BUSIT

Masters in Industrial Electronics and Computers Engineering

Embedded Systems

University of Minho

Authors:

Marco Cunha PG50583
Matheus Costa PG50649

Professor:

Adriano Tavares

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

BLE Bluetooth Low Energy

CPU Central Processing Unit

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System

FTDI Future Technology Devices International Limited

GUI Graphical User Interface

GPIO General Purpose Input Output

HTTP Hypertext Transfer Protocol

ICMP Internet Control Message Protocol

IEEE Institute of Electrical and Electronics Engineers

IPC Interprocess Communication

LCD Liquid Crystal Display

LCM Liquid Crystal Module

LED Light-Emitting Diode

LPDDR Low-Power Double Data Rate

OS Operating System

POSIX Portable Operating System Interface

QR Quick Response

RAM Random Access Memory

SRAM Static Random Access Memory

SD Secure Digital

SPI Serial Peripheral Interface

TCP Transmission Control Protocol

TTL Transistor-Transistor Logic

UART Universal Asynchronous Receiver-Transmitter

UDP User Datagram Protocol

USB Universal Serial Bus

USART Universal Synchronous/Asynchronous Receiver-Transmitter

UML Unified Modeling Language

WiFi Wireless Fidelity

1 Introduction

1.1 Problem

In big cities, people are always in a hurry and do not have time to stop and enjoy the little things in life or build a relationship. At this pace, people tend to become isolated and stressed. A city should be a place where people can come together more easily and build relationships in a safe environment, without traffic noise or big streets with light signals.

A robust transportation system can make all the difference to a city's success and people's social lives, and reduce its environmental footprint. Nowadays, however, transit systems in big cities are full of problems and their public opinion along with public transport have always been considerable negative by the population.

One of the problems is the lack of traffic management, which occurs mainly during rush hours, when there are obstructions in the road or during large events when some roads are closed. This leads to the increase of traffic congestion, the travel time and the air pollution.

Yet another issue is the long waits for transportation and the lack of public transportation, as mentioned before. Some people have to get up more than a hour and a half earlier in the morning just to catch the public transportation on time or because the trip time is very long. In another case, after work, a lot of people have to wait inconveniently more than fifteen minutes for transportation to arrive. These scenarios encourage people to use their own personal cars.

Technology can contribute greatly to solving these transportation problems. For example, data analytic allows transportation planners to determine the exact transportation needs of a city's residents and tracks how those needs change over the course of a day or season, so that resources can be allocated precisely where they are needed most.

Not only that dynamic analysis could act in the management of urban transportation but also in the whole city's traffic through the implementation of a semaphores controlling system and instantaneous feedback to drivers about the best route considering all the traffic analysis of the city.

1.2 Solution

Busit will be a interactive traffic manager that allows a automatic and dynamic management of public transport, focusing primordially on buses. The system intends to update buses routes in real time based on the actual transit and by the feedback of passengers in stations.

The system seeks to improve a city traffic, as well as creating a more health environment by encouraging the use of public transport instead of personal vehicles by the population.

An example of application would be when someone is standing in the bus station and sends the information to an app about where it wants to go and, with that information along with the location of the user, the Busit informs the driver that could change its route about the existence of a passenger in a stop that normally has few movement. In this scenario, with no people at the station the driver could just continue with another (and faster) route, and only when there really is a passenger that the driver goes to the station, as shown in figure 1.

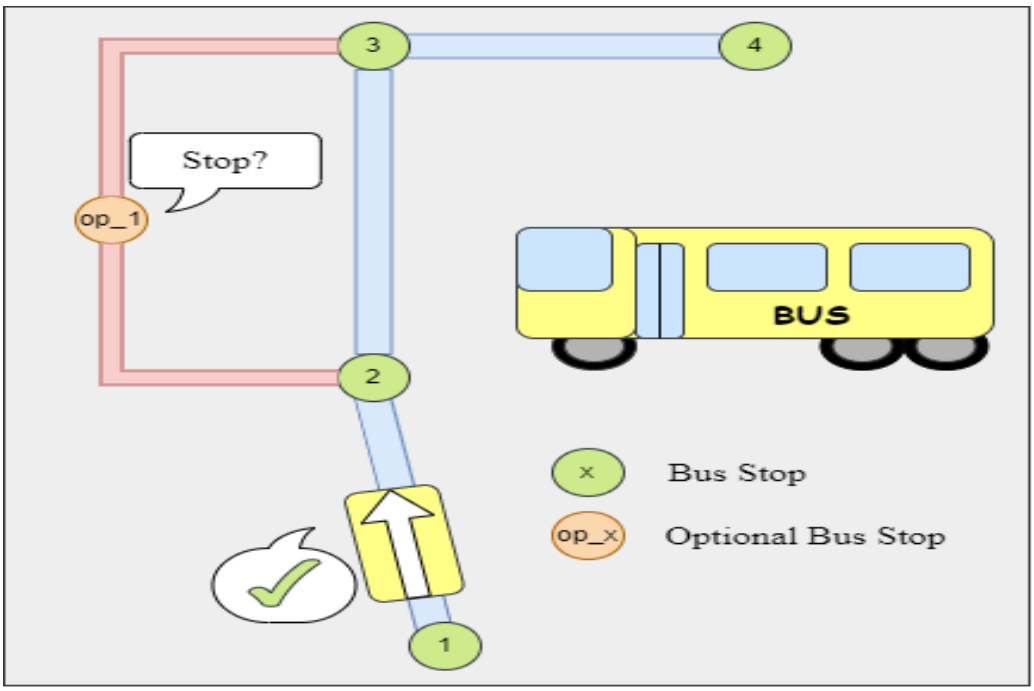

Figure 1: Example Illustration

1.2.1 Solution Analysis

In other to simplify the complex problem into smaller ones, the following chart was created (figure 2):

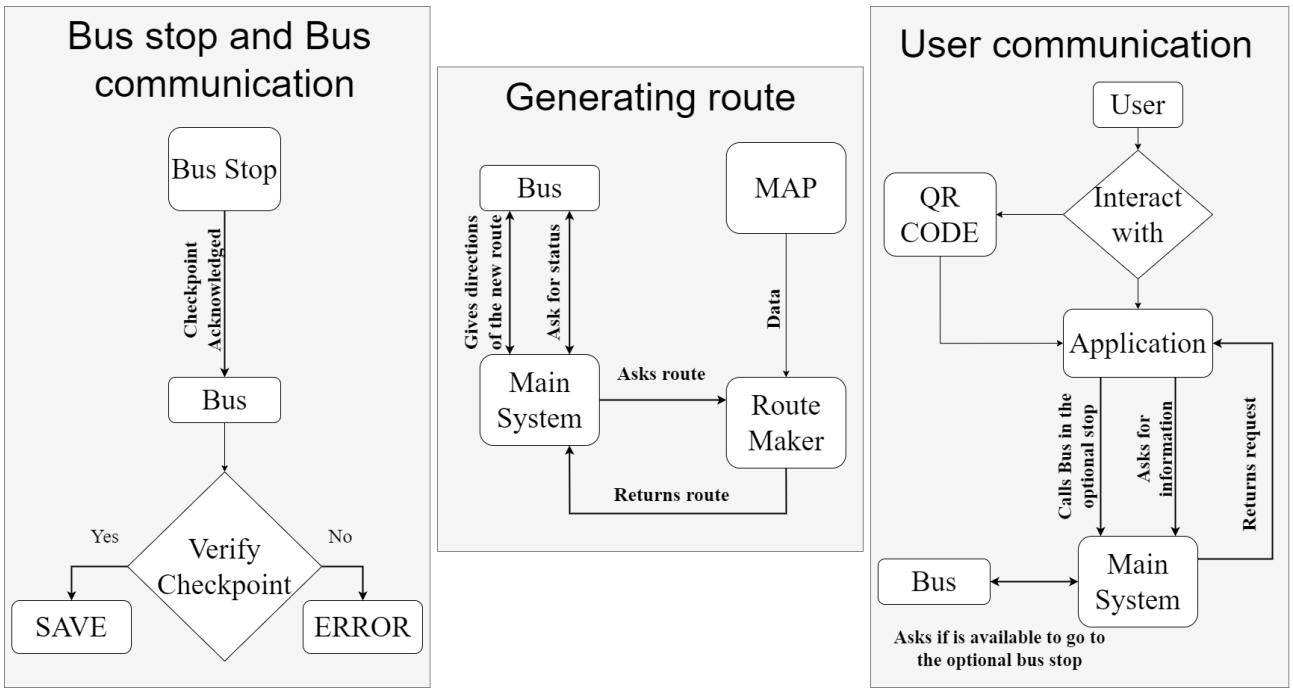

Figure 2: Solution Analysis Chart

- **User:** The user interacts with the system by scanning the QR code in the bus stop inside the application, where it waits for the system to answer.
- **Bus Stop:** The bus stop has the QR code to be scanned by the user and communicate with the bus when it arrives in order to update the bus local data.
- **Main System:** The main system only receives data directly from the application as a request from a user in a bus stop, but it sends data to all other subsystems when needed.
- **Route Maker:** The route maker consists in a software that has real time information about the traffic and has the city map configured in its memory and when requested by the main system calculates the best route to send to the bus.

2 Target Market

The Busit seeks to attend the city's population needs, such as students at school, workers, retirees, tourists, and more. Despite focusing on buses, it could also apply in a private economy by making a partnership with private transportation companies, even micromobilities.

3 Market Research

3.1 Bus Market

According to the IMARC Group (the leading market research company that provides market and business research intelligence across the globe), the global bus market in 2021 reached a value of US$ 43.84 Billion and it keeps tending to increase reaching about the value of US$ 68.79 Billion by 2027, exhibiting a growth rate of 7.20% during 2022-2027, as showing in the image below (figure 3).

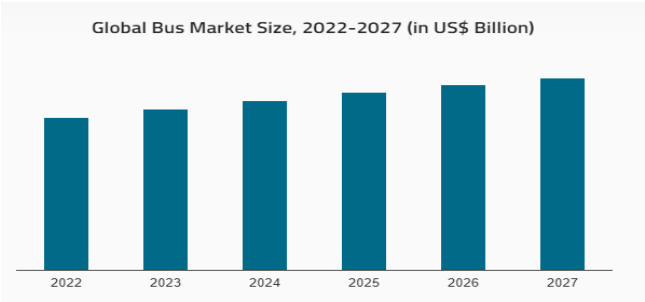

Figure 3: Global Bus Market Size, 2022-2027 (in US$ Billion) [7]

Due to the environmental problems of the planet, people tend to use public transport more often as a means of transportation, being this one of the key factors bolstering the growth of the bus market.

They are also a great advertising method for companies or to spread political messages and public information campaigns. In addition, Buses are operated by professional drivers creating employment and reassure the safe transportation to the public as they have a lowest accident rate then other automobiles.

As the market rises, so do its technologies and methods, so some transportation companies are focusing on management strategies on their systems to reduce road congestion such as concessionary travel bus passes, rising automation in the transportation industry, the increasing use of big data to optimize routes, vehicle dispatch and smart traffic light.

3.2 Competition Market

The existing softwares that Busit can compete/ relate are well established on the industry and are very useful for checking the public transportation's hours, keeping track of the real time traffic and choosing the best route to a destination.

Although those companies are specialized on making routes based on the actual traffic, the user needs to adapt to the system (figures 4 and 5), but Busit wants to improve the traffic by adapting the public transport routes and making the system adapt to the population's needs and that's how the product wants to stand against the concurrency.

Figure 4: Transit • Subway & Bus Times APP [1]

Figure 5: Moovit: Public Transport [11]

4 Requirements

4.1 Functional Requirements

- Able to generates routes autonomously;
- Must have the potential to diverge to an optional route if necessary;
- Show schedules and other information about the transport.
- Be able for users to request boarding at optional stops.

4.2 Non-Functional Requirements

- Be user friendly;
- Be adaptable for different cities or transports;
- Secure;
- Easy maintenance;
- Wireless communication;
- Reliable;
- Inclusive.

5 Constraints

5.1 Technical Constraints

- Use raspberry Pi as development board;
- C++ as programming language;
- Pthreads implemented on the device;
- Buildroot to a custom linux image.
- Device-Drivers implementation.
- Makefiles.

5.2 Non-Technical Constraints

- Group of 2;
- Limited budget for prototyping;
- Validation and Certification can be compromised due to non-professional tools used;

6 System Overview

The system overview shows a representation of how the different subsystems interact with each other (figure 6) and allows a better understanding of the product intentions.

The **bus stop** shall have a QR code that will be scanned by the user in the application and that application will send the information to the server.

The **server**, in turn, will process the data and, along with the traffic information, will calculate if it is possible for a near bus to change its route to the specified optional bus stop.

Each bus will have a microcontroller that communicate with the server to update the actual route with a better one or to add a new optional stop.

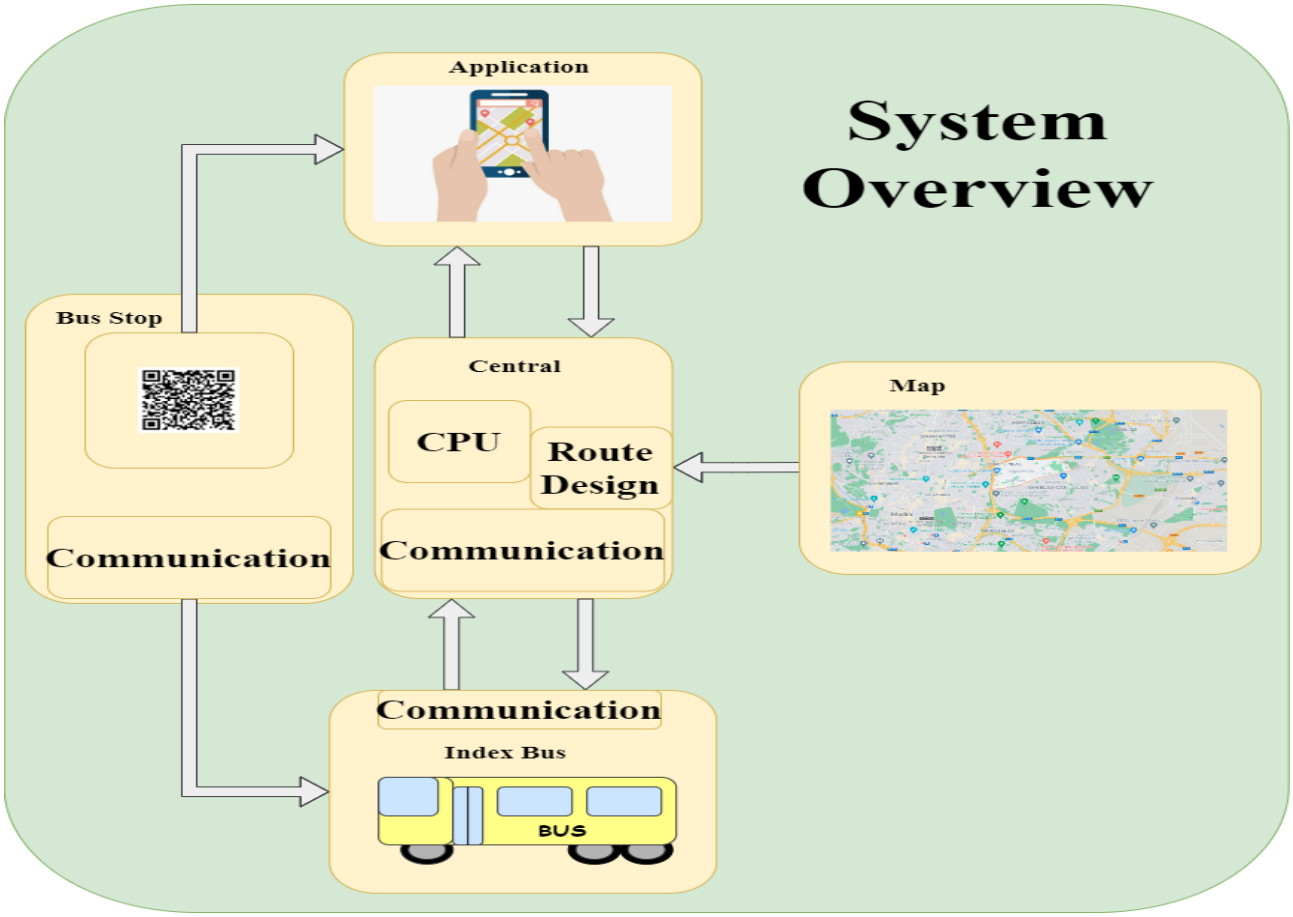

Figure 6: System Overview Representation

7 System Architecture

In order to better understand how the BUSIT will work, a system architecture has been conceived. Such system architecture was divided into hardware and software architectures.

7.1 Hardware Architecture

As shown in figure 7, the main system will be located in the **Raspberry Pi 4 Model B** and such system will communicate with the different clients by a **WiFi Module**.

The raspberry will send as output **Status LEDs** to inform which task is executing, for example the communication with a specific client or data processing, and a **Display** to inform different kinds of data, for example the default routes that currently exists.

There is also an external **Global Server** to store important data and to process specific data in order to make efficient the multitasking in the main system.

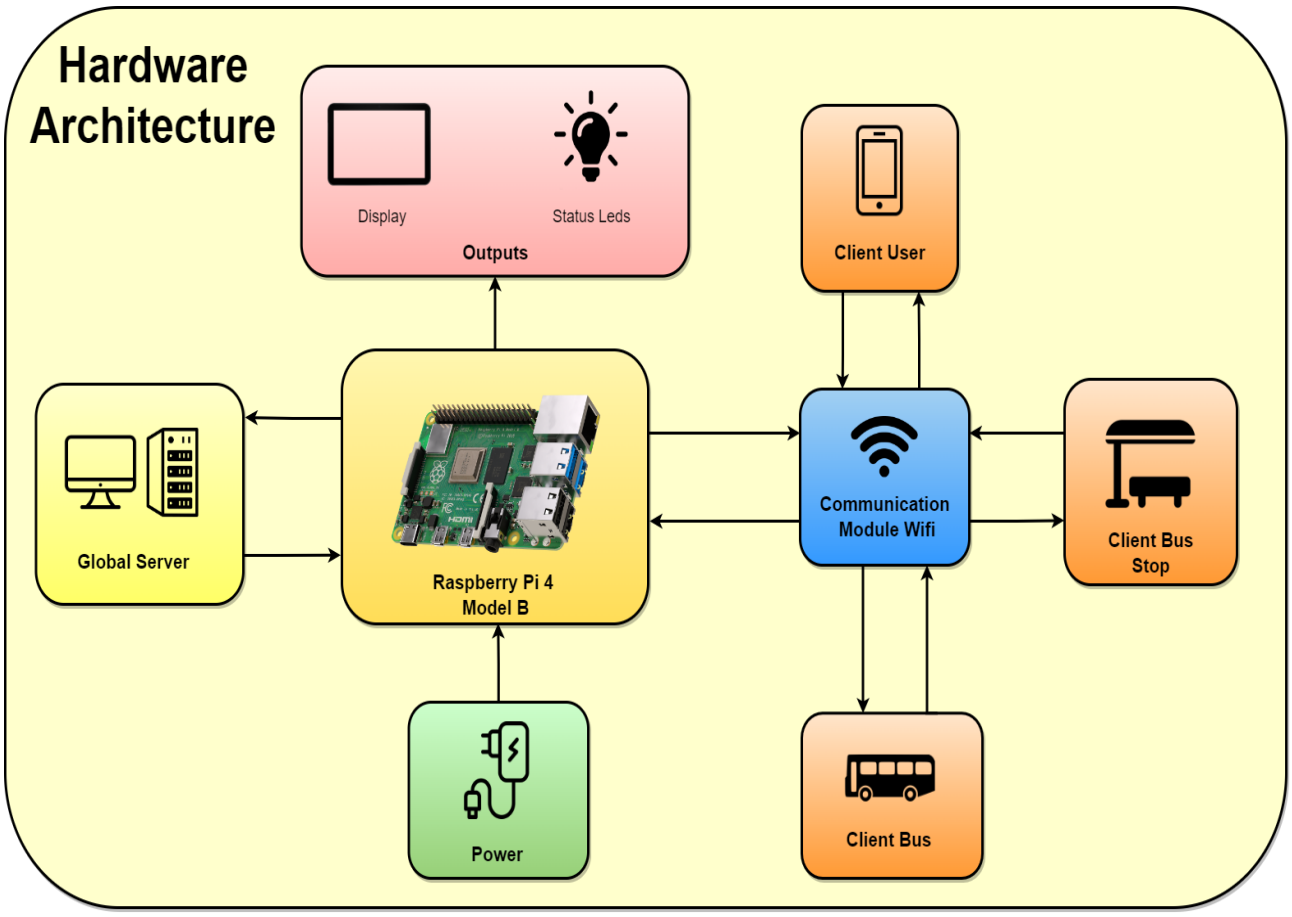

Figure 7: Hardware Architecture Representation

7.2 Software Architecture

In the software architecture there will be three layers: Application, Middleware and Operating System, as shown in figure 8.

In the **Application** layer, it will be implemented the high-level Application Programming Interface (API) for all different sub-systems, from the Smartphone GUI to the Status LEDs.

In the **Middleware** layer is where most of the code will be, since it is, as the name suggests, in the middle. The Database Management and the Pthreads implementation are some of the modules inside the layer.

In the **Operating System** layer, the lowest-level layer, one can encounter all the drives that the upper layers need in order to make the bridge between the software above and the hardware below. The Display driver and the WiFi driver are examples of drivers that need to be implemented.

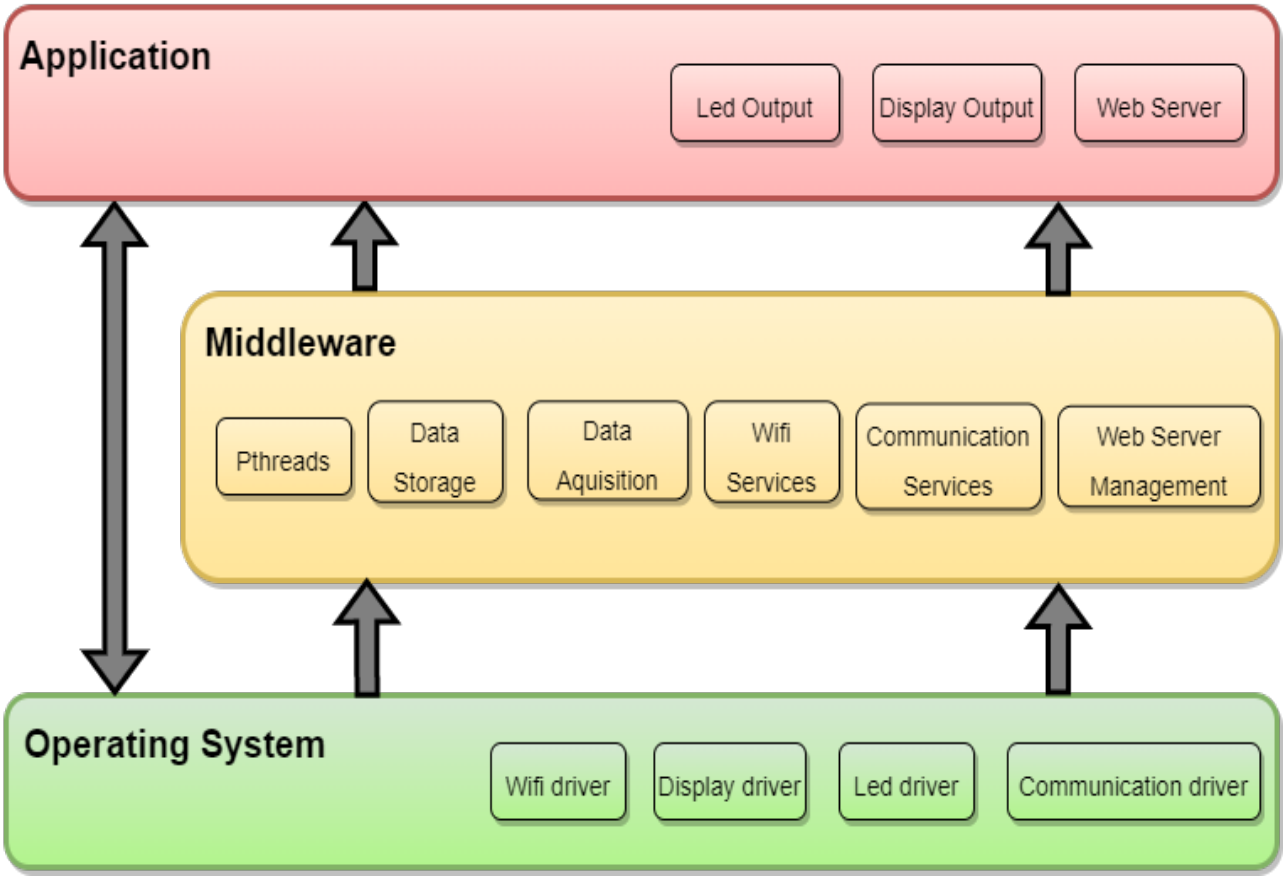

Figure 8: Software Architecture Representation

8 System Analysis

To understand how the different actors interact with the system and how the different subsystems communicate with each other, a system analysis was made and divided in the four topics: the **events table**, the **use cases**, the **state charts** and the **sequence diagrams**.

8.1 Events Table

To understand the system behavior, the most significant events must be defined, as well as how the system should deal with these events. In the table 1 the main events on the main system and the corresponding System Response can be seen.

Table 1: Local System Events

Event	System Response	Source	Type
Make default routes	Generate new default routes	Local System	Asynchronous
Make optional/ single route	Generate new routes	Local System	Asynchronous
Get route	Search and acquire a specific route	Local System	Asynchronous
Set bus route	Assign a specific route to a certain bus	Local System	Asynchronous
Update fleet	Updates all bus routes	Local System	Asynchronous
Removing route	Removes a specified route	Local System	Asynchronous
Request bus stop ID	The bus requests the bus stop ID	Remote System	Asynchronous
Set bus	Adds a bus to the fleet	Local System	Asynchronous
Set bus stop	Adds a bus stop to the list	Local System	Asynchronous
Remove bus	Removes a specified bus from the fleet	Local System	Asynchronous
Remove bus stop	Removes a specified bus stop from the list	Local System	Asynchronous
Send bus position	The bus sends a location of his position	Remote System	Synchronous
Get bus log	Saves the log of every bus to the data base	Local System	Synchronous
Request bus status	Requests the status of a specified bus	Remote System	Asynchronous
Request bus schedule	Get the schedule of a specified bus	Remote System	Asynchronous
Request optional bus stop pick up	The bus stop requests a pick up to nearby buses	Remote System	Asynchronous
Read QR-code	It activates the QR-code reader	Remote System	Asynchronous

8.2 Uses cases

The use cases diagrams can help to understand the system behavior when interacting with each user (known as actor). In BUSIT there are five actors: the **user**, the **administrator**, the **database**, the **bus stop** and the **bus**, as shown in figure 9.

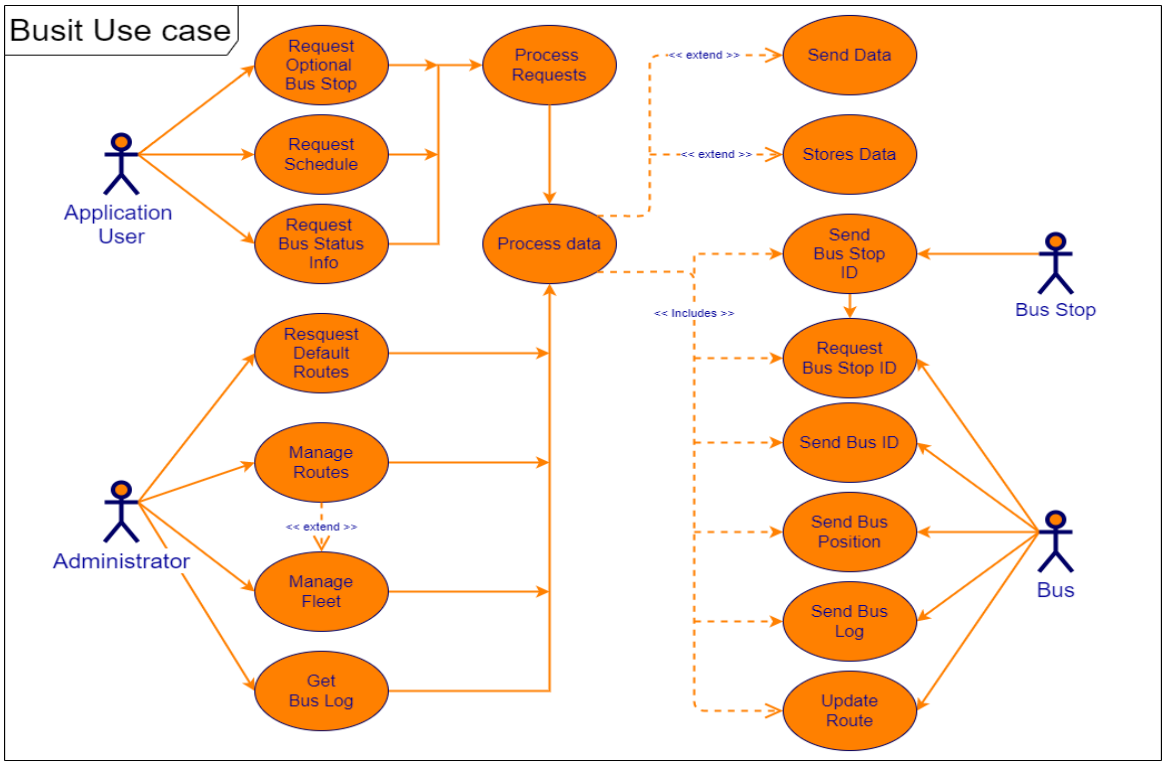

Figure 9: Use case diagram

- **User:** The user interacts with the system only by requests and it can request for an optional bus stop, a bus schedule or a bus status.
- **Administrator:** The administrator is responsible for the management of routes and fleets and also sends data to be processed in the main system.
- **Database:** The database stores and sends information about the city map, as well as the default routes, the bus stops location and the buses identification.
- **Bus Stop:** The bus stop sends its identification when requested by other processes.
- **Bus:** The bus can have its route updated, can request for a bus stop identification, and also send its current position, its own identification and the activity record in a day.

8.3 State Chart

The state charts are extremely important to comprehend the system and subsystems behaviour. For BUSIT it was conceived three state machines to represent its behaviour: the **Generate Route**, the **Bus Stop and Bus Communication** and the **Application**.

8.3.1 Generate Route State Machine

The Generate Route State Machine is implemented in figure 10 and it has five states: IDLE, READY, ROUTE MAKER, UPDATE ROUTE and STOP. The variables that are responsible for state transitions are *is_routing*, *data_acquire*, *route_finished*, *Error_x*, *Error_ready*, *Error_update* and *Error_MR*. Each state description is in table **??**.

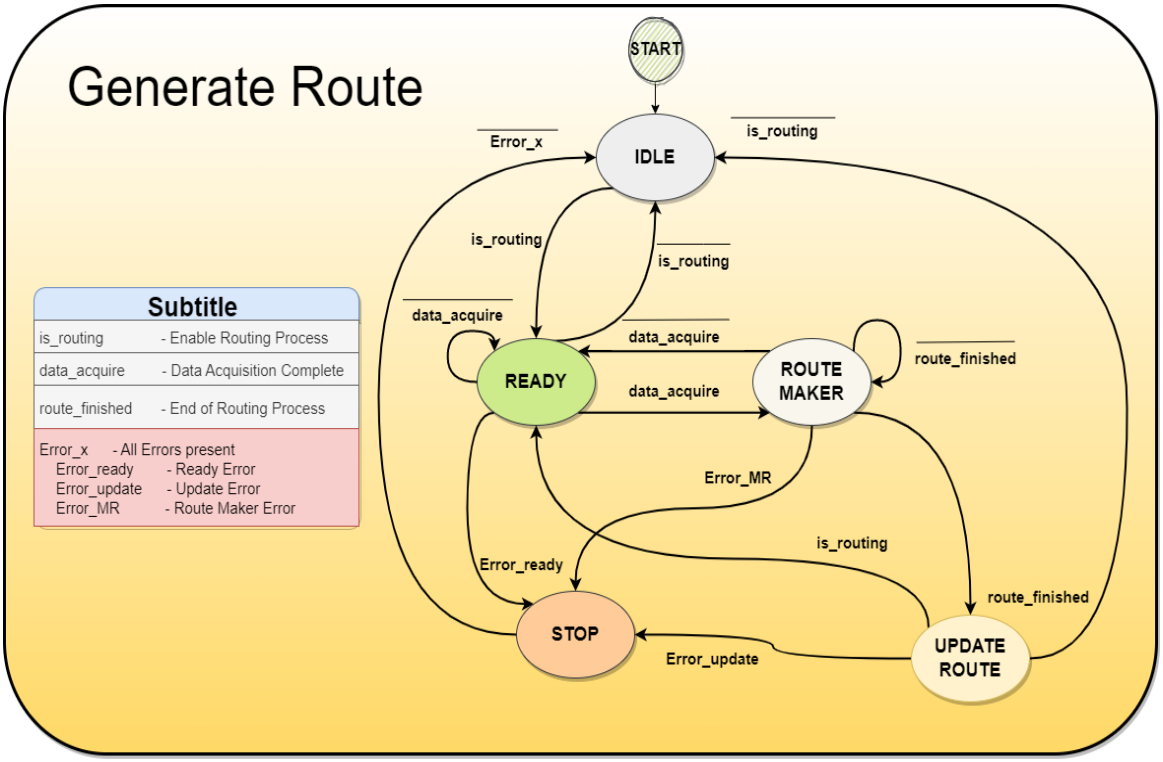

Figure 10: Generate Route state machine Representation

- **IDLE:** The system stays in this state until occurs a order or a request to generate a route ("is_routing" = true), enabling the routing mode;
- **READY:** The system in this state is ready to start generating a route, but it wait for all data acquire to process and be validated, once it is finished with no error occurred ("Error_ready" = false) and with the system still in routing mode ("is_routing" = true), it will start generating the route and jump to the "ROUTE MAKER" state;
- **ROUTE MAKER** In this state, the system is generating the route based on the information processed in the previous state ("READY"), if the system cannot generate a route

or if, for some reason, the information wrongly validated, them the system terminates the process and jump to "STOP" state (Error_MR = true), although if a route is successfully created them the system jumps to "UPDATE ROUTE" to save it ("route_finished" = true);

- **UPDATE ROUTE** In this state, the system saves the route created and, if necessary, update it to the fleet or the specific bus, them if the routing mode is still active, it will jump to "READY" state, if not, jump to "IDLE" state, but if a error occurs it will jump to "STOP" state ("Error_update" = true);

- **STOP:** If the system jumps to this state, them it means that a error has occur, so he will identify it and report it, them he disable the routing mode ("is_routing" = false) and jump to "IDLE";

8.3.2 Bus Stop and Bus Communication

The Generate Route State Machine is implemented in figure 11 and it has five states: IDLE, Request Bus Stop ID, Next Route, Generate Route and STOP. The variables that are responsible for state transitions are *id_request*, *id_validated*, *route_finished*, *Error_x*, *Error_MR* and *Error_RBSID*.

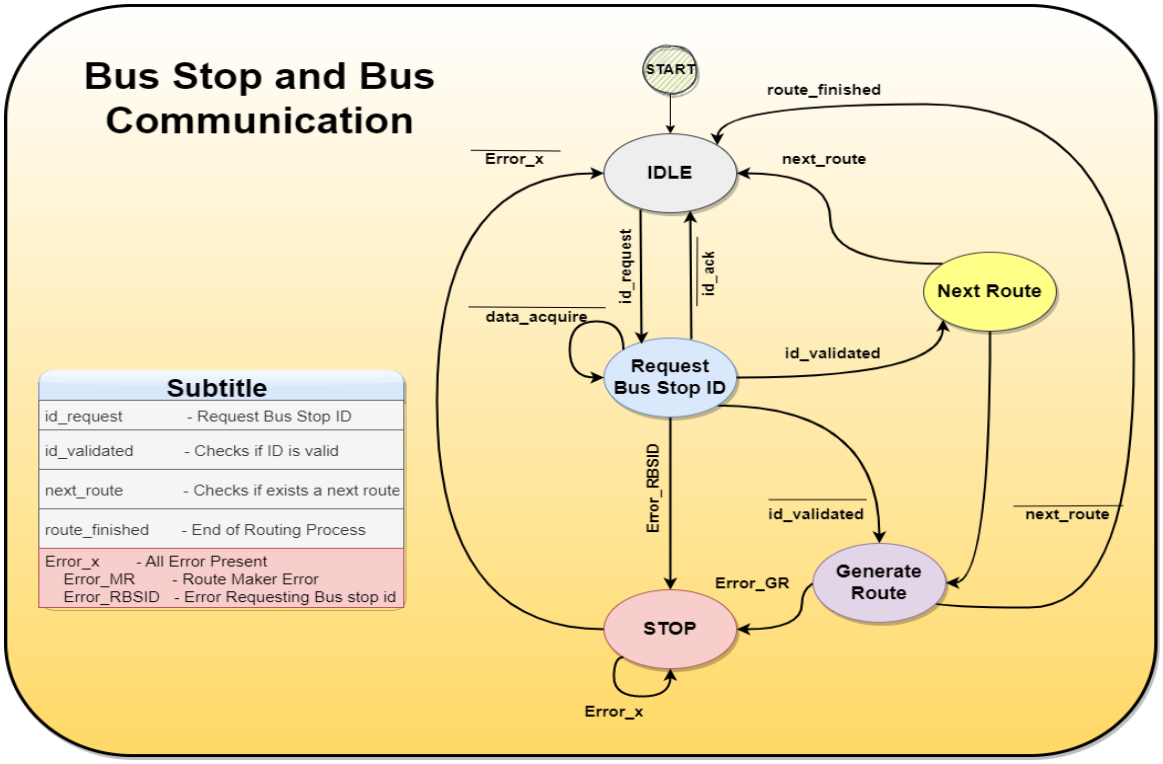

Figure 11: Bus Stop and Bus Communication state machine Representation

- **IDLE:** The system stays in this state until occurs a request for Bus Stop ID ("id_request" = true) to start the process of communication.

- **REQUEST BUS STOP ID:** In this state, the system is requesting the Id of the closets bus stop from the bus that made the request, them it will compare with the id of the bus stop that is mark on the route of the bus, if it is the same them the system send a confirmation to the bus and he can proceed to the next route, making the system jump to the "NEXT ROUTE" state ("id_validated" = true), if is not the same the system jump to the "GENERATE ROUTE" state ("id_validated" = false), if occurs a Error the system will jump to the "STOP" state (("Error_RBSID" = true)).

- **GENERATE ROUTE:** If the system is in this state, it means that the bus is lost or their is no next route, so the system will generate a new route to correct him or get a another default route to follow, and them update it to him, making the system jump to "IDLE" state ("route_finished" = true), if an error occur the system will to the "STOP" state ("Error_GR" = true).

- **NEXT ROUTE:** If the bus is in this state, them it means that he kept following the correct path, and now the system will verify if there is another route, if is true them the system update the bus with the next route and jump to "IDLE" state ("next_route" = true), if not, the system jumps to the "GENERATE ROUTE" state ("next_route" = false).

- **STOP:** If the system jumps to this state, them it means that a error has occur, so he will identify it and report it and jump to "IDLE".

8.3.3 Web Application

The Web Application State Machine is implemented in figure 12 and it has six states: IDLE, Process Request, Get Schedule, Get Bus Info, Optional Bus Stop and Error Feedback. The variables that are responsible for state transitions are *ID_X*, *ID_BInfo*, *ID_Schedule*, *ID_OpBus*, *Error_x*, *Error_OpBus*, *Error_PRqs*, *Error_Sch* and *Error_BI*.

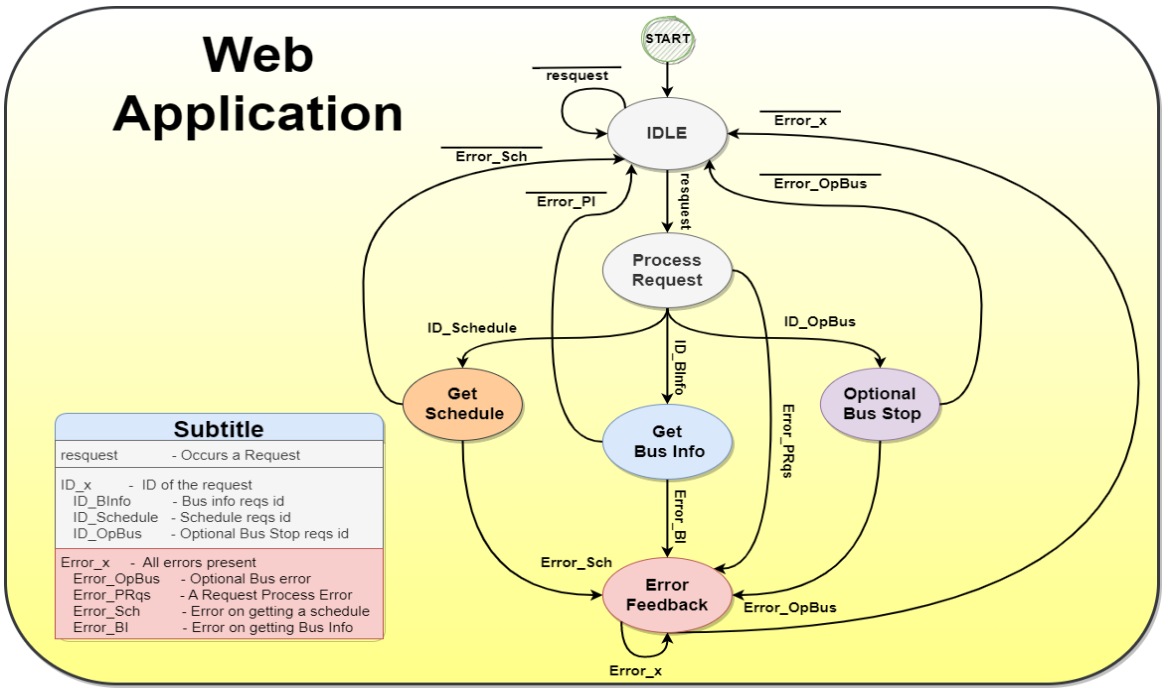

Figure 12: Application state machine Representation

- **IDLE:** The system stays in this state until occurs a request from the app client ("request" = true) so he can process it.

- **PROCESS REQUEST:** In this state the system will process the request made by the application client and them jump to the designated state: jump to "GET SCHEDULE" state if the ID is "ID_Schedule", jump to "GET BUS INFO" state if the ID is "ID_BInfo", jump to "GET OPTIONAL BUS STOP" state if the ID is "ID_OPBus", if an error occurs the system jumps to "ERROR FEEDBACK" ("Error_PRqs" = true).

- **GET SCHEDULE:** In this state the system will get the designated schedule of an specific Bus and return it to the application client, if no error occur in this process the system jump to the "IDLE" state("Error_Sch" = false) otherwise if an error occur the system will jump to the "Error Feedback" state.("Error_Sch" = true)

- **GET BUS INFO:** In this state the system will get the designated bus info of an specific Bus and return it to the application client, if no error occur in this process the system jump to the "IDLE" state("Error_BI" = false) otherwise if an error occur the system will jump to the "Error Feedback" state("Error_BI" = true).

- **OPTIONAL BUS STOP:** In this state the system will request a pick up to the three nearby buses in the area and return the answer to the application client, if no error occur in this process the system jump to the "IDLE" state("Error_OpBus" = false) otherwise if an error occur the system will jump to the "Error Feedback" state("Error_OpBus" = true).

- **ERROR FEEDBACK:** If the system jumps to this state, them it means that a error has occur, so he will identify it and report it and jump to "IDLE".

8.4 Sequence Diagram

The sequence diagram shows process interactions arranged in time sequence in order to understand the flow of communication between different subsystems.

8.4.1 Generate Route Sequence Diagram

The Generate Route Sequence Diagram is implemented in figure 13 and it has four processes: Main System, Route Maker, Map and Bus.

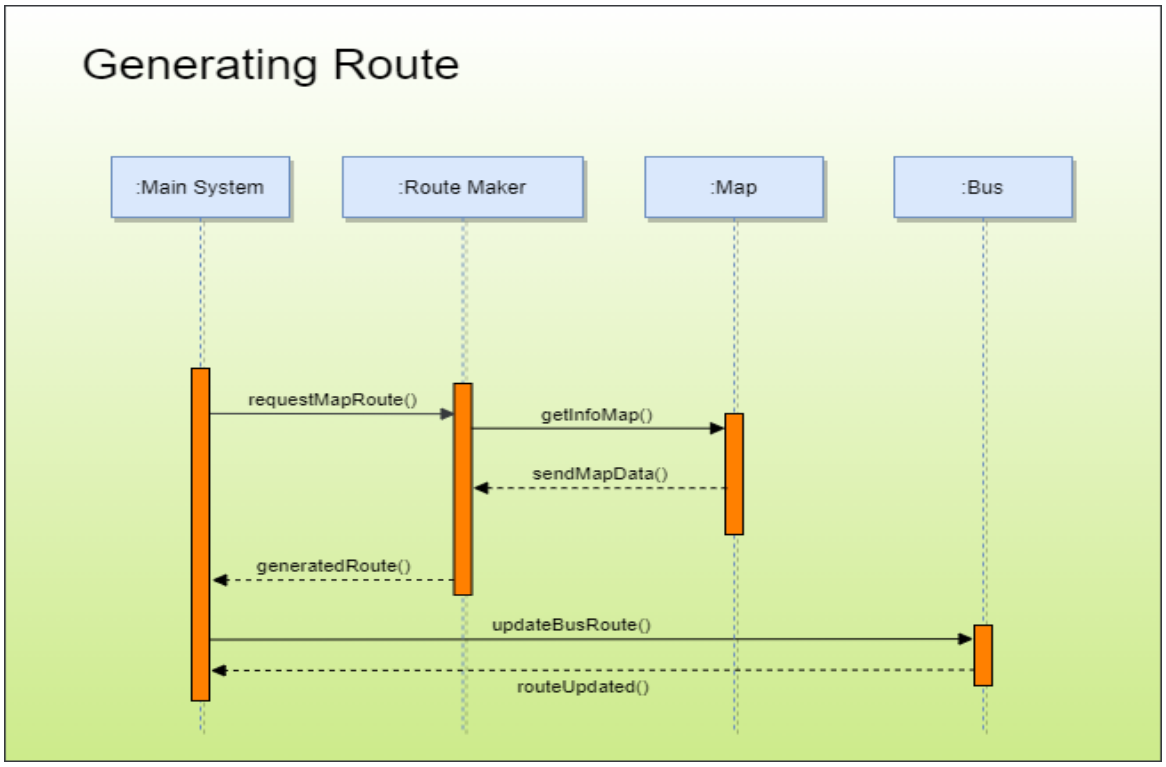

Figure 13: Generate Route sequential diagram Representation

8.4.2 Bus Stop and Bus Communication Sequence Diagram

The Bus Stop and Bus Communication Sequence Diagram is implemented in figure 14 and it has three processes: Bus, Main System and Bus Stop.

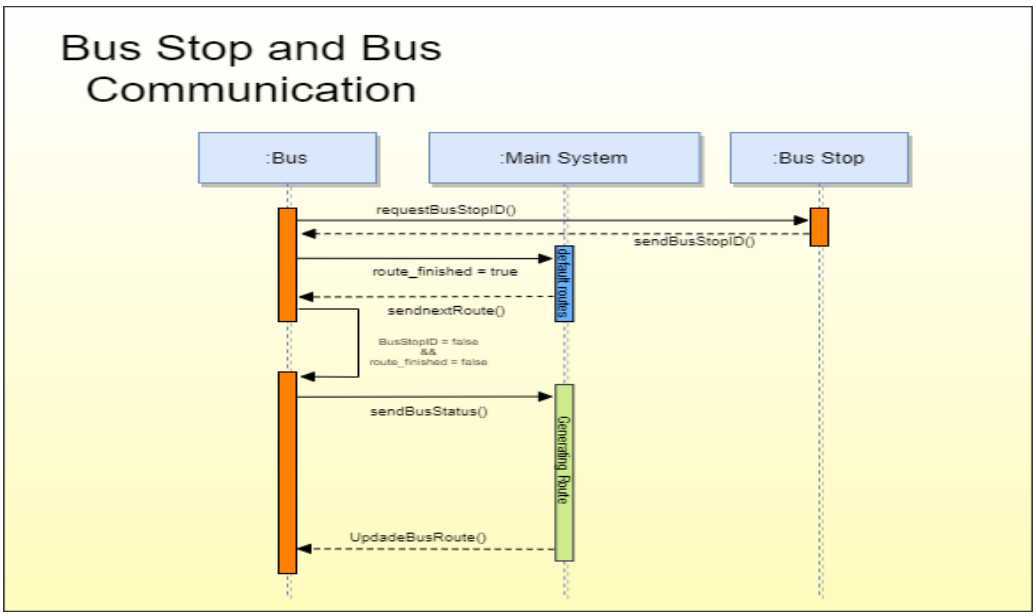

Figure 14: Bus Stop and Bus Communication sequential diagram Representation

8.4.3 Web Application Sequence Diagram

The Application Sequence Diagram is implemented in figure 15 and it has four processes: User, Server, Main System and Bus.

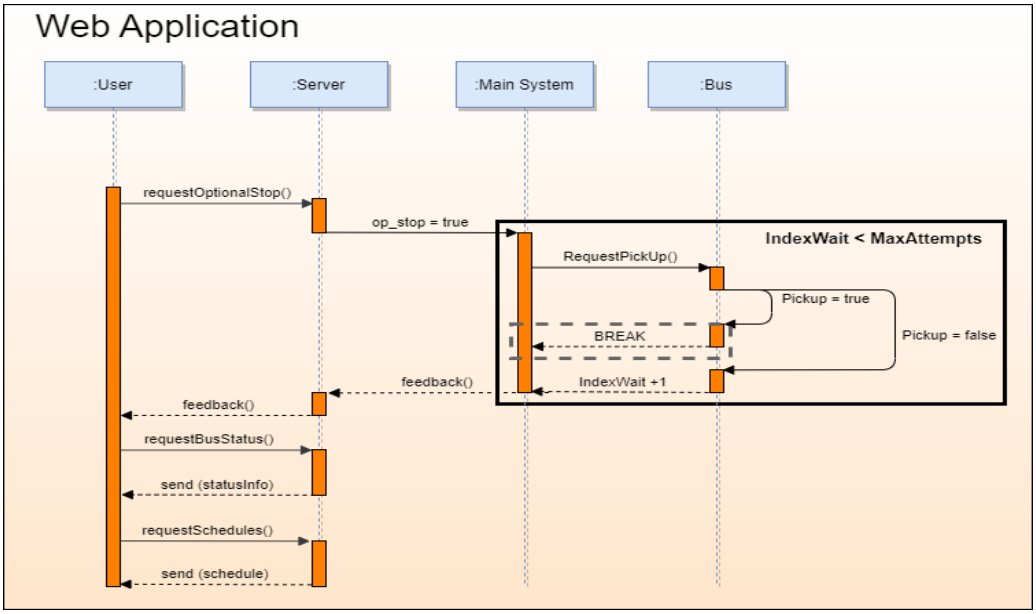

Figure 15: Application sequential diagram Representation

9 System Design

9.1 Algorithms

The BUSIT algorithms is a core part of the project, since it will dictate how the system and subsystems have to behave, as well as the communications between them. The main algorithms that needs to be strictly followed on implementation to grant that the base structure can work in the most predictable and correct way are **Generate Route**, **Bus and Bus Stop Communication** and **Optional Bus Stop**.

9.1.1 Generate Route: Simple and Multiple Route

There are two ways to generate routes, one of them is the *Simple Route*, which is the normal "start to finish" route, and the other is the *Multiple Route*, which generates, in sequence, a set of simple routes and then joins them into one.

The following figure 16 explains the algorithm behind the two options:

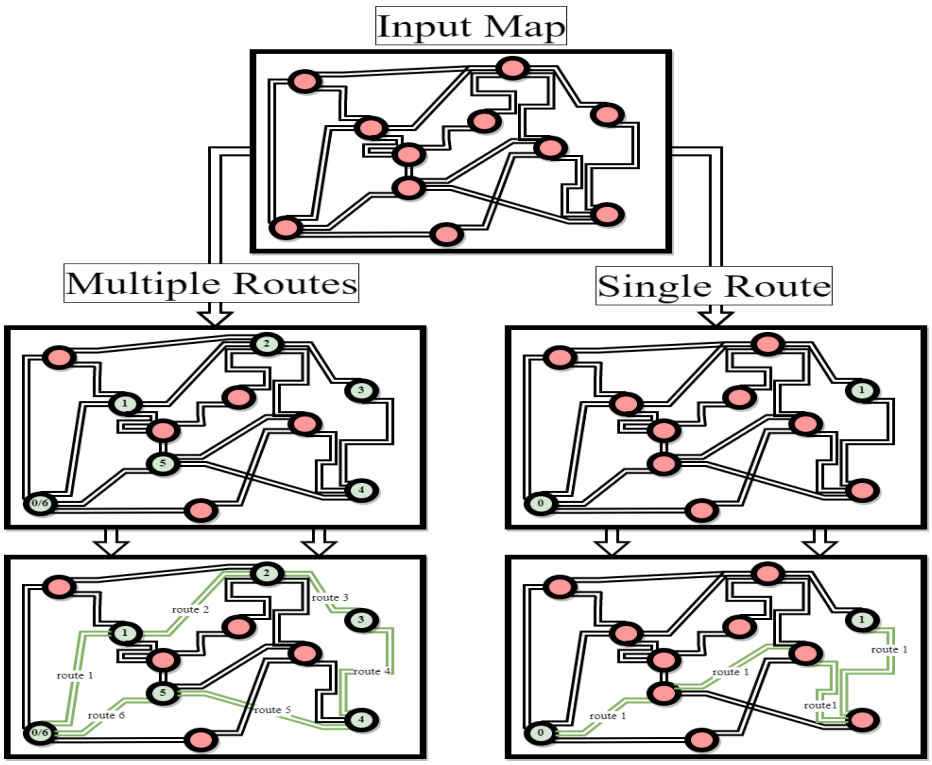

Figure 16: Multiple and Simple Route

Algorithm:

1 - Input the map information.

2 - Define what the operation is (Simple Route or Multiple Route).

3 - If the option is Simple Route:

- **a -** Define the 2 nodes (start and end) and them generates a route.

4 - If the option is Multiple Route:

- **a -** Define the set nodes sequentially and them generates the route to all of them starting with the first add node to the last.

9.1.2 Generate Route: Make Route

The follow algorithm explains the steps to generate a route: (figure 17)

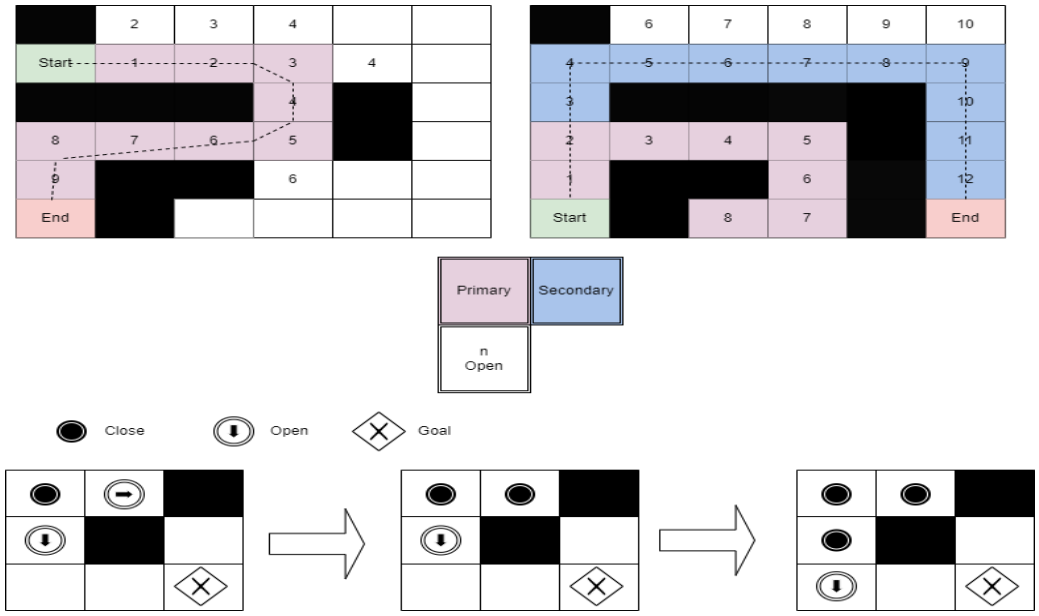

Figure 17: Make Route Algorithm

Algorithm:

1 - The system identifies the route start and gives a gain of 0 to that position.

2 - Identify the coordinates of the present location (node) and its gain and save it.

3 - If the coordinates are the same as the goal:

a - The system stops and jumps to the step 7.

4 - If not in the goal location:

a - Define the set nodes sequentially and them generate the route to all of them starting with the first add node to the last.

b - If the system only opens one node:

i - The system jumps to the open node, close it, save it, and increment the gain by 1.

c - Else if the system opens more them one node:

i - The System marks the node, saying that he has more than one open direction.

ii - The system takes a decision to dictate the best direction for the route.

iii - The system jumps to the decided node, close it, save it, and increment the gain by 1.

d - Else if the system doesn't open any nodes and the present node is not the start (gain $>$ 0):

i - The system jumps to the previous node where at least one direction is open.

ii - Decrements the gain by 1 for each node returned and jumps to step **4.a)**.

e - Else if no open nodes are found and the system is on the start (gain $=$ 0):

i - The system activates a error flag and proceeds to step **5**.

5 - If the present node is not the same as the goal and no error occurs:

a - The system jumps to the step **4.a)**.

6 - If some error occurs:

a - The system informs that the route couldn't be created.

7 - If the present node is the same as the goal and no error occurs:

a - The system informs that the route was created and saves it.

9.1.3 Generate Route: Optimize Route

The follow algorithm explains the steps to Optimize the route: (figure 18)

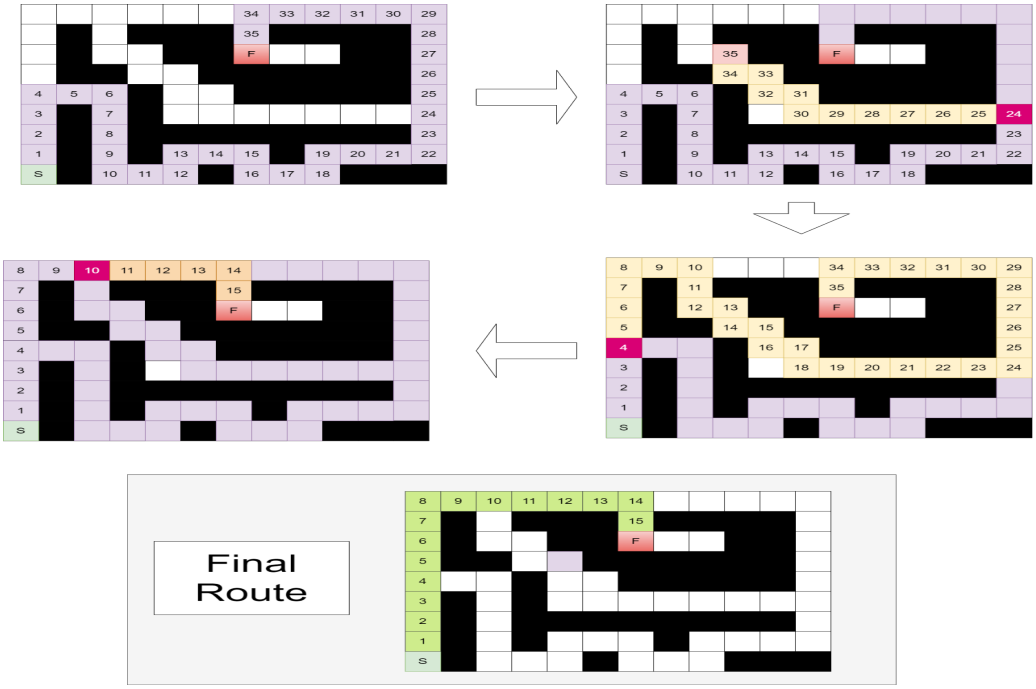

Figure 18: Optimize Route Algorithm

Algorithm:

1 - Searches for the previous node with the highest gain that has at least one open direction.

2 - If no node is found:

  a - Saves the route

3 - If a node is found:

  a - Tries to generate a route from that node with the gain less than the current route.

  b - If the route was generated:

    i - The system saves the new route and its characteristics.

  c - The system jumps to the step **1**.

9.1.4 Generate Route: Decision Maker priorities

The system decision is based the following priorities, "1" being the highest:

1 - Direction: If the next node has one direction and is towards the last, the jump must be refused;

2 - Distance: The system must recognize the closest node to the goal and jumps towards him. If two close nodes have the same distance to the goal the system doesn't jump to any of them;

3 - Road Reserved to buses: The system must recognize only buses roads and jumps to them or the first one that he found;

4 - Default: If no conditions is accepted the system jumps to the first opened node;

9.1.5 Generate Route: Flowcharts

In order to visualize the algorithms showed previously more clearly, flowcharts were created, since they are a very useful tool to visualize steps in a process.

Simple and Multiple Routes

The following flowchart (figure 19) dedicates to create a simple or multiple routes.

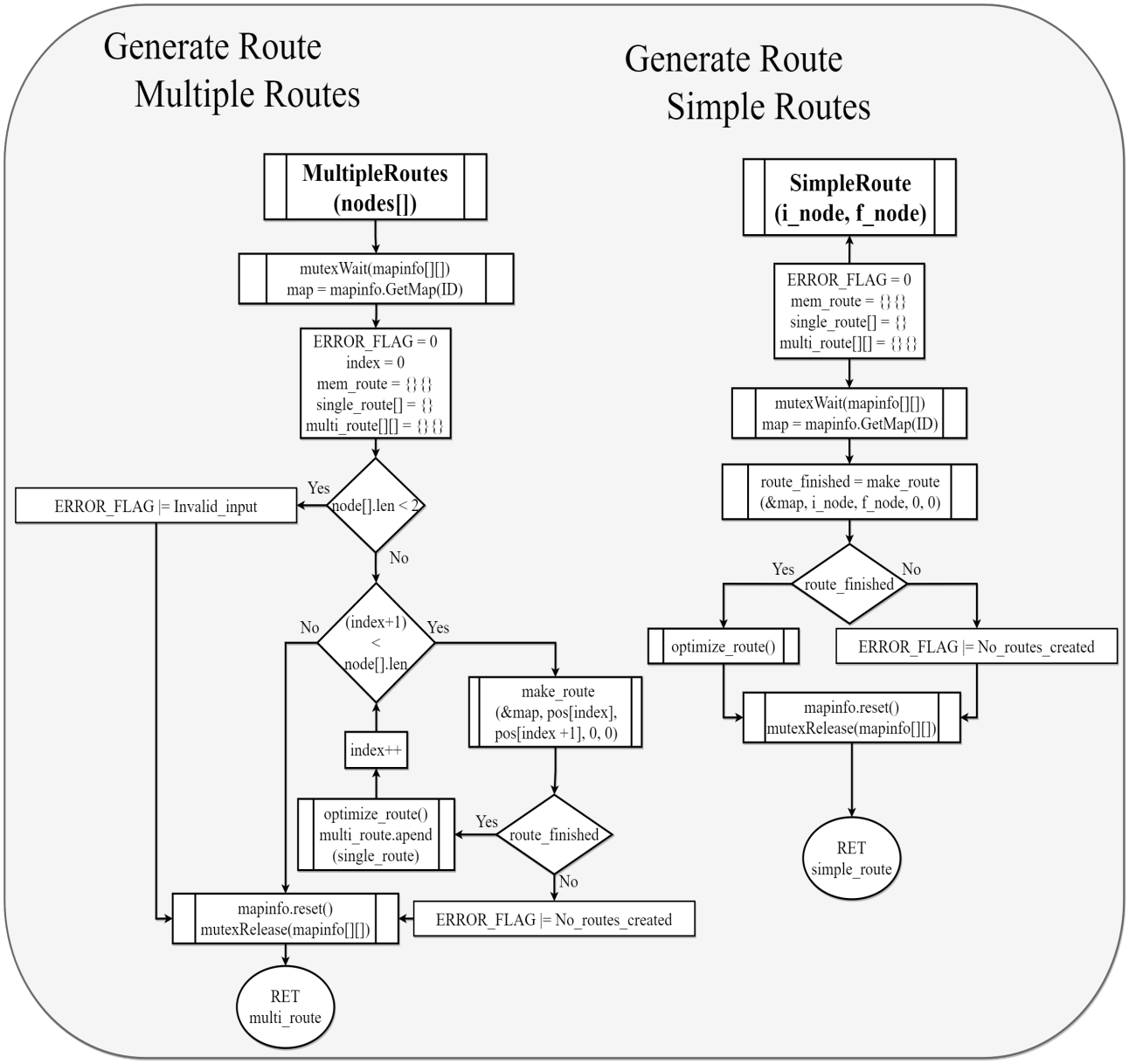

Figure 19: Multiple and Simple Routes Flowchart

Make Routes

The following flowchart (figure 20) dedicates to implement the *make_route* function.

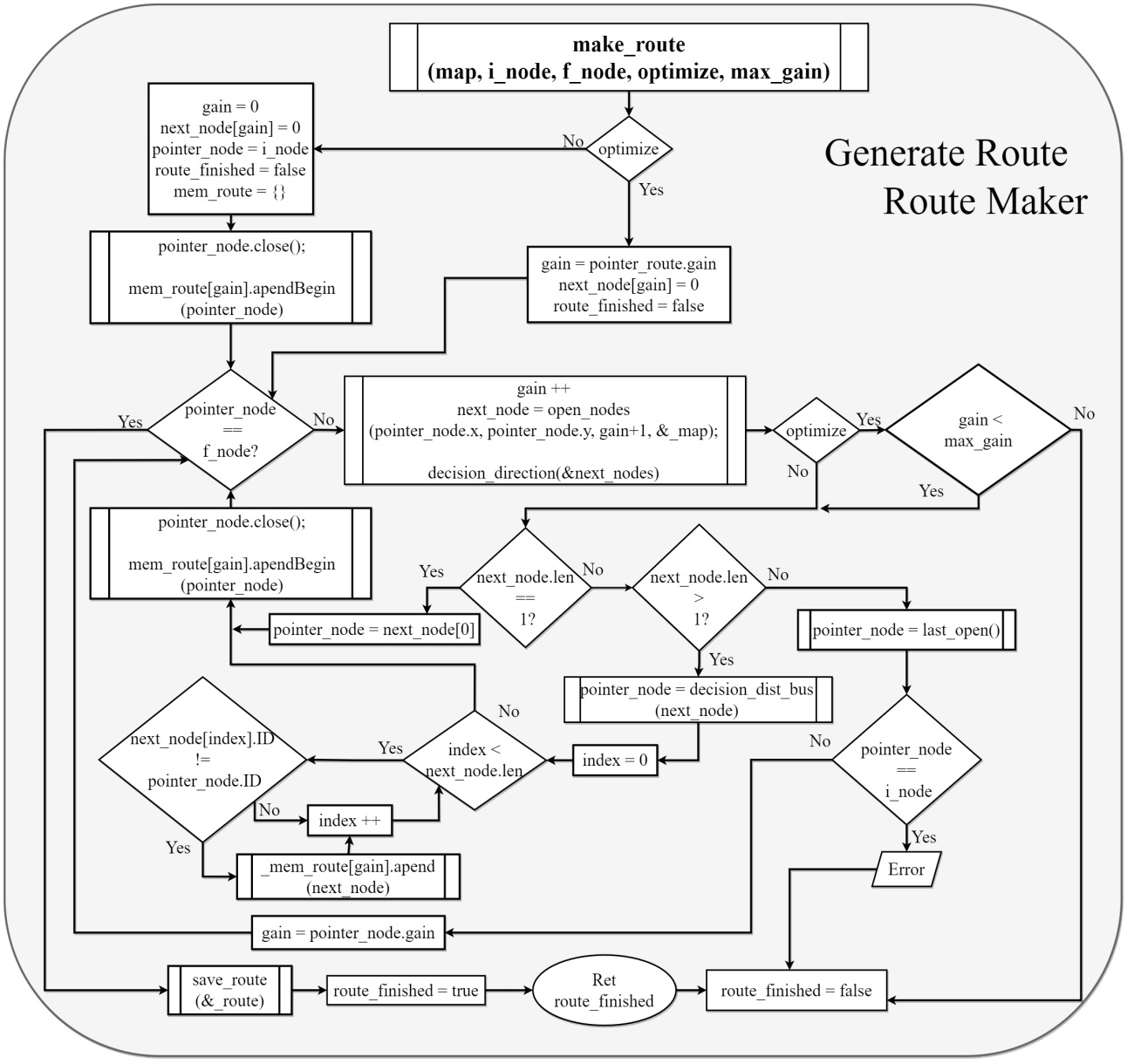

Figure 20: *make_route* Flowchart

Open Node

The following flowchart (figure 21) dedicates to implement the *open_ node* function.

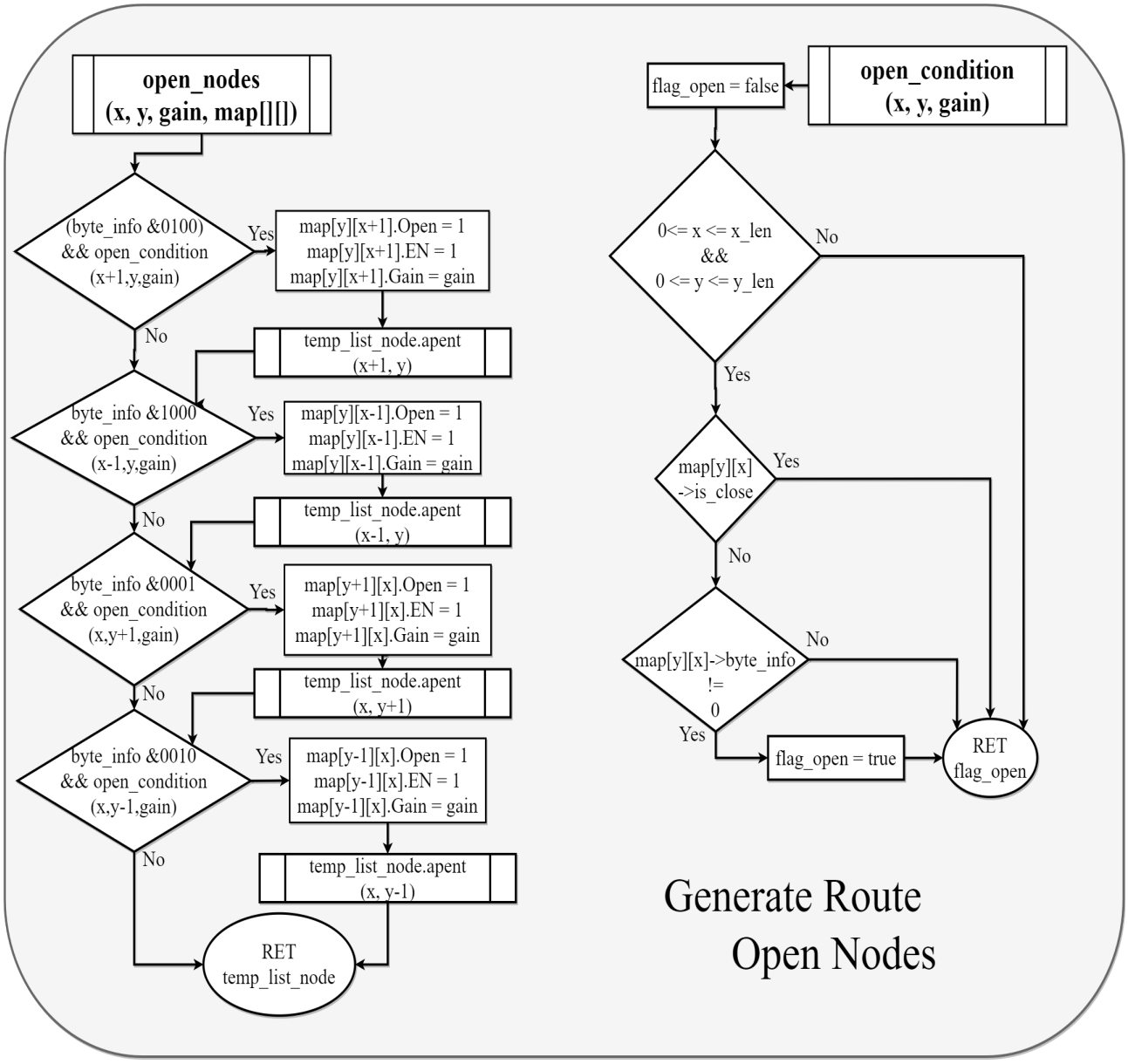

Figure 21: *open_ node* Flowchart

Decision

The following flowcharts (figures 22 and 23) dedicates to implement the two types of decision making. First the decision based on the node directions and the second the decision based on the node distance to the end.

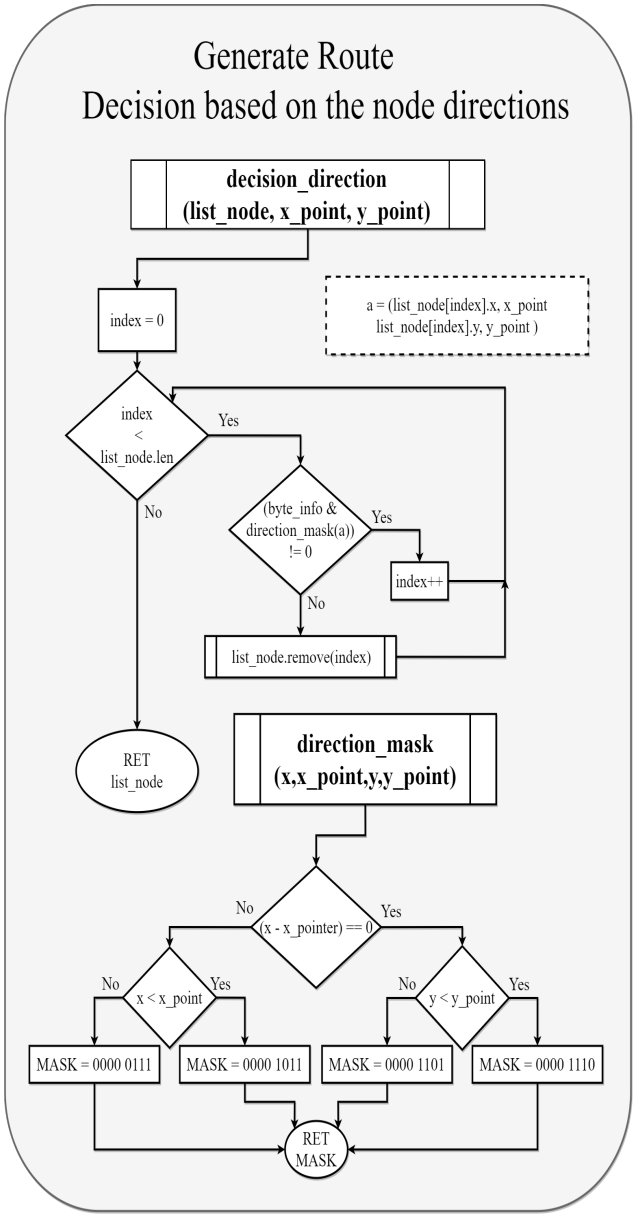

Figure 22: Decision based on node direction Flowchart

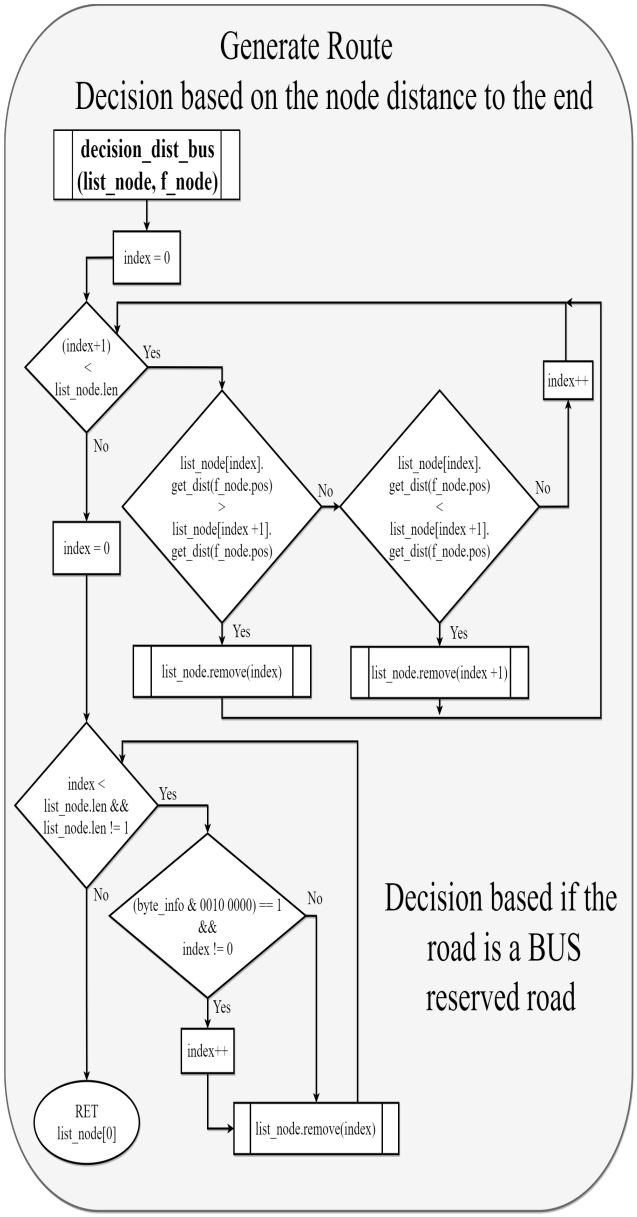

Figure 23: Decision based on distance to the end Flowchart

Last Node Opened

The following flowchart (figure 24) dedicates to implement the *last_ open* function.

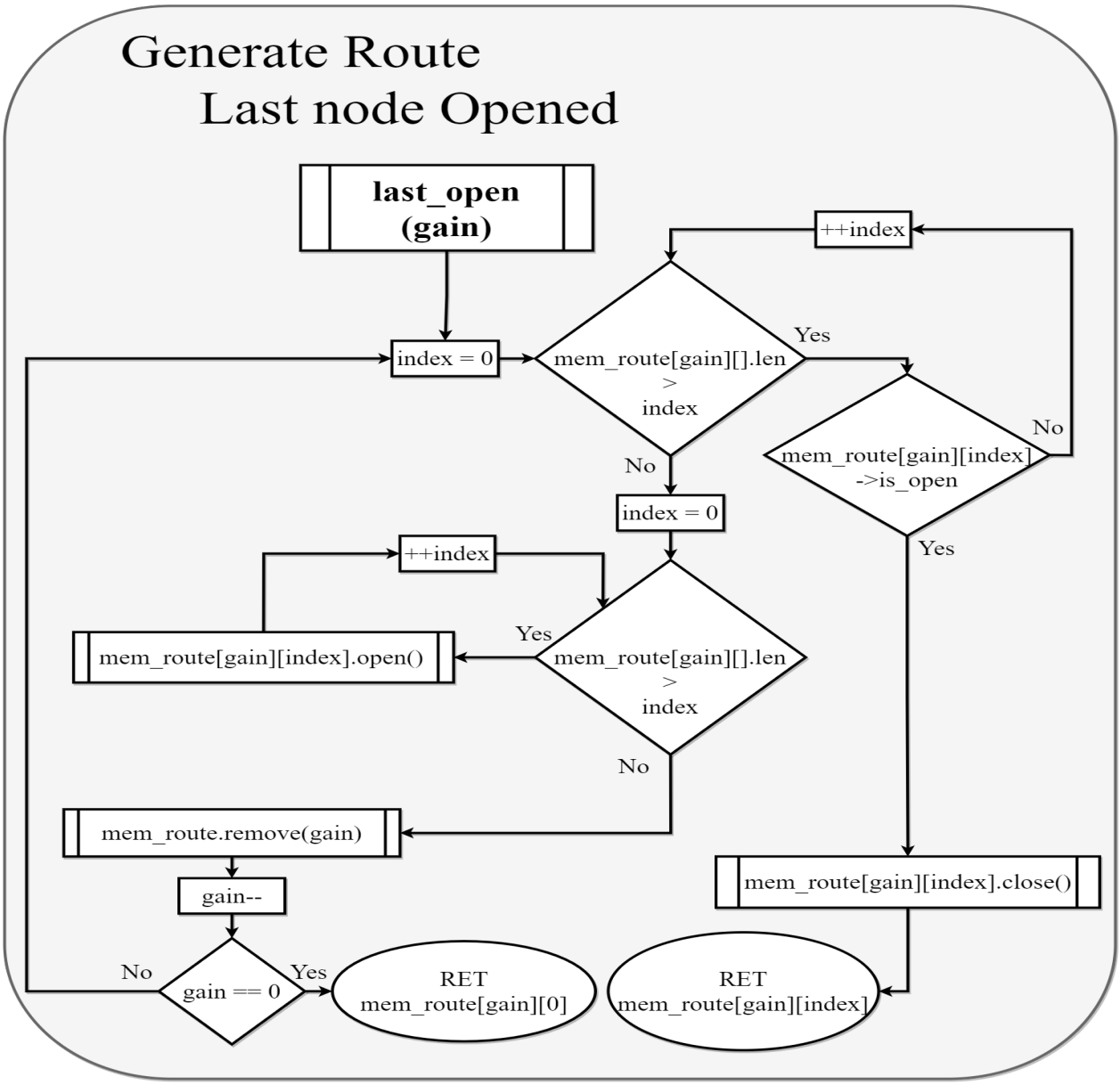

Figure 24: *last_ Open* Flowchart

Save Route

The following flowchart (figure 25) dedicates to implement the *save_ route* function.

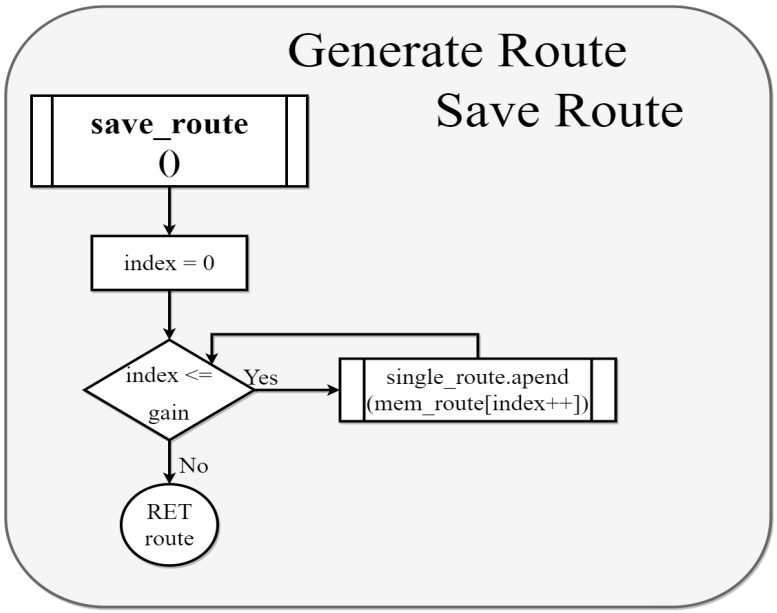

Figure 25: *save_ route* Flowchart

Optimize Route

The following flowchart (figure 26) dedicates to implement the *optimize_ route* function.

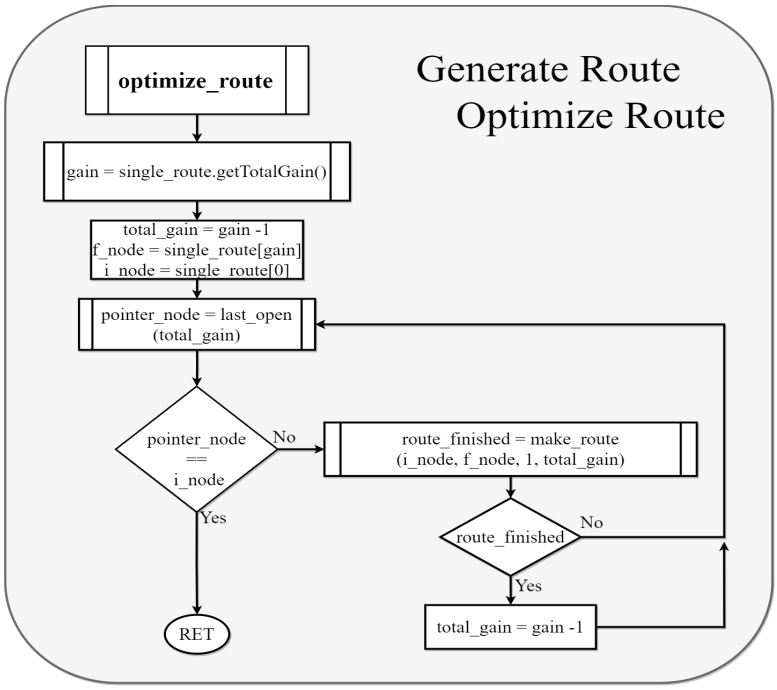

Figure 26: *Optimize_ route* Flowchart

9.1.6 Bus and Bus Stop Communication: Request Bus Stop ID

Whenever a bus arrives at the destination stop, the driver himself does the acknowledge by pressing a button. Then he waits for it to be validated. If not validated, the bus will need to get back on the right track, so the system generates a route to the right destination. Otherwise, if it is validated, the system allows the driver to continue his default route.

The follow algorithm explains the steps and the actions of validating a Bus Stop ID: (figure 27)

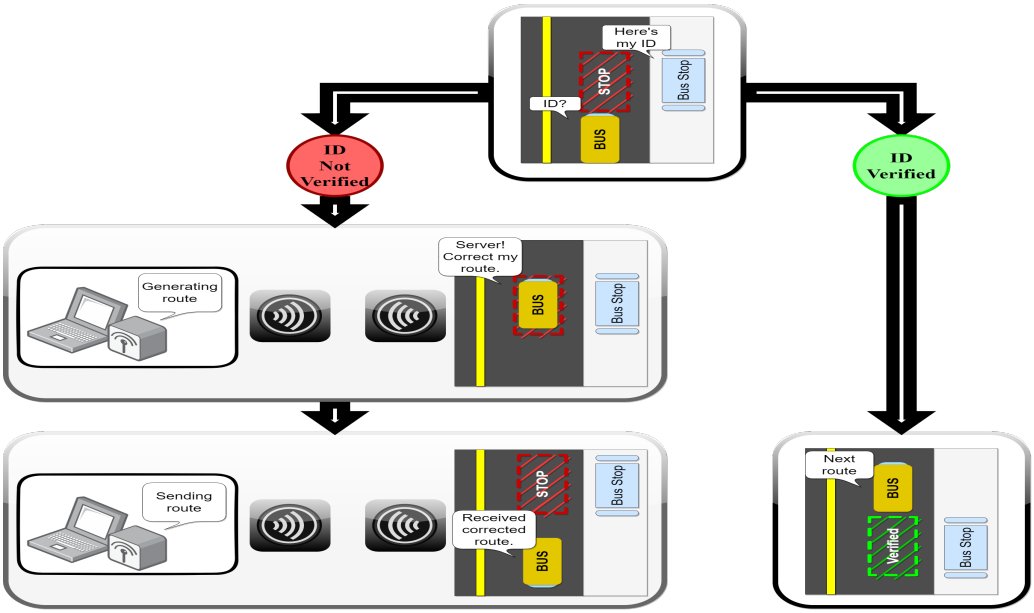

Figure 27: Bus and Bus Stop Communication Algorithm

Algorithm:

1 - When the bus driver is arriving a bus stop, he presses the "ID REQUEST" button.

2 - If the Bus is to far from the bus stop:

- **a -** Transmit an Error saying that the bus is too far from any bus stop, and return to Step 1.

3 - If the Bus is near to a bus stop:

- **a -** Requests the nearest Bus Stop ID

4 - If the ID is the same as the one illustrated in the defined route:

- **a - If the bus system doesn't have a next route:**
  - **i -** Sends an Error informing that there is no route defined.
- **b - Else if the bus system have a next route:**
  - **i -** The Bus system proceeds to the next route.
  - **ii -** Updates the bus arrival time to the next bus stop.
  - **iii -** Informs the second next bus stop that he will arrive soon and send it's arrival time.
- **c - If the ID is NOT the same as the one illustrated in the defined route:**
  - **i -** The bus system asks for directions to the main system.
  - **ii -** The main system generates a route to the correct Bus Stop.
  - **iii - If the route is not generated:**
    - **1 -** The main system sends an Error to the bus system and to itself, saying it could not generate the route.
  - **iv - If the route is generated:**
    - **1 -** The main system sends the route to the bus system.
    - **2 -** The bus system saves it.

9.1.7 Bus and Bus Stop Communication: Flowchart

Request BusStop ID

The following flowchart (figure 28) dedicates to implement the *Request_ BusStop_ ID* function.

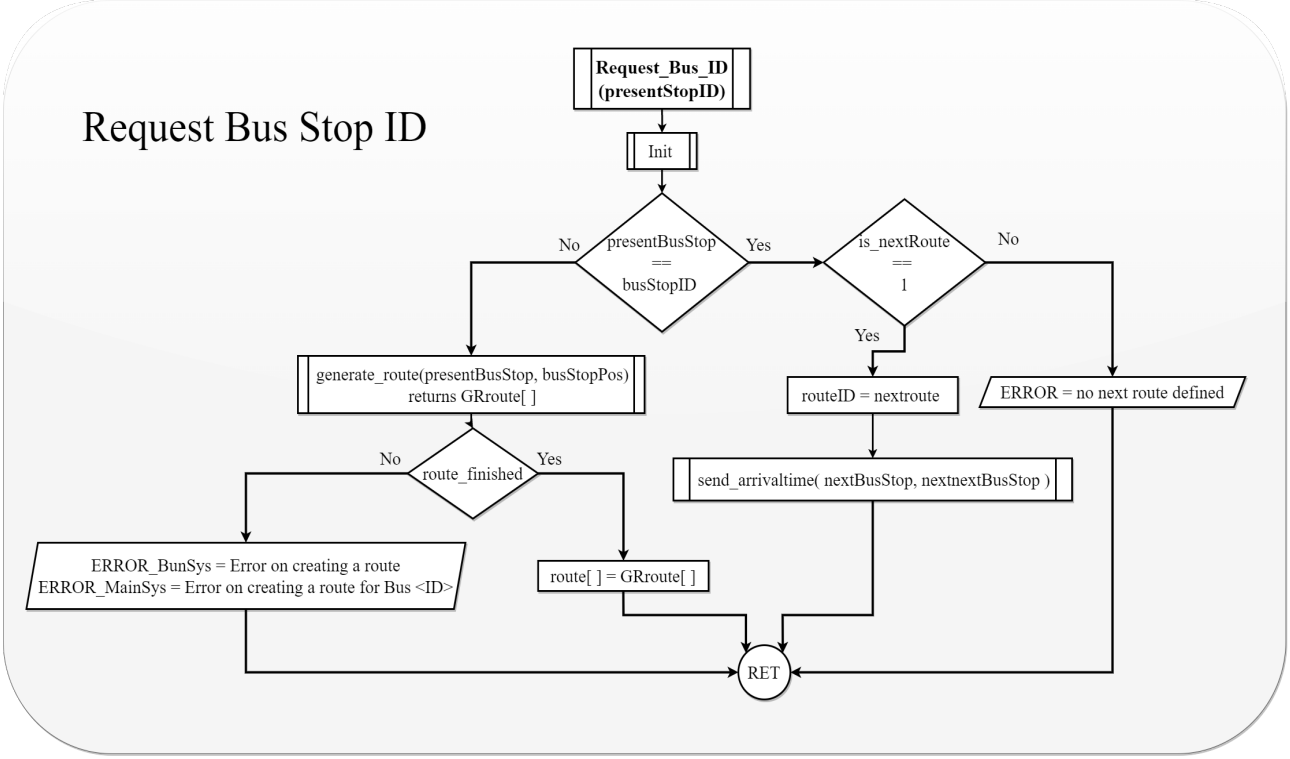

Figure 28: *Request_ BusStop_ ID* Flowchart

9.1.8 Web Application: Process Request

- **The Web BUSIT application consists in three Requests:**

The **Request Bus Schedule** that is when the user wishes to see the schedule of some specific bus.

The **Request Bus Info** that is the information of the bus status in terms of arrival time and his location.

The **Request Optimal pick up/Bus** which is when the user asks to be picked up at a particular bus stop.

To process the orders mentioned above, it is necessary to implement a processing algorithm, as the figure 29 below suggests.

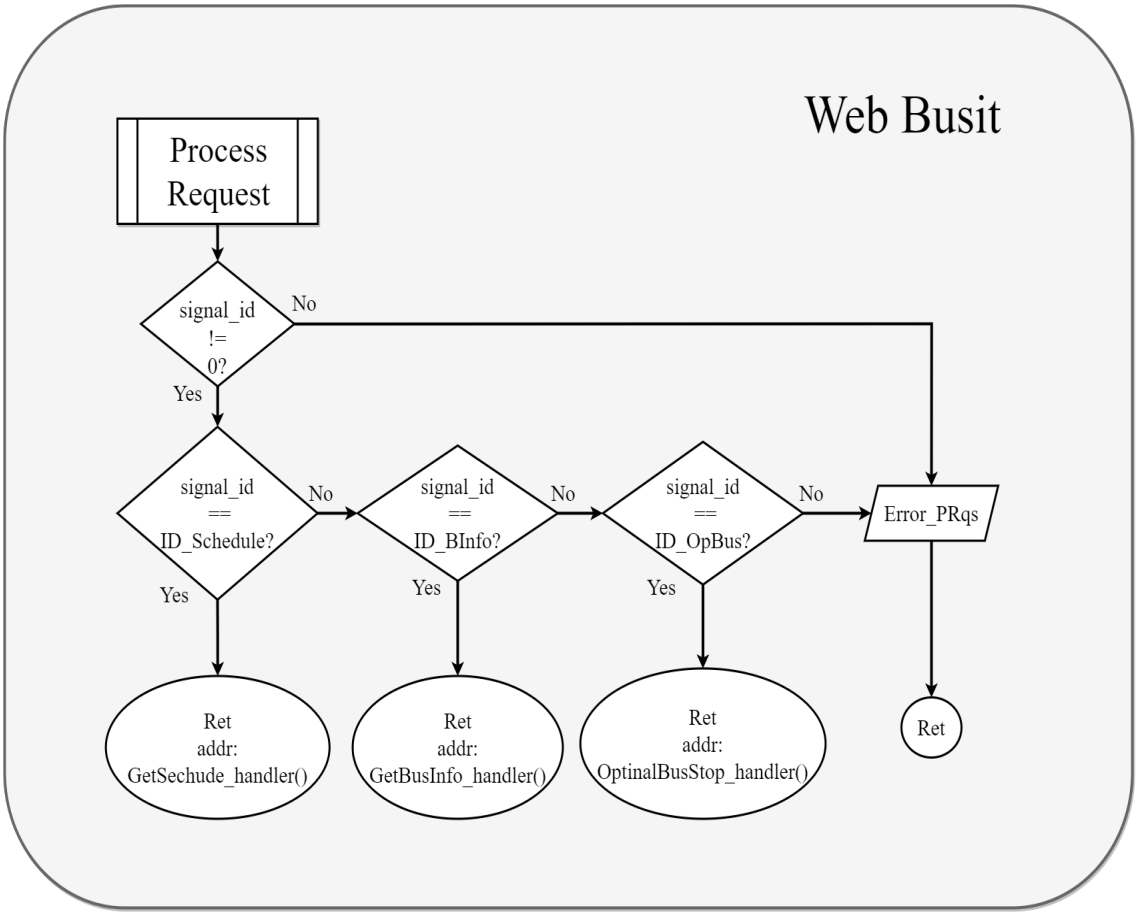

Figure 29: Process Request Flowchart

By using Signals to identify the ID of an request send by the user and Daemon as a background process, the system can identify each request execute it.

9.1.9 Web Application: Optional Bus Stop

When a user Requests a pick up at a particular bus stop the system sends a message to one nearby small bus saying that a pick up request occurred. From there the small bus driver decides whether to accept it or not. If he doesn't accept the system will send the request to another one, but if no other is found within 3 minutes, the request fails and sends the feedback to the user saying that he couldn't find a small bus to pick him up.

The follow algorithm explains the steps when a request for pick up occurs (figure 30):

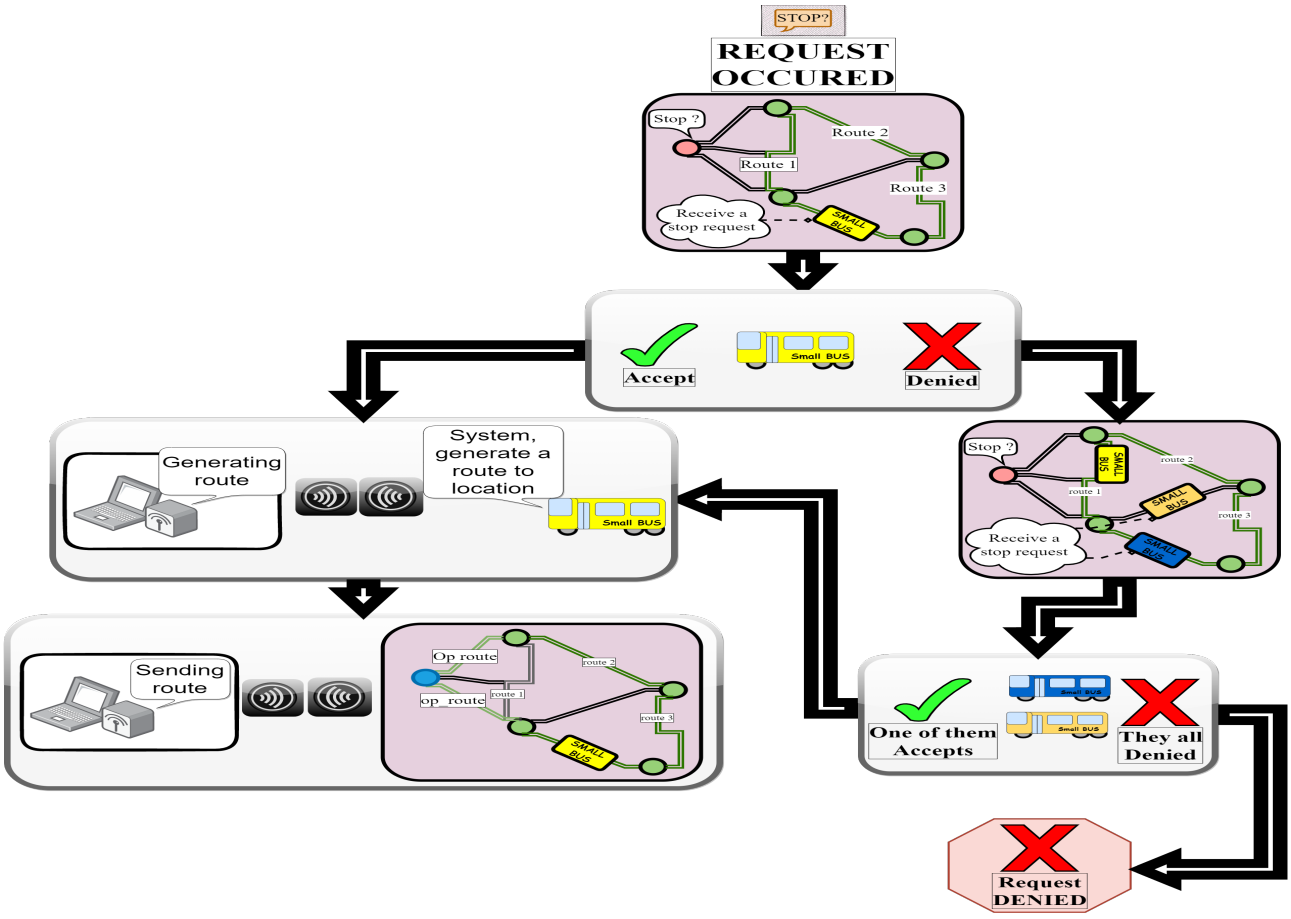

Figure 30: Optional Bus Stop Algorithm

Algorithm:

1 - The main system looks for the small bus available that are or will pass in near Bus Stops.

2 - If no bus answered within 3 min (180s):

a - Report to the application that no small bus is available.

3 - Else if a small bus is found:

a - The main system sends a request to it.

b - If it denies:

i - The main system takes note and returns to step 1.

c - Else if he accepts:

i - The small bus asks for directions to the main system.

ii - The main system generates a route to the optional Bus Stop.

iii - If the route is not generated:

1 - The main system sends an Error to the small bus, saying it could not generate the route.

2 - And returns to step 1.

iv - If the route is generated:

1 - The main system sends the route to the small bus and saves it.

2 - Report to the application that the small bus is available and send his ID and the ARRIVAL TIME.

9.1.10 Web Application: Optional Bus Stop Flowchart

Optional Bus Stop

The following flowchart (figure 31) dedicates to implement the *Optional_ Bus_ Stop* function.

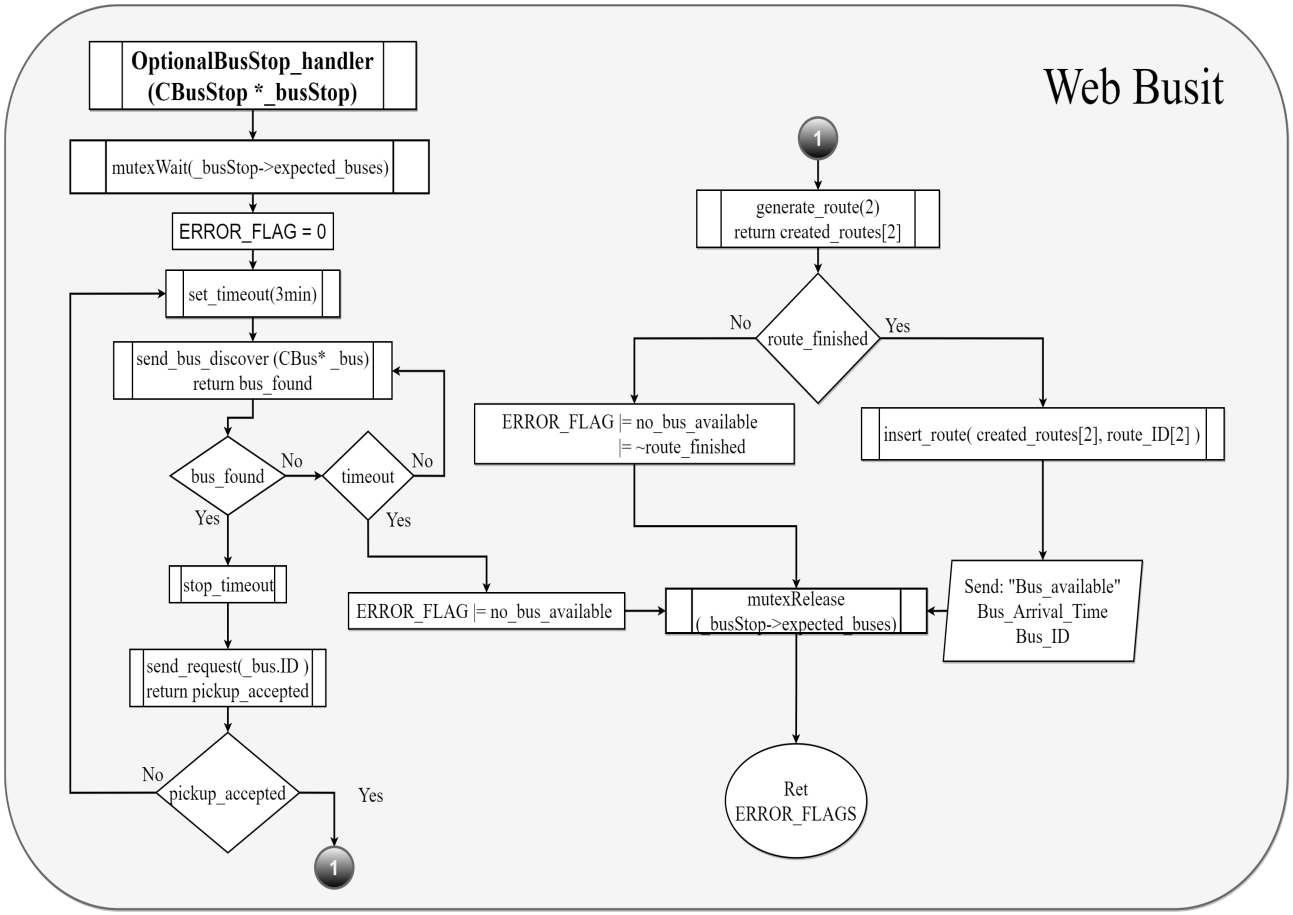

Figure 31: *Optional_ Bus_ Stop* Flowchart

9.2 Software Specifications

In the software specifications, it will be presented the class diagram of BUSIT, along with the threads division and its priority..

9.2.1 Class Diagram

The class diagram is a tool used to visualize object oriented systems. It uses the UML notation and it is a static representation view of BUSIT project. In figure 32 is illustrated the classes, attributes, operations, and relationships between the projects' objects.

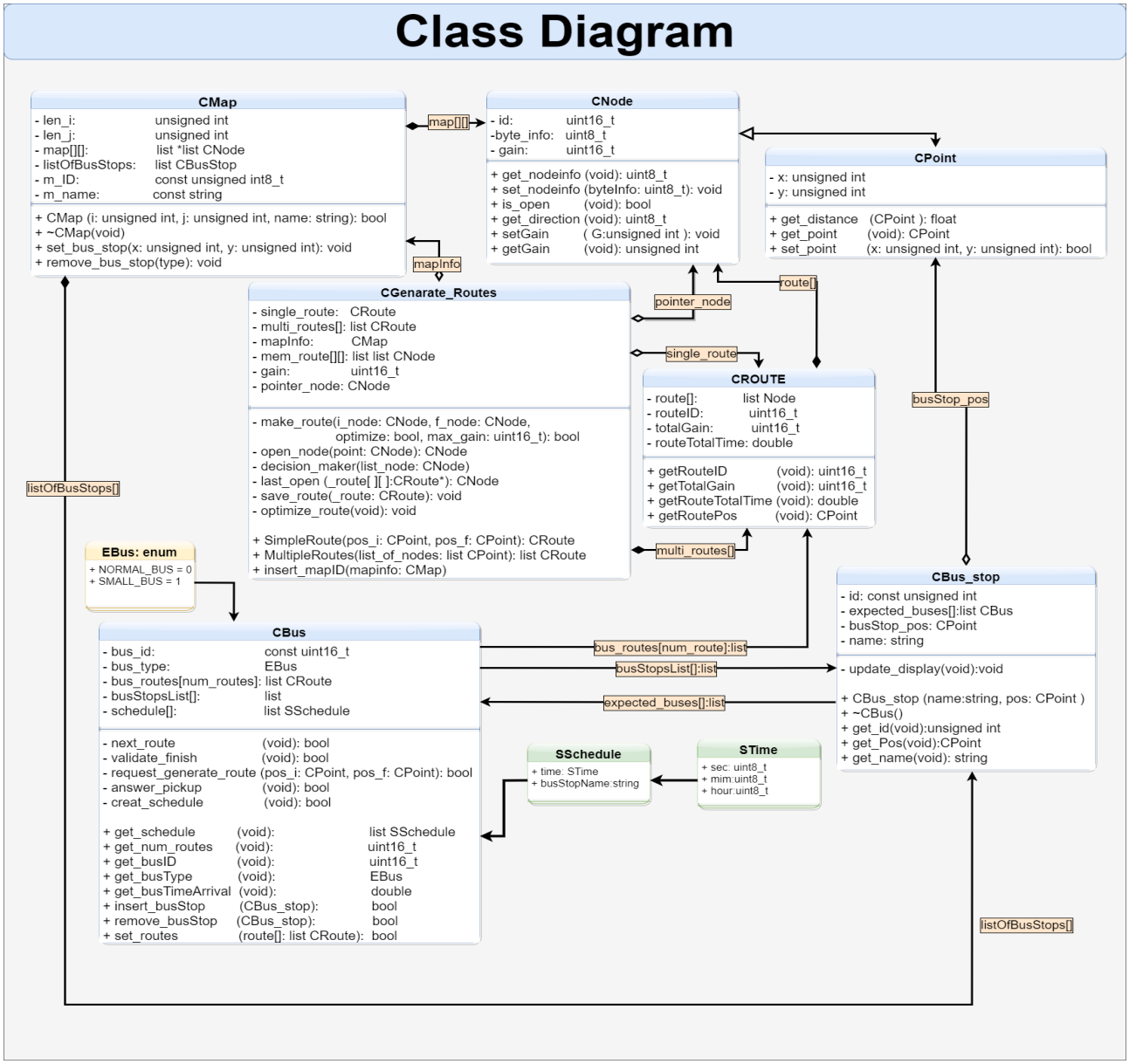

Figure 32: BUSIT Class Diagram

9.2.2 Threads Division

Pthread is an API design for creating and synchronizing threads.
***Busit* requires the following pthreads:**

- **PMain_System -** A thread that is in charge of controlling all the system.
- **PGenerate_Route -** A thread that generates routes.
- **PBus_BusStop_Communication -** A thread that is in charge of maintaining the communication between Bus and Bus Stop.
- **PRequest_Schedule -** A thread that is responsible for handling the schedule request to the main system.
- **PRequest_BusInfo -** A thread that is responsible for handling the bus info request to the main system.
- **PRequest_Optional_BusStop -** A thread that is responsible for handling the optional bus stop request to the main system.
- **PError -** A thread that is responsible to handle all possible errors that can occurs in the other threads.

9.2.3 Thread Priority

All the thread must be assign a priority since it is a multitasking system. The figure 33 shows the priority for the threads, from the lowest to the highest.

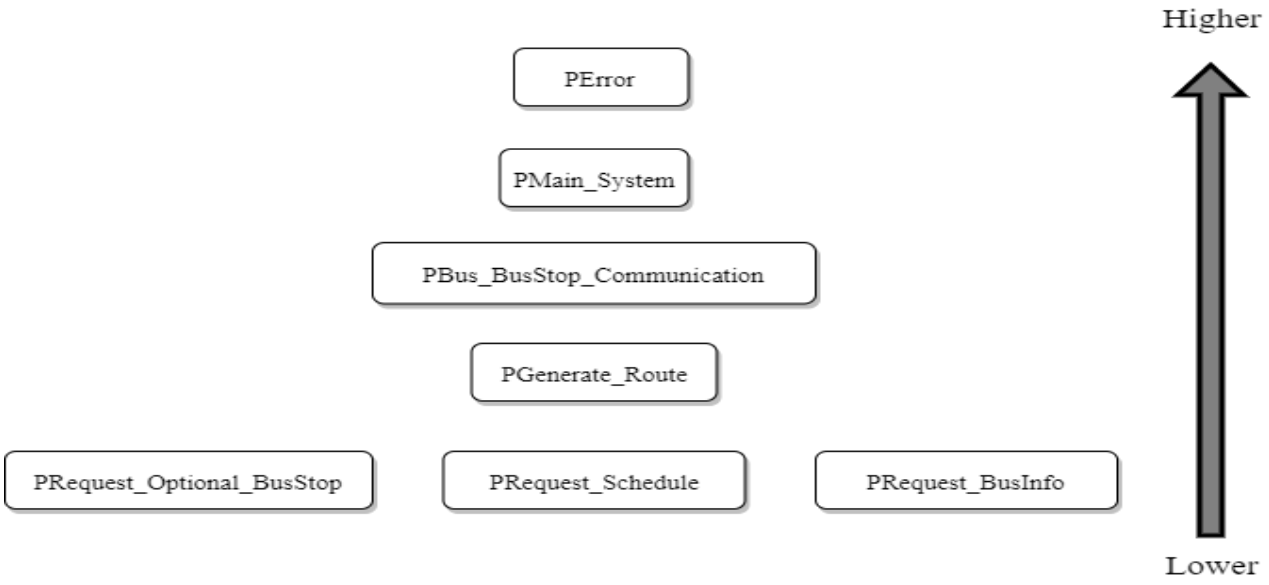

Figure 33: Thread Priority

9.3 Hardware Specifications

In the hardware specifications, it will be shown all the hardware BUSIT will need, along with the peripherals, pinout and connections between different components.

9.3.1 Prototype Design

To demonstrate the hardware functionalities, a prototype was created to simulate the communication between the various subsystems (figure 34), *Bus Stop*, *Bus* and *Main System*:

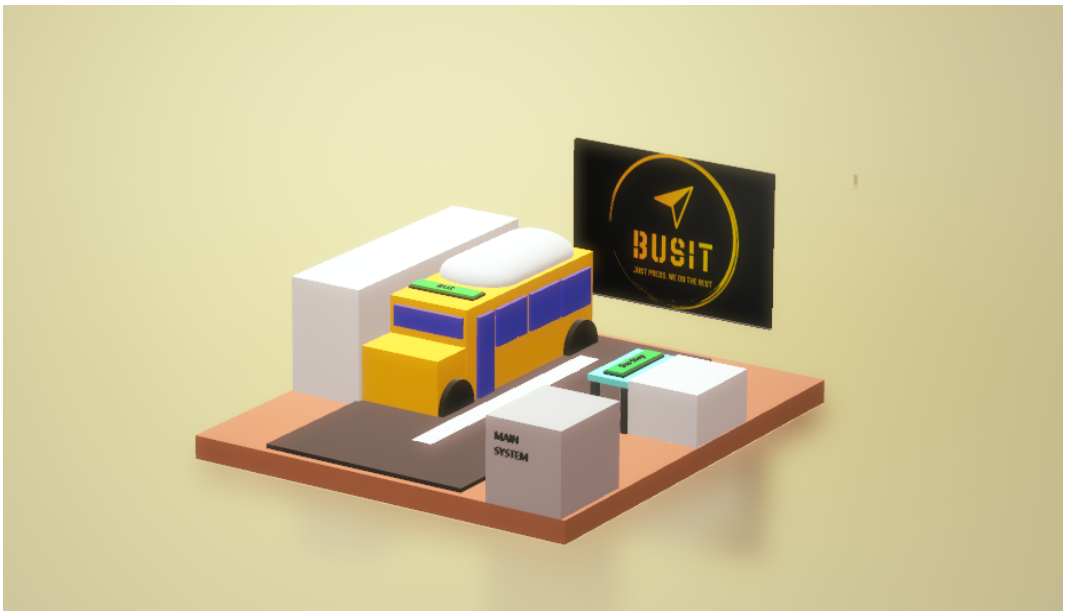

Figure 34: Prototype Scenario

The prototype must illustrate the following scenario cases:

- **Communications between the Bus Stop and the Bus:**
  - Confirmation of the Bus Stop ID accepted;
  - Confirmation of the Bus Stop ID denied;
  - Time arrival of the bus;

- **Communications between the main system and the Bus:**
  - Send a optional pick up request, and answer it;
  - Set or request a route for the bus;

- **Communications between the main system and the Bus Stop:**
  - Get or Set a Bus Stop ID;

9.3.2 Prototype Circuit

In order to make the prototype schematic showed in figure 34, it was analysed the hardware components that were necessary. The list of components are listed below and the prototype circuit can be seen in the figure 35:

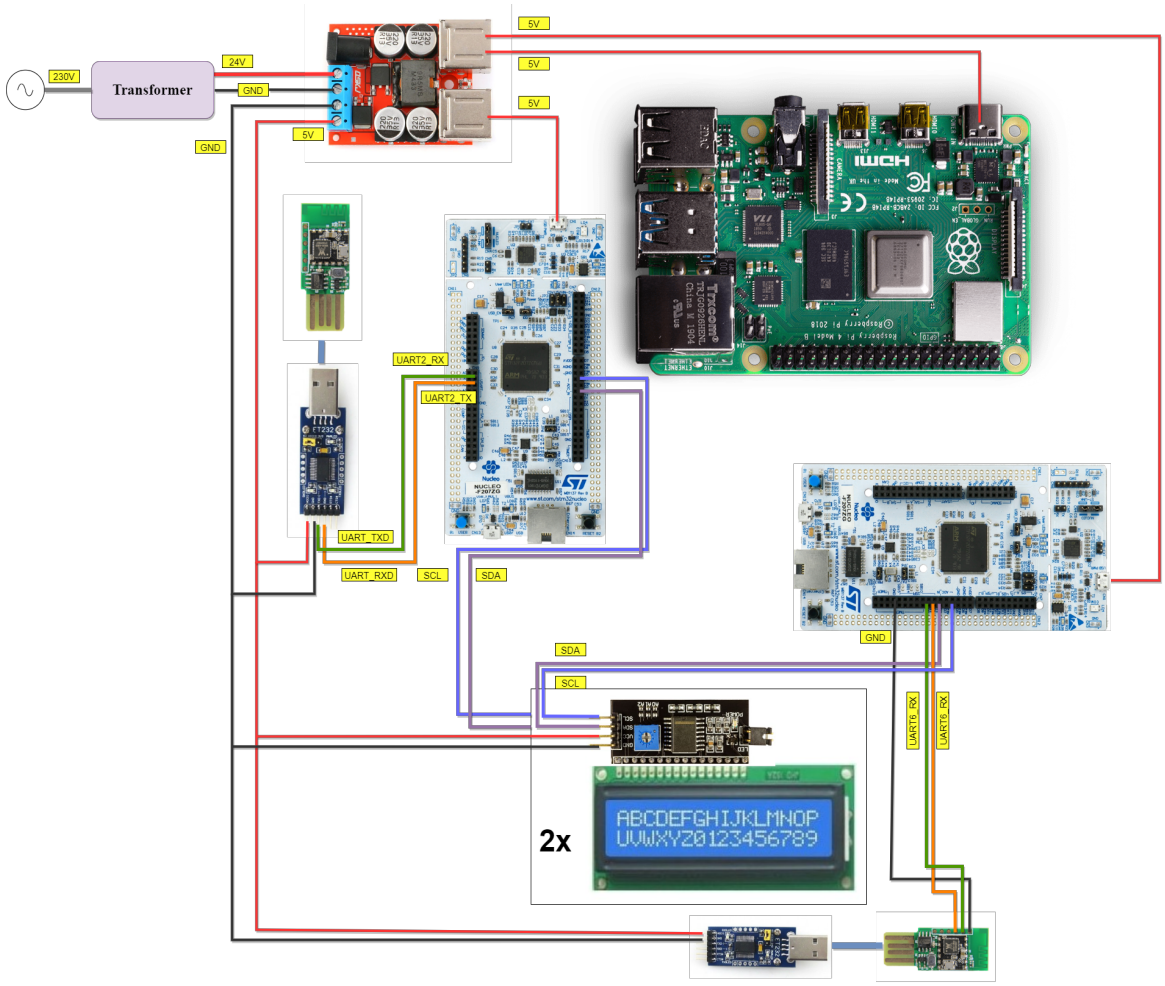

Figure 35: Prototype Circuit

Components To Be Used:

- **Raspberry Pi** As the *main system.*
- **Two Microcontroler STMs:** One for the *Bus Stop system* and another for the *Bus system.*
- **Two WiFi Development Boards:** One for the *Bus Stop* and another for the *Bus.*
- **Two LCD Displays With an I2C Driver Module** One for the *Bus Stop interface* and another for the *Bus interface.*
- **Step-Down Module** to act as the system power supply.

9.3.3 Raspberry Pi 4 - Model B

The development board that BUSIT will use is the Raspberry Pi 4 - Model B (figure 36), as mentioned in the constrains.

The Raspberry is where the most complex processes will be done and it is also important to understand how the pinout and peripherals are implemented.

Some of the features that this component possess that is particularly useful for BUSIT are:

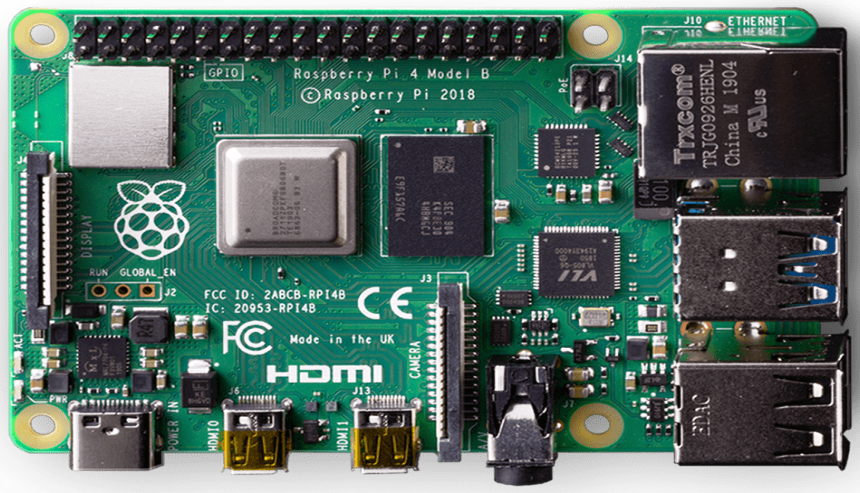

Figure 36: Raspberry Pi 4 - Model B

- **SD Card:** The Raspberry Pi 4 - Model B has a dedicated SD card socket.
- **Power Supply:** This device requires a good quality USB-C power supply capable of delivering 5V at 3A.
- **Processor:** The Raspberry has a Quad core 64-bit ARM-Cortex A72 processor running at 1.5GHz.
- **Memory:** Besides the SD card slot, the device has 1, 2 and 4 Gigabyte LPDDR4 RAM options.
- **Connectivity:** The Raspberry has the following connectivity features:
  - **Ethernet:** One Gigabit Ethernet port.
  - **Bluetooth:** Bluetooth 5.0 with Bluetooth Low Energy (BLE).
  - **USB:** Two USB2 ports and two USB3 ports.
  - **Wireless:** 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless.
- **GPIO:** The Raspberry has a standard 40 pin GPIO header and its specification are presented on figure 37.

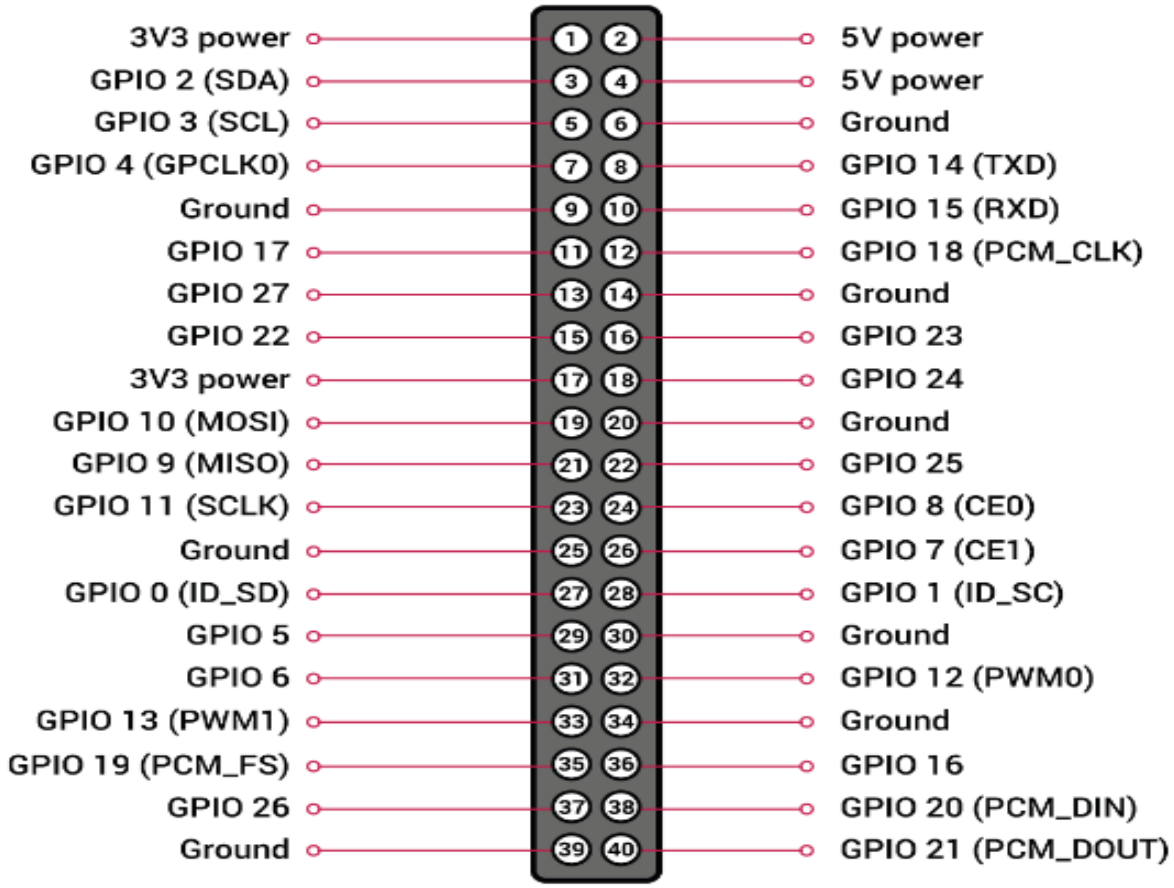

Figure 37: GPIO of Raspberry Pi 4 - Model B [10]

9.3.4 STM32 Nucleo-F767ZI

The microcontroller STM32 (figure 38) will be used for simulating the client bus presented in the hardware architecture and it will communicate to other subsystems by a WiFi module.

Some of the features that this board has that will be useful for BUSIT are:

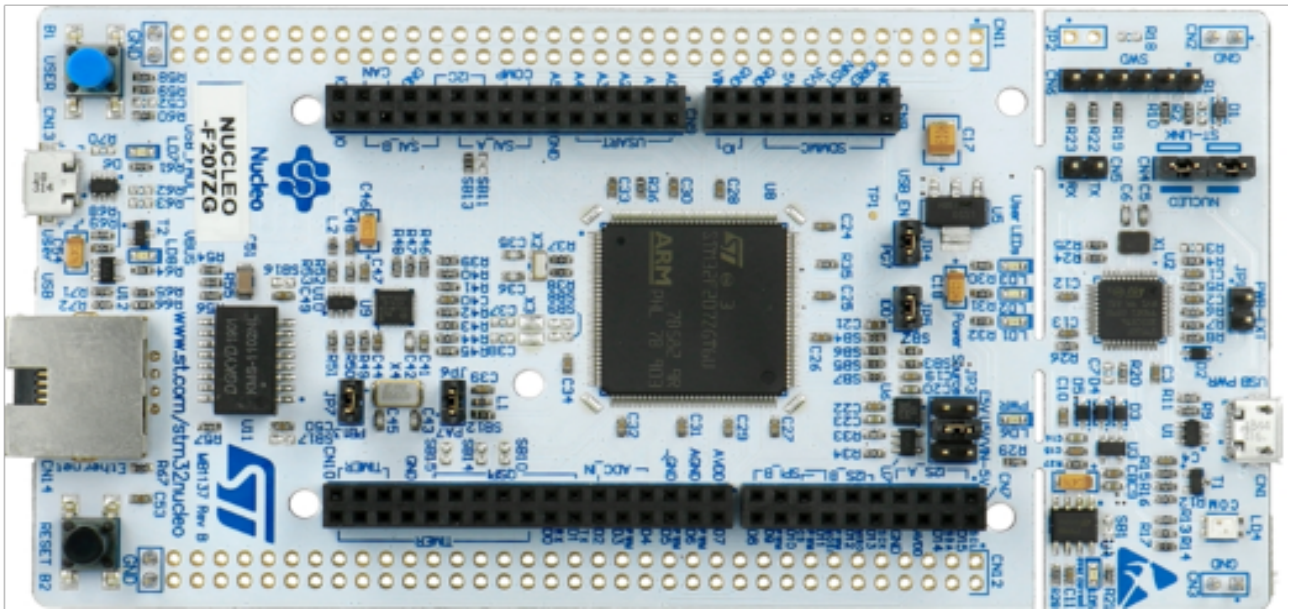

Figure 38: STM32 Nucleo-F767ZI

- **Microcontroller features:**
  - ARM®32-bit Cortex®-M7;
  - 216 MHz max CPU frequency;
  - 2 MB Flash;
  - 512 KB SRAM;
  - USART/UART (8);
  - GPIOs (114) with external interrupt capability.

- **Nucleo board features:**
  - STMicroelectronics Morpho extension pin headers for full access to all STM32 Inputs/Outputs;
  - On-board ST-LINK/V2-1 debugger/programmer with SWD connector;
  - Three User LEDs;

The pinout specification of STM32 Nucleo-F767ZI are presented on figures 39, 40, 41 and 42.

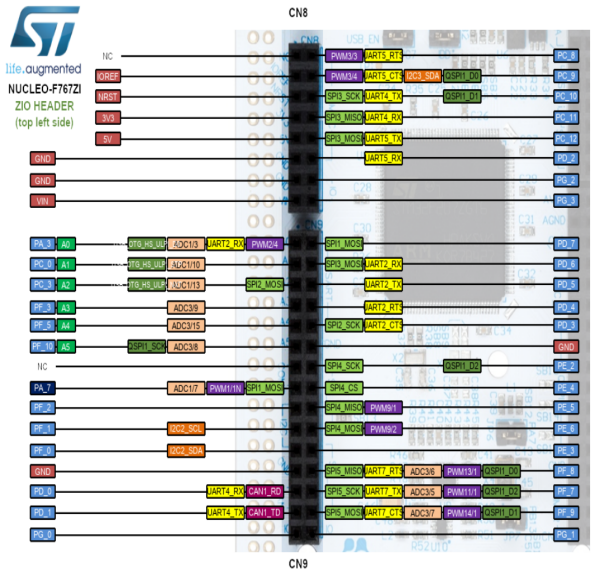

Figure 39: STM32 Nucleo-F767ZI - pinout 1 [8]

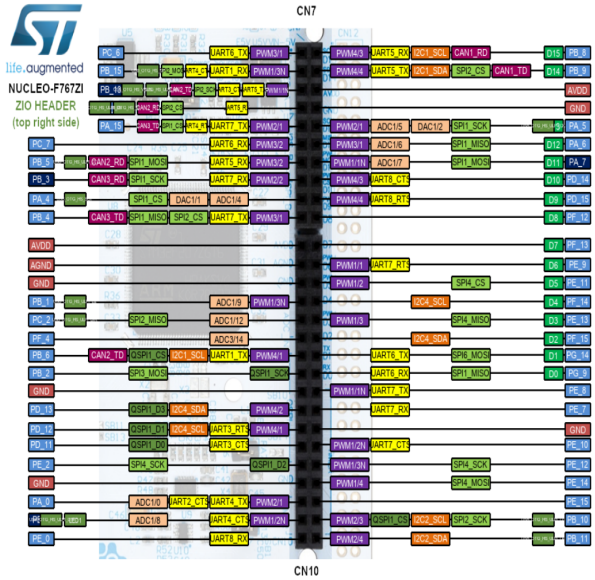

Figure 40: STM32 Nucleo-F767ZI - pinout 2 [8]

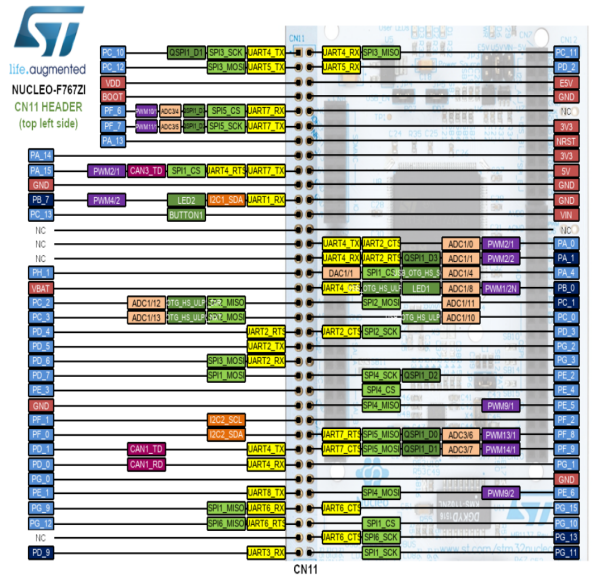

Figure 41: STM32 Nucleo-F767ZI - pinout 3 [8]

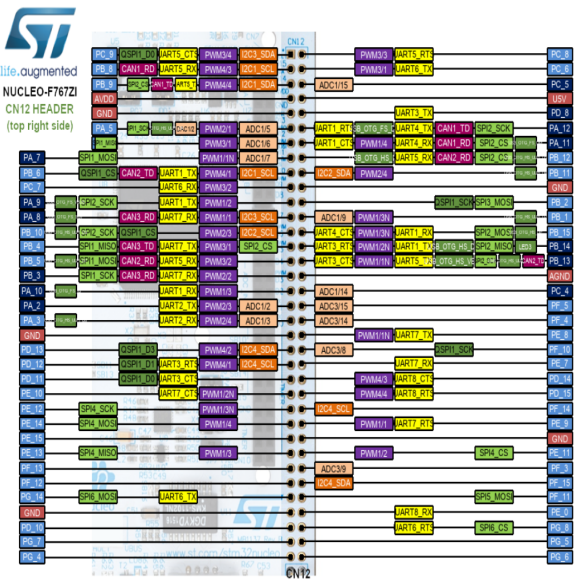

Figure 42: STM32 Nucleo-F767ZI - pinout 4 [8]

9.3.5 Step-Down 5V

For the project, it will be used a special step-down to 5V that is able to power 4 USB devices, as shown in figures 44 and 43. The two main characteristics of this driver are listed below. ´

Figure 43: Step-Down - left side view

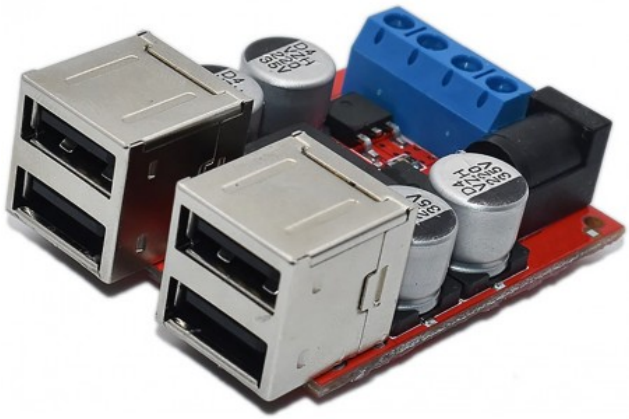

Figure 44: Step-Down - right side view

- 4 independent charging management integrated circuit.
  - The 4 USB is supported by 2A current charging, the total output current can reach 8A.
- 8V to 35V input voltage, with a stable output of 5V.

The pinout of this component are illustrated in the figure 45. ´

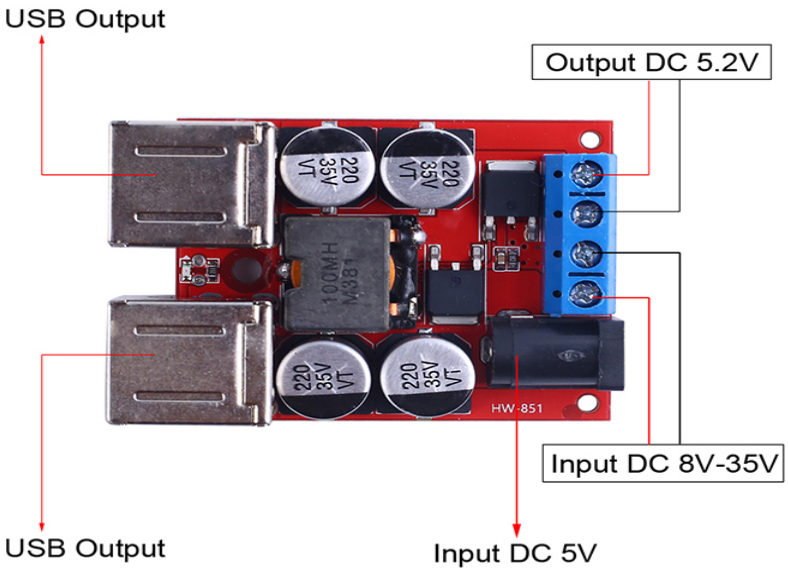

Figure 45: Step-Down pinout

9.3.6 LCD Display with driver

In order to inform the different subsystems of the system or another subsystem status, it will be used LCD Displays. The one that BUSIT will use will be a 2x16 with a driver to make the I2C interface (figure 46) and its characteristics are listed below.

- LCD display module with blue and yellow Green backlight.
- LCM type: Characters
  - Can display 2-lines of 16-characters each.
- I2C chip: PCF8574.

Figure 46: LCD Display 2x16

The pinout of the LCD component are illustrated in the figure 47.

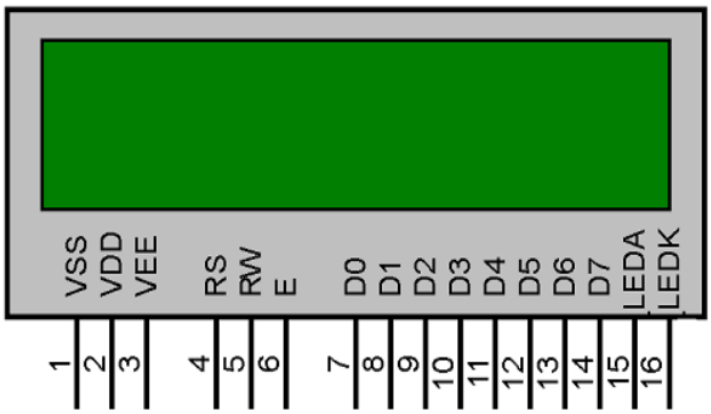

Figure 47: LCD Display 2x16 pinout

9.3.7 WiFi Development Board

To implement the WiFi communication in the various subsystems that BUSIT will have, it will be used the WiFi Development Board Air602 (figure 48). This development board specifications are listed below.

- USB interface;
- Integrated GPIO device controller;
- UART/SPI;

- Supports WiFi Direct;
- Supports WiFi WMM/ WMM-PS/ WPA/ WPA2/ WPS;
- Supports IEEE802.11 b/ g/ e/ i/ d/ k/ r/ s/ w/ n;
- Support multiple network protocols:
  - TCP;
  - UDP;
  - ICMP;
  - DHCP;
  - DNS;
  - HTTP;

Figure 48: WiFi Development Board

The pinout of this component are illustrated in the figure 49.

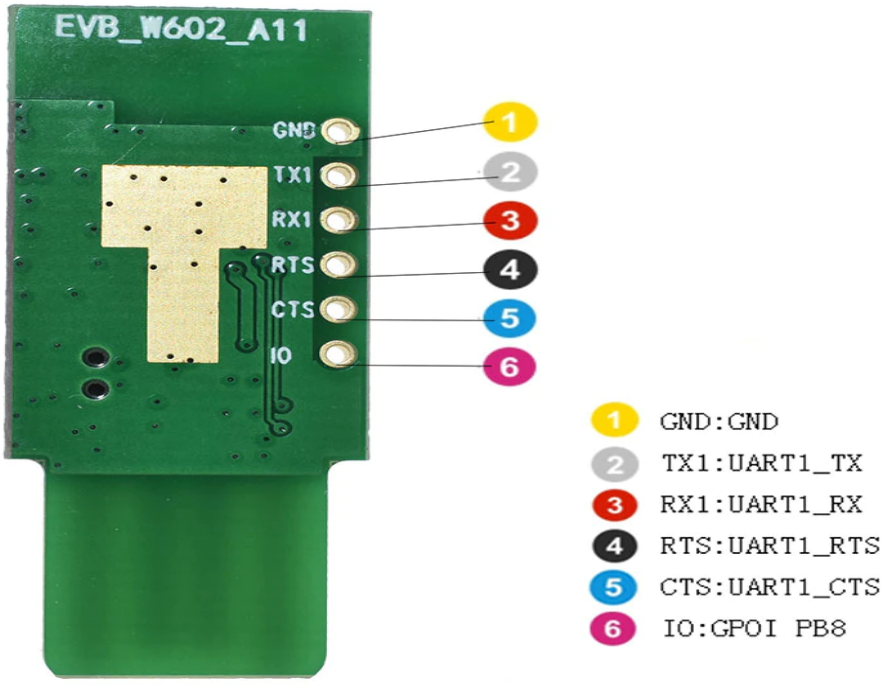

Figure 49: WiFi Development Board pinout

9.3.8 FTDI USB to TTL Serial Converter Adapter Module

Since the WiFi Development Board requires a USB connection, it will be used a FTDI adapter (figure 50) in order to implement it correctly.

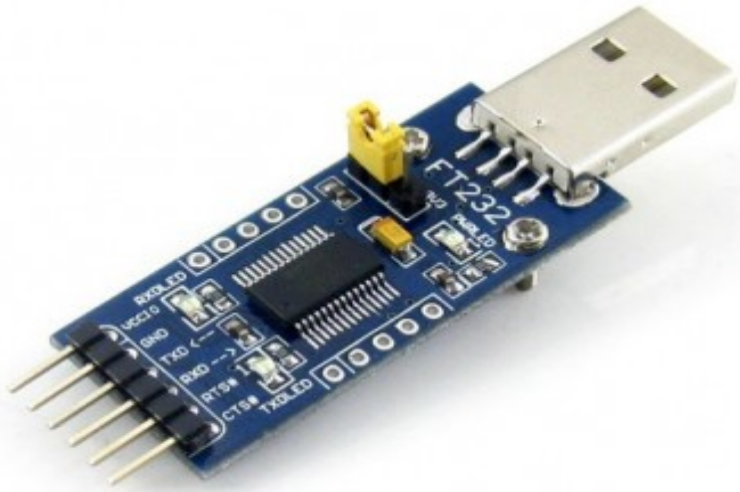

Figure 50: FTDI adapter

9.4 Test Plan

Since BUSIT will be a mainly software project, the Test Plan must be done carefully and with full detail in order to grant the correct integration of all the different algorithms and to make sure that the Interprocess Communication (IPC) is working as supposed. The Test Plan can even identify critical sections that were not taken into account and its implementation avoids launching the project with errors that could go unnoticed.

Master Test Plan

In this section it will be presented all the tests that must be done, from the bottom to the top in terms of complexity.

In the hardware tests, the components that requires tests are:

- **Step-Down:** It must be tested to the correct power supply to all modules.

- **WiFi Development Board:** This module is extremely important to the subsystems communication and it should be correctly tested to understand how to rightly configure the component as a transmitter and as a receiver.

- **LCD Display:** This display should print correctly the message whenever the main system sends a specific signal and the integration with the previous board also needs to be tested.

- **STM32 Nucleo-F767ZI:** The STM needs to be tested to simulate both the bus client and the bus stop client and also be integrated with the display.

- **POSIX - Message Queue:** Message queues is a tool often used for thread communication and the test that is going to be implemented will dictate if this IPC is recommended for BUSIT.

- **POSIX - Shared Memory:** Sometimes, instead of using message queues, one can prefer using shared memory and, since it is the fastest IPC, the performance increase may be a need of BUSIT and, in the tests, the concepts of shared memory must be deeply understood because, despite being the fastest, it is also the most dangerous Interprocess Communication.

In terms of functionalities and libraries that must be understood deeply, there are:

- **Daemons:** It must be tested in order to implement the background program for the application communication.

- **Portable Operating System Interface (POSIX) - Pthreads:** For achieving multitasking in the BUSIT project, the use of Pthreads library in the POSIX API is a necessity. For that, a test for threads needs to be done to assign each thread the right priority.

- **POSIX - Signals:** To use signals in Linux for the BUSIT needs, a study and test must be implemented to use the software timer correctly.

- **POSIX - Semaphores:** IPCs will have an important role in the project, since BUSIT will have critical sections and different threads, they will control the threads communication and handle any race conditions that might happen. A semaphore is one example of IPC and needs to be tested to ensure that those race conditions will be handled correctly.

- **POSIX - Message Queue:** Message queues is a tool often used for thread communication and the test that is going to be implemented will dictate if this IPC is recommended for BUSIT.

- **POSIX - Shared Memory:** Sometimes, instead of using message queues, one can prefer using shared memory and, since it is the fastest IPC, the performance increase may be a need of BUSIT and, in the tests, the concepts of shared memory must be deeply

understood because, despite being the fastest, it is also the most dangerous Interprocess Communication.

About BUSIT complex tests, it was divided according to the algorithms presented previously:

- **Generate Route:** This algorithm is surely the most complex of them all, so its test must be done with special attention.
- **Bus and Bus Stop Communication:** This algorithm will dictate how the system can efficiently communicate between different critical subsystems.
- **Application:** The application, despite not being as critical as the previous two, is equally important to the project since its the direct way to the user to communicate with the others subsystems.

For the device drivers, it should be tested the following ones:

- **WiFi:** The WiFi device driver needs to be tested carefully since it will be the main communication between subsystems in BUSIT.
- **LEDs:** The LED device driver, despite being simpler than the previous one, is equally important to ensure its correct operation.
- **Display:** For the LCD Displays device driver, it is important to grant the communication with the others device drivers as well as the application.
- **Communication:** The communication driver that will be used is by UART, therefore, it needs to be tested even before implementing other more complex drivers that also use UART.

Test Phase Plan

For each test listed in the Master Test, a strategy will be presented following the previously defined order to achieve the expected results.

- **Step-Down:** This test needs to be done before supplying other subsystems and components and after integrating each new component. This test, besides being simple, must always be kept in mind in order to grant the correct voltage and current input to each component.
- **WiFi Development Board:** This module will first be tested directly in a desktop machine to understand and configure the module. Later, it will be integrated with the STM to both receive information to send to another module and to receive send the information to the display.

- **LCD Display:** The display will be tested firstly directly with the STM and later will be integrated with the WiFi board.
- **STM32 Nucleo-F767ZI:** The STM will be configured to communicate with the WiFi board by UART and also to communicate with the display by I2C protocol.
- **Daemons:** To test the Daemons, a test using Linux OS will be done, using also the signals as inputs to simulate an application call.
- **Portable Operating System Interface (POSIX) - Pthreads:** For the Pthreads, it will be need to test functions that the library has, for example the joins, exits, and attributes.
- **POSIX - Signals:** For the signals it will be analysed the signals that Linux OS can handle, as well as how to utilize with the Daemons.
- **POSIX - Semaphores:** The Semaphores, because it is an independent library, its test will be done separately to understand its tools and later it will be integrated to the main threads.
- **POSIX - Message Queue:** As an IPC, the message queues will be tested by communicating information between different threads.
- **POSIX - Shared Memory:** As a dangerous, but effective IPC, the shared memory will be tested for the same purpose as the message queues and its results will be compared to see if using this IPC instead of the previous one justifies in BUSIT project.
- **Generate Route:** To test this algorithm, it will be necessary to split its functionalities into smaller ones and later integrate them all.
- **Bus and Bus Stop Communication:** To test this algorithm, it will first be necessary to have all the communications drivers working perfectly and later, the integration of those different drivers can act as an algorithm test.
- **Web Application:** For the Web application, like the previous one, will include different device drivers to test, so the same approach will be implemented.
- **WiFi:** To test the WiFi, it will be carefully implemented the driver using the most adequate protocol and later communicate with all the subsystems that requires this device driver.
- **LEDs:** To test the LEDs it will be implemented the corresponding device driver and later activate the different status LEDs by sending different inputs signals.
- **Display:** For the LCD Display, like the LEDs device driver, it will be implemented to later test with input signals and write the right message according to it.

- **Communication:** For this device driver, the UART will be implemented to assure that the core communication works perfectly, since other modules, for example the WiFi and the Display device driver, uses UART for its communication. To test it, different scenarios will be simulated to write and read from subsystems.

9.5 Test Cases Table

Hardware Test

The test table that relates the listed complex algorithms that needs to be tested and the expected result is shown in the table 2.

Table 2: Hardware Test

Test	Module	Expected Output	Real Output
Send information to display	STM	Information sent	
Send information to WiFi Development Board	STM	Information sent	
Send information to STM	WiFi Development Board	Information received	
Get information from WiFi Development Board	WiFi Development Board	Information received	
Receive information from STM	Display	Show correct information	

Device Drivers Test

The test table that relates the listed device drivers that needs to be tested and the expected result is shown in the table 3.

Table 3: Device Driver Test

Test	Module	Expected Output	Real Output
Send a message to main system	Communication	Message received	
Receive and save a message from main system	Communication	Message sent	
Write message on Display	Display	Message wrote	
Activate status LED	LEDs	Status LED changed to right value	

Complex Algorithms Test

The test table that relates the listed complex algorithms that needs to be tested and the expected result is shown in the table 4.

Table 4: Complex Algorithm Test

Test	Module	Expected Output	Real Output
Request Optional Bus	Application	Pickup Accepted	
Request Bus Schedule	Application	Schedule received	
Request Bus Info	Application	Bus Info received	
Process Request	Application	Activate Interruption	
Simple Route	Generate Route	Route created	
Multiple Route	Generate Route	Multiple routes created	
Optimize Bus Route	Generate Route	Route optimized	
Save Route	Generate Route	Route saved	
Request Bus Stop ID	Bus and Bus Stop Communication	ID received	

9.6 Dry Runs

9.6.1 Trace Table: Simple Route Scenario

The BUSIT route creation is one of the most crucial processes of all, therefore it is important that its operation is as robust as possible. Because of that necessity, trace tables were used to certify that the flowcharts were working correctly and without bugs.

The following image below (figure 51) illustrates the scenario picked for this dry run test:

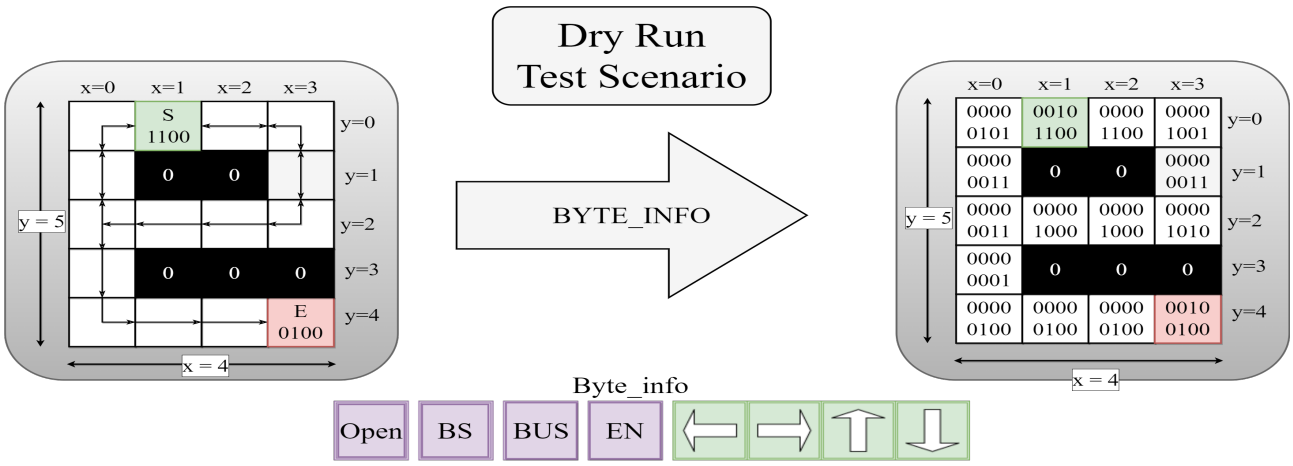

Figure 51: Scenario picked for the dry run test

Defining the start as node(0; 1) and the finish as node(4; 3) the *simple_route* task, must generate a route between them and them optimize it.

The "*Byte_info*" is a byte of flags that store certain information about a node. This way it is possible to access them all at the same time using masks

- **OPEN:** Flag that states if an specific node is open;
- **BS:** Flag that signalize an Bus Stop;
- **BUS:** Flag that signalize that the road is reserved only for public transport
- **EN:** Enables the node for processing.

9.6.2 Trace Table: Simple Route Test Steps

- **Step 1:** Test the pointer movement algorithm;
- **Step 2:** Test the route optimization algorithm;
- **Step 3:** Test all possible cases in the *"Open_nodes"*;
- **Step 4:** Test the system decision making by following this order of priorities:

  1 - Direction: If the next node has one direction and is towards the last, the jump must be refused;

  2 - Distance: The system must recognize the closest node to the goal and jump towards him. If the both distance are the same system doesn't jump to any of them;

  3 - Road Reserved to busses: The system must recognize only busses roads and jump to them or the first one he found;

  4 - Default: If no conditions is accepted the system jumps to the first opened node;

9.6.3 Trace Table: Simple Route Flowcharts Call

The thread *simple_route* calls the following flowcharts:

- **make_route;**
- **open_nodes and open_condition;**
- **decision_direction, decision_mask and decision_dist_bus;**
- **last_open;**
- **save_route;**
- **optimize_route;**

9.6.4 Trace Table: Simple Route First Step's Result

SimpleRoute and Make_route

Call	S	ERROR_FLAG	mem_route[gain][]	Index	multi_route[]	single_route	i_node	f_node	route_finish	optimize	max_gain	gain	pointer_node	next_node[]	map[y][x].Open	p_node.Open
call 1 from simpleRoute	1	0	N/A	N/A	N/A	N/A	node (0;1)	node (4;3)	false	0	0	0	node(0,1)	N/A	open	open
call 1 pointer.close	2		node(0,1)												close	close
call 1 open_nodes	3		N/A									1		node(0;2); node(0;0)		
call 1 decision_direction	4													node(0;2); node(0;0)		
call 1 decision_dist_bus	5												node(0,2)		open	open
	6			0												
	7		node(0;0)	1												
call 2 pointer.close	8		node(0;2); node(0;0)	2											close	close
	9	0	N/A	N/A	N/A	N/A	node (0;1)	node (4;3)	false	0	0	2	node(0,2)	node(0;2); node(0;0)	close	close
call 2 open_nodes	10													node(0;3)		
call 2 decision_direction	11												node(0;3)		open	open
call 3 pointer.close	12		node(0;3)												close	close
	14	0	N/A	N/A	N/A	N/A	node (0;1)	node (4;3)	false	0	0	3	node(0,3)	node(0;3)	close	close
call 3 open_nodes	15													node(1;3)		
call 3 decision_direction	16												node(1;3)		open	open
call 4 pointer.close	17		node(1;3)												close	close
	18	0	N/A	N/A	N/A	N/A	node (0;1)	node (4;3)	false	0	0	4	node(1;3)	node(1;3)	close	close
call 4 open_nodes	19													node(2;3)		
call 4 decision_direction	20												node(2;3)		open	open
call 5 pointer.close	21		node(2;3)												close	close
	22	0	N/A	N/A	N/A	N/A	node (0;1)	node (4;3)	false	0	0	5	node(2;3)	node(2;3)	ID = 1	close
call 5 open_nodes	23													node(2;2)		
call 5 decision_direction	24												node(2;2)			open
call 6 pointer.close	25		node(2;2)													close
	26	0	N/A	N/A	N/A	N/A	node (0;1)	node (4;3)	false	0	0	6	node(2;2)	node(2;2)	close	close
call 6 open_nodes	27													node(2;1)		
call 6 decision_direction	28												node(2;1)		open	open
call 7 pointer.close	29		node(2;1)												close	close

Figure 52: Make Route Trace Table First Call 1

Call	S	ERROR_FLAG	mem_route[gain][]	Index	multi_route[]	single_route	i_node	f_node	route_finish	optimize	max_gain	gain	pointer_node	next_node[]	map[y][x].Open	p_node.Open
	30	0	N/A	N/A	N/A	N/A	node (0;1)	node (4;3)	false	0	0	7	node(2;1)	node(2;1)	close	close
call 7 open_nodes	31													node(2;0)		
call 7 decision_direction	32												node(2;0)		open	open
call 8 pointer.close	33		node(2;0)												close	close
	34	0	N/A	N/A	N/A	N/A	node (0;1)	node (4;3)	false	0	0	8	node(2;0)	node(1;2)	close	close
call 8 open_nodes	35													node(3;0); node(1;0)		
call 8 decision_direction	36													node(3;0); node(1;0)		
call 2 decision_dist_bus	37												node(3;0)		open	open
	38			0												
	39		node(1;0)	1												
call 9 pointer.close	40		node(3;0);node(1;0)	2											close	close
	41	0	N/A	N/A	N/A	N/A	node (0;1)	node (4;3)	false	0	0	9	node(3;0)	node(3;0); node(1;0)	close	close
call 9 open_nodes	42													node(4;0)		
call 9 decision_direction	43												node(4;0)		open	open
call 10 pointer.close	44		node(4;0)												close	close
	45	0	N/A	N/A	N/A	N/A	node (0;1)	node (4;3)	false	0	0	10	node(4;0)	node(4;0)	close	close
call 10 open_nodes	46													node(4;1)		
call 10 decision_direction	47												node(4;1)		open	open
call 11 pointer.close	48		node(4;1)												close	close
	49	0	N/A	N/A	N/A	N/A	node (0;1)	node (4;3)	false	0	0	11	node(4;1)	node(4;1)	close	close
call 11 open_nodes	50													node(4;2)		
call 11 decision_direction	51												node(4;2)		open	open
call 12 pointer.close	52		node(4;2)												close	close
	53	0	N/A	N/A	N/A	N/A	node (0;1)	node (4;3)	false	0	0	12	node(4;2)	node(4;2)	close	close
call 12 open_nodes	54													node(4;3)		
call 12 decision_direction	55												node(4;3)		open	open
call 13 pointer.close	56		node(4;3)												close	close
call 1 save_route	57					route [(0;1)->(4;3)]										
returns to simpleRoute	58								true							

Figure 53: Make Route Trace Table First Call 2

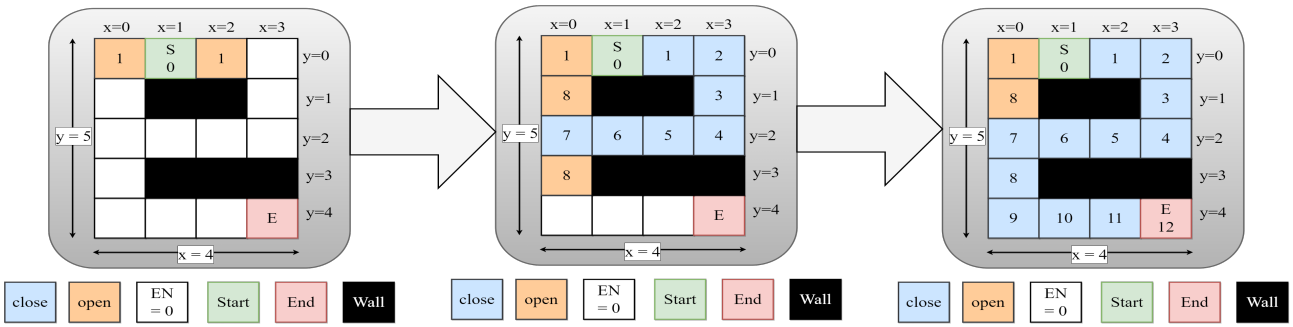

Figure 54: First Dry Run Scenario: Make Route

9.6.5 Trace Table: Simple Route Second Step's Result

Optimize_route

Calls	S	total_gain	i_node	f_node	pointer_node	route_finished
call 1 from simpleRoute	1	12	node(0;1)	node(4;3)	N/A	N/A
call 1 last_open	2				node(1;0)	
call 1 make_route	3					true
	4	8				
call 2 last_open	5				node(0;1)	
return to SimpleRoute	6					

Figure 55: Optimize Route Trace Table

Last_open

Calls	S	gain	index	mem_route.len	mem_route[gain].len	mem_route[gain][index].is_open	OUTPUT
call 1 from Optimize_route	1	12	0	13	1	close	
	2		1				
	3		0			open	
	4	11	1	12			
	5		0		1	close	
	6		1				
	7		0			open	
	8	10	1	11			
	9		0		1	close	
	10		1				
	11		0			open	
	12	9	1	10			
	13		0		1	close	
	14		1				
	15		0			open	
	16	8	1	9			
	17		0		2	close	
	18		1			open	
call 1 mem_route[gain][index].close()	19					close	node(1;0)

Figure 56: Last Open Trace Table first call

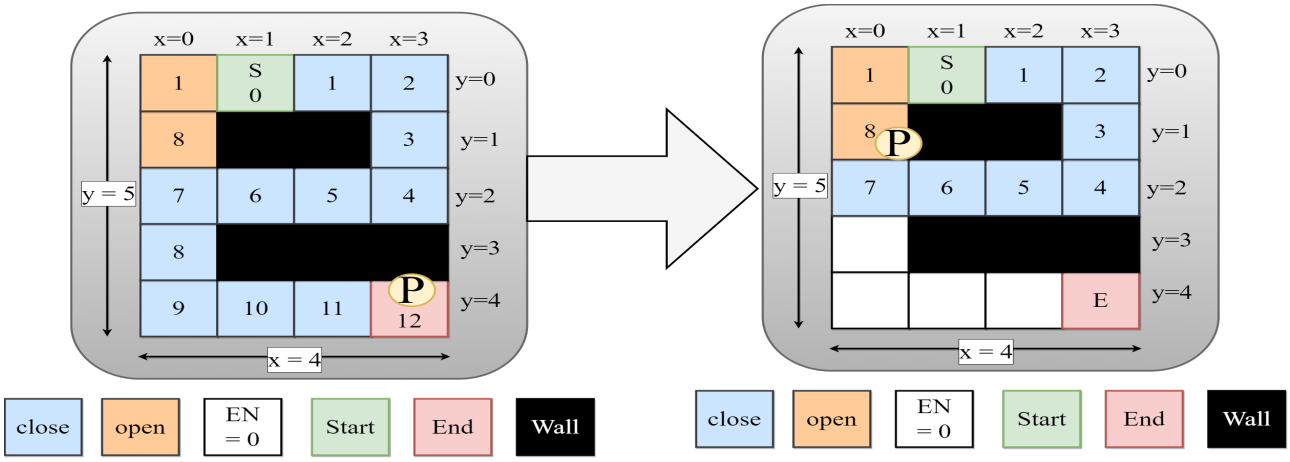

Figure 57: Second Dry Run Test Picked Scenario

Call	S	ERROR_FLAG	mem_route[gain][]	Index	multi_route[]	single_route	i_node	f_node	route_finish	optimize	max_gain	gain	pointer_node	next_node[]	map[y][x].Open	n_node.Open
call 1 from Optimize_route	60	0	node(3;0);node(1;0)	N/A		route [(0;1)->(4;3)]	node (0;1)	node (4;3)	false	1	12	8	node (1;0)	N/A	close	close
	61		N/A									9				
call 13 open_nodes	62													node(0;0)		
call 13 decision_direction	63												node(0;0)		open	open
call 14 pointer.close	64		node(0;0)												close	close
	65	0	N/A	N/A		route [(0;1)->(4;3)]	node (0;1)	node (4;3)	false	1	12	10	node(0;0)	node(0;0)	close	close
call 14 open_nodes	66													N/A		
call 1 last_open()	67		node(0;2); node(0;0)								12	1	node(0;0)			
	68	0	N/A	N/A		route [(0;1)->(4;3)]	node (0;1)	node (4;3)	false	1	12	2	node(0;0)	node(0;0)	close	close
call 15 open_nodes	69													node(1;0)		
call 14 decision_direction	70												node(1;0)		open	open
call 15 pointer.close	71		node(1;0)												close	close
	72	0	N/A	N/A		route [(0;1)->(4;3)]	node (0;1)	node (4;3)	false	1	12	3	node(1;0)	node(1;0)	close	close
call 16 open_nodes	73													node(2;0)		
call 15 decision_direction	74												node(2;0)		open	open
call 16 pointer.close	75		node(2;0)												close	close
	76	0	N/A	N/A		route [(0;1)->(4;3)]	node (0;1)	node (4;3)	false	1	12	4	node(2;0)	node(2;0)	close	close
call 17 open_nodes	77													node(3;0)		
call 16 decision_direction	78												node(3;0)		open	open
call 17 pointer.close	79		node(3;0)												close	close
	80	0	N/A	N/A		route [(0;1)->(4;3)]	node (0;1)	node (4;3)	false	1	12	5	node(3;0)	node(3;0)	close	close
call 18 open_nodes	81													node(4;0)		
call 17 decision_direction	82												node(4;0)		open	open
call 18 pointer.close	83		node(4;0)												close	close
	84	0	N/A	N/A		route [(0;1)->(4;3)]	node (0;1)	node (4;3)	false	1	12	6	node(4;0)	node(4;0)	close	close
call 19 open_nodes	85													node(4;1)		
call 18 decision_direction	86												node(4;1)		open	open
call 19 pointer.close	87		node(4;1)												close	close
	88	0	N/A	N/A		route [(0;1)->(4;3)]	node (0;1)	node (4;3)	false	1	12	7	node(4;1)	node(4;1)	close	close
call 20 open_nodes	89													node(4;2)		
call 19 decision_direction	90												node(4;2)		open	open
call 20 pointer.close	91		node(4;2)												close	close
	92	0	N/A	N/A		route [(0;1)->(4;3)]	node (0;1)	node (4;3)	false	1	12	8	node(4;2)	node(4;2)	close	close
call 21 open_nodes	93													node (4;3)		
call 20 decision_direction	94												node (4;3)			
call 21 pointer.close	95		node (4;3)													
call 2 save_route	96					route [(0;1)->(4;3)] (2)										
returns to OptimizeRoute	97								true							

Figure 58: Route Maker Trace Table Second Call

Calls	S	gain	index	mem_route.len	mem_route[gain].len	mem_route[gain][index].is_open	OUTPUT
call 1 from Route_Maker	20	10	0	11	1	close	
	21		1				
	22		0			open	
	23	9	1	10			
	24		0		1	close	
	25		1				
	26		0			open	
	27	8	1	9			
	28		0		1	close	
	29		1				
	30		0			open	
	31	7	1	6			
	32		0		1	close	
	33		1				
	34		0			open	
	35	4	1	5			
	36		0		1	close	
	37		1				
	38		0			open	
	39	3	1	4			
	40		0		1	close	
	41		1				
	42		0			open	
	43	2	1	3			
	44		0		1	close	
	45		1				
	46		0			open	
	47	1	1	2			
	48		0		2	close	
	49		1			open	
call 1 mem_route[gain][index].close()	50					close	node(0;0)

Figure 59: Last Open Trace Table Second Call

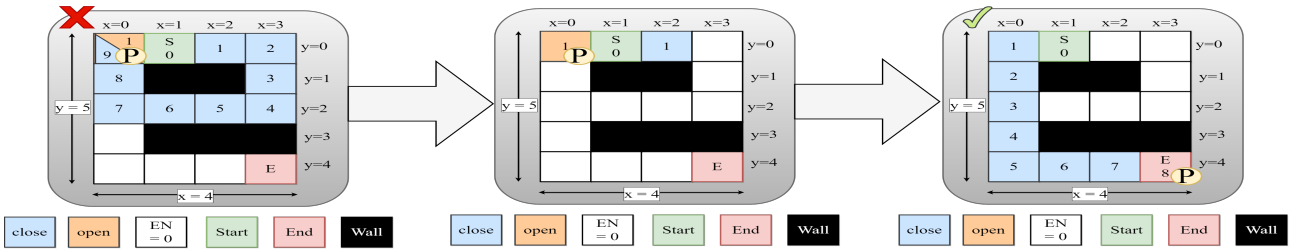

Figure 60: Third Dry Run Test Picked Scenario

9.6.6 Trace Table: Simple Route Third Step's Result

For a node to be opened, the following rules must be met:

- The coordinates of the nodes must be within the limits of the map;
- The node cannot be closed;
- The node must be a road (*"Byte_info"* must be different than zero);

The figure 61 represents some of the result of the open conditions in the step 1:

OPEN_NODES and OPEN_CONDITION

Calls	S	x	y	x_cond	y_cond	x_len	y_len	gain	map[x_cond][y_cond].byte_info	map[x][y].byte_info	map[x_cond][y_cond].Open	flag_open	temp_list_node
call 1 from make_route	1	1	0			4	5	1		0100 1100			N/A
call 1.1 open_condition	2			2	0				0000 1100		open	true	
	3								1000 1100				node(0;2)
call 1.2 open_condition	4			0	0				0000 0101		open	true	
	5								1000 0101				node(0;2); node(0;0)
call 1.3 open_condition	6			1	1				0000 0000			false	
call 1.4 open_condition	7				-1				N/A			false	
call 2 from make_route	8	2	0			4	5	2		0000 1100			N/A
call 2.1 open_condition	9			3	0				0000 1001		open	true	
	10								1000 1001				node(0;3)
call 2.2 open_condition	11			1	0						close	false	
call 2.3 open_condition	12											false	
call 2.4 open_condition	13											false	
call 3 from make_route	14	3	0			4	5	3		0000 1001			
call 3.1 open_condition	15											false	
call 3.2 open_condition	16			2	0						close	false	
call 3.3 open_condition	17			3	1				0000 0011		open	true	
	18								1000 0011				node(1;3)
call 3.4 open_condition	19			3	-1							false	
call 4 from make_route	20	3	1			4	5	4		0000 0011			N/A
call 4.1 open_condition	21											false	
call 4.2 open_condition	22											false	
call 4.3 open_condition	23			3	2				0000 1010		open	true	
	24								1000 1010				node(2;3)
call 4.4 open_condition	25			3	0						close	false	
call 5 from make_route	26	3	2			4	5	5		0000 1010			N/A
call 5.1 open_condition	27											false	
call 5.2 open_condition	28			2	2				0000 1000		open	true	
	29								1000 1000				node(2;2)
call 5.3 open_condition	30											false	
call 5.4 open_condition	31			3	1						close	false	

Figure 61: Open Node Trace Table 1

The figure 62 represents some of the result of the open conditions in the step 2:

Calls	S	x	y	x_cond	y_cond	x_len	y_len	gain	map[x_cond][y_cond].byte_info	map[x][y].byte_info	map[x_cond][y_cond].Open	flag_open	temp_list_node
call 8 from make_route	32	0	2			4	5	8		0000 0011			N/A
call 8.1 open_condition	33											false	
call 8.2 open_condition	34											false	
call 8.3 open_condition	35			0	3				0000 0001		open	true	
	36								1000 0001				node(3;0)
call 8.4 open_condition	37			0	1				0000 0011		open	true	
	38								1000 0011				node(3;0); node(1;0)
call 13 from make_route	39	0	0			4	5	9		0000 1100			N/A
call 13.1 open_condition	40			0	1						close	false	
call 13.2 open_condition	41			0	-1							false	
call 13.3 open_condition	42											false	
call 13.4 open_condition	43											false	

Figure 62: Open Node Trace Table 2

9.6.7 Trace Table: Simple Route Fourth Step's Result

All decision made in step one and step two (figure 63 and figure 64) according to the priorities defined:

Decision_dist_bus

Calls	S	index	list_node	list_node.len	list_node.dist[index]	list_node.dist[index +1]
call 1 from make_route	1	0	node(0;2); node(0;0)	2	4,123105	5
	2		node(0;2);	1		
	3	0		1		
call 2 from make_route	4	0	node(3;0); node(1;0)	2	3,162278	4,24264
	5		node(3;0)	1		
	6	0		1		

Figure 63: Distance and Reserved Only Busses Decision Trace Table

DECISION_DIRECTION

Calls	S	index	list_node	list_node.len	x	y	x_pointe	y_point	byte_info	MASK
call 1 from make_route	1	0	node(0;2); node(0;0)	2						
call 1.1 direction_mask	2				2	0	1	0		0000 0111
	3								1000 1100	
	4	1								
call 1.2 direction_mask	5				0	0	1	0		0000 1011
	6	2							1000 0101	
	7		node(0;2); node(0;0)							
call 2 from make_route	8	0	node(0;3)	1						
call 2.1 direction_mask	9				3	0	2	0		0000 0111
	10								1000 1001	
	11	1	node(0;3)							
call 3 from make_route	12	0	node(1;3)	1						
call 3.1 direction_mask	13				3	1	3	0		0000 1101
	14								1000 0011	
	15	1	node(1;3)							
call 4 from make_route	16	0	node(2;3)	1						
call 4.1 direction_mask	17				3	2	3	1		0000 1101
	18								1000 1010	
	19	1	node(2;3)							
call 5 from make_route	20	0	node(2;2)	1						
call 5.1 direction_mask	21				2	2	3	2		0000 1011
	22								1000 1000	
	23	1	node(2;2)							
call 8 from make_route	24	0	node(3;0); node(1;0)	2						
call 5.1 direction_mask	25				0	3	0	2		0000 1101
	26								1000 0001	
	27	1								
call 5.2 direction_mask	28				0	1	0	2		0000 1110
	29								1000 0011	
	30	2	node(3;0); node(1;0)							

Figure 64: Direction Decision Trace Table

10 Budget

The estimated budget for BUSIT is showed in the table 5.

Table 5: Project Budget

Amount	Component Description	Unit Price	Q Price
1	KIT RASPBERRY PI 4B	110 €	110 €
2	STM32 Nucleo-F767ZI	36 €	72 €
2	DISPLAY I2C LCD1602 16X2 with DRIVER	7.90 €	15.80 €
2	BOARD WIFI AIR602 (COM02048)	5 €	10 €
1	Step-Down 5V	8 €	8 €
2	FTDI USB-UART	9.80 €	19.60 €
TOTAL			**235.40€**

11 Gantt Chart

The Gantt chart is a powerful tool when it comes to planning and scheduling the different tasks of a project, specially those ones that happens in concurrency. The following chart (figure 65) shows the BUSIT project Gantt chart.

Figure 65: Gantt Chart

References

[1] Apple. Transit • subway bus times. https://apps.apple.com/pt/app/transit-subway-bus-times/id498151501. accessed on September 28, 2022.

[2] SAR Soluções de Automação e Robótica. Display i2c lcd1602 16x2 c/ driver - azul. https://www.botnroll.com/pt/lcds-e-displays/2989-display-i2c-lcd1602-16x2-c-driver-p-funduino-azul.html. accessed on October 26, 2022.

[3] SAR Soluções de Automação e Robótica. Step-down para 5v atÉ 8a 40w com 4 portas usb entrada 8v 35v. https://www.botnroll.com/pt/conversores-dcdc/3650-step-down-para-5v-at-8a-40w-com-4-portas-usb-entrada-8v-35v.html. accessed on November 11, 2022.

[4] SAR Soluções de Automação e Robótica. Wifi board (air602). https://www.botnroll.com/pt/ethernet-wi-fi/4085-placa-de-desenvolvimento-wifi-air602.html. accessed on October 26, 2022.

[5] Circuit Digest. 16x2 lcd display module. https://circuitdigest.com/article/16x2-lcd-display-module-pinout-datasheet. accessed on November 17, 2022.

[6] Digi-Key Electronics. Stm32 nucleo-144 board manual datasheet by stmicroelectronics. https://www.digikey.com/htmldatasheets/production/1867435/0/0/1/nucleo-h743zi2.html#pf45. accessed on November 11, 2022.

[7] imarc. Bus market: Global industry trends, share, size, growth, opportunity and forecast 2022-2027. https://www.imarcgroup.com/bus-market. accessed on September 28, 2022.

[8] Arm Limited. Nucleo-f767zi. https://os.mbed.com/platforms/ST-Nucleo-F767ZI/. accessed on November 11, 2022.

[9] Raspberry Pi (Trading) Ltd. Raspberry pi 4 model b datasheet. https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf. accessed on November 10, 2022.

[10] Raspberry Pi (Trading) Ltd. Raspberry pi documentation. https://www.raspberrypi.com/documentation/computers/raspberry-pi.html. accessed on November 10, 2022.

[11] Moovit. Tempo real para ganhar a fidelidade dos seus passageiros. https://moovit.com/pt/maas-solutions-pt/timepro/. accessed on September 28, 2022.