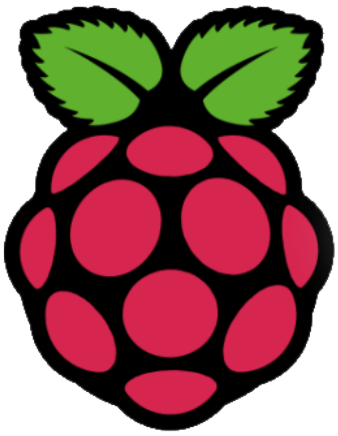
Hugo Ribeiro PG47241 Ricardo Mendes PG47612



Gate

Ricardo Roriz
Sérgio Pereira
Tiago Gomes

Embeded Systems master's course
February, 2022

Contents

List of Figures

List of Tables

List of Acronyms

AI Artificial Intelligence

ACK Acknowledge

API Application Programming Interface

BJT Bipolar Transistor

CAGR Compound Annual Growth Rate

CP Ceiling Priority

CPS Cyber-physical system

CSI Camera Serial Interface

CSS Cascading Style Sheets

ERD Entity-Relationship Diagram

FCS Frame Check Sequence

FIFO First In, First Out

GND Ground

GPIO General Purpose Input/Output

GUI Graphic User Interface

HDMI High-Definition Multimedia Interface

HTML HyperText Markup Language

I2C Inter-IC Communication

IC Integrated Circuit

IPC Inter-process communication

JS Java Script

LED Light-Emitting Diode

MCU Microcontroller Unit

MIPI Mobile Industry Processor Interface

NF Normal Form

OS Operating System

PDB Power distribution board

PDF Portable Document Format

PWM Pulse Width Modulation

SCL Serial Clock

SDA Serial Data

SDK software Development Kit

TCP-IP Transmission Control Protocol - Internet Protocol

TTL Transistor–transistor logic

UART Universal Asynchronous Receiver-Transmitter

UI User Interface

USB Universal Serial Bus

Wi-Fi Wireless Fidelity

OCR Optical character recognition

A Indrodution

Within the scope of embedded systems master's course was proposed to do a project which will apply all the taught subjects, like threads, TCP-IP, daemons, etc. The project started at the beginning of the semester and the deadline is January 13th.

This project must be useful, in other words, it must be developed to solve a real problem. The main idea is to picture a scenario when some real companies like Samsung, LG, Bosh, etc. come and say that they want a new product, so then they can put it on the market. Of course, they want a product that will be profitable, thus it cannot be something just for doing.

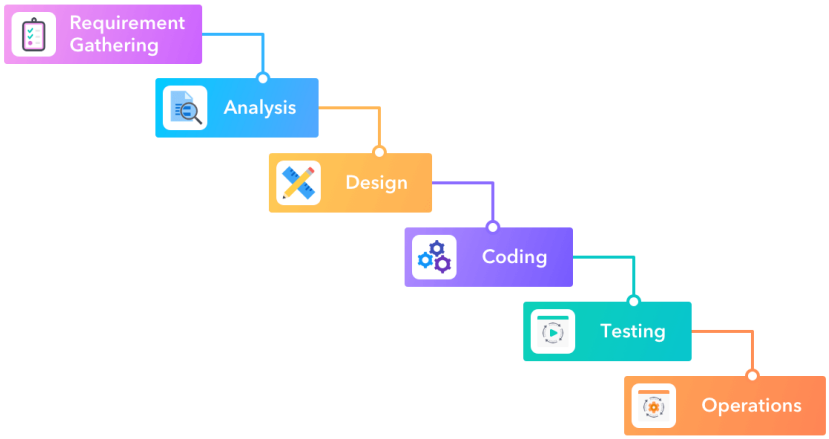


Figure A.1: Waterfall Methodology

This project will be developed using the waterfall methodology, so each chapter corresponds to a phase in this methodology. It's important to notice that this process isn't linear. The will be stepbacks and problems that will come, so it's important to have a critical spirit and, if needed, go back and fix or complement some phase.

B Problem Statement

Nowadays technology is everywhere and everyone wants a smarter, safer and quicker way of doing things. Everyone commutes by car and most people like to store their vehicles inside a garage or at least a private area. Usually access to these spaces is controlled by a gate, which in turn is activated by a radio remote control. Such devices are fragile and can be easily lost or damaged. Furthermore, users must always carry it, which can be inconvenient for someone with multiple vehicles or people accessing such places at different times of the day. This is a very big reality in the case of gated communities or condominiums. It can also be a real hassle to allow someone who isn't a regular to enter the gate as it requires a remote, requiring prior handout of such a device or the presence of someone with a remote to open the gate at their arrival.

With all of this in mind, this project aims to create a system to control access to a private area using license plate recognition. The system would read the license plate of a vehicle trying to enter the area, if such license plate is an authorized one, access is granted. The goal of the project is not to replace remotes altogether but rather to complement them. The user will still be able to open the gate as always, but this product will facilitate the process of getting home. Thus, no need to develop a gate from scratch.

C Problem Statement Analysis

This type of system isn't new. They exist in most paid parking and private parking like university ones. Although, these systems are very directed to the commercial market. They have a lot of functionalities that a simple person that has property doesn't need like:

- Streaming;
- Recording;
- Artificial Intelligence (AI), in order to detect the colour, brand, model, etc. of the car;
- Car parking time control, so the system can be able to charge the time that has been parked;
- And so on.

Because they have all these features, the price comes very high.

C.1 Market Research

The worldwide market for automatic number plate recognition systems size is projected to reach USD 262.7 million by 2026, from USD 186.3 million in 2020, at a Compound Annual Growth Rate (CAGR) of 5.9% during 2021-2026.

As said in the problem statement analysis section, the pricing for these solutions is very high. For example, just a camera to operate can cost 399$, and it doesn't have all the features. Don't forget that this camera doesn't operate alone, so a specific computer is needed in order to make the system work. So the whole package, camera, computer, and cables, can cost more than a thousand dollars. Also, maintenance is needed over the years and it, of course, isn't cheap.

Figure C.1: License Plate Capture Camera

With a CAGR of 5.9% and a gap in plate recognition on personal properties, it's clear that it's a good time-to-market. Our goal is to develop a product that will include all the characteristics and functioning principles needed for a simple personal property, but with a much lower price. The tradeoff here will be some advanced features like brand, model and colour recognition, recording, and streaming. The price for this product will be around 150 euros.

C.2 System Overview

As said in chapter B.1, there will be two main systems. The following diagram, present in figure 2, gives a better idea who these two systems will operate and communicate with each other.

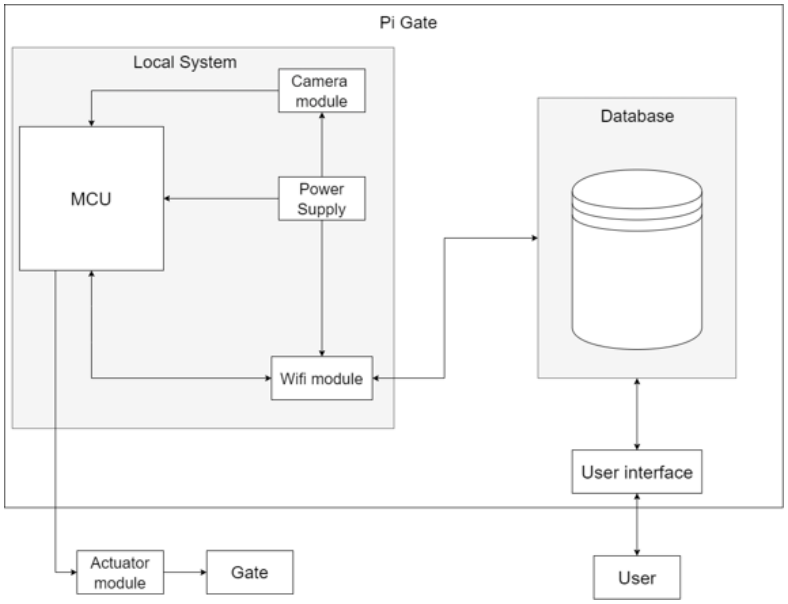


Figure C.2: System Overview

Both systems will communicate via wireless, so cables can be avoided. Also, they will need to have a power supply.

In the local system, the camera module will only output the image to Microcontroller Unit (MCU), so it can make a decision and send the correct signal for the actuator. The actuator is outside of the PIgate because it is on the gate's board. PIgate will add features into actual gates, thus this actuator is already implemented.

In the database, the MCU will communicate with the user by a user interface, in order to provide the best user-friendly experience. Also, it will communicate with the local system permanently, so this system can operate 24 hours/day.

C.3 Requirements

Requirements can be split into functional and non-function.

Functional:

- Work 24 hours/day;
- Wireless communication between the two main systems;
- Add or remove plates without being near to the gate;
- Only whitelisted plates can open the gate;
- Record entries history

Non-functional:

- Low/Medium power consumption;
- Low delay to open the gate;
- An user-friendly interface;
- Be waterproof;
- Be discreet;

C.4 Constraints

Constraints can be split into technical and non-technical.

Technical:

- Use as development board the Raspberry PI 4;
- Use C/C++ language;
- Use buildroot for custom linux system;
- Use device drivers;
- Use Cyber-physical system (CPS);

Non-Technical:

- Only two students working in the project;
- Project deadline at the end of the semester;
- 120 Euro Budget;

C.5 System Architecture

C.5.1 Hardware Architecture

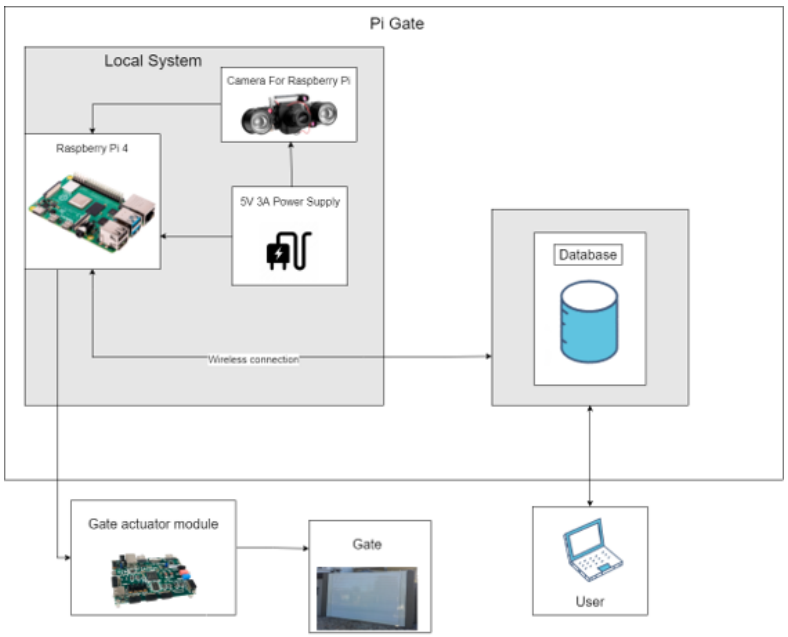


Figure C.3: Hardware Architecture

The local system will be composed of various hardware components. For starters, the processing will be done by a Raspberry PI 4. Because it already has a wifi module, there's no need to buy one. The camera must be used to collect images so that Raspberry can analyze them. Also, it needs to have a night mode feature in order to be able to work at night. To power all this, a 5V 3A power supply will be used because it's the minimum for the Raspberry to work.

The database will be online so there is no physical hardware for it.

C.5.2 Software Architecture

In terms of software, the project is composed of 3 main layers. These layers are:

Operating System:

- This layer is composed of the operating system, board support module, and device drivers.

Middleware:

- The middleware layer will provide software abstraction from the Operating System (OS) layer to the application level by handling the communication protocols.

Application level:

- The application level is where the system's main functionalities will be implemented, using the lower level API's. This is where image computing will occur.

The following picture presents a software architecture for this project.

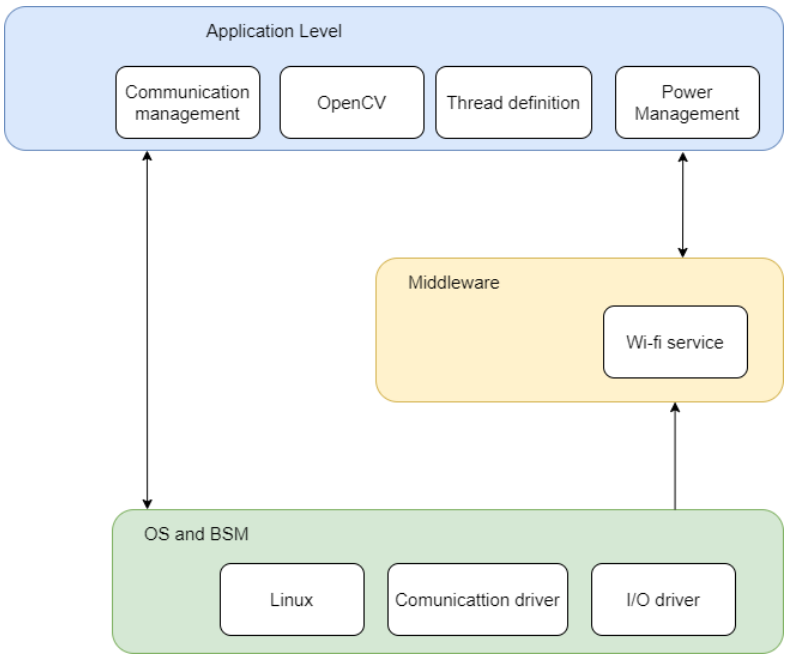


Figure C.4: Software Arhitecture

C.6 Task division/Gantt chart

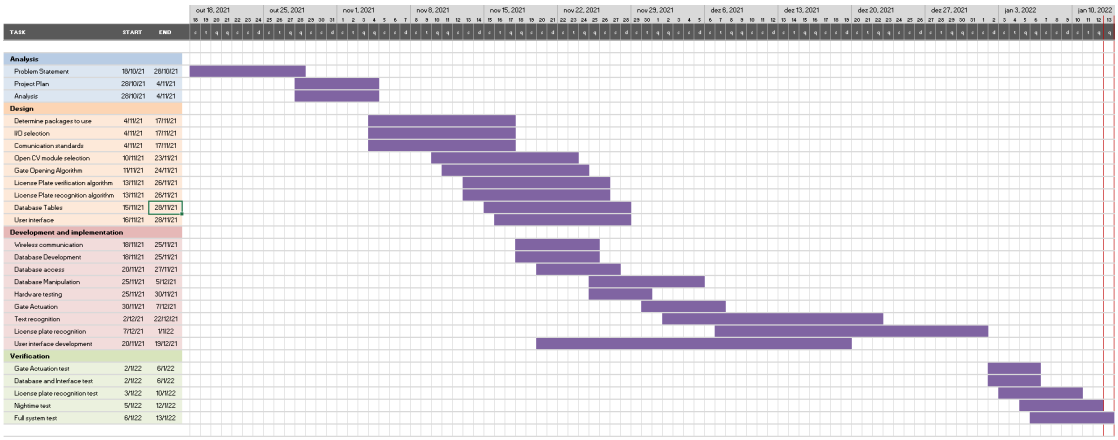


Figure C.5: Gantt Diagram

D Analysis

As shown in figure C.2, there will be two systems, the local one, with the raspberry, and the user interface.

Each system has its characteristics, thus they can't be in the same analysis.

D.1 Local System

D.1.1 Events

As soon as the Raspberry PI turns on, a lot of stuff is going to happen in order to make all the system work, so it's important to know what the main events are and how they respond because it helps to better understand the system. The following table shows those main events, how they respond, their source, and the type of synchronization (synchronous or asynchronous).

Events	System Response	Source	Type
Power on	Initialize camera and connect to the database	User	Asynchronous
Sample Camera	Get the actual image on the camera	Local System	Synchronous
Open Gate	Send a signal to the gate actuator	Local System	Asynchronous
Update plates	Receive the updated plates and storage them	Local System	Synchronous
Send Entry	Send to database the car's plate that has entered	Local System	Asynchronous
Identify Plate	Analyze the plate to check if it's a whitelisted one	Local System	Asynchronous
Camera Check	Checks if the camera is working	Local System	Synchronous
Database Check	Checks if the database is online	Local System	Synchronous

Table D.1: Local System Events

Most of the events are asynchronous because the system never knows when it's the time to actuate, so it needs to be always prepared for any arrival at any time.

Every time the system identifies a plate, it will use the most recent storage white list. Because this list is in constant update, there is no problem of a user arriving at home and the PIgate hasn't updated. Another solution could be to request the most recent white list in time execution but it brings several problems. The most critical one is the internet connection failure. If this happens, there is no way to open the gate. Another issue is the time. When the PIgate detects a plate, it needs to be as fast as possible to recognize that plate and open the gate, if that's the case. Requesting the plates at this moment turns the system slower what is it not wanted.

To keep the system operational, it needs to do some verifications, so it can check if everything is ok. Camera check and database check events are the ones that do it, each one for each part.

D.1.2 Use Case Diagram

In order to help better understand the system context, a use case diagram has been developed. Here it's easy to visualize how the local system interacts with the actor, which, in this case, it's just the database. The following figure shows the use case diagram for the local system.

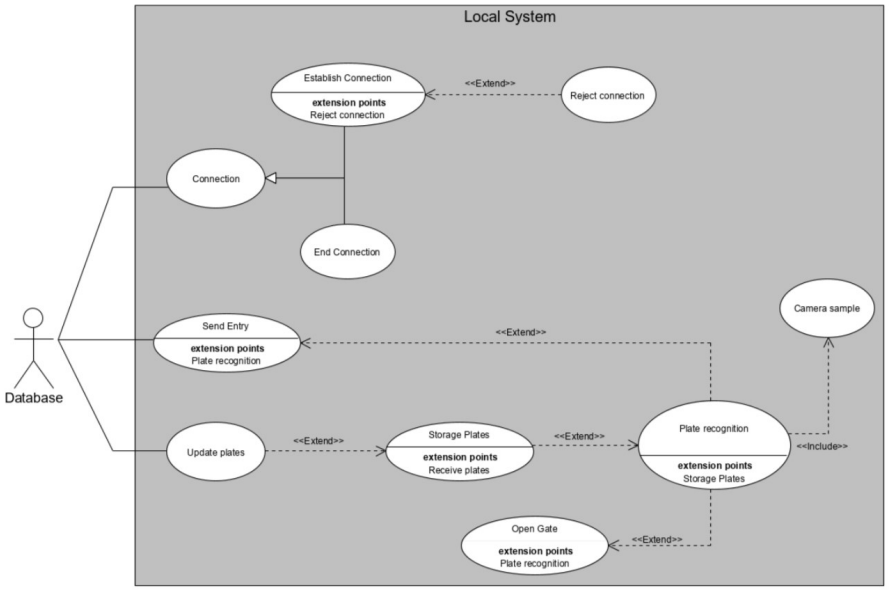


Figure D.1: Local System Use Case Diagram

First of all, there has to be a connection between the database and the local system, otherwise, send entry and update plates use cases will never be executed. This connection can be rejected if the database is in maintenance or if it is the wrong database.

Then there is the send entry use case. Here is where the local system sends to the database the entries. It only sends the plate number in order to not spend a lot of time in this operation. The database, when receiving the plate, automatically stores the entry with the time and date.

Finally, there are the update plates use case. As said before, it will be asynchronous and synchronous. Every time this executes, the plates are stored in one reserved zone for them. This zone is managed by storage plates use case.

In the local system, the main use case is the plate recognition one. It is responsible to do all the work of recognition, as the name says. For that, it uses the image output from the camera sample use case. Then it decides if the image is a plate and if it is an allowed one.

D.1.3 State Diagram

After having an end user's perspective with the use case diagram, it's also necessary to have an abstract description of the behavior of the system with the state diagram. The picture D.2 shows the local system state diagram.

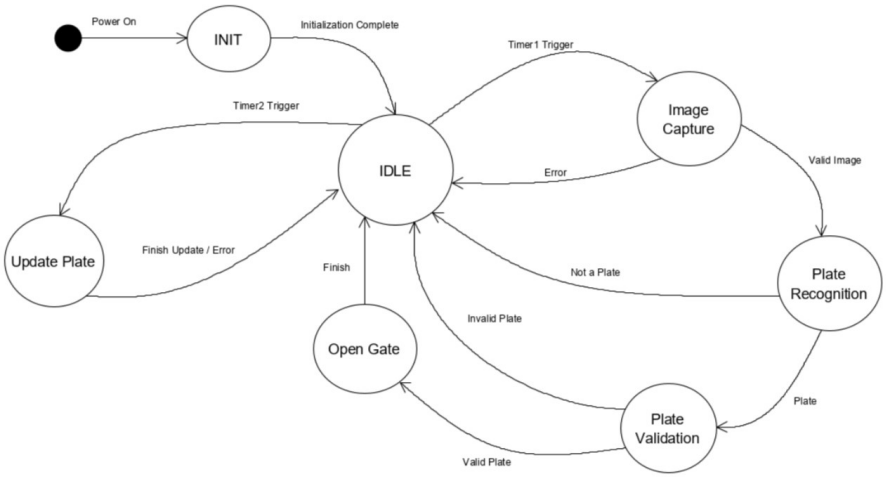


Figure D.2: Local System State Diagram

Everything starts when the user powers on the system. Then, when all the initializations are done (Database connection, camera feedback, etc.) the system goes to an idle state. In this way, the PIgate will be in low consumption mode. This is possible because the client isn't a company or a parking lot. In the private scenario, gates aren't opening all the time so there is no need to be always verify the environment. However, the delay can't be too high. Waiting 30 seconds to open a gate is unimaginable. So timer1 needs to have a trigger time that balances these two scenarios.

After the Timer1 trigger, everything runs smoothly. An image of the environment is captured. If some error occurs, it goes immediately to an idle state. Then, in the plate recognition state, the system will analyze the image to detect if there is a plate. If it's found, go to the next state. If not, it should go to Idle state and wait. In the plate validation state, the found plate will be compared to the whitelisted ones. If there is a match, it means the person is allowed to enter so the gate must open. Otherwise, nothing changes and the system goes idle. To finish, as the name suggests, the open gate state only has the task to open the gate. After that task is over, the cycle ends and the system goes idle.

Timer 2 will be responsible for the plate's update. When it triggers, the system goes to update plate state and the plates are updated. There are two possible outputs. The finish update one when everything went ok. The error one when something wrong happened. This could be a missed network connection, an unreachable database, etc.

D.1.4 Sequence Diagram

The sequence diagram allows to detail how operations are carried out. These types of diagrams are time focused and they show the order of the interaction visually by using the vertical axis of the diagram to represent time, what messages are sent, and when. The following images show the local system sequence diagrams.

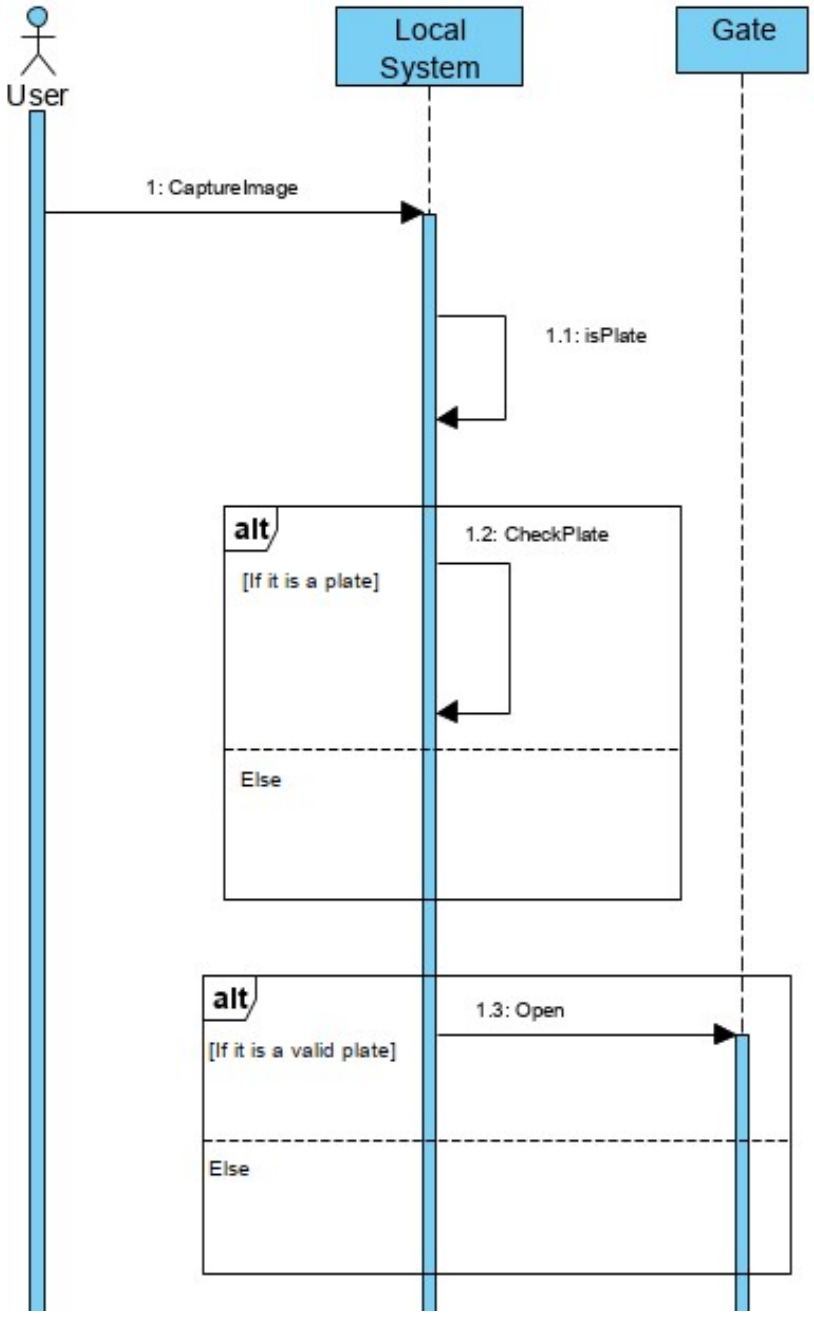


Figure D.3: Local System Main Sequence Diagram

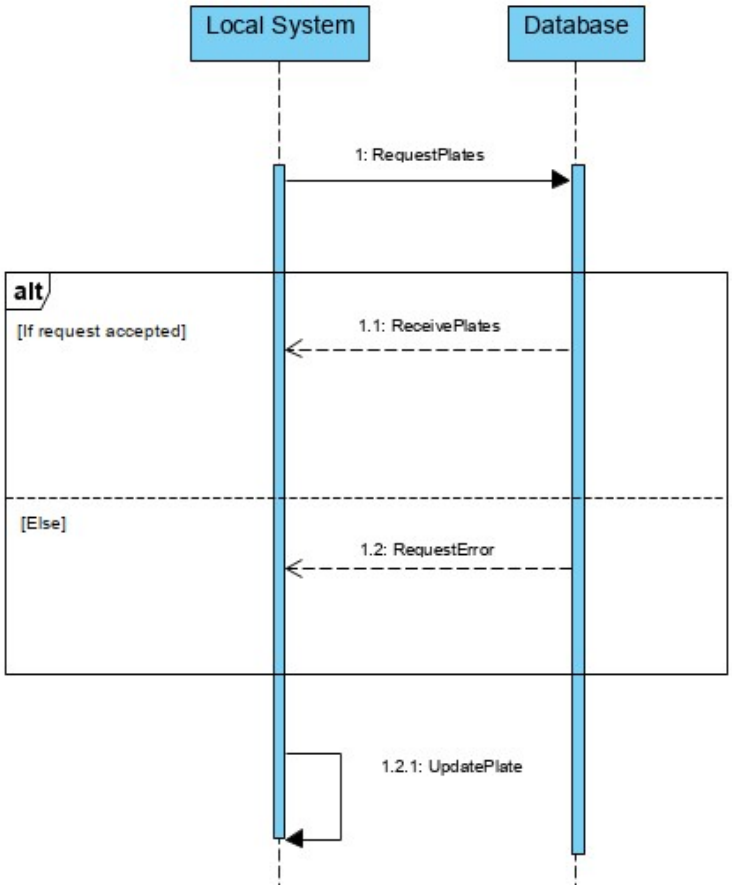


Figure D.4: Local System Update Plate Sequence Diagram

The first sequence diagram is the main one. The explanation of this is very similar to the one given in the previous sub-section. The local system captures an image, then verifies if there is a plate. If not, the system goes idle, so no more iterations. If yes, request the latest whitelist plates. The local system will receive them if there is a connection established with the database. Then the recognized plate is analyzed and compared. If it is a valid one, a signal to the gate is sent. After this, the system goes idle. Notice that the database is not always being used. This allows better performance for both. The local system doesn't need to always communicate with the database, so the processor can do other things. The database doesn't need to be preoccupied with all PIgates at the same time, so it won't be overloaded.

The second diagram represents what happens when the synchronous event, update plates, occurs. First, there is a request for the latest white listed plate. Then two scenarios can happen. The first one, everything went OK and the system received those plates. The system will delete the last version and store the new one. The second possible scenario is an error occurs. It can happen for multiple reasons. Internet connection failed. The database is unreachable. Access permission denied. No free memory. This last one is very unlikely to happen because Raspberry PI has a lot of memory and this data isn't heavy.

D.2 User Interface

The user interface is responsible for providing a way for the user to interact with the database, in order to add, remove or check the license plates in the white list.

D.2.1 Events

The user interface will provide to the user a lot of functionalities, so the user can manage the system. The next figure shows all the necessary events, how they respond, their source, and the type of synchronization (synchronous or asynchronous).

Events	System Response	Source	Type
Add plate	Add license plate to whitelist	User	Asynchronous
Remove Plate	Remove plate from whitelist	User	Asynchronous
Check Entries	Show entry list	User	Asynchronous
Check Whitelist	Show whitelist	User	Asynchronous
Rename Gate	Renames the PIgate	User	Asynchronous
Add PIgate	Add PIgate to user account	User	Asynchronous
Remove PIgate	Removes PIgate from user account	User	Asynchronous
User Login	Verify Identity	User	Asynchronous
User Logout	End user session	User	Asynchronous
User Register	Registers a new user	User	Asynchronous

Table D.2: User Interface Events

First of all, there is the user register event. Here the system will register the user in the database, so the user login event can verify the user identity.

When the user logs in, there will be presented the PIgates which the user has. Is possible to add a PIgate that the user doesn't own, but he needs to know the PIgate's password. To remove also needs it to re-force the delete idea. When a gate is selected, some options will be presented. Add plate, Remove plate, Check entries, Check whitelist and Rename gate. For each option, there's an event, so the system can respond.

Important to notice that all the events are user source. This happens because it is the user that controls the user interface and so is responsible for all the events. Furthermore, there is no way to know when the user wants to login, add plates, etc. thus all these events are asynchronous.

D.2.2 Use Case Diagram

Use case diagrams can be helpful to illustrate how the system is structured, who are the actors, and how they interact with the system. The following image shows this diagram.

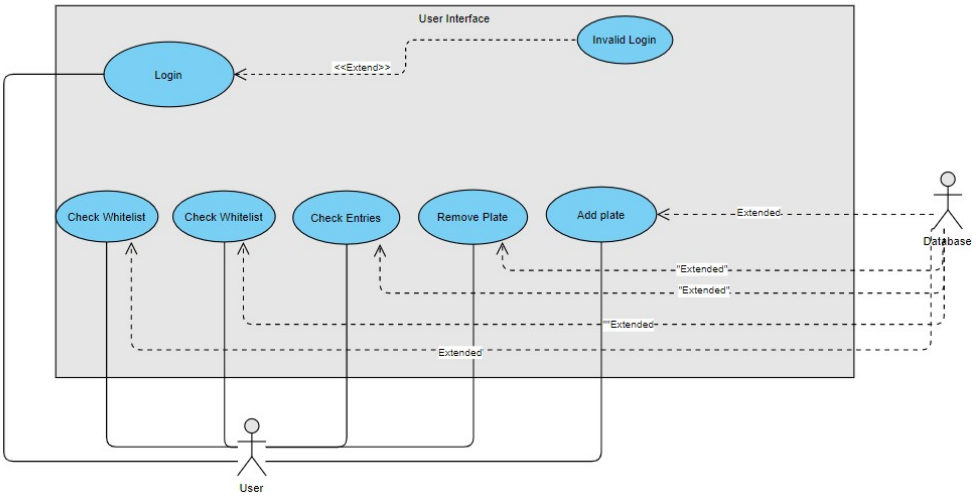


Figure D.5: User Interface Use Case Diagram

In the particular case of the user interface of PIgate, there are two actors, the user, and the database.

The user will be able to login to the platform, then, he can manage the PIgate's that he has added. The options are: Change the White list by adding and removing plates, check the entry log, check the full list and rename the PIgate's name. All of these features require the exchange of information with the database which will have this information stored.

The invalid login use case only is executed when either login credentials are wrong. These credentials are the username and the password.

D.2.3 State Diagram

Picture D.6 presents a state diagram for the user interface which describes, in an abstract manner, how the program will work and how a license plate can be found.

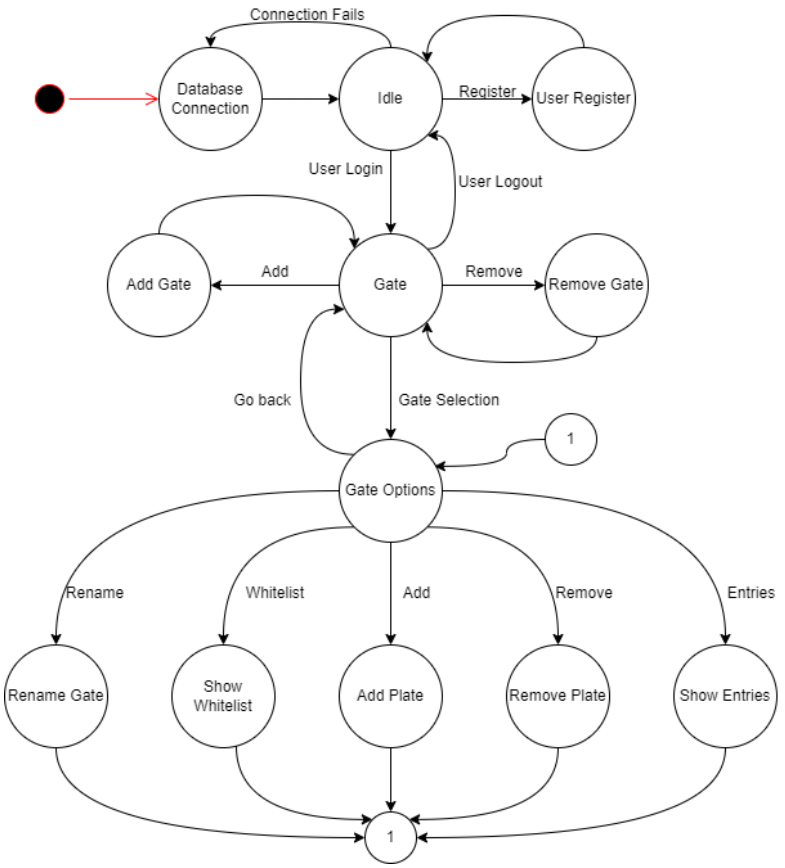


Figure D.6: User Interface State Diagram

Right after the system booting, it is important to establish a connection to a database. If there is no connection to the database, the system will not proceed.

Then it can fall into an idle state until a user logs in or the database connection fails. Once the login process is complete, the system waits for a decision to be made. Any of the options can be selected. Once the tasks are over, the system will switch back to the "Show options" state.

In case of a logout option selection, the system goes back to the "idle" state, until a user logs again.

D.2.4 Sequence Diagram

The sequence diagram is important to plan how each action is spread in time. The diagrams are split because the actions are different and they do not execute precisely one after another. The user is who decides what wants to execute. The following images show the diagrams for each action.

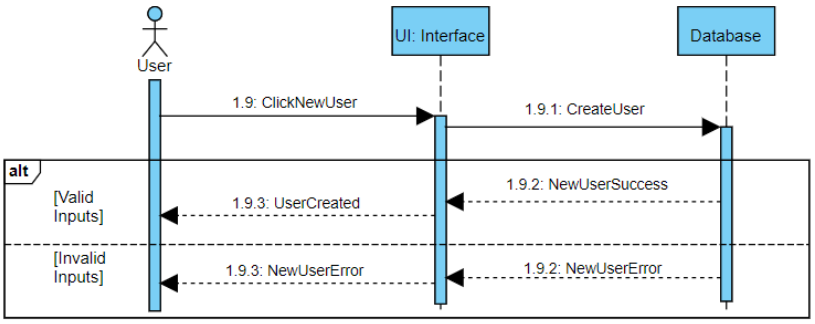


Figure D.7: Register Sequence Diagram

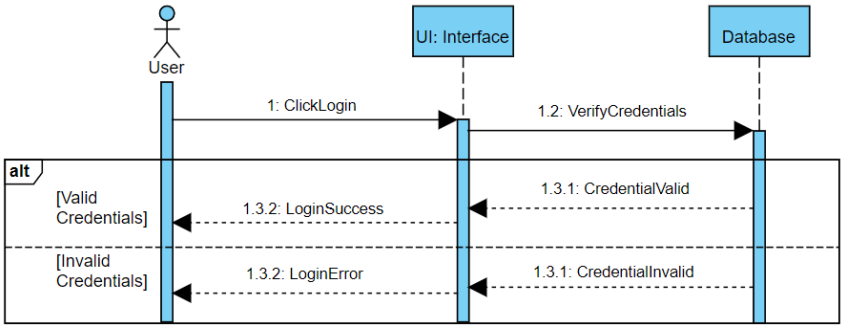


Figure D.8: Login Sequence Diagram

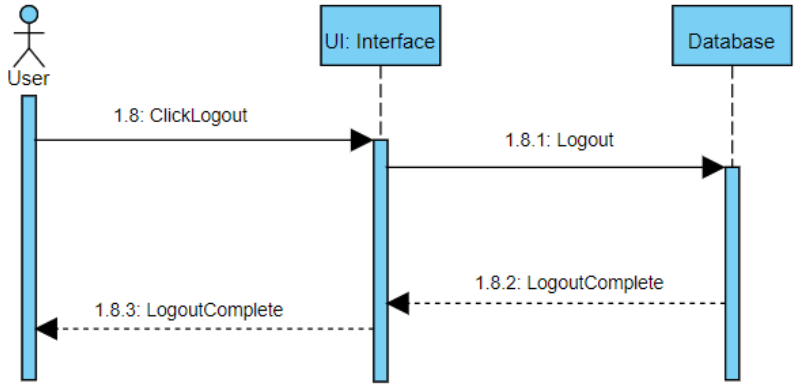


Figure D.9: Logout Sequence Diagram

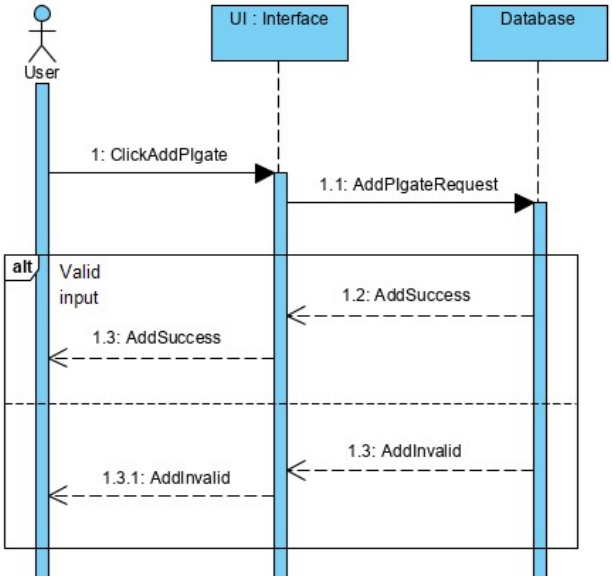


Figure D.10: Add PIgate Sequence Diagram

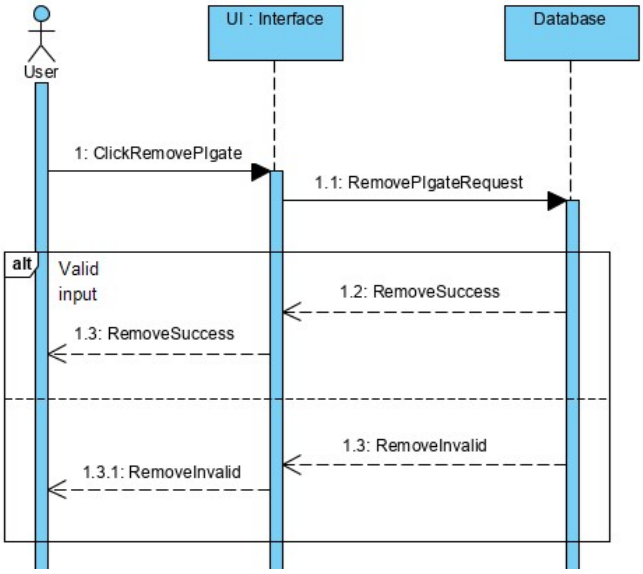


Figure D.11: Remove PIgate Sequence Diagram

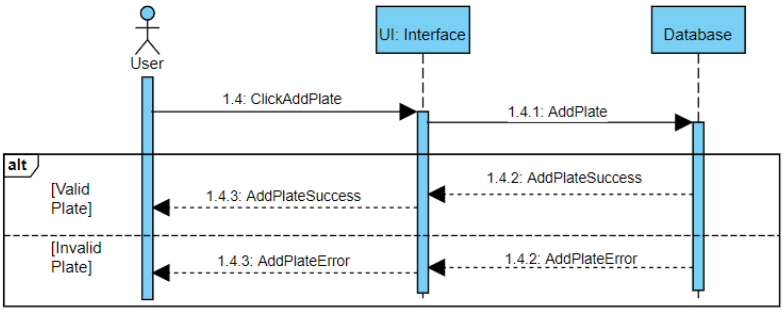


Figure D.12: Add Plate Sequence Diagram

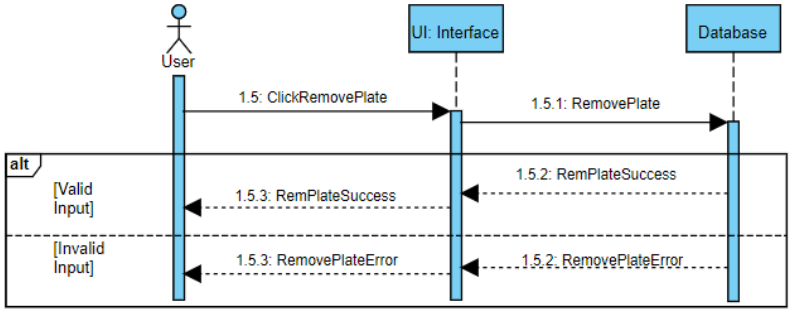


Figure D.13: Remove Plate Sequence Diagram

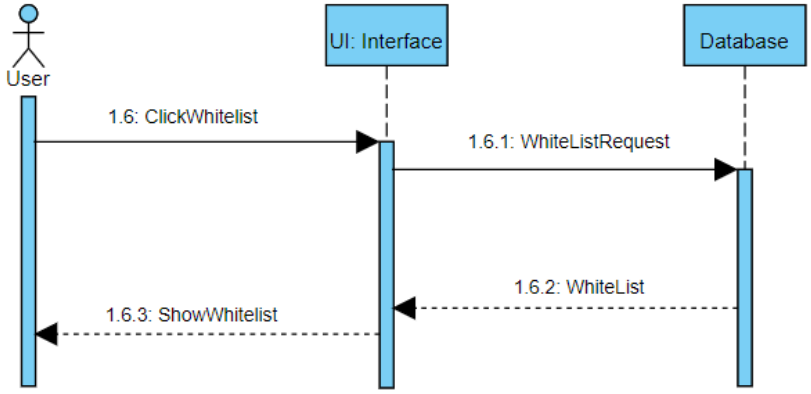


Figure D.14: White list Sequence Diagram

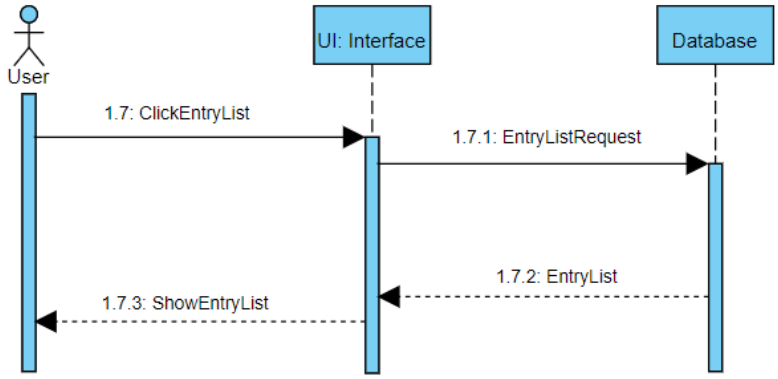


Figure D.15: Entry List Sequence Diagram

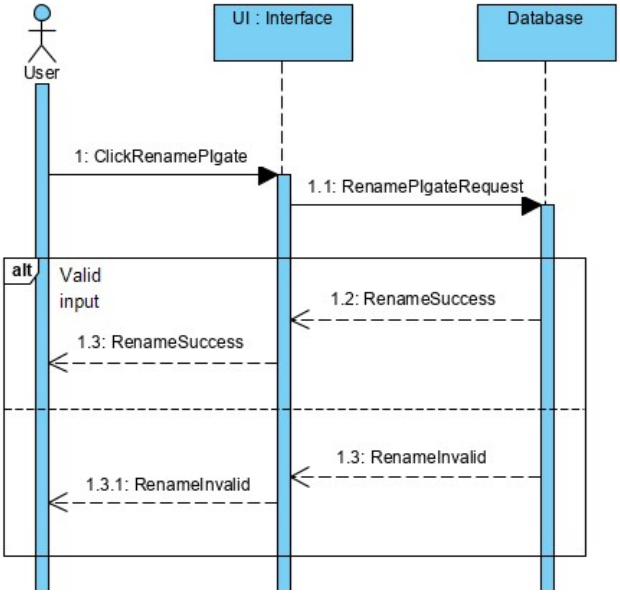


Figure D.16: Rename PIgate Sequence Diagram

Its explanation is very similar to the previous subsection. When the user tries to log in, there is an authentication process, if successful the user can then continue to the other options, if not, then he has to try again.

Once logged in and selected the PIgate, the user can add or remove a plate from the list. The user can also ask to see a list of the whitelisted plates, a list of the entries, and rename the PIgate as well.

When the user wishes to leave, he can also logout, this option will end his session and no further changes can be made to the database until a new login.

D.3 Budget Estimation

For this project, some components will be needed, like the camera, the power supply, etc. and so it's necessary to know how much they are going to cost. After research on the internet and local stores, an estimated budget was done. The next table shows how much each component will cost and the total budget.

Components	Price (Euro)
Raspberry PI 4	52
Camera for Raspberry	14
2x Relay Module	6
Power Supply + Cable	6
Magnetic sensor	4
Encapsulation	15
Total	97

Table D.3: Budget Estimation

Compared to the budget constraint, there is a difference of 30 euros which is good room for maneuver.

E Design

As said in the analysis chapter, the local system and the user interface are different, so it is not possible to do a design for them together. Furthermore, the database needs also to have a design because otherwise, it can make the whole system slow and heavy.

E.1 Theoretical Concepts

To better understand the design chapter, some theoretical concepts about some protocols have to be provided. However, a background in micro-controllers and electronics is also needed.

E.1.1 I2C

The Inter-IC Communication (I2C) is a serial protocol that uses two wires to communicate with low-speed devices like microcontrollers, sensors, etc. It is used by most of the Integrated Circuit (IC) manufacturers because it's simple to use, can be more than one master and only two wires (Serial Data (SDA) and Serial Clock (SCL)) with pull-up resistors in each line.

Sure but, what is a master? A master is the one who controls the communication. Typically, in every communication protocol, there are masters and slaves. The slaves are the ones who respond to the masters' requests. So, imagine that a MCU whats to get the values of a sensor. The MCU is the master and the sensor is the slave.

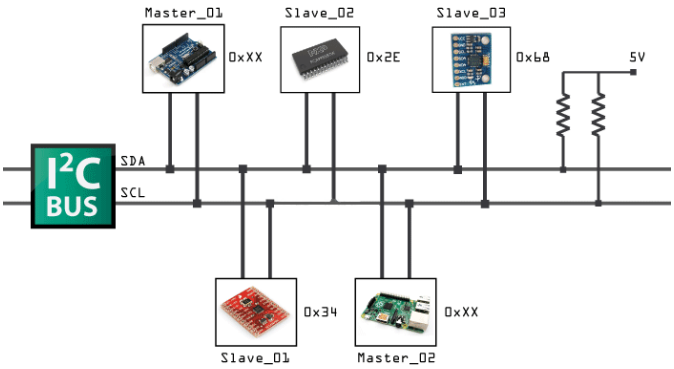


Figure E.1: I2c Communication Example

As shown in the previous image, this is simple to build and does not spend lots of resources. But if all the masters and slaves are connected to the same BUS[1], how do they know that the request or the response is for them? Well, every slave has a unique 7-bit address on the BUS, so when a master wants to communicate with a slave, first it needs to specify the slave.

[1] A bus is a subsystem that is used to connect computer components and transfer data between them

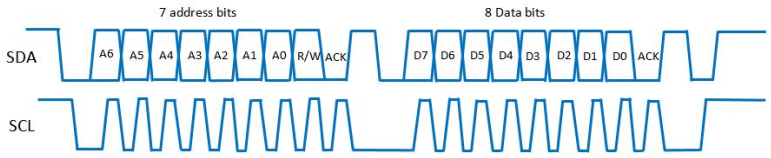


Figure E.2: I2c Protocol

Initially, both lines are high logic level. To start/stop the communication, a start/stop condition needs to be done. This condition can be seen in the figure E.2.

I2C always transfers every octet [2] of data and, in the end, there is an Acknowledge (ACK) bit that is responsible to check if there were any problems during the communication.

As said, the first byte is to specify the salve, although the slave address is 7-bit, so the last bit is to specify if it's a read or a write action. Accordingly, with this bit, the slave will respond to the request.

E.1.2 CSI

The Camera Serial Interface (CSI) is a specification of the Mobile Industry Processor Interface (MIPI) and it defines an interface between the camera and the host. The main advantages of this protocol are that it has a high performance and has low consumption.

There are some interface specifications. The raspberry uses the CSI-2 one, among with a D-PHY physical layer with 2 data lines (data0 and data1).

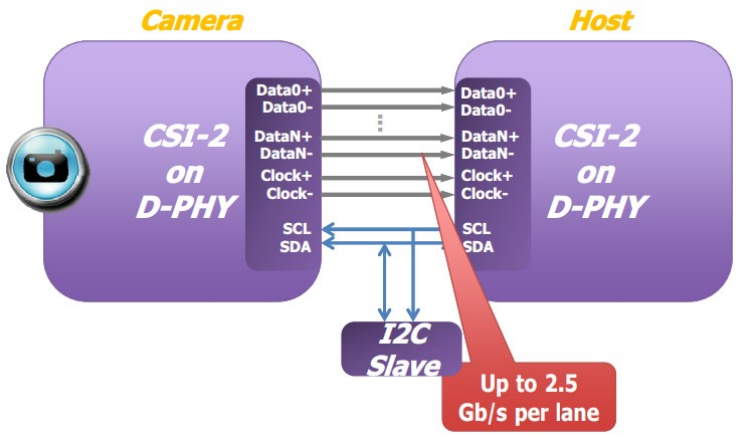


Figure E.3: CSI diagram

Of course, this diagram is missing the Ground (GND) and power connections. As the name says, this protocol uses serial communication which means that the data is sent according to the clock source. This connection is half-duplex which means that the host just receives the data.

[2] Octet is a unit of digital information in computing and telecommunications that consists of eight bits

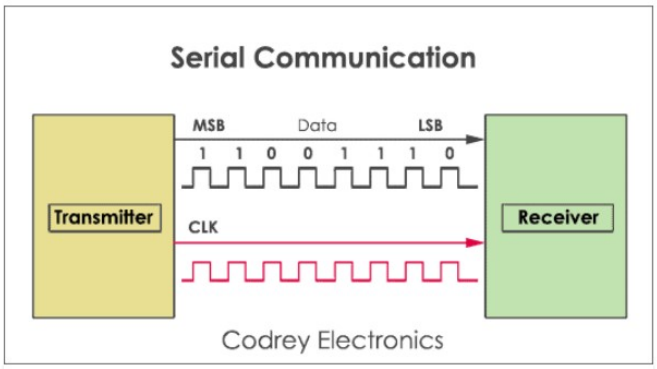


Figure E.4: Serial Communication

This protocol is synchronous, so it operates with a source clock. SDA is in high logic level on idle. When the communication begins, SDA can turn a '0' or a '1', depending on what byte the sender wants to transmit. After the 8 bits, a parity bit is transmitted, to detect errors. This bit is calculated with the sent ones. It is '1' for even binary ones. It is '0' for odd binary ones. The parity bit is not mandatory in serial communication, although CSI uses it. The following picture shows an example of a CSI communication.

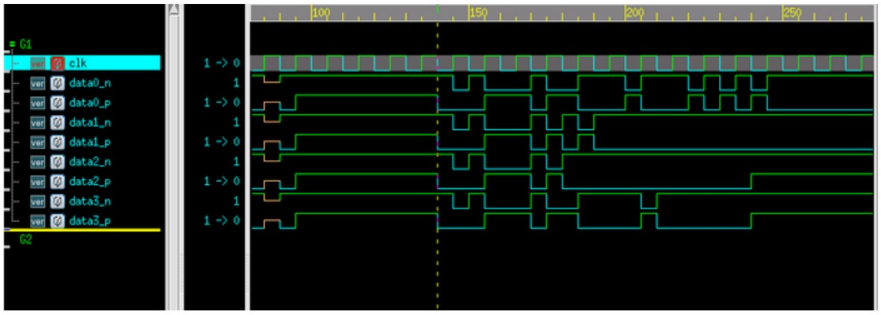


Figure E.5: CSI Example

E.1.3 Wi-Fi

Wireless Fidelity (Wi-Fi) is used to define any of the IEEE 802.11 wireless standards for Wi-Fi protocol. This protocol is an advanced high-speed protocol mostly used in a public Internet connection to overcome the bandwidth issue on Bluetooth.

Raspberry PI has a Wi-Fi module in its board. It uses the 802.11ac protocol, which means a 433 to 6933 Mbit/s link rate and 2.4/5 GHz.

Figure E.6: Generic 802.11 Frame

This protocol transmits 8-bit frames. Each field can transmit one or more frames. Those can be general and specific. The general ones are present in all types of frames. The specific, as the name suggests, are specific of some types. The type frame on the figure E.6 is what raspberry uses.

The frame control field can be split into 11 sub-fields.

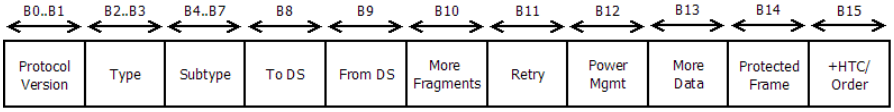


Figure E.7: 802.11 Frame Control

The B0, B1, etc. are the bit 0, bit 1, and so on. The first 3 sub-fields are general ones with the Frame Check Sequence (FCS), present on the figure E.6.

E.1.4 Haar cascade classifier

Object Detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in their paper, "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001. It is a machine learning-based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images. There are various types of haar-features. This method makes use of three of these types, as shown in the figure below.

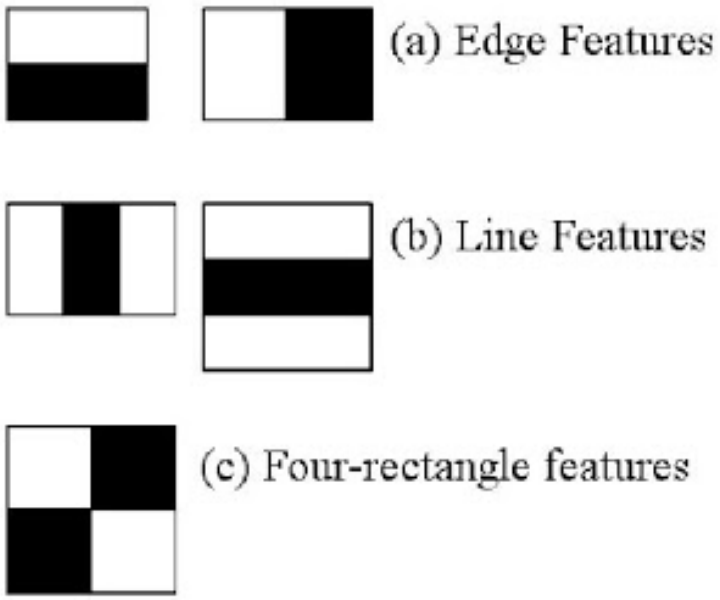


Figure E.8: Haar features

"This work is distinguished by three key contributions. The first is the introduction of a new image representation called the "integral image" which allows the features used by our detector to be computed very quickly. The second is a learning algorithm, based on AdaBoost, which selects a small number of critical visual features from a larger set and yields extremely efficient classifiers. The third contribution is a method for combining increasingly more complex classifiers in a "cascade" which allows background regions of the image to be quickly discarded while spending more computation on promising object-like regions. The cascade can be viewed as an object-specific focus-of-attention mechanism which unlike previous approaches provides statistical guarantees that discarded regions are unlikely to contain the object of interest." [33]

"The cascade training process involves two types of trade-offs. In most cases, classifiers with more features will achieve higher detection rates and lower false-positive rates. At the same time, classifiers with more features require more time to compute. Each stage in the cascade reduces the false positive rate and decreases the detection rate. A target is selected for the minimum reduction in false positives and the maximum decrease in detection. Each stage is trained by adding features until the target detection and false positives rates are met (these rates are determined by testing the detector on a validation set). Stages are added until the overall target for false-positive and detection rate is met."[33]

E.2 Design Tools

SolidWorks: To develop the physical module.

SolidWorks is a 3D CAD software. It isn't free but it's one of the most popular CAD tools, which means that it has lots of support online. Also, the interface is clean and easy to use.

Figure E.9: SolidWorks

Firebase: To develop the database.

Google Firebase is a backend tool to develop high-performance and scalable apps for web and mobile at low cost, time, and effort. Firebase provides software Development Kit (SDK) for many programming languages and is in constant update. Also provides a lot of documentation, tutorials, and support for its services.

It has free and paid services. For this project, the free one will be used, which is good enough.

Figure E.10: Firebase

Visual Studio IDE: To develop all the software.

This is a powerful IDE that is open source and has a lot of extensions, so it will become easier to find support for any problems that might appear. Also, it's available in Linux, Windows, and Mac OS.

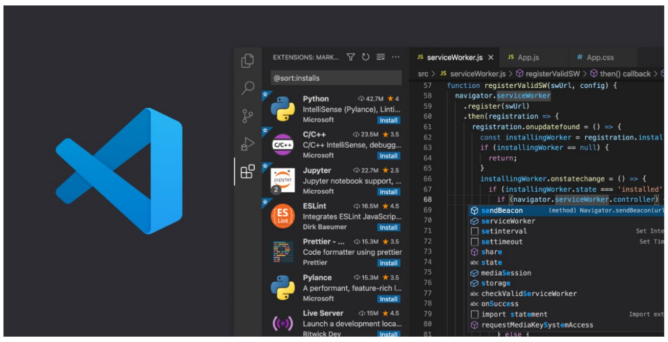


Figure E.11: Visual Studio IDE

Proto.io: To design the Graphic User Interface (GUI).

It has some features that make the experience smooth and intuitive. It's possible to export all the work in HyperText Markup Language (HTML) and Portable Document Format (PDF) format and preview it in real-time. This tool isn't free but it provides a free trial of 15 days, which is totally enough.

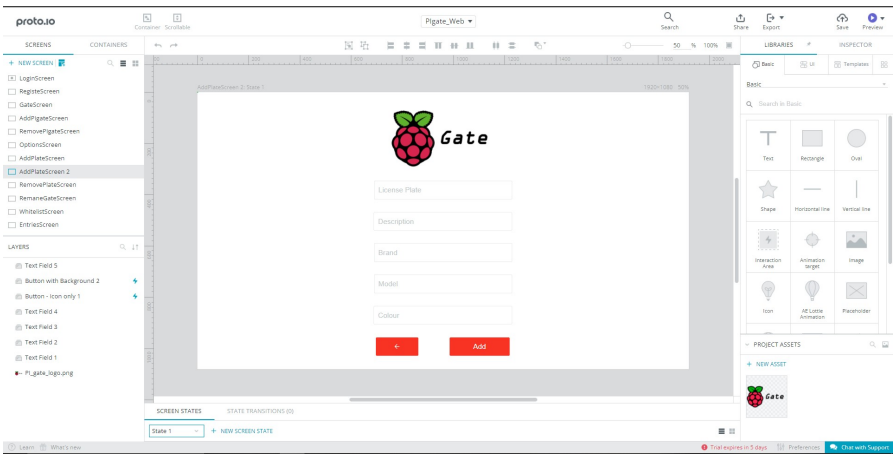


Figure E.12: Proto.io User Interface

Git and GitHub: To have a version control tool.

With this, it's possible to have an always-online repository where every project element can download the latest version. Also, it tracks the versions, thus if some bug suddenly appears, it becomes easy to go back a few versions back and find the problem.

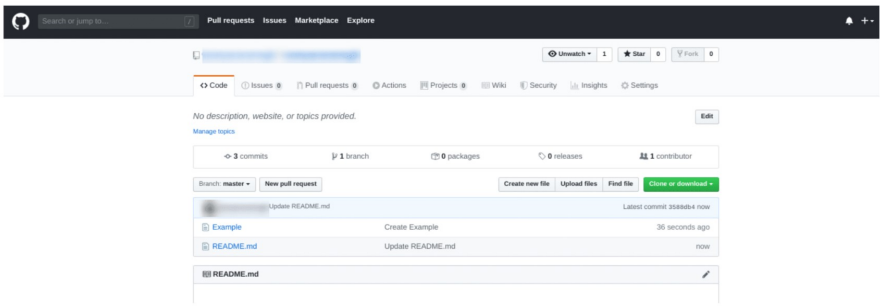


Figure E.13: GitHub User Interface

Meld: To resolve merge problems.

The reason is that it solves the problem in a simple and very intuitive way. Also, it is free.

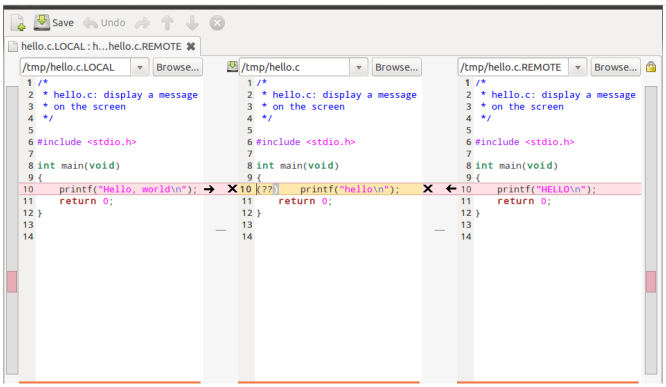


Figure E.14: Meld User Interface

Buildroot: To create the custom Linux system.

Buildroot is a set of Makefiles and patches that simplifies and automates the process of building a complete and bootable Linux environment for an embedded system.

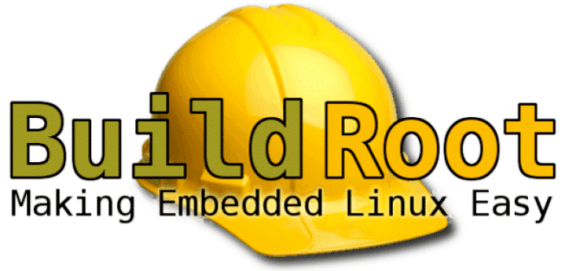


Figure E.15: Buildroot

E.3 Hardware

All the products that use components have a hardware design. It needs to be well designed because it will affect the system directly. Take the camera as an example. One requirement for this project is to be able to work at night. If the camera doesn't have night mode, it will be nearly impossible to respect that. All the components selection has to have that in mind. But also the physical structure. Another requirement is to be waterproof. If the encapsulation is weak, the water will get in and damage all the components.

E.3.1 Architecture

The system will be based on a Raspberry Pi4 which will do all the local processing. This board will be connected to some peripherals. As mentioned before there will be a camera, used to capture images of the area in front of the gate, there is also the need for two relays which will provide an interface with the motor controller. To open the gate from the inside there is a magnetic field sensor, which detects the presence of a vehicle. To finish the system there will also be 3 LEDs along with their respective actuation hardware, composed of resistors and BJT transistors. All of this will be powered using a USB-c power supply, connected to a power distribution board (PDB).

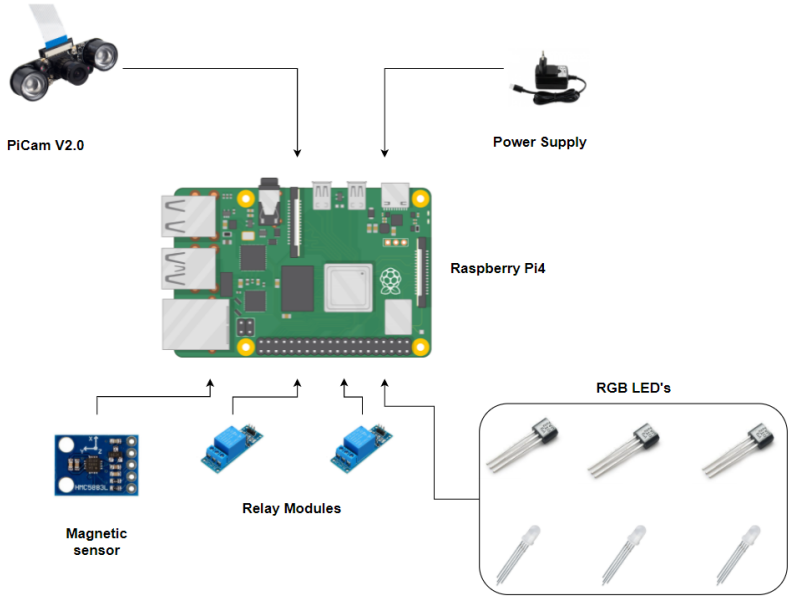


Figure E.16: System Architecture

E.3.2 Component Specification

Raspberry Pi 4:

- Quad-core 64-bit Cortex-A72 processor;
- 2 Gb ram memory;
- Onboard Wireless module;
- MIPI CSI camera port;
- I2C communication;
- 26 GPIO pins.
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless

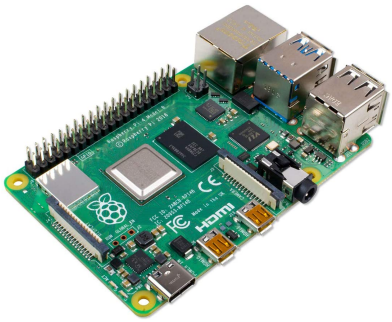


Figure E.17: Raspberry PI 4

PIcam V2.0:

- 5MP sensor;
- MIPI CSI communication;
- Night Vision capable with IR LED and light sensors;
- 1080P video capture;
- Adjustable focus;

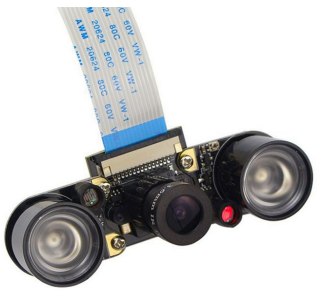


Figure E.18: PIcam V2.0

Relay Module:

- TTL control signal of 3,5V-12V;
- AC: 250V 10A;
- DC: 30V 10A;
- Pull-Down Circuit

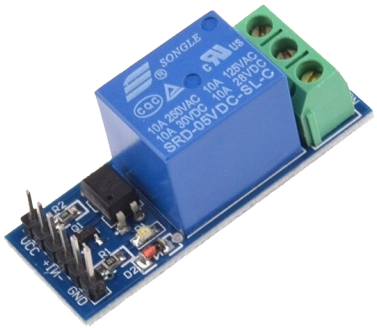


Figure E.19: Relay Module

Magnetic Sensor:

- Model: HMC588L;
- 3.3V supply voltage;
- 3-axis
- I2C communication protocol

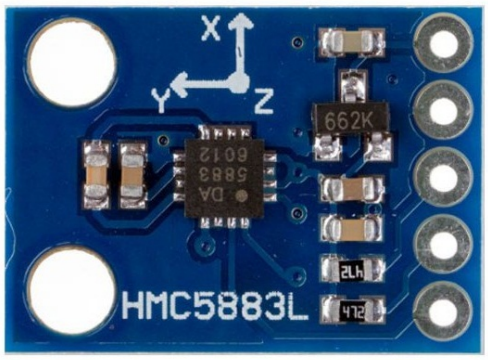


Figure E.20: Magnetic Sensor

E.3.3 Peripherals Connection Layout

In the following image, one can observe how the hardware will be connected to the mainboard. The camera uses a CSI connector and ribbon cable to communicate with the raspberry pi. The magnetic sensor will communicate through I2C, and the LEDs will be actuated through PWM and a GPIO pin set as an output. The relays will make use of one GPIO pin each, one configured as an input and another as an output.

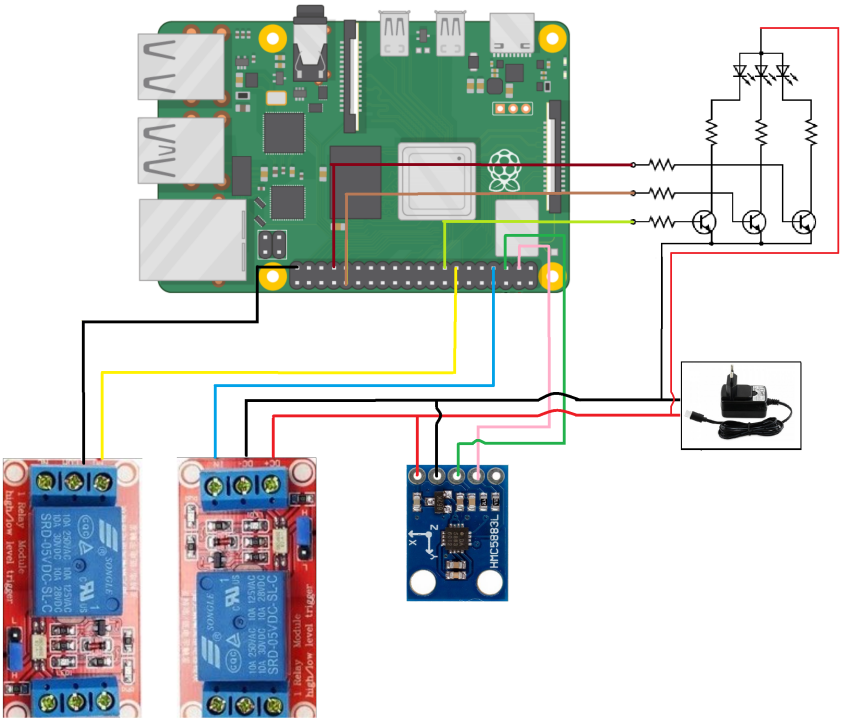


Figure E.21: System Pinout

The following table displays the connections used to connect all the hardware on the local system.

Device	Interface	Connection
Camera	CSI	CSI connector
Input Relay	GPIO PIN as input	GPIO PIN 27
Output Relay	GPION Pin as output	GPIO PIN 4
RGB LED	GPIO Pin as PWM output	GPIO PIN 12
		GPIO PIN 13
	GPIO Pin as output	GPIO PIN 22
Magnetic Sensor	I2C - CLK	GPIO PIN 3
	I2C - SDA	GPIO PIN 2

Table E.1: System Pinnout table

E.3.4 Physical Structure

The PIgate will have an encapsulation, so it will be able to work outside. It needs to be waterproof and have a simple and discreet design. To have an idea how it will look like, a 3D module has been developed.

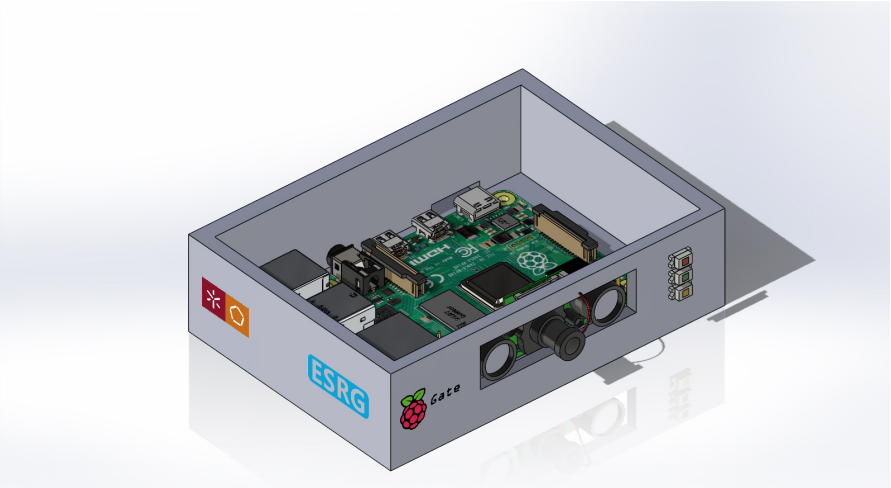


Figure E.22: Physical Structure Without Cover

Figure E.23: Physical Structure

The encapsulation will be small, nearly the size of the Raspberry. It will have two gaps. One for the camera and another for the cables that might get in, like the power source. These gaps will be properly isolated. To access the inside, just unlock the upper cover.

These pictures do not show the connections between the Raspberry and the camera,LEDs, power, etc.

E.3.5 Test Cases

In order to establish the proper function of the system, various tests must be made. The purpose of these tests is to ensure each part of the system is working as expected, making it easier to track mistakes or errors.

Hardware: Tests	Expected Results	Real Results
Camera Image capture	Clear image	
Camera night image capture	Clear night image	
Turn the LEDs on	Leds turned on	
Put a car above the magnetic sensor	Magnetic Field Changes	
Input relay test	Read pin value	
Output relay test	Write pin value	

Table E.2: Hardware test cases

E.4 Software - COTS and Third-party Libraries

- **Firebase JavaScript SDK:** To communicate between web app and Firebase database.
- **Firebase C++ SDK:** To communicate between raspberry and Firebase database.
- **Libcamera:** To take the camera pictures.
- **PThreads:** To create and manipulate threads.
- **OpenCv:** To process images and recognize license plates in them.

E.5 Software - Database

As said in the analysis chapter, this project will need to have a database to store all the information related to plates, users, cars, etc. This information needs to be properly organized and associated otherwise, the database won't work as desired. Information access problems, relationships between entities, and bad definitions of primary and foreign keys are the most usual problems in databases. To prevent all of these problems from happening, a database design was done.

E.5.1 Entity relationship diagram

All databases have an Entity-Relationship Diagram (ERD). Furthermore, Entity-Relationship Diagrams are a type of flowchart that illustrates how entities such as people, objects, or concepts relate to each other within a system. With this type of diagram, it becomes easy to understand how the database will be organized. There are a lot of styles and ways to do this type of diagram. Here will be presented the information engineering style.

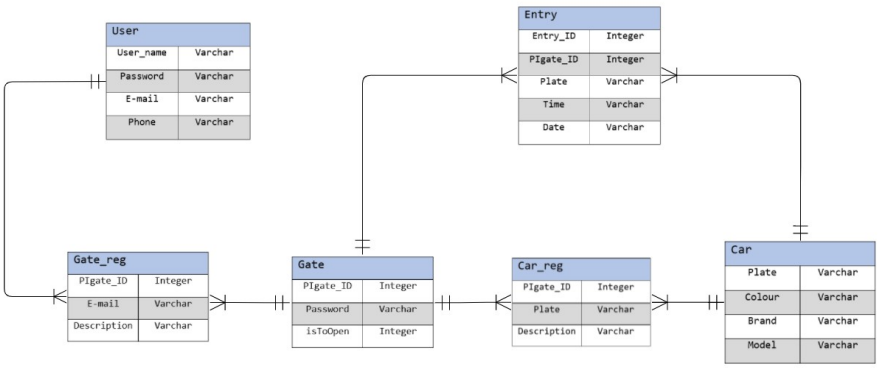


Figure E.24: Database Entity Diagram

The database will have 6 entities, each one with its attributes. Although some of the attributes which are present in some tables do not belong to them, they are foreign keys. This is the result of the data normalization. Data normalization is a type of process in which data inside a database is reorganized in such a way that users can better use that database. It's important because it increases data coherence, speed response and allows further storage. The preview ERD diagram is the result of this normalization. To achieve that, the 3 Normal Forms were used. In the end, the relational logic diagram can be written the following way.

User (E-mail, User_name, Password, Phone)

Gate (PIgate_ID, Password, isToOpen)

Car (Plate, Brand, Model, Colour)

Entry (Entry_ID, PIgate_ID, Plate, Time, Date)

Gate_Reg (PIgate_ID, E-mail, Description)

Car_Reg (PIgate_ID, Plate, Description)

Figure E.25: Database Relational Logical Diagram

E.6 Software - User Interface

User Interface (UI) is a huge part of this project. It's here where the user can interact with the whole system, thus it needs to be the most user-friendly possible. UI must be simple, intuitive, and fast.

Nowadays, the internet is present everywhere.

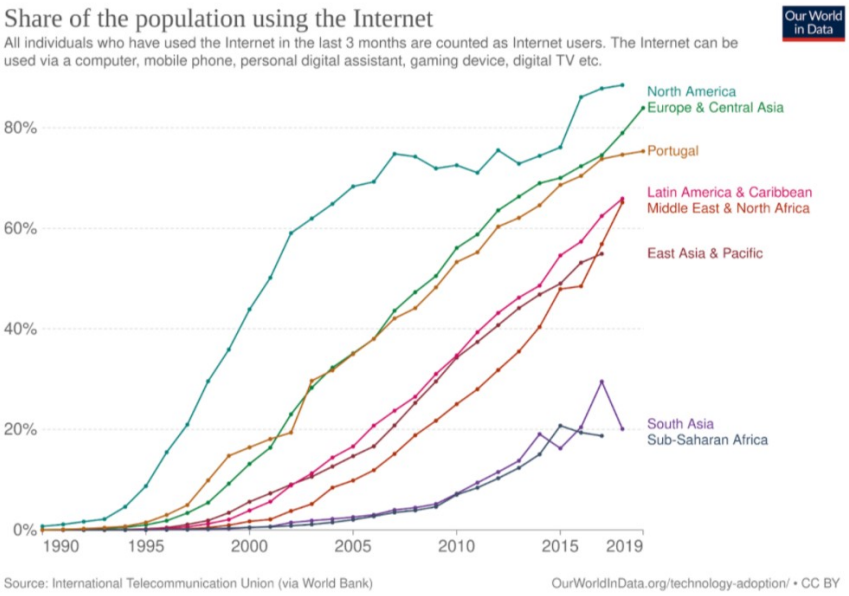


Figure E.26: Share of population using the internet

According to the International Telecommunication Union, internet users are growing each year. In 2019, North America, Central Asia, and Europe across 80% of individuals who have used the Internet. Portugal also has a pretty high percentage. This means it's a good venture to go with an online user interface. This brings a lot of advantages to the user. The most important one is to manage the PIgate anywhere at any time. People use mostly smartphones and computers to access the internet. There are a lot of operating systems, like android, IOS, Linux, Windows, etc. but all of them have something in common. They can access the internet with a web browser, and the output is the same, whatever the operating system is. Take www.google.com as an example. Thus, the PIgate user interface will be a web application, which will be online using Firebase.

E.6.1 GUI Layouts

As said in the tools subsection, all the GUI for the web application was done in proto.io. The following images show how the web app will look like.

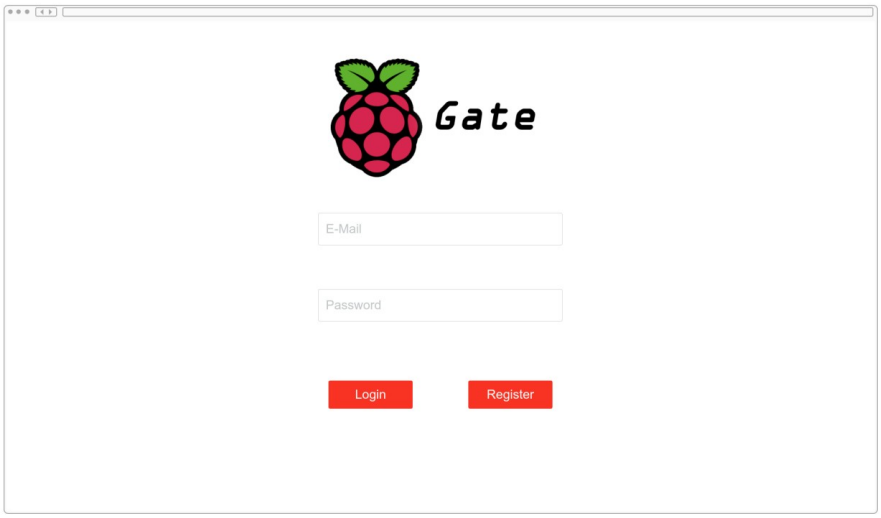


Figure E.27: Login Screen User Interface

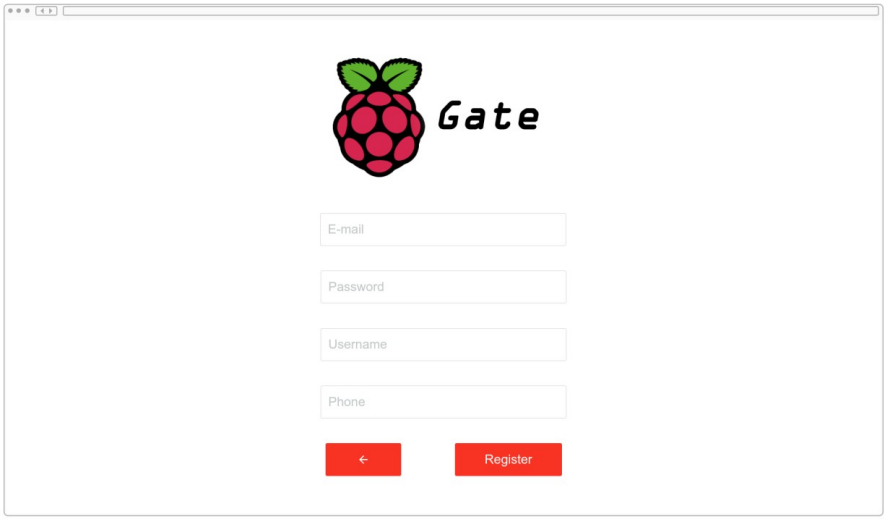


Figure E.28: Register Screen User Interface

The web app will have different pages or screens. Each image is a page that only shows up when something triggers it. For example, the register screen will only show up when the user clicks on the register button.

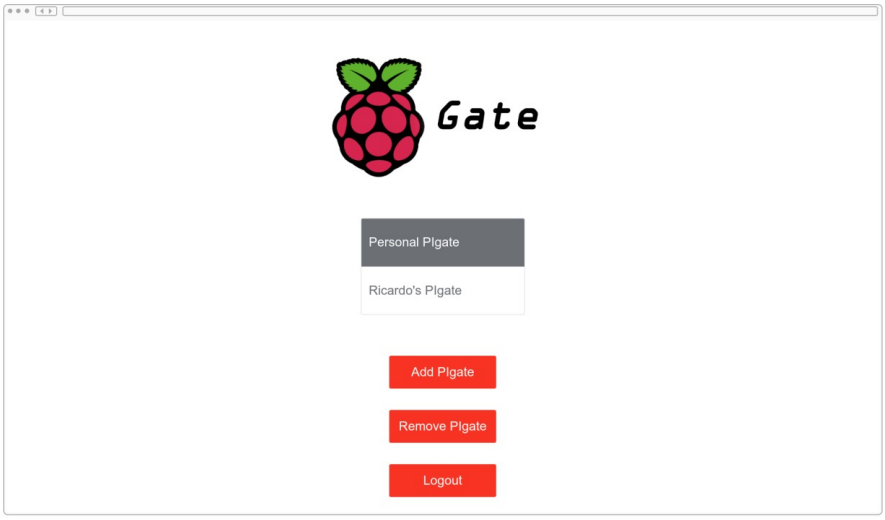


Figure E.29: PIgate Screen User Interface

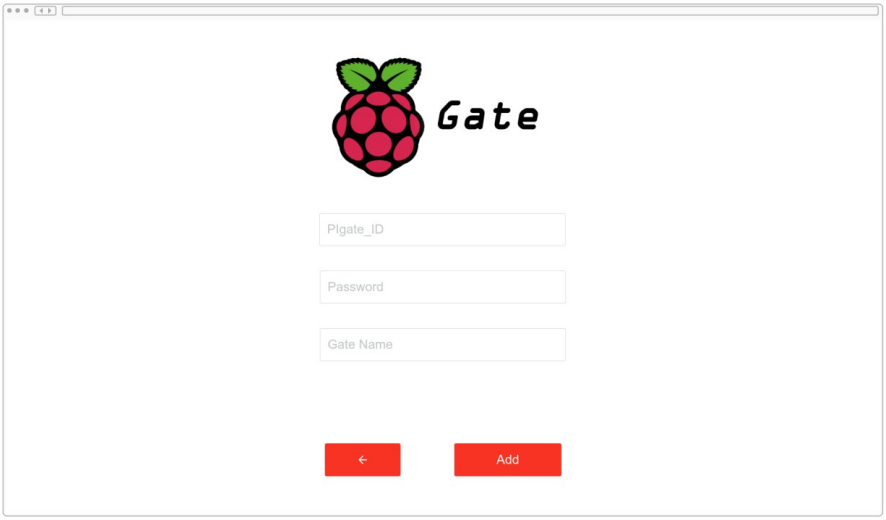


Figure E.30: Add PIgate Screen User Interface

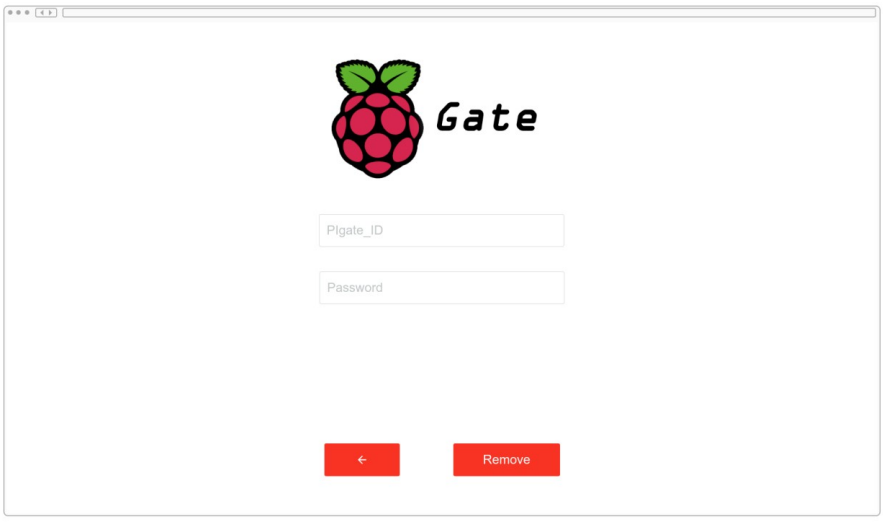


Figure E.31: Remove PIgate Screen User Interface

When a user logs in, the PIgate page will show up. Here the user can add or remove a new PIgate in their account, select the desired PIgate or log out, of course. After selecting the PIgate, the user will go to another screen where will be presented with all the options that he can do.

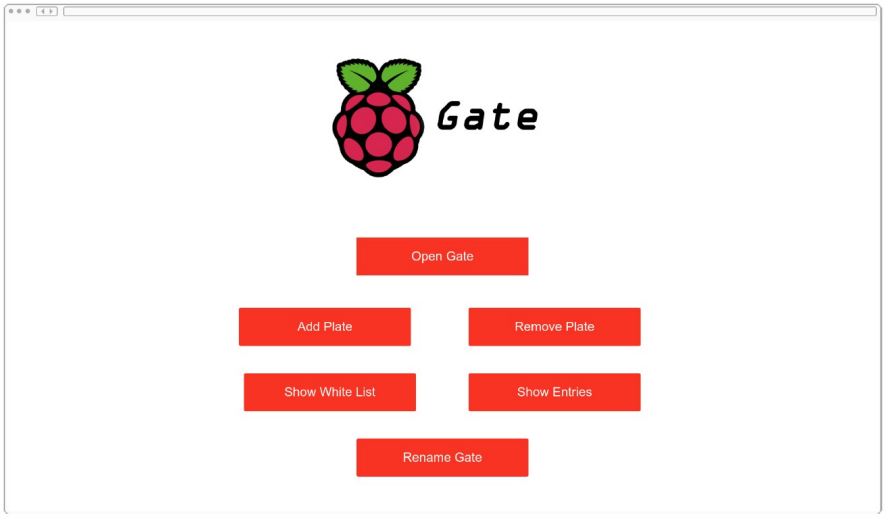


Figure E.32: PIgate Options Screen User Interface

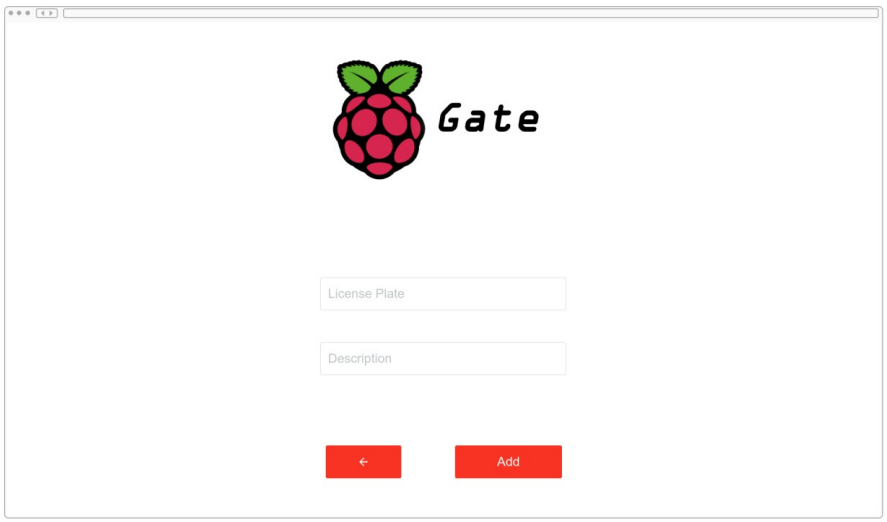


Figure E.33: Add Plate Screen User Interface

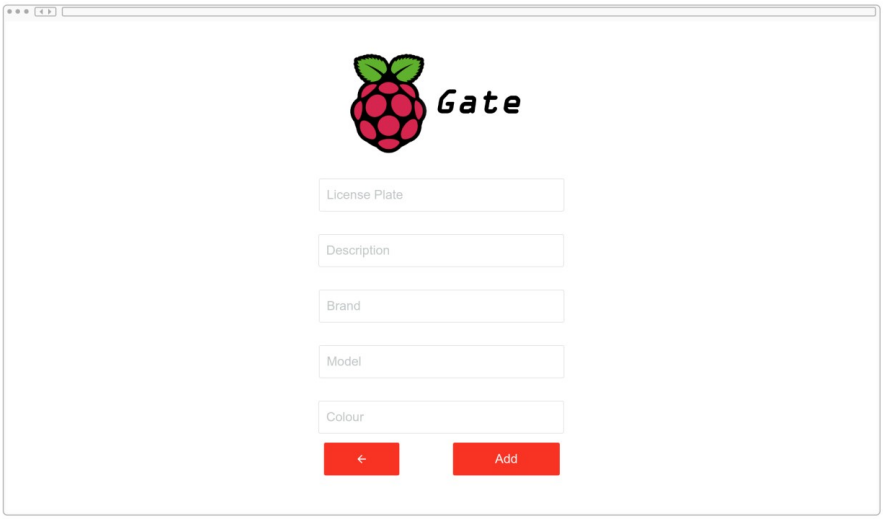


Figure E.34: Add Plate Complete Screen User Interface

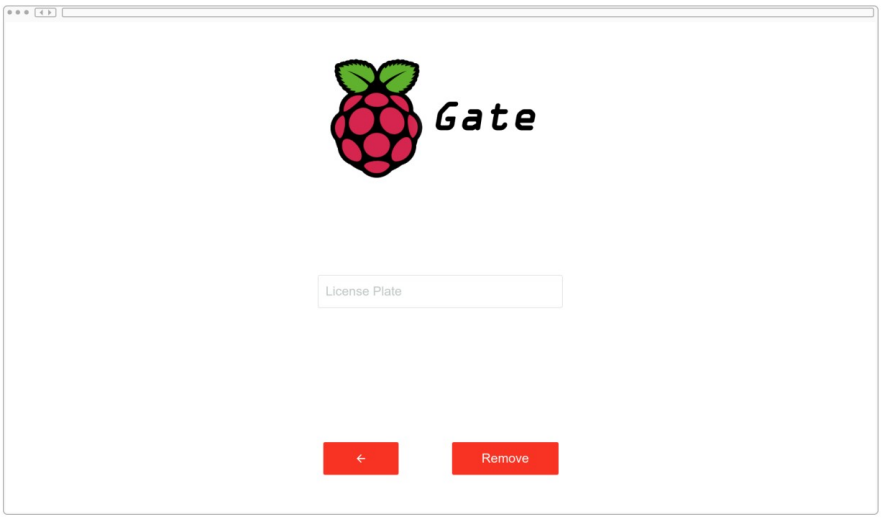


Figure E.35: Remove Plate Screen User Interface

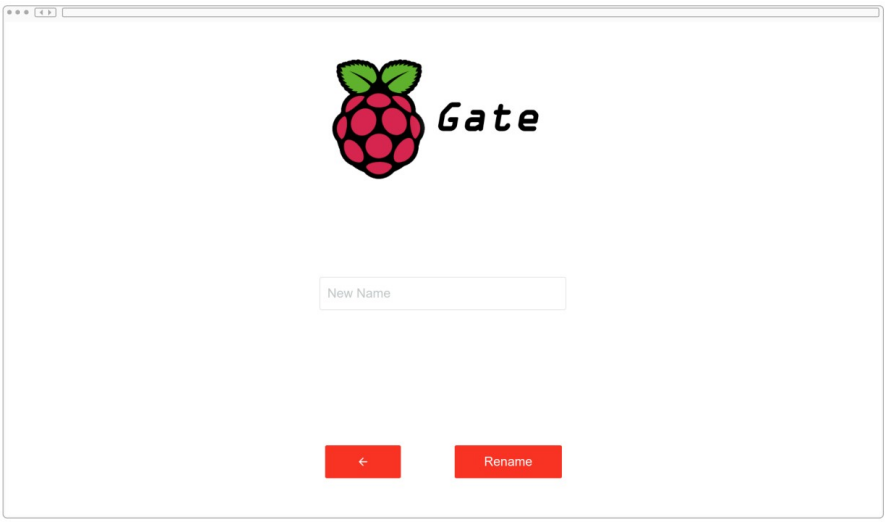


Figure E.36: Rename Screen User Interface

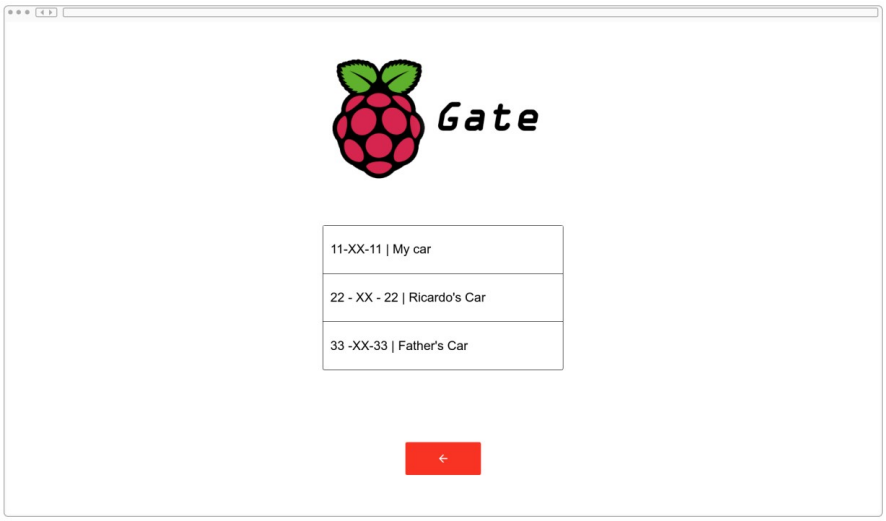


Figure E.37: Whitelist Screen User Interface

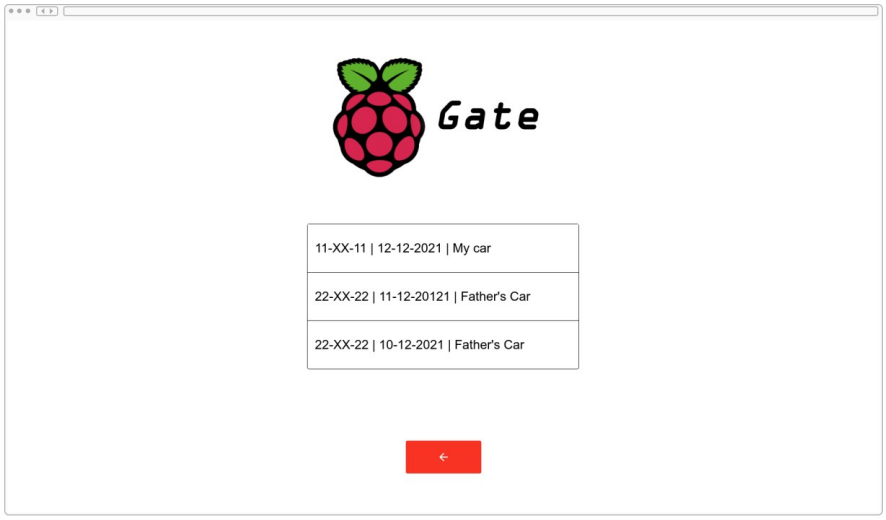


Figure E.38: Entries Screen User Interface

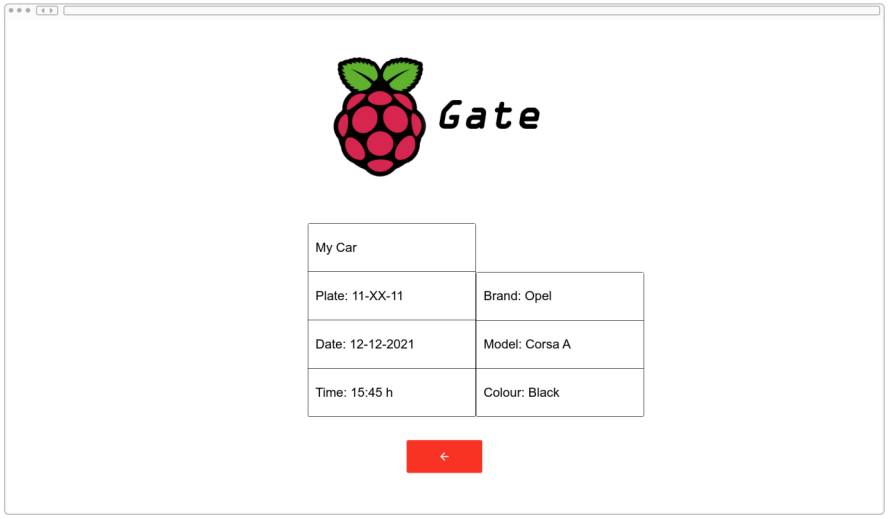


Figure E.39: Entry Detail Screen User Interface

When the user tries to add a plate into a PIgate and the car does not exist in the database, the web app will request some more info about that car, like the brand, model, and colour. The description parameter is only visible for the PIgate, meaning the same car can have two different descriptions. This allows that every PIgate user can label the car with freedom, being easier to distinguish between cars.

By clicking on an entry, the app will show its details. Time, brand, model, and colour will be displayed for the user, being even easier to identify the entry.

All the design was done having in mind that mobile users also will be accessing the web app, that's why it's all focused in the middle of the page. The objective was to keep it simple without sacrificing style.

E.6.2 Data Formats

Data	Data Format
Whitelist	car_t (Object that have the plate and description parameters)
Entry	entry_t (Object that have the plate, date and description parameters)
Date	date_t (Object that have the date and time parameters)

Table E.3: Data Formats User Interface

E.6.3 Class Diagram

The web application will have, as previously demonstrated, some screens or pages to work with. They will work with each other. All of them will have variables and functions. Some of them are private, some are not. The following picture shows how all of this is related.

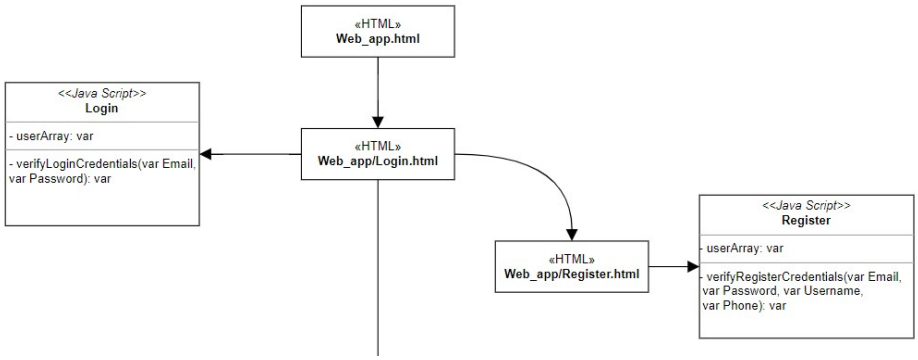


Figure E.40: Class Diagram User Interface Part 1

The application will start on the login page. Login and Register will have a Javascript file that is responsible for the validations of these pages. Then, when the user logs in, he will be redirected to the gate page.

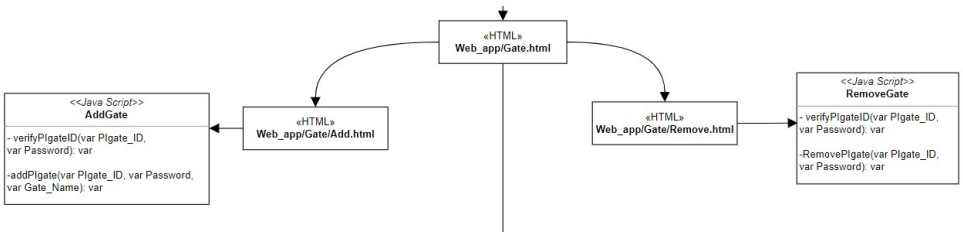


Figure E.41: Class Diagram User Interface Part 2

AddGate and RemoveGate will also have their javascript files, which will be used for verifying the inserted PIgate_IDs' and to add or remove a gate. Finally, when a user chooses a PIgate, he will be redirected to the Gate Option page where will have all the options.

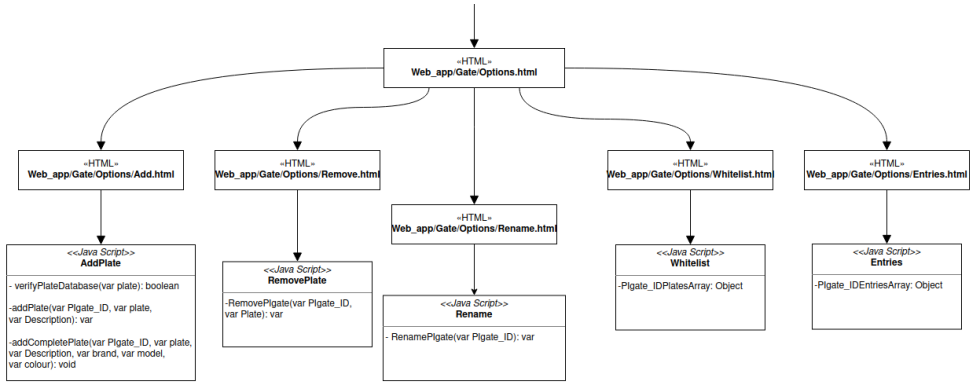


Figure E.42: Class Diagram User Interface Part 3

The AddPlate will have two add functions because, as seen in the GUI section, there will be two possible ways to add a plate. verifyPlateDatabase will verify the if plate is already stored on the database, so the system can present the complete or the simple version of addPlate. Because each whitelist and entry have more than one parameter, their functions return an object which contains all the necessary info.

E.6.4 Flowcharts

To better understand how the system will accomplish all the activities, the web application flowcharts were done. In the following images are presented all designed ones.

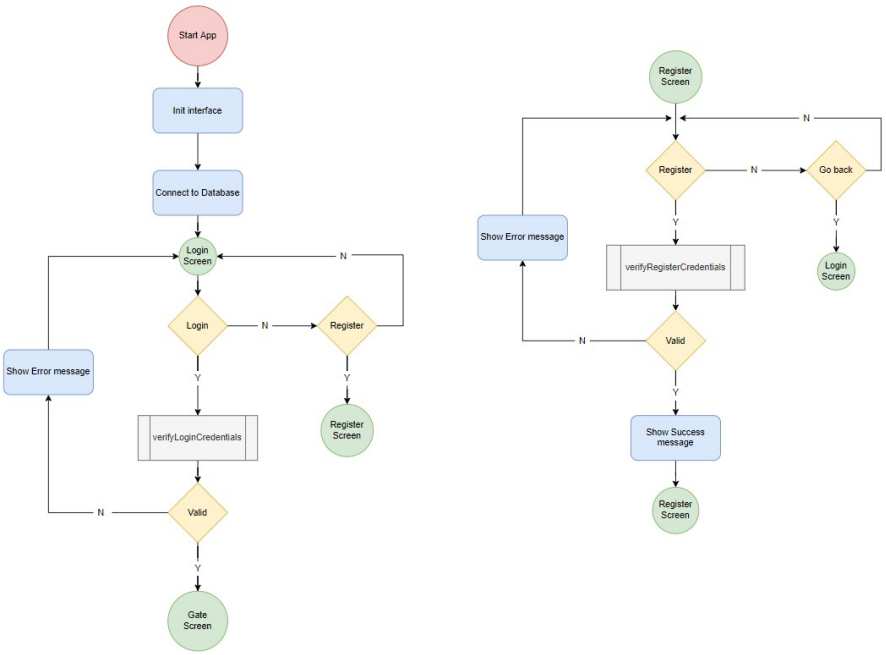


Figure E.43: Login And Register User Interface Flowcharts

Everything starts when the user decides to use the app. Firstly there is an interface initialization, a database connection, and finally the login screen. Error messages will always depend on what error the function behind returns.

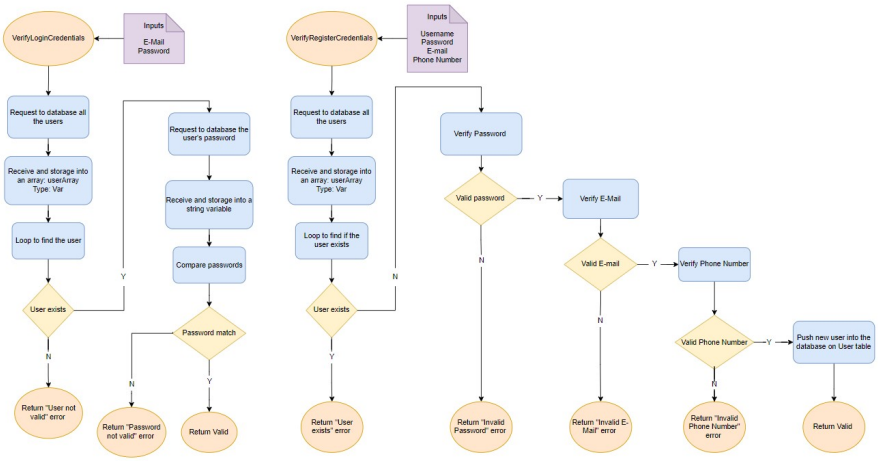


Figure E.44: Login And Register Functions User Interface Flowcharts
110 To Augmented Figure

For example, verifyLoginCredentials and verifyRegisterCredentials are very similar, but with distinguished jobs. They have a lot of possible returns, each one with a different message, so the user can exactly know what error he got.

When a user logs in, the gate screen will appear. There are two possible outputs for this screen.

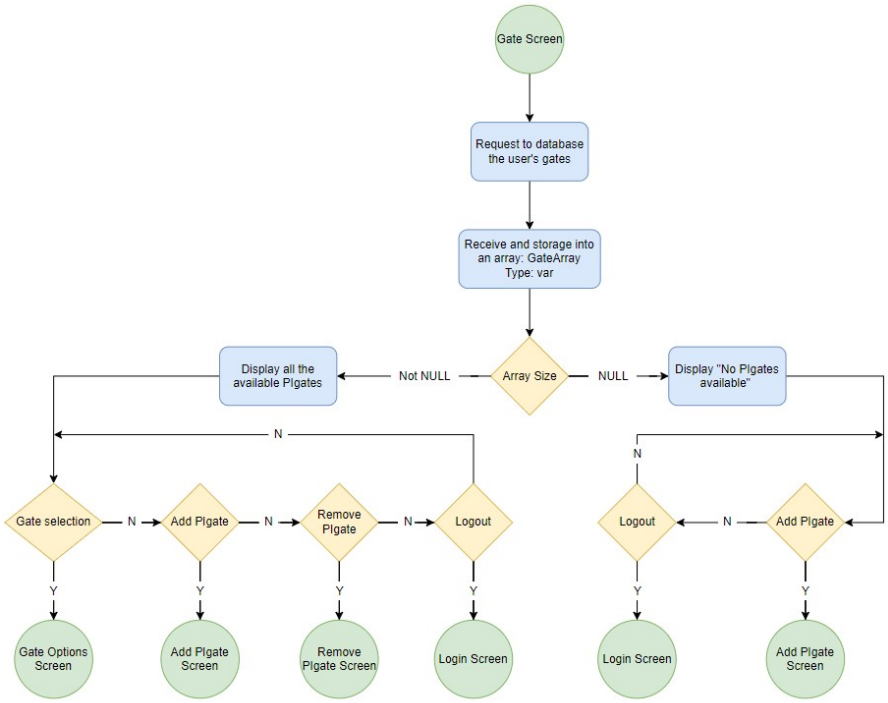


Figure E.45: Gate User Interface Flowchart

The first one happens when the user has at least one PIgate registered. Here he has some options to work with. If the user does not have any PIgate, only the Add PIgate and Logout options will be shown.

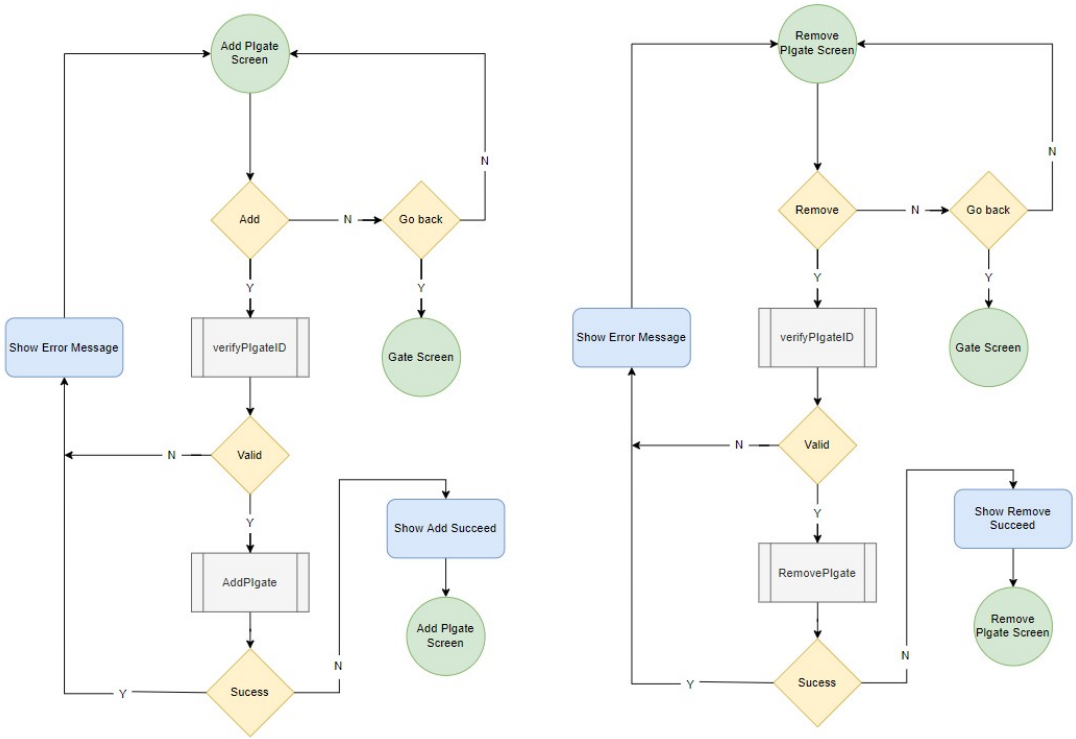


Figure E.46: Add and Remove Gate User Interface Flowcharts
112 To Augmented Figure

The flow charts for Add PIgate and Remove PIGate screens are also quite similar. This

happens because the verifications and the work methodology are the same.

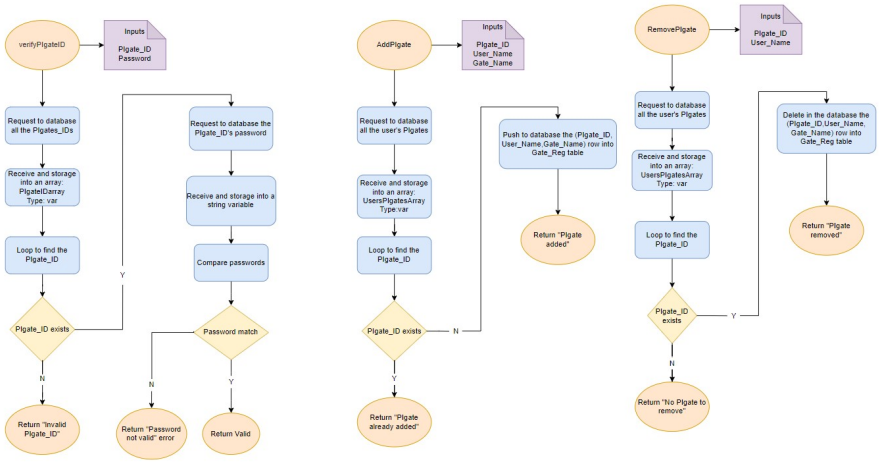


Figure E.47: Add and Remove Gate Functions User Interface Flowcharts

First, they verify the PIgate_ID so it's possible to know if it is a valid one. AddPIgate and removePIgate do exactly what the name says. The possible returns are a valid or invalid add or a valid or invalid remove.

When the user selects a PIgate, the app will pass to the Gate Options screen.

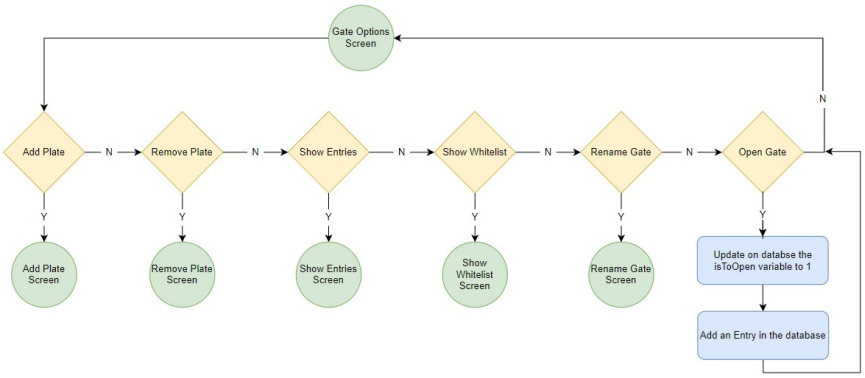


Figure E.48: Gate Options User Interface Flowchart

Basically, this is a menu where it's possible to choose the applicable option.

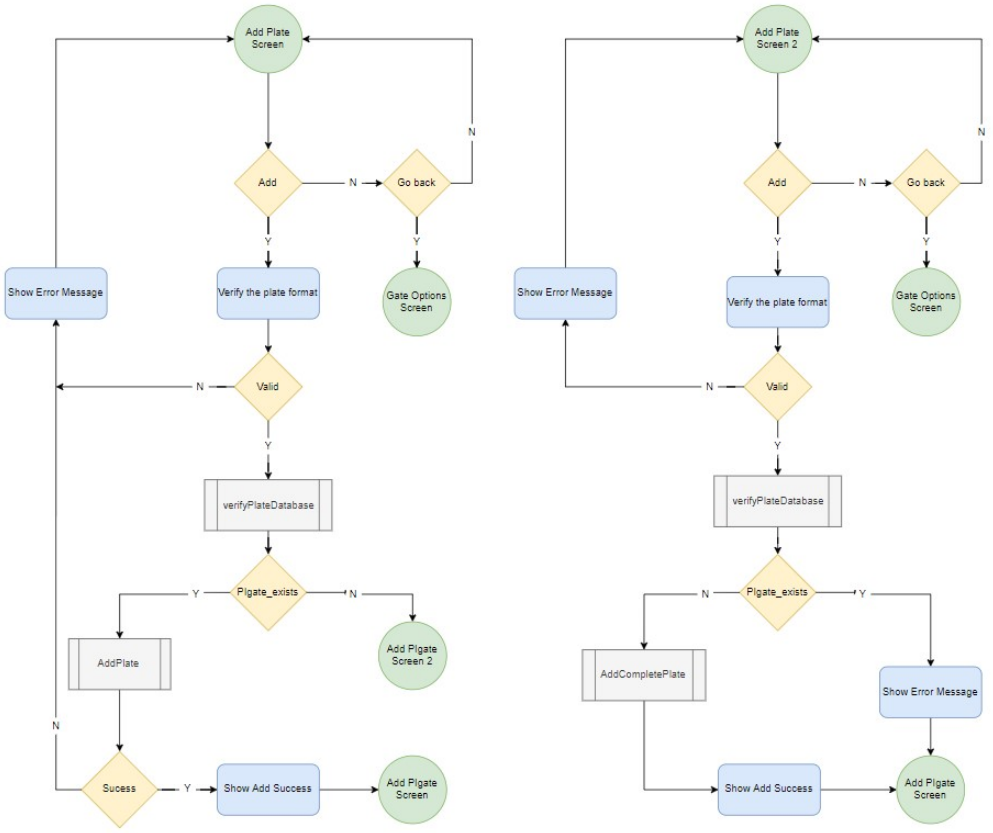


Figure E.49: Add Plate User Interface Flowcharts

On the Add Plate option, there are two possible screens. Mainly, the user will only look at the Add Plate screen. The second one appears when the system does not retain the plate the user wants to add. So it will be presented with some more options.

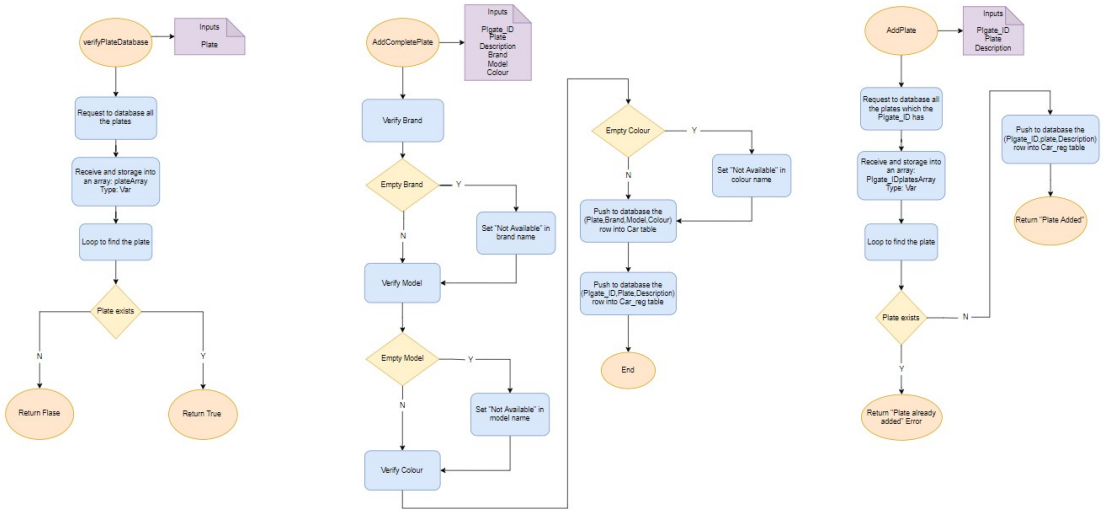


Figure E.50: Add Plate Functions User Interface Flowcharts
114 To Augmented Figure

To complete all of this work, there are three auxiliary functions. verifyPlateDatabase returns true or false, addPlate returns error if the plate is already added or plate added in case of success. The addCompletePlate does not return anything because if the plate is not in the database, it's impossible to have it in the user's PIgate. If the brand, model, and colour were empty, the system will label them as "not available".

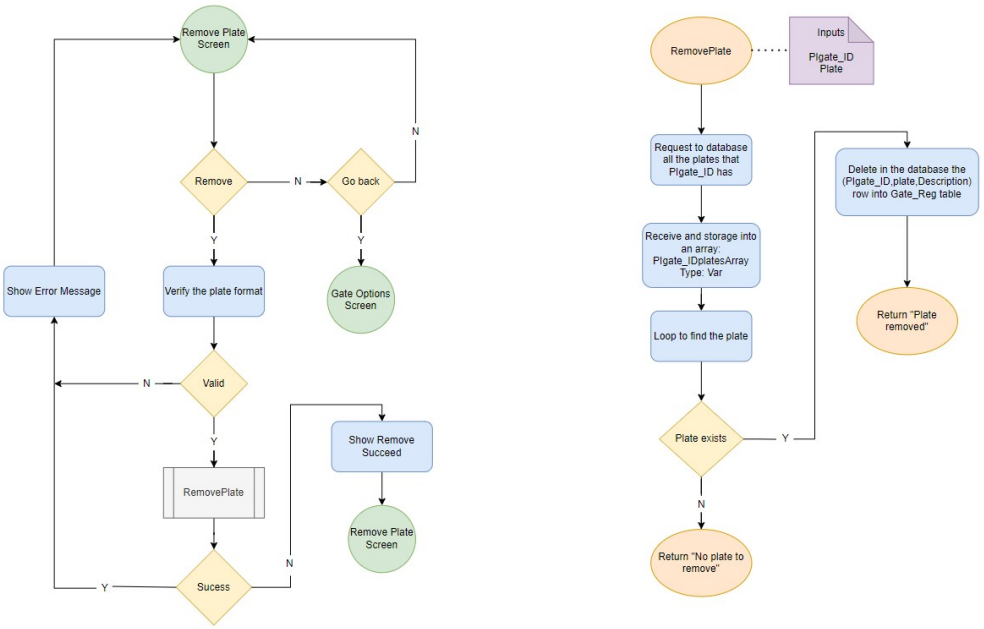


Figure E.51: Remove Plate Functions User Interface Flowcharts

Remove Plate screen works with the same methodology as Remove PIgate screen. Instead of having a PIgate, it is a plate. RemovePlate also is very similar to removePIgate. The outputs are "No plate to remove" in case of error and "Plate removed" in case of success.

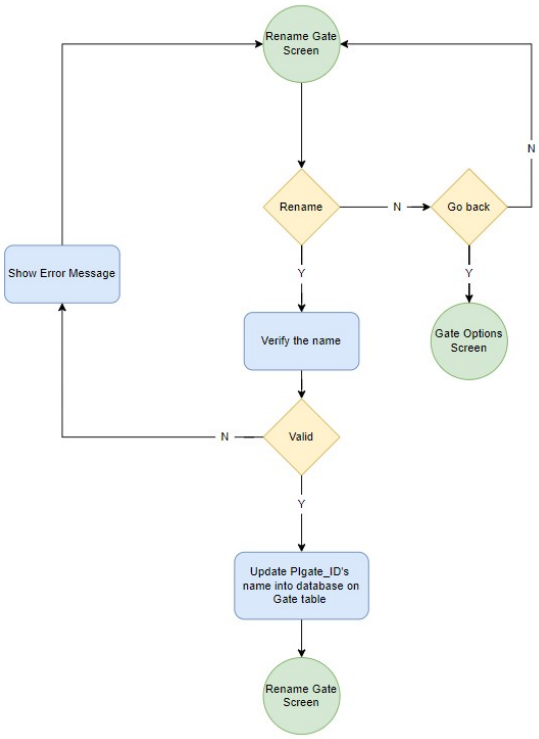


Figure E.52: Rename User Interface Flowcharts

In the Rename Gate screen, it's where it's possible to rename an added PIgate. It is quite simple, so there is no need for auxiliary functions. To change the name the system only sends an update request to the database.

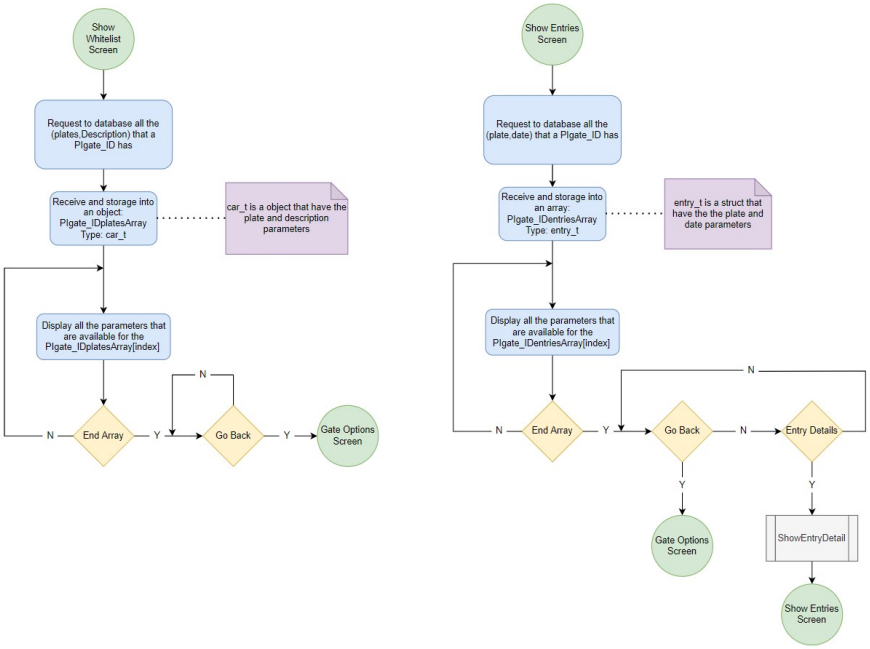


Figure E.53: Whitelist and Entries User Interface Flowcharts
115 To Augmented Figure

Finally, Show Whitelist and Show Entries screens. They are pretty identical in terms of working steps, but they do completely different things, as the name says. After receiving the

request data both will display it to the user. If the user wants to see in detail on entry, he must click on the entry and another screen will show. The following flowchart presents how it is going to work.

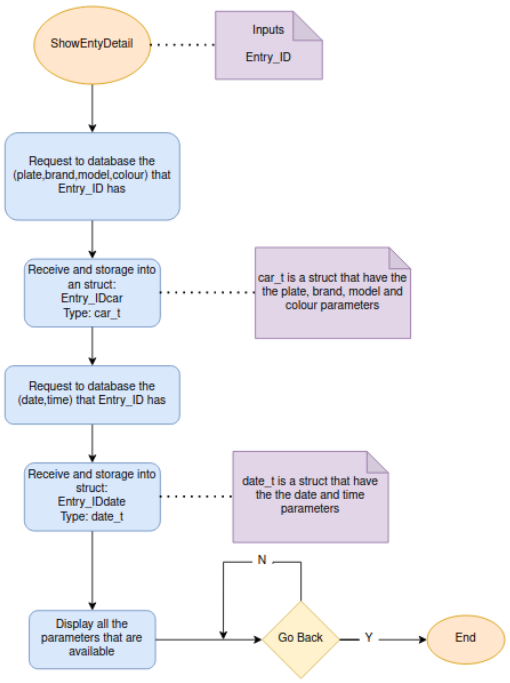


Figure E.54: Detail Entry User Interface Flowchart

E.6.5 Test Cases

Web App: Tests	Expected Results	Real Results
Register a new user	Regist success	
Login a user	Login success	
Add PIgate	Associate the PIgate to the user in case of success	
Remove PIgate	Remove the PIgate association in case of success	
Add Plate	Associate the plate to the gate in case of success	
Remove Plate	Remove the plate association in case of success	
Show White List	Show all the Whitelisted cars	
Show Entries	Show all the done entries	
Rename Gate	Rename the PIgate name	

Table E.4: Web App Test Cases

E.7 Software - Local System

The local system is the most important because it will be the one who's going to activate the gate. The Raspberry PI is the "brain" of all this control. It has a lot of features like High-Definition Multimedia Interface (HDMI) interface, USBs, Ethernet port, etc. but for this project, it isn't all necessary. To have the best performance, it will have a custom Linux system. Linux is, behind Windows Desktop, the most used OS by software developments.

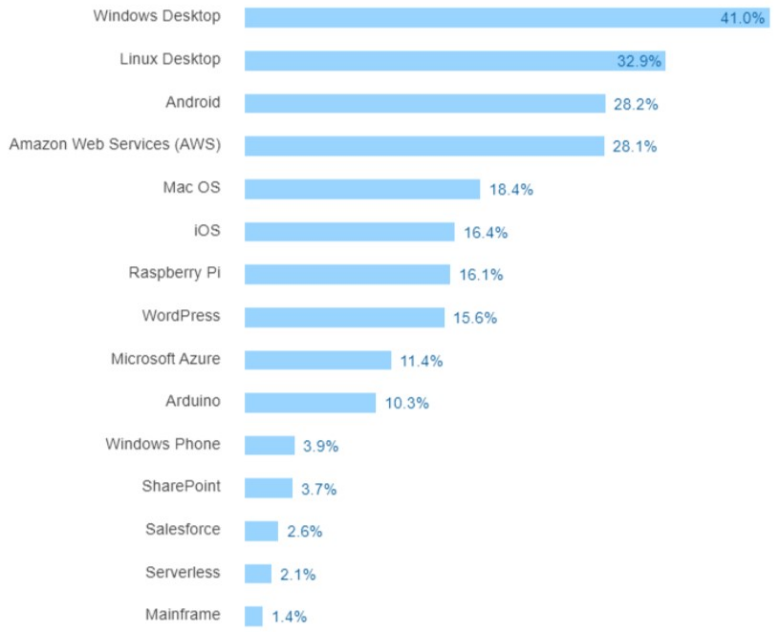


Figure E.55: Used Platform By Developers Survey

It is a constraint, as said in the analysis chapter, but it provides some advantages. Linux is one of the most secure OS. It's light, so even old or non-powerful computers can run it. But, the most important one, is that Linux is open-source which means lots of support.

This section will be described how the local system will work in detail.

E.7.1 Tasks Division

Raspberry has a quad-core ARM processor, although, for this project, there will be used just one core which means that it can only execute one instruction at a time. However, by introducing multitasking, it is possible to have virtual parallelism. At high view, creates the illusion of simultaneous execution of tasks, but, in fact, they are switching among them. The main reason to use multitasking is to deal with complexity. Thus, it's possible to effectively solve the one core problem.

For the PIgate project, there will be some threads that will be responsible to do specific tasks. The following list presents the threads name and what they do.

- **Task Capture_Cut_Image:** This task will be responsible for taking a picture and cutting it. The image is cut because not all the capture area is relevant. Only the part of the image that the plate might be. By doing this, the processing process will be faster.
- **Task Plate_Recognition:** After the image is cut, this task will pick the photo and try to find a plate. This is necessary because the car can have text all over it, and that text is not the desired one. Take the example of a commercial car, like REMAX or some other company.
- **Task Text_Recognition:** Once the previous task recognizes the plate, this task will extract the text from it, returning a string that will be used to compare with the whitelist.
- **Task Plate_Validation:** This task grabs the found plate and compares it with the whitelisted ones. If it is a valid one, it activates the relay to open. If not, just pretend as

if nothing happened and continue the work.

- **Task Update_Plate:** To finish, the Update_Plate task is responsible to update the whitelist. So the Plate_Validation will have always the most recent one. To synchronize these two tasks, a mutex will be used.

E.7.2 Task Priority

From the very first moment that the user turns the PIgate on, the scheduler will be working. The scheduler is the one who controls all the tasks. It's impossible to have control of it although, one thing that is possible to do, is to define priorities. Thus, if a high-priority task wants the processor, the scheduler will be prioritizing it. The following diagram shows the priority distribution.

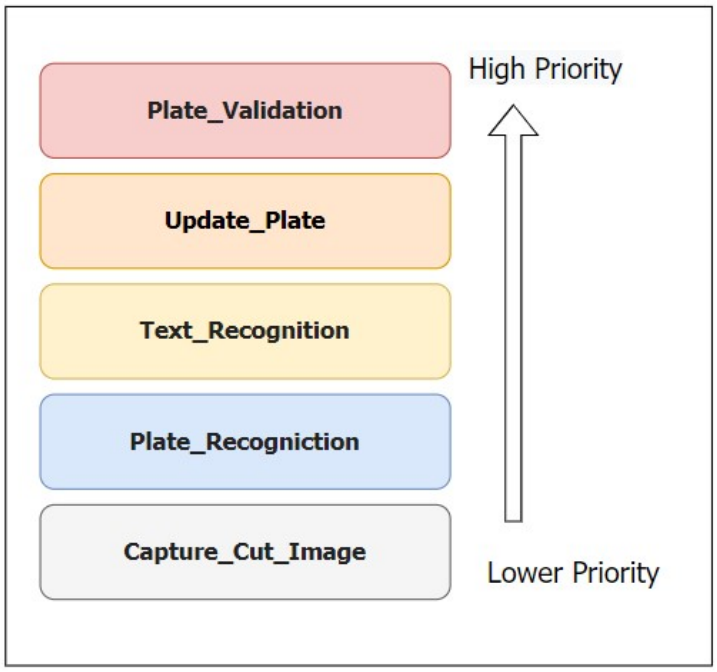


Figure E.56: Tasks Priority

The colour means the priority. Tasks with the same colour mean they have the same priority. Capture_Cut_Image has the lower priority. This happens because it's most important to recognize and analyze a plate instead of taking pictures; The Plate_Validation is the most critical one because it's the one who's going to open the gate.

E.7.3 Daemons

- **Daemon UpdatePlate:** This daemon is responsible to get the most recent whitelist from the database. Because it needs to be done periodically and not every single moment, the daemon will be idle most of the time. When it gets the list, it will be sent to the Update_Plate task by a PIPE.
- **Daemon EntriesDB:** This daemon is responsible to send one entry to the database. To do that, it is always reading the PIPE which Send_Entry writes. When the read succeeds, it cast the data to the database format and sends it.

- **Daemon OpenGateDB:** This daemon triggers an open gate action by the web application. On the gate's options, there is a button that when pressed, writes in a variable on the database. With this information, this daemon can verify that variable. When it changes, it's time to open the gate.
- **Daemon PresenceDetect:** This daemon also triggers the open gate action but only when there is a vehicle on the inside of the gate. The daemon reads the magnetic sensor's data and determines whether there is a vehicle there.

E.7.4 System Overview

After defining all the tasks and daemons, it's important to know how they are going to interact and communicate. By doing a system overview diagram, it's easy to verify the previous topics, without going into too much detail.

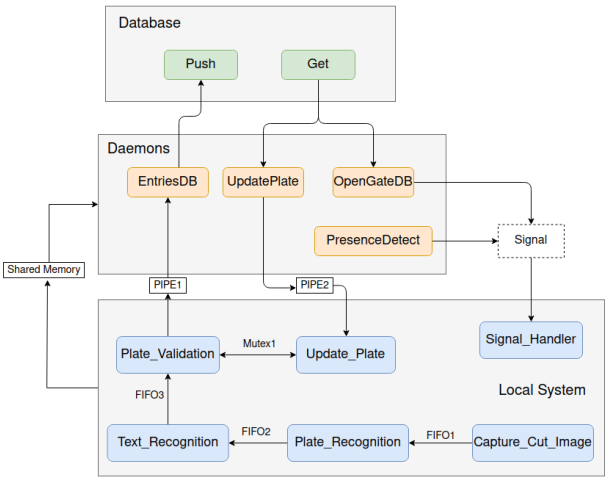


Figure E.57: Local System Overview

On the local system, most of the tasks will communicate by FIFOs because they have different execution times. Imagine that Capture_Cut_Image takes 1 ms to do its job and Plate_Recognition takes 2ms. This means that Plate_Recognition can't keep up with Capture_Cut_Image, so it will be a time that the raspberry memory will be full and the system crashes. To prevent that, FIFOs are used. With it, it's possible to define a limit, and when it is reached, the system just ignores the new data.

Plate_validation and Update_Plate will both access the most recent whitelist. The first to compare with the recognized plate, and the other to update that list. Since both will use this resource, it must have an IPC. The chosen one was the mutex because only these threads will access the recourse.

To communicate with the daemons, there will be used PIPEs because the communication is unidirectional. The EntriesDB daemon just will receive the new entry and the Update_Plate task will just receive the most recent whitelist.

The OpenGateDB and PresenceDetect will communicate by signals. These two daemons will give the order to open the gate, so there is no need to use a PIPE. It was possible to use shared memory but it will only make the local system more complex since it will require also some type of synchronization mechanism.

To make the diagram simpler, some names weren't displayed. Although, it must be described

so the following list shows the none displayed names.

- **PIPE1:** recPlatePipe
- **PIPE2:** EntriesDBPipe
- **FIFO1:** ImagesFifo
- **FIFO2:** PlatesFifo
- **FIFO3:** TextFifo
- **MUTEX1:** UpdatePlatesMutex
- **Shared Memory:** shmPIgateID

E.7.5 Data Formats

Data	Data Format
Capture Image	MAT (OpenCV Struct)

Table E.5: Local System Data Formats

E.7.6 Classes Diagram

The local system will have different models, that have their variables and functions. These models are related to each other. Some use others to accomplish their purpose. With the following class diagram, it's possible to better understand the relationships of all the models.

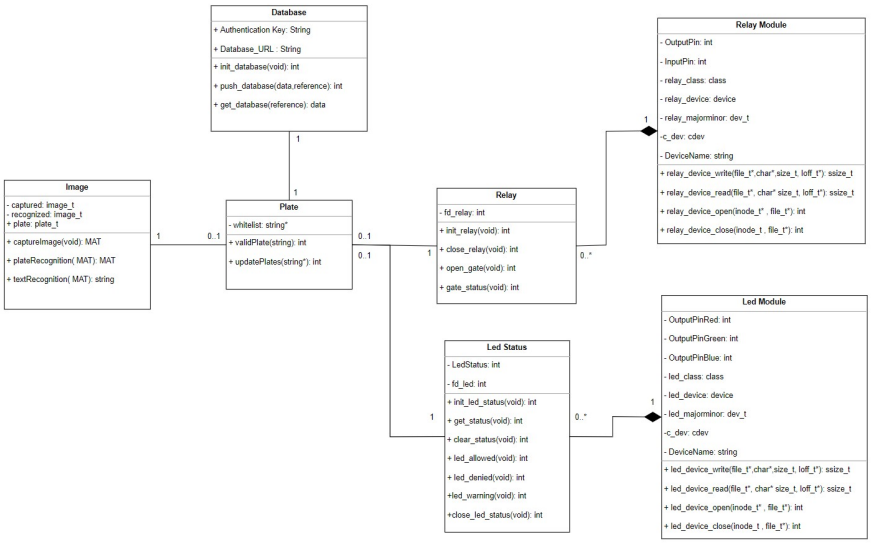


Figure E.58: Local System's Class Diagram
117 To Augmented Figure

The relay and led status modules will have their drivers implemented. Because the magnetic sensor communicates by I2C, the driver implementation is complex, so an Application Programming Interface (API) will be used.

Image module also will use APIs to do its functions. These APIs are provided by openCV and Libcamera. Some examples are the CascadeClassifier, TextRecognitionModel and so on.

For the database module, the firebase SDK should be used to work with. Communication with firebase isn't just as simple as creating a socket and listening. It has, for example, data encryption, which turns the problem to another level.

E.7.7 Flowcharts

In order to better understand how the system will work, the tasks and daemons flowcharts were done. In the following images are presented all designed ones.

Firstly, all the tasks verify if the gate is open. If the gate is open, there is no need for the system to keep its' work, so the tasks can go idle. Before that, every task must clean their FIFOs. To do that, it just ignores the received data until the FIFO is empty.

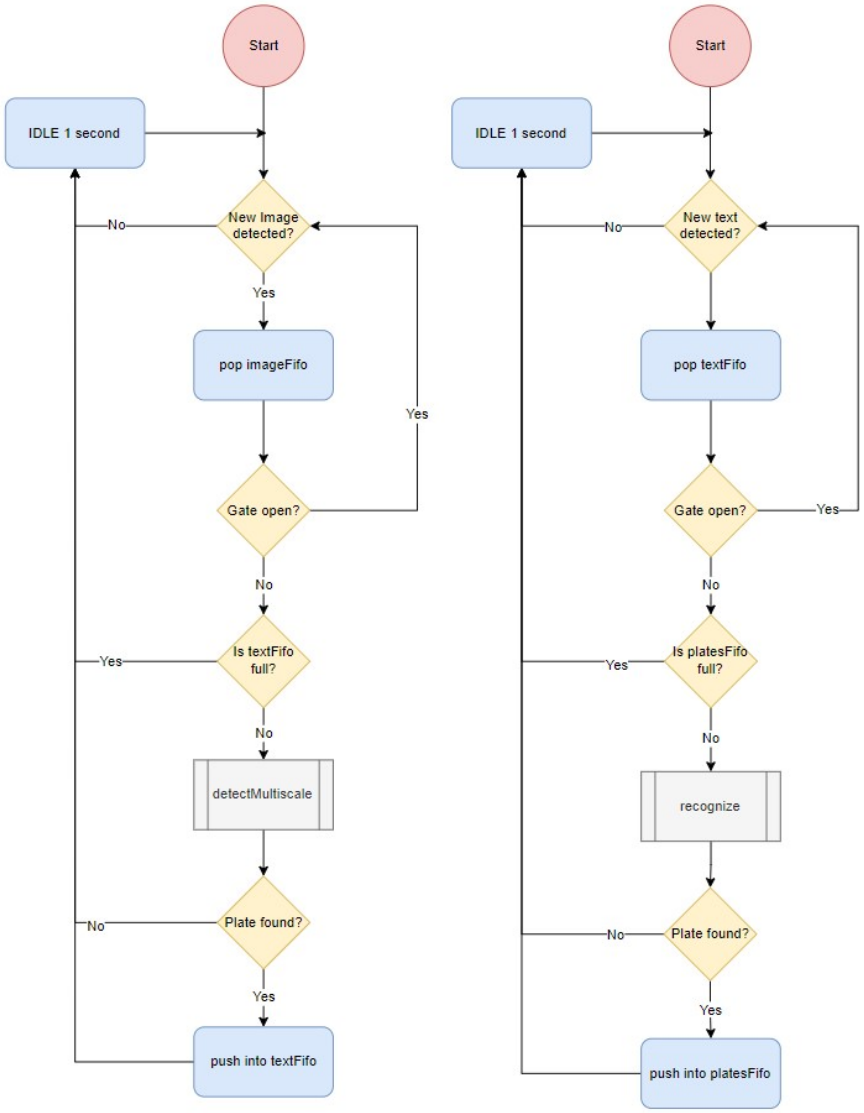


Figure E.59: Plate_recognition and Text_Recognition Flowcharts

The Capture and InputArray are libcamera functions besides detectMultiscale and Recognize, which are OpenCV functions. Because they are functions of these libraries, there is no flowchart for them.

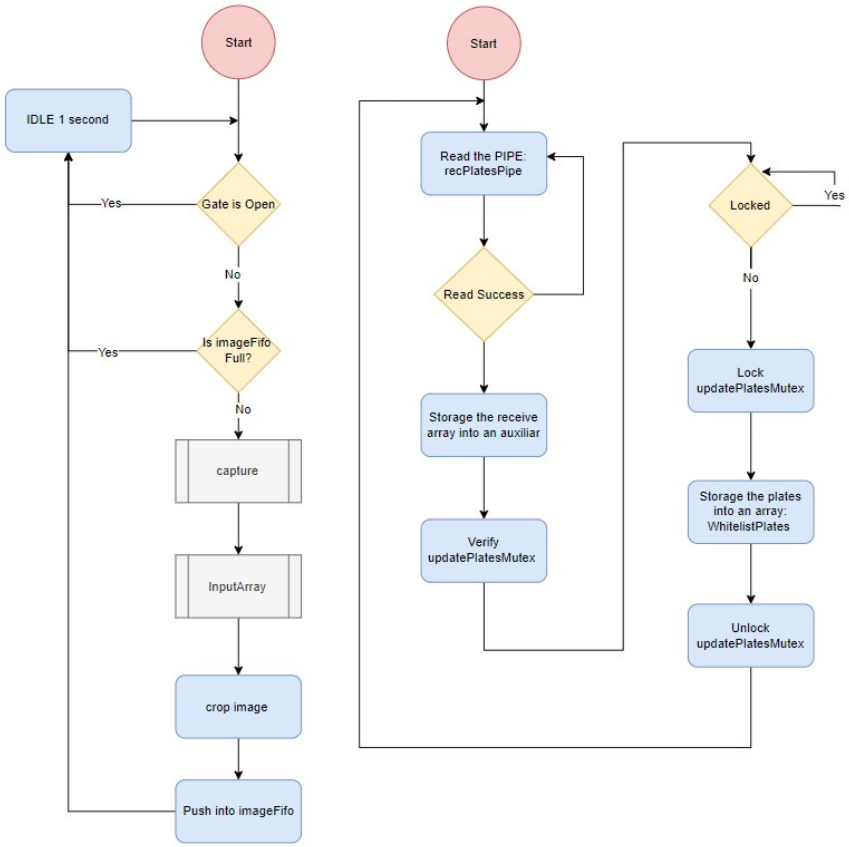


Figure E.60: Capture_Image and Update_Plate Flowcharts

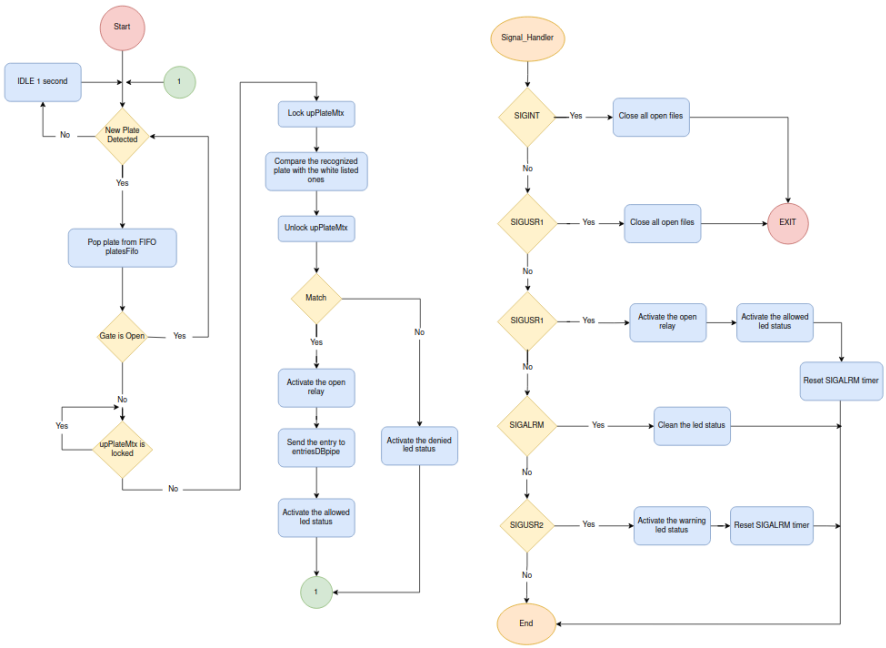


Figure E.61: Plate_validation and Signal_Handler Flowcharts
118 To Augmented Figure

As it's possible to notice, Plate_Validation and Update_Plate share the same resource, so it needs to have an Inter-process communication (IPC) to control. As said, the chosen one was a mutex, UpdatePlatesMutex. When the Update_Plate is updating, the mutex is locked, and the same happens when the Plate_Validation wants to read.

The deadlock scenario is impossible to happen. When the whitelist is updating and the Plate_Validation wants to execute, this task will try to lock the mutex but, because it is already locked, it goes to wait state. The Update_Plate task has the highest priority, next to Plate_Validation. So, the scheduler will execute the Update_Plate task until the end. If Update_Plate were one of the lowest priority tasks, the Ceiling Priority (CP) needed to be used.

The signal_handler, as the name says, will handle all the signals that will be used. The SIGINT is to close all the open files and exit the program. SIGALRM is an alarm clock signal meaning that it will trigger when the timer overflows. It will be used to clean the lead status. The time that will be denied for SIGALRM will be 30 seconds because it's less than the daemons cycle time, thus if the error is solved, the status is already reset. SIGUSR1 and SIGUSR2 are user signals and they will be used to daemons communicate with the main process.

The following flowcharts are for the daemons. They are a crucial part of the system, so they also need to be designed.

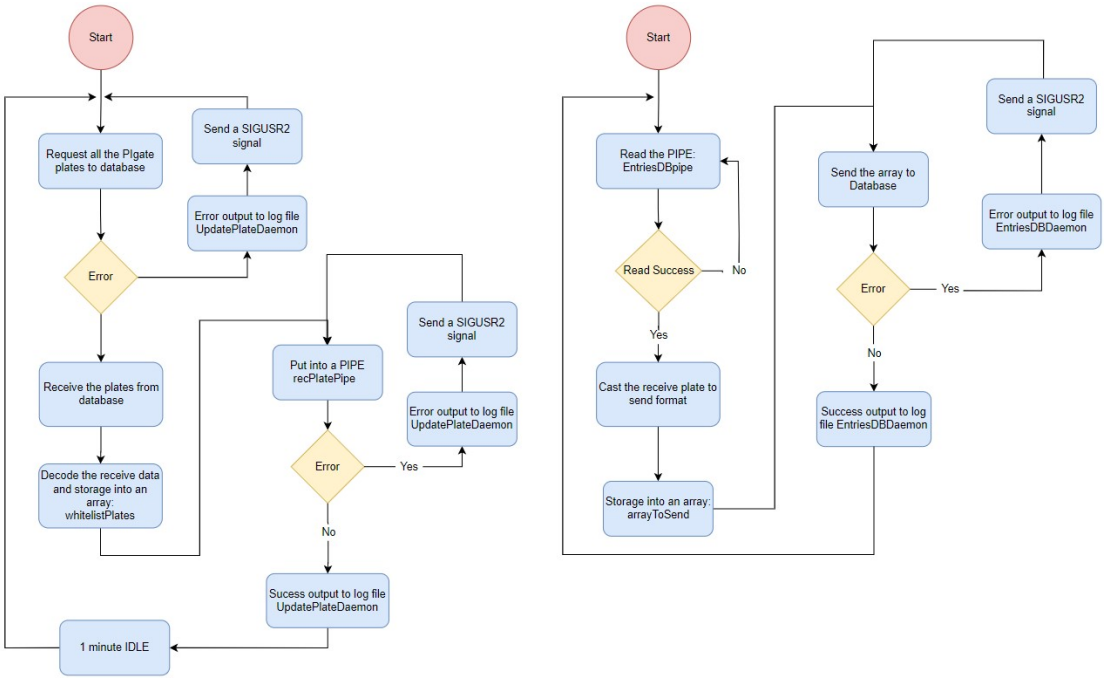


Figure E.62: UpdatePlate and EntriesDB Flowcharts

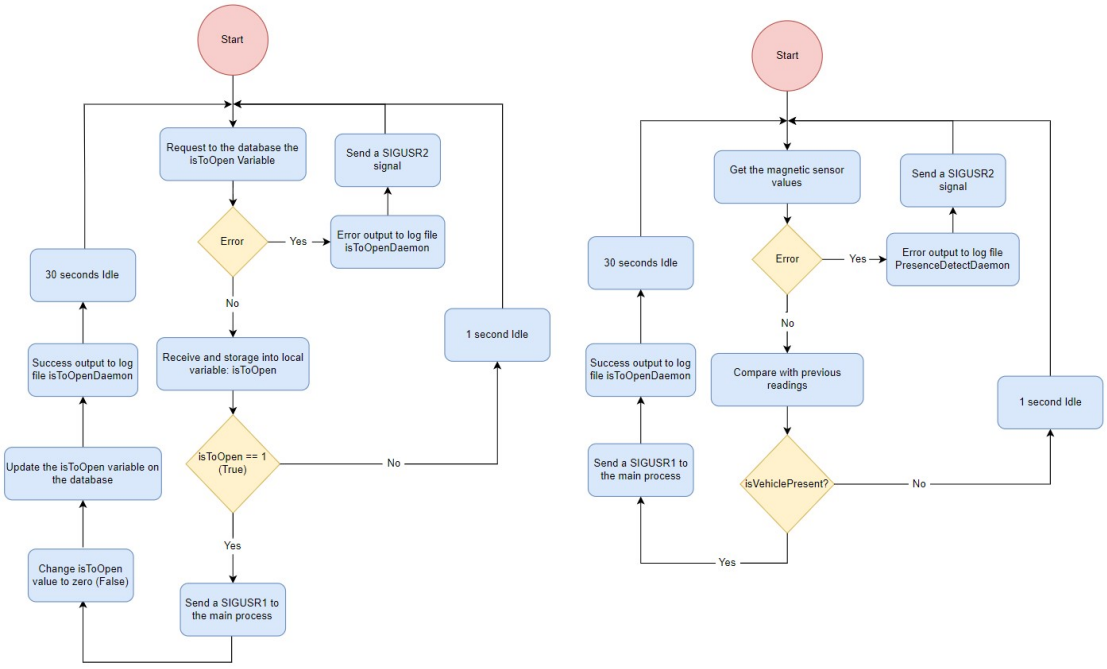


Figure E.63: OpenGateDB and PresenceDetect Flowchart
120 To Augmented Figure

As said, to communicate with the tasks will be used PIPEs. UpdatePlate writes on the recPlatePipe the received plates. EntriesDB reads the EntriesDBPipe in order to send the entry to the database.

The OpenGateDB does not send an entry to the database because the web app already does it. On the database, there are no boolean variables, so, in other to detect if it's time to open the gate, it will be use 0 and 1 numbers, where 0 means false and 1 means true.

As said, the daemons also will use signals to communicate with the main process. There are different types of signals for different types of actions. SIGUSR1 will be used to open the gate and SIGUSR2 to report an error. All daemons' actions will be stored in log files, so if an error occurs, it's easy to know what is wrong.

Led status and Relay will be the interface between the user space and the accorded device driver. All of them have an open and close function to start or finish the device driver.

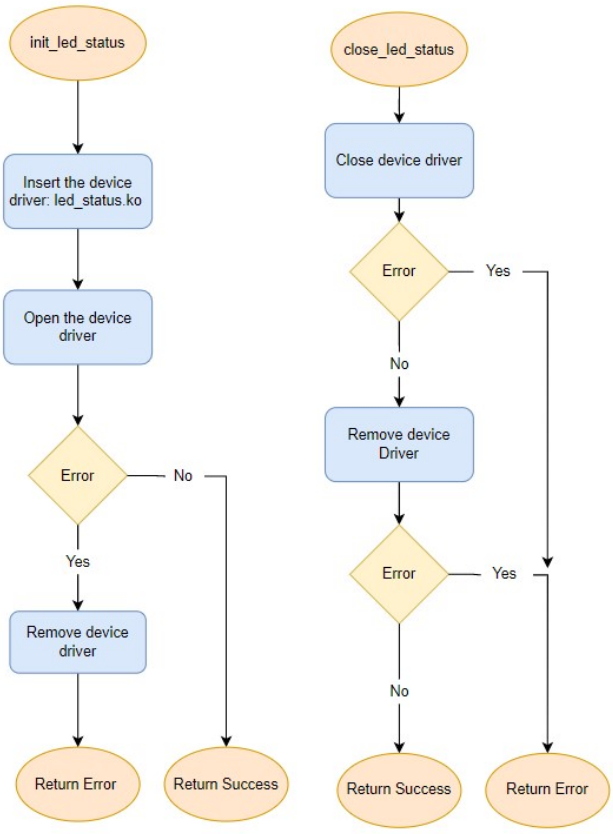


Figure E.64: Led Status: Init and close Flowcharts

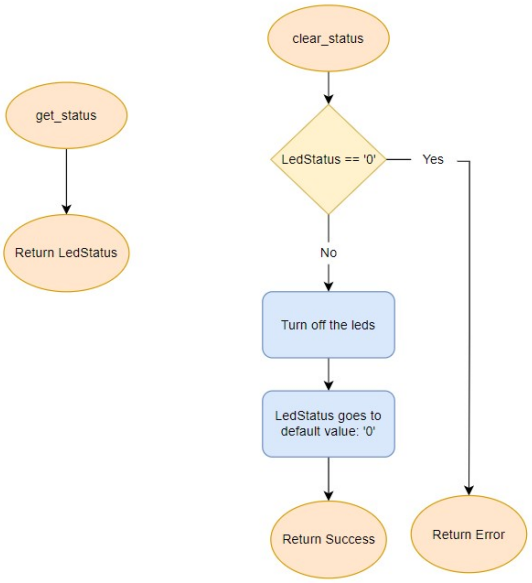


Figure E.65: Led Status: Get and Clear Flowcharts

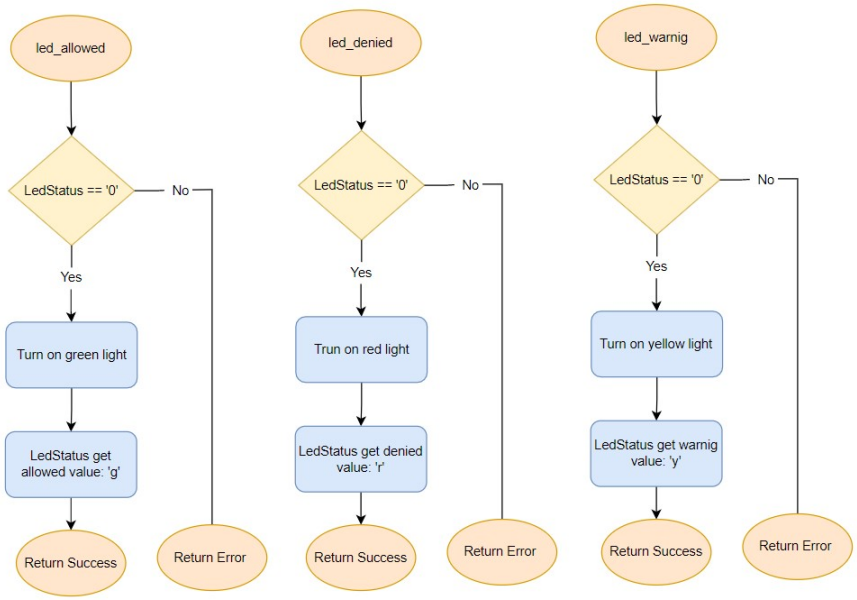


Figure E.66: Led Status: Allowed, Denied and Warning Flowcharts

The Led Status will control the indicator LEDs. For that, it will have 3 functions that will send an order to emit that corresponding colour. After their job, they update the LedStatus variable, so get_status and clear_status can work properly.

LedStatus can have 4 different values. '0' means light OFF. 'y', 'r' or 'g' means that the yellow, red, or green light is ON.

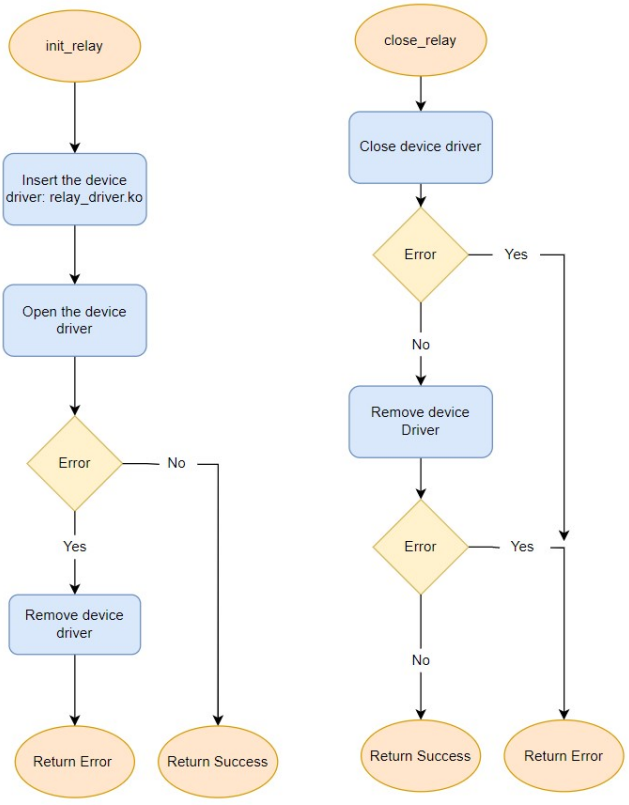


Figure E.67: Led Status: Init and close Flowcharts

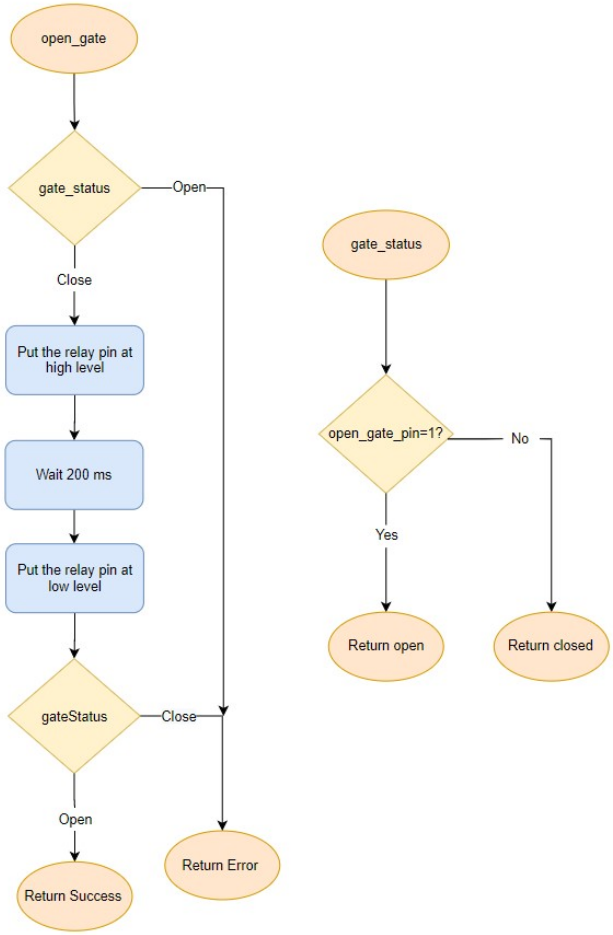


Figure E.68: Led Status: Get and Clear Flowcharts

The Relay module will control the two relays that the PIgate has. One relay is to open the gate, the other is to check if the gate is open or close. To do that, the gate_status function will verify if the gate is open, by checking the voltage on pin 27.

The open_gate function uses the gate_status function to see if the gate is open or not. If it is, it returns immediately because doesn't make sense to open a gate that is already opened. After that check, it puts the output relay pin at a high level, 5 Vols, and after 200 ms put it in low level, GND. Then, it needs to check if the gate is effectively opened. If not, an error message will be output.

E.7.8 Test Cases

Local System: Tests	Expected Results	Real Results
Receive Plates from Database	Receive success	
Send an entry to Database	Database receives the entry	
Allowed Car Tries to Enter Home at day	Gate Open and Green Light	
Allowed Car Tries to Enter Home at night	Gate Open and Green Light Turns ON	
Non-Allowed Car Tries to Enter Home	Red Light Turns ON	
Problem with the system	Yellow Light Light Turns ON and Problem Description In Log File	
Car Tries to Exit home	Gate Open and Green Light	

Table E.6: Local System Test Cases

E.8 Final Tests

After each module test, it's time to verify the system has one. This allows to check the final product and see if all the parts are synchronized. If some of these tests fail, means that the product isn't qualified to sell. Because this product is split into modules, it will become easier to fix any errors that might appear.

Final Tests	Expected Results	Real Results
Allowed Car Tries to Enter Home	The Gate Opens Green Light Turns ON Possible to Check the Entry in User Interface	
Open Gate Now	The Gate Opens Green Light Turns ON Possible to Check the Entry in User Interface	
Car Tries to Exit	Gate Open and Green Light Turns ON	
Add/Remove Allowed Plate	Local system will update the whitelist	
Non-Allowed Car Tries to Enter Home	Red Light Turns ON	
Open Gate with gate controller	Gate Opens	

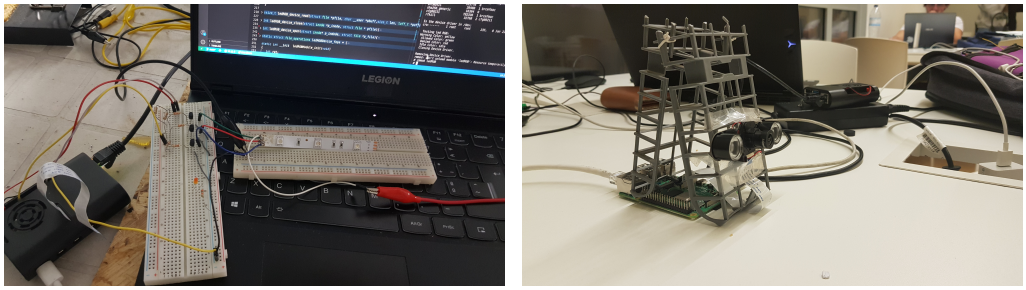
Table E.7: Final Tests

F Implementation

This phase, as mentioned in the design chapter, was supported by git and meld, so all implementation process, buildroot packages and code are available in the PIgate repository.

F.1 Hardware

The hardware implementation was done in two parts. One just with the camera and the other with the Light-Emitting Diode (LED)s and the relay. When everything was validated, the two parts were combined and the project prototype was done.



(a) LEDs and Relay Structure (b) Camera Structure

Figure F.1: Hardware Implementation Parts

The following images show the final prototype. The best material to produce this product is plastic because it is waterprof and it's cheap, but the resource wasn't available. Thus, to save costs, the prototype was done with a paper box, however it is very similar to the designed one.

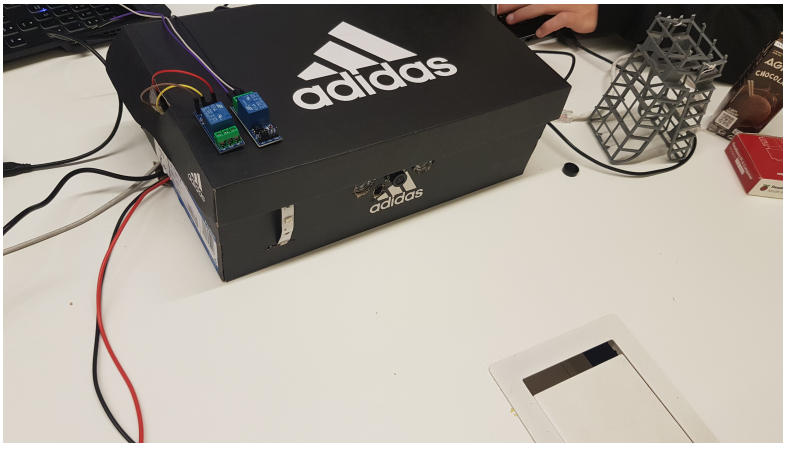
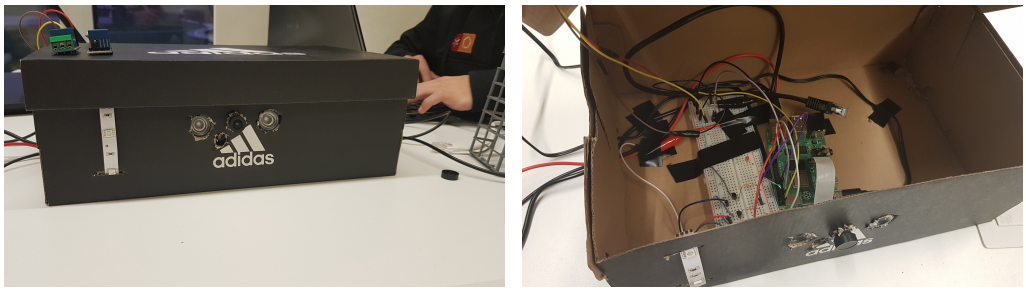


Figure F.2: Upper View



(a) Front View (b) Inside View

Figure F.3: Developed Prototype

F.2 Database

The database implementation, as said in the design chapter, was done in Firebase. For that, a PIgate email has been created, so the database can be only accessed by the PIgate owners. To create a database on Firebase is very simple. First, go to the Firebase website and hit the button "get started". Then select the add project option, which will only require the project name that, in this case, was PIgate. It is also possible to monitor the project with Google Analytics, although it won't be used.

Firebase has lots of services, so after the project creation, the database service had to be selected. The selected database location was Belgium because it was the nearest. After this, the database was successfully created.

Because Firebase is a NoSQL database, the creation of tables isn't mandatory, so adding or removing data can be done freely. However, to have a functional and light database, the tables concept was forced. The following picture shows the final database, with some data already added.

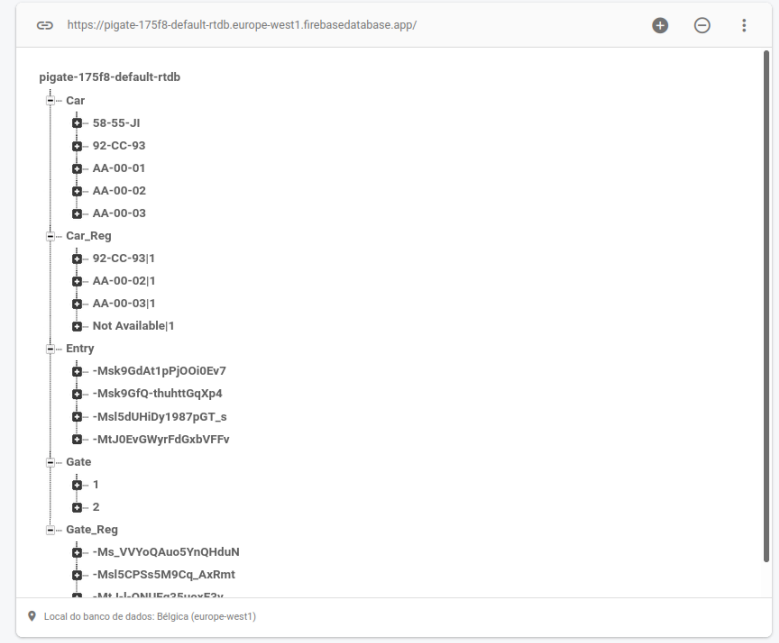


Figure F.4: Database Implementation

F.3 User Interface

All user interface implementation was done in visual studio. To have a better experience with HTML and CSS coding, the HTML CSS Support and HTML Snippets extensions were installed.

For each webapp screen was created an HTML and Java Script (JS) file, so each screen could be implemented individuality. The index and 404 error screens aren't mandatory but they were implemented because it's a good practice.

F.3.1 HTML

Every HTML file was structed as shown in the image bellow.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="/style.css">
  <link rel="shortcut icon" href="/Images/raspberry-pi-icon-17.ico">
  <!-- Page name -->
</head>

<body>
  <div class="container">
    <header>
          <!-- PIGate Logo -->
    </header>

      <main>
        <h1>
            <!-- Page name, ex: Login, Register, ... -->
        </h1>

        <div >
            <!-- Page Content -->
        </div>
        </main>

        <nav>
            <!-- Buttons -->
        </nav>

        <footer>

            <a href="https://www.uminho.pt/PT" target="blank" rel="noopener noreferrer">
                UMinho Website</a>

            <a href="mailto:PIgate.project@gmail.com? Subject=WebApp Help">
                Any Problem? Contact us </a>

        </footer>

    </div>
    <!-- Scripts Declaration -->
</body>

</html>
```

Code 1: HTML File Structure

The HTML file starts with the head section, where the wanted style sheets, the web page logo and the web page name are defined. The meta tags are automatically generated and they are used to define metadata about the HTML document. The body section is where the document's

body is defined. It is divided into two sections, one for the page content and the other for the page scripts. The page content can be split into four subsections, each one with a specific purpose. The header is used to display the PIgate logo. The main, to display the page name and the page content. The nav, where the needed buttons are, and the footer, where is possible to find some links to get support and more information.

F.3.2 Java Script

Every JS file was structed as shown in the image bellow.

```
// Import firebase libraries

// -------- Init Initialization -------------//

const firebaseConfig = {
    /* Firebase configuration */
  };

const app = initializeApp(firebaseConfig);
const auth = getAuth();
const database = getDatabase();
const dbRef = ref(database);

  //Variables initialization

// --------- End initialization ----------- //

// -------- Funtions -------------//

/* GET URL PARAMETER FUNCTION */

function getParameterByName(name, url) {
    /* ... */
}

/* REPORT MESSAGE FUNCTION */
function sendReportMsg(msg,color){
    /* ... */
}

/* VERIFY LOGIN DELAY FUNTION*/
let delayInMilliseconds = 900;
//Set a 900 mseconds delay so the auth can update
setTimeout(function() {
    /* ... */
  }, delayInMilliseconds);


// -------- Funtions -------------//

document.getElementById( /* Button */ ).onclick = function(){

    /**
     * Code Here
    */
}
```

Code 2: Java Script (JS) File Structure

The JS files are not so linear as the HTML ones. They were adapted for each case, so some functions aren't present in some files. The getParameterByName function is one example. Although, all start importing the needed firebase libraries and initializing the firebase variables. The setTimeout function is defined with 900 ms because it's the needed time to synchronize with the database. After that time, the application does all the needed verifications to check if it is a valid request. These verifications are a valid user and valid id, in some cases. If yes, the page

will "unlock" and all the content will be present. If not, the user will be redirected to the login screen. The last function is the sendReportMsg which just displays a feedback message to the user with a wanted colour. Events were used to know when the user clicks on one button. In JS, to verify when a button is pressed, there are use the onClick method, which can trigger a function to execute.

F.3.3 CSS

To make the WebApp GUI, the CSS was used. Every HTML file have the same CSS file associated because the GUI for every page is very similar. There are generic and specific styles. The generic ones are applied to all the HTML files in the same way, like the body, header, footer, nav, etc. styles. The specific ones need to be referenced in the HTML file, and they can overlap the generic ones. This property was very useful because sometimes the generic wasn't fully correct.

There were created styles for the buttons, inputs, feedback messages, scroll content and all the referred page content subsections.

When looking at the body style, it's possible to notice that it has *Visibility = hidden*. This happens to prevent an unauthorized user from getting in. Firstly the program needs to do the all the needed verification and then, if all went correct, the program will automatically change the visibility. Like this example are others, for example, the add plate inputs. There are hidden options but, if the added plate aren't registered in the database, the program will automatically turn them visible.

F.3.4 Hosting

After the implementation and the needed tests, the WebApp was ready to go online. Firebase also provides a hosting service and it was the one used. Firstly the hosting service needed to be activated. For that, on the Firebase main page, the hosting option was selected and pressed the first steps option. Then firebase shows what needs to be installed and how to deploy the WebApp. After that, the hosting service is available. To deploy the developed WebApp, it's mandatory to be logged in on the firebase project account, so the *firebase login* command was used on terminal. Then was used the *firebase init hosting* command and to complete all the proccess, the *firebase deploy –only hosting* was used. The following pictures shows the output of these commands.

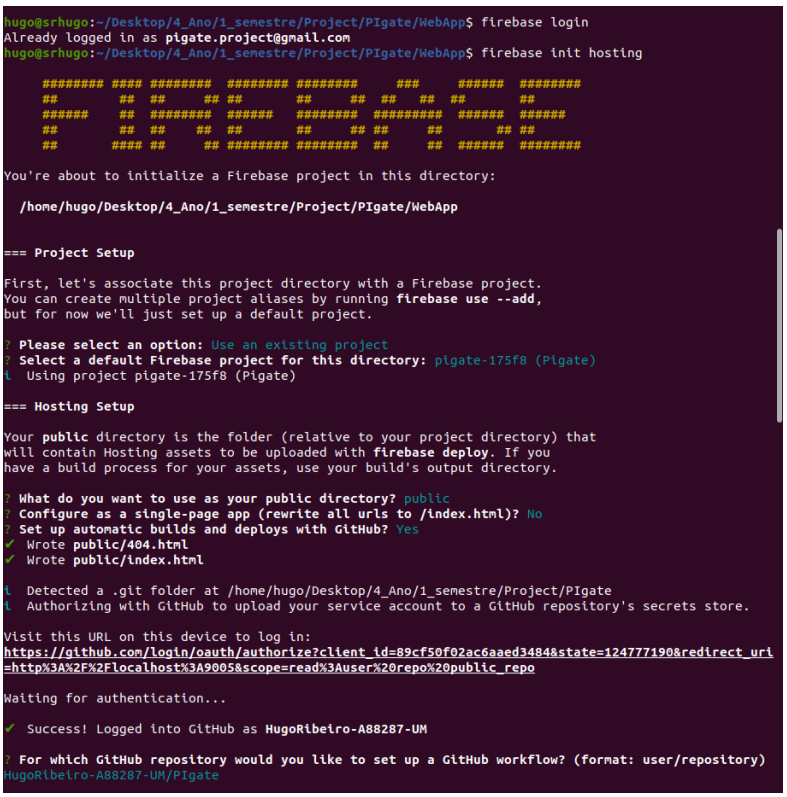


Figure F.5: *firebase deploy –only hosting* command output

Figure F.6: *firebase init hosting* command output

When the *Deploy Complete* message shows, the WebApp is online and ready to be used. The link to access it is: PIgate WebApp.

F.4 Local System

Due to the complexity of the local system, it will be split into several parts, each one explaining the done implementation. The following topics are organized by timeline, so the buildroot packages was the first part done and the threads were the last to be complete. The auxiliary function and the Makefiles parts were done within the others developed.

One thing that was designed and wasn't implemented was the magnetic sensor. Due to problems with OpenCV and the Tesseract, the time to implement this module was very short and it became barely impossible to do. For that reason, this part of the project wasn't implemented, however, it isn't a crucial piece. The system will be working perfectly without the module.

F.4.1 Buildroot Packages

As said in the design chapter, the builroot tool will be used to create a custom Linux system. Firstly, in the buildroot project folder, was created a default 32-bit image for raspberry PI 4. After that, some minor changes were done, like changing the compiler to the *glibc* one, increasing the root file system size to 3000Mb, activateUART, setting a password to log in and selecting the *Dynamic using devtmpfs + eudev* in /dev management. Then the following packages were added so all the modules can operate without problems.

Camera Module

- BR2_PACKAGE_FFMPEG
- BR2_PACKAGE_GSTREAMER1
  - BR2_PACKAGE_GST1_PLUGINS_BASE
  - BR2_PACKAGE_GST1_PLUGINS_GOOD_JPEG
  - BR2_PACKAGE_GST1_PLUGINS_GOOD_PNG
  - BR2_PACKAGE_GST1_PLUGINS_BAD_PLUGIN_CAMERABIN2
- BR2_PACKAGE_V4L2GRAB
- BR2_PACKAGE_V4L2LOOPBACK
  - BR2_PACKAGE_V4L2LOOPBACK_UTILS
- BR2_PACKAGE_RPI_FIRMWARE
  - BR2_PACKAGE_RPI_FIRMWARE_VARIANT_PI4
  - BR2_PACKAGE_RPI_FIRMWARE_X
  - BR2_PACKAGE_RPI_FIRMWARE_INSTALL_DTB_OVERLAYS
  - BR2_PACKAGE_RPI_FIRMWARE_INSTALL_VCDBG
- BR2_PACKAGE_RPI_USERLAND
- BR2_PACKAGE_PYTHON3
  - BR2_PACKAGE_PYTHON3_PY_PYC
  - BR2_PACKAGE_PYTHON_PYYAML
  - BR2_PACKAGE_PYTHON_PLY
  - BR2_PACKAGE_PYTHON_JINJA2
- BR2_PACKAGE_JPEG
- BR2_PACKAGE_BCM2835
- BR2_PACKAGE_LIBV4L
- BR2_PACKAGE_LIBV4L_UTILS
- All packages in BR2_PACKAGE_LIBCAMERA
- BR2_PACKAGE_LIBEVENT

Most of these packages are to give support to other modules. For example, the libcamera package needs the three python packages to work. The V4L2 is the acronym of Video For Linux 2 and it provides the abstraction for camera hardware. The BCM2835 package provides access to General Purpose Input/Output (GPIO) and other IO functions on the Broadcom BCM 2835 chip, allowing access to the GPIO pins on the 26 pin IDE plug on the Raspberry PI board. The reason for FFMPEG installation was that with this tool it's possible to take a photo from the terminal, so the camera could be validated.

Database communication

- BR2_PACKAGE_PYTHON_AENUM
- BR2_PACKAGE_PYTHON_ASN1CRYPTO
- BR2_PACKAGE_PYTHON_CCHARDET
- BR2_PACKAGE_PYTHON_CERTIFI
- BR2_PACKAGE_PYTHON_CFFI
- BR2_PACKAGE_PYTHON_CRYPTOGRAPHY
- BR2_PACKAGE_PYTHON_DEPRECATED
- BR2_PACKAGE_PYTHON_GOOGLEAPIS_COMMON_PROTOS
- BR2_PACKAGE_PYTHON_HTTPLIB2
- BR2_PACKAGE_PYTHON_IDNA
- BR2_PACKAGE_PYTHON_JWCRYPTO
- BR2_PACKAGE_PYTHON_JWT
- BR2_PACKAGE_PYTHON_OAUTH2CLIENT
- BR2_PACKAGE_PYTHON_PIP
- BR2_PACKAGE_PYTHON_PROTOBUF
- BR2_PACKAGE_PYTHON_PYASN1
- BR2_PACKAGE_PYTHON_PYASN1_MODULES
- BR2_PACKAGE_PYTHON_PYPARSING
- BR2_PACKAGE_PYTHON_REQUESTS
- BR2_PACKAGE_PYTHON_REQUESTS_TOOLBELT
- BR2_PACKAGE_PYTHON_RSA
- BR2_PACKAGE_PYTHON_SETUPTOOLS
- BR2_PACKAGE_PYTHON_SIX
- BR2_PACKAGE_PYTHON_URLLIB3
- BR2_PACKAGE_PYTHON_WRAPT
- BR2_PACKAGE_NTP_NTPD

All the referred python packages are needed for the well functioning of pyrebase. The last package is the time synchronization daemon, so it will be keeping the local system date synchronized, which will be crucial when the entries are sent.

Wi-Fi

- BR2_PACKAGE_CRDA
- BR2_PACKAGE_IW
- BR2_PACKAGE_IPROUTE2
- BR2_PACKAGE_WIRELESS_REGDB
- BR2_PACKAGE_WIRELESS_TOOLS
- BR2_PACKAGE_WIRELESS_TOOLS_LIB
- BR2_PACKAGE_WPA_SUPPLICANT
- BR2_PACKAGE_WPA_SUPPLICANT_NL80211
- BR2_PACKAGE_WPA_SUPPLICANT_AUTOSCAN
- BR2_PACKAGE_WPA_SUPPLICANT_CLI
- BR2_PACKAGE_WPA_SUPPLICANT_WPA_CLIENT_SO
- BR2_PACKAGE_WPA_SUPPLICANT_PASSPHRASE
- BR2_PACKAGE_RPI_WIFI_FIRMWARE
- BR2_PACKAGE_NANO
- BR2_PACKAGE_NANO_TINY

The IW and IPROUTE2 packages are to verify network connections, so by typing *iwconfig* is possible to verify if the raspberry is connected to the wanted Wi-Fi. The nano package is to verify the content of .config, .txt, etc. files. The remaining ones are, effectually, to provide the Wi-Fi capability.

Plate Recognition

- All packages in BR2_PACKAGE_OPENCV3 except the Python, Build Tests and Build Performance Tests Packages
- BR2_PACKAGE_TESSERACT_OCR
- BR2_PACKAGE_TESSERACT_OCR_LANG_ENG

The openCV package is used for plate recognition and Tesseract is used for text recognition. It is important to note that OpenCV must be built after Tesseract in order to use Tesseract within OpenCV. It is also important to note that the text module needed to have Optical character recognition (OCR) in OpenCV is not part of the core of OpenCV and therefore, OpenCV must be compiled from source including the Opencv_contrib modules. Buildroot does not include these modules, as such, it needed to be manually cross-compiled and built including OpenCV and Opencv_contrib for the target architecture and using buildroot's toolchain tools.

After adding all the referred packages and building the new image, other additional steps needed to be done, so all can be initialized properly. Going to the config.txt file, which is in this directory *./output/images/rpi-firmware/*, the following lines of code were added or, in the gpu_men case, replaced.

```
#========================
gpu_mem_256=512
gpu_mem_512=512
gpu_mem_1024=512

# fixes rpi (3B, 3B+, 3A+, 4B and Zero W) ttyAMA0 serial console
dtoverlay=disable-bt

dtparam=i2c_arm=on
dtparam=i2c_vc=on
dtoverlay=vc4-fkms-v3d
start_x=1

enable_uart=1
#========================
```

Code 3: Config.txt file

To make possible setup the needed files for the Wi-Fi communication, the post-build.sh was modified. This script, which is present in *./board/raspberrypi4/* directory, can insert automatically the wanted files in the target. Because the Wi-Fi settings won't change this can be done, thus the following lines were added in this script.

```
#========================
cp board/raspberrypi4/interfaces ${TARGET_DIR}/etc/network/interfaces
cp board/raspberrypi4/inittab ${TARGET_DIR}/etc/inittab
cp board/raspberrypi4/wpa_supplicant.conf ${TARGET_DIR}/etc/wpa_supplicant.conf
#========================
```

Code 4: Post-build.sh file

Important to notice that these files were copied from the generated target to the post-build.sh directory. So, in each file, the following code was added.

```
#========================
# Load Wifi driver
::sysinit:/sbin/modprobe brcmfmac
#========================
```

Code 5: Inittab.sh file

```
#========================
auto wlan0
iface wlan0 inet dhcp
  pre-up wpa_supplicant -B -D wext -i wlan0 -c /etc/wpa_supplicant.conf
  post-down killall -q wpa_supplicant
  wait-delay 15
#========================
```

Code 6: Interfaces file

```
#========================
network={
  ssid= NETWORK NAME
  psk= NETWORK PASSWORD
}
#========================
```

Code 7: Wpa_supplicant.conf file

Finally, to start the program in boot time, a script was created and added to the init.d folder. This script must be the last to be executed so the script name must have the highest index number. To chosen number was 70.

```
#!/bin/sh

#Execute the PIgate program
cd /etc/PIgate/
/etc/PIgate/PIgateProgram.elf
```

Code 8: S70InitPIgate.sh

F.4.2 Firebase Module

The firebase module is constituted of three files, the firebase.h, firebase.c and firebase.py. The python file is needed to communicate with the firebase database because pyrebase only has its API for python, android and javascript.

In the python file are four functions: sendEntry, getPlates, checkIsToOpen and getPIgates. The functions prototypes are the following.

```

"""
 * @brief Sends an Entry to the database
 *
 * @param PIgate_ID
 * @param Plate
 * @return -EINVAL: if and error occurs ;
 *          Otherwise SUCCESS
 *
"""
def sendEntry(PIgate_ID, Plate):


"""
 * @brief Receive all the existent plates in the database
 *
 * @return -EINVAL: if and error occurs ;
 *          Otherwise an array with all received plates
 *
"""
def getPlates(PIgate_ID):

"""
 * @brief Checks if the isToOpen variavel in ON. If yes, this function
 *  automactly changes that value to OFF
 *
 * @param PIgate_ID
 * @return -EINVAL: if and error occurs ;
 *          Otherwise the received isToOpen variavel
 *
"""
def checkIsToOpen(PIgate_ID):

"""
 * @brief Receive all the existent PIgates in the database
 *
 * @return -EINVAL: if and error occurs ;
 *          Otherwise an array with all received PIgates
 *
"""
def getPIgates():
```

Code 9: firebase.py

The firebase.h and firebase.c use the python functions. To be able to communicate between .c and .py files, the C/Python API was used. The developed firebase c module has similar functions as the python one, although these functions have all the needed steps to communicate with a .py file. The firebase header files are present in the following picture.

```
/**
```

```
 * @file firebase.h
 * @author PIgate
 * @brief Firebase communication Module
 * @version 0.1
 * @date 2022-01-31
 *
 * @copyright Copyright (c) 2022
 *
 */
#ifndef FIREBASE_H
#define FIREBASE_H

/**
 * @brief Sends an Entry to the database
 *
 * @param PIgate_ID
 * @param Plate
 * @return -EXIT_FAILURE: if and error occurs ;
 *          Otherwise EXIT_SUCCESS
 *
 */
int sendEntry(const char* PIgate_ID, const char* Plate);


/**
 * @brief Requests and Receive all the existent plates in the database.
 * Also it put all the received plates into the recPlatePIPE
 *
 * @return -EXIT_FAILURE: if and error occurs ;
 *          Otherwise EXIT_SUCCESS
 */
int receivePlates(const char* PIgate_ID);


/**
 * @brief Verifies if the user wants to open the gate by the WebApp
 *
 * @param PIgate_ID
 * @return -EXIT_FAILURE if and error occurs ,
 *          0 to Not Open the Gate and
 *          1 to Open the Gate ;
 */
int isToOpen(const char* PIgate_ID);


/**
 * @brief Checks if the given PIgate_ID exists in the Database
 *
 * @param PIgate_ID
 * @return -EXIT_FAILURE: if and error occurs ;
 *          Otherwise EXIT_SUCCESS
 */
int validPIgate(const char* PIgate_ID);

#endif //FIREBASE_H
```

Code 10: firebase.h

All of these functions have the same sequence. Firstly they try to get the python file reference. After that, the *PyDict_GetItemString* function is called to get a pointer to the wanted python function. Then the python function is used with the *PyObject_CallFunction* function, which receives as arguments the python function pointer, the data type and the python function parameters. For example, to use the sendEntry python function, the .c file should have: *PyObject_CallFunction(pFunc,"ss",PIgate_ID,Plate)* , where "ss" means PIgate_ID and Plate are strings. This function returns the same result as the called python function, so each .c function does its work properly.

It's important to notice, as explained in the header file, the receivePlates function also puts

the received plates in recPlatePIPE, so all the process is optimized. Because it's the updatePlates daemon that uses this function, there is no problem doing it. Although the first process calling this function must be the one who is going to communicate with the parent one, otherwise, the program will be compromised.

F.4.3 Relay Device Driver

The relay implementation can be split in two files, one for the kernel space and the other for the user space. They communicate like any other device, with open, close, read and write functions, so the relayModule.c, which is the kernel space file, needs to implement these callbacks. The following code shows the done implementation for the write and read.

```
ssize_t relay_device_write(struct file *pfile, const char __user *pbuff, size_t len, loff_t *
    off) {
  struct GpioRegisters *pdev;

  pr_alert("%s: called (%u)\n",__FUNCTION__,len);


  if(unlikely(pfile->private_data == NULL))
    return -EFAULT;

  pdev = (struct GpioRegisters *)pfile->private_data;

  pr_alert("%s: register value is 0x%x\n",__FUNCTION__,(1<<(relayOutput %32)));

  if (pbuff[0]=='0')
    s_pGpioRegisters->GPCLR[relayOutput / 32] = (1 << (relayOutput % 32));
  else
    s_pGpioRegisters->GPSET[relayOutput / 32] = (1 << (relayOutput % 32));

  return len;
}

ssize_t relay_device_read(struct file *pfile, char __user *pbuff,size_t len, loff_t *poffset)
    {
  pr_alert("%s: called (%u)\n",__FUNCTION__,len);

  if(unlikely(pfile->private_data == NULL))
    return -EFAULT;

  char buffer[2];
  int i = gpio_get_value(relayInput);
  sprintf(buffer, "%d", i);

  copy_to_user(pbuff, buffer, 1);
  printk(KERN_INFO "PIN -> %d\n", i);

  return len;

}
```

Code 11: relayModule.c

To turn on the relay, the write function analyses the received buffer. If the first letter was a "O" (Open), the relay will be powered off, otherwise, the relay will be powered because of the relay configuration. To read the relayInput status, the *gpio_get_value* function is used. Every action in this module result in a feedback message, so if some error occurs, it becomes easy to find the problem.

Also is important the mention that in the code initialization, there is an ioremap request, in other to get the pointer to the GPIO structure. For that reason, before exiting the module, the memory address that was previously requested needs to be set free, so the iounmap function is used.

The relay.c file, which is the user interface module, defines some functions that provide the abstraction for the communication between these two spaces. The header file, relay.h, was implemented as shown below.

```c
/**
 * @file relay.h
 * @author PIgate
 * @brief Relay Module
 * @version 0.1
 * @date 2022-01-31
 *
 * @copyright Copyright (c) 2022
 */
#ifndef RELAY_H
#define RELAY_H

/**
 * @brief Possible Gate Status
 *
 */
typedef enum {gateOpen = 0, gateClose} gateStatus_t;

/**
 * @brief Initialize the relay Module. This function must be called before any other.
 *
 * @return -EXIT_FAILURE if an error occurs ; Otherwise EXIT_SUCCESS
 */
int initRelay(void);

/**
 * @brief Deactivate the relay Module.
 *
 */
void remRelay(void);

/**
 * @brief Activate the relay to open the gate.
 * To do that, this function creates a pulse that will be used to trigger the gate.
 *
 * @return -EXIT_FAILURE if an error occurs ; Otherwise EXIT_SUCCESS
 */
int openGate(void);

/**
 * @brief Get the Gate Status object
 *
 * @return -EXIT_FAILURE if an error occurs ; Otherwise the gate Status
 */
int getGateStatus(void);


#endif //RELAY_H
```

Code 12: relay.h

This module must be d before its use because it needs to instantiate the relay device driver. If this function isn't called, all openGate and getGateStatus won't work and they will return -EXIT_FAILURE. Also, there are present feedback messages if something went wrong.

F.4.4 LED RGB Device Driver

To control the RGB LEDs was used the same strategy as before. There are the ledModule.c, which is defined in the kernel space, and the ledRGB.h and ledRGB.c, which are in the userspace. Starting with the ledModule, the first thing that was done was to understand how the PWM works, on the raspberry. Going to the BCM2711 datasheet, which is the chip used in raspberry pi 4, is possible to see the following image in chapter 8.

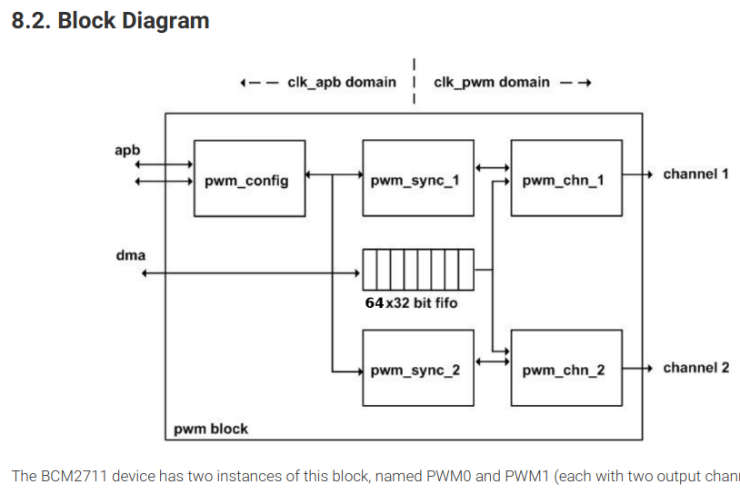


Figure F.7: BCM2711 PWM Block Diagram

Is possible to notice that BCM2711 has two PWMs, each one with two channels. Watching the GPIO assignment, it's possible to notice that only the PWM0 channels are available in the board pins, so it's only possible to have two PWM outputs unless implemented in software. So, because the RGB leds required 3 control pins, the blue colour will be controled as digital output.

To control the PWM it's needed the PWM registers. The following code show the create structs to access that registers. Notice that all of them are volatile because these registers can be change without any visible code action, like other threads running.

```
1
2  volatile struct pwmRegisters
3  {
4    uint32_t CTL;
5    uint32_t STA;
6    uint32_t DMAC;
7    uint32_t reserved0;
8    uint32_t RNG1;
9    uint32_t DAT1;
10   uint32_t FIF1;
11   uint32_t reserved1;
12   uint32_t RNG2;
13   uint32_t DAT2;
14 } *s_pPwm0Registers;
15
16 volatile struct S_PWM_CTL {
17   unsigned PWEN1 : 1;
18   unsigned MODE1 : 1;
19   unsigned RPTL1 : 1;
20   unsigned SBIT1 : 1;
21   unsigned POLA1 : 1;
22   unsigned USEF1 : 1;
23   unsigned CLRF  : 1;
24   unsigned MSEN1 : 1;
25   unsigned PWEN2 : 1;
26   unsigned MODE2 : 1;
27   unsigned RPTL2 : 1;
28   unsigned SBIT2 : 1;
29   unsigned POLA2 : 1;
30   unsigned USEF2 : 1;
31   unsigned Reserved1 : 1;
32   unsigned MSEN2 : 1;
33   unsigned Reserved2 : 16;
34 } *pwm0_ctl;
35
36 volatile struct S_PWM_STA {
37   unsigned FULL1 : 1;
38   unsigned EMPT1 : 1;
```

```
  unsigned WERR1 : 1;
  unsigned RERR1 : 1;
  unsigned GAPO1 : 1;
  unsigned GAPO2 : 1;
  unsigned Reserved1 : 2;
  unsigned BERR : 1;
  unsigned STA1 : 1;
  unsigned STA2 : 1;
  unsigned Reserved2 : 21;
}*pwm0_sta;
```

Code 13: PWM Structs

In the device driver initialization, there are four moments. The first one is the device driver creation. The second is the blue pin, PWM registers and PWM clock assignment, which is similar to the relay one, with the ioremap function. The third is the PWM frequency definition. It was defined as 1kHz and it won't change during the program. The last one is when the PWM duty cycles and the blue colour are defined.

Going to the frequency dentition, it starts with the clock-killing and disabling the PWM. It's important to pay attention when modifying clock registers because they can affect the whole system. These registers are protected with a password which is 5A, so to kill the clock, for example, the value that the clock control register must have is:

```
   /* Kill clock */
  (s_pPwmClkRegisters+PWMCLK_CNTL) = 0x5A000020;
```

Code 14: Kill Clock Command

After that, the divisor is calculated respecting the following formula.

$$f = \frac{source\_frequency}{DIVI}$$

Figure F.8: Frequency Calculation Formula

Finally, the GPIO pins are assigned with the PWM mode and the PWM characteristics are defined.

To define the duty cycle, firstly the PWM is disabled. Then the range and the data are set accordingly to the received parameters. So if the wanted duty cycle is 55%, the range can be 1000 and the data 550. After defining it, there is an error check. If some error flag is ON, the software must clean it, by writing one to that register. To complete, the PWM is enabled again.

This module has an internal state, pwmState variable, which is changed when the userspace module wants, otherwise it will be always idle. To change the PWM state the write function must be used and the received buffer must have the wanted colour. For example, if the user wants a red colour, the pBuff must receive "red". If the user wants an unavailable colour, the module will display the idle colour and set an idle state. To get the pwmState variable, the user must use the read function. The buffer should be an int type.

The ledRGB.c uses this device driver to display the program status. There is 4 possible status. Allowed, denied, warning and idle. Each status represents a colour. Green, red, yellow and light blue respectably. The led RGB header is present in the next code.

```
/**
 * @file ledRGB.h
 * @author PIgate
 * @brief ledRGB Module
```

```
 * @version 0.1
 * @date 2022-01-31
 *
 * @copyright Copyright (c) 2022
 */
#ifndef LEDRGB_H
#define LEDRGB_H

/**
 * @brief Possible LED status
 *
 */
typedef enum {allow = 0 , denied , warning, idle}status_t;

/**
 * @brief Initialize the Module. This functions must be called before any other.
 *
 * @return -EXIT_FAILURE if an error occurs ; Otherwise EXIT_SUCCESS
 */
int initLedRGB(void);

/**
 * @brief Deactivate the ledRGB Module
 *
 */
void remLedRGB(void);

/**
 * @brief Put the RGB leds with the respective state color
 *
 * @param status Program Status
 * @return -EXIT_FAILURE if an error occurs ; Otherwise EXIT_SUCCESS
 */
int ledRGBStatus(status_t status);


/**
 * @brief Get the LedRGB Status object
 *
 * @return -EXIT_FAILURE if an error occurs ; Otherwise returns the status
 */
int getLedRGBStatus(void);

#endif //LEDRGB_H
```

Code 15: ledRGB.h

As in relay.c, this module needs to be firstly initialized. If not, the ledRGBStatus and getLedRGBStatus won't work, giving always a -EXIT_FAILURE error. Also, this module provides error messages, so it's easy to know what might happen.

Is important to note that the relay and the led RGB modules cannot be initialized by daemons because they haven't access to the terminal, which is mandatory.

F.4.5 Daemons

A Daemon is a background process that runs without user input and usually provides some service, either for the system as a whole or for user programs. In this project they will be used to do the communication between the database, like request the most recent whitelist, send an entry to the database and verify the isToOpen flag.

As said, there will be three daemons, so the daemon module have 3 initialization functions. The following code presents the daemon module header file.

```
/**
 * @file daemon.h
 * @author PIgate
```

```
 * @brief Daemons Module
 * @version 0.1
 * @date 2022-01-31
 *
 * @copyright Copyright (c) 2022
 */

#ifndef DAEMON_H
#define DAEMON_H

/**
 * @brief Inits the EntriesDB daemon.
 *  One process just can init the daemon 1 time.
 *
 * @return Daemon's PID in success
 *         EACCES if it isn't the first time
 */
pid_t initDaemonEntriesDB(void);

/**
 * @brief Inits the UpdatePlate daemon
 *  One process just can init the daemon 1 time.
 *
 * @return Daemon's PID in success
 *         EACCES if it isn't the first time
 */
pid_t initDaemonUpdatePlate(void);

/**
 * @brief Inits the OpenGateDB daemon
 *  One process just can init the daemon 1 time.
 *
 * @return Daemon's PID in success
 *         EACCES if it isn't the first time
 */
pid_t initDaemonOpenGateDB(void);

#endif //DAEMON_H
```

Code 16: Daemon.h

One process can only initialize one daemon one time providing some control. The parent process receives, in case of success, the daemon's PID, which is needed, for example, to kill daemons when the program is over. The daemon also keeps the parent PID, so later it can send a signal to the parent process to open the gate, for example. The initialization process is similar for all the cases. The next code shows the initDaemonEntriesDB example.

```
    // create a new process
    pid = fork();

    if (pid < 0)
    { // on error exit
        syslog(LOG_ERR, " Error in fork \n");
        exit(EXIT_FAILURE);
    }

    if (pid > 0)
    {
        //Parent Process
        //Return the Deamons PID
        firstTime = 0;
        return pid;
    }

    //Deamon Process

    sid = setsid(); // create a new session

    if (sid < 0)
    { // on error exit
        syslog(LOG_ERR, "%s\n", "setsid");
```

```
25          exit(EXIT_FAILURE);
26      }
27
28      // make '/' the root directory
29      if (chdir("/") < 0)
30      { // on error exit
31          syslog(LOG_ERR, "%s\n", "chdir");
32          exit(EXIT_FAILURE);
33      }
34      umask(0);
35
36      close(STDIN_FILENO);  // close standard input file descriptor
37      close(STDOUT_FILENO); // close standard output file descriptor
38      close(STDERR_FILENO); // close standard error file descriptor
```

Code 17: Daemon Creation

All the three daemons share the same signal handler function, which only has the SIGTERM definition. It is used to kill the daemons process. To communicate with the parent process are used IPCs, which will be explained later.

F.4.6 Signal Handlers

A signal handler is a function that is called by the target environment when the corresponding signal occurs. These signals are asynchronous, so when they occur, the program suspends its execution until the signal handler returns. Signal handlers must be atomic, otherwise, the system can become slow. In this project signal handlers were used to open a gate, set a warning status or idle and shut down the entire program. As said, all the daemons have the handler for the SIGTERM signal, which kills the process. The main process has an handler for SIGTERM, SIGINT, SIGUSR1, SIGUSR2 and SIGALRM signals.

SIGTERM/SIGINT: They are used to kill all the programs. Firstly the handler kills the daemons. Next up, it closes and destroys the created shared memory. Finally, it removes the relay and ledRGB device drivers.

SIGUSR1: It is used to open the gate. When the user opens the gate by the app, the isToOpen daemon sends this signal to the parent process. The handler opens the gate, change the LED's colour and reset the SIGALRM timer.

SIGUSR2: It is used to define a warning status. For example, if some error occurs getting the isToOpen variable, the daemons send this signal. The handler change the LED's colour and reset the SIGALRM timer.

SIGALRM: It is used to clear the current status to the default one, which is idle. The SIGALRM is periodic, so from time in time, the SIGALRM signal raises, so there isn't any need to trigger it.

F.4.7 IPCs

To communicate between processes were use two IPCs: PIPEs and shared memory. PIPEs are one-way communication only, so one process can only write and other only read from the PIPE. They were used for the main process to receive the whitelist of plates. For that, the main process creates the two pipes before the daemons creation, so the children can also use it. To use PIPEs the following steps must be done.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{

    int fd1[2]; // Used to store two ends of the pipe
    pid_t p;

    if (pipe(fd1) == -1) {
        fprintf(stderr, "Pipe Failed");
        return 1;
    }

    p = fork();

    if (p < 0) {
        fprintf(stderr, "fork Failed");
        return 1;
    }

    // Parent process
    else if (p > 0) {

        close(fd1[0]); // Close reading end of first pipe

        //Write Operations Only

        // Close writing end
        close(fd1[1]);

        exit(1);

    }

    // child process
    else {
        close(fd1[1]); // Close writing end of first pipe

        // Read Operations Only

         // Close reading end
        close(fd1[0]);

        exit(0);
    }
}
```

Code 18: PIPE Creation

The PIPE array must be int type and have 2 slots. The first slot is the write and the second one is to read. However, it's not possible to start writing or reading after the pipe creation. The process that will read from the PIPE must close the write channel, otherwise, the PIPE won't know if that process wants to read or write. The same happens to the writing process. When the processes want to close the PIPE, they must close the remaining write and read channels.

Shared memory is the most dangerous IPC because there is no control of who is reading or writing, although it's the fastest. It will be used to share the PIgate_ID for all the processes, so the main process will write the PIgate_ID and the others will just read, so there's no need for a synchronization method.

To create the shared memory, the main process starts to open a shared memory object. The created object name is "shm_PIgateID" and it has permissions for the file owner to read, write and execute and the group only has permissions to read.

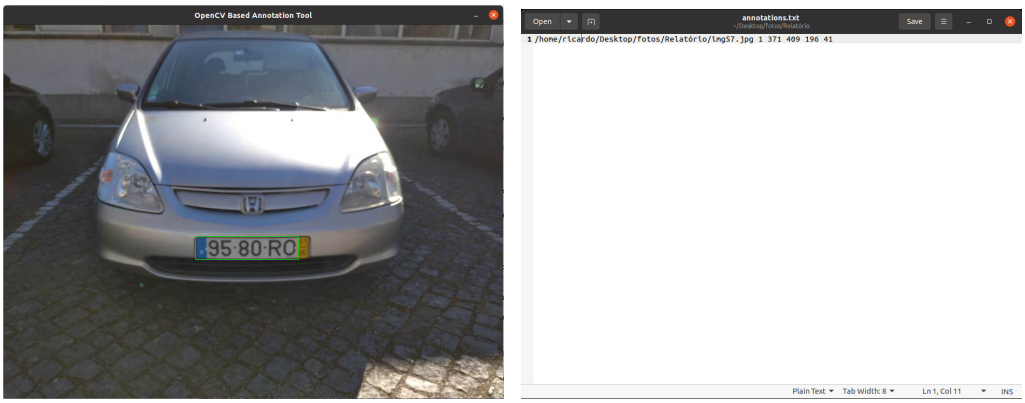
After that, it preallocates a shared memory area by determining the current value of a configurable system limit for page size. For that, the *ftruncate* function is used to truncate the object to a precise size, which in this case is the PIGATELEN.

Finally, the *mmap* function is called to create a new mapping in the virtual address space. Now, for the daemons to access this memory, they need to firstly open the get the shared memory object in its processes and then call the *mmap* function to get a pointer to created memory.

F.4.8 Plate Recognition Module

The plate recognition module was implemented with OpenCV, using a Haar-feature cascade classifier to find a license plate in an image. Then, Tesseract is used to recognize the characters from the found license plate. A Haar classifier can be used to detect all kinds of objects in a picture, to do so, there are trained models, which store the information necessary to identify each object. To identify portuguese license plates it was necessary to train a custom model.

Model training can be done using OpenCV's tools. First of all, it is necessary to have a large collection of pictures of the object to be detected (positive samples) and a large collection of background pictures which do not contain the object (negative samples). The opencv_annotations tool is used to create annotations for each picture, it opens each picture in a given directory and with the mouse the user must draw a rectangle on the object to be detected. The output of this tool is a text file with each picture's name, the number of objects and their coordinates.



(a) Annotation tool

(b) Annotation tool output 2

After annotations are created, the negative samples can be prepared, this is done manually by creating a text file with the path and name for each negative sample, one per line. Having both positive and negative images ready, one can run the opencv_createsamples tool, which generates the actual samples used in training, this corresponds to a binary file for each license plate found on the positive pictures. As input this tool takes the annotations file, the positive and negative samples as well as the desired resolution for the samples. After generating the samples, everything is ready to train a cascade model, which is done by running the opencv_traincascade tool. These tools are built with openCV, however, the traincascade tool has been discontinued and to use it, a previous version (3.4)of OpenCV must be built.

To identify a license plate there are various steps to be taken, first of all it is necessary to capture an image. Then the image is processed and fed into the cascade classifier, which identifies objects contained in an image, accordingly to it's training of course. Once a license plate is found, a cropped image of it is stored and later read by the Tesseract-OCR program. For this application, Tesseract is being used as a command-line application since the OpenCV version

built by buildroot could not find it and therefor use it directly. The program takes an image as input aswell as a configuration file, which contains patterns a caracter whitelist and a caracter blacklist. It's output can be in various maners, fot this application, it is a .txt file.

In order to recognize characters accurately, Tesseract must be configured accordingly to the text it is trying to identify. As such, there are several options to change. The settings used for this project were chosen through some experimentation as are as follows:

- PSM Page segmentation mode : 8 - Treat Image as single word
- OEM OCR engine mode : 2 - Tesseract + LSTM
- Whitelist: Uppercase letters and numbers
- Blacklist: Special characters and punctuation
- Patterns: All the possible combinations of numbers and letters for license plates.

```
1
2  int detectPlate( Mat frame, int len )
3  {
4      Mat frame_gray;
5      std::vector<Rect> license_plates;
6
7      /*turns image into gray scale*/
8      cvtColor( frame, frame_gray, COLOR_BGR2GRAY );
9      equalizeHist( frame_gray, frame_gray );
10
11
12     /*finds all license plates on the image, stores coordinates in cv::Rect vector */
13     plate_cascade.detectMultiScale( frame_gray, license_plates );
14     string plate;
15
16     //If Plate not found
17     if(license_plates.size() == 0)
18         return -EXIT_FAILURE;
19
20     //for each found license plate, run read_license_plates
21     for ( size_t i = 0; i < license_plates.size(); i++ )
22     {
23         //draw a rectangle around the license plate on the original image
24         Point corner_1(license_plates[i].x, license_plates[i].y);
25         Point corner_4(license_plates[i].x + license_plates[i].width, license_plates[i].y +
       license_plates[i].height);
26         rectangle(frame, corner_1, corner_4, Scalar( 255, 0, 255 ), 1, LINE_8, 0);
27         Mat plate_pic = frame_gray( license_plates[i] );
28
29         //create a .jpg file of the cropped license plate
30         string imageName = "/etc/pic" + to_string(len) + ".jpg";
31
32         imwrite(imageName.c_str(), plate_pic);
33
34     }
35
36    return len;
37 }
```

Code 19: Plate detection function implementation (detectPlate)

This function is responsible for detecting license plates in a given image. This function begins by creating a grayscale image and augmenting it's contrast through histogram equalization. Then the cascade classifier is ran, if a license plate is found, it's coordinates are stored in the vector license_plates wich stores coordinates, width and length of a rectangle. Once a license plate is found, a rectangle is drawn in the original picture surrounding the license plate, this served the purpose of visually checking if the identifier was working correctly during development. The

image is also cropped and saved in as a jpeg file containing only the license plate. This will then be used to recognize the text in it. The file name under which each file is saved depends on the variable "len", it's value is received as a parameter and if a file is saved it is returned, indicating the index for the license plate image. If no license plates are found on the image, the function will return EXIT_FAILURE, indicating that no file was saved.

F.4.9 FIFO module

The communication between threads will be done by FIFOs, so they need to be implemented. The FIFOs will allow better data control. Imagine that thread 1 takes a photo in 1 second and thread 2 takes 2 seconds to analyse it. If nothing is done, will be a time that the system will crash. To prevent that, the implemented FIFO has a defined length that cannot be suppressed.

```
/**
 * @file fifo.h
 * @author srhugo Pedro Martins
 * @brief FIFO module
 * @version 0.1
 * @date 2021-05
 * @review 2022-02-03
 *
 * @copyright Copyright (c) 2022
 */

#ifndef _FIFO_MODULE_H
#define _FIFO_MODULE_H

#include <stdint.h>
#include "/home/hugo/Downloads/buildroot-2021.02.5/output/host/arm-buildroot-linux-gnueabihf/
    sysroot/usr/include/opencv2/opencv.hpp"

extern "C" {
#include "utilits.h"
}

/**
 * @brief C String Defenition
 *
 */
typedef char arrayString[PLATESSIZE];

/**
 * @brief Char FIFO
 *
 */
typedef struct serialFifo
{
}fifo8_t;

/**
 * @brief Unsigned Short FIFO
 *
 */
typedef struct serialFifo16
{
}fifo16_t;

/**
 * @brief unsigned int FIFO
 *
 */
typedef struct serialFifo32
{
}fifo32_t;

/**
 * @brief c String FIFO
```

```
 *
 */
typedef struct serialFifoString
{
}fifoString_t;


/**
 * @brief OpenCv Mat FIFO
 *
 */
typedef struct serialFifoPhoto
{
}fifoPhoto_t;



/**
 * @brief Initializes one fifo with the default indexs and
 * and receive parameters
 *
 * @param fifo Specifics the fifo
 * @param array Pointer to a buffer that fifo can storage info
 * @param len Buffer length. Must be iqual to base two
 * @return  Return EXIT_SUCESS if all went well, or ENODATA
 *            if an error occurs
 */
char fifo8_init(fifo8_t *fifo, char *array, uint8_t len);
char fifo16_init(fifo16_t *fifo, uint16_t *array, uint16_t len);
char fifo32_init(fifo32_t *fifo, uint32_t *array, uint16_t len);
int fifoString_init(fifoString_t *fifo, arrayString* array, uint16_t len);
int fifoPhoto_init(fifoPhoto_t *fifo, cv::Mat* array, uint16_t len);


/**
 * @brief Adds the wanted data into the fifo
 *
 * @param fifo Specifics the fifo
 * @param data What the user want to add in the fifo
 * @return Return EXIT_SUCESS if all went well, or -ENOBUFS
 *            if an error occurs
 */
char fifo8_push(fifo8_t *fifo,char byte);
char fifo16_push(fifo16_t *fifo,uint16_t integer);
char fifo32_push(fifo32_t *fifo,uint32_t integer);
int fifoString_push(fifoString_t *fifo,const char* data);
int fifoPhoto_push(fifoPhoto_t *fifo,cv::Mat data);

/**
 * @brief This funtion gets a byte from the fifo
 *
 * @param fifo Specifics the fifo
 * @param return Where the string will be return
 * @return return one position of the buffer or
 *      If the fifo is empty return -ENODATA
 */
int fifo8_pop(fifo8_t *fifo);
int32_t fifo16_pop(fifo16_t *fifo);
int64_t fifo32_pop(fifo32_t *fifo);
int fifoString_pop(fifoString_t *fifo,char* returnString);
int fifoPhoto_pop(fifoPhoto_t *fifo,cv::Mat* returnString);

/**
 * @brief Get the FifoBuffSize object
 *
 * @param fifo Specifics the fifo
 * @return Return the
 */
char get_Fifo8BuffSize(fifo8_t fifo);
uint16_t get_Fifo16BuffSize(fifo16_t fifo);
uint16_t get_Fifo32BuffSize(fifo32_t fifo);
uint32_t get_FifoStringBuffSize(fifoString_t fifo);
uint32_t get_FifoPhotoBuffSize(fifoPhoto_t fifo);
```

```
127
128 /**
129  * @brief Discarts all the fifo content. No way to reverse
130  *
131  * @param fifo Specifics the fifo
132  */
133 void clear_fifo8(fifo8_t *fifo);
134 void clear_fifo16(fifo16_t *fifo);
135 void clear_fifo32(fifo32_t *fifo);
136 void clear_fifoString(fifoString_t *fifo);
137 void clear_fifoPhoto(fifoPhoto_t *fifo);
138
139 #endif
```

Code 20: Fifo.hpp

The FIFO module has support for some data types. The original module, which was created and used in LPI2 curricular unit, had support for uint8, uint16 and uint32, however, these types won't be used in this project. Moreover, the new data types, c string and mat (openCV Struct), were added following the same methodology, turning the module more complete.

This module has all the needed functions to control a FIFO. Because it was created in .c, templates weren't available, so that's why there is a single function for each data type.

F.4.10 Threads

As said in the design chapter, the main process will have 5 tasks, each one with a specific purpose. Following the captureCutImage thread example, all threads were created respecting the next order.

```
1
2 pthread_attr_t thread_attr;
3 struct sched_param thread_param;
4
5 pthread_t captureCutImage_id
6
7 pthread_attr_init(&thread_attr);
8 pthread_attr_getschedparam(&thread_attr, &thread_param);
9
10 //Setup the thread priority and trys to create it. If successed the program continues.
11 setupThread(captureCutImagePrio, &thread_attr, &thread_param);
12 checkFail(  pthread_create(&captureCutImage_id, &thread_attr, t_captureCutImage, NULL)  );
13
14  pthread_join(captureCutImage_id,NULL);
```

Code 21: Thread Creation

After all variables creation and initialization, the first step was to set up the thread priority. This priority was previously defined in the code. After that, the program tries to create the thread. The *pthread_create* return will be verified by the checkFail function that will decide if the thread was successfully created. If everything went OK, the created thread is joined.

Because every task does a specific thing, they need to be analysed individually.

t_captureCutImage

```
1 int gateStatus;
2 Mat imageCapture(2,2,CV_8UC3, Scalar(0,0,255));
3
4 printf("t_captureCutImage Thread is Ready \n");
5
6 while (1)
7 {
8     sleep(5);
9
10     gateStatus = getGateStatus();
```

```
11
12      switch (gateStatus)
13      {
14          case gateOpen:
15              /*Do Nothing*/
16              break;
17
18          case -EXIT_FAILURE:
19              ledRGBStatus(warning);
20              break;
21
22          case gateClose:
23
24              if(get_FifoPhotoBuffSize(imagesFifo) == FIFOLEN)
25              {
26                  /*Ignore, FIFO is FULL*/
27              }
28              else
29              {
30                  imageCapture = take_picture();
31
32                  fifoPhoto_push(&imagesFifo,imageCapture);
33              }
34
35              break;
36
37          default:
38              /*FATAL ERROR*/
39              printf("Error Getting gateStatus in t_captureCutImage \n");
40              kill(getpid(),SIGTERM);
41              break;
42      }
43
44  }
```

Code 22: Update Plate Thread Implementation

After initializing all the needed variables, the thread will try to take a picture. However, before that, it needs to verify if the gate is open because if it is, there's no need to take pictures. The default value for the switch case is a fatal error since the getGateStatus function just can return three different values.

t_plateRecognition

```
1
2   void *t_plateRecognition(void *arg)
3   {
4       int gateStatus;
5       int detectPlateReturn;
6       Mat receivedImage;
7       Mat plateImage;
8
9       printf("t_plateRecognition is ready \n");
10
11      while (1)
12      {
13
14          while(fifoPhoto_pop(&imagesFifo,&receivedImage) == -ENODATA )
15          { /*Waits for an image*/
16              sleep(1);
17          }
18          gateStatus = getGateStatus();
19
20          switch (gateStatus)
21          {
22              case gateOpen:
23                  /*Clears the FIFO since the gate is already open*/
24                  clear_fifoPhoto(&imagesFifo);
25                  break;
26
27              case -EXIT_FAILURE:
```

```
                ledRGBStatus(warning);
                break;

            case gateClose:

                if(get_Fifo16BuffSize(platesFifo) == FIFOLEN)
                {
                    /*Ignore, FIFO is FULL*/
                }
                else
                {
                    detectPlateReturn = detectPlate(receivedImage, (platesFifo.writeIndex & (
    platesFifo.buff_len-1) ) );

                    if(detectPlateReturn == -EXIT_FAILURE)
                    {
                        /*ERROR, PLATE NOT FOUNDED*/
                    }
                    else
                    {
                        cout << "Pushing index photo" << endl;

                        fifo16_push(&platesFifo,detectPlateReturn);
                    }
                }
                break;

            default:
                /*FATAL ERROR*/
                printf("Error Getting gateStatus in t_plateRecognition \n");
                kill(getpid(),SIGTERM);
                break;
        }
    }

}
```

Code 23: Plate Recognition Thread Implementation

After initializing all the variables, the thread then checks the current gate status. If the gate is closed, and the buffer is not full, the function detectPlate is called. The function looks for license plates in the receivedImage, if one is found, a jpeg file is saved containing only the license plate and it's index is returned. If no license plate is found, the function returns failure.

t_textRecognition

```

void *t_textRecognition(void *arg)
{
    int gateStatus;
    int16_t receivedPlate;
    string plateString;

    printf("t_textRecognition is ready! \n");

    while (1)
    {

        while( (receivedPlate = fifo16_pop(&platesFifo) ) == -ENODATA )
        { /*Waits for an image*/
            sleep(1);
        }

        gateStatus = getGateStatus();

        switch (gateStatus)
        {
            case gateOpen:
                /*Clears the FIFO since the gate is already open*/
                clear_fifo16(&platesFifo);
                break;
```

```
26
27              case -EXIT_FAILURE:
28                  ledRGBStatus(warning);
29                  break;
30
31              case gateClose:
32
33                  if(get_FifoStringBuffSize(textFifo) == FIFOLEN)
34                  {
35                      /*Ignore, FIFO is FULL*/
36                  }
37                  else
38                  {
39
40                      plateString = read_license_plates(receivedPlate);
41
42                      if(plateString == "ERROR")
43                      {
44                          /*Plate String not founded*/
45                      }
46                      else
47                      {
48                          cout << "Pushing text string" << plateString << endl;
49                          fifoString_push(&textFifo,plateString.c_str());
50                      }
51                  }
52
53                  break;
54
55              default:
56                  /*FATAL ERROR*/
57                  printf("Error Getting gateStatus in t_textRecognition \n");
58                  kill(getpid(),SIGTERM);
59                  break;
60          }
61      }
62  }
```

Code 24: Text recogniton thread Implementation

This thread is responsible for the text recognition. If a license plate is found, this thread will check the gate's current state, if it is closed and the buffer for read plates is not full, the thread will call the function read_license_plates. This function returns "ERROR" if no license plate text is found or is incomplete, if a license plate if found, the function returns the found string, which is then pushed into the buffer.

t_updatePlate

```
1
2  char inBuff[PLATESSIZE];
3  close(recPlatePIPE[1]); // Close writing end
4
5  printf("t_updatePlate is ready! \n");
6
7  while (1)
8  {
9
10     while( read(recPlatePIPE[0] , inBuff, PLATESSIZE) < 1 )
11     {}
12
13     //Convert a string into integer
14     PLATESLEN = atoi(inBuff);
15
16     pthread_mutex_lock(&updatePlatesMutex);
17
18     for(int i = 0 ; i < PLATESLEN ; i++)
19     {
20         read(recPlatePIPE[0] , inBuff, PLATESSIZE);
21
22         strcpy(whitelistPlates[i] , inBuff);
23
```

```
24          // printf("Storage Plate: %s \n", whitelistPlates[i] );
25      }
26
27      pthread_mutex_unlock(&updatePlatesMutex);
28
29      //Little delay
30      sleep(2);
31
32  }
```

Code 25: Update Plate Thread Implementation

The first thing is to close the writing end of recPlatePIPE. After that, the threads waits for the plates. The receive sequence is: Number of plates, Plate1, Plate2, etc., thus, when it receives the first value, it must convert to integer, so it can be used in the for cycle. To make sure that plateValidation threads doesn't read the whitelist at the same time that updatePlates is writing, a mutex is used. When it is locked, the plateValidation cannot use the resource. After setting the new whitelist, the mutex is released.

t_plateValidation

```
1  char foundedPlate[PLATESSIZE];
2  close(entriesDBPIPE[0]); // Close reading end
3  bool whitelistPlate = false;
4
5  printf("t_plateValidation is ready! \n");
6
7  while (1)
8  {
9      while (fifoString_pop(&textFifo,foundedPlate) == -ENODATA)
10     { /*Waiting for need plates*/ }
11
12
13     pthread_mutex_lock(&updatePlatesMutex);
14
15     for(int i = 0 ; i < PLATESLEN && whitelistPlate == false ; i++)
16     {
17         whitelistPlate = ( strcmp(foundedPlate,whitelistPlates[i]) == 0);
18     }
19
20     pthread_mutex_unlock(&updatePlatesMutex);
21
22     if(whitelistPlate)
23     {
24
25         write(entriesDBPIPE[1], foundedPlate, strlen(foundedPlate));
26
27         printf(" GOOD PLATE \n ");
28         setitimer (ITIMER_REAL, &itv, NULL); //Reset SIGALARM timer
29         clear_fifoString(&textFifo);
30         ledRGBStatus(allow);
31         openGate();
32     }
33     else
34     {
35         printf(" BAD PLATE \n ");
36         setitimer (ITIMER_REAL, &itv, NULL); //Reset SIGALARM timer
37         ledRGBStatus(denied);
38     }
39
40     //clean withlistPlate flag
41     whitelistPlate = false;
42
43     sleep(1);
44 }
```

Code 26: Plate Validation Thread Implementation

First of all, the t_plateValidation thread starts to close the reading channel of entriesDBPIPE, so it can communicate with the entriesDB daemon. Moreover, it waits for a plate text to analyse.

The updatePlatesMutex is locked, the thread verifies if it's a valid plate and unlocks the mutex. The whitelistPlates is a shared resource, thus the mutex should be used for synchronization. Furthermore, if it is a valid plate, it clears the textFifo, show a green light and activate the relay. Otherwise, an error message is displayed and the red light is shown.

F.4.11 Auxiliary Functions

Within the project implementation, there were moments that some work needed to be done that made the program harder to read. For example, to validate the thread creation. With this scope, the utilits module was created. It has functions and variables that all modules can use, including the PIPEs arrays definitions, so the daemons and main module can both access the variables.

```
/**
 * @file utilits.h
 * @author PIgate
 * @brief Usefull functions for all the modules
 * @version 0.1
 * @date 2022-01-31
 *
 * @copyright Copyright (c) 2022
 *
 */
#ifndef UTILITS_H
#define UTILITS_H

#include <pthread.h>

/**
 * @brief Global Defenitions
 *
 */
#define MAXPLATESLEN 512
#define MAXPLATELENDIGITS 4
#define PLATESSIZE 8
#define PIGATELEN 8
#define READWRITE_PERMISSION 0666
#define SHM_PIGATEID_NAME "shm_PIgateID"

/**
 * @brief PIPE array's defenition
 *
 */
extern int entriesDBPIPE[2];
extern int recPlatePIPE[2];


/**
 * @brief Setup a given thread, so it can be successfully created
 *
 * @param priority Thread Priority
 * @param pthread_attr Thread Attribute
 * @param pthread_param Thread Paramenter
 */
void setupThread(int priority, pthread_attr_t *pthread_attr, struct sched_param *
    pthread_param);


/**
 * @brief Verifies if a thread was successfully created
 *
 * @param status Pthread_create return
 */
void checkFail(int status);


/**
```

```
 * @brief For a given buffer, this funtions removes all the present hiffens (-)
 *  Before: AA-00-00
 *  After: AA0000
 *
 * @param buffer
 * @param len buffer's length
 *
 */
void removeHiffen(char* buffer, int len);


/**
 * @brief Following the Portuguese plates, this functions puts an hiffen (-) between
 * a par of numbers or letters
 *
 * Before: AA0000
 * After: AA-00-00
 *
 * @param buffer Must be big enough
 * @param len buffer's length
 *
 */
void insertPlateHiffen(char* buffer, int len);


#endif //UTILITS_H
```

Code 27: Utilits.h

F.4.12 Makefile

Makefiles were used to compile the whole program. They were split into modules. Each one compiled a module and then everything is grouped into a folder, so then it can be sent to raspberry. An important step was to define the compiler. Since the used image is 32 Bits, the used compiler was the arm-buildroot-linux-gnueabihf-gcc one.

The ledRGB and relay makefiles are pretty similar since both are device drivers modules. Because the implemented code here was for kernel space, the KDIR is needed to get the kernel libraries.

```

obj-m := ledRGB/relay.o
ledRGB/relay-objs:= ledModule.o/relayModule.o
KDIR := /home/hugo/Downloads/buildroot-2021.02.5/output/build/linux-custom
ARCH ?= arm
CROSS_COMPILE ?= arm-buildroot-linux-gnueabihf-
PIgateFolder = ../PIgate
```

Code 28: Utilits.h

The src Makefile is where all the source code is compiled. Some modules are required to link flags, so they can use their libraries. The following flags were used.

```
#pthread library
-pthread

#sharead memory and signal handler functions
-lrt

#c/python \ac{api}
-lpython3.9

#Use c++ std libraries
-lstdc++

#To opencv and the tesseract libraries
```

```
14 -lopencv_core -lopencv_highgui -lopencv_imgproc -lopencv_objdetect -lopencv_imgcodecs
```

Code 29: Utilits.h

The main Makefile calls the modules' Makefiles, get the needed generated files and cleans everything. It creates a folder named PIgate and everything that the PIgate program needs is in there. Also, this Makefile generates the Doxygen documentation, which will be present in the docs folder.

F.5 Documentation

All implemented modules were properly documented, so in the future, if someone or even the project developers want to use the project or some module will understand what it does, without spending hours trying to figure it out.

To generate all the documentation Doxygen was used. Doxygen is a specific tool created for that purpose. To start using it, firstly it needs to be installed. Then, in the project folder, the command *doxygen "File Name"* was used, and it created a doxyfile named "File Name". Then, inside that file, the following parameters were modified.

```
1
2 PROJECT_BRIEF           = "Plate Recognition System"
3 PROJECT_LOGO            = ./docs/PIgate_logo.png
4 OUTPUT_DIRECTORY        = docs/Doxygen/
5 INPUT                   = ./inc ./src
```

Code 30: Utilits.h

Typing the same command again, the documentation was generated.

G Testing

After implementing everything, the designed tests must be done, so the product can be validated. The following chapter is also divided between hardware, user interface and local system. The project parts were tested independently and after all validations, the final tests were done.

G.1 Hardware

For the hardware tests, the results were the expected ones. Every component did what it should do, thus the overall evaluation is perfect.

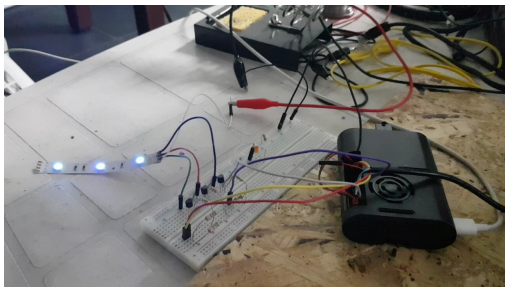
Hardware: Tests	Expected Results	Real Results
Camera Image capture	Clear image	Clear image
Camera night image capture	Clear night image	Clear image
Turn the LEDs on	Leds turned on	All 3 colours ON
Put a car above the magnetic sensor	Magnetic Field Changes	———————-
Input relay test	Read pin value	Read pin value
Output relay test	Write in value	Write pin Value

Table G.1: Hardware Test Cases Results

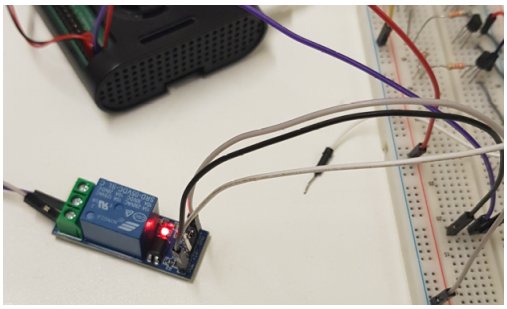
Some pictures of the hardware tests results. The LEDs test was done the idle colour, which is a mix of all the 3 LEDs

(a) Day Mode Photo

(b) Night Mode Photo



(c) Leds test



(d) Relay ON

Figure G.1: Hardware Tests

G.2 User Interface

For the user interface validation, all the design tests were done. Meanwhile, the database was also a target of tests since both work together. If the database was broken it wouldn't be possible to store a plate or associate a PIgate. After all the tests, the user interface was able to complete all without any flaws.

Web App: Tests	Expected Results	Real Results
Register a new user	Regist success	User registed
Login a user	Login success	Login success
Add PIgate	Associate the PIgate to the user in case of success	PIgate associated
Remove PIgate	Remove the PIgate association in case of success	PIgate dissociated
Add Plate	Associate the plate to the gate in case of success	Plate added successfully
Remove Plate	Remove the plate association in case of success	Plate removed successfully
Show White List	Show all the Whitelisted cars	Whitelist shown
Show Entries	Show all the done entries	Entries shown
Rename Gate	Rename the PIgate name	PIgate renamed

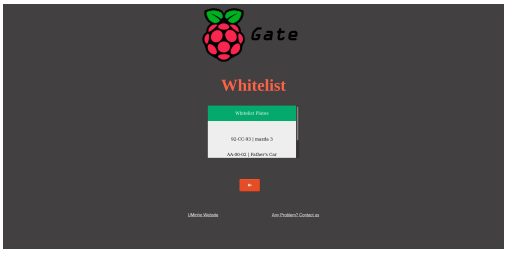
Table G.2: Web App Test Cases Results

The following images show the user interface final look.

(a) Login Page

(b) Register Page

(c) Gate Page



(d) Whitelist Page

Figure G.2: User Interface Tests

G.3 Local System

The local system tests were the last to be done, since the problems that the group faced, although the final results were good. The last test required the magnetic sensor module, so it was the only one that wasn't done. All the situations were tested. It was tested with real car photos and just with the plate and in both cases it was able to identity the plate.

Local System: Tests	Expected Results	Real Results
Receive Plates from Database	Receive success	Plates Received
Send an entry to Database	Database receives the entry	Entry sent
Allowed Car Tries to Enter Home at day	Gate Open and Green Light	Gate opens and green light
Allowed Car Tries to Enter Home at night	Gate Open and Green Light Turns ON	Gate opens and green light
Non-Allowed Car Tries to Enter Home	Red Light Turns ON	Red light output
Problem with the system	Yellow Light Turns ON and Problem Description In Log File	Yellow light and problem feedback
Car Tries to Exit home	Gate Open and Green Light	———

Table G.3: Local System Tests Results

Figure G.3: Outside Car Plate Detection

(a) Nigth Vision Plate Detection (b) Plate Detection and Text Recognition

Figure G.4: Inside Plate Detection Tests

G.4 Final Tests

To do the final test the developed prototype was used (3(a)). Because time was short due to the referred problems, the last final test wasn't done but since this system is only to complement

the existent ones, the result will be the expected for sure.

Final Tests	Expected Results	Real Results
Allowed Car Tries to Enter Home	The Gate Opens Green Light Turns ON Possible to Check the Entry in User Interface	The Gate Opens Green Light Turns ON Possible to Check the Entry in User Interface
Open Gate Now	The Gate Opens Green Light Turns ON Possible to Check the Entry in User Interface	The Gate Opens Green Light Turns ON Possible to Check the Entry in User Interface
Car Tries to Exit	Gate Open and Green Light Turns ON	————————
Add/Remove Allowed Plate	Local system will update the whitelist	Whitelist updated
Non-Allowed Car Tries to Enter Home	Red Light Turns ON	Red light turned ON
Open Gate with gate controller	Gate Opens	————————

Table G.4: Final Tests Results

H Conclusions

After all the project development lots of things happened and it's possible to make some conclusions about the project performance and about the project overall.

Project Performance

The finished product proved to be functional. The license plate detection works as intended and triggers the opening of the gate. All other implemented functionality works as well. This product provides it's user the freedom of not having a remote controller for each gate user or the restriction to be home (or very close to the gate) to actuate it. Furthermore, it achieves this at a much more accessible cost and without the need to install a whole new system, it complements existing ones. However, the products performance in recognizing and reading license plates could be more reliable. An even more complex cascade training together with some improvements to the character recognizing model could deliver the extra reliability.

Project Overall

The first conclusion is that git was essential. It allowed to same code versions and prevented bugs from appearing from nowhere. Also, it made possible for both group elements to stay synchronized on the project. Furthermore, that was a time that the led device driver development gave a bug. A week passed trying to find the bug and the problem was that the structures weren't declared as volatile, causing the bad module function. With git was possible to save the done work and find that bug, without losing any file or project part. Moreover, the Doxygen tool was also an excellent tool to use. Code documentation was very fast and easy, leaving to understandable modules without too much effort.

The second one is that nothing is as easy as it seems. When the project was designed, the

group thought that the project would be simple and adding more features would turn it more complete. One example is the magnetic module. After all, the plate recognition module and the communication between the local system and the database weren't so simple as predicted, delaying the project deadlines.

The third conclusion is: not waste too much time on one thing. This could be a ridiculous conclusion but it was something that happen a lot in the project development, for example, in the camera verification. When trying to verify if the raspberry could take a picture with a custom Linux image, the rasppistill program was always saying *Segmention Fault*. The vcgencmd was outputting *Detected: 1 Supported: 1*, so it wasn't any reason to not work. Turned out that the raspistill had a problem and using the FFMPEG the picture was successfully taken. This took more or less 2 weeks when it should be days.

Another conclusion is that training a cascade classifier is complicated and has lots of flaws. In order to detect text in a licence plate a model was trained, so it could recognize Portuguese plates. There were used more than 150 photos, so the module could be perfectly trained. After all, it isn't perfect. The module can detect licence plates however some angles, environments and distances didn't succeed, thus it needed more training with more angles, more environments and more distances. Moreover, there were times that it didn't detect correctly the plates text. Sometimes a letter was missing or an extra character was presented or even it mix up a character, like an S with a 5 and vice versa. So it's important to set detection rules if they were possible therefore the detecting error could be minimized.

The last conclusion was that nothing is done by itself. There were an enormous time spent and lots of learning since the answers weren't by our side. It was, undoubtedly, an experience that will be remained.

Also thank the teacher Tiago Gomes, Ricardo Roriz and Sergio Pereira for the advice and all the support they gave.

References

[1] License Plate Capture Cameras, LPR Camera, License Plate Recognition *https://www.cctvcamerapros.com*

[2] Automatic Number Plate Recognition Systems Market Size In 2021 MarketWatch *https://www.marketwatch.com/press-release/automatic-number-plate-recognition-systems-market-size-in-2021-59-cagr-with-top-countries-data-who-are-the-top-key-players-in-the-global-automatic-number-plate-recognition-systems-industry-latest-119-pages-report-2021-10-27*

[3] Visual Paradigm: Use case diagram

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/

[4] Wikipedia: State diagram

https://en.wikipedia.org/wiki/State_diagram

[5] Visual paradigm: Sequence Diagram

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/

[6] Raspberry PI 4 Model B Specifications

https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

[7] Introduction To Google Firebase

https://www.c-sharpcorner.com/article/introduction-to-google-firebase/

[8] What is Firebase?

https://medium.com/firebase-developers/what-is-firebase-the-complete-story-abridged-bcc730c5f2c0

[9] What is an Entity Relationship Diagram

https://www.lucidchart.com/pages/er-diagrams

[10] What Is Data Normalization

https://www.datascienceacademy.io/blog/what-is-data-normalization-why-it-is-so-necessary/

[11] Internet

https://www.datascienceacademy.io/blog/what-is-data-normalization-why-it-is-so-necessary/

[12] Proto.io

www.proto.io

[13] Firebase Documents

https://firebase.google.com/docs/

[14] Advantage of Linux

https://www.educba.com/advantage-of-linux/

[15] Stack Overflow Developer Survey Results 2017

https://insights.stackoverflow.com/survey/2017#methodology

[16] Benefits Of Using Github

https://apiumhub.com/tech-blog-barcelona/using-github/

[17] Buildroot

www.buildroot.org

[18] What is OpenCV?

https://www.educba.com/advantage-of-linux/

[19] Libcamera
https://www.libcamera.org/

[20] What is Class Diagram?
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/

[21] I2C Info – I2C Bus, Interface and Protocol
https://i2c.info/

[22] What is a Bus? - Definition from Techopedia
https://www.techopedia.com/definition/2162/bus

[23] MIPI Camera Serial Interface 2
https://www.mipi.org/specifications/csi-2

[24] Camera Serial Interface
MIPI Camera Serial Interface 2

[25] Raspberry Pi MIPI CSI Camera Pinout
https://www.arducam.com/raspberry-pi-camera-pinout/

[26] Debugging CSI-2: Waveform + Protocol Analyzer = Happiness!
https://protocol-debug.com/2017/02/01/debugging-csi-2-waveform-protocol-analyzer-happiness/

[27] What is Serial Communication and How it works?
https://www.codrey.com/embedded-systems/serial-communication-basics/

[28] Rapid object detection using a boosted cascade of simple features
https://ieeexplore.ieee.org/abstract/document/990517/authors#authors

[29] Tutorial Cascade Classifier
https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html

[30] Wi-Fi
https://en.wikipedia.org/wiki/Wi-Fi

[31] IEEE 802.11ac-2013
https://en.wikipedia.org/wiki/IEEE_802.11ac-2013

[32] 802.11 Frame Types
https://en.wikipedia.org/wiki/802.11_Frame_Types

[33] @INPROCEEDINGS990517, author=Viola, P. and Jones, M., booktitle=Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001, title=Rapid object detection using a boosted cascade of simple features, year=2001, volume=1, number=, pages=I-I, doi=10.1109/CVPR.2001.990517
https://ieeexplore.ieee.org/abstract/document/990517/authors#authors

[34] BCM2711 Datasheet *https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf*

[35] Camera Raspberry Pi Documentation *https://www.raspberrypi.com/documentation/accessories/camera.html*

list=no

Appendix

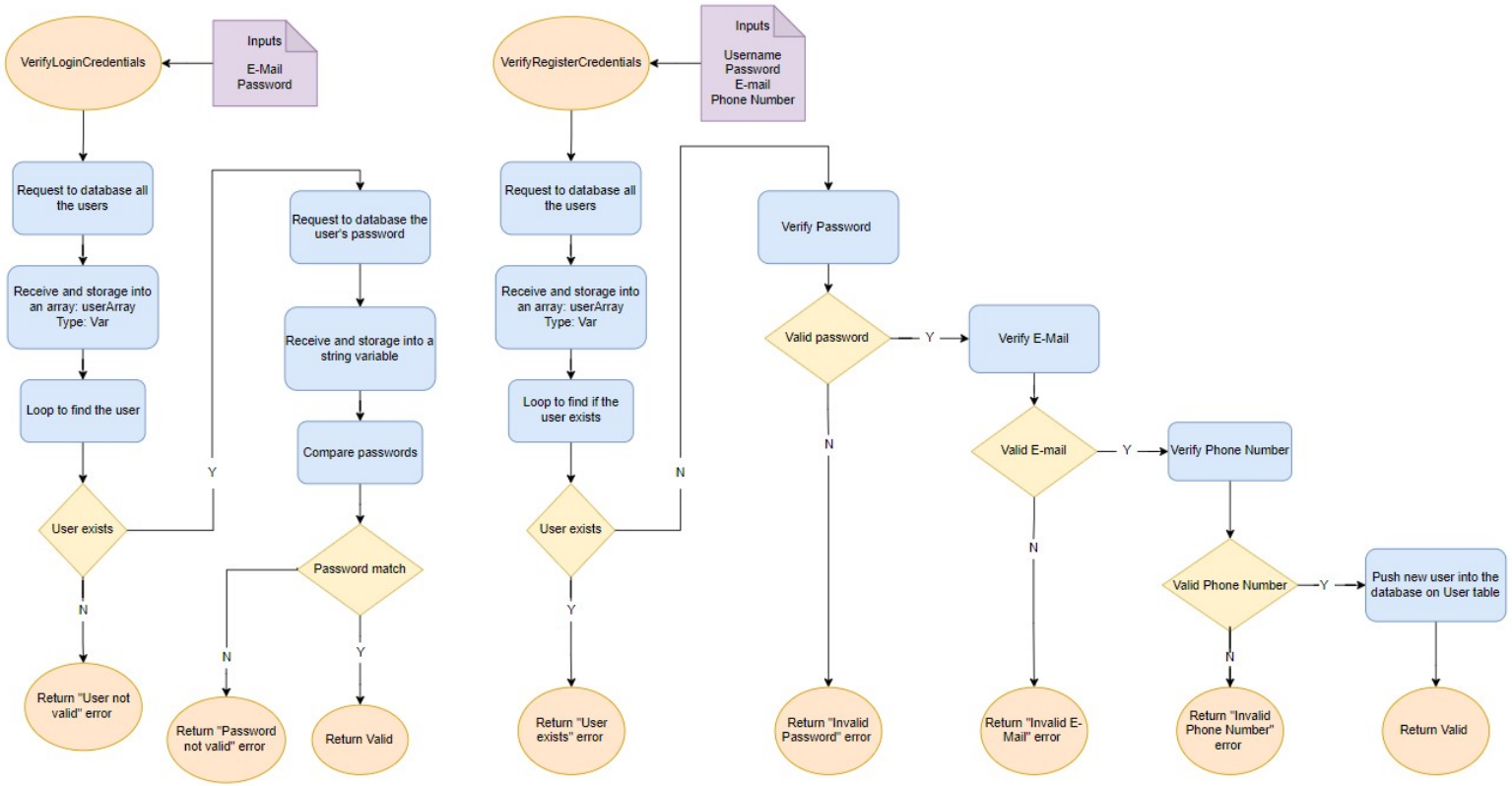


Figure .1: *

Login And Register Functions User Interface Flowcharts (Augmented)

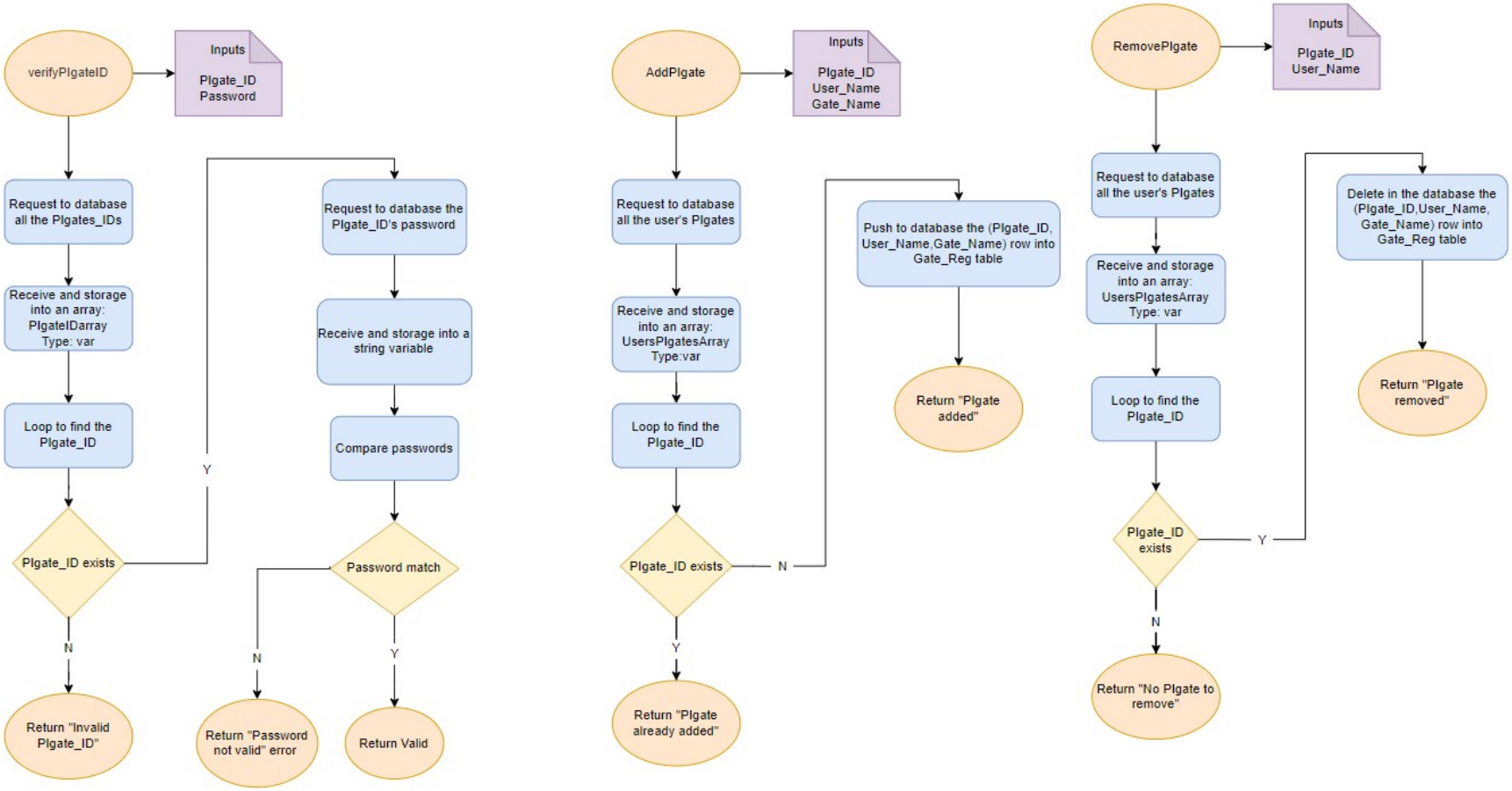


Figure .2: *

Add and Remove Gate Functions User Interface Flowcharts (Augmented)

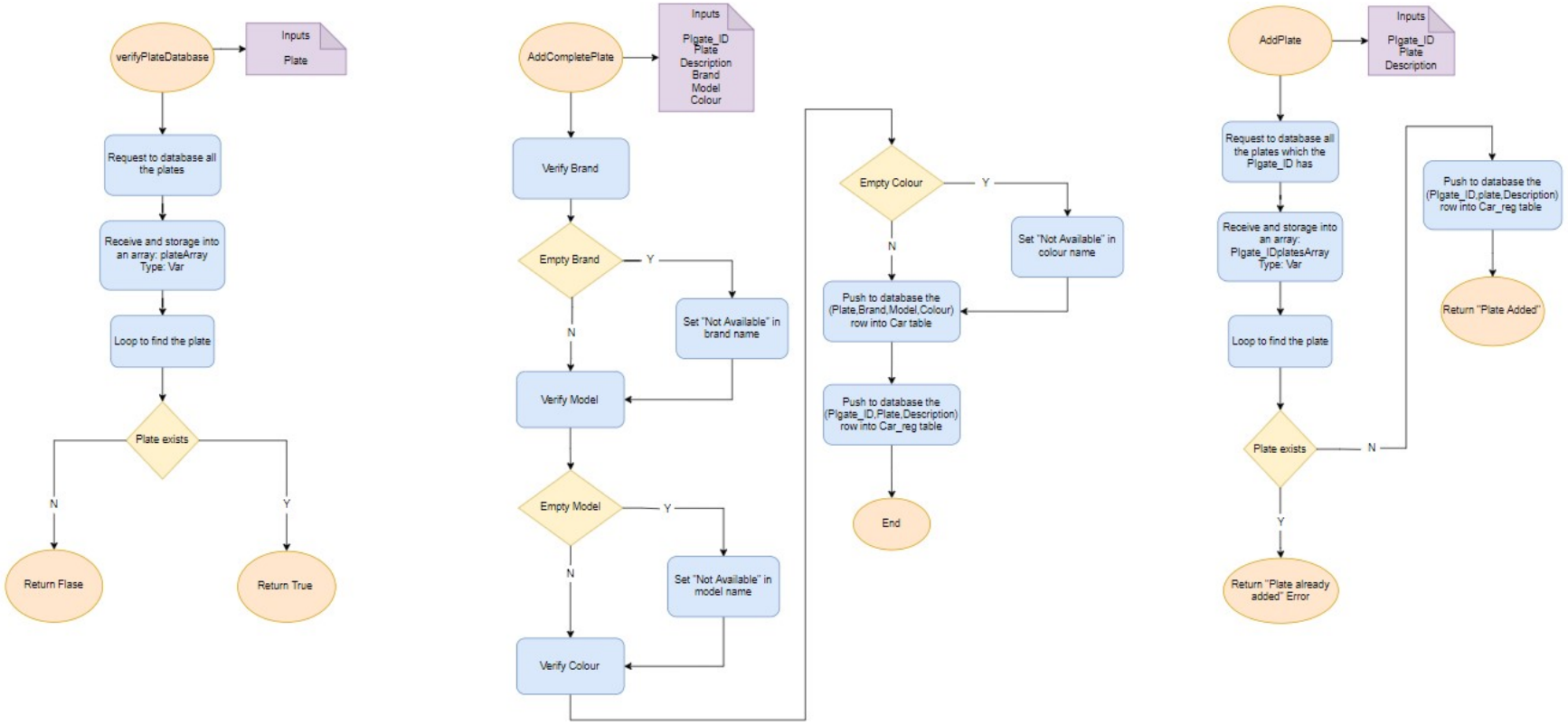


Figure .3: *
Add Plate Functions User Interface Flowcharts (Augmented)

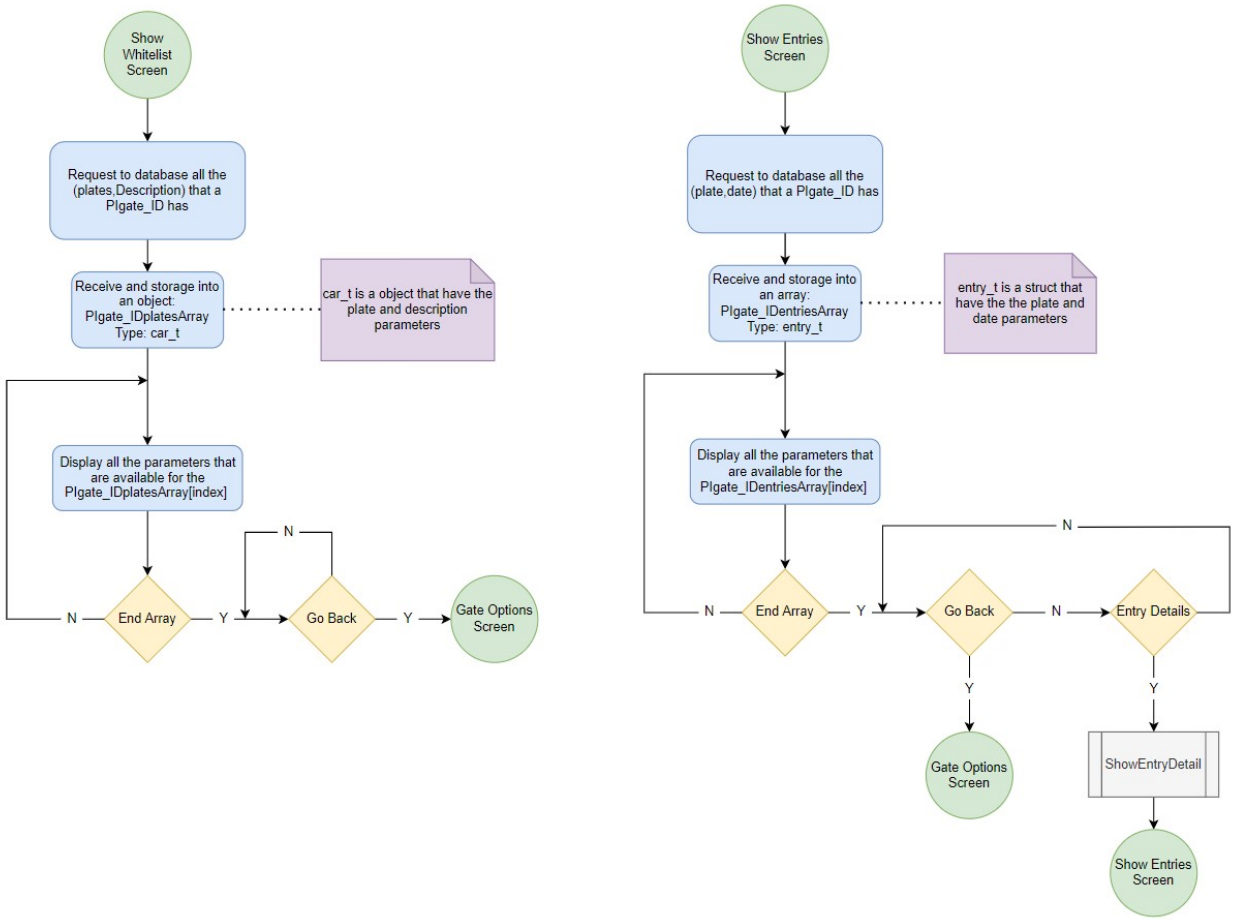


Figure .4: *
Whitelist and Entries User Interface Flowcharts (Augmented)

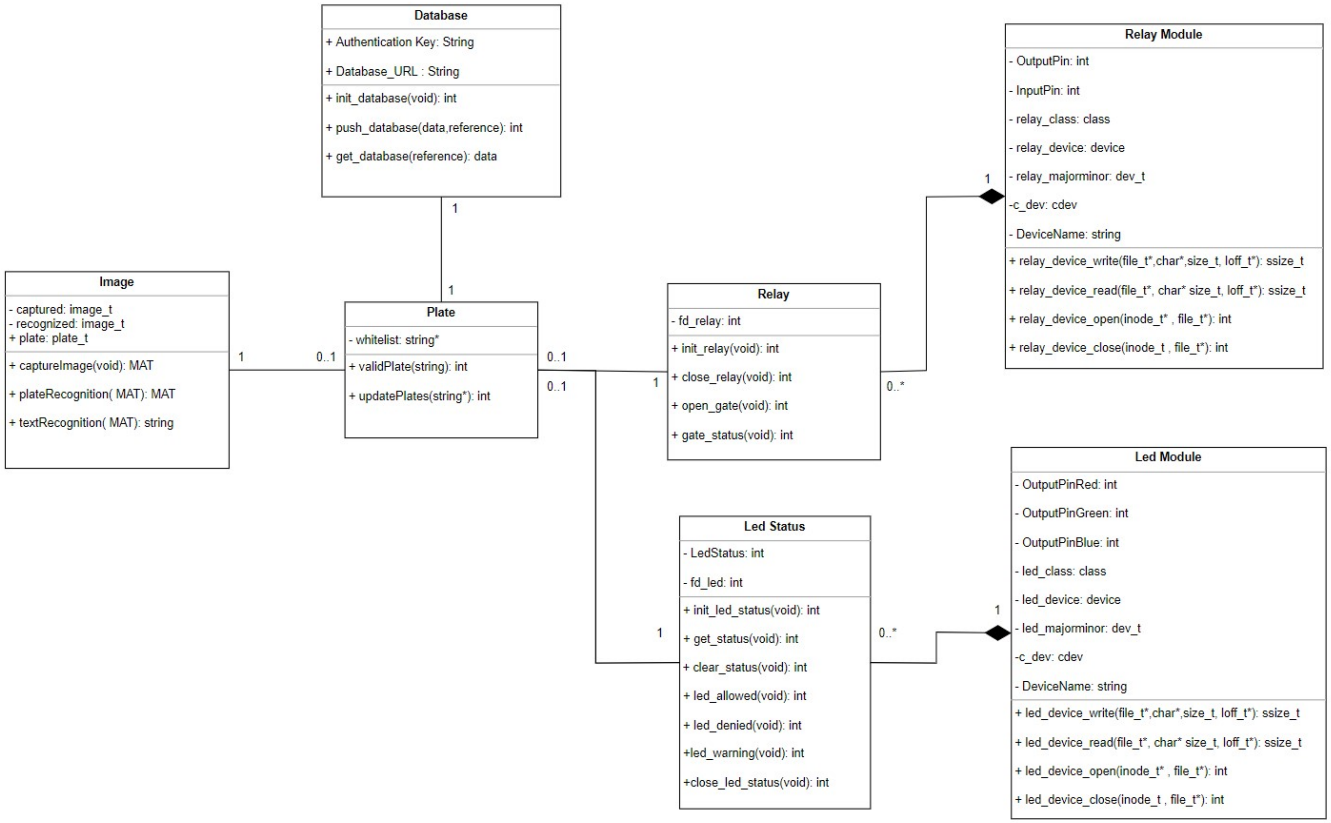


Figure .5: *
Local System's Class Diagram (Augmented)

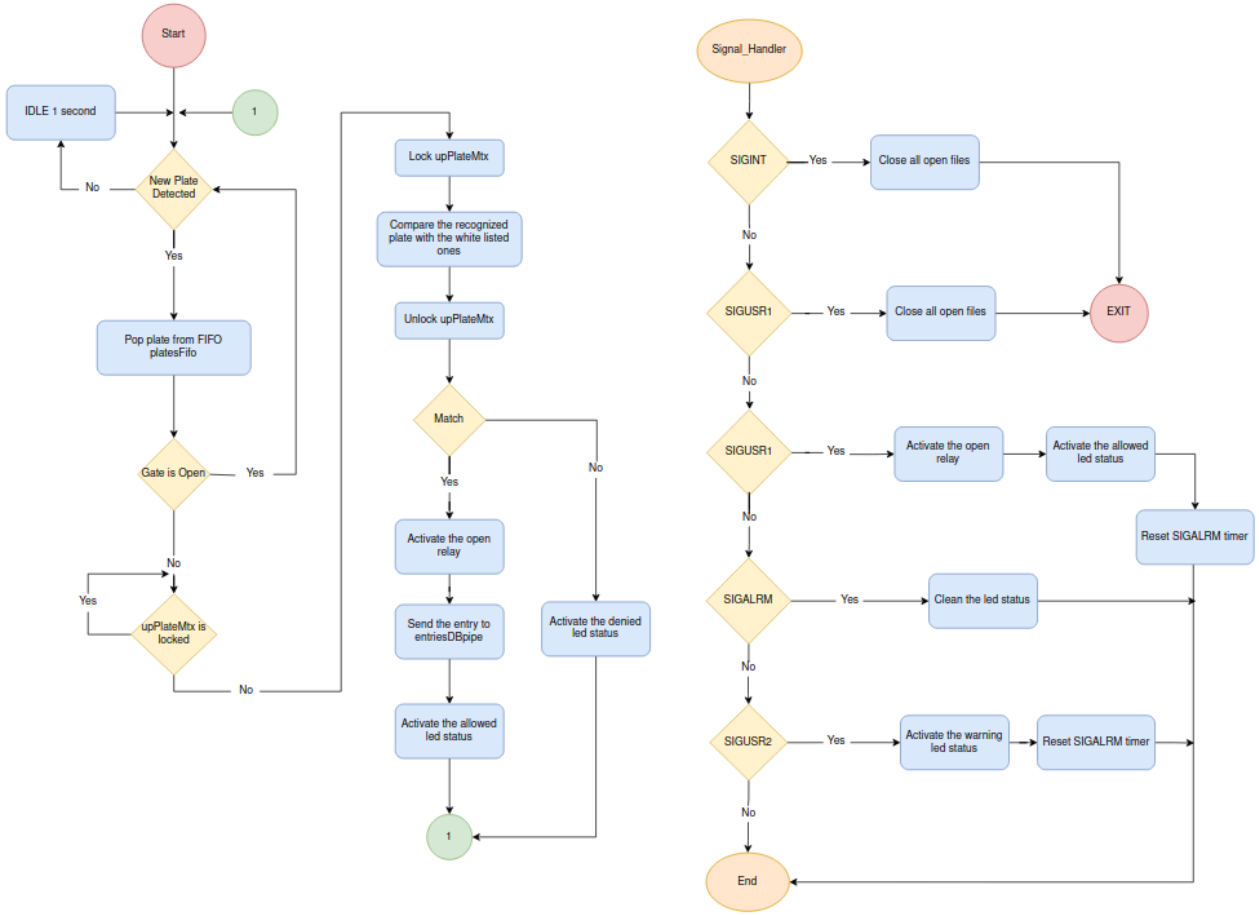


Figure .6: *
Plate_validation and Signal_Handler Flowcharts (Augmented)

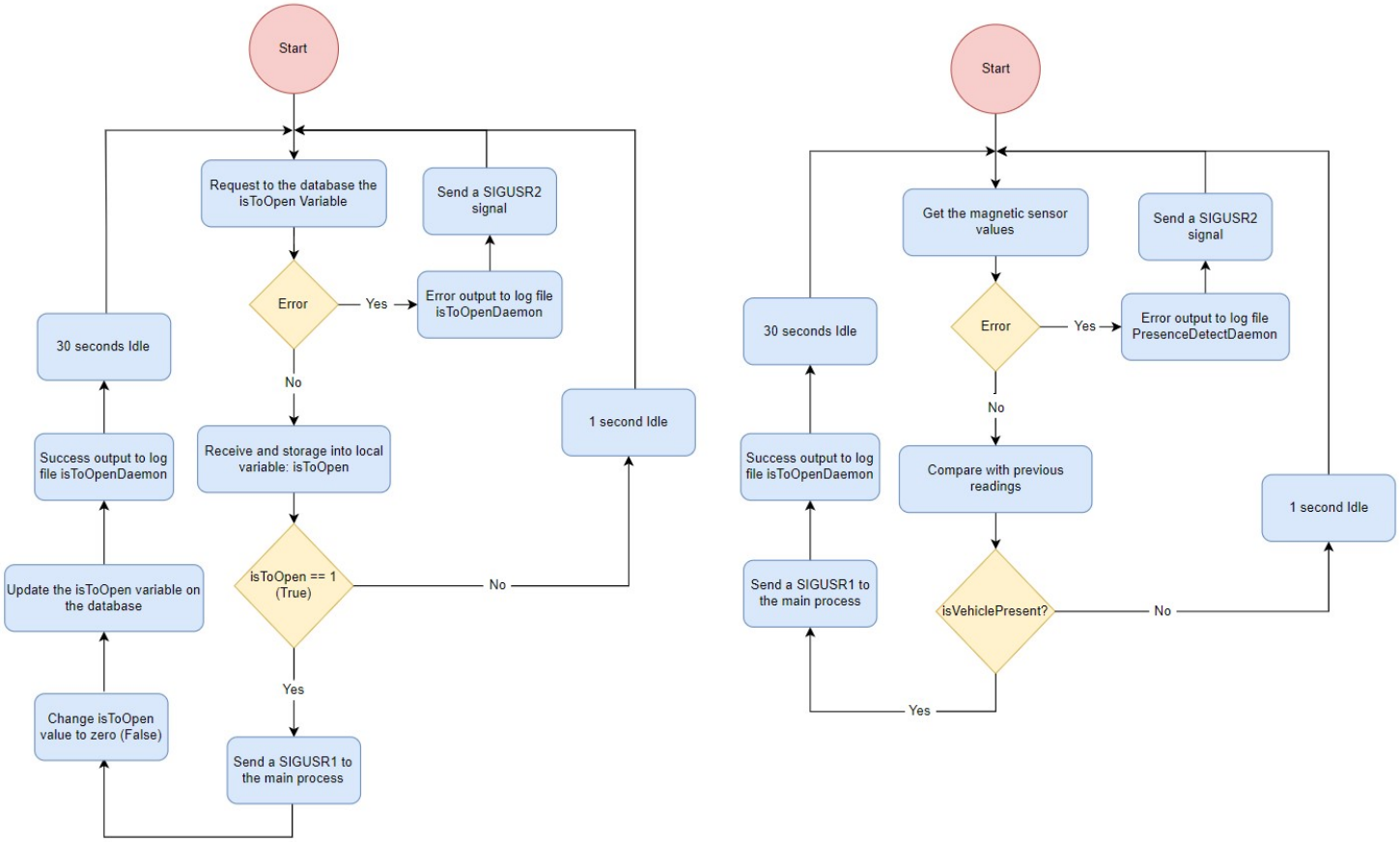


Figure .7: *
OpenGateDB and PresenceDetect Flowchart (Augmented)