adcMutex->UnlockMutex();

        std::cout << "[Battery] Current Battery Voltage: "
                  << voltage << " V" << std::endl;

        voltageQueue->send(voltage);
    }
}
```

Listing 4.42: Battery monitoring thread with condition variable synchronization

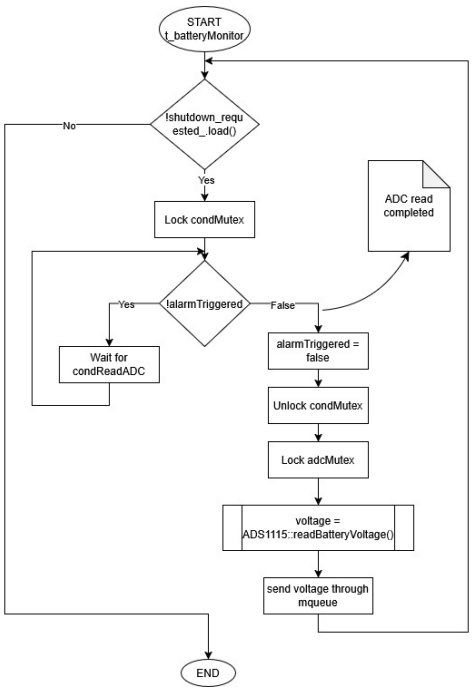


Figure 4.25: Battery monitoring task design previously

The signal handler mechanism, shown in Listing 4.43, provides the timing source for periodic battery measurements. When SIGALRM is raised by the interval timer (configured for 120-second intervals), the handler safely updates the shared `alarmTriggered` flag and broadcasts to the condition variable, waking the monitoring thread. This design ensures thread-safe communication between the asynchronous signal context and the monitoring thread.

```cpp
void Battery::sigalrmHandler(int)
{
    if (!instance) return;

    instance->condMutex->LockMutex();
    instance->alarmTriggered = true;
    instance->condReadADC->condBroadcast();
    instance->condMutex->UnlockMutex();
}
```

Listing 4.43: SIGALRM handler for periodic battery monitoring

Inter-thread communication is facilitated by a POSIX message queue (wrapped in `CppWrapper::MQueue`), which implements a producer-consumer pattern. The monitoring thread produces voltage readings and sends them to the queue, while the LED warning thread consumes these readings to determine the appropriate warning state. This decoupling allows each thread to operate at its own pace while maintaining data consistency.

4.4.4 Cloud and API Communication

The cloud interface provides critical functionality for retrieving WiFi network credentials from the database and configuring the system to connect to intersection control box access

points. The process begins with querying the cloud API to obtain SSIDs associated with a specific Traffic Management Center.

The `getWifiNetworksByTMC` method, shown in Listing 4.44, performs a two-stage API query. First, it retrieves the TMC ID by name, then uses this ID to query all control boxes associated with that TMC. The method parses the JSON response to extract SSID and password pairs, validating each entry before adding it to the network list.

```
std::vector<WifiNetwork> CloudInterface::getWifiNetworksByTMC(const
    std::string &tmcName) const
{
    std::vector<WifiNetwork> nets;

    std::string tmcid = getTableID("tmc", tmcName);
    if (tmcid.empty())
        return nets;

    nlohmann::json result = querySSID("controlbox", "tmcid", tmcid)
        ;
    if (!result.is_array())
        return nets;

    for (const auto& item : result) {
        if (!item.is_object()) continue;
        if (!item.contains("ssid") || !item.contains("ssid_password
            ")) continue;

        std::string ssidStr = item.at("ssid").get<std::string>();
        std::string passwordStr = item.at("ssid_password").get<std
            ::string>();
        if (!ssidStr.empty())
            nets.push_back({ssidStr, passwordStr});

    }
    return nets;
}
```

Listing 4.44: Retrieving WiFi networks from cloud API

Once network credentials are retrieved, the system updates the WPA supplicant configuration file. The `addNetwork` method, presented in Listing 4.45, appends each network configuration to the `/etc/wpa_supplicant.conf` file in the standard WPA-PSK format. This enables the emergency vehicle to automatically connect to intersection control box access points when in range.

```
void CloudInterface::addNetwork(const WifiNetwork& net) const
{
    if (net.ssid.empty() || net.password.empty())
        return;

    std::ofstream outFile(wpaConfigFile_, std::ios::app);
    if (!outFile.is_open())
        throw std::runtime_error("CloudInterface::addNetwork: "
                                 "failed to open wifi configuration
                                    file");

    outFile << "\nnetwork={\n";
    outFile << "  ssid=\"" << net.ssid << "\"\n";
    outFile << "  psk=\"" << net.password << "\"\n";
    outFile << "  key_mgmt=WPA-PSK\n";
    outFile << "}\n";
}
```

Listing 4.45: Adding WiFi networks to WPA supplicant configuration

After updating the configuration file, the system reinitializes the WPA supplicant to apply the new network settings, as shown in Listing 4.46. This ensures immediate connectivity to the newly configured access points without requiring a system reboot. Now the emergency vehicle is ready to send DDS messages.

```
void CloudInterface::reinitializeWifiNetworks() const
{
    int ret = system("wpa_cli -i wlan0 reconfigure");
    if (ret != 0)
        std::cerr << "Warning: wpa_cli reconfigure failed\n";
}
```

Listing 4.46: Reinitializing WPA supplicant with new network configurations

4.4.5 System Shutdown

The system implements a graceful shutdown mechanism that ensures proper cleanup of resources, thread termination, and device driver removal. The shutdown process is triggered by Linux signals (SIGINT, SIGTERM, SIGHUP) which set a global atomic flag monitored by all active threads.

The signal handler, shown in Listing 4.47, sets the `shutdown_requested_` atomic boolean, which is checked by all running threads in their execution loops. This non-blocking approach allows each thread to complete its current operation before termination.

```
void evcontrolsystem::signalHandler(const int signum)
{
    shutdown_requested_.store(true);
}
```

Listing 4.47: Signal handler initiating system shutdown

The main loop waits for the shutdown flag before invoking the cleanup sequence, as demonstrated in Listing 4.48. This ensures all subsystems receive the shutdown signal and have time to terminate gracefully.

```
while (!evcontrolsystem::shutdown_requested_.load()) {}
evSystem->shutdownSystem();
```

Listing 4.48: Main loop shutdown sequence

The `shutdownSystem` method, presented in Listing 4.49, is responsible for explicitly destroying all major subsystem components. Each subsystem is deallocated using the `delete` operator, followed by setting the corresponding pointer to `nullptr` to prevent dangling references. This approach ensures that the destructors of each component are invoked and that their respective cleanup procedures are properly executed.

```
void evcontrolsystem::shutdownSystem()
{
    delete battery_;
    battery_ = nullptr;

    delete publisher_;
    publisher_ = nullptr;
}
```

Listing 4.49: System shutdown destroying all subsystems

The DDS publisher follows the same structured shutdown pattern used throughout the system. As shown in Listing 4.50, the destructor implements a clean and deterministic termination sequence. The publisher first wakes any threads blocked on condition variables by broadcasting on the associated condition variable, ensuring no thread remains suspended. It then joins the DDS worker thread to guarantee orderly completion before proceeding with resource deallocation. Finally, all DDS entities owned by the publisher are explicitly deleted, and the domain participant is released via the DDS factory. This example illustrates the general approach adopted across subsystems to ensure safe thread termination and proper resource cleanup.

```
void DDSPublisher::stop() const
{
    stateMutex->LockMutex();
    stateCV->condBroadcast();
    stateMutex->UnlockMutex();

    ddsThread->join();
}

DDSPublisher::~DDSPublisher()
{
    stop();

    if (participant_) {
        participant_->delete_contained_entities();
        dds::DomainParticipantFactory::get_instance()
            ->delete_participant(participant_);
    }
}
```

Listing 4.50: DDS publisher shutdown and resource cleanup

The PWM device driver shutdown, illustrated in Listing 4.51, demonstrates kernel module cleanup. The destructor closes the device file descriptor, removes the kernel module from the kernel space, and ensures the LED is disabled before driver removal.

```
PWM_DeviceDriver::~PWM_DeviceDriver()
{
    close_dd();
    while (!removeKernelModule()){};
    file_descriptor = -1;
}
```

Listing 4.51: PWM device driver cleanup and kernel module removal

This coordinated shutdown sequence ensures that no resources are leaked, all threads terminate cleanly, kernel modules are properly unloaded, and the system returns to a clean state suitable for restart or power-down.

4.4.6 Final result

This section presents the final integrated behavior of the emergency vehicle control box, illustrating how all system components operate together in a real execution scenario. On the left terminal, the emergency vehicle control system is running. The output shows that the system is successfully initialized, including cloud connectivity and DDS publisher creation. While waiting for a subscriber to match with the publisher, the battery monitoring task runs continuously in the background, periodically displaying voltage measurements on the terminal.

On the right terminal, a separate process acts as a DDS subscriber. Once the subscriber successfully matches with the publisher, the emergency control box transmits a warning

message containing the information required by the intersection control box to change its operational state. This message is correctly received and processed by the subscriber. The subscriber remains active until the emergency vehicle control system terminates.

Upon system shutdown, all threads and resources are cleanly terminated as described in previous sections. Finally, the wireless network configuration can be verified, confirming that the WPA configuration contains the SSID retrieved from the cloud service, demonstrating successful end-to-end system integration.

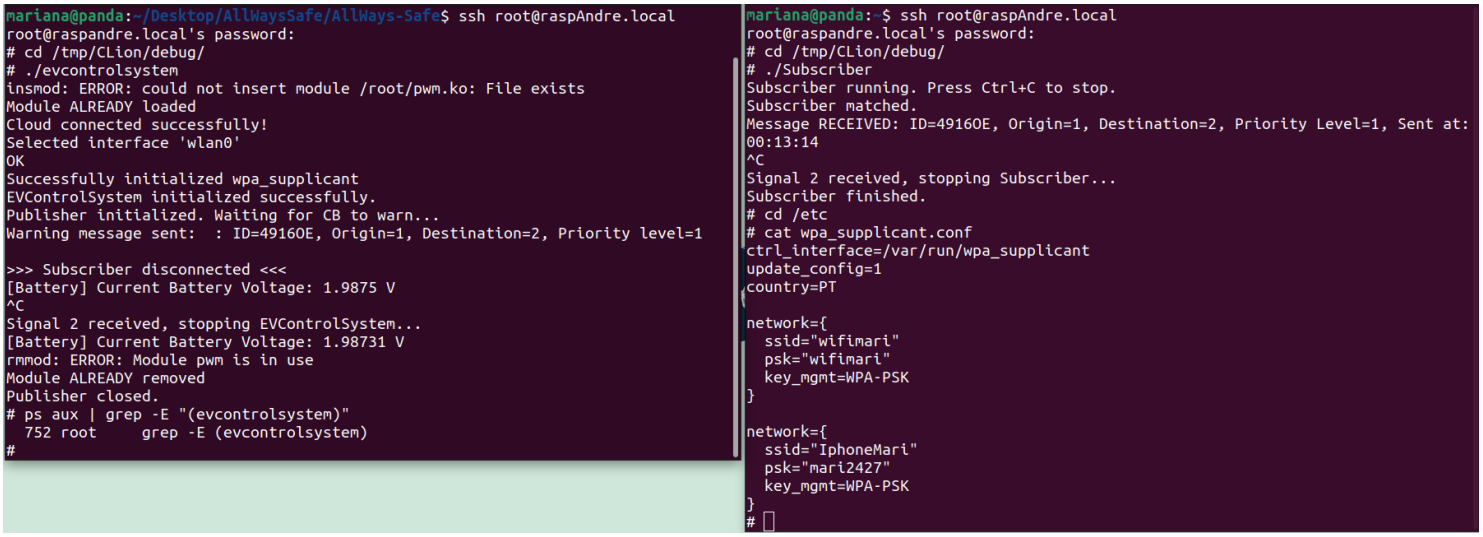
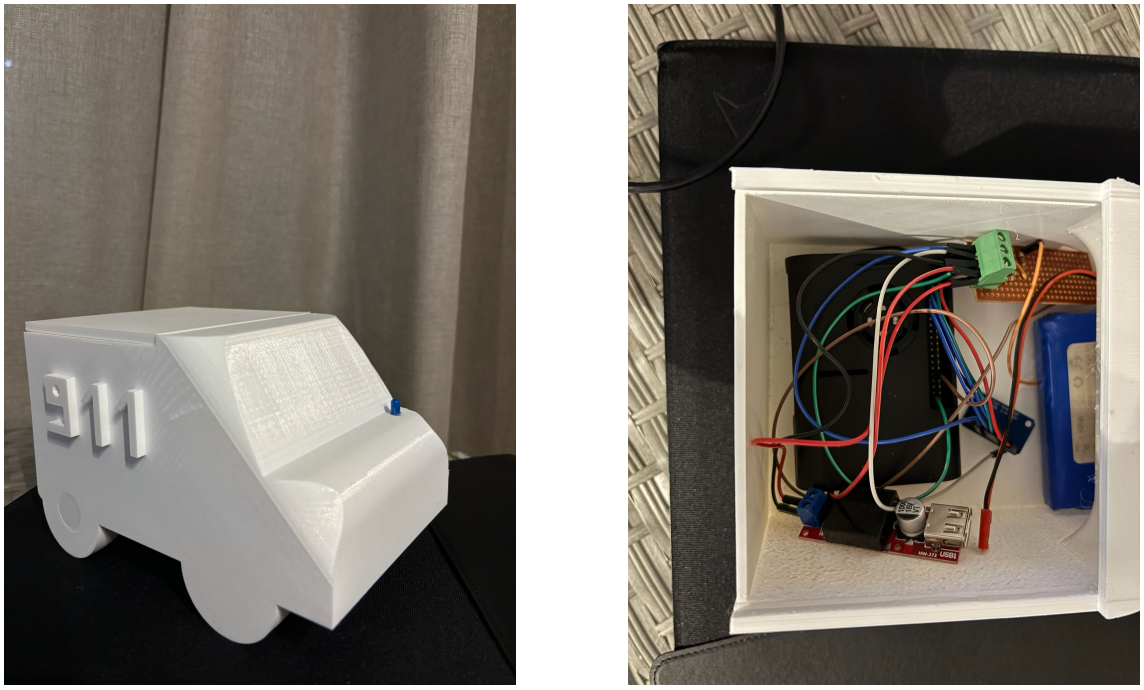


Figure 4.26: Runtime execution of the emergency vehicle control system showing successful initialization, battery monitoring, DDS publisher–subscriber matching, warning message transmission, and clean system shutdown



(a) Emergency vehicle

(b) Hardware inside the emergency vehicle

Figure 4.27: Emergency vehicle prototype and internal hardware configuration

4.4.7 Test Cases

Test Case	Test Description	Expected Result	Real Result
Login in cloud	Verify authentication with valid credentials	System returns success, user authenticated and connected to Cloud	Success message sent from cloud and controlbox logged in
Logout off cloud	Disconnect user from cloud system	Session terminated successfully, Cloud disconnected	Control box logged out of cloud
ADS1115 Warning Activation	Battery voltage drops below threshold	Blinking LED turns on	Led blinked
ADS1115 Warning Deactivation	Battery voltage returns to normal levels	Blinking LED turns off	Led is off
ADS1115 Voltage Reading	Read battery voltage from ADS1115 sensor	Correct voltage reading	Battery voltage read correctly
Network Connection to Control Box AP check	Verify network connectivity status	Emergency Vehicle has IP address	Emergency vehicle connected to access point from intersection control box
DDS Message Publishing	Publish MessageData through DDSPublisher	Message published to topic successfully, timestamp and senderID correct	Emergency message sent to intersection control box
LED Blink Control	Control LED blinking through LedBlinkDriver	LED blinks at specified rate, ioctl commands executed successfully	Led blinked at specified rate

Table 4.4: Emergency Vehicle test cases

Conclusion

Overall, this project was successfully completed in accordance with the objectives initially defined. It involved multiple and distinct areas of knowledge, demonstrating that their integration is not only feasible but also capable of producing positive and meaningful results.

Furthermore, we were able to implement emerging technologies such as Vehicle-to-Infrastructure (V2I) and Infrastructure-to-Pedestrian (I2P), which are becoming increasingly relevant in the technological and automotive domains due to continuous advancements in intelligent transportation systems.

By building upon the concepts learned throughout the course, it was possible to deepen our understanding and expand into additional areas of knowledge. This process enabled the acquisition of skills and competencies beyond those originally required, contributing significantly to our technical and analytical development.

From the implementation of a middleware layer designed to improve and optimize communication efficiency, to the development of a scalable algorithm capable of controlling any type of traffic intersection, as well as the creation of a monitoring interface, it was possible to build a system that is robust, extensible, and, above all, fully functional.

Appendix A

Appendix

A.1 Conflict Graph Set Up

The conflict graph aims to store the undirected relations between semaphores. It represents the if conflicts exist (1) or not (0). In theory, the conflict graph is typically constructed using consecutive indices. However, given that intersections rarely involve hundreds or thousands of semaphores, constructing the conflict graph based on the actual intersection locations of existing semaphores—despite these indices being non-consecutive—does not compromise the correctness or performance of the system.

```
1
2  /*  Matrix indexes are identified by the Location attribute (
       Location are contiguous)
3   *       - Item in conflictGraph[i][j] is
4   *           - true: if there is conflict
5   *           - false: if there isn't conflict
6   */
7  void TrafficControlSystem::setUpGraphMatrix(){ // O(T  ) + O(T*P)
8      // Traffic Semaphores
9      for (size_t i = 0; i < TrafficSemVector.size(); ++i)
10     {
11         for (size_t j = i + 1; j < TrafficSemVector.size(); ++j) {
12             if (conflictTrajectory(*TrafficSemVector[i], *
                   TrafficSemVector[j])) {
13                 conflictGraph[TrafficSemVector[i]->getLocation()][
                       TrafficSemVector[j]->getLocation()]
14                 = conflictGraph[TrafficSemVector[j]->getLocation()
                       ][TrafficSemVector[i]->getLocation()]
15                             = true; // Undirected Conflict Graph
16             }
17         }
18         elementByLocation[TrafficSemVector[i]->getLocation()] =
               TrafficSemVector[i].get();
19     }
20     // Crosswalks/ Pedestrian Semaphores
21     for (auto & crosswalk : crosswalks)
22     {
23         for (auto & j : TrafficSemVector)
24         {
25             if (conflictTrajectory(*j, *crosswalk))
26             {
27                 conflictGraph[j->getLocation()][crosswalk->psem1->
                       getLocation()]
28                 = conflictGraph[j->getLocation()][crosswalk->psem2
                       ->getLocation()]
29                 = conflictGraph[crosswalk->psem1->getLocation()][j
```

```
                    ->getLocation()]
30                  = conflictGraph[crosswalk->psem2->getLocation()][j
                    ->getLocation()]
31                  = true; // Undirected Conflict Graph
32              }
33          }
34          elementByLocation[crosswalk->psem1->getLocation()] =
                crosswalk.get();
35          elementByLocation[crosswalk->psem2->getLocation()] =
                crosswalk.get();
36      }
37  }
```

Listing A.1: Conflict Graph Set Up Method

A.2 Backtracking with prunning Algorithm implementation

```
1   // Apply backtracking w/ pruning Algorithm
2   /*      vector current holds the current locations to insert on the
        same configuration
3    *      vector candidates holds all the locations, initially
4    */
5   void TrafficControlSystem::backtrack(std::vector<int>& current, std
        ::vector<int>& candidates)
6   {
7       if (candidates.empty()) {       // Last iteration of each
            configuration
8           if (!current.empty()) {
9               Configuration cfg;
10              for (const int loc : current)
11              {
12                  const IntersectionElement& c = elementByLocation[
                        loc];
13
14                  std::visit([&](auto&& obj) {
15                      using T = std::decay_t<decltype(obj)>;
16
17                      if constexpr (std::is_same_v<T,
                            TrafficSemaphore*>) {
18                          cfg.activeTsem.push_back(obj);
19                      }
20                      else if constexpr (std::is_same_v<T, Crosswalk
                            *>) {
21                          if (std::find(cfg.crosswalk.begin(),
22                                        cfg.crosswalk.end(),
23                                        obj) == cfg.crosswalk.end())
24                              cfg.crosswalk.push_back(obj);
25                      }
26                  }, c);
27              }
28
29              // Check if it is a maximal independent set - local
                    validation:
30              //      if any of the other locations (vertices) are
                    compatible with this config
31              bool maximal = true;
32              for (int v : vertices) {
33                  if (std::find(current.begin(), current.end(), v) ==
                        current.end()) {
```

```
                    bool ok = true;
                    for (int u : current)
                        if (conflictGraph[v][u]) {
                            ok = false;
                            break;
                        }
                    if (ok) {
                        maximal = false;
                        break;
                    }
                }
            }

            if (maximal)
                configurations.push_back(cfg);
        }
        return;
    }

    while (!candidates.empty()) {
        int v = candidates.back();
        candidates.pop_back();

        // New subset of compatible candidates - next recursive
           iteration
        std::vector<int> newCandidates;
        for (int u : candidates) {
            if (!conflictGraph[v][u])           // pruning
                newCandidates.push_back(u);
        }

        current.push_back(v);
        backtrack(current, newCandidates);
        current.pop_back();
    }
}
```

Listing A.2: Backtracking wiith Prunning Algorithm

Bibliography

1. https://edition.cnn.com/2019/08/22/us/traffic-commute-gridlock-transportation-study-trnd
2. https://www.post-gazette.com/news/politics-local/2024/09/05/red-light-cameras-pittsburgh/stories/202409040109
3. https://www.precedenceresearch.com/intelligent-traffic-management-system-market
4. https://www.ti.com/lit/an/sbaa565/sbaa565.pdf?ts=1761421491223
5. https://www.handsontec.com/dataspecs/RC522.pdf
6. https://erhanbakirhan.medium.com/introduction-to-dds-data-distribution-service-real-time-data-communication-made-easy-d6f4badddd6f
7. https://www.visual-paradigm.com/guide/data-modeling/what-is-entity-relationship-diagram/
8. https://medium.com/@vikkasjindal/the-best-way-to-return-responses-in-rest-apis-f248113e385e
9. https://restfulapi.net/resource-naming/
10. https://www.botnroll.com/pt/