
UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS ESPECÍFICAS DE TELECOMUNICACIÓN

# Four level autonomous vehicle for an automatized parking

Autor:

**D. Ignacio Royuela González**

Tutor:

**Dr. D. Juan Carlos Aguado Manzano**

**D. Jairo Gurdiel González**

VALLADOLID, JULIO 2020

**TRABAJO FIN DE GRADO**

| | |
|---|---|
| TÍTULO: | **Four level autonomous vehicle for an automatized parking** |
| AUTOR: | **D. Ignacio Royuela González** |
| TUTOR: | **Dr. D. Juan Carlos Aguado Manzano** |
| | **D. Jairo Gurdiel González** |
| DEPARTAMENTO: | **Teoría de la Señal y Comunicaciones e Ingeniería Telemática** |

**TRIBUNAL**

| | |
|---|---|
| PRESIDENTE: | **Dr. D. Ignacio de Miguel Jiménez** |
| VOCAL: | **Dr. D. Ramón Durán Barroso** |
| SECRETARIO: | **Dr. D. Juan Carlos Aguado Manzano** |
| SUPLENTE: | **Dr. Dª Noemí Merayo Álvarez** |
| SUPLENTE: | **Dr. D. Juan Blas Prieto** |

FECHA:

CALIFICACIÓN:

## Resumen de Trabajo Fin de Grado

En este Trabajo Final de Grado se han cumplido una serie de objetivos para la implementación de un sistema acoplable a un vehículo para darle capacidades de vehículo autónomo de nivel 4 para un aparcamiento automatizado. Se realiza un diseño completo de los sistemas necesarios y su interconexión, haciendo un análisis de las tecnologías óptimas para el proyecto, su coste, y los elementos disponibles en el mercado, estudiando su viabilidad en el proyecto. Se diseña una interfaz de control del vehículo a través de un protocolo apropiado, mediante el cual un back-end puede enviar órdenes al vehículo y recibir información de este; y una mensajería CAN para mantener sincronizados y permitir una gestión de errores de los dos módulos de procesamiento que utiliza el sistema. Se implementa el software necesario en dichos módulos para que lleven a cabo las tareas de comunicación entre ellos y con el back-end. Finalmente, se desarrolla un simulador de los sistemas implementados que permite realizar múltiples simulaciones de estos simultáneamente variando ciertos parámetros, para así realizar una evaluación de la integración de los sistemas del vehículo con el back-end y comprobar el correcto funcionamiento del sistema completo en la realidad.



## Abstract

In this Final Degree Project, several objectives have been achieved for the implementation of a system that can be coupled to a vehicle to give it level 4 autonomous vehicle capabilities for an automatized parking. A complete design of the necessary systems and their interconnection is carried out, performing an analysis of the optimal technologies for the project, its cost, and the elements available in the market, studying their viability in the project. A vehicle control interface is designed via an appropriate protocol, through which a back-end can send orders to the vehicle and receive information from it; and a CAN messaging to keep synchronized and allow error management of the two processing modules used by the system. The necessary software is implemented in these modules to carry out the communication tasks between them and with the back-end. Finally, a simulator of the implemented systems is developed, that allows multiple simulations of these simultaneously varying certain parameters. This way it is possible to carry out an evaluation of the integration of the vehicle systems with the back-end, and to check the correct operating of the complete system in reality.



## Keywords

ODD, DDT, OEDR, AGV, line-tracking system, carsharing, CAN, MQTT, RFID, process, work queue, back-end.

# Acknowledgements

*First of all, I would like to thank my university tutor Juan Carlos Aguado Manzano for giving me and my colleagues the opportunity to enter the Twizy Contest and, in general, for his commitment to the education of his students. Also, to my colleagues of the TwizyLine team for their passion and help with the project.*

*I would like to thank Jairo Gurdiel González and Daniel Castaño Navarro, qualified personnel from Renault, for their constant involvement and support in the project.*

*I would also like to thank the company Maxon for their interest and support to the project with the donation of an electric motor and a controller. This donation has been very important to ensure the future viability of the project.*

*Finally, I would like to thank my parents and Paula Méndez Mediavilla for supporting me and motivating me to carry out this project. There have been stressful times and they encouraged me to do my best.*

*To all of them, thank you very much.*

# Index

# Figure index

# Table index

# 1. Introduction

## 1.1. Project motivation

Today, private vehicle is one of the most commonly used means of transport. In Spain, there are 24.5 million private cars [1], which corresponds to half of the population over 18 years old [2] and therefore, with the possibility of driving a vehicle. This means of transport currently confronts several problems. Due to its widespread use, it is of vital importance to eliminate these problems and to optimize the use of this means of transport. These problems are related to traffic accidents and pollution due to low energy efficiency.

Traffic accidents are one of the main problems in today's transport sector. In 2018 there were 102,299 accidents with victims only in Spain. The total number of injured was 140,415 people, of these 1806 died [3]. This rate has varied over time (Figure 1). Since the private car became a common product, the death rate in traffic accidents has been increasing until 1989. From this year onwards, thanks to public awareness, the improvement of vehicle safety systems (safety bels, ABS technology, airbags...), and the improvement of signalling and road conditions, the death rate has been falling. However, since 2012, the death rate has stabilized and in recent years there has been no decrease [3].

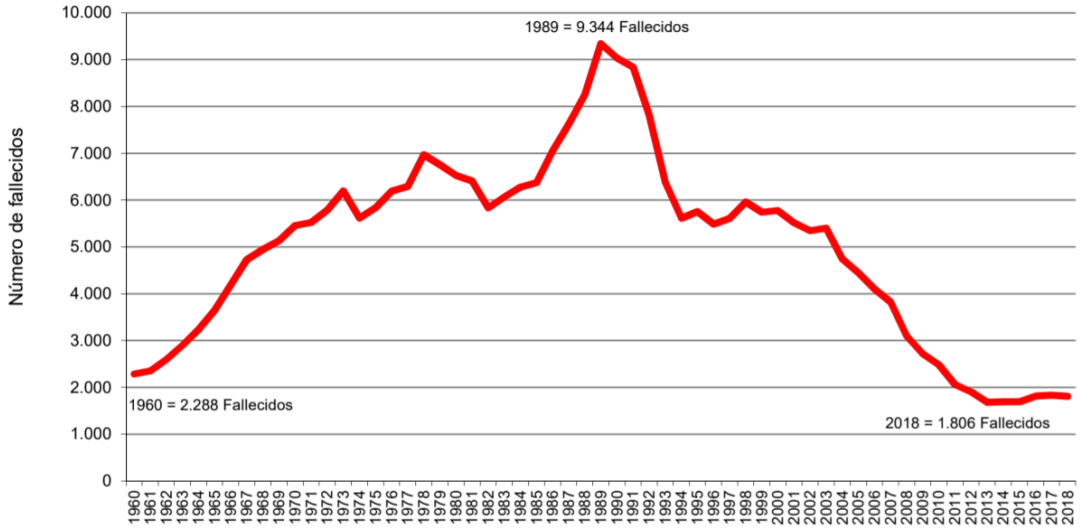



Figure 1. Death rate in Spain due to traffic accidents per year [3].

Currently, most of these accidents are due to human error. According to a study made by the National Highway Traffic Safety Administration of the United States [4], approximately 94% of accidents that occur in this country are caused by the driver (Table 1).

| ***Critical Reason Attributed to*** | ***Estimated*** | |
|---|---|---|
| | Number | Percentage ± 95% conf. limits |
| Drivers | 2,046,000 | 94% ±2.2% |
| Vehicles | 44,000 | 2% ±0.7% |
| Environment | 52,000 | 2% ±1.3% |
| Unknown | 47,000 | 2% ±1.4% |
| Total | 2,189,000 | 100% |

Table 1. Driver, Vehicle, and Environment Related Critical Reasons [4].

The main causes of accidents caused by drivers are the driver's inattention, internal and external distractions, decision errors such as driving too fast, or performance. In general, human factors, which are difficult to remedy, are in the causation of most of the road traffic crashed (Table 2).

| ***Critical Reason Attributed to*** | ***Estimated*** | |
|---|---|---|
| | Number | Percentage ± 95% conf. limits |
| Recognition Error | 845,000 | 41% ±2.2% |
| Decision Error | 684,000 | 33% ±3.7% |
| Performance Error | 210,000 | 11% ±2.7% |
| Non-Performance Error (sleep, etc.) | 145,000 | 7% ±1.0% |
| Other | 162,000 | 8% ±1.9% |
| Total | 2,046,000 100% | 100% |

Table 2. Driver-Related Critical Reasons [4].

Cities are a major focus for traffic accidents. In Spain, 63% of traffic accidents with total victims occur in urban areas [3].

Another major transport problem is pollution. Transport is responsible for a substantial fraction of worldwide energy consumption and greenhouse gas emissions. While emissions from other sectors are generally decreasing, those from transportation have increased since 1990 [5]. It is necessary to reduce the impact of transport on the environment, a task that can only be achieved if passenger vehicles are considered one of the main problems, since they constitute almost half of the entire sector [5]. Improving the energy efficiency of these vehicles can be a major step forward in this area.

In any motor vehicle, there are always 3 main energy conversions of energy (Figure 2). The first consists of the distribution of energy to the vehicle from the energy sources, either gasoline or electricity. Through the second conversion, this energy is then used to move the wheels of the vehicle. In the third conversion, the mechanical energy of the wheels is converted into kinetic and potential energy required to produce the movement. All these conversions cause relevant energy losses, a waste of energy. Therefore, it is necessary to optimize the efficiency of this process.

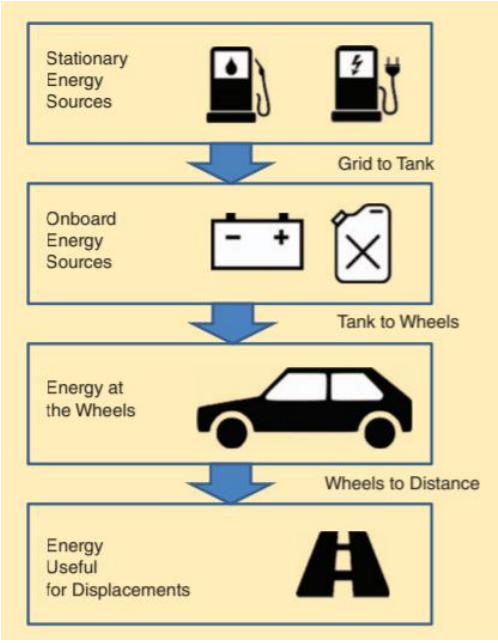



Figure 2. The three main energy conversions steps for road vehicles [5].

The efficiency of the second conversion (Tank to Wheels) is determined by the type of propulsion engine being used, its characteristics, and the friction forces in the vehicle transmission. Electric motors can use 77% of the electrical energy. On the other hand, conventional combustion engines are only capable of converting between 12% and 30% of the energy stored in gasoline [6].

In the third conversion efficiency, several vehicle factors are involved, but also parameters that depend on the driver's profile. The following table (Table 3) shows the relative influence of vehicle parameters and driving profiles on the wheels-to-distance energy efficiency of a typical full-size passenger car. Sensitivity is a parameter that gives the percentage of reduction of energy consumption per percentage of reduction of the considered variable.

| ***Vehicle Parameter*** | ***Sensitivity*** |
|---|---|
| Weight | 0.7 |
| Rolling friction | 0.35 |
| Aerodynamics | 0.3 |
| ***Driving Profile Parameter*** | ***Sensitivity*** |
| Mean square speed | 0.6 |
| Mean acceleration | 0.35 |

Table 3. Influence of vehicle parameters and driving profiles on the wheels-to-distance energy efficiency [7].

Vehicle parameters are being improved year after year, lighter materials, aerodynamic studies... to improve efficiency and, therefore, less pollution and greater energy savings. However, the parameters that depend on driving the vehicle are more complicated to control. A very aggressive driving of the vehicle, braking and acceleration improperly, poor anticipation, not respecting the safety distance, and in general poor driving may decrease the efficiency in enormous amounts [8] and consequently, nullify the advances achieved.

Fortunately, the world of transport is experiencing one of the greatest innovations in the world economy. This is the development of Automated Driving Systems (ADSs), commonly referred to as automated or self-driving vehicles. This technology, applied to the maximum of its capabilities, will allow the driver to become a passenger, who can leave the control of the vehicle to the system that it incorporates. As seen before, the human factor is the cause of most traffic accidents. If this type of error can be avoided, the death rate will be reduced. ADSs have the potential to accomplish exactly this by addressing the root cause of these tragic crashes. These systems can save thousands of lives, as well as reduce congestion, enhance mobility, and improve productivity [9].

The following table (Table 4) shows the driving strengths of a human versus a machine today.

| ***Aspect*** | ***Human*** | ***Machine*** |
|---|---|---|
| Sensing | Excellent vision<br>Strong inertial (inner ear) | Strong Multi-sensor<br>Direct measurement |
| Perception | Excellent detection, tracking and prediction | Poor detection, tracking and prediction |
| Information Processing | Single focus<br>Limited attention | Excellent multiple source processing<br>Perfect attention |
| Memory | Strong recall of strategies and data<br>Flexible | Precise recall of detail<br>Rigid |
| Reasoning | Intuitive, approximate<br>Excellent at handling ambiguity<br>Excellent recall and learning | Deductive, precise<br>Poor at handling ambiguity<br>Excellent fleet-wide learning and recall |
| Reaction Speed | Slow, inconsistent | Fast, limited by computation |
| Command Output | Sufficient, inconsistent | Strong, Precise |

Table 4. Human vs. Machine - Driving Strengths [10].

Although ADSs have certain drawbacks, especially around prediction and handling of ambiguous situations, they offer very interesting advantages related to the speed of reaction, the parallel processing of multiple information sources, and a stronger control of the vehicle.

Moreover, the difficulties in which the sector finds itself about efficient driving were previously discussed. Thanks to the precise control of the vehicle by the machine, driving efficiency can be improved. ADSs allows to optimize energy consumption by using only the energy needed to travel, without unnecessary acceleration and braking, and with anticipation.

In view of the current transport problems, and the emergence of these new technologies, in September 2019, Renault unveiled its third call for the "Twizy Contest", an international competition in which participants from universities around the world must provide this time a mobility solution of transport at the 2024 Olympic Games in Paris. The contest asks the students to find a frugal, durable, and innovative solution to facilitate the access to the different events reducing the environmental impact [11]. It also recommends using the Twizy model to achieve these goals. The student team made up of Adrián Mazaira, Mario Martín, Samuel Pilar and Ignacio Royuela from the Universidad de Valladolid participated in the competition and is currently representing Spain.

The proposed project is called *TwizyLine* (Figure 3). The idea is to reinvent the car-sharing service through autonomous car parks based on line tracking systems. Users, through a mobile application, can request a vehicle that will be delivered to the pickup zone automatically. When users wish to leave the vehicle in a car park, they must only leave it in the leaving zone and the vehicle will be automatically parked (Figure 4).

Figure 3. Logotype and symbol of *TwizyLine*.

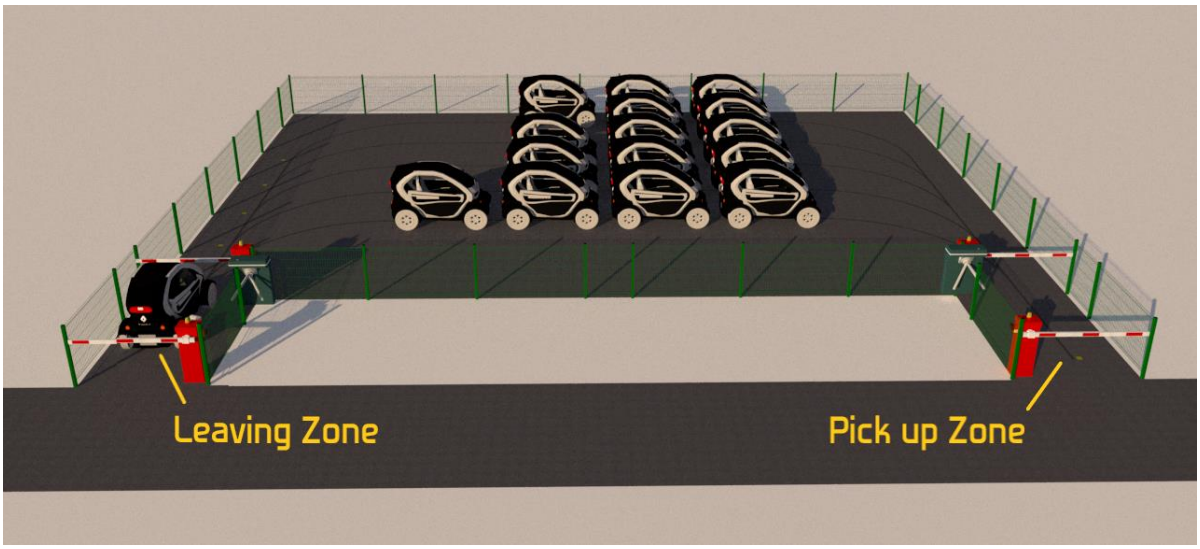



Figure 4. Illustration of a *TwizyLine* car park.

This project takes advantage of the benefits of autonomous vehicles to provide an efficient and innovative solution. Thanks to the autonomy of the vehicles, they can be parked in much smaller areas so that the use of space is more efficient. Users do not have to worry about finding parking spaces and performing complicated parking manoeuvres, they simply must leave the vehicle in the corresponding area and the system takes care of the rest. Therefore, the risk of possible accidents

when parking in urban areas is avoided. Since the vehicles are in a controlled environment when they are operating autonomously, the disadvantages of ADSs about ambiguity management are minimized. Also, the project promotes the electric vehicle since the project is designed to be implemented in the Twizy vehicle and, as seen before, this type of vehicle is much more efficient.

To carry out the implementation of this project, some main objectives have been defined:

1. **Front-end** implementation in a smartphone application: Development of user software capable of identifying and locating the user. It must allow to request a vehicle and to implement the interfaces of pick-up and release of this one. It must also have a special interface for administrators to view the status of all vehicles in real-time. This objective has been developed in the Final Degree Project of Adrián Mazaira [12].
2. **Back-end** implementation for the control of autonomous parking: The system must be able to monitor and update a database with the status of all vehicles in the service and keep track of all users. It must also be capable of giving the necessary instructions to the vehicles for the execution of the service without risk to the customer. This objective has been developed in the Final Degree Project of Samuel Pilar [13].
3. **Design** of a system to add **autonomy capabilities** to a vehicle and **implementation** of **ECUs** responsible of control and communication of the vehicle: The vehicle must be controllable through a defined interface and a simulator of the vehicle's operation in reality must be carried out to check the integration of this system with the back-end system. This objective has been developed in this Final Degree Project.
4. System **integration** and **installation** in a real vehicle: The designed modifications in the previous point must be made. All the necessary sensors and actuators must be integrated, and their operation must be checked. This is the main future line of this project.

Therefore, this final degree project explains how objective 3 of the joint project *TwizyLine* was developed.

## 1.2. Objectives

This project has two main purposes. On the one hand, to carry out a complete design of a vehicle-mounted system to give it level 4 autonomous vehicle capabilities, and the development of the ECUs that are part of this design. On the other hand, to develop a simulator of this system to simulate several vehicles at the same time and to check the integration of this system with the back-end.

The specific objectives covered in this project are the following:

- Design of a system that can be attached to a vehicle in order to add level 4 autonomy capabilities using line following system.
- Development of the necessary software in two ECUs that work in a synchronized way.
- Design and implementation of a vehicle communication system through the CAN bus to achieve this synchronization.
- Design and implementation of a communications interface through an appropriate protocol for vehicle control.
- Management of errors in this system.
- Development of a simulator of the previously developed ECUs in order to simulate several vehicles at the same time and to check the integration of this system with the back-end.
- Development of a real-time viewer for "Google Earth" that allows to see the status of the whole system in real-time.

## 1.3. Stages and methods

To carry out this project, a series of phases were followed:

1. Initially, a **review of current technological developments** about the autonomous vehicle and automatic guided vehicles (AGVs) was performed. With this in mind, a study of the technologies needed to implement an autonomous vehicle according to the needs of our project was carried out.
2. Then a **design of the system** to be implemented in the vehicle was made and the necessary **software** was **developed** in the added ECUs.
3. Next, an adaptation of the original code was made to **create a simulator** capable of emulating the behaviour and messaging of several vehicles simultaneously.
4. Afterwards, an **evaluation of the use cases of the real system**, programmed in the ECUs, was carried out, checking success cases and possible failures that may occur in practice.
5. Finally, an **evaluation of the integration** of several simultaneous vehicles with the back-end was carried out to prepare the "Twizy Contest" competition.

It is important to remember the two main objectives of this project mentioned in section 1.2. Phases 2 and 4 are necessary to carry out the design of the system and the development of the ECUs. Phase 4 evaluates all the use cases that can occur with the vehicle. In this phase the check is made without the need of back-end, only through messages sent manually following the designed vehicle interface. This is because, since the system is not mounted in a real vehicle, checking the operation of the systems otherwise would not be valid. On the other hand, phases 3 and 5 are necessary to create the simulator and check the integration with the back-end. The objective of creating the simulator is precisely to be able to test with the back-end. In this case, the simulated vehicles, which behave as the real vehicle, would send GPS coordinates consistent with the service to be provided and, therefore, expected by the back-end.

## 1.4. Resources

To develop this project, the following materials have been used, which have been chosen specifically for this purpose:

- 2 *Humming Board CBi* single-board computers [14] to operate as ECUs. These have a 1 GHz ARM processor, 2GB of ram, 1 CAN interface, 1 RS485 interface, 4 USB 2.0, 1 RJ45 port, and a PCI express port with SIM card slot. They are compatible with the 4.4x Linux kernel, resistant to extreme conditions, and have an extruded aluminium (IP32) enclosure. They also allow a power input between 7 and 36 volts which gives them a lot of flexibility.
- 1 Garmin *GPS18x USB* GPS receiver [15]. A Garmin GPS18x USB GPS receiver. It incorporates differential DGPS capability using real-time WAAS corrections yielding position accuracy of less than 3 meters. It is waterproof and uses Garmin's private binary protocol to transmit information via a USB interface.
- 1 *Huawei 4G Dongle E3372* 4G modem [16] to give the vehicle an internet connection to communicate. It is connected via USB and is compatible with 4G, LTE, UMTS, HSUPA, 2G, FDD, UMTS, and GSM networks. Through 4G it can support up to 150Mbps.

Apart from these main resources, cables, resistors, and LEDs have been used to make a CAN bus that simulates that of the real vehicle and an information panel that gives the status of the system in real-time.

# 2. State of the art

In this chapter, a complete overview of the autonomous vehicle will be given, explaining the key concepts of it, the technology currently used, and the different levels of driving automation that exist today. It also shows a review of the technologies used to implement an Automatic Guided Vehicle (AGV) since the vehicle to be implemented will use one of these technologies. Then, based on these studies, the comparisons made and the conclusions reached it will be explained in order to carry out the project.

## 2.1. Autonomous vehicle

When defining an autonomous vehicle, it is necessary to previously define a set of key concepts in order to clarify the different levels of autonomy and the issues that the vehicle should be able to solve in every one of them. These concepts that are summarized below are defined by the SAE standardization organization in its J3016 standard [17]. It gives a logical taxonomy for classifying driving automation features along with a set of terms and definitions that support the taxonomy and otherwise standardize related concepts, terms, and usage in order to facilitate clear communications. The standard defines more than 30 concepts. Here I only introduce those that are crucial to understand the main questions of autonomous vehicles.

*Dynamic driving task* (**DDT**) refers to all the necessary tasks for driving a vehicle. It comprises all of the real-time operational and tactical functions required to operate a vehicle in on-road traffic. It can be performed by a human driver or a machine. In the case this task can be carried out by a machine ("collection of software and hardware that are collectively capable of performing the entire DDT on a sustained basis" [17]) the whole system is called an *automated driving system* (**ADS**). When this system takes control of the vehicle, i.e. performs the DDT, the vehicle is in *driverless operation*. In this context, the term *dispatch* refers to place an ADS-equipped vehicle into service in *driverless operation* by engaging the ADS. Therefore, the entity that performs this action is called *dispatching entity*. Depending on the level of autonomy of the implemented system it will have different conditions to achieve this goal. These performance limitations depend on the *operational design domain* (**ODD**) of the system. The limitations of the designed system can be, among others, environmental, geographical, time-of-day restrictions, legislative and/or presence or absence of certain traffic or road characteristics. It is critical to define what happens when the ODD does not meet the conditions for the autonomous vehicle to be in driverless operation. This is called an ODD exit.

To carry out DDT, two main sub-tasks are necessary. On the one hand, *sustained longitudinal and transverse control of the vehicle* in real-time is required. This includes the basic driving tasks, such as following the road and not colliding with vehicles in front of it. On the other hand, *object and event detection and response* (**OEDR**) is also required. This last sub-task must monitor the driving environment and execute an appropriate response to these objects and events.

Another relevant task in autonomous driving is the achievement of the *minimal risk conditions*, that is, in the case of failure of DDT, or upon ODD exit, or the detection by the ADS of being in that condition (e.g. accident risk caused by another vehicle), to reduce the risk of a crash when a given trip cannot or should not be completed. This task is called **DDT fallback** and, like the DDT task, can be carried out by a human or a machine depending on the level of driving automation of the system. Note that the DDT and the DDT fallback are distinct functions, and the capability to perform one does not necessarily entail the ability to perform the other. Depending on the level of autonomy of the implemented system, and according to the situation, the system may ask the conventional driver to resume control of the vehicle. This request is called a *request to intervene*.

In order to carry out a risk analysis of an autonomous vehicle, it is necessary to know a series of concepts defined by the *International Organization of Standardization* (ISO) in its ISO 26262 standard [18]. *Harm* means physical damage to a living being. *Risk* is the probability of a certain event occurring that implies damage. *Safety* is the process that avoids risks of unreasonable damage. *Hazard* is a potential source of unreasonable harm risk or safety threat

Based on the capabilities of a vehicle's ADS systems, SAE defines 6 levels of driving automation (Table 5).

| Level | Name | Narrative definition | DDT | | DDT fallback | ODD |
|---|---|---|---|---|---|---|
| | | | Sustained lateral and longitudinal vehicle motion control | OEDR | | |
| **Driver performs part or all of the DDT** | | | | | | |
| 0 | **No Driving Automation** | The performance by the *driver* of the entire *DDT*, even when enhanced by *active safety systems*. | *Driver* | *Driver* | *Driver* | *n/a* |
| 1 | **Driver Assistance** | The *sustained* and *ODD*-specific execution by a *driving automation system* of either the *lateral* or the *longitudinal vehicle motion control* subtask of the DDT (but not both simultaneously) with the expectation that the *driver* performs the remainder of the *DDT*. | *Driver and System* | *Driver* | *Driver* | *Limited* |
| 2 | **Partial Driving Automation** | The *sustained* and *ODD*-specific execution by a *driving automation system* of both the *lateral* and *longitudinal vehicle motion control* subtasks of the *DDT* with the expectation that the *driver* completes the *OEDR* subtask and *supervises* the *driving automation system*. | *System* | *Driver* | *Driver* | *Limited* |
| ***ADS* ("*System*") performs the entire *DDT* (while engaged)** | | | | | | |
| 3 | **Conditional Driving Automation** | The *sustained* and *ODD*-specific performance by an *ADS* of the entire DDT with the expectation that the *DDT fallback-ready user* is *receptive* to *ADS*-issued *requests to intervene*, as well as to *DDT performance relevant system failures* in other *vehicle* systems, and will respond appropriately. | *System* | *System* | *Fallback ready user (becomes the driver during fallback)* | *Limited* |
| 4 | **High Driving Automation** | The *sustained* and *ODD*-specific performance by an *ADS* of the entire *DDT* and *DDT fallback* without any expectation that a *user* will respond to a *request to intervene*. | *System* | *System* | *System* | *Limited* |
| 5 | **Full Driving Automation** | The *sustained* and unconditional (i.e., not *ODD* specific) performance by an *ADS* of the entire *DDT* and *DDT fallback* without any expectation that a *user* will respond to a *request to intervene*. | *System* | *System* | *System* | *Unlimited* |

Table 5. Summary of levels of driving automation [17].

**Level 0** in the classification refers to all vehicles that do not have any kind of driving automation. The driver must perform the entire DTT. This level also includes vehicles that contain warning or support systems in very specific cases, such as a momentary emergency intervention.

**Level 1** contains all vehicles that can, via a driving automation system, take longitudinal or transversal control of the vehicle in specific environments, that is with a limited ODD. Note that this level only includes vehicles that can autonomously control one of the two main parameters of the vehicle control, that is, either the longitudinal or the lateral vehicle motion control subtask of the DDT. The rest of the DDT tasks remain in the hands of the conventional driver. A clear example of this level is the well-known adaptive cruise control system. This system allows vehicles to maintain their longitudinal control by programming a cruising speed at which the driver wishes to move. If the system detects a vehicle driving ahead, it reduces the speed of the vehicle to maintain a safe distance. Therefore, this system is only valid for highways or roads without traffic lights or other limitations that depend on more factors than the possibility of having a vehicle in front of it. The driver continues to have transversal control through the steering wheel, he must supervise the performance of ADS and, of course, he is the one who must intervene in the case of risk. Most of the original equipment manufacturers (OEMs) currently offer this system as standard equipment in most of the vehicles and therefore they belong to this level.

**Level 2**, unlike Level 1, covers vehicles that have an ADS capable of performing the DDT tasks of longitudinal and lateral control simultaneously. Therefore, in these cases the vehicle is able to accelerate and brake, stay in the lane and take any turn. At this level, as level 1, the conventional driver is expected to perform the remainder of the DDT, monitor the performance of the autonomous system's features, and supervise the ADS to achieve a *minimal risk condition* as needed. The ODD in this level is limited, so vehicles in this level can only operate with ADS active under certain conditions. Currently, there are in the market some vehicles that belong to this level such as the new *Renault Clio* and *Renault Captur*, the *Nissan* vehicles with the *Propilot* system, the *Mercedes-Benz E Class 2021,* or most *Tesla* vehicles with the *Autopilot* feature. All these vehicles recommend using this mode only on highways or roads in good condition since the system may fail in other conditions.

From **level 3** all sub-tasks of DDT can be controlled by ADS. This means that in contrast to Level 2, in Level 3, apart from the lateral and longitudinal control of the vehicle by the system, the system is also able to recognize any type of object and event that may occur in the driving environment (OEDR task). Therefore, the system is more intelligent. On the one hand, the vehicle can make strategic decisions that not only focus on maintaining speed and following the path of the lane but also on choosing the best route, overtaking slower vehicles if necessary, predicting the behaviour of the environment's vehicles by taking into account, for example, turn signals, and even detecting traffic lights and signals. On the other hand, it can determine when it is necessary to launch a *request to intervene* (if the ODD limits are going to be exceeded or if there is a failure of the ADS performance) to the DDT fallback-ready user. It is important to note that, while Level 3 vehicles may be designed to manage fallback and achieve a *minimal risk condition* in some circumstances (e.g. avoiding an obstacle), their ADS do not guarantee that a minimal risk condition will be automatically achieved in all cases even if the vehicle is operating in the ODD.

The main added value of **level 4** is the management by the system of accident risk conditions to achieve the minimum risk condition, that is, the performance of the DDT fallback task by the system. This system, in the case of any risk, warns the user with a *request to intervene*. If the user does not respond and does not start to perform the DTT, the system continues to have control and activates the DDT fallback subtask in order to achieve a *minimal risk condition*. The correct operation of all these features is only guaranteed in the areas determined by the ODD. In fact, according to the standard, if the user is not in these zones, the system should never allow him to activate it. Currently, there are vehicles from different companies that promise to have ADSs with features of this level. *Navya*, a French company, is already building and selling Level 4 electric shuttles, cabs, and trucks that can reach a top speed of 90km/h [19]. The use cases of these vehicles are oriented to airports, industrial areas, campuses... clear examples of ODD limitations. Today they promise to be able to

bring this technology to the cities. Other companies like *Google* with *Waymo* [20], *Manga* [21], and the alliance between *Volvo* and *Baidu* [22] are starting to develop prototype vehicles of this level.

Finally, **level 5** comprises all the characteristics of level 4 but without any use limitation, i.e. without ODD limitations. These vehicles will allow their occupants to be taken to any point on the planet (as long as it is viable to go by car), without the need at any time for a driver, just as an experienced driver would drive today. They are the future of transport. Although some articles promise the arrival of the level 5 autonomous vehicle in 2025 [23], these contradict themselves by saying that they will only be able to be used in specific areas (which means level 4 vehicles). A true level 5 vehicle has no restrictions and must be able to cope with any type of route that a conventional driver might take today.

In order to implement an autonomous vehicle, the company General Motors defines 3 main capabilities [24]:

1. **Perception**: It includes all the technologies needed to monitor any event that happens in the environment around the vehicle or to detect any object near it. This capability includes two very different capabilities, on the one hand external perception of dynamic and static objects, and on the other hand the, so called, ego localization (position, velocity, and others).
2. **Planning**: It determines the desired behaviour of the vehicle based on the data received by the detection technologies.
3. **Control**: It controls the longitudinal and transversal movement of the vehicle to follow the route marked by the planning technology.

The following figure shows the different technologies used in General Motors' autonomous prototype vehicles (Figure 5) to achieve a complete **perception** of everything that happens around the vehicle.

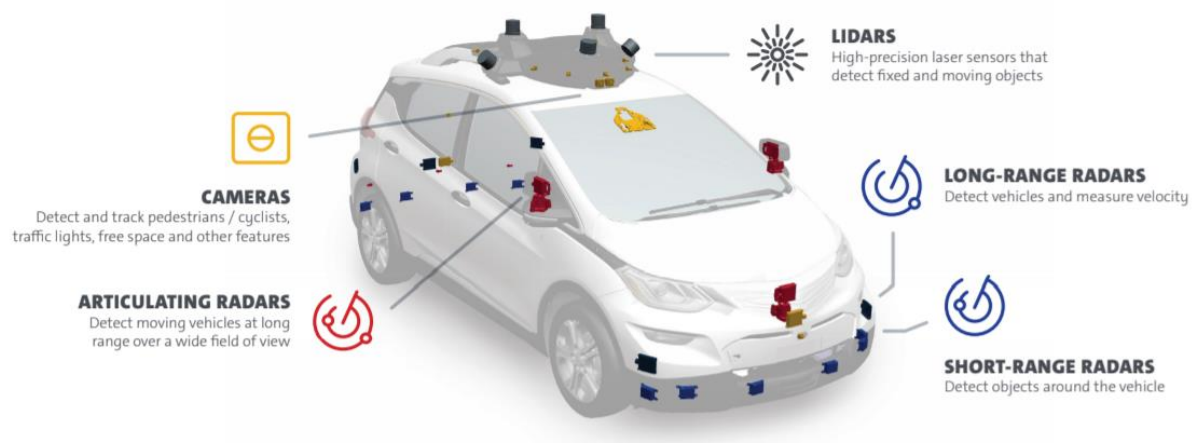



Figure 5. Detection technologies used in General Motors' autonomous vehicle prototypes [24].

Its prototypes currently have 5 LIDARs (Laser Imaging Detection and Ranging), 16 cameras, and 21 radars [24]. The combination of all the data they provide is used to monitor the environment and act upon it. In this way the system can detect and classify objects, the position of the vehicle, and its direction. These technologies allow the generation of a three-dimensional model of the environment. With all these technologies, not only is the perception system able to detect objects, but to predict their future movements. In this way, the execution of the above-mentioned OEDR task by the system is achieved. With this model, the **planning** software running on a vehicle computer calculates which route the vehicle should take and how it should proceed (lane change, acceleration, braking, turning, etc.) also in the case of risk. Finally, thanks to the electric motors coupled in the direction of the vehicle and the possibility of controlling the vehicle engine by this system (**control** technologies) the planned route is carried out.

Figure 6 shows the diversity and redundancy of the systems used in General Motors' autonomous vehicle prototypes. According to the figure, there is complete redundancy in the information given by all the sensors, the vehicle communications, the power supply of the systems and the detection of possible collisions. A series of redundant electric motors are used to control the braking and steering system. It is also seen that two large embedded computers as electronic control units (ECU) are used to manage all the sensor data and send commands to the actuators to control the vehicle. An ECU is a dedicated task computer that is installed in a vehicle to control various electrical subsystems of it. The main difference with a conventional computer, apart from being designed to perform a specific task and therefore have specific communication interfaces (such as CAN), is its robustness. They offer characteristics such as allowing temperatures between -40°C/+85°C or tolerance to vibrations and physical shocks. [25] Therefore, these computers, which make the operation of ADS possible, must be well protected.

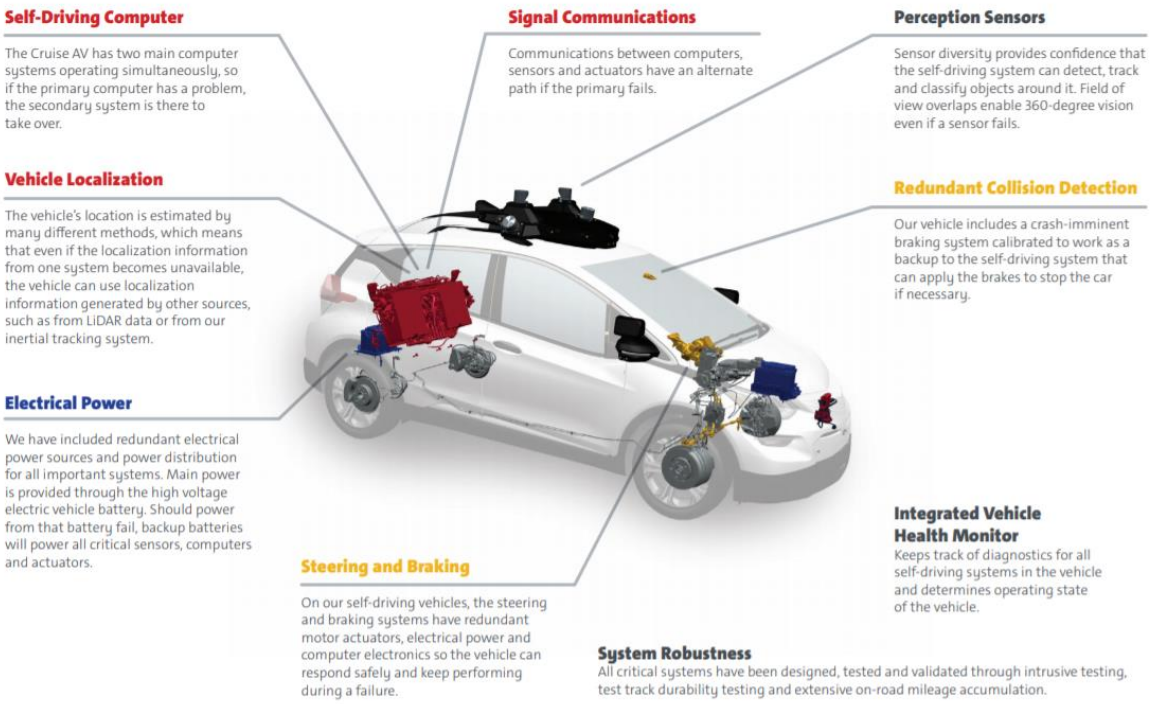



Figure 6. Systems diversity and redundancy at General Motors' autonomous vehicle prototypes [24].

Other companies such as *Tesla*, in their ADS-incorporated commercial vehicles, do not use LIDAR technology and, to carry out the perception of the environment, use only cameras and radars [26]. *Tesla* guarantees that the hardware mounted in the vehicles it currently sells is ready to reach level 5 of autonomy and that the only limitation falls on the software. They promise to release vehicle updates as they develop new versions of their *Autopilot* system so that, gradually, those vehicles already purchased, acquire greater autonomy capabilities [26].

As we have seen, perception systems are very relevant when developing an autonomous vehicle. Next a review of the state of the art of the technologies currently in use and their different purposes will be shown. To perform the **external perception of dynamic and static objects** 4 technologies are commonly used:

- **Cameras**: Used in conjunction with artificial intelligence software capable of interpreting the images they make it possible to recognize and classify objects in the environment. Furthermore, by using two separate cameras pointing in the same direction (called stereo-camera), depth estimation can be achieved. The cameras can be classified according to the following metrics: resolution of the image provided; field of view, much field of view implies

the need to have a higher image resolution if the detection of small and/or distant objects is required; and dynamic range, a poor dynamic range can difficult the detection in environments with a lot of lighting contrast such as shadows. The main disadvantage of cameras is the need for a powerful computer that is capable of processing images in real time. In addition, these systems are not reliable in bad weather conditions that produce low visibility, such as rain, snow, or fog.

- **LIDAR** (Light Detection And Ranging sensor): Allows to estimate a 3D scene geometry through a laser light and measuring the reflection with a sensor, taking into account delay and intensity. They usually include spinning elements with multiple light sources so that more information can be collected. These systems can be classified according to certain metrics: number of light sources and sensors; point rate per second; rotation rate, if spinning elements are incorporated; detection range, that is, limit distance that can be covered; and viewing angle. LIDARs are more accurate than cameras in depth estimation as they cannot be fooled by shadows, sunlight, or the headlights of other vehicles. They also allow to save computer power. LIDARs give immediate absolute information about the distance to a point, while camera-based systems need to process and interpret the images before a reliable result can be obtained. However, as with cameras, LIDARs are unreliable in bad weather conditions such as snow and rain.
- **RADAR** (Radio Detection And Ranging sensor): It uses a method similar to LIDAR but employing radio waves instead of light. It allows object detection and motion estimation They can be classified according to certain metrics: detection range, that is, the maximum distance at which it can detect; field of view; and accuracy of position and speed. Generally, if the radar provides a long detection range, it presents a smaller field of view and vice versa. RADAR technology provides less accuracy and flexibility than LIDAR. However, in bad weather conditions it provides better results, and it does not need light to operate.
- **Ultrasonic sonar**: Through the emission and detection of ultrasounds it is able to measure the distance from the sensor to the target object. They are very useful for detecting close objects but not for distant objects. These devices can be classified depending on the detection range and field of view. Like radars they offer good results even in bad weather and do not need light to operate.

On the other hand, to carry out the **internal perception of the vehicle**, called **ego localization**, a series of sensors are commonly used:

- **GNSS** (Global Navigation Satellite Systems) and **IMU** (Inertial Measurement Units): GNSS allows the vehicle system to know its position and estimated speed in real time. The accuracy of these systems depends on the technology used. On the other hand, IMU calculates the accelerations and the angular rate of rotation of the vehicle.
- **Wheel odometry**: It allows to know the speed at which the wheels of the vehicle turn and therefore the speed of the vehicle.

Finally, a **self-driving brain** is needed to carry out all the tasks of the ADS. This takes in all sensor data and, depending on them, operates the vehicle actuators necessary to carry out the autonomous driving tasks.

## 2.2. Automatic Guided Vehicles

Automatic guided vehicles (AGVs) are robots capable of following pre-set routes autonomously. Their main objective is to help in the industry sector. They are always electric and can incorporate manual or automatic systems for recharging or replacing the battery. They are generally used in factories and warehouses to transport materials performing slow and repetitive operations [27].

However, today their use is expanding, and they can be seen in some hospitals (transporting necessary material), and even theme parks.

To follow the pre-set routes, multiple technologies can be used by these robots. The following paragraphs provide a brief review of the main technologies commonly used in this sector.

- **Radio Frequency Guidance**: A wire is buried along the entire route that the robot must follow. This cable is connected to an AC generator, so it emits magnetic fields at a certain frequency. The robot must have a sensor to detect the magnetic field from the cable and be able to determine where it is. This way, it can follow the cable. If the robot reaches a point where the cable splits, and therefore there are two possible routes, there is a method for the robot to differentiate the magnetic field of both cables and, depending on the configuration, choose to follow one or the other. This method consists of using different frequencies in each cable. The sensor of the robot is able to detect and differentiate the magnetic field of the cables at different frequencies. In this way, the pre-programmed vehicle knows where it should go. This possibility brings a lot of flexibility to the system as it can be used not only to differentiate routes, but also to mark areas where the vehicle should behave in a certain way (e.g. changing speed) [27].
- **Magnetic tape**: Through magnetic tape, routes are created that define the robot's path. It incorporates a magnetic sensor that allows it to know where the tape is in real-time. The magnetic tape can be superficial or buried under the ground at a short distance from the surface so that it can be detected by the robot's sensor. In the case there is a point where the tape splits, as in the previous technology there is a method to get the robot to follow the correct path. The magnetic sensor of the robot is able to detect multiple tapes simultaneously, however, it is necessary to be able to inform it of where it is so that, depending on the path it has to take, it follows one track or another. The method consists of adding pieces of tape with different combinations of polarities, sequences, and length at the sides along the path (Figure 7). Thus, the robot gets information about where it is and how it should act [28] [27].

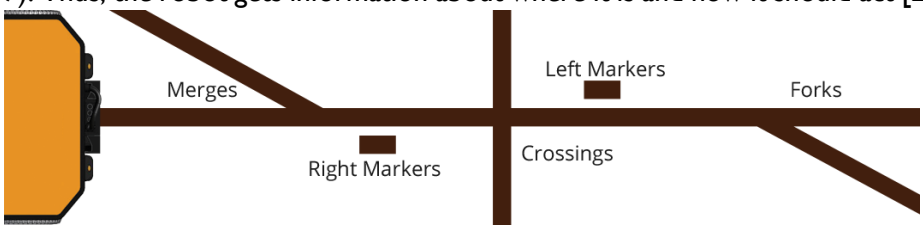



Figure 7. Example of the possibilities of the magnetic tape [28].

- **Coloured tape**: Similar to the magnetic tape. In this case a coloured tape is used which is detected by the robot through an optical sensor. Therefore, this tape must be on the surface. In order to allow the robot to make decisions during its route, different pieces of tape can be added to the sides along the path, similar to the magnetic tape [28].
- **Pulsed or modulated laser**: To use this technology to guide the vehicle, it is necessary to mount a reflective strip on all the walls of the environment where the robot will operate. This must incorporate a laser transmitter and receiver that constantly rotates and must have a preloaded map of the environment in which it will work. The rotating laser transmitter and receiver sends information to the robot about the angles at which it has detected the reflection of the laser and those at which it has not. In this way, thanks to the robot's preloaded map of the site, it can determine where it is located [29] [27].
- **Vision guidance**: It consists of the mounting of a series of cameras on the robot which, through artificial intelligence, creates a 3-dimensional mapping of the environment. In this way, it is able to know its position and, therefore, it can follow pre-established routes, and change its behaviour depending on its location. This technology makes it possible not to

modify the environment or add new elements to it. However, achieving a high level of accuracy can be complicated in certain environments [30].

In AGVs, the use of these technologies is often complemented by others. Radio frequency guidance technologies or magnetic or coloured band guidance are usually complemented with RFID (Radio Frequency Identification) technology to allow the robot to know where it is on the path and how it should behave.

**RFID** is a system for the transmission of identifiers from a device (active or passive), called an RFID tag, to another device, called an RFID sensor, at relatively short distances. Generally, RFID tags carry a unique identifier that allows them to be unequivocally identified. Some tags allow not only to be read but also to be written, being able to overwrite their identifiers by other more convenient ones [31]. The maximum communication distance depends on the type of tags (active or passive) and the frequency range in which they operate. Active tags have a power supply and since they have such power, their identifier can be read up to 100 meters away. On the other hand, in order to transmit their identifier, passive tags need to be charged by the RFID sensor which wants to read it. Since they acquire power in this way, they cannot be placed as far away from the RFID sensor. Also, due to the characteristics of the system they have limited power. In summary, it is possible to read these tags at a maximum distance of 25 meters. Passive tags usually operate in 3 different frequency ranges [31]:

- Low Frequency (LF): 125 -134 kHz
- High Frequency (HF): 13.56 MHz
- Ultra-High Frequency (UHF): 856 MHz to 960 MHz

This technology can be helpful for AGVs. Instead of using tape strips or different frequencies to differentiate specific areas, RFID tags can be used along the path. These tags can be located at decision points, where the robot must decide to follow one or another path. They can also be placed at different points of the route so that the robot knows where it is and therefore can make changes in its behaviour. The robot would contain a memory with the identifiers of those tags to know where they are and what to do according to the identifier it reads.

## 2.3. Conclusions for the Project and selected technologies

In this section, once the main concepts and technologies of the autonomous vehicle and the AGVs have been studied, a discussion will be carried out about which systems are the most recommendable to achieve a level 4 autonomous vehicle for the automated parking of the *TwizyLine* project. As seen above, a level 4 autonomous vehicle can carry out all the tasks of a conventional driver (DDT and DDT fallback tasks) autonomously under specific conditions (ODD) and, therefore, without the need for a driver. The following subsections show how these concepts correspond to the system to be designed.

### 2.3.1. ODD

The *TwizyLine* service car park is like the one shown in Figure 8. It can be divided into 3 different zones: Leaving Zone, where users leave the vehicle and, once they have left, the vehicle goes into autonomous mode; Park Zone, where vehicles operate in autonomous mode to be parked and unparked; and Pick-Up Zone, where the vehicle switches off the autonomous mode to be picked up by a user. Note that the Pick-up Zone of all the car parks includes a barrier that only opens if the system detects that a vehicle is closer than 15 metres to that barrier. More car park designs are available for the *TwizyLine* project. They can be seen in the final degree project by Samuel Pilar Arnanz [13]. Although they have differences, they are all divided into the same areas and have the same characteristics relevant to autonomous driving inside them.

As we have seen, the vehicles should only engage their ADS inside the car parks. These car parks are controlled environments closed to the public where there are only other autonomous cars and, occasionally, qualified maintenance personnel. These are the only possible obstacles inside the car park. However, the possibility that the vehicles may encounter some other obstacle ahead not initially contemplated is also considered. All vehicles will move inside at a speed of between 5 and 10 km/h for either parking or unparking. In this area, due to the vehicle guidance characteristics that will be shown later, there is no risk of lateral collision. Therefore, there is no possibility of one vehicle colliding with the lateral zone of another.

This car park corresponds to the operational design domain (**ODD**) of the system.

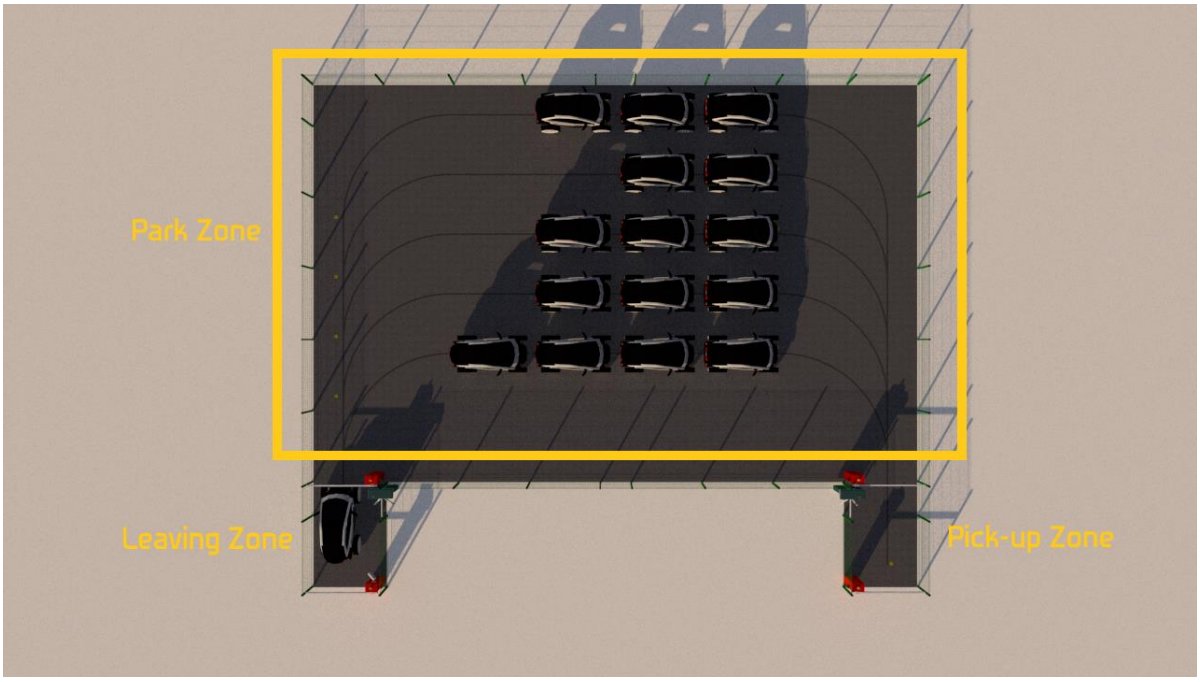



Figure 8. Small car park of the *TwizyLine* service seen from above.

### 2.3.2. Sustained longitudinal and lateral control

In order to implement the longitudinal and lateral control, we have look for solutions in the literature and we have used one of these solutions depending on the requirements of our project and other factors.

The **longitudinal control** of the vehicle can be carried out in several ways. In current ADS systems, in order to carry out the acceleration control, the information given by the pressure sensor on the accelerator pedal is ignored and other similar messages from the ADS are sent to the engine to simulate the action of the accelerator pedal. For brake control, the Electronically Controlled Brake system of modern vehicles is used [32]. In order to have control of the accelerator, a bypass will be performed on it, and its analog signals will be simulated. To carry out the control of the brake, two main possibilities are considered. On the one hand, it is possible to use the system that was used by a research group at the University of Cartagena for the conversion of a *Twizy* vehicle into an autonomous vehicle [33]. This consists of an electric motor coupled to a cam that operates directly on the brake pedal (Figure 9). On the other hand, the hydraulic braking system of the vehicle can be modified to incorporate an Electronically Controlled Brake system and to control it through messages from the ADS [32]. This is necessary because the *Twizy* model does not incorporate any electronic braking system, not even ABS [34].

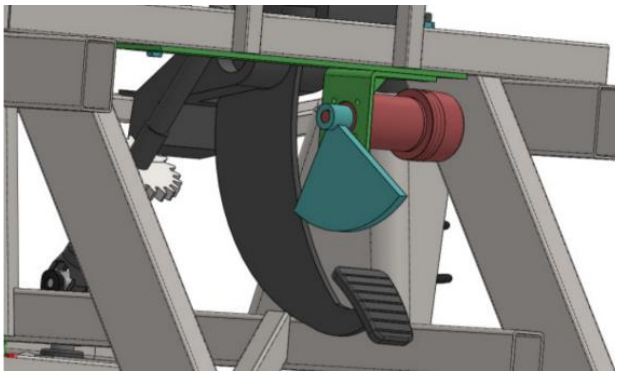


Figure 9. Control of the vehicle's braking system using an electric motor [33].

At present, it has not been decided which braking method will be used. This will be decided in the next main objective of the project shown in section 1.2, which is the main future line of the project.

We have found 2 main alternatives for carrying out the **lateral control** of the vehicle. The first consists in replacing the steering column of the Twizy with that of a Renault Clio II as was done in the thesis of Christopher Dunkel [35]. The difference between this and the original steering column is that the Clio has a power steering engine, therefore messages can be sent to it to move the steering wheel when necessary. The other alternative is to use the method used by the research group at the University of Cartagena [33]. This consists of coupling gears that connect to an electric motor on the current steering column of the Twizy.

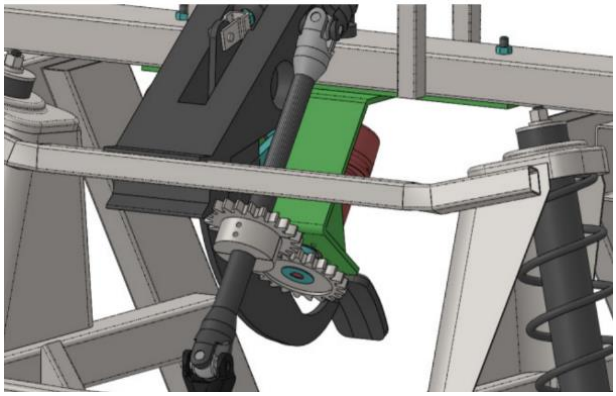


Figure 10. Control of the vehicle's steering wheel using an electric motor [33].

As can be seen in the following table (Table 6) the replacement of the steering column is a very complex task. Furthermore, there is no public information about how to control the power steering of the Clio II model. Therefore, it was decided to use the second method.

| *Method* | *Assembly complexity* | *Use complexity* |
|---|---|---|
| Changing the steering column | Requires disassembly and assembly of the entire steering column | No public information about the control of the power steering of a vehicle. |
| Adding electric motor | Requires assembly of new parts on the vehicle | There are motors with controllers on the market with clear user interfaces. For example Maxon motors [36]. |

Table 6. Methods for the control of the vehicle's direction.

### 2.3.3. OEDR

The object and event detection and response (**OEDR**) task includes the detection of the events that the vehicle must face when its ADS is working and the calculation of the necessary procedures to act

correctly according to the events detected. To explain how the vehicle should perform this task, we will focus on the Perception and Planning tasks explained in section 2.1.

#### 2.3.3.1. Perception

To carry out the **external perception** tasks, a series of sensors are needed.

The aim of this project is to achieve a vehicle that is guided by systems analogous to those of the AGVs. In view of the variety of technologies that currently exist (discussed above), a comparison of these is made and can be seen in the following table (Table 7).

| *Method* | *Installation of the system in the parking* | *Programming complexity* | *Reliability* | *Flexibility* |
|---|---|---|---|---|
| Radio Frequency Guidance | Requires buried cable installation connected to an AC generator plugged into the power grid | No products were found on the market with public available manuals for the use of this technology. | Reliable in the presence of dirt or rain<br>Buried [28] | Difficult to lay and difficult to change. |
| Magnetic tape | Allows simple surface mounting | Easy to program<br>For example: Roboteq magnetic guide systems [28]. | Reliable in the presence of dirt or rain [28]<br>It allows for burying to preserve its integrity over time | Easy to lay and easy to change for testing - Allows for burying in the case of final installation to preserve the integrity of the tape |
| Coloured tape | Allows simple surface mounting | Easy to program<br>For example: Roboteq optical guide systems (discontinued) [28]. | No reliable in the presence of dirt or rain<br>No possibility of burying [28] | Easy to lay and easy to change for testing |
| Pulsed or modulated laser | Requires installation of reflectors on all edges of the environment | Difficult to program<br>It is necessary to be able to interpret the data in order to create a site map | No reliable in the presence of dirt or rain | The system must be reprogrammed if the parking space is changed |
| Vision guidance | No installation required in the environment | Difficult to program<br>Works with artificial intelligence that must be prepared for the environment | No reliable in the presence of dirt or rain | The system must be reprogrammed if the parking space is changed |

Table 7. Comparison between the main current AGV guiding technologies.

As shown in Table 7, the best option for our project is the use of the magnetic strip. This is easy to install and change and can be buried, so it provides a lot of flexibility. It is not affected by dirt or rain due to its characteristics, and in addition, it can be buried to prevent degradation over time. Furthermore, there are currently companies on the market such as *Roboteq* [28] that offer this type of sensor with a manual that explains its interface in detail. With a good design of routes made with the magnetic strip, avoiding crossings, and maintaining sufficient distance between each magnetic strip, the risk of lateral collision of a vehicle with another is eliminated. This sensor will allow a correct lateral control of the vehicle.

Regarding the ODD of the system (subsection 2.3.1), the vehicle with the ADS activated inside the car park, must move at very low speeds (between 5 and 10 km/h) and must be prepared to act if it detects a vehicle or any other obstacle in front of it. Remember that it is only necessary to detect events in the front of the vehicle as there is no risk of lateral collisions. Therefore, the vehicle must be able to adapt its speed if there are other vehicles or obstacles in front of it and to stop if necessary. This does not require a long detection range since at such low speeds the braking distance is minimal and a wide anticipation is not necessary. There are multiple detection technologies seen in section

2.1 to accomplish this. The following table (Table 8) shows a comparison of these technologies to be used in the project.

| *Technology* | *Range detection* | *Computing workload* | *Weather conditions* | *Price* | *Utility* |
|---|---|---|---|---|---|
| **Cameras with artificial intelligence** | Varies according to resolution and field of view. | It needs a powerful computer, trained to detect different events. | Unreliable in bad weather. | Very variable depending on the quality required. | Recognize, classify, and locate objects in the environment and predict their future movements. Need two cameras to estimate depth. |
| **LIDAR** | Up to 200 meters [37]. | It needs specific connections and hardware and software to receive its information. | Unreliable in bad weather. | Expensive. For example [38]. | Very precise data about the distance to objects. Certain data is unnecessary for the project. |
| **RADAR** | Depends on the type. There can be obtained ranges of up to 200m with very little field of view or much shorter ranges with a lot of field of view [37]. | It needs specific connections and hardware and software to receive its information. | Reliable in bad weather. | Not as expensive as LIDAR. | Precise data about the distance to an object. A short-range radar with a wide field of view is good for the project. |
| **Ultrasonic sonar** | Some can detect up to 8 meters away. | Easy to use. They return only the distance of the object they are pointing at. | Reliable in bad weather. | Cheap. For example [39]. | Accurate data on distance to nearby objects and vehicles. |

Table 8. Comparison between the detection technologies currently used in ADS.

As shown in Table 8, the best options for this project are the use of ultrasonic sonars. The use of cameras is discarded because they imply a high computational workload, they may not work well in bad weather conditions, and at least two cameras are needed to be able to estimate the distance to an object. The use of LIDAR is discarded because of its price and because it provides unnecessary data for this project. On the other hand a short-range RADAR or ultrasonic sonars can be very effective. However, radars are more expensive and complicate the design of the vehicle due to their operation and interconnection. This is why it was decided to use ultrasonic sonars as they are easy to use, cheap, and perfect for the system ODD. The use of this technology allows the vehicle to detect objects and vehicles in front of it at low speeds and adapt its speed or even stop if necessary. This technology allows a correct longitudinal control of the vehicle.

If an obstacle or vehicle is detected, the vehicle will adapt its speed to it. If the detected vehicle or obstacle remains stationary, the vehicle will reduce its speed and stop 30 cm away from it. If two ultrasonic sensors are used on the front of the vehicle, they must have a minimum range of 2.5 metres and a lateral range of 1.2 metres. In Figure 11 we see that, with these measures, the vehicle would be able to detect vehicles close even in tight curves.

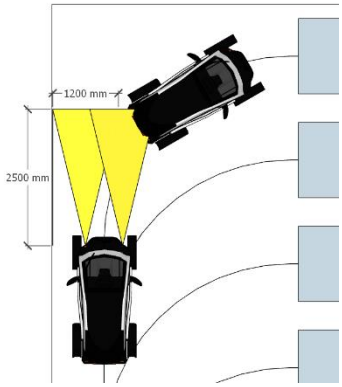



Figure 11. Example of the detection of a vehicle with ultrasonic sensors.

To carry out the **internal perception** tasks, the system must be able to access to some information.

The vehicle's ADS must always be able to know the speed of the vehicle. This will allow it to control the actuators of the vehicle's longitudinal control in order to achieve a speed of between 5 and 10 km/h, as required by the system's ODD (section 2.3.1). Currently all vehicles incorporate an odometer in the wheels to know their speed and to count the distance travelled by the vehicle.

The vehicle needs to know its location. If we remember the ODD of the system, the entry barrier only opens if the system detects that the vehicle is less than 15 meters from it. Therefore, it is necessary to know the location of the vehicle with an accuracy of less than 7 meters approximately. There are multiple technologies that offer valid location accuracies for the system. GPS (Global Positioning System) technology allows to obtain accuracies of up to 3 meters provided that a correct GPS receiver is used [40]. On the other hand, to know its location inside the car park, the RFID technology explained in section 2.2 is used. A collection of passive RFID tags will be placed at different points in the car park. The vehicle needs to have an RFID antenna to be able to read them and thus know its position inside the car park.

#### 2.3.3.2. Planning

The planning task determines the desired behaviour of the vehicle. It can be divided into two: long-term planning and short-term planning.

In the TwizyLine project, the **long-term planning** is carried out by the back-end. The back-end is informed about the position of each vehicle at all times, determines the routes the vehicles must take, and sends a message to the vehicles instructing them to follow it. This route decision comprises the long-term planning task of the system. The back-end will order the vehicles to follow one or another route depending on their battery charge and the number of vehicles parked in the car park. The final project degree by Samuel Pilar Arnanz [13] explains the operation of the back-end in depth. The RFID technology explained in section 2.2 is used to define these routes. This way the back-end define the route by informing the vehicle about what it should do when it detects an RFID (turn or vary speed). In addition, the back-end decides when the vehicle should go into autonomous mode or manual mode. Therefore, it is the dispatching entity that decides when to dispatch the vehicle in driverless operation.

The **short-term planning** is carried out by the vehicle. Thanks to the information perceived by the sensors mentioned above (subsection 2.3.3.1), and the route designated by the back-end, the vehicle decides if it should vary the speed or stop, and when it should turn.

### 2.3.4. DDT Fallback

The vehicle, in the case of any risk, will immediately go into error mode. In this mode, the vehicle stops immediately, reaching the minimum risk condition. It then informs the back-end of what has

happened. In this way, the back-end, which has information about the location and status of all the vehicles, can decide when the vehicle should be restarted and, at the same time, can send other routes to the other vehicles operating autonomously in the car park if necessary. This operation forms the **DDT fallback task**.

One of the future lines of this final degree project consists of carrying out a risk analysis, analysing each possible failure in depth. Annex II shows the errors that have been considered until now when implementing the actual system. However, these errors are only for the implemented part, i.e. the communication tasks. During the execution of the next project objective, the table will be completed with all possible error cases.

### 2.3.5. Self-driving brain

It is necessary to debate which central device, the system's "brain", will be used. This computer, as seen in section 2.1, must be able to take in all sensor data and, based on this, operate the vehicle's actuators to carry out all the tasks of the ADS. In this way, at the beginning we considered installing an ECU that we would program to process the data. However, they are proprietary solutions which provide in general little flexibility as they are closed environments and consequently, the idea is discarded. One cheaper alternative is the use of Single Board Computers (SBC) based on ARM (Advance RISC Machine) architecture, like for example the well-known *Raspberry*. These low-cost SBC can operate as an ECU inside the vehicle, provided that a good selection of the necessary characteristics is made. It is possible to install many different Linux distributions in those SBC, what will give flexibility. Moreover, there are hardware modules to add mobile network connectivity and geolocation to them.

For the programming of the software in this module, the *Python 3* programming language will be used. It facilitates fast programming by allowing the developer to focus on the algorithms and disregard other less relevant issues. It allows quick and easy management of subprocesses and threads and is compatible with many libraries such as paho-mqtt, python-can or can-isotp. These libraries are very important in this project since the communication through CAN and MQTT is fundamental. Besides, the can-isotp library is exclusive for *Python 3*. Furthermore, the knowledge about this language before starting this final degree project was very high so the choice of this programming language allows to save time.

# 3. System design and ECUs implementation

This chapter explains how the first main objective of this final degree project shown in section 1.2 has been achieved. It is complemented by chapter 5 which shows an evaluation of the system's use cases. It shows two main issues. On the one hand, the design of the system, i.e. which devices will be used and how they will be interconnected inside the vehicle. On the other hand, the whole implementation of the ECUs is shown, its states, the CAN messaging used, the error management, the operation of their processes, synchronization between both, etc. Therefore, this chapter corresponds to phase 2 of this final degree project shown in section 1.3.

## 3.1. System design

This section explains the general design of the system that will be implemented in the vehicle. It is important to note that certain sections, such as the control of the vehicle, are not fully defined as, due to the global COVID-19 pandemic, the delivery of the real Twizy vehicle to the university has been delayed and there are certain parameters that need to be checked in the real vehicle. Therefore, these parts will be defined in objective 4 of the *TwizyLine* project, as explained in section 1.1, so they are not part of this final degree project.

### 3.1.1. Processing

As shown in section 2.3, *Single Board Computers* (SBC) based on ARM (*Advance RISC Machine*) architecture will be used to implement the necessary processing tasks in the vehicle. At present there are multiple alternatives on the market. Table 9 shows a comparison of those studied in terms of the main characteristics that an ECU must have, as seen in section 2.1. It compares its robustness, incorporation of CAN interface (indispensable in an ECU to be able to receive and send information to the rest of the components of the vehicle), the flexibility in its source of power (the electrical source of a vehicle can have wide variations of voltage during its use), and its processing speed and RAM memory.

| *Device* | *Robustness* | *CAN interface* | *Power source flexibility* | *Processing speed and RAM* |
|---|---|---|---|---|
| Raspberry Pi 4 B **[41]** | Optional plastic case<br>It is necessary to buy a metal case separately<br>No guarantees | Not included<br>Necessary to buy separate module | 5V only | 1.5 GHz<br>Up to 4GB DDR4 RAM |
| Raspberry Pi 3 B+ **[42]** | Optional plastic case<br>It is necessary to buy a metal case separately<br>No guarantees | Not included<br>Necessary to buy separate module | 5V only | 1.4 GHz<br>1GB DDR2 RAM |
| Humming Board CBi **[14]** | Extruded aluminium (IP32) enclosure<br>Operating temperature: -40°C to 85°C<br>Guaranteed | Included<br>Speed up to 1Mb/s | From 7V to 36V | 1 GHz<br>Up to 2GB DDR3 |
| RevPi Core **[43]** | Operating temperature: -40°C to 55°C<br>EMI tests: according to EN 61131-2<br>Guaranteed | Not included<br>Necessary to buy separate module | From 10.2V to 28.8V<br>ESD protection | 1.2 GHz<br>1 GB DDR3 RAM |
| Strato Pi CAN board mounted on RPi **[44]** | It is necessary to buy a separate case<br>Not guaranteed | Included<br>Speed up to 1Mb/s | From 9V to 65V<br>Surge and reverse polarity protection | 1.5 GHz<br>Up to 4GB DDR4 RAM |

Table 9. Comparison between the embedded systems contemplated to implement an ECU.

As shown in Table 9 the most recommended options are the *Humming Board Cbi* and *the Strato Pi CAN Board* mounted on a *Raspberry Pi*. The problem with the second option is that it is necessary to buy a *Raspberry Pi* and then attach the CAN Board to it. In addition, after doing this, it would be necessary to buy a separate case which generally does not provide any guarantees. On the other hand, the *Humming Board CBi* has a slightly limited processing speed. However, as it will be seen later, there is no intention of developing resource-intensive software. Furthermore, as it will be seen below, there are two well-differentiated groups of tasks when it comes to implementing this system. Therefore, it is decided to use two ECUs instead of one. This makes it much more flexible as it is a more modular system and, as the load is distributed between both, the risk of lack of processing power is lower. Figure 12 shows a picture of a *Humming Board CBi* like the ones used in this project.

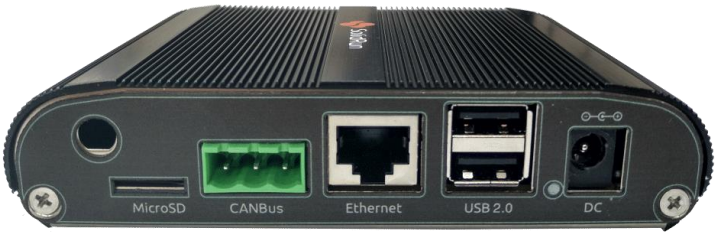



Figure 12. A *Humming Board CBi* [14].

Hence two *Humming Board CBi* are going to be used as ECUs in the vehicle. These will have the Debian 10.0 Linux distribution installed. This distribution comes pre-configured with a set of repositories that allows easy installation of most of the software available for Linux. This model incorporates a CAN interface, an RS 485 interface, 4 USB ports, an RJ45 port and 20 general purpose pins [14]. Figure 13 shows all its interfaces.

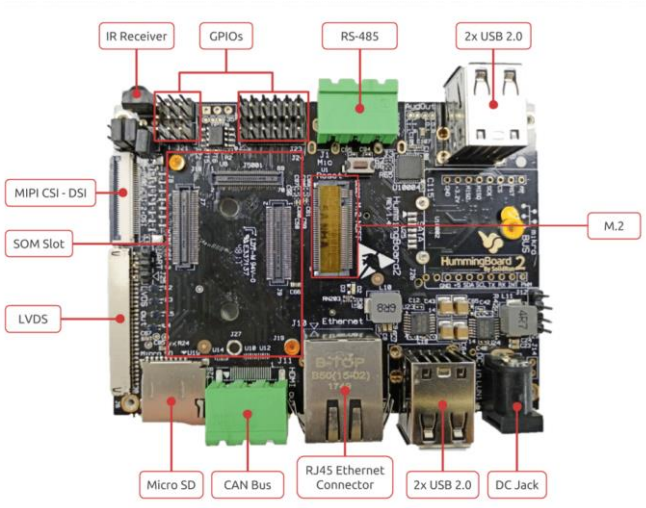



Figure 13. Interfaces available on the *Humming Board CBi* [14].

They must interpret the information from all the sensors, control the actuators installed in the vehicle to control it, check the status of the vehicle and manage the communications of the vehicle with the MQTT server, receiving orders and sending information. These tasks are distributed between the two ECUs. The first is called the **Communication module** and the second the **Control module**. They must have a permanent communication channel between them, which allows not only the transmission of orders and information between them, but also the sending and receiving of error messages in the case of failure in order to achieve full synchronization between them.

The Communication module must manage the GPS signal reception, the vehicle battery status, the communication with the MQTT server (sending information and receiving orders) and control and

monitor the Control module. On the other hand, the Control module must take care of all tasks related with the control of the vehicle, that is, interpreting the information from the sensors and from the car (such as speed) and controlling the installed actuators so that the vehicle is able to follow the marked route.

As we have just seen, both ECUs need information from the vehicle. The Control module needs at least vehicle speed information and the Communication module needs battery status information. On the *Twizy* vehicle this information can be found on the well-known and very used in automotive industry Controller Area Network (CAN) bus [45]. Therefore, both ECUs must be connected to this bus through the CAN interface provided by the *Humming Board CBi*. This bus can also be used to carry out the communication between both ECUs, by adding a new collection of messages. In this way, with the same connection we can obtain vehicle information and interconnect both ECUs. Figure 14 shows a summary of the tasks carried out by the modules and their interconnection.

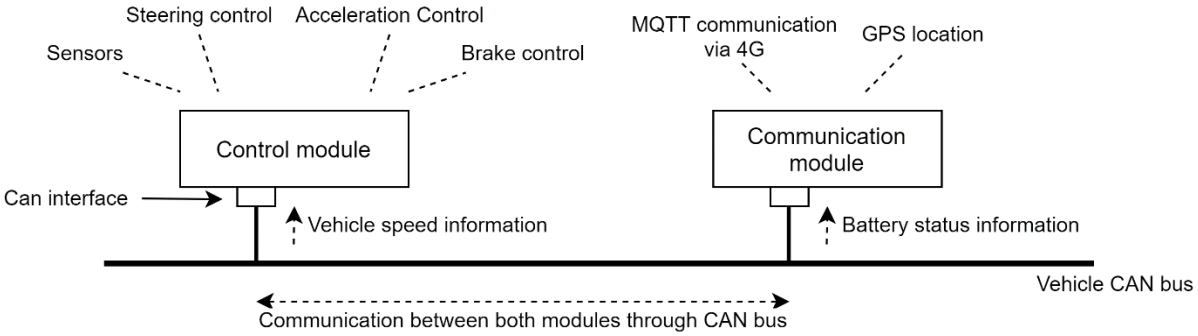



Figure 14. Tasks of the modules and interconnection.

In summary, the Communication module will maintain a constant connection with the server, informing it about the GPS position, the battery status, and any failures that may occur. It will interpret the commands it receives from the server and will control the Control module. The Control module will carry out the control of the vehicle according to the orders it receives from the Communication module and will inform it of its status and of possible failures that may occur in its tasks.

### 3.1.2. Connectivity

In order for the autonomous system to know the position of the vehicle in real time, as explained in section 2.3.3.1, it is necessary to install a GPS receiver with an accuracy of less than 7 meters. This must be connected to the Communication module. By means of one of the USB ports incorporated in the *Humming Board CBi*, a GPS receiver can be connected to carry out this task. The GPS receiver to be used is the Garmin *GPS18x USB* [15]. It incorporates differential DGPS capability using real-time WAAS (*Wide Area Augmentation System*) corrections yielding position accuracy of less than 3 meters, valid for the requirements of the project. It is waterproof and uses Garmin's private binary protocol to transmit information via an USB interface [15]. Due to its precision and robustness it is a good choice for the project. Figure 15 shows an image of this GPS receiver.


Figure 15. Image of the Garmin GPS18x USB [15].

The Communication module must have an internet connection to be able to connect to the back-end. To achieve this, it is necessary to connect a modem through another USB port of the *Humming Board CBi*. The system will not need high transmission rates, so it is not necessary to use high-bandwidth technologies such as 4G. However, it is interesting to use a modem that is compatible with many wireless technologies so that it can be used in most areas and the probability of not getting coverage is as low as possible. There are multiple USB modems on the market. The 4G Huawei E3372 modem has been chosen because of the experience that this modem works well with a Linux kernel and its high compatibility with different wireless technologies. It is compatible with 4G, LTE, UMTS, HSUPA, 2G, FDD, and GSM networks. Through 4G it can support up to 150Mbps [16]. Figure 16 shows an image of this 4G modem.


Figure 16. Image of the 4G Huawei E3372 modem [16].

In order to test the correct operation of the vehicle, the GPIO pins of the Communication module will be connected to a series of LEDs that will light up to give certain information. This collection of LEDs will be called the information board.

### 3.1.3. Sensors

As seen in section 2.3.3, proximity sensors are needed to prevent crashes and to adapt the speed of the vehicle to its environment. The company Maxbotix distributes a multitude of proximity sensors with different characteristics that work through the RS 232 protocol (*ParkSonar-EZ Sensor series*) [46]. This sensor series comprises the sensors from MB1001 to MB1009. The difference between these is the detection area they cover. Remember that in section 2.3.3.1 it is explained that if two sensors are used, they must have a minimum range of 2.5 meters and a lateral range of 1.2 meters. The sensor chosen for the project is the MB1009 (EZ-144) since it is the one that offers more longitudinal and lateral range. Figure 17 shows the range specifications of this sensor. These specifications define the range of the sensor according to the size of the object to be detected. As the aim of the proximity sensors is to detect vehicles in front of it, we can follow the specifications of the C and D drawings in the figure. Therefore, these sensors will be able to detect vehicles or obstacles that are 3 meters

away in a lateral range of 1.7 meters from side to side of the sensor, enough for the project according to the criteria discussed above. Hence, with two of these sensors installed on the sides of the front of the vehicle, we will have full coverage.

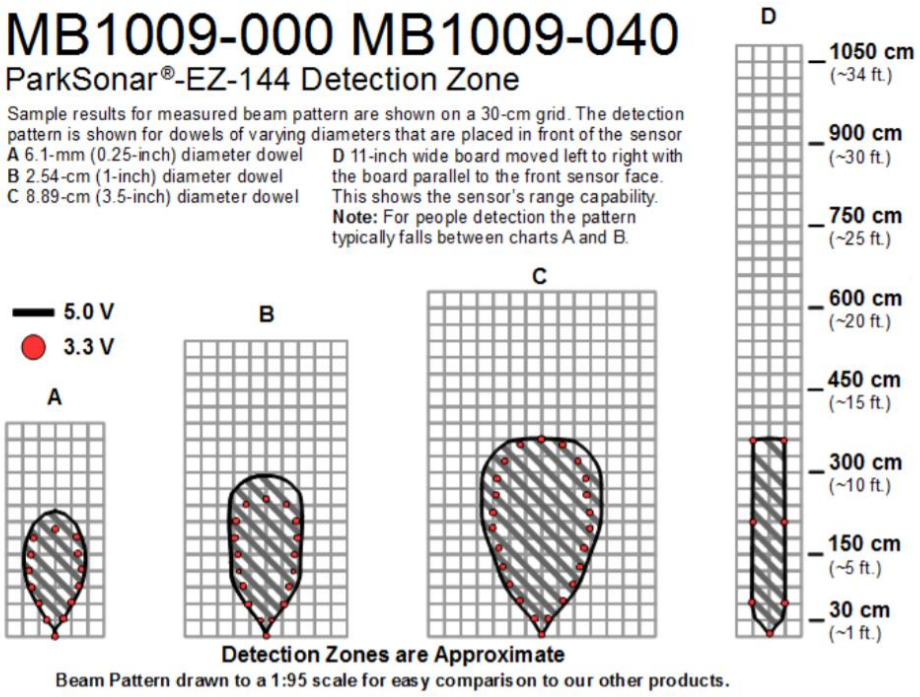
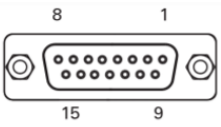



Figure 17. EZ-144 Sensor Range Specifications [46].

The *Humming Board CBi* does not incorporate as many RS232 interfaces for these sensors but it incorporates several USB interfaces. A conventional USB-RS232 adapter solves this problem.

The technology chosen to guide the vehicle, as seen in section 2.3, is magnetic tape guidance. The company *RoboteQ* sells different magnet sensors especially designed for line-guided systems. They have millimetric precision and are prepared to detect line merges and forks, magnetic side markers, and line crossings (see Figure 7). It is programmable through a software available on their website. With this software it is possible to watch and modify multiple parameters. They work with magnetic bands of 25mm or 50mm wide, sold by the same company. The sensor comes with CAN interface, RS232 serial, USB, analogic output and PWM so there are several possibilities to connect it to the *Humming Board CBi*. They also come with 2 meters wires, with all the necessary connectors, which is very suitable for easing the installation in the vehicle and connection to the ECU. It works with voltages between 4,5 and 30 volts, also convenient for installation in the vehicle [47]. The following figure (Figure 18) shows the pinout of the device.




| Wire color | Signal | Type | DSub pin | Description |
|---|---|---|---|---|
| braid | Ground | Power | 5 | Ground |
| black | Ground | Power | 5 | Ground |
| red | Power In | Power | 14 | 4.5V to 30V DC Power supply input |
| yellow + black | Power Control | Input | 10 | Power down |
| light green + black | CANL | I/O | 6 | CAN Low |
| red + black | CANH | I/O | 7 | CAN High |
| purple | Fork Right | Input | 8 | Select right track |
| pink | Fork Left | Input | 1 | Select left track |
| yellow | Analog Out | Output | 4 | 0-3V (1.5V center) Analog track position |
| blue | PWM Out | Output | 15 | Track position PWM output |
| brown | Left Marker | Output | 9 | Left marker detected |
| orange | Right Marker | Output | 11 | Right marker detected |
| green | Track Present | Output | 13 | Track detected |
| grey | RxData | Input | 3 | RS232 receive data |
| white | TxData | Output | 2 | RS232 transmit data |
| white + black | Reserved | N/A | 12 | Do not connect |

Figure 18. Input and output pins on the 15-pin DSub connector of the RoboteQ magnetic sensors [47].

As seen in Figure 18, the pins available to interact with the sensor are as follows:

- Power Down: If this pin is connected to ground, the sensor will automatically turn off. If it is set to a voltage higher than 1.5V, the sensor will turn on. Never apply a voltage higher than 5V on this pin.
- Track Present is an output pin. It is set at 5 volts if the sensor detects a line. If not, it is connected to ground.
- Analog Out returns the position of the tape using a voltage range from 0 to 3 volts, therefore 1.5V means that the line is in the centre of the sensor. It provides a resolution of 18.75mV. When no line is detected it is held at 1.5V, so it is very important to monitor the Track Present signal mentioned above.
- PWM Out also returns the position of the line, but with a pulse width modulation.
- Fork Right and Fork Left are two inputs. Depending on which of the two we set to 5 volts the sensor will follow one or the other line. This means that, if we have a fork of lines to the right and we keep 5 volts at Fork Left pin, the sensor will still give the position of the line that continues straight. If we keep 5 volts at Fork Right pin the sensor will give the position of the line that turns right. See Table 10.
- Left Marker and Right Marker are two outputs. They are set with 5 volts if a left or right marker is detected. The marks detected by the sensor consist of portions of the magnetic band, located in the opposite direction on the sides of the line.

| ***Fork Left*** | ***Fork Right*** | ***Analog and PWM Output*** |
|---|---|---|
| Low | Low | No change |
| High | Low | Left track position |
| Low | High | Right track position |
| High | High | Left or right track position depending on command received on RS232 |

Table 10. Route selected depending on Fork Left and Fork Right pins [47].

The sensor can also be controlled by commands sent via RS232 interface or via CAN messages. In order to receive information from it through CAN messages, it is necessary to program a script in the sensor so that it sends the information of the position of the line by means of some specific messages.

To carry out the project, the following pins and interfaces will be used (Figure 19). To control the sensor and receive information from it the pins Power Control, Fork Right, Fork Left & Track Present will be connected right to the *Humming Board CBi* GPIOs of the Control module. Using these we will be able to turn on or off the sensor, make it follow the left or right path, and check if the line is being detected. At the same time, the sensor will send the line's position through CAN frames, thanks to a previously programed script. The sensor is therefore connected directly to the vehicle's CAN bus, to which the control and Communication module is also connected.

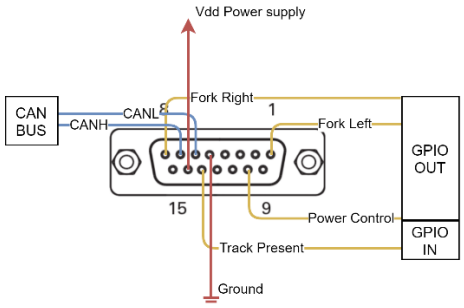



Figure 19. Connections to be made in the 15-pin DSub connector of the RoboteQ magnetic sensors.

As mentioned above, *RoboteQ* sells various sensors of this type depending on their size. In order to choose the optimum sensor, a study has been carried out, considering that the magnetic band used will be 50mm wide, and the resolution of every sensor is 160 points side to side. Figure 20 shows an example of the calculation of the maximum detection range that a 160mm sensor can carry out with a 50mm magnetic tape.

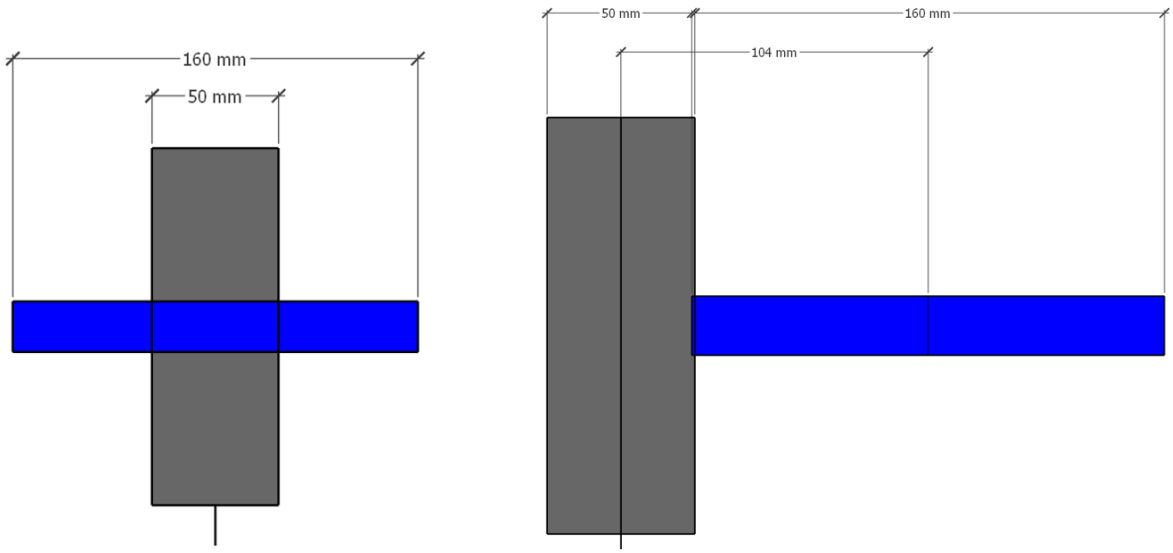



Figure 20. Calculation of the maximum detection range with a 160mm sensor and a 50mm magnetic tape.

Figure 21 shows the results obtained for each of the *RoboteQ* magnetic sensors. The Dmax parameter refers to how much the centre of the magnetic strip can be deviated from the centre of the sensor without losing its location by the sensor. For example, if using the 160mm sensor, and the vehicle deviates more than 104mm from the line, the sensor would lose the line and the system would fail.

| *Sensor name* | *Range detection* | *($D_{máx}$) One side maximum distance oscillation* |
|---|---|---|
| **MGS1600GY** | 160mm | 104mm |
| **MGSW3200** | 320mm | 183mm |
| **MGSW4800** | 480mm | 262mm |

Figure 21. Comparison of magnetic sensors depending on the one side maximum distance oscillation.

In order to know the period of the CAN frames sent from the sensor, containing the information about the position, some calculations have been made to find out the maximum deviation of the vehicle when it encounters the tightest possible curve (3.4 metres for the Twizy)[1] at a particular speed. The results can be seen in Table 11.

| *Frame's period* | *5km/h deviation* | *10km/h deviation* |
|---|---|---|
| **50ms** | 1mm | 3mm |
| **100ms** | 3mm | 11mm |
| **200ms** | 11mm | 45mm |
| **300ms** | 25mm | 102mm |
| **400ms** | 45mm | 184mm |
| **500ms** | 71mm | 293mm |

Table 11. Deviation before the system acts according to the period of the CAN message and the speed of the vehicle.

Interpretation and decision-making according to these studies are part of the last main objective of the full *TwizyLine* project shown in section 1.1. Therefore, it is not explained in this final degree project.

As seen in section 2.3, the vehicle must have an RFID antenna in order to read the RFID tags placed along the route and thus make decisions. The company *SparkFun* sells 125KHz RFID antennas that are capable of reading RFID tags up to 20cm away [48], enough for the project. The antenna is the ID-20LA. It works with 5 volts and when it detects an RFID tag it returns a specific frame (see figure12) indicating its identifier in hexadecimal numbers represented in ASCII with a checksum [49]. In order to connect it to the Control module, there is an adapter sold by the same company to be able to transmit the information from the antenna via USB [50].

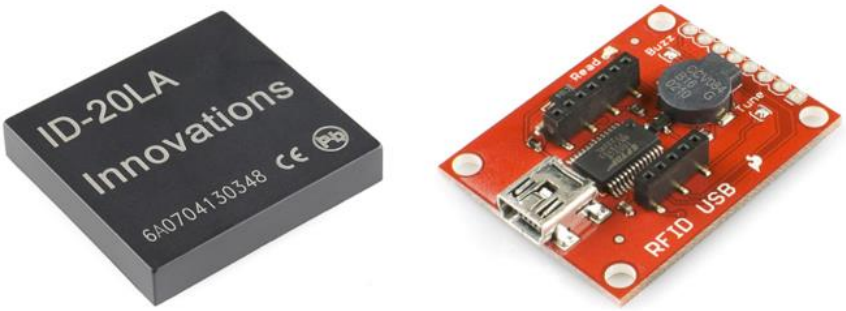



Figure 22. ID-20LA RFID antenna and RFID USB adapter [48] [50].

| STX (02h) | DATA (10 ASCII) | CHECK SUM (2 ASCII) | CR | LF | ETX (03h) |
|---|---|---|---|---|---|

Figure 23. ID-20LA RFID antenna frame format [49].

[1] The tightest possible curve is the one with the radius of the vehicle's turning radius. To follow it correctly, the direction of the vehicle must be turned to its maximum.

### 3.1.4. Control

To carry out the lateral control of the vehicle, as seen in section 2.3, due to the lack of assisted steering on the Twizy vehicle, and the complications resulted from installing a new steering column, an electric motor coupled to the steering column will be used in a similar way to the work carried out by a research group at the University of Cartagena [33]. This engine must offer the necessary torque to move the direction of the vehicle in any circumstance, not only in movement, but also when the vehicle is stopped (when the vehicle is stopped it is necessary to apply a greater force to be able to turn the steering). It must also be able to turn it at a sufficient speed so that the vehicle is agile during its autonomous movement. This last requirement is less relevant in the project since the vehicle will move at very slow speeds. The member of the team, Adrián Mazaira, made some measurements at the Renault engineering site in Valladolid of the necessary torque to move the direction of a *Twizy* when this one is stopped applying weight in a lateral of the steering wheel and considering the radius of this one. Approximately 14 Nm of torque is needed to move the direction under these circumstances.

After several investigations and supported by the *Maxon Motor* company [36], it was decided to use the 100W EC 60 flat brushless electric motor [51] mounted with a 1:81 GP 52C gearhead [52] and a MILE encoder with 4096 counts per turn [53]. This motor can provide a continuous torque of 261mNm. To achieve a higher torque than mentioned above, a gearhead with a ratio of 81:1 is coupled to it. Hence, a torque of 21,141 Nm is available at the output of the gearbox, which is greater than 14 Nm and therefore sufficient. A gear will be attached to the output shaft of the gearhead. This will be coupled to another gear of the same size which will be attached to the steering column. The ratio between the two shafts is therefore equivalent and the torque offered by the gearhead is the torque to be applied in the steering column. In order to have precise steering control by the system, it is necessary to know the position of the steering at all times. This is why an encoder is also attached to the motor. This encoder is not absolute, but incremental[2], therefore it is necessary to make a previous calibration in order to have reliable data of the steering position. This electric motor is capable of delivering a continuous speed of 3210 rpm. This means that, with the reduction of the gearhead, it can turn the steering column in 2.27 seconds. However, according to information given by Maxon Motor engineers, with good engine power management, speeds of up to 6000rpm (1.21 seconds per 1.5 turns of the steering column) can be reached at specific times without overloading the engine. It is possible to use this speed to implement the system as these aggressive turns will be made at specific times.

A motor controller is needed to manage the power and movement of the motor and the data provided by the encoder. Maxon's EPOS 4 70/15 controller [54], fully compatible with the chosen motor and encoder, will be used (Figure 24). This simplifies the control of the motor; through a series of pre-set commands it is possible to calibrate the encoder information and order the motor to move to a specific position. The controller is pre-programmed to make full use of the electric motor's capabilities without overloading it. Therefore, thanks to this one, we will be able to reach the speeds previously mentioned. It is protected from overcurrent, excess temperature, undervoltage, overvoltage, voltage transients and short-circuits in the motor cable. It works with voltages between 10 and 70 volts so there is no problem connecting it to the vehicle's power source. By means of some libraries for *Linux* that can be downloaded from the web page of *Maxon*, the system can be controlled by means of some pre-established commands via CAN, USB, or RS232. Therefore, the controller will

[2] Incremental encoders do not give an absolute value of the motor position. They give an incremental value starting from zero at power-up. Therefore, their value is only valid if the system is calibrated at that time in order to know the equivalence of the offered encoder values with the vehicle's steering position.

follow the commands sent by the Control module according to its status and the data it receives from the magnetic sensor.

It is intended that communication between the Control module and the engine controller will take place via CAN bus of the vehicle, however, due to the lack of a real vehicle for testing, it cannot currently be guaranteed that this communication channel will be used. In the case this would produce some kind of interference (due to the use of message identifiers already used by the vehicle systems) the communication would be done via USB.

Figure 24. Maxon EPOS4 70/15 Controller [54].

For throttle control, a bypass will be made in the connection between the throttle and the vehicle systems. According to meetings held with qualified Renault personnel and the article from the University of Cartagena [33], the pedal consists of two potentiometers that give different voltages depending on the pressure applied to it. This is done for safety, if the two values received by it are consistent with each other, the vehicle accelerates. These voltage values are interpreted by an ECU that controls the power of the Twizy's electric motor and sends status information via the CAN bus. Through tests carried out by Adrián Mazaira at the Renault engineering site in Valladolid we know that the signals returned by the accelerator potentiometers vary between 4 and 12 volts and between 2 and 8 volts. However, as we do not yet have the vehicle at the university for more detailed studies, we have no more information about its operation. The design shown below approximates what can be implemented if certain assumptions are correct.

As we have just said, there are two potentiometers inside the accelerator. We know that the connection of the accelerator with the ECU of the Twizy consists of 6 pins, so we can think that 3 pins will be for one potentiometer and other 3 for the other one. This way two pins would give the voltage for the potentiometer to work and the third one would be the output voltage of the potentiometer. If these assumptions are true, we will use two digital potentiometers controlled by an SPI architecture. Since the Control module does not have this interface, another simple embedded system with this interface can be used to manage the potentiometers. A good candidate is *Arduino Micro* [55], a small and low-cost embedded system that can be programmed to receive commands via USB and, based on these, send commands via the SPI bus to the potentiometers or turn GPIO pins on or off. Non-latching relays will be used to bypass the connection between the accelerator pedal and the vehicle ECU. These relays only change state when they receive current. If they stop receiving current the switches return to their original position. It has been decided to use this system so that,

in an emergency, if the whole system is switched off, the relay always returns to the original position and the driver retrieves control of the vehicle.

In Figure 25 we can see a scheme of the possible implementation to have the control of the accelerator of the vehicle. The controller of the potentiometer would be the *Arduino Micro*. This would also be responsible for controlling the relay. The Control module would send commands to it via USB and can be located in another area of the vehicle.

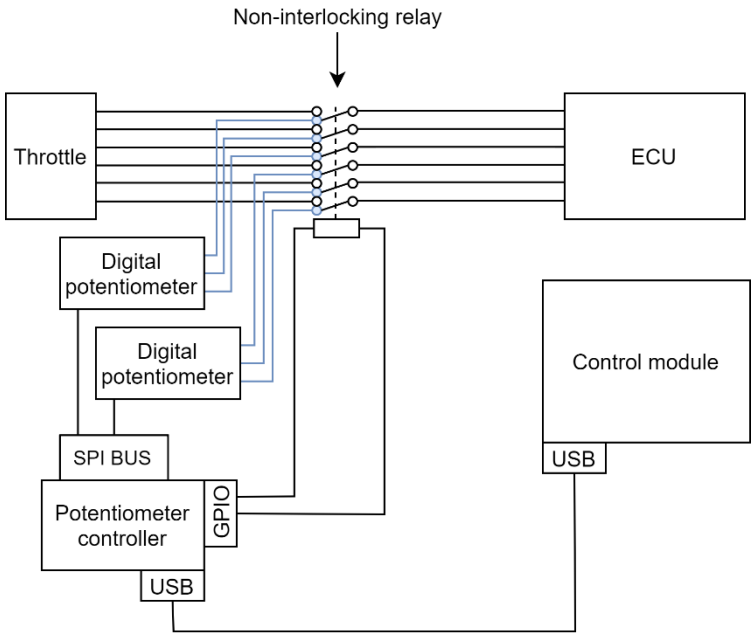



Figure 25. Possible implementation of throttle control.

### 3.1.5. Security and complete design

This entire system must have a safety mechanism that allows it to be disconnected at any time. The simplest and most effective mechanism is to use an emergency button (Figure 26) that acts as a switch and disconnects power to all the added systems in the vehicle. In this way, everything would stop working and the driver would have control again.

Figure 26. Example of an emergency button.

With this button we have completed the design made in this phase of the *TwizyLine* project. In Figure 27 a diagram of this can be seen. In this design the braking system, the verification of the vehicle's

acceleration system and the checking of the communication via CAN of the Control module with the engine controller is missing. In the case this communication cannot be carried out via the CAN bus, it would be necessary to add more USB ports to the Control module. This can be done using a USB port hub. In the next objective of the project, when the vehicle is available at the university, the design will be completed and implemented.

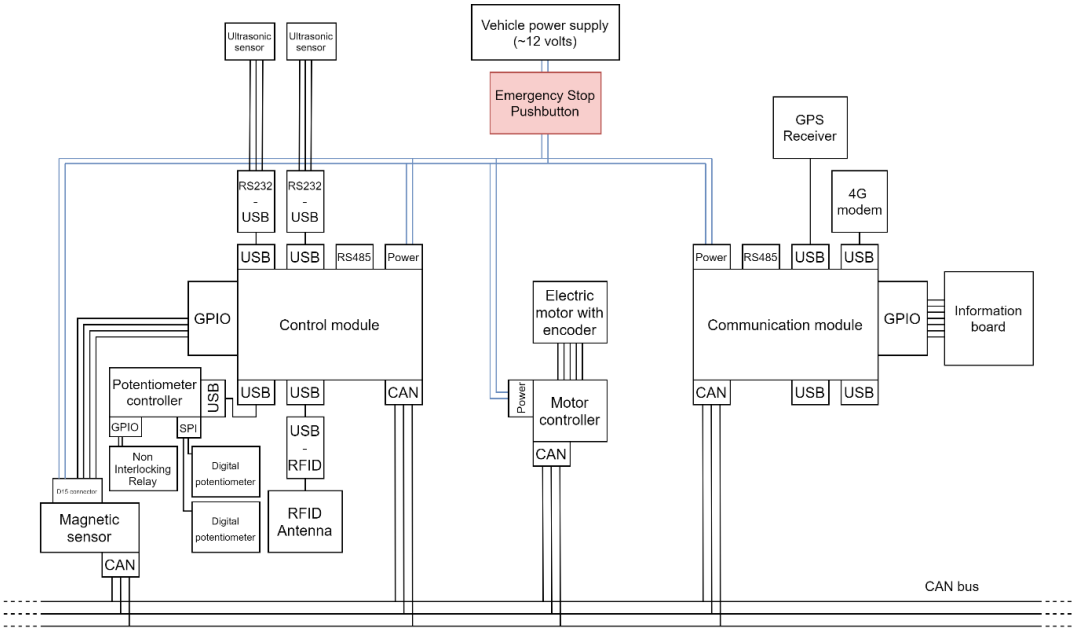



Figure 27. Interconnection of the different system modules.

To achieve the objectives described in Section 1.2 of this final degree project the following components have been provided: the Control module, the Communication module, a simulated CAN bus connected between both, the GPS receiver and the 4G modem. Figure 28 shows how all these elements have been interconnected to work with them.

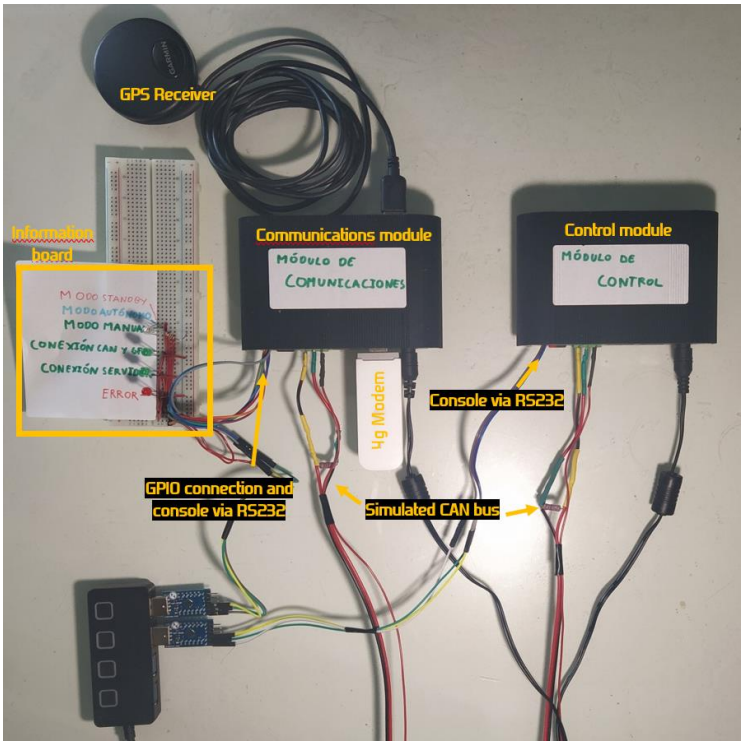



Figure 28. Equipment used and connected for the realization of this final degree project.

The *Humming Board CBi* does not include an HDMI port to connect a display. Therefore, to control and program it, an RS232 interface, which is included in the Humming Board for this purpose, was used. A terminal is accessible from this one. The CAN bus has been made by connecting two 120 ohm resistors in parallel at the ends, simulating the CAN bus of the vehicle. Figure 29 shows how the CAN bus is connected to the Communication module. To make the information board connected to the GPIO of the Communication module (Figure 30), a multicolour LED has been used to inform about the status of the vehicle; two green LEDs inform about the status of the connection with the back-end, with the GPS receiver, and with the Control module; and a red LED that lights up in the case of failure.

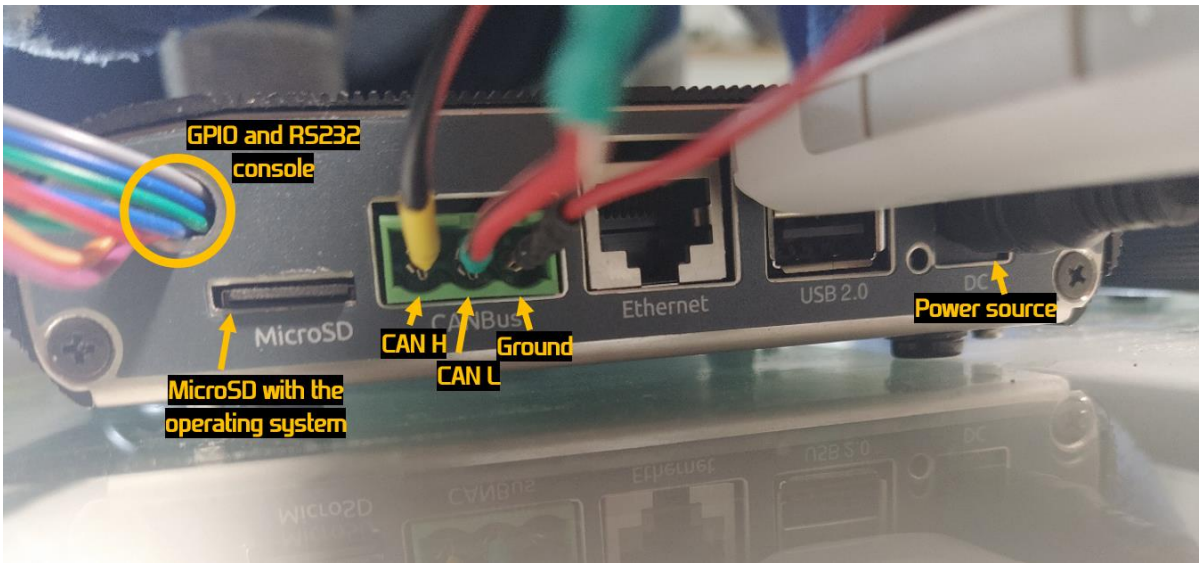



Figure 29. Rear view of the Communication module.

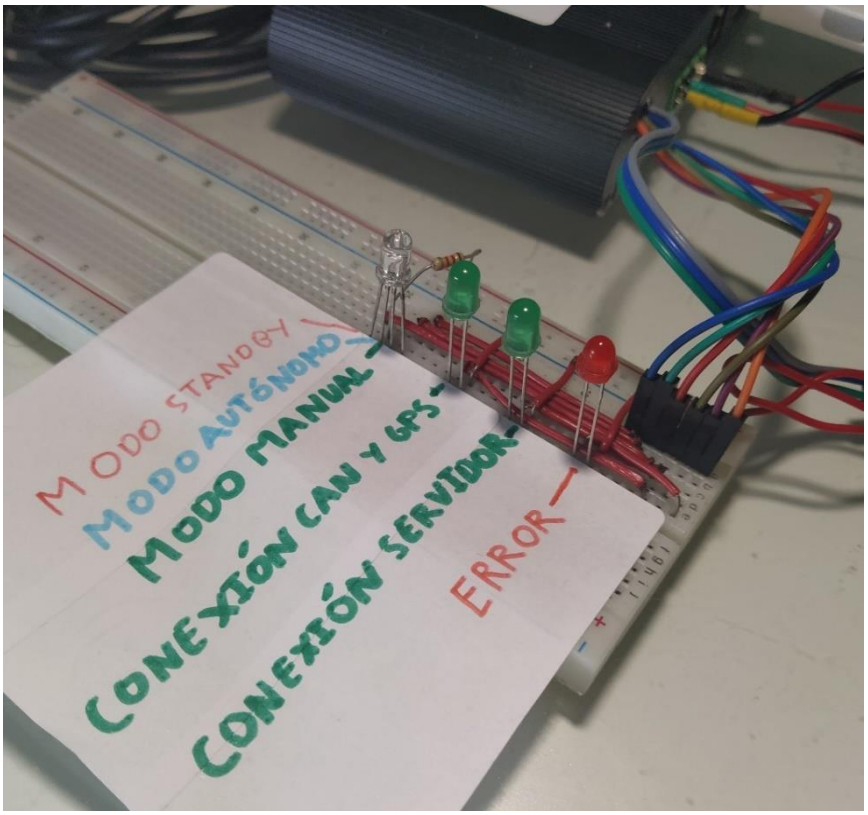



Figure 30. Information board.

## 3.2. Vehicle system states

To implement the system 5 states have been defined in which the vehicle can be. Depending on the state, the system will perform different tasks. In all states the communication and Control modules remain switched on, so that they can carry out their tasks or await a change of state. On the other hand, the devices connected to these modules can be switched off or on, depending on the status. In addition, because the processing system is composed of these two modules, it is necessary to guarantee a complete synchronization between both so that they are always in the same state. To achieve this, a specific CAN messaging has been designed that will be shown in the next section 3.3. Meanwhile, the states are: Start Up, Normal, Autonomous, Standby, and Failure mode.

The **Start Up mode** is the state the system is starting up, or rebooting due to a system failure. In this state both modules start simultaneously, check that all devices are connected and get synchronized. The Communication module also checks if a connection can be established with the configured MQTT broker and if the back-end is available. When both are ready, the Communication module goes to the next default state (Normal, Autonomous or Standby) and instructs the Control module to do the same. In the case of *errors* at any point in this procedure the system immediately goes into Failure mode.

The **Normal mode** is the state the system is in when the vehicle is being driven by a conventional driver. In this state, most systems connected to the Control module remain off since the vehicle is controlled by a person. The Control module keeps on and checks only the RFID sensor so that if an RFID tag is detected, it informs the Communication module, and this informs the back-end. This is necessary due to the operation of the car park, the first RFID of the parking located at the entrance must be detected in normal mode to confirm to the back-end that the vehicle is in the right place to start its autonomous operation[3]. On the other hand, the Communication module informs the back-end, in addition to the RFID tags detected by the system, about the battery status and the location of the vehicle periodically, and about *warnings*[4] issued by both modules. At the same time, this module waits orders from the server to change the status. In the case of *errors* triggered by any of the two modules, the system goes into Failure mode.

The **Autonomous mode** is the state the system is in when it is in driverless operation, i.e. when the driving tasks of the vehicle are carried out by the installed system. In this state all the devices installed in the vehicle are switched on and are used. The Control module waits to receive a route through the Communication module and, once received, based on the information read by the magnetic sensor, controls the vehicle laterally and longitudinally with the previously mentioned actuators (Section 3.1.4) to follow the magnetic band. In the car park there are certain RFID tags that are placed before the forks of the magnetic band so that the system of the vehicle can decide which band to follow. In the case of detecting an RFID tag, the Control module will check the route it has received through the Communication module to act in one way or another. In addition, when it detects an RFID tag it always informs the Communication module. If ultrasonic proximity sensors detect any obstacle or vehicle in front of the vehicle, the Control module will adapt the speed of the vehicle and stop it if necessary, as explained in section 2.3.3.1. If the obstacle disappears before a pre-set time, the vehicle continues its route normally. If it does not disappear, a timer will be triggered, and the Communication module will be informed. On the other hand, the Communication module, as in normal mode, periodically reports the position of the vehicle and the status of its battery. In this

[3] In the Final Degree Project of Samuel Pilar Arnanz [13] the complete design of the parking lots where this vehicle will operate can be seen.

[4] In this system, *warnings* and *errors* are the two types of failures that can occur. Section 3.5 goes into more detail about their meaning, their differences and all the possible errors that can occur.

state, the module expects to receive from the back-end a route to follow, orders to stop the vehicle, or to change the status. When any of these orders are received, the Communication module informs the Control module. In addition, the Communication module informs the back-end about the RFID tags detected by the Control module, the expiration of the obstacle timer, and any *warnings* issued by any module. In the case of *errors* triggered by any of the two modules, the system goes into Failure mode.

The **Standby mode** is the energy-saving state of the system. In this mode, sensors, actuators, and the GPS receiver remain off. Therefore, the Control module does not carry out any tasks, it just waits for the Communication module to give it the command to change state. The Communication module maintains communication with the back-end by informing it about the battery status. In the case of any *warning* launched by either module when changing to this state or while in it, the Control module informs the back-end. In the case of *errors* triggered by any of the two modules, the system goes into failure mode.

The **Failure mode** is the state that is passed to when any of the two modules throws an *error*. In this state, if the last state was Autonomous mode, the vehicle stops automatically. Through CAN messaging, both modules are placed in this state no matter which one of them has triggered the *error*. If the Control module is the one that has triggered the *error*, it sends the information of the error (error code and parameters) to the Communication module. Once they remain synchronized and the Communication module has all the necessary information, each module performs a different task. On the one hand, the Control module restarts and waits to receive commands from the Communication module to start in the Start Up state. On the other hand, the Communication module checks if there is an established connection with the back-end. If there is (and therefore the *error* has not been caused by a failure in the connection with it), the module sends the error code and its parameters to the back-end and waits for a restart message to be sent from the back-end. When it receives it, it goes to the Start Up state, informs the Control module and synchronizes with it to restart everything. In the case of no connection with the back-end, it waits for a timer with predefined time and tries to establish a new connection. This process is carried out continuously until the connection with the back-end is achieved. Once it succeeds, it goes to the Start Up state, informs the Control module and synchronizes with it to start everything again. Annex II shows a table with all the failures defined and contemplated in this phase of the project.

As we can see, the changes of state always begin in the Communication module except for the failure mode. This is the one that knows when to change state since it has communication with the back-end. Therefore, the Control module follows the status change commands received by the Communication module.

## 3.3. CAN Messaging

In this section the CAN messaging that is introduced in the vehicle for the communication between the Control module and the Communication module will be shown. This section will not show the messaging that will be used to communicate with other added devices such as the magnetic sensor or the steering motor. This is part of objective 4 of the project shown in section 1.1 and therefore not part of this final degree project.

As seen above, this messaging must allow a complete synchronization between the control and Communication modules. At the same time, it must allow the immediate detection of any anomaly in the system, such as the malfunction of one of the modules, or the disconnection of the CAN bus. Also, it must allow the transmission of all the orders and information that must be sent between both modules to change the status, inform about the route to be followed, expiration of the obstacle timer, information about *warnings* and *errors*, etc. Six different message types have been designed and therefore with different CAN frame IDs. Thanks to the information provided by a GitHub user who

made reverse engineering in a Twizy to know the meaning of the messages sent by the Twizy ECUs through the CAN bus [45], it is possible to know which IDs are not being used in order to use them. This is why the range of IDs from 100 to 105 is used. When a Twizy is finally delivered to the University, it will be checked if these IDs are actually available and, in the case they are not, others will be used. Table 12 shows the messages used, with their ID and the different signals sent in each CAN frame.

| ID | Type | Period (ms) | Source | Dest. | Name | Length (bytes) | Bit(s) | Signal name |
|---|---|---|---|---|---|---|---|---|
| **100** | Cyclic | 100 | COM | CON | COMM-STATUS | 1 | 0-1 | STATE-COM |
| **100** | Cyclic | 100 | COM | CON | COMM-STATUS | 1 | 2 | PAUSE-COM |
| **100** | Cyclic | 100 | COM | CON | COMM-STATUS | 1 | 3 | RFID-ACK |
| **100** | Cyclic | 100 | COM | CON | COMM-STATUS | 1 | 4 | CON-ERR-ACK |
| **101** | Cyclic | 100 | CON | COM | CONT-STATUS | 1 | 0-1 | SATE-CON |
| **101** | Cyclic | 100 | CON | COM | CONT-STATUS | 1 | 2 | PAUSE-CON |
| **101** | Cyclic | 100 | CON | COM | CONT-STATUS | 1 | 3 | TIMEOUT |
| **101** | Cyclic | 100 | CON | COM | CONT-STATUS | 1 | 4 | GOTO-ACK |
| **102** | Cyclic & spontaneous | 100 | CON | COM | RFID | 5 | 0-39 | ID |
| **103** | Cyclic & spontaneous | 100 | CON | COM | CON-ERR | 8 | 0-1 | ERR-TYPE |
| **103** | Cyclic & spontaneous | 100 | CON | COM | CON-ERR | 8 | 2-15 | ERR-INFO |
| **103** | Cyclic & spontaneous | 100 | CON | COM | CON-ERR | 8 | 16-63 | ERR-ATTR |
| **104** | ISO-TP | | COM | CON | GOTO | | | |
| **105** | ISO-TP | | CON | COM | GOTO | | | |

Table 12. CAN messaging used for communication between both system processing modules.

The utility of the messages shown in table 12 and their signals will be explained as follows.

Both the Communication module and the Control module have an assigned message that reports their status periodically every 100 milliseconds, COMM-STATUS from the Communication module and CONT-STATUS from the Control module. The periodicity of this message is necessary since it allows both modules to know if the other module is connected and working correctly. In this way, if a module does not receive the message from the other module during a determined time, it will know that there is an anomaly and will be able to act appropriately (entering the failure mode and, if possible, informing the back-end). In addition, these messages allow at the same time to inform about the state of the module and to give and acknowledge orders. With the ACKs (acknowledge) signals it is possible to check the existence of anomalies in the system as well. These checks are very relevant in the system as it can take control of the vehicle and therefore, for security reasons, it must be able to detect any possible failure.

The **COMM-STATUS** message includes the following signals:

- STATE-COM: It consists of 2 bits. It reports the status of the Communication module, except for the failure mode. In this way, it informs about the state it is in to the Control module and orders it to change its state.
- PAUSE-COM: It consists of 1 bit. It makes sense in autonomous mode. When the vehicle is following a route there may be times when it is necessary to stop temporarily. The back-end informs the Communication module of this and the module, by setting this bit to 1, orders the Control module to stop the vehicle until the bit returns to 0.
- RFID-ACK: It consists of 1 bit that serves to confirm the reception of an RFID message sent by the Control module. This bit remains at 1 for half a second. Not receiving this bit at 1 after sending an RFID message would mean that 5 COM-STATUS messages had been lost or that the Communication module was not working properly, which implies a system anomaly.

- CON-ERR-ACK: It consists of 1 bit that serves to confirm the reception of a CON-ERR message. It serves to optimise the processing of the Failure state so that the Control module stops sending its error message and can be restarted earlier. If this assent is not received after sending a CON-ERR message, the Control module remains sending it until a timer expires, where it will also restart and wait for the Communication module to start in the Start Up state.

The **CONT-STATUS** message includes the following signals:

- STATE-CON: Similar to STATE-COM. Its purpose is to report the status of the Control module. In this way it can confirm the status to the Communication module.
- PAUSE-CON: Similar to PAUSE-COM. It reports that the vehicle has stopped, following the commands from the Communication module.
- TIMEOUT: It consists of 1 bit that informs that the obstacle timer has expired and that the vehicle is stopped waiting for new orders.
- GOTO-ACK: It consists of 1 bit that confirms the reception of the chained GOTO message. Like the other ACKs, this bit is set to 1 for half a second when this message is received.

It is possible to think that certain information, such as ACK signals, sent in these messages may be sent through other non-periodic messages. However, as seen before, it is necessary to send one periodic message per module to ensure the correct functioning of the system and it is recommended that the status of each module be confirmed periodically. When doing this, since the minimum size with payload of a CAN frame is 1 byte, there are more bits than needed to transmit the status, and therefore, they can be used to introduce the ACK signals, performing a more efficient use of the bandwidth.

The **RFID** message is sent by the Control module to the Communication module when it detects an RFID tag. It is sent periodically, and stops when the RFID-ACK signal is received at 1. The message includes a single signal (called ID) that serves to transmit the number of the detected tag. As seen in section 2.2 in figure 2.3, the length of the RFID tag number is 5 bytes and therefore the payload length of this message is 5 bytes.

The message **CON-ERR** informs the Communication module about the triggering of a *warning* or an *error* on the Control module. This includes 3 signals:

- ERR-TYPE: It is 2 bits that indicate the type of error. Currently there are only two types of errors: *errors* and warnings. However, it has been decided to reserve an extra bit for new error type definitions.
- ERR-INFO: It consists of 14 bits indicating the error code. In the Annex II the established error codes and their meaning can be seen.
- ERR-ATTR: It consists of 42 bits indicating the error attribute. Depending on the error, it may be necessary to send an error attribute, that is, a relevant value when the error occurs. This may indicate the reason for the error, or any other parameter of interest.

The **GOTO** message is used to send the route to be followed by the vehicle when it is in autonomous mode from the Communication module, which receives it from the back-end, to the Control module. It consists of a collection of blocks of 6 bytes that specify what the vehicle should do when an RFID is detected. It specifies if the vehicle must take the route to the right or left in the case of a fork and the speed at which it must go. Initially, the initial speed that the vehicle must reach is specified by means of the 2-byte initial block. Then the rest of the blocks specify the speed and the line that must follow according to detected RFID. Table 13 shows how these blocks are coded.

| Frame | Byte 1 | | Bytes 2-6 |
|---|---|---|---|
| | 7 | 6-0 | |
| **First Block** | Default Fork | Initial speed | - |
| **Normal Block** | Fork | Speed | RFID |
| **Last Block** | No matter | 0 | RFID |

Table 13. Encoding of the blocks in a CAN GOTO message.

The default fork parameter indicates which deviation the vehicle should take as default. The fork parameter indicates if the vehicle should turn left (0) or right (1) at the next fork after the specified RFID is detected. Once a new RFID is detected after this, if this RFID it is not in the GOTO message it will follow the default fork again. The speed parameter gives the speed in km/h that the vehicle must maintain at the beginning of the route or after detecting an RFID. The structure of this message is designed to provide flexibility and scalability in the system, and so that its messaging is prepared to follow more complex routes than those taken in the designed car park.

This message, as we can see, will always be larger than 8 bytes, the maximum payload size of a CAN frame. This is why a transport layer is needed, to be able to send messages with a length bigger than 8 bytes. The ISO 15765-2 standard [56], also known as ISO-TP (Transport Protocol), defines a transport protocol that works over CAN and allows sending messages with a maximum length of 4095 bytes, with the option of adding an extra addressing to the messages. To achieve this, it fragments the message, which is sent in several CAN frames and adds a specific header to them. To prevent the collapse of the receiver it implements a flow control mechanism, by which the receiver can report how many bytes it can receive at one time and can make the sender wait.

The protocol defines 4 types of frames [56]:

- Single frame: Frames that can carry a maximum payload of 7 bytes if the extra addressing is not used, or 6 bytes if it is.
- First frame: First frame of a fragmented message transmission. The message is fragmented if it does not fit into the payload of a single frame.
- Consecutive frame: Frames of a transmission of a fragmented message. It comprises all frames carrying fragments of the message except the first frame.
- Flow control frame: Frame sent by the message receiver to acknowledge the first frame and give connection parameters. Depending on these parameters, the sender must wait for the reception of another flow control frame each certain bytes sent.

The extra addressing of this protocol can be used in several ways. However, in this case it is not necessary to use it since the objective is to send messages longer than 8 bytes and we have two IDs for sending and receiving ISO-TP messages in both modules. Table 14 shows the format of the ISO-TP message that is introduced into the payload of a CAN frame with a specific ID, using normal addressing, i.e. no extra addressing.

| Frame | Byte 1 | | | | | Byte2 | Bytes 3-8 |
|---|---|---|---|---|---|---|---|
| | Bits 7-4 | | | | Bits 3-0 | | |
| | 7 | 6 | 5 | 4 | | | |
| **Single Frame** | 0 | 0 | 0 | 0 | Length | Data | |
| **First Frame** | 0 | 0 | 0 | 1 | Bytes to transmit | | Data |
| **Consecutive Frame** | 0 | 0 | 1 | 0 | SN | Data | |
| **Flow Control** | 0 | 0 | 1 | 1 | FS | BS | STim |

Table 14. ISO-TP frame format with normal addressing [57].

*SN* is the sequence number of the consecutive messages. It indicates in number of sent fragmented message. *FS* indicates the flow state. 0x0 means that the sender can continue sending, 0x1 means that the sender must wait. *BS* indicates the number of bytes the sender can send before waiting for another Flow Control frame. *STim* indicates the minimum time separation between each message sent by the sender.

Figure 31 shows an example of a segmented message transmission using ISO-TP.

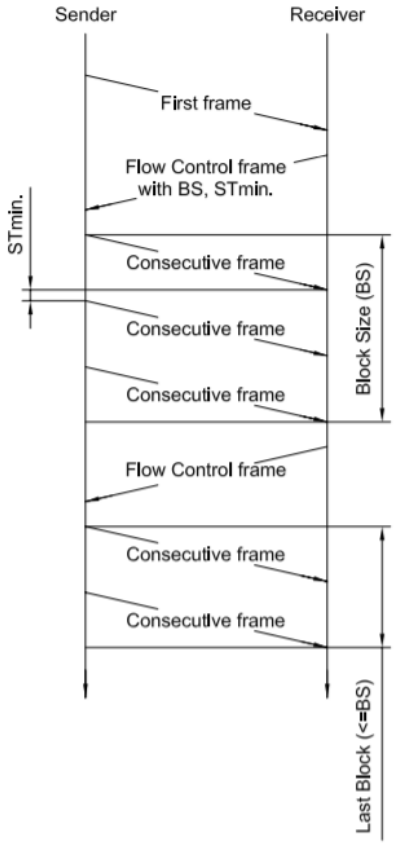



Figure 31. Example of a segmented message sent via ISO-TP [56].

The transmission of the GOTO message, i.e. the route to be followed by the vehicle, will be carried out via this protocol. The result will be the transmission of a series of messages between the Communication module and the Control module very similar to the one shown in Figure 31. However, this transport layer does not provide a complete reliable transmission, since, as we see in this figure, not all the messages are acknowledged. The receiver (Control module), as it knows the number of bytes it must receive, can detect if a message has not arrived but must inform the sender (Communication module) in some way. This is why the previously mentioned GOTO-ACK signal exists.

## 3.4. Vehicle control interface

As we have seen before, the vehicle is controlled by the back-end. It receives its orders and sends it information. To be able to do this, it is necessary to define a communication interface, that is, the communication protocol to be used and a collection of messages. In this project, it has been decided to use the MQTT protocol. The benefits of this protocol and a detailed explanation of its operation can be seen in Samuel Pilar Arnanz's final project degree [13]. It uses a client-server architecture based on publications and subscriptions on topics. Clients can send messages by linking them to a topic. Clients who want to receive messages linked to a specific topic must subscribe to this topic. In

this way, once the MQTT broker receives a message, it checks the topic, and forwards the message to those clients who are subscribed to it. The topics can be ordered in a hierarchical way. Therefore, a client can subscribe to a general topic that includes other topics.

In our project, each vehicle in the service will be assigned a unique identifier. This identifier will form their general topic. This general topic contains four specific topics: *battery*, *location*, *info,* and *order*. The vehicles will send periodic messages about the percentage of battery charge and the location of it through the *battery* and *location* topics, respectively. They will send any other type of information through the *info* topic and will be subscribed to the *order* topic by which they will receive orders from the back-end. The back-end will be connected to the MQTT broker as another client, will be subscribed to the *battery*, *location*, and *info* topics of all the vehicles, and will send orders to each of these through their respective *order* topic. In this way, the back-end will be able to receive information from all vehicles simultaneously and will be able to send specific orders to each one. Table 15 shows all the messages defined for the control and monitoring of the vehicle from the back-end.

| Publisher | Subscriber | Topic | Payload |
|---|---|---|---|
| Vehicle | Server | vehicleID/battery | [0-100]OR[-1] |
| Vehicle | Server | vehicleID/location | [(latitude),(longitude)]OR[No signal]OR[GPS not connected] |
| Vehicle | Server | vehicleID/info | CONNECT regist_plate |
| Vehicle | Server | vehicleID/info | TIMEOUT |
| Vehicle | Server | vehicleID/info | RFID (ID number) |
| Vehicle | Server | vehicleID/info | WRN (error code) [(attribute)] |
| Vehicle | Server | vehicleID/info | ERR (error code) [(attribute)] |
| Vehicle | Server | vehicleID/info | STARTING UP |
| Vehicle | Server | vehicleID/info | AM-ON OK |
| Vehicle | Server | vehicleID/info | AM-OFF OK |
| Vehicle | Server | vehicleID/info | STANDBY OK |
| Vehicle | Server | vehicleID/info | PAUSE OK |
| Vehicle | Server | vehicleID/info | CONTINUE OK |
| Vehicle | Server | vehicleID/info | GOTO OK |
| Server | Vehicle | vehicleID/order | CONNECTED |
| Server | Vehicle | vehicleID/order | STANDBY |
| Server | Vehicle | vehicleID/order | AM-ON |
| Server | Vehicle | vehicleID/order | AM-OFF |
| Server | Vehicle | vehicleID/order | CONTINUE |
| Server | Vehicle | vehicleID/order | PAUSE |
| Server | Vehicle | vehicleID/order | RESTART |
| Server | Vehicle | vehicleID/order | GOTO initialSpeed [T(10bytes ASCII RFID) [speed]] [V(10bytes ASCII RFID) (speed)] S(10bytes ASCII RFID) |

Table 15. MQTT messaging used for communication between the vehicle and the back-end.

As we can see in Table 15, there are different messages that can be sent through the topics mentioned above. The following paragraphs list and explain the usefulness of these messages.

The ***battery* topic** is used to send the percentage of charge of the vehicle's battery periodically. Therefore, under normal conditions, messages may contain a number between 0 and 100. However, if the Communication module is unable to collect this information from the vehicle, it sends the value -1 periodically. The period of this message is 5 seconds when the vehicle is in the Standby state and 1 second in the other states.

The ***location* topic** is used to send the coordinates (latitude and longitude) of the vehicle's position periodically. In the case the GPS receiver is not able to receive the necessary GPS signal, the message

*No signal* is sent instead of the coordinates. In the case the GPS receiver is not connected, the message *GPS not connected* is sent instead of the coordinates. This message is sent in all states with a period of 1 second except in the Standby state. When the vehicle is in Standby it is not moving and therefore it is not necessary to periodically receive its GPS information.

The ***info* topic** is used to send any kind of non-periodical information from the vehicle to the back-end. Through this topic a series of messages have been defined.

- The CONNECT message is used to verify communication between the Communication module and the back-end. The Communication module sends it in the Start Up state and waits to receive a reply from the back-end by the *order* topic with a CONNECTED message. In order for the back-end to have the registration number of the vehicle, it is also sent in this message.
- The TIMEOUT message indicates that the obstacle timer has expired.
- The RFID message indicates the identifier of the RFID tag that has been detected.
- The WRN message indicates that a *warning* type error has been issued, its error code, and its attributes if any.
- The ERR message indicates that an *error* type error has been issued, its error code, and its attributes if any. Sending this message implies that the vehicle's system is in Failure mode and is therefore waiting to receive the RESTART message through the *order* topic from the back-end to restart the system and go to the Start Up state.
- The messages STARTING UP, AM-ON OK, AM-OFF OK, STANDBY OK, CONTINUE OK, PAUSE OK and GOTO OK confirm the status of the vehicle system, confirm the stopping or continuation of the vehicle in autonomous mode, and confirm the correct reception of the GOTO message.

The ***order* topic** allows sending orders from the back-end to the Communication module. Through this topic a series of messages have been defined.

- The CONNECTED message, as seen above, is complemented by the CONNECT message sent by the vehicle. It is sent through the back-end after the vehicle sends the CONNECT message in order to confirm the connection.
- The STANDBY message instructs the vehicle's system to go into Standby mode. After sending this message, the back-end expects to receive a STANDBY OK message from the vehicle, confirming the correct status change.
- The AM-OFF message instructs the vehicle system to go into Normal mode. After sending this message, the back-end expects to receive an AM-OFF OK message from the vehicle, confirming the correct status change.
- The AM-ON message instructs the vehicle system to go into Autonomous mode. After sending this message, the back-end expects to receive an AM-ON OK message from the vehicle, confirming the correct status change.
- The CONTINUE and PAUSE messages are used to order the vehicle to stop or continue when it is following a route in Autonomous mode.
- The RESTART message is the one sent to the vehicle when it is in Failure mode, that is, when the vehicle has sent an ERR message. This message is designed to be sent once the necessary maintenance work has been carried out so that the Failure does not occur again. Once sent, the system is restarted.
- The GOTO message is used to specify the route to be followed by the vehicle in Autonomous mode. This message is equivalent to the GOTO message explained in section 3.3, therefore it is divided into 3 blocks (Table 16).

| Block | Sent text string |
|---|---|
| **First Block** | initialSpeed defaultFork |
| **Second Block** | T(RFID) [(speed)] |
| | V(RFID) (speed) |
| **Last Block** | S(RFID) |

Table 16. Encoding of the blocks in a MQTT GOTO message.

All blocks are separated by a space character in the message. The **first block** is mandatory and gives the initial speed that the vehicle must reach in km/s, and which deviation the vehicle must take as default (L for left and R for right). The **second block** is optional. Multiple blocks of this type can be concatenated to specify a route. It allows to specify if the vehicle must take the non-default deviation between this RFID and the next one, and the speed it must maintain after detecting a certain RFID. Note that there are two possibilities. On the one hand, we can specify that when a specific RFID tag is detected, the vehicle must follow take the non-default deviation between this RFID and the next one and, optionally, the new speed that it must maintain. This is done by entering the character T followed by the RFID identifier and, optionally, a space and the new speed. On the other hand, we can specify that, when an RFID tag is detected the speed of the vehicle must change but the vehicle must continue to take the default deviation. This is done by introducing the character V followed by the RFID identifier, a space, and the new speed in km/h. Finally, the **last block** is mandatory and specifies the final RFID. When the vehicle detects this RFID tag, it must stop since this is the end of the path. Two examples of the use of this message to define a route are shown in Annex III.

## 3.5. Failures management

Due to the risks involved in an autonomous vehicle, it is necessary to carry out a correct error management to stop the autonomous driving of the vehicle in the case of risk and thus guarantee safety. This is why two types of failures have been defined in this system: *errors* and *warnings*.

***Errors*** are any type of anomaly that involves a safety risk. This is for example the loss of communication between both modules (and therefore the Control module being out of control by the back-end), the loss of communication between the back-end and the Communication module, the non-response to status change commands, etc. In these cases, in order to minimize the risk, the system goes into Failure mode (state explained in Section 3.2) and therefore immediately stops the vehicle.

***Warnings*** are any type of anomaly in the system that does not imply a safety risk but that may be relevant to know about it. These are, for example, the incorrect reception of the CAN frame of the vehicle that indicates the battery status, the non-reception of GPS signal by the GPS receiver for a long time, or the reception of an unexpected message from the back-end (for example a malformed message). These anomalies need to be detected and managed, and the back-end must be informed of them, but they do not imply a security risk and therefore the system continues to operate normally.

To address these failures, several error codes have been defined. Each of them refers to a specific anomaly. These error codes are 14 bits long and are organized in a specific way to facilitate their management and filtering according to the origin or reason. Table 17 shows how the error codes have been organized. Many codes are still available for future extensions of the project. Moreover, only the errors related to their communication with the Communication module have been defined in the Control module, which is what has been implemented in this final degree project. When starting to perform objective 4 of the *TwizyLine* project (explained in section 1.1), the list of error codes related to the Control module will be extended.

| | | | |
|---|---|---|---|
| **Vehicle [0-255]** | Communication module [0-127] | Communication with internal devices [0-15] | Communication with Control module [0-8] |
| | | | Communication with GPS and battery data reception [9-15] |
| | | Communication with back-end [16-31] | - |
| | Control module [128-255] | Communication with Control module | - |
| **Back-end [256-511]** | - | - | - |

Table 17. Error codes depending on the origin and type of error.

Annex II shows a table with all the failures defined and contemplated in this phase of the project. The table contains their error code, their name, the meaning of their attribute (if any) and an explanation of the attribute including possible reasons. Although in the table we see that the error codes defined are always linked to a *warning* or an *error*, it has been decided to maintain the possibility that the same error code can be of different types depending on the moment it is triggered. This is why the message CON-ERR explained in the previous section (Section 3.3), it has a signal that indicates the type of error and another signal that indicates the error code.

## 3.6. Communication module

As explained before, the Communication module is responsible for monitoring the state of the vehicle's battery and receiving information about the location of the vehicle; managing communication with the back-end, receiving orders and giving information about the state of the vehicle; and controlling the Control module, by sending it orders and interpreting its messages.

To achieve this, a series of processes, which work together, has been implemented. Each process carries out a specific fundamental task of the module. Communication between them is carried out through work queues that follow a specific messaging and shared global variables. In order to make the software as modular as possible, each process follows a well-defined specific interface, so that, in the case of future extensions of the project, the modification or replacement of any specific task of the module is as simple as possible.

Figure 32 shows a diagram of all the processes and threads that run in the Communication module, as well as the working queues and shared global variables used to carry out communication between them. Note that communications through work queues are done in one direction, that is, they perform simplex communications. All messages sent through these queues consist of ASCII-coded text strings for easy programming and managing.

There are **7 different processes** that can be executed in the Communication module. The main process is responsible for managing the status and errors of the vehicle. It creates the child processes *MQTT receiver*, *GPS receiver* and *CAN receiver*. These in turn can launch other child processes or threads needed to perform all the tasks of the module. Note that the work queues are used for most but not all communications between processes. Some shared global variables are used to send the coordinates of the vehicle location and the percentage of the vehicle battery to the process that sends this information periodically to the back-end (*MQTT periodic sender*). This is because the information received by the GPS or by the vehicle can be received at a higher rate than the rate of sending information to the back-end and, therefore, if working queues were used, the information sent could be outdated and the queue could be filled.

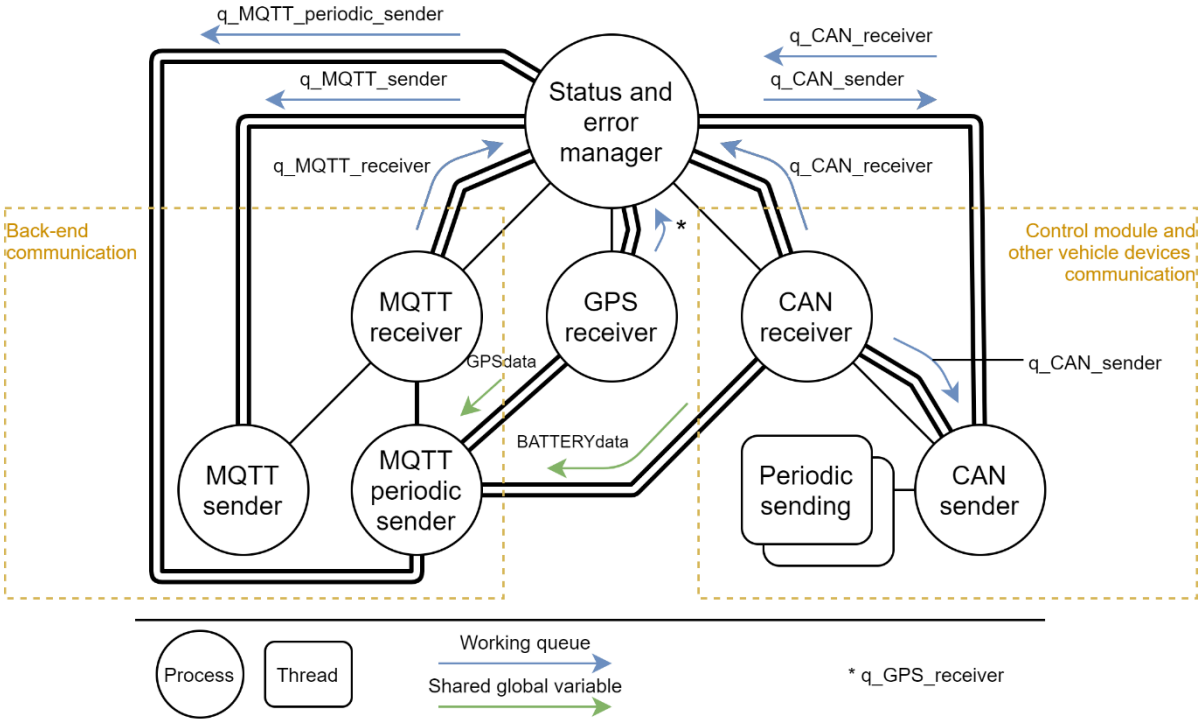



Figure 32. Diagram of the processes that are executed in the Communication module.

The following subsections will explain how each of these processes works. The operation of the processes related to the communication with the back-end is explained in subsection 3.6.1. The operation of the processes related to communication with the Control module and the reception of messages from the vehicle via the CAN bus is explained in subsection 3.6.2. The access to the information given by the GPS receiver and the operation of the process in charge of handling it is explained in subsection 3.6.3. Finally, the operation of the main process that manages the rest of processes, the state and the errors of the module is explained in subsection 3.6.4.

Note that at the beginning of the programmed code, a series of global variables are also defined with the main parameters that can be modified depending on the vehicle in which the module is installed, the location of the server, and some default values. These variables define: the vehicle ID in the *TwizyLine* service, the vehicle number plate, the IP address of the MQTT broker, user and password to establish the connection with it, the IDs of all the CAN frames used and the state to go to by default after the Start Up state.

### 3.6.1. MQTT communication

To carry out the communication of the vehicle with the back-end, the Communication module must have an internet connection. As seen in section 3.1.2, the 4G *Huawei E3372* modem is used for this purpose. This modem incorporates a flash memory with the necessary drivers for its correct operation in Windows. However, in Debian, these drivers are not supported, and the operating system interprets the modem as a storage unit. To solve this, there is a package called *usb-modeswitch* that is able to detect this problem and send a command to the operating system to interpret the device as a 4G modem and not as a storage unit [58]. Once this package is installed and having the *usb-modeswitch* daemon enabled and running, the operating system is able to connect to the internet through the mobile network when the modem is connected. It is also able to manage disconnections due to lack of signal, detecting it and trying to re-establish the connection automatically.

Once we have an internet connection it is necessary to program the necessary processes to implement the MQTT messaging of the system explained in section 3.4. To use the MQTT protocol,

the *paho-mqtt 1.5.0* library for *Python 3* [59] has been used. As seen in Figure 32 there are 3 processes that take care of this task. Each of them will be explained below.

The ***MQTT receiver*** process establishes and monitors the connection with the MQTT broker, subscribes to the specific vehicle *order* topic and receives the messages linked to this topic. It sends messages to the *Status and error management* process through the *q_MQTT_receiver* work queue following a defined messaging (Table 18). This process, when launched, configures the MQTT parameters of the connection (keep alive timer and user and password to establish connection). After that, it tries to connect to the MQTT broker with the pre-configured IP stored in the global variable defined at the beginning of the code. If the connection cannot be established, the process sends an *error* message through the work queue with the error code of the *error* that has occurred (See Table 18). If the connection is successfully established, it sends an OK through the work queue and launches the *MQTT sender* and *MQTT periodic sender* processes. Once the connection is established, the process has two main tasks. On the one hand, it is in charge of receiving all the MQTT messages linked to the *order* specific topic of the vehicle, unpack them and send the payload of these through the work queue, adding an M character ahead to indicate that it is a message. On the other hand, it sends MQTT ping requests to the MQTT broker every 5 seconds and checks if the server responds to ensure that the connection remains correctly established. In the case the connection is lost, it sends the corresponding *error* message through the work queue.

| Work queue | Message | Explanation |
|---|---|---|
| **q_MQTT_receiver** | OK | Connection with broker MQTT established |
| **q_MQTT_receiver** | M (payload) | Payload received from server linked with *order* topic |
| **q_MQTT_receiver** | E (error code) [(attribute)] | Triggering an *error* type failure |
| **q_MQTT_receiver** | W (error code) [(attribute)] | Triggering a *warning* type failure |

Table 18. Messages sent through the *q_MQTT_receiver* work queue of Communication module.

The ***MQTT sender*** process sends messages through the specific vehicle info topic. Through the work queue *q_MQTT_sender* the payload of the message that must be sent through that topic is given to this process. Therefore, this process must listen to this queue and send a message when it is ordered to do so. If the message TERMINATE is sent to it via the work queue, the process that received it finishes its execution.

| Work queue | Message | Explanation |
|---|---|---|
| **q_MQTT_sender** | (payload) | Payload to send through *info* topic |
| **q_MQTT_sender** | TERMINATE | Stop process |

Table 19. Messages sent through the *q_MQTT_sender* work queue of Communication module.

The ***MQTT periodic sender*** process sends the percentage of the vehicle battery charge and the location of the vehicle periodically through the *battery* and *location* specific topics of the vehicle. This information is obtained through the shared global variables *GPSdata* and *BATTERYdata*. As seen in section 3.4, the sending of these messages varies depending on the status of the vehicle system. If the vehicle is in Standby mode, this process sends only the battery status every 5 seconds. If the vehicle is in any other state, the process sends the battery status and vehicle location periodically every second. The *q_MQTT_periodic_sender* queue is used to inform the process when it must act in one way or another and to order it to finish its execution if necessary (Table 20).

| Work queue | Message | Explanation |
|---|---|---|
| **q_MQTT_periodic_sender** | STANDBY | Only send battery information every 5 seconds |
| **q_MQTT_periodic_sender** | NOTSTANDBY | Send location and battery information every second |
| **q_MQTT_periodic_sender** | TERMINATE | Stop process |

Table 20. Messages sent through the *q_MQTT_periodic_sender* work queue of Communication module.

In summary, through the *q_MQTT_receiver* work queue the *Status and error manager* process will receive messages about the state of the connection with the MQTT broker and all the payload of the messages linked to the *order* topic received. Through the work queue *q_MQTT_sender* the payload of the messages that should be sent linked to the *info* topic will be sent. Through the work queue q_MQTT_periodic_sender the *Status and error manager* process will order how the periodic messages should be sent through the battery and location topic.

### 3.6.2. CAN communication of Communication module

In order to be able to send messages via CAN with the acquired *Humming Boards CBi*, their CAN interfaces must be enabled. The procedure for enabling the CAN interface on a *Humming Board CBi* is explained in Annex IV. Once we have the CAN interface of the module correctly configured, we must have access to it through the software programmed in *Python 3*. The *python-can* library [60] is used to achieve this goal. This library offers a set of functions to send and receive messages through the CAN interface. It also allows to configure periodic messages, automatically generating threads that send the messages periodically. In addition, it allows to establish filters in the reception of the messages in order to only receive the messages of interest. In order to implement the ISO-TP transport layer explained in section 3.3 (necessary to send the GOTO message) the can-isotp library for *Python 3* [61] is used, which simplifies the use of this protocol. With this library it is possible to specify in a function the message to be sent. The library automatically fragments and sends the messages, considering the reception of the flow control messages from the receiver.

As shown in Figure 32, there are two processes that handle the CAN communication of the module: *CAN receiver* and *CAN sender*. The tasks of each of them will be explained below, as well as the messages used in the corresponding work queues.

The ***CAN receiver*** process is responsible for receiving and filtering the messages from the CAN bus. It has assigned the *q_CAN_receiver* work queue that is used to send information from this process to the *Status and errors manager* process. Table 21 shows all the possible messages that can be sent through this work queue. The process has two operating-modes that are configured using an argument passed at the launch of the process: a normal mode and a power-saving mode.

The power-saving mode is used in the Standby state of the system. In this mode the process only receives the messages that inform about the battery status, extracts the percentage of battery charge, and updates the value of the shared global variable BATTERYdata.

In the normal mode of the process, when it is launched, it accesses the CAN interface, configures a reception filter to receive only the messages that the Communication module must receive, and launches the *CAN sender* process. After this, it waits to receive the first CONT-STATUS message via the CAN bus. If it does not receive it after a certain time, it sends an error message through the work queue indicating that the Control module is not responding. If it receives it, it sends an OK message through the work queue indicating that the Control module has correctly responded to the messages of the Communication module. Once the initial tasks have been completed, the process is responsible for receiving and interpreting all messages received through the CAN Bus. Depending on the one received, it will act in one way or another:

- Reception of the CONT-STATUS frame: The signals are extracted from the message. If the STATE-CON, PAUSE-CON, or TIMEOUT signals have changed with respect to signals received in the previous message, this is reported through the work queue. If the GOTO-ACK signal has the value 1 when before it had the value 0, a command is sent through the *q_CAN_sender* work queue to inform the *CAN sender* process that the ACK has been received and that it should not retry sending the message. A timer is restarted each time the CONT-STATUS message is received, so if it is not received for a while, an error is triggered.

- Reception of the RFID and CON-ERR frame: Through the *q_CAN_receiver* work queue the received message is reported. Through the *q_CAN_sender* work queue, the *CAN sender* process is ordered to acknowledge the message by putting at 1 the RFID-ACK signal or the CON-ERR-ACK signal. A timer is established so that, after half a second, another order is sent to the CAN sender process indicating that it should put at 0 the RFID-ACK signal or the CON-ERR-ACK signal.
- Reception of the battery status frame sent by the subsystems of the vehicle. The percentage of battery charge is extracted from the message and the *BATTERYdata* shared global variable is updated with the new value.

| Work queue | Message | Explanation |
|---|---|---|
| **q_CAN_receiver** | OK | Connection with Control module established |
| **q_CAN_receiver** | S0 | STATE-CON signal=00 (Start Up) |
| **q_CAN_receiver** | S1 | STATE-CON signal=01 (Normal mode) |
| **q_CAN_receiver** | S2 | STATE-CON signal=10 (Autonomous mode) |
| **q_CAN_receiver** | S3 | STATE-CON signal=11 (Stand By) |
| **q_CAN_receiver** | P0 | PAUSE-CON signal=0 |
| **q_CAN_receiver** | P1 | PAUSE-CON signal=1 |
| **q_CAN_receiver** | T0 | TIMEOUT signal=0 |
| **q_CAN_receiver** | T1 | TIMEOUT signal=1 |
| **q_CAN_receiver** | R (ID) | RFID frame received |
| **q_CAN_receiver** | W (error code) [(attribute)] | Triggering a *warning* type failure |
| **q_CAN_receiver** | E (error code) [(attribute)] | Triggering an *error* type failure |

Table 21. Messages sent through the *q_CAN_receiver* work queue of Communication module.

The ***CAN sender*** process is responsible for sending CAN messages from the Communication module via the CAN bus. This process is controlled through the *q_CAN_sender* work queue. Using this queue, the process can be ordered to send certain messages or to modify the value of the periodic message signals. Table 22 shows all the possible orders that can be sent through this work queue.

| Work queue | Message | Explanation |
|---|---|---|
| **q_CAN_sender** | S0 | STATE-COM signal=00 |
| **q_CAN_sender** | S1 | STATE-COM signal=01 |
| **q_CAN_sender** | S2 | STATE-COM signal=10 |
| **q_CAN_sender** | S3 | STATE-COM signal=11 |
| **q_CAN_sender** | P0 | PAUSE-COM signal=0 |
| **q_CAN_sender** | P1 | PAUSE-COM signal=1 |
| **q_CAN_sender** | R0 | RFID-ACK signal=0 |
| **q_CAN_sender** | R1 | RFID-ACK signal=1 |
| **q_CAN_sender** | E0 | CON-ERR-ACK signal=0 |
| **q_CAN_sender** | E1 | CON-ERR-ACK signal=1 |
| **q_CAN_sender** | G0 | Stop resending GOTO message through ISO-TP |
| **q_CAN_sender** | G1 (goto chain) | Send GOTO message through ISO-TP |
| **q_CAN_sender** | TERMINATE | Stop process |

Table 22. Messages sent through the *q_CAN_sender* work queue of Communication module.

When this process is launched, it generates a thread (with the *python-can* library) to start sending the periodic COMM-STATUS message. It is sent with STATE-COM signal indicating the Start Up state, and PAUSE-COM, RFID-ACK and CON-ERR-ACK with the value 0. From this moment on, it remains waiting for orders, listening in the *q_CAN_sender* queue. Depending on the command received (Table 22), it will change the value of the signals, or send the GOTO message with the ISO-TP transport protocol (with the help of the *can-isotp* library). In the case of sending the chained GOTO message, the process will wait for a confirmation that the message has been received, that is, through the GOTO-ACK signal sent by the Control module. As we have seen before, the *CAN receiver* process, when detecting the reception of this signal with the value 1, sends a message to the *CAN sender*

process through the *q_CAN_sender* work queue. If the CAN sender process does not receive such confirmation, it will retry sending the message via ISO-TP up to 5 times. If it continues not receiving the confirmation, it will send an error through the *q_CAN_receiver* work queue.

### 3.6.3. GPS data reception

As seen above, the used GPS receiver is the Garmin GPS18x USB. It sends information via Garmin's private binary protocol. To enable the software programmed in *Python 3* to receive coordinates from the GPS, detect lack of signal, or detect module disconnection, the *gpsd* daemon has been used.

The ***gpsd*** software is a service daemon that serves to monitor the information received by a GPS receiver via USB or RS232, and make it available to be queried on TCP port 2947 [62]. This daemon is capable of handling most of standards (such as NMEA) and proprietary protocols (such as the Garmin binary protocol). The daemon hides the differences between the different GPS receivers, converting the data received into a common format (information in JSON format organized in different classes), which can be interpreted by a program. In this way it can be used as an intermediary, so that applications avoid connecting to a serial device and interpreting the information.

Once the gpsd daemon is installed, we configure it to receive data from the USB-connected GPS receiver using the following command: `gpsd /dev/ttyUSB0 -F /var/run/gpsd.sock`. The installation of this daemon also installs the libraries needed to access the information provided by the daemon on the TCP port 2947 from a software programmed in *Python 3*.

The ***GPS receiver*** process is the process responsible for receiving the information provided by the *gpsd* daemon. This process receives all the messages sent by the daemon, inserts the coordinates in the shared global variable *GPSdata* and triggers *errors* and *warnings* through the *q_GPS_receiver* work queue. Table 23 shows the messages that can be sent through this queue.

The information given by the *gpsd* daemon is in JSON format, structured in objects with attributes [63]. When this process is launched, it starts (or restarts) the *gpsd* daemon and checks the reception of two types of object:

- "DEVICE" objects: These objects give the information about the GPS receivers connected and detected by the *gpsd* daemon. The object is sent once at the start of the daemon, and every time there is a change, that is, the GPS receiver is connected or disconnected. Thanks to this object, the *GPS receiver* process knows when the GPS receiver is connected or not and can trigger errors if the receiver is disconnected.
- "TPV" objects: These objects give a time-position-velocity report. That is, they have attributes that give the current GPS time, coordinates, and estimated speed. Thanks to these objects, the *GPS receiver* process is able to obtain the coordinates and update the shared global variable *GPSdata*. This object is received each time the GPS receiver gives new information. The GPS receiver used has a 1-second update rate, therefore, the *GPS receiver* process, using a timer that starts when the process is started and restarts each time one of these objects is received, can detect if the receiver has no signal, since, in this case, no "TPV" objects are received.

| Work queue | Message | Explanation |
|---|---|---|
| **q_GPS_receiver** | W (error code) (attribute) | Triggering a *warning* type failure |
| **q_GPS_receiver** | E (error code) (attribute) | Triggering an *error* type failure |

Table 23. Messages sent through the *q_GPS_receiver* work queue of Communication module.

### 3.6.4. Status, process, and error management of Communication module

The ***Status and error manager*** process is in charge of launching or stopping the rest of the processes; and managing the status[5] and errors of the Communication module based on the information provided by the rest of the processes and giving new orders to them, through the work queues mentioned above.

Through the *q_MQTT_receiver* (Table 18), *q_CAN_receiver* (Table 21), and *q_GPS_receiver* (Table 23), the process obtains information about the status of the MQTT communication and the orders received from the back-end, the status of the communication through the CAN bus and the signals sent by the Control module, and the status of the GPS receiver. On the other hand, using the *q_MQTT_sender* (Table 19), *q_MQTT_periodic_sender* (Table 20) and *q_CAN_sender* (Table 22) work queues, the process can send orders to send messages to the back-end, modify the frequency and number of periodic parameters sent to the back-end, and modify periodic message signals or send new messages through the CAN bus.

This process starts when the system is turned on. When launched, it defines the working queues and a series of auxiliary variables necessary for its proper functioning. The *state* variable defines the state of the vehicle system. It can take values from 0 to 4 to be able to refer to all possible system states. The initial value of this variable is the one referred to the Start Up status. Then, using a while loop that includes a switch statement that depends on this variable, the process acts depending on the state it is in. The following paragraphs show what the process performs in each of these states. Note that, in the next paragraphs, when it is mentioned that the state is changed, the value of the *state* variable is changed and a break is executed so that a new iteration of the while loop is performed and, therefore, the code of the new state is executed.

The following paragraphs contain very precise information. For their correct understanding it is important to know how the rest of the processes work and the messages that can be sent through the work queues. Annex V shows all the messages that can be sent through the different work queues. In addition, a look at the use cases explained in chapter 5 is also recommended.

In the **Start Up** state the process launches the *MQTT receiver* process and waits to receive a message through the *q_MQTT_receiver* queue. If an OK is received (connection to the MQTT broker established), the order to send the CONNECT message is sent through the *q_MQTT_sender* queue and the reception of the CONNECTED message through the *q_MQTT_receiver* queue is expected. Once received, and therefore confirmed the correct communication with the back-end, the *GPS receiver* process and the *CAN receiver* process are launched and the reception of a message through the *q_CAN_receiver* queue is expected. If an OK is received, the process confirms that the Control module is connected and turned on, and goes into the next default state (Standby, Normal mode or Autonomous mode, depending on the pre-set value). If at any point in this procedure an error message is received instead of the expected one or nothing is received for a while, the process immediately goes into Failure mode.

In the **Normal mode**, the process sends the S1 message (indicate in the CON-MOD signal the normal mode) through the *q_CAN_sender* queue. It then listens only to the ***q_CAN_receiver*** queue. If *error* messages are received, the process goes into Failure mode. If *warning* messages are received, the order to send a WRN message is sent through the *q_MQTT_sender* queue. If an RFID message is received, an RFID message is sent through the *q_MQTT_sender* queue. If an S1 message is received it means that the Control module has changed its status and, therefore, the status has been correctly

[5] The states the vehicle system can be in are explained in section 3.2.

established in the vehicle system. Hence, an order to send the AM-OFF OK message to the back-end is sent through the *q_MQTT_sender* queue. If the message S1 is not received for a while, an *error* is triggered, and the process goes into the Failure mode. Once the status has been established, the process starts listening in the ***q_MQTT_receiver*** queue too[6]. Through this queue, if *error* messages are received, the process goes into Failure mode. If *warning* messages are received, the order to send a WRN message is sent through the *q_MQTT_sender* queue. If, through the *q_MQTT_receiver* queue, a message is received informing about the reception of the AM-ON or STANDBY message, the process goes to the Autonomous mode or to the Standby status, respectively. If any other type of message is received through this queue, the order to send a WRN message is sent through the *q_MQTT_sender* queue, reporting that an unexpected message has been received.

In the **Autonomous mode**, the process sends the S2 message (indicate in the CON-MOD signal the Autonomous mode) through the *q_CAN_sender* queue. Then, in a similar way to that explained for the normal mode, it listens only to the ***q_CAN_receiver*** queue. This, listening in this queue, performs the same operations that were performed in the Normal mode but may receive some new messages. If it receives a T1, P1, or P0 message, it sends a command through the *q_MQTT_sender* queue so that the TIMEOUT, PAUSE OK, or CONTINUE OK messages are sent to the back-end. In addition, in this case, the status is considered established when the S2 message is received through the *q_CAN_receiver* queue. When the status is established, it starts to listen in the ***q_MQTT_receiver*** queue too. In this queue, apart from the operations that were already carried out in the Normal mode, others are carried out according to new messages. If a report is received indicating that a PAUSE or CONTINUE message has been received, the P1 or P0 order is sent through the *q_CAN_sender* queue and an ACK timer is activated. The timer is cancelled once the message P1 or P0 is received through the queue *q_CAN_receiver*. If the timer expires, an *error* is triggered, and the process goes into Failure mode. On the other hand, if a report is received indicating that a GOTO message has been received, this GOTO message is sent to the *q_CAN_sender* queue. If, through the *q_MQTT_receiver* queue, a message is received informing about the reception of the AM-OFF or STANDBY message, the process goes into the Normal mode or the Standby status.

In the **Standby** state, the process sends the S3 message (indicate in the CON-MOD signal the Standby state) through the *q_CAN_sender* queue and waits to receive the S3 message through the *q_CAN_receiver* queue. If it does not receive it after a while, or receives an error message, it immediately goes into Failure mode. If it receives the S3 message, the status is correctly established. Then the *GPS receiver* process and the *CAN sender* process are terminated, and the *CAN receiver* process is restarted in low power mode. Once this is done, the order to send the STANDBY OK message to the back-end is sent through the *q_MQTT_sender* queue. From this moment on, the process listens simultaneously in the *q_MQTT_receiver* and *q_CAN_receiver* queues for any *error* messages, warning messages or a report informing about the reception of the AM-ON or AM-OFF message. In this last case, the system would go into the Start Up state in order to restart all the necessary processes. Moreover, there is a variable that defines the state the system should go after the Start Up state. This variable will be configured according to the message received.

In the **Failure mode**, the error code, and the attribute of the *error* (if any) that caused the entry in this state are collected. The *CAN sender* and *CAN receiver* processes are terminated and the connection with the MQTT broker is checked (a variable is set to 1 when the connection with the MQTT broker is established and is set to 0 if there is no connection or the connection is lost). If the connection is established, an order is sent, through the queue *q_MQTT_sender*, to send an ERR message to the

[6] Note that, from this point on, the process is listening to both queues simultaneously. This is done to avoid performing new tasks ordered by the back-end without ensuring that the status has been correctly established.

back-end informing about the error. Then it waits to receive a message through the queue *q_MQTT_receiver* that informs that the RESTART message has been received. Once the RESTART message is received, all processes are terminated, and the process enters the Start Up state. If there is no connection with the MQTT broker, it waits 10 seconds and perform the same procedure.

Note that in Normal mode and Autonomous mode, the process also listens in the *q_GPS_receiver* queue in order to receive *warning* or *error* messages from it. In these cases, the process does the same as it does with the *warning* or *error* messages that can be received through the rest of the work queues. In addition, depending on the system status, the process enables or disables certain GPIO pins that are connected to the information board LEDs.

## 3.7. Control module

The Control module performs the tasks of the vehicle control. Based on the route received via the chained message GOTO and, with the help of the sensors installed in the vehicle, it controls the actuators installed in order to follow the marked route. Through CAN it can receive orders from the Communication module to stop the vehicle or to stop controlling it.

The implementation of the Control module software is very similar to that of the Communication module (section 3.6). The CAN interface must be enabled, and *Python 3* is used for programming. A series of processes are used, that communicate with each other through work queues and shared global variables to work together. In the Control module, not all the processes required for its full operation have been implemented. As said before, the implementation of the sensors and actuators for vehicle control are not part of this final degree project as the necessary material is not yet available. Therefore, the processes related to the control of the vehicle are also not part of it and have not been implemented. The implementation of these processes will be done during the development of objective 4 of the TwizyLine joint project explained in section 1.1. In this final degree project, the processes of the Control module related to its communication and synchronization with the Communication module through the CAN bus are implemented.

Figure 33 shows a diagram of all the processes and threads that run in the Control module, as well as the working queues used to carry out communication between them. Shared global variables have not been used to implement the CAN communication tasks of the module, but it is proposed to use them for the implementation of the rest of the processes. Note that this section will have many references to the implementation of the Communication module explained in section 3.6. It is recommended to read that section.

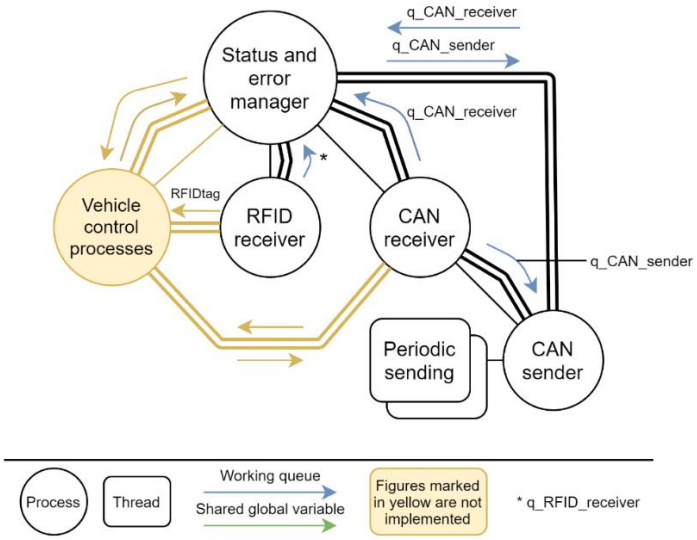



Figure 33. Diagram of the processes that are executed in the Control module.

As seen in Figure 33, the *CAN receiver* and *CAN sender* processes for the CAN communication of the module follow the same structure as those of the Communication module. They use too two work queues called *q_CAN_receiver* and *q_CAN_sender*. There is another process called *RFID receiver* that receives the information given by the RFID sensor. The *Status and error manager* process it is the main process. It is in charge of launching the rest of the processes and receiving and sending orders to them through the work queues according to the status or possible errors.

The following subsections will explain how the processes responsible for CAN communication of the module (section 3.7.1), the process responsible for reading RFID tags (3.7.2) and the main process Status and error manager (3.7.3) work.

### 3.7.1. CAN communication of Control module

To carry out the CAN communication of this module, the same initial steps carried out in the Communication module are followed. In summary, the device tree of the module must be modified to enable the CAN interface (explained in Annex IV) and the *python-can* and *can-isotp* libraries for *Python 3* are used to be able to send and receive messages through CAN and, when necessary, through the ISO-TP transport layer.

Similar to the Communication module, two processes are used for CAN communication. These two processes are explained below, detailing the differences between them and their equivalents in the Communication module.

The ***CAN receiver*** process is responsible for interpreting the CAN messages received via the module's CAN interface. It informs the *Status and error manager* main process about the received messages and the possible *warnings* and *errors* through the *q_CAN_receiver* work queue. In the Table 24 the messages that can be sent through this queue are shown.

When this process is launched, it launches a thread that takes care of the reception of the messages encapsulated in the ISO-TP transport layer, and establishes a filter to interpret only the messages that must be received. It then waits to receive a COMM-STATUS message from the Communication module. Once it receives that message, it sends a CSR message (COMM-STATUS received) through the queue to indicate that the Communication module has been turned on, and launches the *q_CAN_sender* process. From this point on, the process interprets all CONN-STATUS messages and all GOTO messages.

- If a COMM-STATUS message is received, the process extracts all signals. If STATE-COM or PAUSE-COM signals changes at any time, this is reported through the work queue. If the RFID-ACK or the CON-ERR-ACK signals changes to 1, the *CAN sender* process is ordered through the *q_CAN_sender* work queue to stop sending RFID and CON-ERR messages. A timer is restarted each time this message is received, so if it is not received for a while, an *error* is triggered.
- If a GOTO message is correctly received through ISO-TP, the content of this message is sent through the *q_CAN_receiver* queue and the *CAN sender* process is ordered through the *q_CAN_sender* work queue to set the GOTO-ACK signal to 1. After half a second, with the help of a timer, the *CAN sender* process is ordered to set the GOTO-ACK signal to 0 through the *q_CAN_sender* queue as well.

| Work queue | Message | Explanation |
|---|---|---|
| **q_CAN_receiver** | CSR | First COMM-STATUS frame received |
| **q_CAN_receiver** | S0 | STATE-COM signal=00 (Start Up) |
| **q_CAN_receiver** | S1 | STATE-COM signal=01 (Normal mode) |
| **q_CAN_receiver** | S2 | STATE-COM signal=10 (Autonomous mode) |
| **q_CAN_receiver** | S3 | STATE-COM signal=11 (Stand By) |
| **q_CAN_receiver** | P0 | PAUSE-COM signal=0 |
| **q_CAN_receiver** | P1 | PAUSE-COM signal=1 |
| **q_CAN_receiver** | G (goto data) | GOTO ISO-TP message received |
| **q_CAN_receiver** | W (error code) [(attribute)] | Triggering a *warning* type error |
| **q_CAN_receiver** | E (error code) [(attribute)] | Triggering an *error* type error |

Table 24. Messages sent through the *q_CAN_receiver* work queue of Control module.

The ***CAN sender*** process is responsible for sending CAN messages. Through the *q_CAN_sender* work queue receives orders and, according to these, changes signals of the periodic messages or sends new messages. In the Table 25 the messages that can be sent through this queue are shown.

When this process is launched, it waits to receive the READY message through the *q_CAN_sender* queue. Once it receives that message, it starts sending the periodic CONT-STATUS message. From this moment on, depending on the orders received through the *q_CAN_sender* queue, the value of the CONT-STATUS message signals changes and the sending of the periodic RFID or CON-ERR messages starts or ends. In the case of receiving the order through the *q_CAN_sender* queue to send the RFID message or the CON-ERR message, the *CAN sender* process starts the periodic sending of this message and sets a half-second timer, so that, if it does not receive in time the order to stop sending this message, it launches an *error*. This is because, when the Control module starts sending these messages, it expects to receive an ACK before half a second and, as seen above, the *CAN receiver* process, when it receives this ACK, sends an order to the *CAN sender* process to stop sending the message. Errors triggered by this process are sent to the *Status and error manager* main process through the *q_CAN_receiver* queue.

| Work queue | Message | Explanation |
|---|---|---|
| **q_CAN_sender** | READY | Start sending periodic CONT-STATUS frame |
| **q_CAN_sender** | S0 | STATE-CON signal=00 |
| **q_CAN_sender** | S1 | STATE-CON signal=01 |
| **q_CAN_sender** | S2 | STATE-CON signal =10 |
| **q_CAN_sender** | S3 | STATE-CON signal =11 |
| **q_CAN_sender** | P0 | PAUSE-CON signal =0 |
| **q_CAN_sender** | P1 | PAUSE-CON signal =1 |
| **q_CAN_sender** | T0 | TIMEOUT signal =0 |
| **q_CAN_sender** | T1 | TIMEOUT signal =1 |
| **q_CAN_sender** | G0 | GOTO-ACK signal=0 |
| **q_CAN_sender** | G1 | GOTO-ACK signal=1 |
| **q_CAN_sender** | R0 | Stop periodic RFID message |
| **q_CAN_sender** | R1 (RFID) | Start periodic RFID message |
| **q_CAN_sender** | E0 | Stop periodic CON-ERR message |
| **q_CAN_sender** | E1 (error code) [(error info)] | Start periodic CON-ERR message. *Error* type failure. |
| **q_CAN_sender** | W1 (error code) [(error info)] | Start periodic CON-ERR message. *Warning* type failure. |
| **q_CAN_sender** | TERMINATE | Stop process |

Table 25. Messages sent through the *q_CAN_sender* work queue of Control module.

### 3.7.2. RFID tag reading

As seen in section 3.1.3, the ID-20LA RFID antenna and a RFID USB adapter are used to read the RFID tags. When connected to the module, the operating system interprets it as a serial port and through the *pySerial* library for *Python 3* [64] it is possible to receive the messages sent by the reader. Figure 23 shows the message format.

The ***RFID receiver*** process receives this message from the RFID antenna, checks if the message is valid with the checksum and sends the identifier of the tag detected through the *q_RFID_receiver* queue. It is considered that, when the rest of the vehicle control processes are implemented, a shared global variable should be implemented that would always contain the identifier of the last tag detected. In this way, the processes in charge of controlling the vehicle will always be able to know what is the last RFID tag detected and therefore follow the route correctly.

### 3.7.3. Status, process, and error management of Control module

The main process *Status and errors manager* is in charge of starting and stopping the rest of the processes, of managing the state in which the system is, and of handling the errors that any process can trigger. To carry out these tasks it can receive information from the processes with the queues *q_CAN_receiver* and *q_RFID_receiver*, and can send orders to them through the queue *q_CAN_sender*. The method followed to program it is very similar to that used in its equivalent process in the Communication module explained in section 3.6.4. Note that the only functionality currently implemented in the Control module is to receive and respond to orders sent by the Communication module and to send RFID and CON-ERR messages to it.

In the **Start Up** state, the *CAN receiver* process is launched and waits to receive the CSR message (COMM-STATUS message received) through the *q_CAN_receiver* queue. Once received, all devices connected to the module will be turned on and the connection to them will be checked (not implemented). When this is done, the READY message is sent through the *q_CAN_sender* queue so that the module's periodic message starts being sent. From this moment the process waits to receive another message from the queue *q_CAN_receiver*. Depending on the message received by this queue, different tasks can be performed: the status may change, the process can go into the failure mode or an order can be sent through the *q_CAN_sender* work queue to send a CON-ERR message of *warning* type.

In **Normal mode** and **Autonomous mode**, using the *q_CAN_sender* queue, an order is given to set the CON-STATUS signal indicating the new status. Then the RFID receiver process is launched (if it was not launched before), and the main process kept listening in the queues *q_CAN_receiver* and *q_RFID_receiver* simultaneously. In the case of receiving status change orders through the *q_CAN_receiver* queue, the status is changed. In the case of receiving reports of detection of an RFID tag through the *q_RFID_receiver* queue, an order to send an RFID message is sent through the *q_CAN_sender* queue. In the case of receiving a *warning* from any of the processes, the order to send a CON-ERR message of warning type is sent through the *q_CAN_sender* queue. If an *error* type failure is received from any of the processes, the main process goes to the failure mode.

In **Standby** status, the S3 order (indicate in the CON-STATUS signal the Standby state) is sent through the *q_CAN_sender* queue. Then, the main process waits for an *error* message through the *q_CAN_receiver* queue, indicating that the periodic COMM-STATUS message has stopped being received. Once received, all devices are turned off (not implemented) and all processes are terminated. The Control module is then immediately put into the Start Up state, where it waits for a COMM-STATUS message to be received again to restart all processes and turn the devices on.

In the **Failure mode**, the error code and its attribute (if any) are collected. An E1 message is sent through the *q_CAN_sender* queue so it starts sending the CON-ERR periodic message. After 5 seconds, all processes except the main process are terminated. The main process goes into the Start Up state.

# 4. Vehicle simulator for testing

This chapter explains how the second main objective of this final degree project shown in section 1.2 has been achieved. It presents the implementation of a simulator of the system implemented in chapter 3. This simulator, from the back-end point of view, behaves in the same way as the real vehicle system. The advantage of this simulator is to be able to simulate multiple vehicles simultaneously; to be able to simulate routes between car parks and RFID tag detection consistent with the TwizyLine service, and therefore expected by the back-end; and, consequently, to be able to check the correct integration of the vehicle systems with the back-end in different use cases. This chapter corresponds to phase 4 of this final degree project shown in section 1.3, and is complemented by chapter 6, which shows, once the simulator implementation has been completed, the evaluation of the use cases of the TwizyLine service with the back-end and the help of the simulator.

This chapter is divided into 3 sections. Section 4.1 explains the modifications made to the original code (analysed in sections 3.6 and 3.7), in order to simulate the behaviour of real systems in a computer from the point of view of the back-end. Section 4.2 makes an analysis of the programmed software "Vehicle launcher", which allows to launch multiple vehicles simultaneously with different parameters. Section 4.3 explains the implementation of a software that allows to see the status and position of all vehicles (real or simulated) operating in the service in real-time. This software will be very useful when evaluating the operation of the back-end (chapter 6) since it offers a very visual interface and, therefore, a faster evaluation.

## 4.1. Code adaptation of the real system

To implement the simulator, the software programmed in *Python 3* of the Communication module explained in section 3.6 is used. Some processes are removed, modified and added to avoid the use of the Control module code and to simulate the real behaviour of a vehicle based on initial parameters and some files that define the car parks, RFID tags and possible routes between car parks.

Figure 34 shows the process diagram of a vehicle simulator, comparing it with the original processes of the Communication module. As we can see, the processes related to CAN communication and GPS data reception have been removed, the processes *Status and error manager* and *MQTT receiver* have been modified, and two new processes with new work queues and shared global variables have been added. Thanks to the modularity of the previously programmed software, the modifications have been made in a simple and orderly way.

The following subsections show the modifications made in the original processes (subsection 4.1.1), the structure of the files that define the possible routes to be followed by a vehicle and the RFID tags of the parking (subsection 4.1.2) , and the implementation of the new processes *Simulator* and *Battery Sim* (subsection 4.1.3) .

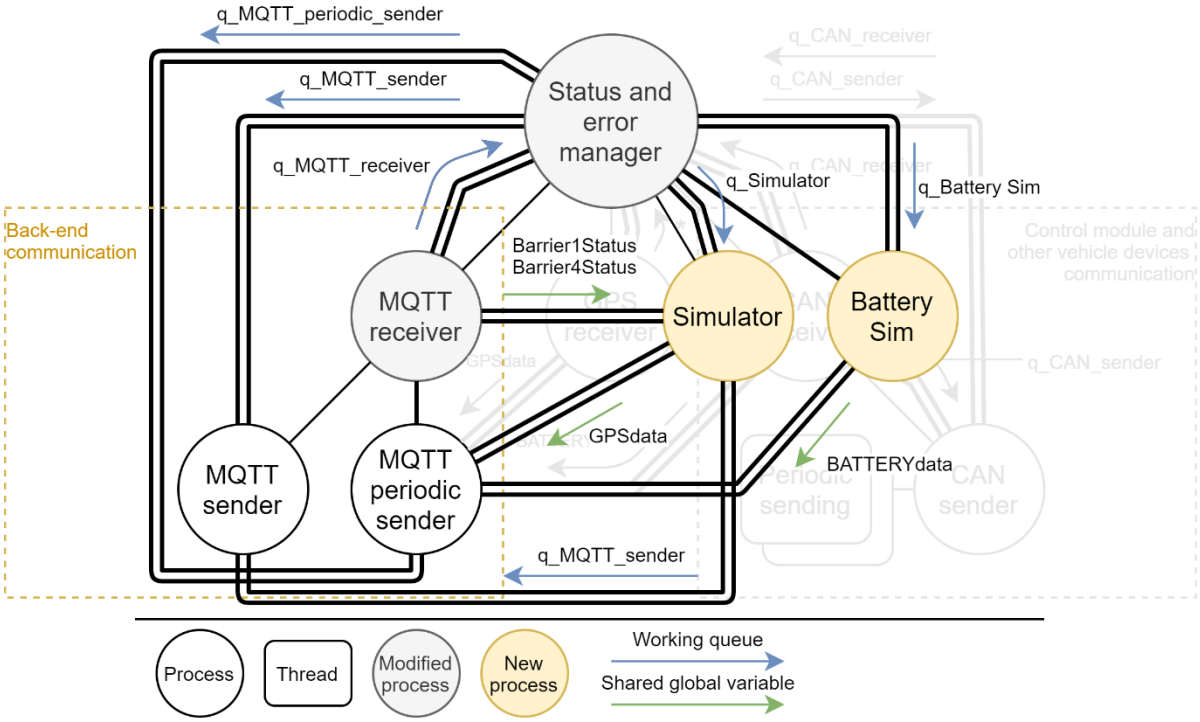



Figure 34. Diagram of the processes that are executed in the simulation of a vehicle.

### 4.1.1. Modifications in the Communication module processes

The *CAN receiver* and *CAN sender* processes have been eliminated. CAN messaging will not be simulated in this simulator. The *GPS Receiver* process has also been removed as the coordinates will not be obtained from a real GPS receiver.

The main process ***Status and error manager*** has been slightly modified. The tasks of sending and receiving messages through the work queues q_CAN_sender and q_CAN_receiver have been eliminated, so the establishment of status is done instantly.

When this process is started, it obtains 4 parameters from the software input arguments:

- **Vehicle ID**, which forms the general vehicle MQTT topic (explained in section 3.4).
- **Register Plate**, which is sent in the CONNECT message, as explained in section 3.4.
- **Initial battery charge** of the vehicle, which will decrease except when the vehicle is parked. At those times, the vehicle would be charging and therefore the value increases.
- **Route**, which defines the initial parking lot to which the vehicle must go, and the following parking lots to which it must go once it leaves the previous one. This parameter is written as follows: `(park_id)-(park_id)-(park_id) …`

Then, it launches the *Simulator* and *Battery Sim* processes that will be explained later (subsection 4.1.3).

In the Standby state the process sends through the new *q_Battery_Sim* work queue the message CHARGE to indicate that the battery charge should be increased. In the rest of the states the process sends through this queue the message DISCHARGE to indicate that the battery charge must decrease.

When it enters in Normal mode, it sends the START message through the *q_Simulator* work queue to indicate to the *Simulator* process that the vehicle must start moving to reach the next parking indicated in the software input argument *Route*. When the process, in autonomous mode, receives a GOTO message, it resends its content through the queue *q_Simulator* so that the *Simulator* process

simulates the coordinates of the vehicle and the detection of some RFID inside the parking according to the row in which the vehicle is ordered to be parked.

The ***MQTT receiver*** process has also been slightly modified. If we pay attention to the small parking of the TwizyLine service (shown in section 2.3.1 and explained in detail in the final project degree by Samuel Pilar Arnanz [13]) we see that there are 4 barriers (Figure 35). When the simulated vehicles arrive at the first barrier (barrier 1 or entry barrier), if the barrier stays down they must wait for it to open, just as a real driver would do. The same applies to the last barrier (barrier 4 or exit barrier), the vehicle must remain waiting for the barrier to open before leaving and going to another car park. The back-end, as explained in the final project degree by Samuel Pilar Arnanz [13], sends messages through MQTT to these barriers to pull up or pull down. As the simulated vehicles must be aware of the state of these barriers, the *MQTT receiver* process must also subscribe to the topics of these barriers. For barriers 2 and 3 it is not necessary since the back-end opens these barriers automatically before sending the route (GOTO message) to the vehicle.

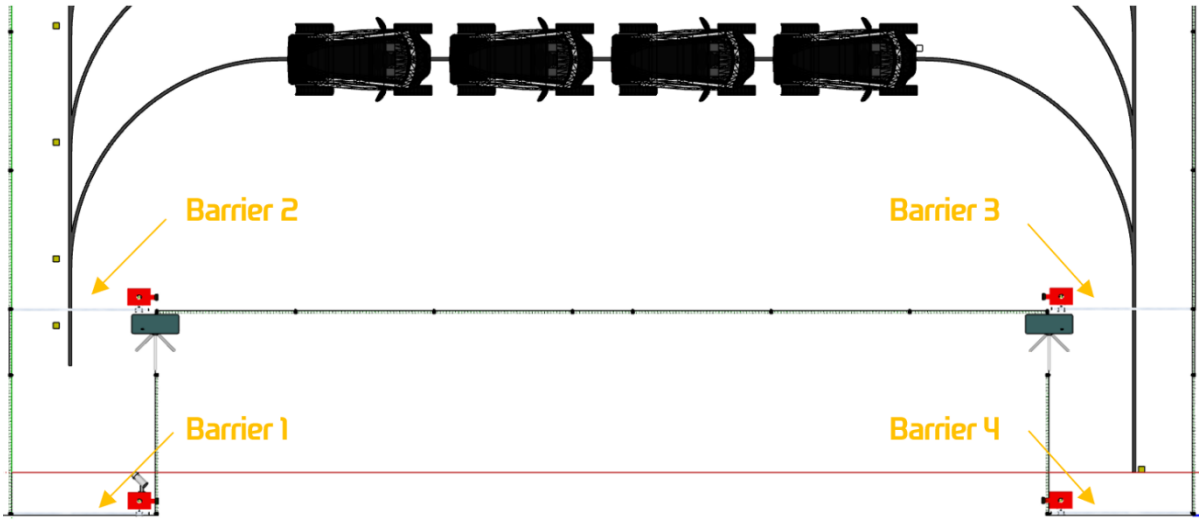



Figure 35. Barriers in the small TwizyLine car park.

Table 26 shows the messages and topics used by the back-end to order the barrier up or down. The *MQTT Receiver* process will subscribe to the topics of barriers 1 and 4 of all parking lots through which the vehicle will pass (defined in the software input argument *Route*).

| Publisher | Subscriber | Topic | Payload |
|---|---|---|---|
| Back-end | Barrier | barrier/(id_park)/(barrier_number) | B_UP |
| Back-end | Barrier | barrier/(id_park)/(barrier_number) | B_DOWN |

Table 26. MQTT messaging used by the back-end to control parking barriers.

Every time it receives a message linked to these topics, it will modify two global shared variables based on an array of boolean values that indicate the status of all barriers 1 (*Barrier1Status*) and all barriers 4 (*Barrier4Status*) through which the simulated vehicle will pass.

### 4.1.2. Route information and RFID tags files

The **routes** that a vehicle can take in the simulator are stored in KML files. KML (Keyhole Markup Language) is an XML notation developed by *Google* and standardized by the *OGC* (Open Geospatial Consortium) that is used to store geographic data [65]. This notation allows to store, among others, geographic routes. The simulator, to be able to simulate vehicles going to a car park or leaving it to go to others, needs to have access to the KML files of the corresponding routes. The following KML files are needed for each car park:

- **Initial route**: Indicates the GPS coordinates of the initial route to the first car park. Its name is `0-(park_id).kml` where park_id is the id of the car park.
- **Route to another car park**: Indicates the route a vehicle must follow to go to a car park from another one. There should be as many files as other car parks exist. Its name is `(origin_park_id)-(destination_park_id).kml`.

All these files must be inside a folder named `KML`. An example of these two files is shown in Figure 36.


Figure 36. Examples of initial route and route to another car park files for the simulator.

The **coordinates and RFID tags** that the simulated vehicle must report to the back-end once it is **inside a car park** are stored in text files that follow a certain structure. These files are organized by folders, so that each car park has its own folder (called by the id of the car park) where the files related to it are located. Each car park's folders must be found in another folder called `RFID`. There are two types of files:

- **Initial RFID tag and coordinates**: It indicates the identifier of the RFID tag located at the entrance to the car park and the coordinates of the entrance to this car park. Its name is always `1.txt` and there is one for each car park in the respective folder.
- **RFID tags and coordinates from entrance to end of row**: It indicates the coordinates and RFID tags to be reported by the simulated vehicle until it is parked. At the same time, it indicates in which coordinates a parking space exists. There is one file per parking row and the name of these is the id of the RFID tag located at the end of the corresponding row.

Figure 37 shows a visual example of how these files should be created. Each white square indicates an entry within the file. As explained above, there is always a file called 1.txt and then as many files as there are rows in the car park. These files are marked with yellow arrows in Figure 37, so that by following the arrow the different file entries can be obtained.

Annex VI shows an example of the file collection required for the operation of the simulator with three car parks.

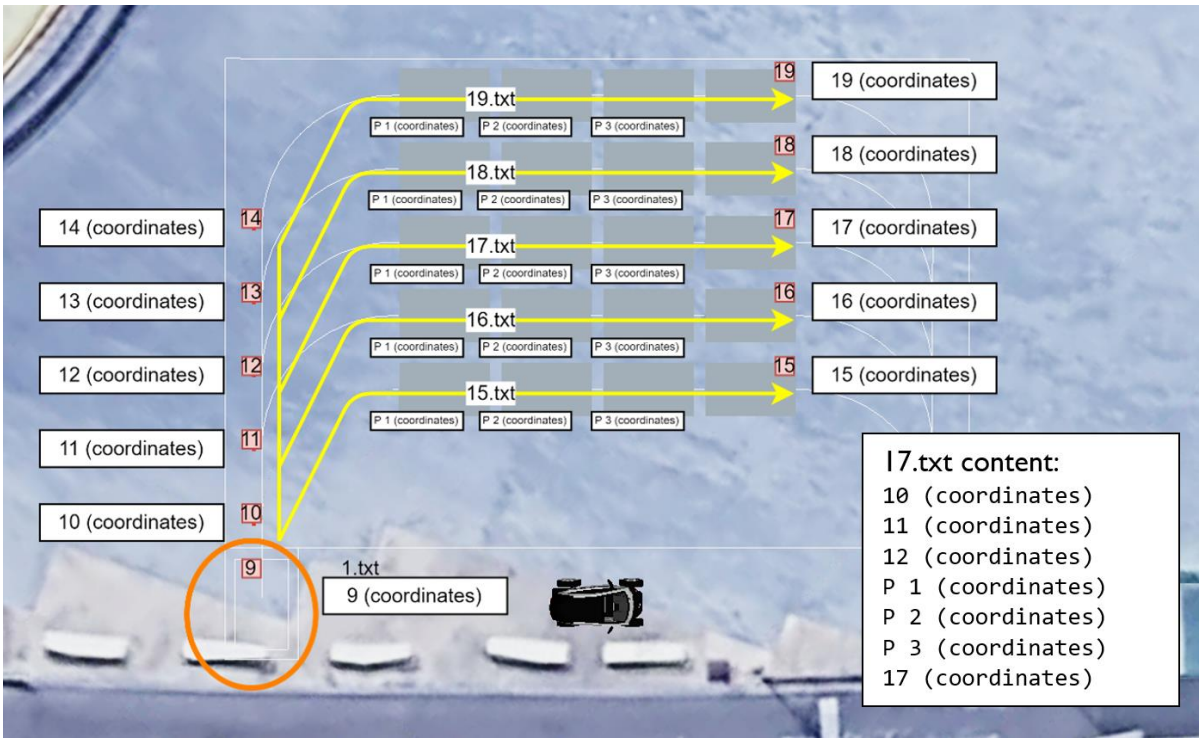



Figure 37. Example of the generation of coordinate files and RFID tags in a car park.

### 4.1.3. Battery Sim and Simulator processes

As explained above, two new processes have been programmed to make the simulator: *Battery Sim* and *Simulator*. The following paragraphs show an analysis of their operation.

The ***Battery Sim*** process simulates the battery charge and enters its value into the shared global variable *BATTERYdata*. It takes the initial value from the software input argument Initial battery charge. From this point on, every 10 seconds a unit is subtracted from the *BATTERYdata* variable. This process, at the same time, listens in the queue *q_Battery_Sim*. If it receives the CHARGE message it increases 1 unit to the variable every second, if it receives the DISCHARGE message it returns to the previous state.

The ***Simulator*** process simulates vehicle coordinates (inserting coordinates values in the shared global variable GPSdata) and RFID tags detection (sending orders to send RFID messages through the q_MQTT_sender work queue).

When this process is launched, it waits to receive the START message through the *q_Simulator* work queue. Once received, it checks the software input argument *Route*. Then it opens the initial route file of the first car park (`0-(park_id).kml`) that appears in the *Route* argument and updates the coordinates of the GPSdata variable every second with the coordinates of this file.

When the process finishes reading the file, and therefore the vehicle is in front of the entrance barrier of the first car park, the process checks, thanks to the shared global variable Barrier1Status, if the entrance barrier is open. If it is not, it waits until it is. If it is open, it accesses the initial tag and coordinates file (`1.txt`) of the car park, and reports the RFID tag and the coordinates that appear in the file. Then, the process waits to receive the content of the GOTO message received through the *q_Simulator* queue.

Once received, the program extracts the final RFID to which the vehicle must go, and opens the corresponding file of coordinates and RFID tags. For the correct operation of the simulator, the back-end also sends the parking column to which the vehicle must be directed via the GOTO message.

This way the *Simulator* process has all the necessary information and simulates the detection of RFID tags and the coordinates as they are in the file. Note that, if the vehicle is not dispatched at the end of the row, the simulator orders the sending of the TIMEOUT message through the *q_MQTT_sender* work queue once it is parked. This is done because in these cases the vehicle would detect another vehicle in front and will stop for this.

Then, the process waits for another GOTO message through the q_Simulator queue. Once it receives it there are two possible events.

- If the GOTO message indicates that the vehicle must go to the RFID of its same row, the vehicle moves to the next column of its row. This vehicle will not be the first one in the row, another vehicle has been ordered to leave the car park and therefore this vehicle will move forward and be placed in the next place.
- If the GOTO message indicates that the vehicle must go to a different RFID than the one in its row means that the vehicle must leave the parking lot. Note that in reality, in order for the vehicle to drive correctly to the exit of the parking lot, the GOTO message must be sent to it with the corresponding RFID. In this case, the simulator only checks that the RFID received is not that of the row in which the vehicle is parked to simplify the development. The process then, thanks to the argument *Route* received at program start, knows which is the next car park the vehicle must go to and opens the file that gives the route coordinates from the current car park to the next one (`(origin_park_id)-(destination_park_id).kml`). After that, the process reports the first coordinate from the file and reports the detection of the RFID tag expected by the back-end. Next, it waits to receive the START message through the *q_Simulator* queue (indicating that the vehicle has switched to Normal mode) and, afterwards, checks if the barrier 4 of the car park that is open (thanks to the global variable *Barrier4Status*). If it is not, it waits until it is. If it is, the route is continued, and the entire procedure is repeated with the next car park.

For information about the operation of the car park go to the final project degree of Samuel Pilar Arnanz [13].

## 4.2. Vehicle launcher

In order to launch multiple vehicles, a program capable of launching multiple instances of the software analysed in the previous section (section 4.1) is required. This program is called "Vehicle launcher". It offers a simple and intuitive user interface that allows the configuration of the parameters of each vehicle to be launched. It is programmed in *Python 3* and uses the *tkinter* library [66] to provide the user interface. Figure 38 shows its user interface.

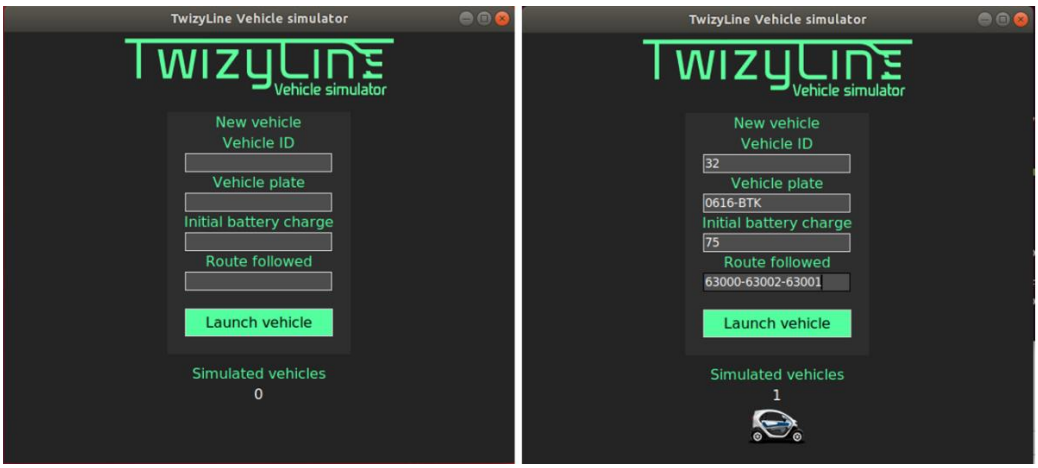



Figure 38. User interface of the Vehicles launcher software.

As can be seen in Figure 35, the user enters the input arguments for each simulated vehicle instance. By clicking on the "Launch vehicle" button, the program starts a new instance of the vehicle simulator in a new console by passing it the parameters introduced.

Figure 39 shows an example of what is shown when simulating 4 vehicles simultaneously. The window of the "Vehicles launcher" software remains active so that new vehicles can be launched at any time. Each vehicle is displayed in a different terminal through which information about the vehicle's status is shown.

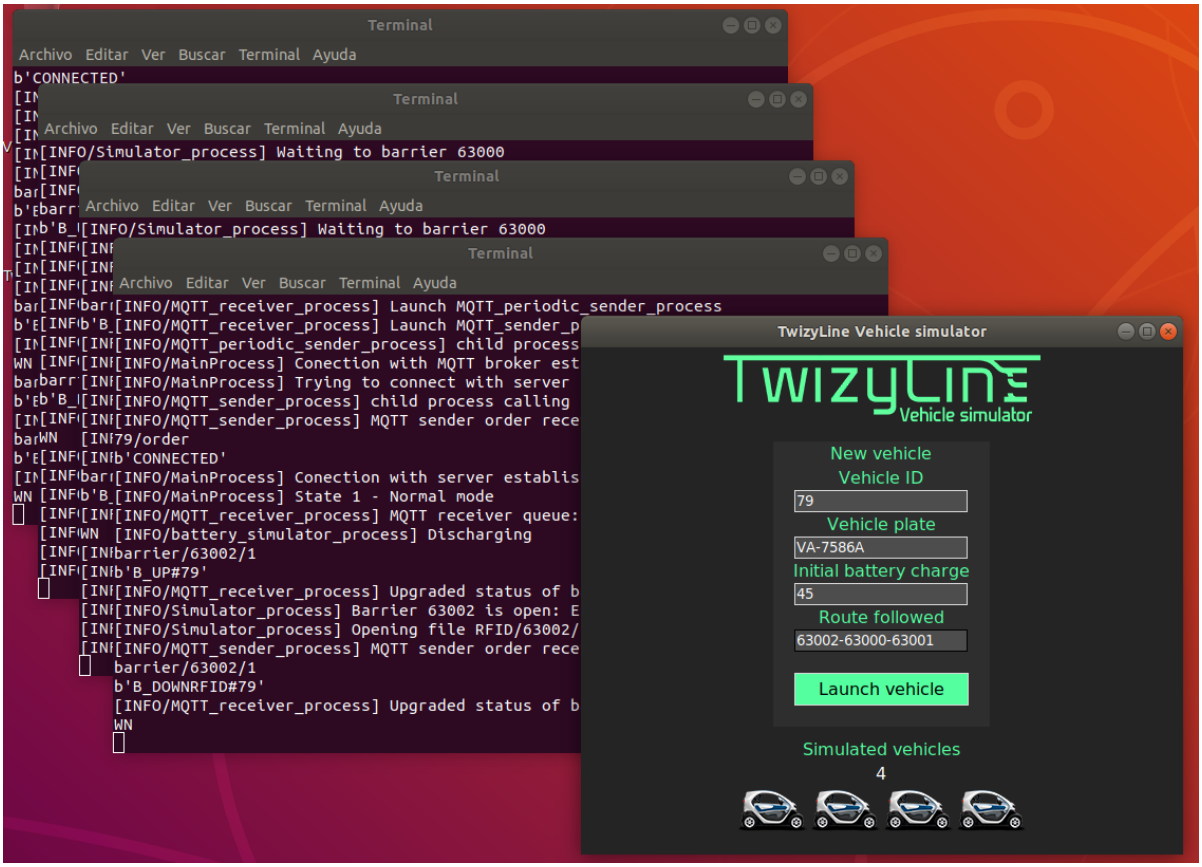



Figure 39. Example of simulation of 4 vehicles simultaneously with the simulator.

## 4.3. Real-time display of system status

New software has been programmed to implement a real-time display of the system status. The aim of this software is to keep a new KML file always updated with the information of all vehicles and barriers. Then, with a KML viewer (such as *Google Earth* [67]) it is possible to see its information in a visual way. Next, an analysis of the functioning of the software and the necessary KML files will be shown.

The new software has been programmed in *Python 3*, as most of the software of this final project degree. This software consists of a MQTT message sniffer for the *TwizyLine* project. The software uses the *paho-mqtt* library [59] (as the Communication module) to be able to subscribe and receive MQTT messages. This software subscribes to the *location* and *battery* topics of all vehicles (topics explained in section 3.4), and to the topics of all barriers (shown in Table 26). This way it can collect all the necessary information from the vehicles and the barriers in real-time. All this information is introduced in a KML file, so that when this file is opened with a KML viewer, all the vehicles and barriers are shown geopositioned. In the case of the barriers, the program has preconfigured the location of each one of them and changes its icon, configuring it in the KML file, depending if it is

opened or closed. Figure 40 shows the different icons configured by the program in the KML file, to be displayed by a KML viewer.

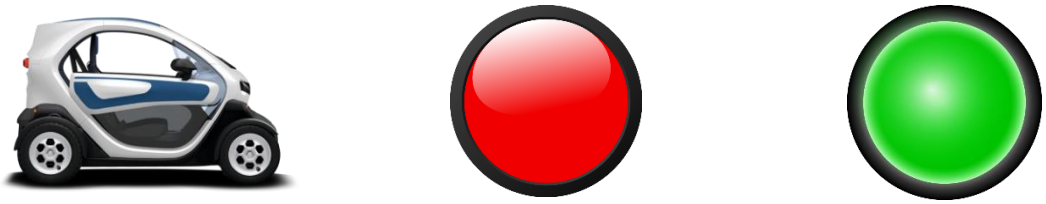



Figure 40. Icons used in the real-time display of system status.

Once we have a KML file that remains updated, it is necessary to make another KML file of the type *NetworkLink* [68] to indicate to the KML viewer that the first file should be opened periodically. Note that a KML viewer, if it opens directly the KML file offered by the software analysed above, will collect its data only at the time of opening and, therefore, the viewer will not update the data. KML files of the type *NetworkLink* solve this problem. It is possible to indicate the update rate of the target KML file. In this way, the KML viewer will open the *NetworkLink* KML file and, when it checks its information, will periodically show the information of the KML file generated by the previously analysed software.

Figure 41 shows an example of the real-time display of the system status using Google Earth as the KML viewer. As we can see, it is possible to see the battery status of all vehicles and their location, differentiated by their identifier. At the same time, we can see the status of all the barriers in the system. This tool is useful and effective to check the operation of the back-end.


Figure 41. Example of the real-time display of the system status using *Google Earth* as KML viewer.

# 5. Use cases in the real systems

This chapter will explain the use cases of the real system, that is, the system implemented in the Communication and Control modules. The exchange of messages that occurs in all possible situations between these modules and the back-end will be analysed, checking success cases and possible failures that may occur in practice. This chapter complements chapter 3 of this document, which explains the design and implementation of the system, and therefore serves to complete the first main objective of this final degree project shown in section 1.2. This chapter corresponds to phase 4 of the development of this project, shown in section 1.3.

## 5.1. Normal use

In this section, cases of use under normal conditions will be analysed, that is, success cases in which no anomalies occur. The following subsections explain the use case of the change to a new state, and the use cases that can occur in those states. It is important to know that, in the diagrams that are going to be shown, the blue squares indicate the different possibilities. Therefore, the diagram makes sense if only one blue square is considered, not all at once.

### 5.1.1. System Start Up

The Start Up state is the state in which all systems and the connection between modules and the back-end are checked. The system, when turned on, enters this state directly. Figure 31 shows the message exchange that takes place during this state. It also shows the exchange of messages performed to reach this state, either by switching on the modules or coming from another state.

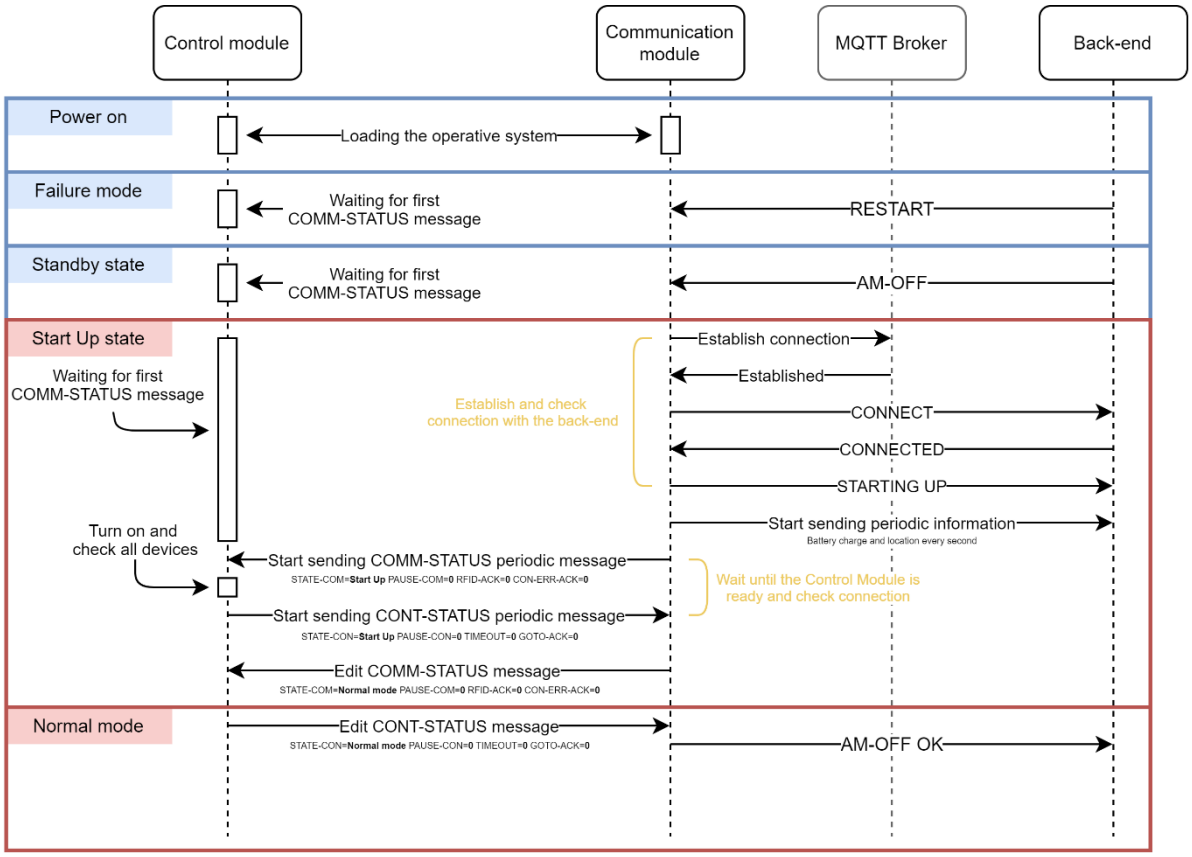



Figure 42. Message exchange between modules and back-end performed in the Start Up state.

When this state is entered, the Control module waits to receive the first STATE-COMM message from the communication module. Meanwhile, the Communication module tries to establish a connection with the MQTT broker and subscribe to the corresponding topics. Once this is achieved, it checks the connection with the back-end by sending it a CONNECT message. The back-end responds with a CONNECTED message. When this message is received, the communication module knows that the back-end is connected and therefore starts sending the periodic battery and location messages to the back-end. Once this is done, the module starts sending the periodic COMM-STATUS message with the STATE-COM signal indicating the Start Up state to wake up the control module, and waits to receive the CONN-STATUS message from it. The control module, on receiving the message, checks the connection with all the devices to which it is connected. If everything is correct, it sends the CONN-STATUS message with the STATE-CON signal indicating the Start Up state. Note that the Communication module does not wait indefinitely for a message from the Control module. If the Control module does not respond after 8 seconds, the Communication module interprets that it is not switched on or connected and therefore goes into Failure mode (see section 5.2.2).

Once these tasks have been performed, the Communication module orders the Control module to go into another state, by changing the STATE-COM signal to the value of the new state, and waiting for the STATE-CON signal from the Control module to change. The state the system goes into after the Start Up state depends on how the Start Up state was reached. If the previous state was the power on of the modules (into which the operating system of the modules is loaded), or the failure mode (in which the reception of a RESTART message sent by the back-end is expected to restart the system), the state that is passed to after the Start Up state is the pre-configured default state. This can be Normal mode (shown in Figure 42), Autonomous mode, or Standby. On the other hand, if the previous state was Standby, the state that is passed to after the Start Up state depends on the message sent by the back-end to wake up the system. If an AM-OFF is received (shown in Figure 42) it switches to Normal mode and if an AM-ON is received it switches to Autonomous mode.

### 5.1.2. Normal mode

The normal mode is the state the system is in when the vehicle driving tasks are performed by a conventional driver, as explained in section 3.2. Under normal conditions, that is, without anomalies, the control module only informs the communication module about the detection of an RFID tag. Figure 43 shows the message exchange performed to switch to Normal mode.

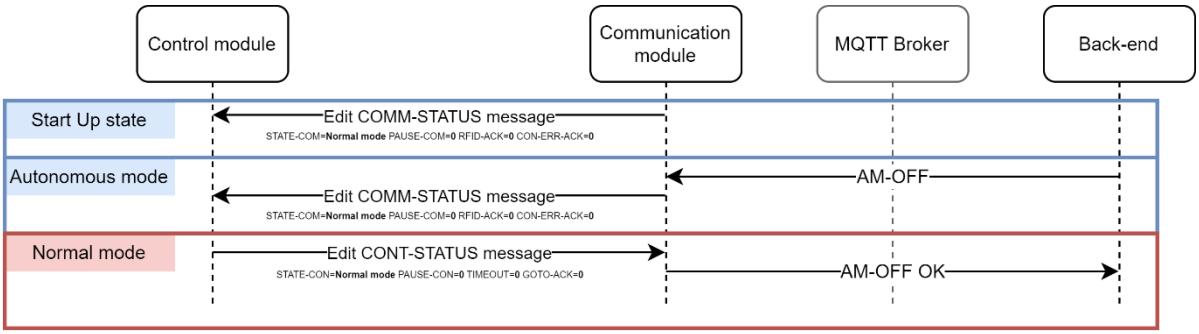



Figure 43. Message exchange between modules and back-end performed to go into the Normal mode.

As shown in Figure 43, it is possible to go into this state from the Start Up state or the Autonomous mode (if an AM-OFF message is received from the back-end). To change the state, the Communication module changes the value of the STATE-COM signal from the COMM-STATUS message to the value of the Normal mode, and waits to detect the change of the STATE-CON signal from the CONT-STATUS message to the value of the Normal mode. The control module, when it detects the change in the value of the STATE-COM signal, changes its status and reports it by changing

the value of the STATE-CON signal. If the STATE-CON signal does not change after 1 second, the communication module will trigger an *error* (see section 5.2.2). Once the status has been correctly established, the communication module sends the AM-OFF OK message through MQTT to inform the back-end.

### 5.1.3. Autonomous mode

As explained in section 3.2, the Autonomous mode is the state the system is in when it is in driverless operation, that is when the driving tasks of the vehicle are carried out by the installed system. Figure 44 shows the message exchange performed to switch to Autonomous mode.

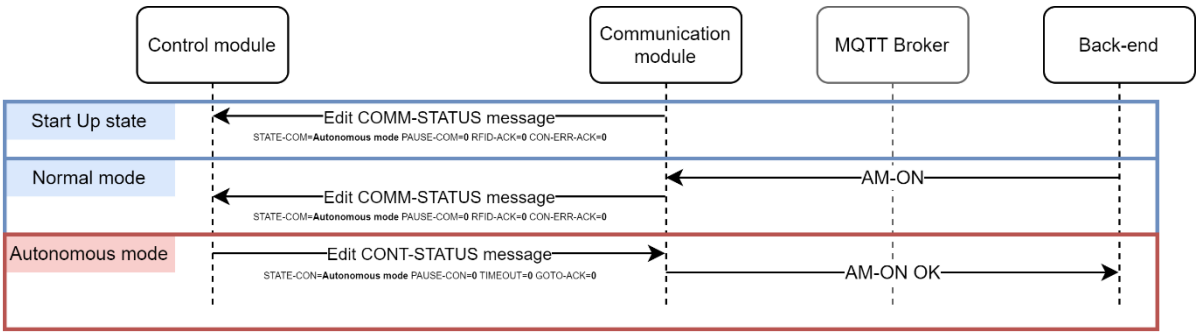



Figure 44. Message exchange between modules and back-end performed to go into the Autonomous mode.

As shown in Figure 44, it is possible to go into this state from the Start Up state or the Normal mode (if an AM-ON message is received from the back-end). To change the state, the Communication module changes the value of the STATE-COM signal from the COMM-STATUS message to the value of the Autonomous mode, and waits to detect the change of the STATE-CON signal from the CONT-STATUS message to the value of the Autonomous mode. The control module, when it detects the change in the value of the STATE-COM signal, changes its status and reports it by changing the value of the STATE-CON signal. If the STATE-CON signal does not change after 1 second, the Communication module will trigger an *error* (see section 5.2.2). Once the status has been correctly established, the communication module sends the AM-ON OK message through MQTT to inform the back-end.

Once the autonomous mode has been established, there are several use cases that can occur. The following subsections will show each of these cases.

#### 5.1.3.1. Configuration of the route to be followed

If the vehicle, even when it is in autonomous mode, does not have a route to follow, it will remain stopped. It will not start moving until it receives a correct GOTO message from the back-end. Figure 45 shows the message exchange that takes place in the system when the back-end sends a correct GOTO message.

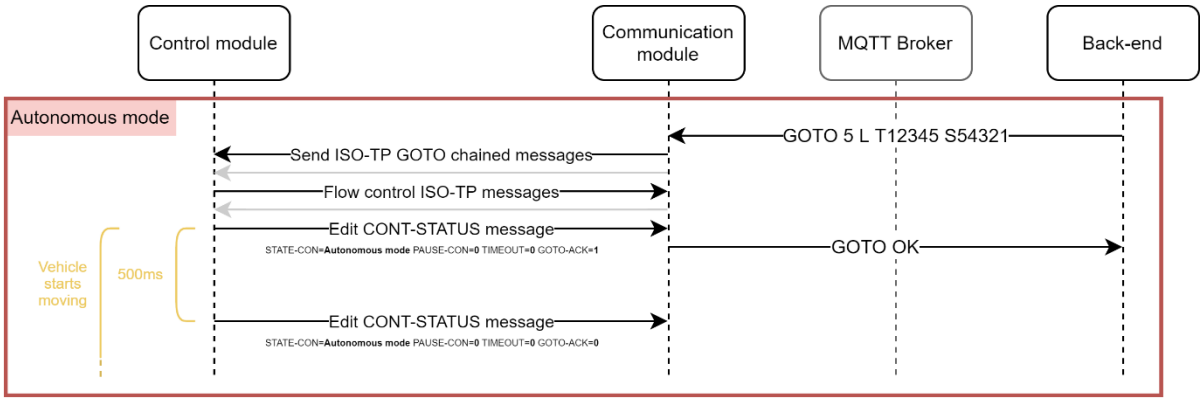



Figure 45. Message exchange between modules and the back-end performed for the configuration of a route.

As shown in Figure 45, initially the back-end sends the route to be followed through the GOTO message. Once the Communication module receives it, it checks if the message is correct. If it is not, it triggers a *warning* (see section 5.2.1). It generates a GOTO equivalent message for transmission through the ISO-TP layer. The sending of this message with this transport layer follows the methodology explained in section 3.3. The Control module, if it receives the message correctly, sets the value 1 in the GOTO-ACK signal of the periodic CONT-STATUS message for half a second, as explained in section 3.3. If the Control module does not receive the message correctly, it does not set the GOTO-ACK signal to 1. When the Communication module receives the GOTO-ACK signal at 1, it sends a GOTO OK message to the back-end informing that the message has been correctly received by both modules. If the Communication module does not receive the GOTO-ACK signal at 1 in half a second after the message has been sent, it tries again. If after 3 attempts the communication module still does not receive the GOTO-ACK signal at 1, it triggers an *error*.

### 5.1.3.2. Pause and continue orders

There are situations where it may be appropriate to order to stop immediately the vehicle in Autonomous mode. To order this, the back-end sends the PAUSE message. To order the vehicle to resume its route the back-end can send the CONTINUE message. Figure 46 shows the message exchange that takes place when both messages are received in autonomous mode.

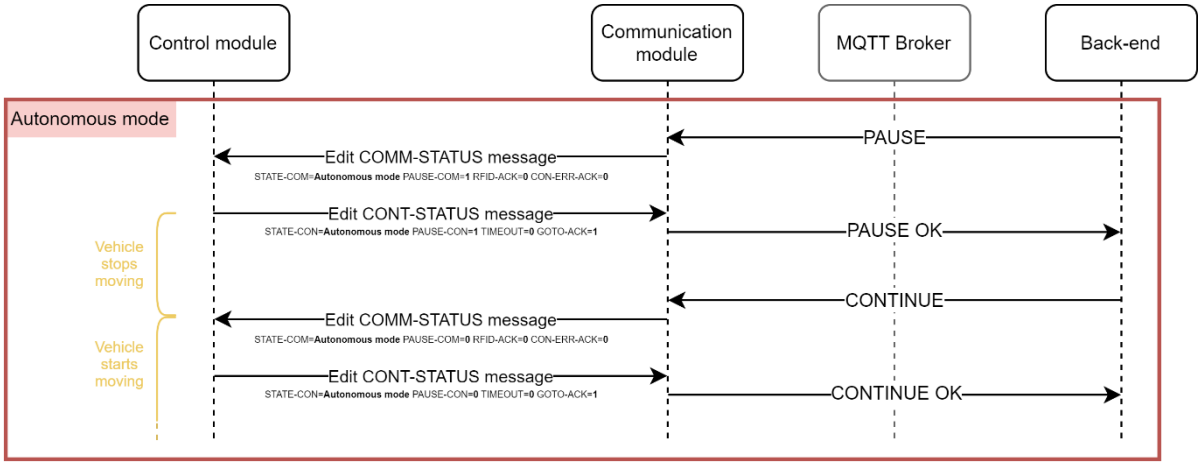



Figure 46. Message exchange between modules and the back-end performed when receiving pause or continue orders.

When the Communication module receives the PAUSE message from the back-end, it sets the value of the PAUSE-COM signal of the periodic COMM-STATUS message at 1, and waits to receive the PAUSE-CON signal of the periodic CONT-STATUS message with the value 1. When the Control module detects the change in the value of the PAUSE-COM signal, it stops the vehicle and sets the value of the PAUSE-CON signal of the periodic CONT-STATUS message to 1. When the

Communication module receives the signal PAUSE-CON at 1, it sends the message PAUSE OK to the back-end informing that the vehicle has stopped correctly. If the Communication module does not receive the PAUSE-CON signal at 1 for half a second after setting the PAUSE-COM signal at 1, it triggers an *error* (see section 5.2.1).

The procedure when the Communication module receives the CONTINUE message is similar to that for the PAUSE message. The Communication module changes the value of the PAUSE-COM signal to 0 and waits to receive the PAUSE-CON signal at 0. When the Control Module receives the PAUSE-COM signal at 0, it resumes the route for which it was configured and sets the PAUSE-CON signal to 0. The Communication module, on receiving this signal at 0, sends the CONTINUE OK message to the back-end informing that the vehicle will continue with the configured route.

#### 5.1.3.3. Obstacle detection

When the vehicle detects an obstacle, it slows down and stops if necessary. If it stays stopped for more than 10 seconds, inform the back-end with the TIMEOUT message. Figure 47 shows the message exchange that takes place in the system when the obstacle timer of the control module expires.

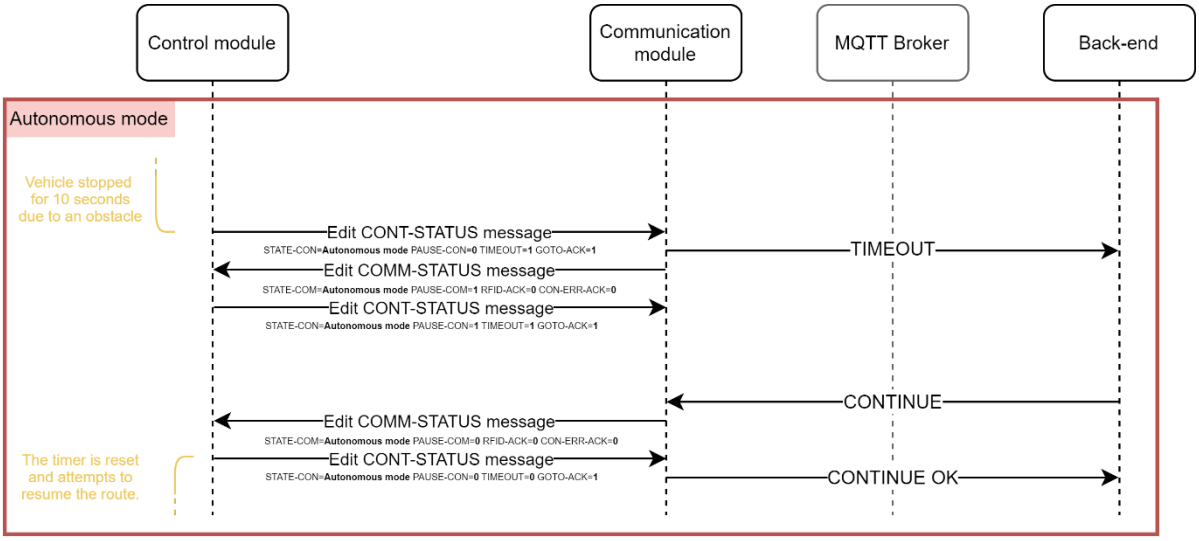



Figure 47. Message exchange between modules and back-end performed when the obstacle timer expires.

When the obstacle timer of the Control module expires, it sets the TIMEOUT signal of the periodic CONT-STATUS message at 1. The Communications module, when it receives this signal, informs the back-end about this by sending a TIMEOUT message, and performs a procedure similar to that performed when it receives the PAUSE message from the back-end (subsection 5.1.3.2); that is, it sets the PAUSE-COM signal of the periodic COMM-STATUS message at 1 and waits to receive the PAUSE-CON signal of the periodic CONT-STATUS message at 1. When the Control module receives the PAUSE-COM signal at 1, it keeps the vehicle stopped no matter if the obstacle has disappeared or not, and sets the PAUSE-CON signal at 1. If the Communication module does not receive the PAUSE-CON signal at 1 for a half a second after setting the PAUSE-COM signal at 1, it triggers an *error* (see section 5.2.1).

When the back-end considers it convenient that the vehicle tries again to resume the configured route, it can send the CONTINUE message. The procedure is similar to that shown in section 5.1.3.2. The communication module, on receiving the CONTINUE message from the back end, sets the value of the PAUSE-COM signal to 0 and waits to receive the PAUSE-CON signal at 0. When the Control Module receives the PAUSE-COM signal at 0, it reset the obstacle timer, resumes the route for which it was configured and sets the PAUSE-CON signal and the TIMEOUT signal at 0. The Communication

module, on receiving those signals at 0, sends the CONTINUE OK message to the back-end informing that the vehicle will try to continue with the configured route.

### 5.1.4. Standby mode

As explained in section 3.2, the Standby mode is the energy-saving state of the system. Figure 48 shows the exchange of messages between the modules and the back-end when the system goes into standby state.

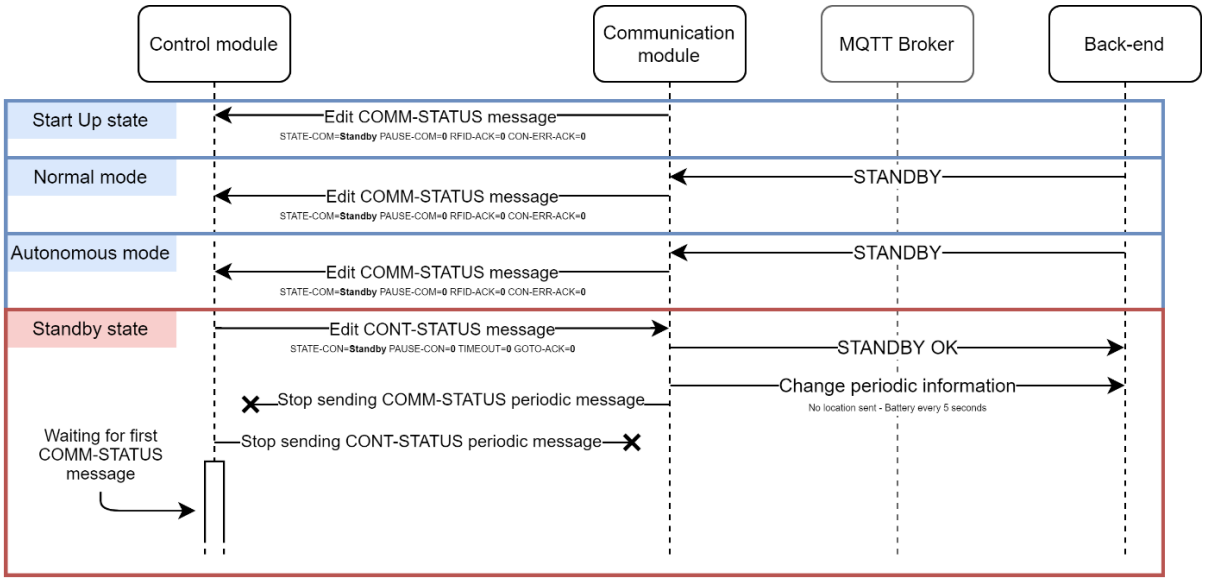



Figure 48. Message exchange between modules and back-end performed to go into the Standby state.

As we can see in Figure 48, it is possible to enter in the Standby state from the Start Up state, or the Normal and Autonomous modes (if the STANDBY message is received from the back-end). The Communication module changes the value of the STATE-COM signal to indicate the Standby state and waits for the Control module to change the value of the STATE-CON signal. The Control module, when it detects the change in the STATE-COM signal, changes the status (which means turning off all the devices it controls), and modifies the value of the STATE-CON signal to indicate the Standby status. The Communication module, on receiving the new value of the STATE-CON signal, considers the status as established and, therefore, sends a STANDBY OK message to the back-end, stops sending periodic information about the location, sends periodic information about the battery every 5 seconds, and stops sending the periodic COMM-STATUS message via the CAN bus. The Control module, when it stops receiving COMM-STATUS messages, stops sending CONT-STATUS messages and waits for new COMM-STATUS messages via the CAN bus to wake up again.

### 5.1.5. RFID tag detection

As explained in section 3.2, the system allows the detection of RFID tags in Normal mode and Autonomous mode. The control module, when it detects the tag, informs the communication module and this in turn to the back-end. Figure 49 shows the message exchange carried out when an RFID tag with the identifier 12345 is detected.

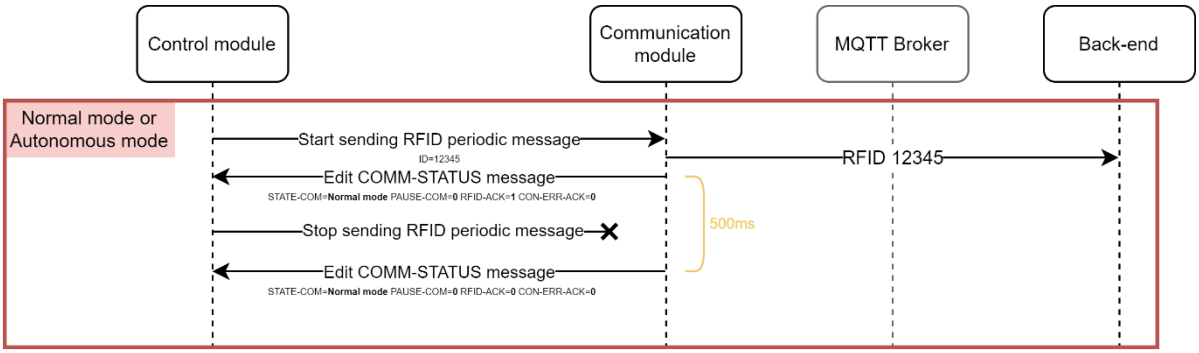



Figure 49. Message exchange between modules and back-end performed when an RFID tag is detected.

As we see in Figure 49, the control module, when it detects the RFID tag, starts sending the periodic RFID message. The communication module, when it receives this message, informs the back-end about it and sets the value of the RFID-ACK signal of the periodic COMM-STATUS message to 1, in order to indicate to the control module that the RFID message has been received. This signal, as explained in section 3.3, is maintained at 1 for half a second. The control module, once it receives the RFID-ACK signal at 1, stops sending the periodic RFID message. Note that if the RFID-ACK is not received in half a second after starting to send the RFID message, the control module throws an *error* (see section 5.2.1).

## 5.2. Failure cases

This section will explain the cases of system failures, that is, the way the system acts in the case of detecting some anomaly. Annex II shows all errors contemplated in the system. Although there are several cases, the system has general procedures for any type of failure, so that whatever it is, the procedures are the same, only varying the error code and attributes. There are 4 general procedures that are followed when an anomaly is detected. Depending on the type of failure (*error* or *warning*) and the device that detects it (Communication module or Control module) one procedure or another is taken. In addition, there is a special procedure that is carried out when *errors* occur that prevent communication with the MQTT broker and therefore with the back-end.

### 5.2.1. Failures triggered by the Control module

#### 5.2.1.1. Warnings

When the Control module triggers a *warning*, it sends the error information via the CON-ERR message which it starts to send periodically. Then it waits for the CON-ERR-ACK signal from the periodic COMM-STATUS message to stop sending the CON-ERR message. If it does not receive it within half a second it triggers an *error*. If it receives it, it stops sending the CON-ERR message and continues its tasks normally. On the other hand, the Communication module, when it receives the *warning* type CON-ERR message, sets the CON-ERR-ACK signal at 1 for half a second and sends a WRN message to the back-end informing it about the failure. A diagram of the message exchange in this use case is shown in Figure 50.

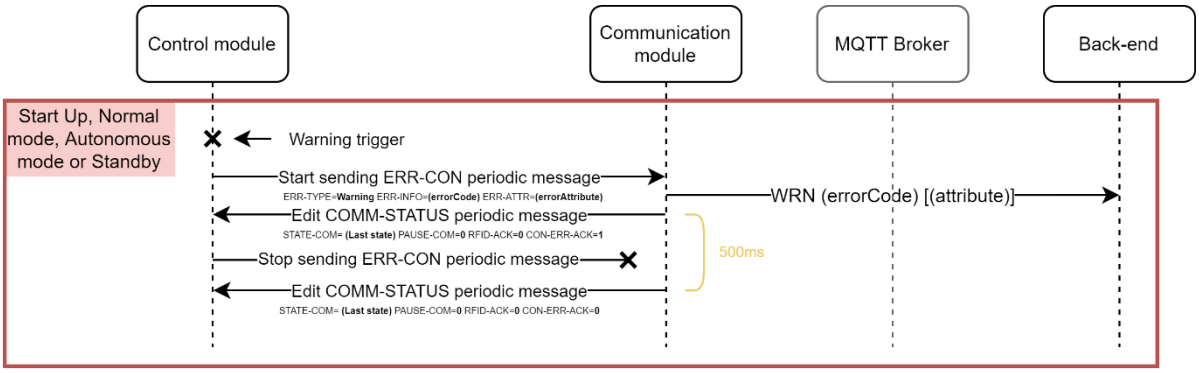



Figure 50. Message exchange between modules and back-end performed when a *warning* is triggered by the Control module.

#### 5.2.1.2. *Errors*

When the Control module triggers an *error,* it goes into the Failure mode. In this mode, the Control module starts sending the message CON-ERR with the *error* information for 5 seconds. It then stops sending that message and the CONT-STATUS message and waits for a new COMM-STATUS message in order to return to the Start Up state. On the other hand, the Communication module, when it receives the *error* type CON-ERR message, enters the Failure mode. In this state, the communication module stops attending the CAN bus, that is, it stops sending and receiving messages from it, and sends an ERR message to the back-end informing about the *error*. Then it waits for the RESTART message from the back-end to go into the Start Up state.

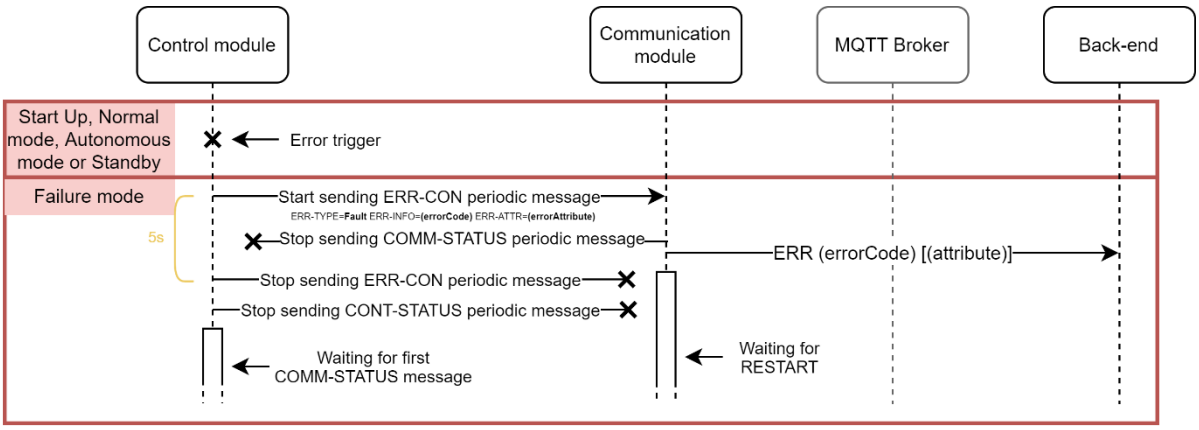



Figure 51. Message exchange between modules and back-end performed when an *error* is triggered by the Control module.

### 5.2.2. Failures triggered by the Communication module

#### 5.2.2.1. Warnings

When the Communication module triggers a *warning*, it sends the WRN message to the back-end informing about it and continues its tasks normally. Figure 52 provides a diagram showing the message sent by the Communication Module in this case.

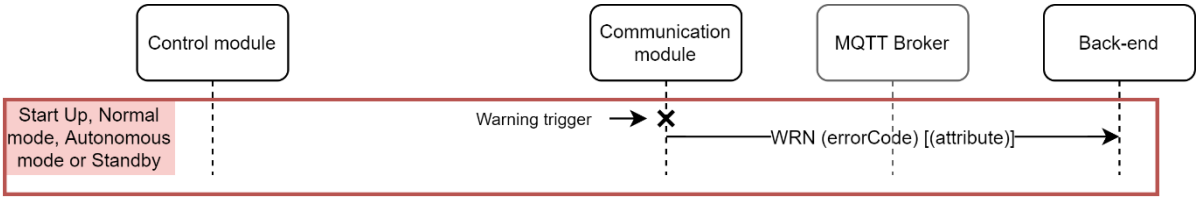



Figure 52. Message exchange between modules and back-end performed when a warning is triggered by the Communication module.

#### 5.2.2.2. *Errors*

When the Communication module triggers an *error*, it goes into Failure mode. Therefore, as seen in subsection 5.2.1.2 in this state the Communication Module stops attending the CAN bus, that is, it stops receiving and sending messages through it; and it sends to the back-end an ERR message informing about the *error*. When the Control Module stops receiving the COMM-STATUS message, it launches another *error* and performs the same tasks seen in section 5.2.1.2. It sends a CON-ERR message reporting the *error* for 5 seconds and then, stops transmitting its periodic CAN messages and waits to receive a new COMM-STATUS message to go into the Start Up state. Figure 53 shows the message exchange between the modules and the back-end in this use case.

Note that the *error* triggered by the Control module is deliberately caused by the Communication module so that the Control module also enters the Failure mode and waits for a new COMM-STATUS message to restart the system. CON-ERR messages sent by the Control module are ignored by the Communication module. This method is effective and allows the use of the same Failure status algorithm in both modules, without having to differentiate between *errors* coming from one module or another.

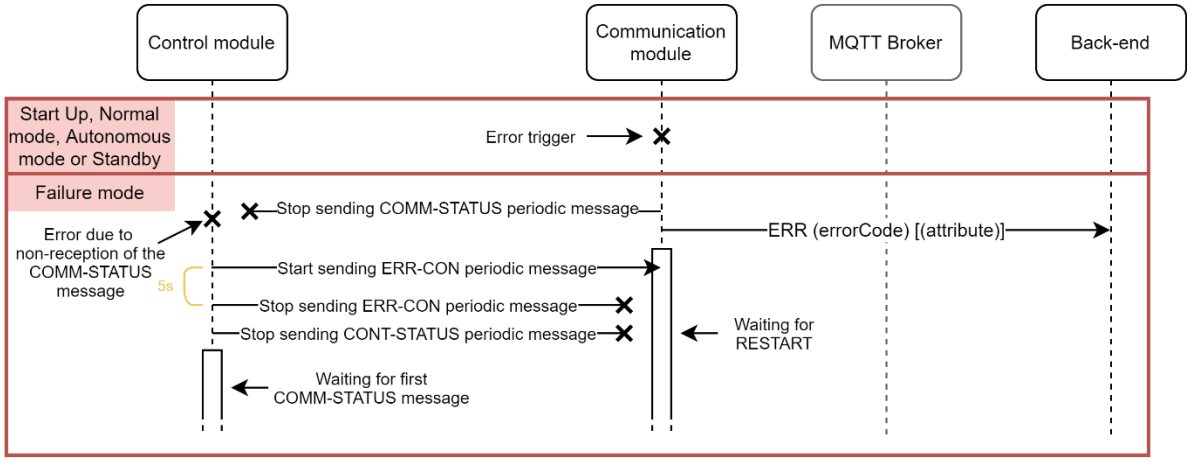



Figure 53. Message exchange between modules and back-end performed when an *error* is triggered by the Communication module.

#### 5.2.2.3. Failure due to non-connection with MQTT broker

There are certain *errors* (defined in Annex II) that imply the impossibility to connect with the MQTT broker. These *errors* can occur at the beginning, due to the inability to establish the connection with the broker; or later, due to a disconnection with the broker detected thanks to the MQTT pings requests sent periodically (explained in section 3.6.1). In these cases of use, the same procedure is followed as in the previous case (subsection 5.2.2.2) with the difference that the Communication module, instead of sending the ERR message and waiting to receive the RESTART message, it waits 10 seconds and goes directly to the Start Up status. There, it will try to establish a new connection with the MQTT broker. If it does not succeed, it will trigger an *error* again and the procedure will be repeated.

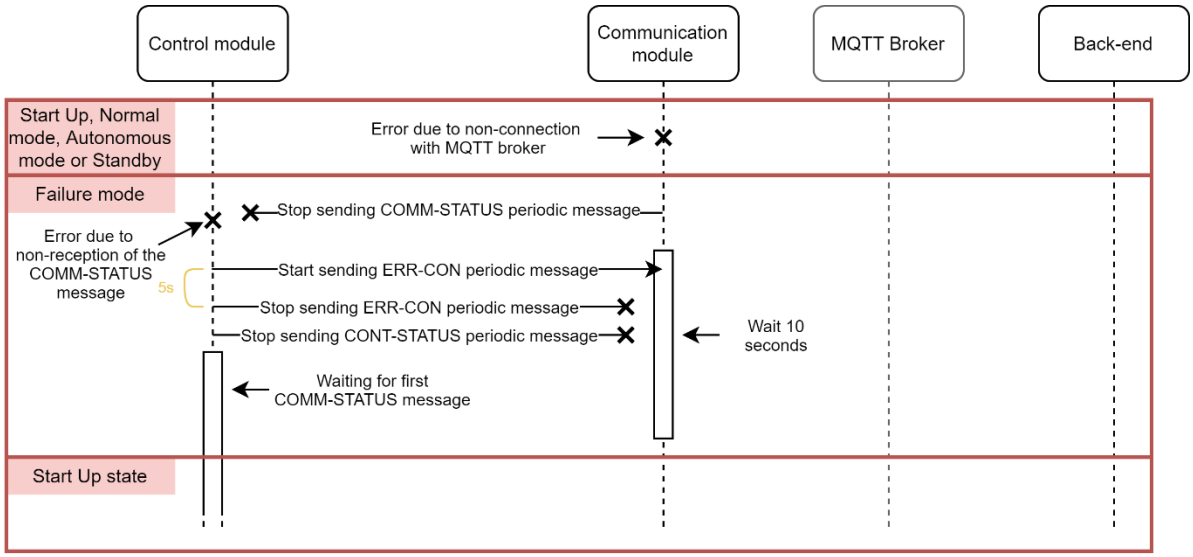



Figure 54. Message exchange between modules and back-end performed when an *error* due to non-connection with MQTT broker is triggered.

# 6. Testing and integration with simulator

This chapter shows the use cases of the simulator used to evaluate the integration of the vehicle system with the back-end (the development and operation of the back-end is explained in depth in the final degree project by Samuel Pilar Arnanz [13]). This chapter complements chapter 4 of this document (where the development of the simulator and the real-time viewer is explained) and, therefore, achieves the second main objective of this final project degree shown in section 1.2. This chapter corresponds to the last phase of this final project degree shown in section 1.3.

## 6.1. Preparation of the working environment

To evaluate the integration of the vehicle systems with the back-end, it is necessary to prepare the working environment to be able to launch several simulated vehicles and check the status of the complete TwizyLine system in real-time.

To achieve this, initially the files indicated in section 4.1.2 must be correctly configured. In the use case that will be shown in this chapter, it has been supposed that the TwizyLine service provides 3 parking areas distributed in the city of Valladolid (Figure 55). These car parks have the identifiers 63000, 63001, and 63002, respectively. Annex VI shows the files required to operate the simulator with these 3 parking areas.


Figure 55. Supposed car parks to carry out the simulation.

Once we have all the files correctly generated, it is time to launch the necessary programs. First, the **"Vehicles launcher" software**, analysed in section 4.2, must be launched. In this way the simulated vehicles can be launched with different parameters at any time during the simulation. Then, the **MQTT message sniffing software**, analysed in section 4.3, must be launched. This software updates periodically the KML file that will open the KML file viewer. From this moment on, this KML file will remain updated. Finally, a **KML viewer** must be launched (in this final project degree the *Google Earth* software is used) and configured to open the KML file *NetworkLink* to periodically open the KML file generated by the software previously mentioned.

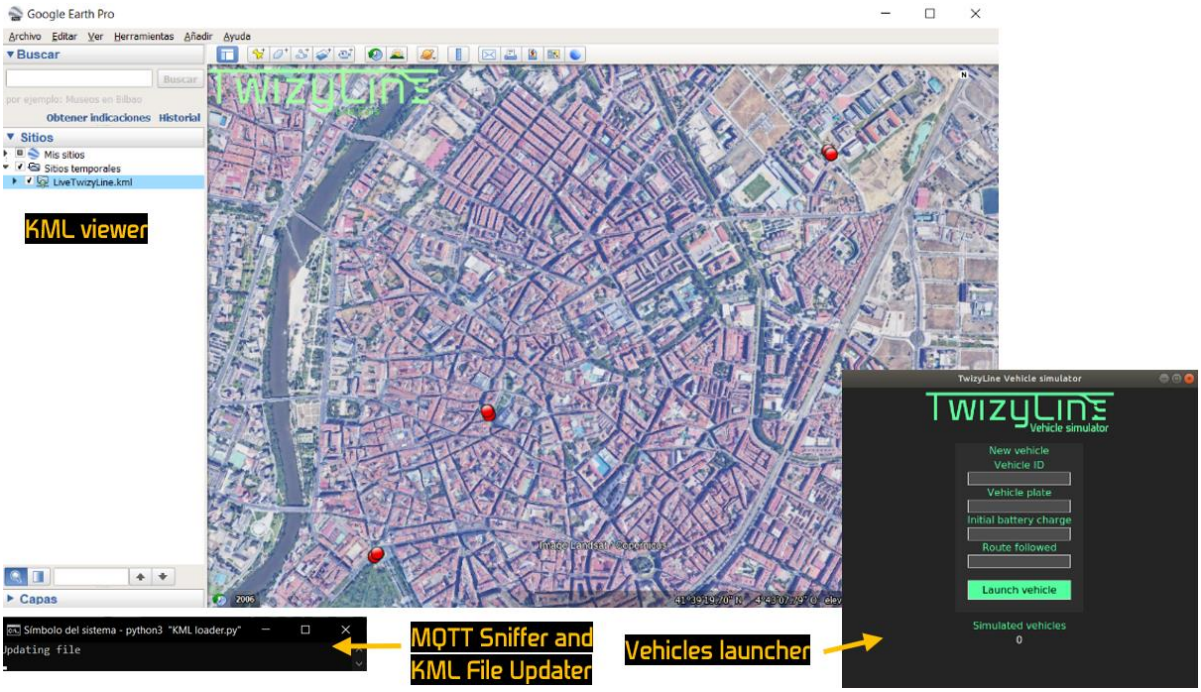



Figure 56. View of the working environment to check the integration with the back-end via the simulator.

## 6.2. General use case of the simulator

In this section the general use case of the simulator when a vehicle enters and leaves the car park with id 63000 mentioned above will be analysed. By knowing this use case, all the simulator use cases are known since the procedure is equivalent for all the car parks. To carry out the simulation, a vehicle has been configured with ID 3, an initial battery charge percentage of 60% and the route 63000-63001 (go to the car park 63000 and, when leaving it, go to the 63001).

When launching the vehicle with the "Vehicles launcher" software, the instance of the simulated vehicle starts sending a CONNECT message which is answered by the back-end with the CONNECTED message. Then, in Normal mode, the vehicle goes to the entrance of the first car park marked in the "Route followed" parameter of the "Vehicles launcher" software. Figure 57 shows how this vehicle arrives at the car park.

Figure 57. Simulated vehicle going to a car park.

Once the vehicle arrives at the car park entrance, there are two possible situations (Figure 58). If the barrier opens automatically when the vehicle arrives, the vehicle passes and is stopped inside the leaving zone of the car park. If the barrier does not open, the vehicle will wait for it to open.[7]


Figure 58. Possible situations when a simulated vehicle arrives at a car park.

Once the vehicle reaches the leaving zone, it informs the back-end about the detection of the corresponding RFID tag. This way the back-end knows that the vehicle is at that point. Afterwards, the simulated vehicle waits for further orders from the back-end. The back-end, when it considers it appropriate, opens barrier 2 and sends the AM-ON command to the vehicle to go into Autonomous mode. Then it sends the GOTO message to inform the vehicle about the route to be followed (Figure 59). Depending on this message, the vehicle will be parked in one row or another.

Figure 59. Simulated vehicle exiting the leaving zone after receiving the AM-ON and GOTO orders.

[7] The back-end opens the barrier automatically when a service vehicle is near. Generally, if the barrier does not open it is because there is another vehicle in the leaving zone. For more information about the decisions made by the back-end see the final project degree of Samuel Pilar Arnanz [13].

From this point on there are two possible cases depending if there are more vehicles parked in the row where the vehicle will be parked or not.

If there are **no more vehicles in the row** in which the vehicle is to be parked, it is placed at the end of the row and reports the detection of the back-end RFID. When the back-end detects this, it sends the STANDBY command to the vehicle so that it goes into Standby mode. From this moment on, the vehicle remains parked and in energy-saving mode waiting to receive wake-up orders from the back-end (Figure 60).

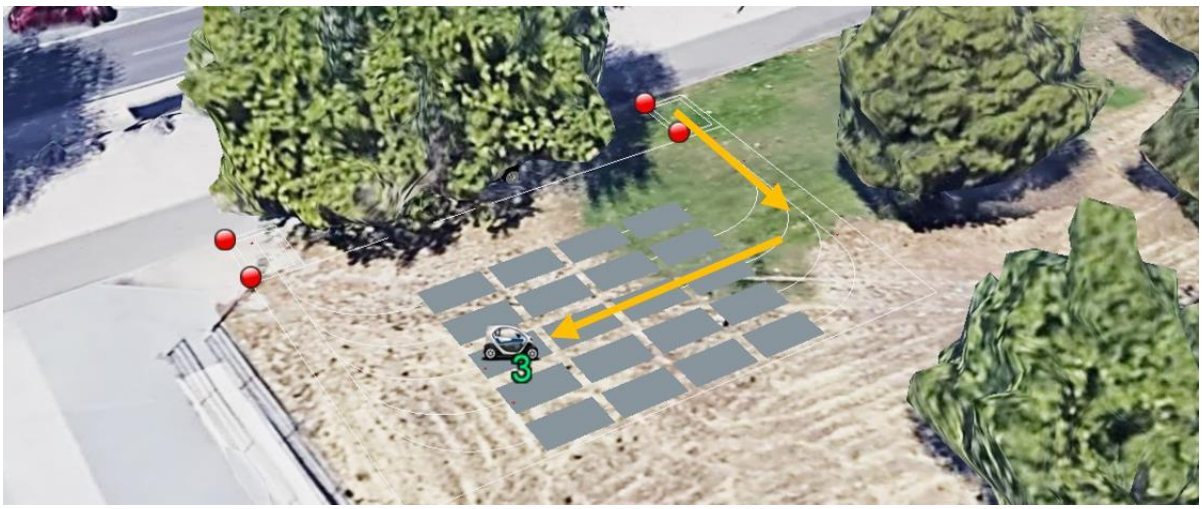



Figure 60. Simulated vehicle parked at the end of a car park row.

When the back-end considers it appropriate, it opens the barrier 3 and sends the AM-ON command to the vehicle so that it wakes up and goes into Autonomous mode. Then it sends the GOTO message indicating that it must go to the pick-up zone (indicating the corresponding RFID). The vehicle, upon receiving these messages, goes to the pick-up zone and reports the detection of the corresponding RFID (Figure 61).


Figure 61. Simulated vehicle arrives at car park pick-up zone.

When the back-end considers it appropriate, it changes the status of the vehicle to Normal mode by sending the AM-OFF order. From this moment on, the vehicle simulates being driven by a conventional driver and waits for the barrier 4 to open. Once the barrier is open, the vehicle leaves the car park and goes to the next car park, which is configured in the "Route followed" parameter of

the "Vehicles launcher" software. When it arrives at the next car park, the same procedure is carried out.

If there are **other vehicles parked in the row** in which the vehicle must be parked, it is placed in the previous park place and, after 10 seconds, it sends the TIMEOUT message to the back-end informing that the obstacle timer has expired (since there is a vehicle in front of it). When the back-end receives this message, it sends the STANDBY message and the vehicle goes into energy saving mode, waiting to receive a wake-up order from the back-end (Figure 62).


Figure 62. Vehicle parked behind another in the same car park row.

When the back-end orders the vehicle in front to leave the car park, it follows the procedure described above with that vehicle (Figure 63).


Figure 63. Vehicle that was in front of another one go the car park's pick-up zone.

It then orders the previous vehicle to wake up and go into Autonomous mode with the AM-ON message. After this, it sends a GOTO message to order it to move to the end of the row. The vehicle goes into Autonomous mode and advances. If it reaches the end of the row, it reports the detection of the corresponding RFID (Figure 64). If there is another vehicle in front, the same procedure is followed as before, sending the TIMEOUT message.


Figure 64. Vehicle advances to the end of a car park row and returns to Standby mode.

From this point on, the procedure explained above is followed. The back-end orders the vehicle to go into Standby mode and the vehicle waits for new orders from the back-end to wake up and go to the pick-up zone and then to another car park.

# 7. Conclusions and future lines

## 7.1. Conclusions

Taking into account the objectives marked in section 1.2 of this final degree project, it can be concluded that a design of a vehicle on-board system has been carried out to give it level 4 parking autonomy, the two processing modules of this system have been developed (except processes related to vehicle control), and a simulator has been developed to test the correct integration of the vehicle systems with the back-end. In this way, the objective 3 of the *TwizyLine* project shown in section 1.1 has been completed in time.

Regarding the different devices that are planned to be installed in the vehicle, a long and meticulous investigation of these was carried out, analysing their manuals, and checking their viability in the project. Certain devices, such as the magnetic sensor, were difficult to find. An effort was made to avoid developing any proprietary electronic device in order to have a functional prototype in the shortest possible time and thus, be able to present it in Renault's Twizy Contest.

The design of the CAN messaging was made considering the specific characteristics of this protocol. Different signals per message and periodic messages were used just as the CAN protocol is used in current vehicles. When this messaging was implemented in the real modules and tested, slight modifications were made to optimize it. The final designs are shown directly in this document. This reflects the importance of design testing. Verifying if a design is effective by checking its application in reality is a fundamental step when designing a new system.

The vehicle control interface was designed to be as simple, effective, and flexible as possible. Through this interface, the vehicle can follow any desired route. The characteristics of flexibility and scalability have been considered in the whole realization of this final project degree. Although the current idea is that the vehicle will only park automatically in a designated location, in the future, making the corresponding modifications (especially in the area of perception sensors of the vehicle), it could be possible that the vehicle could be guided at higher speeds to any area, as long as there is a magnetic strip that reaches there, with the same vehicle control interface.

The implementation of the Control and Communication modules was a long process. Instead of starting to program directly, a design of the processes that should run on each device was initially made, specifying who was going to launch them, what task each one should carry out, and most importantly, how they were going to communicate with each other. The design of these processes was carried out with the aim of making the programming as modular as possible. This makes future extensions or modifications of the project easier. Communication through messages via work queues was used to achieve a correct synchronization between these processes. This involved the definition of other specific messaging for each work queue. Just as with CAN messaging, when the design was implemented, slight modifications were made to optimize it.

The development of the simulator was a task that made it easier to check the integration of the vehicle systems with the back-end. Thanks to the modularity of the software of the real modules, the process to realize a simulator was easier. Although this simulator fulfils the basic tasks for which it was designed, it is expected to develop improvements to it that will be discussed in the future lines of the project.

This final degree project constitutes a great summary of everything I have learned in this degree. Designing and implementing a complete system starting from an idea and facing the different problems that arise during the process has been an exciting experience. The development of this final degree

project has been very motivating and enriching. In addition, being part of the TwizyLine project, I had to work with colleagues. As a team we have planned, discussed, and agreed different aspects of the project, which has helped me to learn how to work in teams on complex projects, a very important aspect that I will appreciate in the future.

## 7.2. Future lines

The main future line of this project is to complete objective 4 of the TwizyLine project explained in section 1.1. This consists of **installing all the elements** discussed in section 3.1 in a Twizy vehicle and **programming the vehicle control processes** in the control module. These processes would use guiding algorithms to make the vehicle operate correctly in the autonomous mode in order to follow the ordered route.

It is also necessary to design and implement the **vehicle's brake control system** to achieve complete control of all the important elements of the vehicle.

On the other hand, it will be necessary to carry out a **complete risk analysis** so that all possible events that may occur during the autonomous driving of the vehicle are deeply analysed.

Once all the appropriate modifications have been made to the vehicle, the necessary processes for its correct control have been programmed and an analysis and management of all the events that may occur in the autonomous driving of the vehicle has been made, this project opens many doors to future projects. These new projects may add functionalities to the vehicle to **expand the ODD** of the system and therefore give new utilities to it.

# 8. Merits and awards received

The TwizyLine project is our proposal for the international Twizy Contest 2020 organized by Renault and its partner Segula. Currently, this project has been the **winner in the national phase** of the contest, and therefore, will represent Spain in the international final.

The TwizyLine project has also been awarded one of the **PROMETEO awards**. These are organized by FunGe (General Foundation of the University of Valladolid) with the aim of obtaining a market-oriented prototypes.

As part of the obligations of the PROMETEO awards, the TwizyLine project has been **submitted for a patent**.

# 9. Bibliography


[1] Dirección General de Tráfico, Gobierno de España, "Parque de vehículos - Anuario," 2019.

[2] Instituto Nacional de Estadística, "Población residente en España," 2020.

[3] Dirección General de Tráfico, Gobierno de España, "Anuario estadístico de accidentes," 2018. [Online]. Available: http://www.dgt.es/Galerias/seguridad-vial/estadisticas-e-indicadores/publicaciones/anuario-estadistico-de-accidentes/Anuario-Estadistico-de-Accidentes-2018.pdf. [Accessed 16 Mayo 2020].

[4] National Highway Traffic Safety Administration of USA, "Critical Reasons for Crashes Investigated in the National Motor Vehicle Crash Causation Survey," *Traffic Safety Facts,* 2018.

[5] A. Sciarretta, G. de Nunzio and L. Leon Ojeda, "Optimal ecodriving control," *IEEE Control Systems Magazine,* no. October 2015, pp. 71-90, 2015.

[6] Office of Energy efficiency & renewable energy, "Fuel Economy, U.S. department of Energy," [Online]. Available: https://www.fueleconomy.gov/feg/evtech.shtml. [Accessed 17 Mayo 2020].

[7] L. Guzzella and A. Sciarretta, "Vehicle Propulsion Systems," Berlin, Springer-Verlag, 2013.

[8] Dirección General de Tráfico, Gobierno de España, "Conducción eficiente," 2014.

[9] National Highway Traffic Safety Administration of USA, "Automated Driving Systems," 2007.

[10] S. Waslander, "Autonomoose: Toward All-Weather Autonomous Driving," University of Toronto & University of Waterloo, 2018.

[11] Groupe Renault & Segula Technologies, "Twizy Contest," 2019. [Online]. Available: https://www.twizycontest.com/. [Accessed 17 May 2020].

[12] A. Mazaira Hernández, "Front-end implementation for an automatized car parking," Valladolid, 2020.

[13] S. Pilar Arnanz, "Back-end implementation for an automatized car parking," Valladolid, 2020.

[14] Solid Run, "Humming Board CBi," [Online]. Available: https://www.solid-run.com/nxp-family/hummingboard-cbi/. [Accessed 18 May 2020].

[15] Garmin, "GPS 18x OEM," [Online]. Available: https://buy.garmin.com/es-ES/ES/p/27594. [Accessed 18 May 2020].

[16] Huawei, "HUAWEI 4G Dongle E3372," [Online]. Available: https://consumer.huawei.com/en/routers/e3372/. [Accessed 18 May 2020].

[17] SAE International, «Surface vehicle recommended practice,» 2018.

[18] International Organization of Standardization, «ISO 26262».

[19] Navya SAS, "Navya," [Online]. Available: https://navya.tech/. [Accessed 2020 May 22].

[20] Alphabet Inc., "Waimo," [Online]. Available: https://waymo.com/. [Accessed 2020 May 22].

[21] Manga, "MAX4: Magna's Formula for Winning the Self-Driving Car Race," [Online]. Available: https://www.magna.com/innovation/driven-people-driving-change/article/max4-magna-s-formula-for-winning-the-self-driving-car-race. [Accessed 22 May 2020].

[22] Automotive News, "Volvo, Baidu team up for Level 4 autonomous EVs in China," [Online]. Available: https://www.autonews.com/article/20181101/MOBILITY/311019997/volvo-baidu-team-up-for-level-4-autonomous-evs-in-china. [Accessed 22 May 2020].

[23] 20 minutos, "Entre 2023 y 2025 habrá coches autónomos nivel 5 en las calles," 7 June 2017. [Online]. Available: https://www.20minutos.es/noticia/3056729/0/coches-autonomos-entre-2023-y-2025/. [Accessed 22 May 2020].

[24] General Motors Company, "Self-driving safety report," 2018.

[25] C. A. Falagán, *Introduction to Vehicle Telematics,* Aula Mercedes-Benz, Universidad de Valladolid.

[26] TESLA, Inc., "TESLA Autopilot," [Online]. Available: https://www.tesla.com/autopilot. [Accessed 2020 May 22].

[27] L. Lynch, T. Newe, J. Clifford, J. Coleman, J. Walsh and D. Toal, "Automated Ground Vehicle (AGV) and Sensor Technologies - A Review," University of Limerick, Limerick, Ireland, 2018.

[28] Nidec Motor Corporation, "Roboteq, Magnetic guide sensors," [Online]. Available: https://roboteq.com/all-products/magnetic-guide-sensors. [Accessed 2020 May 23].

[29] Applied & Integrated Manufacturing Inc., "Laser Target Navigation," [Online]. Available: http://www.aimagv.com/laser-target.html. [Accessed 2020 May 23].

[30] J. Sankari and R. Imtiaz, "Automatic Guided Vehicle (AGV) for Industrial Sector," Sri Manakula Vinayagar Engineering College, Puducherry, India, 2016.

[31] R. Weinstein, "RFID: a technical overview and its application to the enterprise," IT Professional, 2005.

[32] B. Djukic, "Hackernoon, Building an affordable self-driving car development platform," [Online]. Available: https://hackernoon.com/building-an-affordable-self-driving-car-development-platform-b02c6d7828ab. [Accessed 5 May 2020].

[33] R. Borraz Morón, P. J. Navarro Lorente, C. Fernández and P. M. Alcover Garau, "Cloud Incubator Car: A Reliable Platform for Autonomous Driving," Universidad Politécnica de Cartagena, 2019.

[34] Groupe Renault, *Renault Twizy: User manual,* 2012.

[35] C. Dunkel, *Adaption eines Versuchsträgers und Entwicklung.*

[36] Maxon Group, "Maxon motors," [Online]. Available: https://www.maxongroup.es/maxon/view/content/Categorias-de-productos. [Accessed 23 May 2020].

[37] M. Khader and S. Cherian, "An Introduction to Automotive Lidar".

[38] Ros Components, "3D LIDAR 360°," [Online]. Available: https://www.roscomponents.com/es/47-3d-lidar-360. [Accessed 5 May 2020].

[39] DFRobot, "ParkSonar-EZ-144," [Online]. Available: https://www.dfrobot.com/product-1807.html. [Accessed 23 May 2020].

[40] GPS.gov, "GPS Accuracy," [Online]. Available: https://www.gps.gov/systems/gps/performance/accuracy/. [Accessed 27 June 2020].

[41] Raspberry Pi Foundation, "Raspberry Pi 4 Model B: Specifications," [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/. [Accessed 24 May 2020].

[42] Raspberry Pi Foundation, "Raspberry Pi Model B Plus," [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/. [Accessed 24 May 2020].

[43] KunBus, "Revolution Pi: RevPi Core," [Online]. Available: https://revolution.kunbus.com/revpi-core/. [Accessed 24 May 2020].

[44] SferaLabs, "Strato Pi CAN Board," [Online]. Available: https://www.sferalabs.cc/product/strato-pi-can-board/. [Accessed 24 May 2020].

[45] dexterbg (GitHub user), "Open Vehicle Monitoring System, Renault Twizy, CANBUS-Objektverzeichnis," GitHub, [Online]. Available: https://github.com/openvehicles/Open-Vehicle-Monitoring-System/blob/master/docs/Renault-Twizy/CANBUS-Objektverzeichnis.ods. [Accessed 3 June 2020].

[46] Maxbotix, "ParkSonar-EZ Sensor series datasheet".

[47] RoboteQ, "MGS1600GY Datasheet".

[48] SparkFun, "RFID Reader ID-20LA (125 kHz)," [Online]. Available: https://www.sparkfun.com/products/11828. [Accessed 3 June 2020].

[49] ID Innovations, "Low Voltage Series RFID Reader modules Datashet".

[50] SparkFun, "RFID USB Reader," [Online]. Available: https://www.sparkfun.com/products/9963. [Accessed 3 June 2020].

[51] Maxon Group, "EC 60 flat Ø60 mm, brushless, 100 W, with Hall sensors," [Online]. Available: https://www.maxongroup.com/maxon/view/product/625854. [Accessed 6 June 2020].

[52] Maxon Group, "Planetary Gearhead GP 52 C Ø52 mm, 4 - 30 Nm, Ceramic Version," [Online]. Available: https://www.maxongroup.com/maxon/view/product/223093. [Accessed 6 June 2020].

[53] Maxon Group, "Encoder MILE, 4096 cpt, 2-channel, with line driver," [Online]. Available: https://www.maxongroup.com/maxon/view/product/651168. [Accessed 6 June 2020].

[54] Maxon group, "EPOS4 70/15, digital positioning controller, 15 A, 10 - 70 VDC," [Online]. Available: https://www.maxongroup.com/maxon/view/product/control/Positionierung/EPOS-4/594385.

[55] Arduino Inc., "Arduino Micro," [Online]. Available: https://store.arduino.cc/arduino-micro. [Accessed 6 June 2020].

[56] International Organization for Standarization, "Road vehicles - Diagnostics on CAN, Part 2: Network later services, ISO 15765-2," 2016.

[57] J. C. Aguado Manzano, A. Nicolas Gentil Otero and N. H. Cabello Martínez, "Aula Mercedes Benz: 7. ECU Diagnosis," University of Valladolid, Valladolid, 2019.

[58] Debian Support, "Package: usb-modeswitch," [Online]. Available: https://packages.debian.org/en/sid/usb-modeswitch. [Accessed 15 June 2020].

[59] PyPi, "Manual of the paho-mqtt library 1.5.0," [Online]. Available: https://pypi.org/project/paho-mqtt/. [Accessed 2020 Junio 15].

[60] Read The Docs, "python-can," [Online]. Available: https://python-can.readthedocs.io/en/master/. [Accessed June 15 2020].

[61] Read The Docs, "can-isotp," [Online]. Available: https://can-isotp.readthedocs.io/en/latest/. [Accessed 15 June 2020].

[62] GitLab, "gpsd," [Online]. Available: https://gpsd.gitlab.io/gpsd/index.html. [Accessed 17 June 2020].

[63] gpsd, "gpsd request/response protocol," [Online]. Available: https://gpsd.gitlab.io/gpsd/gpsd_json.html. [Accessed 17 June 2020].

[64] Read The Docs, "pyserial," [Online]. Available: https://pyserial.readthedocs.io/en/latest/pyserial.html. [Accessed 18 June 2020].

[65] Open Geospatial Consortium, "Keyhole Markup Language," [Online]. Available: https://www.ogc.org/standards/kml. [Accessed 22 June 2020].

[66] Python Docs, "tkinter," [Online]. Available: https://docs.python.org/3/library/tkinter.html. [Accessed 22 June 2020].

[67] Google Inc., "Google Earth," [Online]. Available: https://www.google.com/earth/. [Accessed 2020 June 23].

[68] Google Inc., "Developers: KML Tutorial," [Online]. Available: https://developers.google.com/kml/documentation/kml_tut. [Accessed 23 June 2020].

[69] Embeebed Linux Wiki, "Device Tree Reference," [Online]. Available: https://elinux.org/Device_Tree_Reference. [Accessed 2020 June 21].

[70] Ubuntu, "DTC man page," [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man1/dtc.1.html. [Accessed 21 June 2020].

# Annex I

This annex shows a table with all the CAN messages entered in the vehicle for communication between the Control module and the Communication module.

| ID | Type | Period (ms) | Source | Destination | Name | Length (bytes) | Bit(s) | Signal name | Meaning |
|---|---|---|---|---|---|---|---|---|---|
| **100** | Cyclic | 100 | COMMUNICATION | CONTROL | COMM-STATUS | 1 | 0-1 | STATE-COM | 00=Start-Up, 01=Normal,<br>10=Autonomous, 11=Standby |
| **100** | Cyclic | 100 | COMMUNICATION | CONTROL | COMM-STATUS | 1 | 2 | PAUSE-COM | 0=Continue<br>1=Pause |
| **100** | Cyclic | 100 | COMMUNICATION | CONTROL | COMM-STATUS | 1 | 3 | RFID-ACK | 0=Not RFID message received<br>1=RFID message received |
| **100** | Cyclic | 100 | COMMUNICATION | CONTROL | COMM-STATUS | 1 | 4 | CON-ERR-ACK | 0=Not CON-ERR message received<br>1=CON-ERR message received |
| **101** | Cyclic | 100 | CONTROL | COMMUNICATION | CONT-STATUS | 1 | 0-1 | SATE-CON | 00=Start-Up, 01=Normal,<br>10=Autonomous, 11=Standby |
| **101** | Cyclic | 100 | CONTROL | COMMUNICATION | CONT-STATUS | 1 | 2 | PAUSE-CON | 0=Continue<br>1=Pause |
| **101** | Cyclic | 100 | CONTROL | COMMUNICATION | CONT-STATUS | 1 | 3 | TIMEOUT | 0=Obstacle timer not exceeded,<br>1=Obstacle timer exceeded |
| **101** | Cyclic | 100 | CONTROL | COMMUNICATION | CONT-STATUS | 1 | 4 | GOTO-ACK | 0=Not GOTO message received,<br>1=GOTO message received |
| **102** | Cyclic and spontaneous | 100 | CONTROL | COMMUNICATION | RFID | 5 | 0-39 | ID | Identification number of the RFID tag detected |
| **103** | Cyclic and spontaneous | 100 | CONTROL | COMMUNICATION | CON-ERR | 8 | 0-1 | ERR-TYPE | 00=Warning<br>01= Error |
| **103** | Cyclic and spontaneous | 100 | CONTROL | COMMUNICATION | CON-ERR | 8 | 2-15 | ERR-INFO | Error explanation |
| **103** | Cyclic and spontaneous | 100 | CONTROL | COMMUNICATION | CON-ERR | 8 | 16-63 | ERR-ATTR | Error attributte |
| **104** | ISO-TP | | COMMUNICATION | CONTROL | GOTO | | | | GOTO chain |
| **105** | ISO-TP | | CONTROL | COMMUNICATION | GOTO | | | | Flow control |

# Annex II

This annex shows a table with all the errors contemplated in the system. It shows the error code, the type of error, the value of the error attribute (if any) and a brief explanation of the error and its possible causes.

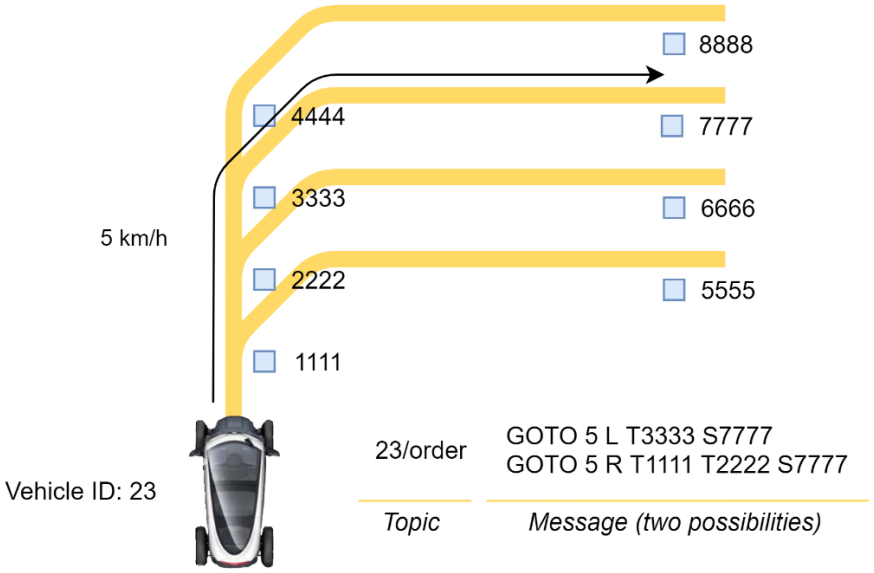
| Source | Dec Code | Hex Code | Type of failure | Failure name | Attribute | Explanation |
|---|---|---|---|---|---|---|
| **Communication module** | 1 | 0x001 | Error | CON-STATUS not received | - | The CON_STATUS frame is not being received. Is the CON-MOD on? Is the CAN-Bus connection of both modules correct? |
| **Communication module** | 2 | 0x002 | Error | Control module does not respond to CON_STATUS signal - Cannot change status | - | CON-MOD does not change status when ordered to do so. Is the CON-MOD correctly configured? |
| **Communication module** | 3 | 0x003 | Error | Control module does not respond to PAUSE signal | - | When sending a PAUSE or CONTINUE command from the COM-MOD, the CON-MOD does not respond correctly as the PAUSE signal does not change. Is the CON-MOD correctly configured? |
| **Communication module** | 4 | 0x004 | Error | GOTO-ACK not received after several attempts | - | CON-MOD does not send an ACK when it has to do so. Is the CON-MOD correctly configured? |
| **Communication module** | 5 | 0x005 | Warning | Malformed GOTO frame received from Communication module | - | COM-MOD has received a GOTO message but its information is not in the proper way. |
| **Communication module** | 6 | 0x006 | Error | Unexpected frame from Control module | ID of frame received | CON-MOD is sending out signals that do not make sense right now. Is CON-MOD well configured? |
| **Communication module** | 9 | 0x009 | Error | GPS not connected | Last coordinates (if any) | The GPS module is not connected to the Communication module. Is the COM-MOD correctly configured? Is the GPS broken? Is the GPS disconnected? |
| **Communication module** | 10 | 0x00A | Warning | GPS is not receiving signal for a long time | Last coordinates (if any) | The GPS module is not receiving any signal from the satellites for a long time. Is the GPS broken? |
| **Communication module** | 11 | 0x00B | Warning | Battery data not received | Last battery data received (if any) | The CAN message informing about the battery charge on the vehicle's CAN-bus is not detected. Is there a problem in the vehicle? |
| **Communication module** | 17 | 0x011 | Error | Network is unreachable | - | COM-MOD is not connected to the internet. Is the COM-MOD connected to the Internet? |
| **Communication module** | 18 | 0x012 | Error | Connection refused - Mosquitto service is not running | - | MQTT messages are arriving at a machine where the mosquito service is not running. Is the mosquitto service running on the server? |
| **Communication module** | 19 | 0x013 | Error | Cannot connect with server | - | MQTT messages are not arriving at any machine. Is the IP address right? |
| **Communication module** | 20 | 0x014 | Error | Connection refused - Not authorised | - | The MQTT broker is rejecting the connection due to no authentication. Is the client sending the correct authentication data? |
| **Communication module** | 21 | 0x015 | Error | Connection refused - Bad username or password | - | The MQTT broker is rejecting the connection due to a bad authentication. Is the client sending the correct username and password? |
| **Communication module** | 22 | 0x016 | Error | Connection refused - Server unavailable | - | Not tested |

| | | | | | | |
|---|---|---|---|---|---|---|
| **Communication module** | 23 | 0x017 | Error | Connection refused - Invalid client identifier | - | Not tested |
| **Communication module** | 24 | 0x018 | Error | Connection refused - Incorrect protocol version | - | Not tested |
| **Communication module** | 25 | 0x019 | Error | Back-end is not responding | - | The server is not responding with the message CONNECTED when called. Is the server running? |
| **Communication module** | 26 | 0x01A | Warning | Unexpected MQTT message | Message received | The server is sending an incorrect message. Is the server well configured? |
| **Communication module** | 27 | 0x01B | Error | Lost connection with MQTT server | - | Ping responses from the server are not being received. Has CON-MOD lost the internet connection? Has the MQTT broker stopped? |
| **Control module** | 129 | 0x081 | Error | COM_STATUS not received | - | The COM_STATUS frame is not being received. Is the COM-MOD on? Is the CAN-Bus connection of both modules correct? |
| **Control module** | 130 | 0x082 | Error | RFID-ACK not received | - | COM-MOD does not send an ACK when it has to do so. Is the COM-MOD correctly configured? |
| **Control module** | 131 | 0x083 | Error | CON-ERR-ACK not received | - | COM-MOD does not send an ACK when it has to do so. Is the COM-MOD correctly configured? |
| **Control module** | 132 | 0x084 | Error | Unexpected frame received from Communication module | ID of frame received | COM-MOD is sending out signals that do not make sense right now. Is COM-MOD well configured? |
| **Control module** | 133 | 0x085 | Error | Malformed GOTO frame received from Communication module | - | CON-MOD has received a GOTO message but its information is not in the proper way. |

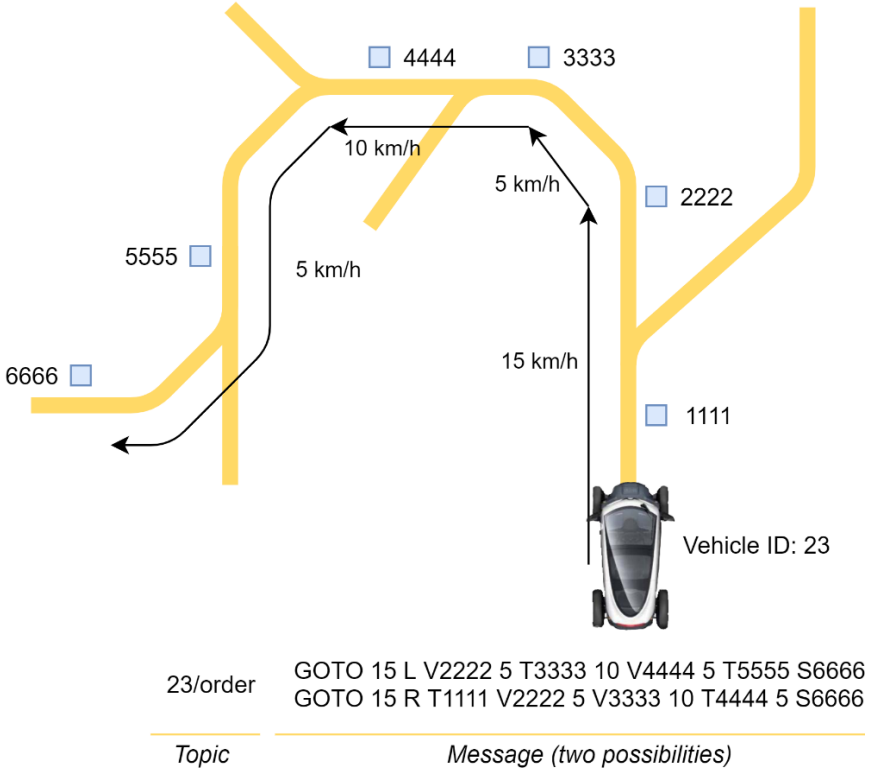
# Annex III

This annex shows two examples of GOTO messages sent to the vehicle. The vehicle, receiving this message, will follow the route marked on the diagram at the speed shown. Each example shows two possible GOTO messages that can be sent to the vehicle depending on the default fork parameter.

The first example shows a typical situation in the designed car parks (detailed in the final project degree of Samuel Pilar Arnanz)




The second example shows a more complex route, this demonstrates the scalability and flexibility of this messaging.



# Annex IV

This annex explains the procedure for enabling the CAN interface on a *Humming Board CBi*.

The operating system available for the *Humming Board CBi* comes with the same kernel as the *Humming Board Edge* and *Humming Board Gate*. The *Humming Board Edge* and *Humming Board Gate* have an HDMI port instead of a transceiver and CAN interface. On the *Humming Board CBi* this kernel, by default, interprets that there is an HDMI port and not a CAN transceiver. To change its configuration, it is necessary to edit the kernel device tree.

The device tree tell the kernel how the internal devices of the system in which it is running are interconnected [69]. This can indicate, among others, the interfaces that the device currently has. The device tree consists of a *dtb* file that is read by the kernel at boot time. While *dtb* is binary and difficult to modify, there are tools such as *device-tree-compiler* [70] that can convert it to a *dts* file. This file is equivalent to the *dtb* file, but it is ASCII coded and allows an easy understanding and modification of it.

To modify the tree of devices, we must initially convert it to a dts file with the following command: `dtc -I dtb -O dts -o mydtsfile /boot/dtbs/4.9.150-imx6-sr/imx6q-hummingboard2-emmc-som-v15.dtb`.

Then we open the *dts* file and modify the following lines:

```
flexcan@02090000 {
                        compatible = "fsl,imx6q-flexcan";
                        reg = < 0x2090000 0x4000 >;
                        interrupts = < 0x00 0x6e 0x04 >;
                        clocks = < 0x02 0x6c 0x02 0x6d >;
                        clock-names = "ipg\0per";
                        stop-mode = < 0x04 0x34 0x1c 0x10 0x11 >;
                        status = "okay";
                        pinctrl-names = "default";
                        pinctrl-0 = < 0x16 >;
                        };

hdmi@0120000 {
                        #address-cells = < 0x01 >;
                        #size-cells = < 0x00 >;
                        reg = < 0x120000 0x9000 >;
                        interrupts = < 0x00 0x73 0x04 >;
                        gpr = < 0x04 >;
                        clocks = < 0x02 0x7b 0x02 0x7c >;
                        clock-names = "iahb\0isfr";
                        status = "disabled";
                        compatible = "fsl,imx6q-hdmi";
                        pinctrl-names = "default";
                        pinctrl-0 = < 0x05 >;
                        ddc-i2c-bus = < 0x06 >;
                        };

hdmi_cec@00120000 {
                        compatible = "fsl,imx6q-hdmi-cec";
                        interrupts = < 0x00 0x73 0x04 >;
                        status = "disabled";
                        pinctrl-names = "default";
                        pinctrl-0 = < 0x05 >;
                        };
```

Finally, we compile the new device tree with the following command: `dtc -I dts -O dtb -o /boot/dtbs/4.9.150-imx6-sr/imx6q-hummingboard2-emmc-som-v15.dtb mydtsfile`

Once the device has been rebooted, we can use the CAN interface of the *Humming Board CBi*.

# Annex V

This annex shows a table with the messaging of all the work queues used in both modules of the system. The table shows: the module in which the queue is implemented; the queue name; the queue direction, that is, if it goes from a secondary process to the main one or vice versa; the message; and an explanation of the message purpose.

| Device | Working queue | Direction | Message | Explanation |
|---|---|---|---|---|
| Communication | q_CAN_receiver | To the main process | OK | Connection with Control module established |
| Communication | q_CAN_receiver | To the main process | S0 | STATE-CON signal=00 (Start Up) |
| Communication | q_CAN_receiver | To the main process | S1 | STATE-CON signal=01 (Normal mode) |
| Communication | q_CAN_receiver | To the main process | S2 | STATE-CON signal=10 (Autonomous mode) |
| Communication | q_CAN_receiver | To the main process | S3 | STATE-CON signal=11 (Stand By) |
| Communication | q_CAN_receiver | To the main process | P0 | PAUSE-CON signal=0 |
| Communication | q_CAN_receiver | To the main process | P1 | PAUSE-CON signal=1 |
| Communication | q_CAN_receiver | To the main process | T0 | TIMEOUT signal=0 |
| Communication | q_CAN_receiver | To the main process | T1 | TIMEOUT signal=1 |
| Communication | q_CAN_receiver | To the main process | R (ID) | RFID frame received |
| Communication | q_CAN_receiver | To the main process | W (error code) [(attribute)] | Triggering a *warning* type error |
| Communication | q_CAN_receiver | To the main process | E (error code) [(attribute)] | Triggering an *error* type error |
| Communication | q_CAN_sender | From the main process | S0 | STATE-COM signal=00 |
| Communication | q_CAN_sender | From the main process | S1 | STATE-COM signal=01 |
| Communication | q_CAN_sender | From the main process | S2 | STATE-COM signal=10 |
| Communication | q_CAN_sender | From the main process | S3 | STATE-COM signal=11 |
| Communication | q_CAN_sender | From the main process | P0 | PAUSE-COM signal=0 |
| Communication | q_CAN_sender | From the main process | P1 | PAUSE-COM signal=1 |
| Communication | q_CAN_sender | From the main process | R0 | RFID-ACK signal=0 |
| Communication | q_CAN_sender | From the main process | R1 | RFID-ACK signal=1 |
| Communication | q_CAN_sender | From the main process | E0 | CON-ERR-ACK signal=0 |
| Communication | q_CAN_sender | From the main process | E1 | CON-ERR-ACK signal=1 |
| Communication | q_CAN_sender | From the main process | G0 | Stop resending GOTO message through ISO-TP |
| Communication | q_CAN_sender | From the main process | G1 (goto chain) | Send GOTO message through ISO-TP |
| Communication | q_CAN_sender | From the main process | TERMINATE | Stop process |
| Communication | q_MQTT_receiver | To the main process | OK | Connection with broker MQTT established |
| Communication | q_MQTT_receiver | To the main process | M (payload) | Payload received from server linked with order topic |
| Communication | q_MQTT_receiver | To the main process | E (error code) [(attribute)] | Triggering an *error* type error |
| Communication | q_MQTT_receiver | To the main process | W (error code) [(attribute)] | Triggering a *warning* type error |
| Communication | q_MQTT_sender | From the main process | (payload) | Payload to send through info topic |
| Communication | q_MQTT_sender | From the main process | TERMINATE | Stop process |
| Communication | q_MQTT_periodic_sender | From the main process | STANDBY | Only send battery information every 5 seconds |
| Communication | q_MQTT_periodic_sender | From the main process | NOTSTANDBY | Send location and battery information every second |

| | | | | |
|---|---|---|---|---|
| Communication | q_MQTT_periodic_sender | From the main process | TERMINATE | Stop process |
| Communication | q_GPS_receiver | To the main process | W (error code) (attribute) | Triggering a *warning* type error |
| Communication | q_GPS_receiver | To the main process | E (error code) (attribute) | Triggering an *error* type error |
| Control | q_CAN_receiver | To the main process | CSR | First COMM-STATUS frame received |
| Control | q_CAN_receiver | To the main process | S0 | STATE-COM signal=00 (Start Up) |
| Control | q_CAN_receiver | To the main process | S1 | STATE-COM signal=01 (Normal mode) |
| Control | q_CAN_receiver | To the main process | S2 | STATE-COM signal=10 (Autonomous mode) |
| Control | q_CAN_receiver | To the main process | S3 | STATE-COM signal=11 (Stand By) |
| Control | q_CAN_receiver | To the main process | P0 | PAUSE-COM signal=0 |
| Control | q_CAN_receiver | To the main process | P1 | PAUSE-COM signal=1 |
| Control | q_CAN_receiver | To the main process | G (goto data) | GOTO ISO-TP message received |
| Control | q_CAN_receiver | To the main process | W (error code) [(attribute)] | Triggering a *warning* type error |
| Control | q_CAN_receiver | To the main process | E (error code) [(attribute)] | Triggering an *error* type error |
| Control | q_CAN_sender | From the main process | READY | Start sending periodic CONT-STATUS frame |
| Control | q_CAN_sender | From the main process | S0 | STATE-CON signal=00 |
| Control | q_CAN_sender | From the main process | S1 | STATE-CON signal=01 |
| Control | q_CAN_sender | From the main process | S2 | STATE-CON signal =10 |
| Control | q_CAN_sender | From the main process | S3 | STATE-CON signal =11 |
| Control | q_CAN_sender | From the main process | P0 | PAUSE-CON signal =0 |
| Control | q_CAN_sender | From the main process | P1 | PAUSE-CON signal =1 |
| Control | q_CAN_sender | From the main process | T0 | TIMEOUT signal =0 |
| Control | q_CAN_sender | From the main process | T1 | TIMEOUT signal =1 |
| Control | q_CAN_sender | From the main process | G0 | GOTO-ACK signal=0 |
| Control | q_CAN_sender | From the main process | G1 | GOTO-ACK signal=1 |
| Control | q_CAN_sender | From the main process | R0 | Stop periodic RFID message |
| Control | q_CAN_sender | From the main process | R1 (RFID) | Start periodic RFID message |
| Control | q_CAN_sender | From the main process | E0 | Stop periodic CON-ERR message |
| Control | q_CAN_sender | From the main process | E1 (error code) [(error info)] | Start periodic CON-ERR message. *Error* type failure. |
| Control | q_CAN_sender | From the main process | W1 (error code) [(error info)] | Start periodic CON-ERR message. *Warning* type failure. |
| Control | q_CAN_sender | From the main process | TERMINATE | Stop process |

# Annex VI

This annex shows the collection of files required for the operation of the simulator in the use case developed in Chapter 4. In this case, 3 car parks with identifiers 63000, 63001, and 63002 are assumed.