Università degli Studi di Padova

Department of Information Engineering

Master's degree course in
Control System Engineering

The new design for Automated Guided Vehicles: from laser triangulation to natural navigation

Supervisor:
Prof. Angelo Cenedese

Author:
Gloria Cecchetto
2022573

Academic Year 2021/2022

"I'm going to let you in on a little secret:
Every day, once a day, give yourself a present.
Don't plan it, don't wait for it, just let it happen.

Agent Cooper, Twin Peaks

Abstract

This thesis is the result of a project followed during an internship at Euroimpianti SPA - Skilled Group. The objective was the conversion of an Automated Guided Vehicles (AGV) from laser triangulation navigation to natural navigation.
This type of AGV is used in warehouse management, so it has to be able to pick and place loads, and cooperate with other vehicles.
Laser triangulation navigation does not require tags or floor guides for path following, but involves the application of reflectors in designed positions. These will reflect laser beams back to the vehicle and, based on these, the vehicle will localize itself.
Natural navigation, or SLAM (Simultaneous Localization And Mapping), uses laser scanner measurements to identify, and then match, structures in the environment. This solution is quick and simple in both installation and modification, as it requires minimal changes to the infrastructure.
In both cases the paths followed are virtual.

This project included a third-party package to enable natural navigation in Skilled's AGV. The crucial points of the project are: the learning of the BlueBotics technology (from software to hardware requirements), the design of an hardware structure to allow proper communication between all components, the programming of the AGV's Siemens PLC for both safety and non-safety behaviour and Ignition code developing for mission and traffic management.

This thesis presents the developed project. First, the current structure of the Skilled AGV is defined, followed by a description of the project and an illustration of some code sections.

Contents

Chapter 1

Euroimpianti - Skilled Group

Skilled Group is based in Schio, Italy, with production facilities and offices in the United States and Europe. It was founded in 1973 in Zanè (VI) by Gastone Trecco with the aim of producing, handling and palletizing systems for all the main types of packaged goods. The first developed product was the traditional palletizer, later replaced by the SCARA and Cartesian palletizer and then by the more modern anthropomorphic robot. Today, Skilled Group is a leading supplier of material handling automation and automated production of various types of industrial products. Since 1992, a new product line has been added to production: Automated Guided Vehicles and Laser Guided Vehicles. These products enable proper management of warehouse logistics, ensuring that every available space is utilised in the most effective and productive way. Laser Guided Vehicles and Palletizing Robots fully automate every product handling and storage process, reducing stock management costs and increasing production. Skilled provides automation solutions and product equipment, offering certified training, installation and service for each product [1].

1.1 Skilled Products

1.1.1 Robot Palletizers

The automated palletizing process includes all the activities required to stack different types of products on a pallet or on a slip-sheet in an automated and optimized way.

1.1.1.1 Skilled SCARA Robot

This Skilled product is a robot structured with a SCARA articulated kinematic and with 4 interpolated axes, allowing a 360° working area. Thanks to its high flexibility and the possibility to customize the picking tool for an unlimited range of applications, the Skilled SCARA robot has been used to solve a wide variety of end-of-line product handling problems in the most diverse plant logistics situations.

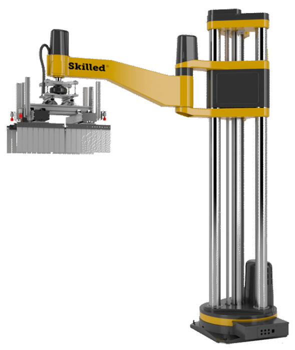


Figure 1.1: Skilled SCARA robot

1.1.1.2 Articulated Robots

Articulated robots are designed with a kinematic structure with revolute joints. This robots can vary from a simple two-joint structures to systems with ten or more joints. An articulated robot can be used for different applications and has a wide coverage of the working area. It is ideal for pick and place applications, rapid product handling and orientation, pallet-box filling and general palletizing.

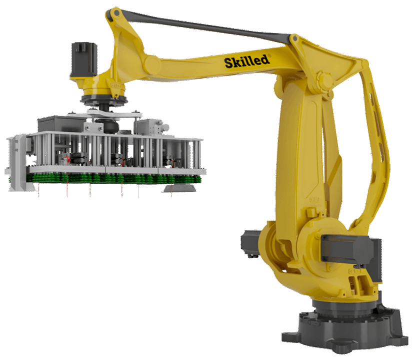


Figure 1.2: Skilled Articulated Palletizer

1.1.1.3 Conveyors

Conveyor systems and packaging equipment are designed for handling and transporting products to and from any location. The products required can be: lifts/lowerators, chains, steel belts, sorting systems, product alignment, diversion and control systems, and product identification systems. This product is equipped with an electric motor that enables the advancement of the required products and raw materials in the product chain.

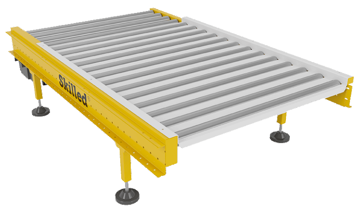


Figure 1.3: Conveyor

1.1.1.4 Gantry

A gantry robot consists of one or more manipulators mounted on an overhead structure that allows them horizontal movement. This system allows pick and place applications in large and wide areas and then the stacking of different objects according to the company's management software. Depending on requirements it can be supplied with a single or double pallet station, with automatic pallet magazine, with slip-sheet magazine and with conveyor for full pallets. The Skilled Gantry robot offers standard payloads of up to 1000 kg and a working area of 75m length and over 15m width. It can provide palletizing solutions with multi-cell gantry palletizers that sort and palletize several products simultaneously.

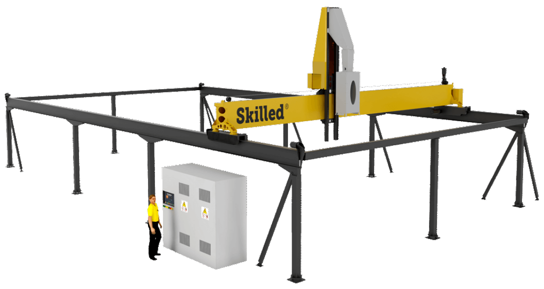


Figure 1.4: Gantry Palletizer

1.1.2 Automated Guided Vehicles

The acronym AGV identifies driverless forklifts that can transport pallets and/or materials to/from programmed positions in production areas and warehouses. The AGV receives orders directly from warehouse management systems or any type of software or hardware interface. AGVs operate continuously in total safety with precisely controlled navigation, acceleration and deceleration to minimise potential damage in material handling.

1.1.2.1 Rail Guided Vehicle (RGV)

Skilled RGV has been created from the LGV project and its operation is, therefore, based on the same general principles. The vehicle moves along a straight track at floor level and is used exclusively for straight-line handling. Its integration with the robot palletizing systems allows complete "end of line" automation. It is equipped with a modular system and therefore has great flexibility.
The absence of electrical connections and the possibility of sinking the rails into the floor facilitate the access to production lines by operators and vehicles.

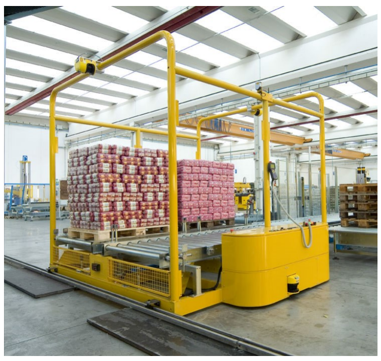


Figure 1.5: RGV

1.1.2.2 Laser Guided Vehicle (LGV)

Skilled LGV (Laser Guided Vehicle) is an automatic forklift vehicle that has a role in the complete automation of the "end of line" production section.
LGVs are usually equipped with forks for a single or double pallet and can also handle 10m pallets lifting. They can also be equipped with gripper, for picking up rolls, or conveyors, for picking up cast products. Thanks to its flexibility, the system can be easily adapted different production needs. The different LGV models differ in lifting height and load capacity [1].

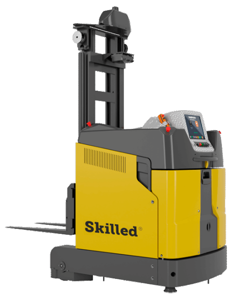


Figure 1.6: LGV

1.2 Laser Guided Vehicles

Laser Guided Vehicles (LGV) are Automated Guided Vehicle (AGV) that use the laser triangulation technique to localize itself in the environment.

1.2.1 Structure

The kinematics aspects to be considered when designing a mobile robot are: mobility, control and positioning.

The first one deals with the possible motions that the robot may follow to reach a final configuration. Mobile robots must be able to reach any position on its plane of motion, with any orientation.

The second aspect deals with the choice of the kinematic variables, generalized velocities or generalized coordinates, that will be directly controlled by the drivers to introduce the required robot motion.

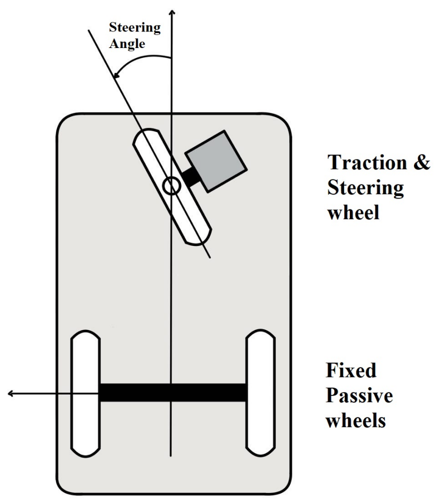


Figure 1.7: Tricycle Model

Finally, the third aspect considers the localization system, included in the robot and used to estimate the actual robot pose (position and orientation), necessary to achieve an autonomous operation [2].

The way the LGV moves in the environment is defined by the tricycle kinematic model. The kinematic model studies the motion of a robotic mechanism regardless of forces and torque that cause it, dealing with the geometric relationships that govern the system and the relationship between control parameters and system's behaviour.

The AGV tricycle model is composed by an active wheel, equipped with a traction motor with velocity control and a steering motor with position control, and two fixed passive wheel for balancing purpose as reported in Figure 1.7. The center of the vehicle is defined as the center of the passive wheels and the traction wheel can be off-centered [17].

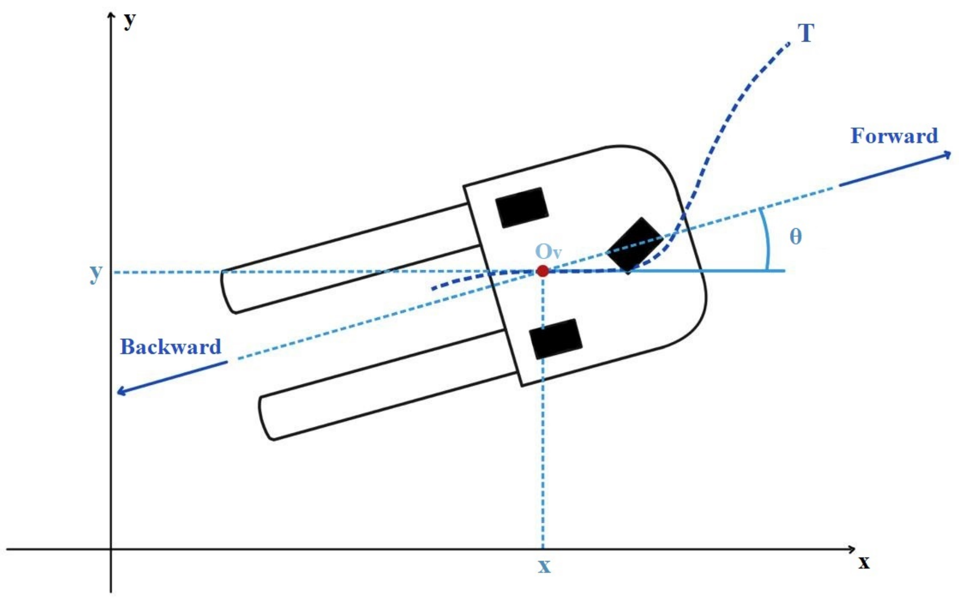


Figure 1.8: AGV Reference system definition where (x, y) are the vehicle horiziontal and vertical position on the map; θ is the heading angle of the vehicle, O_v is the origin of the vehicle reference system and T is the trajectory to be followed

The vehicle reference system is reported in Figure 1.8, where also the forward and backward direction are defined.

Skilled AGV is a non-holonomic systems, which means that the differential equations are not integrable to the final position and the measurement of the distance travelled by each wheel is not sufficient to calculate the robot's final position. To define the final position, additional information on how the movement was performed, as a function of time, is required.
This type of system is described by a set of parameters subject to differential and non-linear constraints.

The maneuverability of a mobile robot is the combination of the mobility available under sliding constraints and the additional freedom provided by steering.

The quantification of the mobile robot maneuverability can be derived using [3]:

- Degree of mobility: is a measure of the number of Degree of Freedom (DoF) of the robot chassis that can be immediately manipulated through changes in wheel velocities.
  In the case of the tricycle model, its value is $\delta_m = 1$ because the speed variations only allow going straight in the defined steering direction.
- Degree of steerability: an increase in Degree of Steerability (DoS) results, eventually, in greater maneuverability but decrease in mobility.
  In the tricycle model case $\delta_s = 1$ cause one independent steerable wheel is present and performs changing in the AGV direction.
- robots maneuverability: is the sum of the degree of mobility and steerability. Therefore in tricycle kinematic case it is: $\delta_M = \delta_m + \delta_S = 2$

The vehicle dynamics is then defined based on the tricycle structure reported in figure 1.9

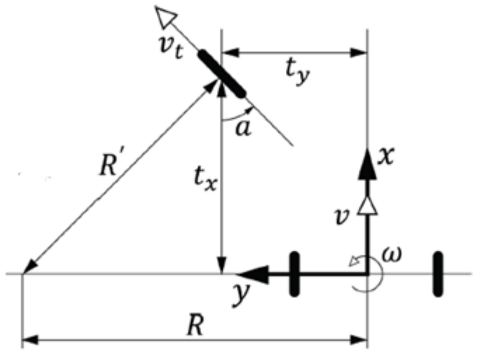


Figure 1.9: Tricycle Model

The kinematic model behaviour can be defined with a set of equations [17]:

$$\begin{cases} \omega = \frac{v}{R} & (1) \\ \omega = \frac{v_t}{R'} & (2) \\ R' = \frac{t_x}{sin(a)} & (3) \\ R = R' \cdot cos(a) + t_y = \frac{t_x \cdot cos(a)}{sin(a)} + t_y & (4) \end{cases}$$

With 1 *and* 2: $$v = v_t \cdot \frac{R}{R'} \tag{5}$$

With 3, 4 *and* 5: $$v = v_t \cdot \frac{\frac{t_x \cdot cos(a)}{sin(a)} + t_y}{t_x} \cdot sin(a) = v_t \cdot \frac{t_x \cdot cos(a) + t_y \cdot sin(a)}{t_x} \tag{6}$$

With 2 *and* 5: $$\omega = v_t \cdot \frac{sin(a)}{t_x} \tag{7}$$

1.2.2 Laser Triangulation

AGV navigation can be developed using different techniques.

Line following navigation requires a specific sensor in the AGV to follow a physical line attached to the ground (magnetic tape, inductive wire). This requires a lengthy installation and the modification will be costly and difficult for the fleet to manage.

The navigation with tags, on the other hand, requires a camera or code reading sensor to read the tags on the floor, so it has the same costs as railway navigation.

Laser triangulation navigation, on the other hand, does not require tags or guides on the floor. A laser-supported NAV positioning system is mounted on top of the LGV, typically centred along the vehicle's width axis and towards the front control panel. The scanner emits a modulated laser light that rotates and covers a 360° angle ten or more times per second, covering several metres. The reflective landmarks, with known coordinates, are installed at several specific locations. Each of these landmarks, which can be either flat or round, send a reflection back to the scanner and this is converted to a coordinate reference. By definition, at least three identified reflections are needed at any time instant in order for the scanner to determine its location in the system.

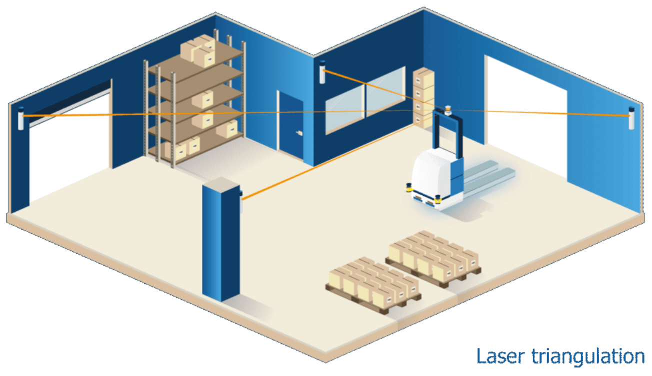


Figure 1.10: Navigation of an AGV with a laser-supported NAV positioning system

1.2.3 Actual LGV Managing and Structuring

The actual AGV is a Laser Guided Vehicle and its movements in the environment are controlled by PLC, that schedules missions, and Router, that provide traffic managing. The overall hardware structure is reported in Figure 1.11.

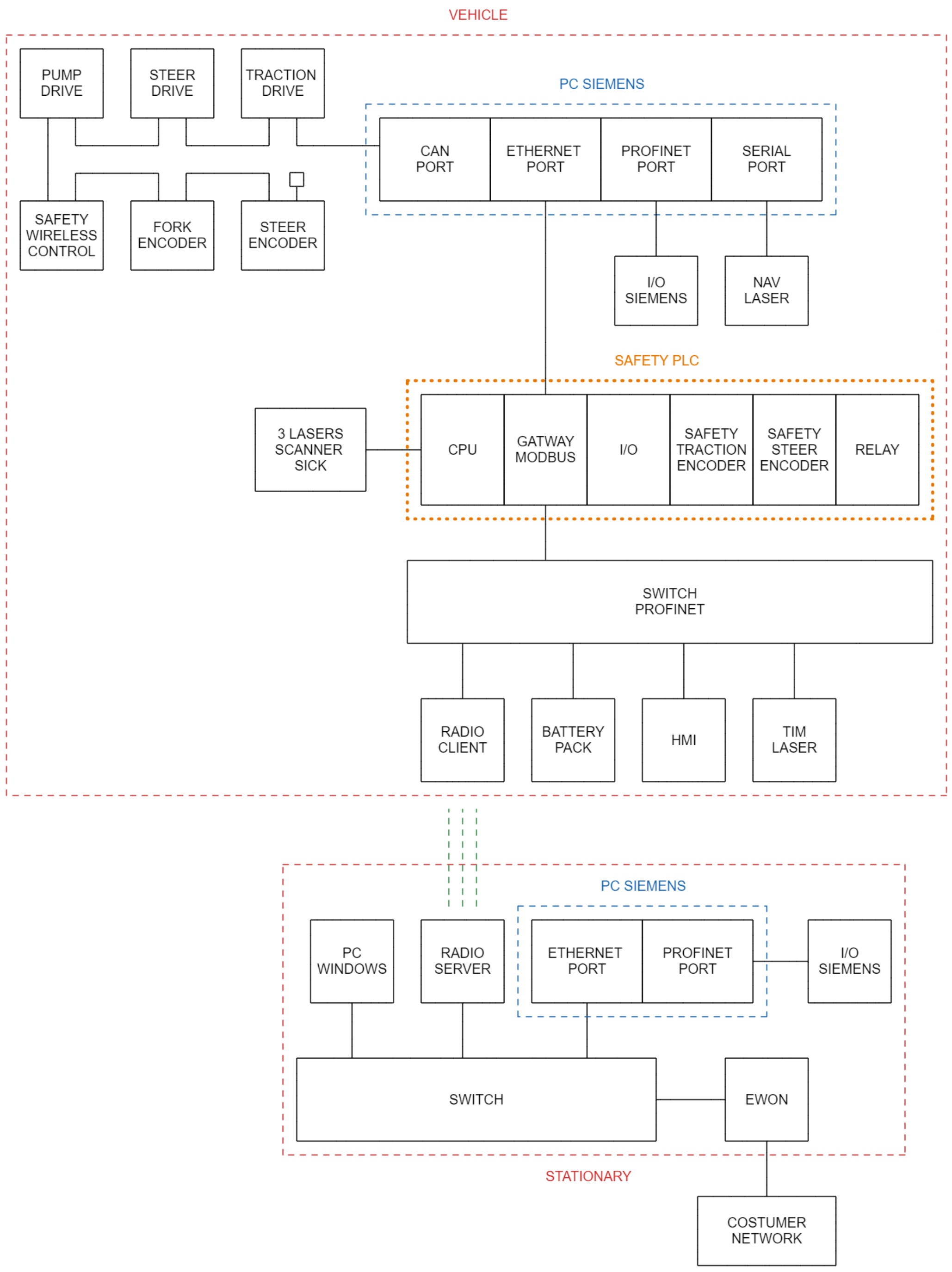


Figure 1.11: Skilled LGV's hardware design

1.2.4 Stationary

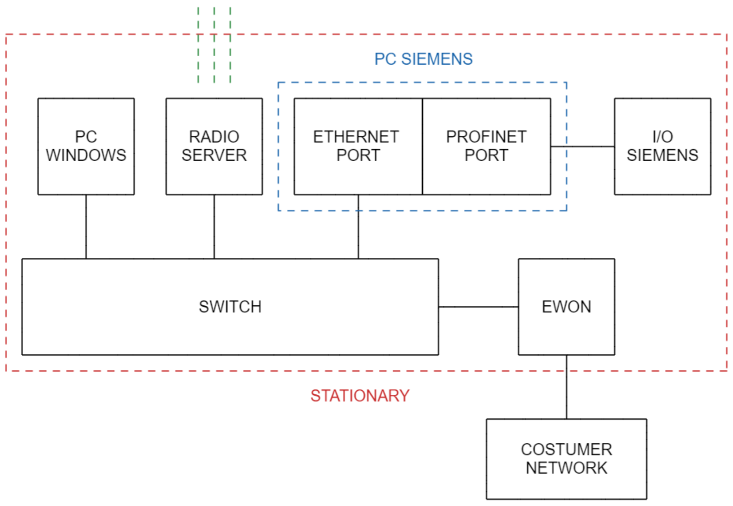


Figure 1.12: Stationary Block Scheme

The Stationary is composed by a Personal Computer (PC) and a Programmable Logic Controller (PLC).
The first one has in it the interface program that shows all the missions, the state of each vehicle, the allowed path in the environment, and a router for the traffic managing.

The router is a software application that relates with the Sstationary and with each AGV by OPC and maintains the data exchange for traffic management.
It receives the mission schedule from the Stationary PLC and provides the calculation of the paths required to complete it successfully.
It first verifies the possibility of compleate it, then sends the planned route to the AGVs.

This is done by sending partial path sequences and a continuous update of the commands. In this way, if the commands of two AGVs have the same sequence, path intersection occurs, and a wait command is given to one of them to avoid a collision.

The second component is a PLC (PC Siemens block) and it is the mission generator. It relates with the signals from the environment, by I/O modules can receive information from presence sensors, and from warehouse system managing.
The Logistic Server is software application that enable to have more information about the warehouse and how the loads are managed and stocked. It generates an event that send a signal to the PC that will generate the required mission.

1.2.4.1 Interface Program

The interface program is defined with a CAD file in which the building plan and the permitted route for LGVs are represented.
This is designed with nodes and links, that defines the paths, and instanced node that represent: parking positions (used to park the LGV until it is employed for a mission), picking and placing stations, charging areas and car-washing station where LGV can leave the load that doesn't have a target destination.

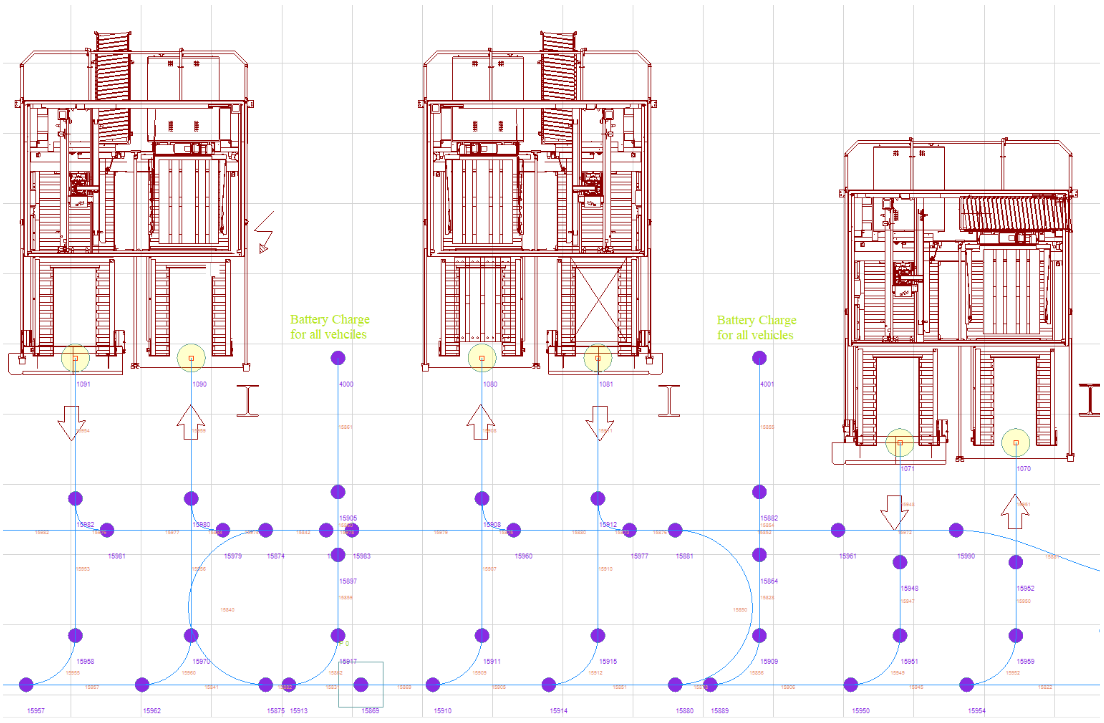

Battery Charge for all vehicles
Battery Charge for all vehicles

Figure 1.13: AGV path layout in CAD

An example can be seen in figure 1.13, where the environment is reported in red and the allowed path are composed by nodes and links. In the image are shown the charger nodes and the picking an placing nodes, colored in yellow.

1.2.4.2 Switch and communication protocols

The stationary block schemes in figure 1.12 shows the Stationary components and their communication. Everything is connected thanks to the Ethernet Switch, so that the PC and the Siemens PC can communicate with all the other involved devices. Ethernet technology provides a set of physical media definitions, a simple frame format and source/destination addressing scheme to move data packets between devices [4].

The Ethernet switch is used to create a network connection between all the connected devices via Ethernet cables and Ethernet ports. It relies on the addressing scheme and sends data packets only to the destination port, limiting the number of data collisions. [5]

1.2.4.3 Radio Server

Radio Server is a radio communication that implement the Client-Server communication protocol and is used to exchange messages and information with each vehicle.

Client-server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Often, clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. Clients and servers exchange messages in a request–response messaging pattern. The client sends a request, and the server returns a response. This message exchange is an example of inter-process communication. To communicate they must have a common language, and must follow rules so that both the client and the server know what to expect.

To further formalise the data exchange, the server can implement an Application Programming Interface (API) that is an abstraction layer for accessing a service. A server can receive requests from many different clients in a short period but a computer can only perform a limited number of tasks and then it relies on a scheduling system to prioritise incoming requests from clients [6].

This Radio Server/Client communication protocol in Skilled AGV is governed by the Ethernet-IP protocol, so each vehicle will have its own IP and each vehicle component also has its own. The two strategy to avoid wrong communication regards the Vehicle's components IP.

- In the first case each component has different IP from the same component in other vehicles so everything is unique. There is only one address family of IP.

- In the second case, instead, each component has the same IP in each vehicle but what differs is the Vehicle Radio Client IP. The radios IP are on the same address family and, once the connection is establish with the selected vehicle, the component IP can be reach uniquely with NAT connection.

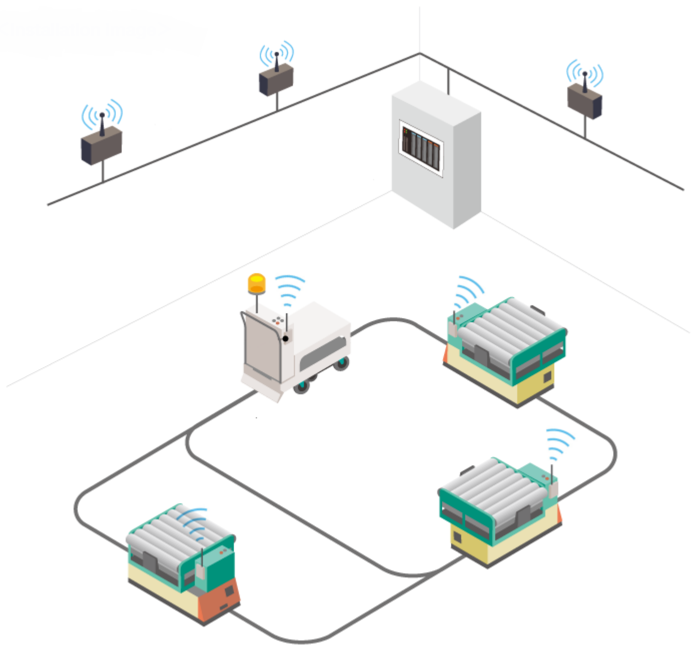


Figure 1.14: Client-Server AGV communication

1.2.4.4 PC Siemens

The PC Siemens block represent the PC that is equipped with an image contained in a SD Memory that has in it the PLC software, files for interface with the LGVs, and the LGV's Firmware to communicate with the LGV's components.
The I/O module is connected to the Siemens PC and act as mediators between the processor and the input/output devices. The input modules receive signals from switches or sensors and send them to the processor, the output modules take back the processor signals to the control devices like relays or motor starters.

The signals from the environment that are used in Siemens PC for mission managing are, for instance, the signals from automatic doors, from the sensors in the picking and placing positions, fire sensors to send the LGV in safe positions to not occlude the emergency exits, and many others.

It also exchanges this information with the interface program and router to develop proper mission and traffic management. This is enabled by the Ethernet port IP, which is necessary for proper communication with the Ethernet Switch and then with all other Stationary's components.

1.2.4.5 EWON

The EWON block represents the way to remotely service installations, so that technicians can solve problems and change the code in the rest of the world.

Ewon routers are built to fit within the automation panel and can communicate with both Ethernet and serial devices. The Ewon device makes an outbound connection via UDP (User Datagram Protocol) or HTTPS (HyperText Transfer Protocol over Secure Socket Layer) to the VPN (Virtual private network) servers. UDP, HTTPS are communication protocols with different characteristics: UDP is a simple connection-less communication model with a minimum protocol, HTTPS is used for secure communication over a computer network.
VPN is a private network, established as a connection between parties using a shared, public transmission protocol. Using the VPN client, authorized users are able to log into their account and connect to their Ewon router anywhere in the world.

The Ewon router is typically configured as a DHCP (Dynamic Host Configuration Protocol) Client so it receives network settings automatically [7].

User devices are used to remotely access the installations and this can be done also with PLC programs. Talk2M is an industrial cloud, secure, reliable and scalable.

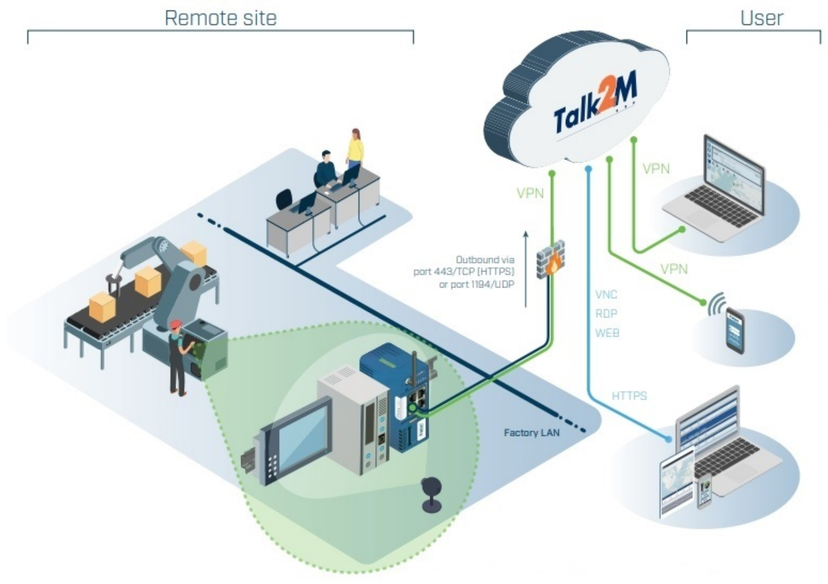


Figure 1.15: Ewon connection

1.2.5 LGV

The actual Laser Guided Vehicle structure is reported in figure 1.16 and it is the upper section of the figure 1.11.

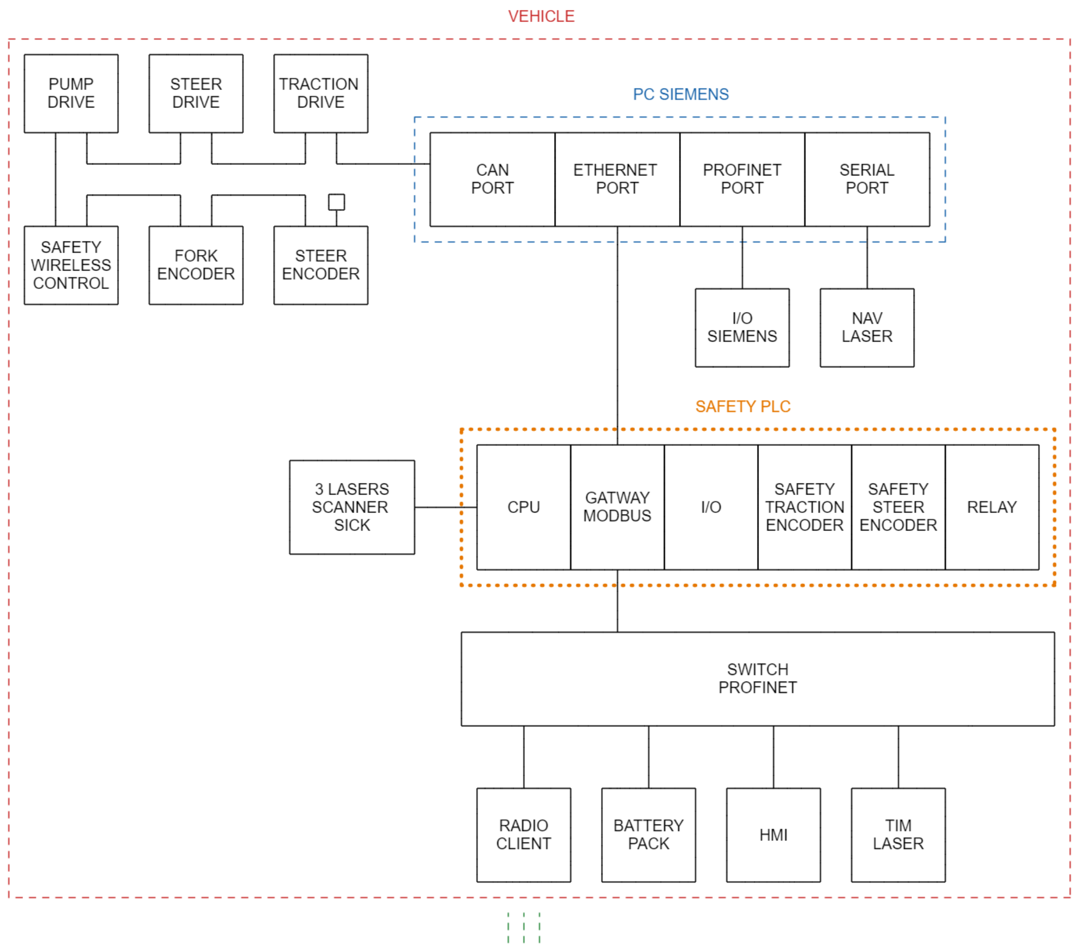


Figure 1.16: LGV Block Scheme

The block scheme reported in Figure 1.16 represent the hardware structure of each Skilled LGV. These components collaborate to enable proper handling of all devices and ensure the correct vehicle behaviour.

The PC Siemens block and Safety PLC block are the two most important devices in the LGV. Those two receive information from all the devices in the LGV structure and manage it by sending as output the proper signal to handle every case.

1.2.5.1 Laser Scanners

Skilled LGVs perform their localization with laser triangulation, but collision avoidance and safety behaviour are performed with laser scanners and safe areas definition around the vehicle.

In the first LGVs the safe stopping was performed with bump strips, contact sensors used to allow an LGV to stop safely without collisions, but the vehicles had to move at low speed to react in time.

Today, safety laser scanners allow the AGVs to move at higher speeds because they provide the safe non-contact detection of personnel and obstacles far ahead of their path, allowing higher speeds.

The design of these safety areas takes into account many aspects of the application of industrial autonomous vehicles, such as the environment, the break system and a comprehensive risk assessment.
The use of laser scanners in stationary and mobile applications enables a machine or vehicle to slow down or stop safely as soon as it detects a person, body part or unexpected obstacle within the protective field.

Safety laser scanners use the time-of-flight principle, in which a light pulse is transmitted, reflected and then detected. The distance (d) from the object to the scanner is calculated using the return time of the beam (Δt) and the speed of light ($c = 3 \cdot 10^8 \frac{m}{s}$):

$$d = \frac{c \cdot \Delta t}{2}$$

A mirror inside the scanner rotates this light beam, allowing measurements to be taken around a radius in a plane.
This means that the scanner can build up a profile of the surrounding area, and operating entities can configure different fields in the scanner that can be used to activate and deactivate different outputs for use in safety functions. Some scanners can also be used for multiple safety functions due to their ability to evaluate multiple fields simultaneously [8].

With Laser Scanner it is possible to define Warning and Protective fields (Figure 1.17) and then Warning and Protection Areas around the LGV, Figure 1.18.
The safe behaviour of the LGVs is defined by the detection of obstacles in these areas and is encoded in the Safety PLC.

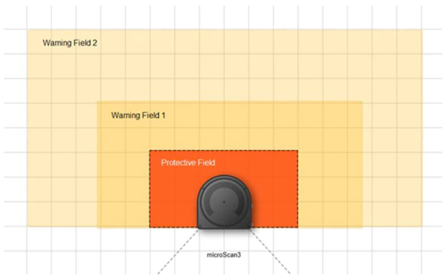


Figure 1.17: Lasers fields

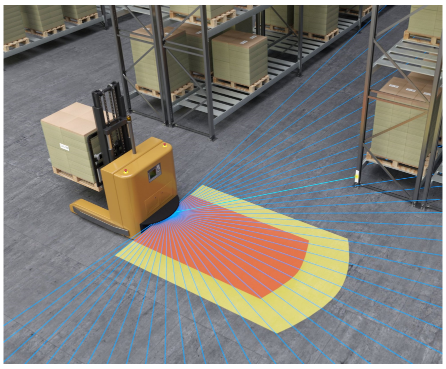


Figure 1.18: Safety AGV areas

1.2.5.2 Safety PLC

To be considered a Safety PLC, a PLC must meet a number of strict international standards. It supports all the applications that a standard PLC does, however a safety PLC contains integrated safety functions that allow it to control systems safely [9].

A safety PLC is designed to achieve two important goals: Not to fail and, if unavoidable, to fail only in a predictable and safe way. It achieves these goals with its redundant microprocessors, eliminating the need for safety relays to create redundancy. In addition, it has built-in diagnostics to continuously monitor inputs and outputs.

If an internal fault or error is detected, the PLC shuts down safely. [10]

Figure 1.19: SICK Safety PLC

The safety PLC (Figure 1.19) is programmed using SICK's Flexi Soft Safety Controller and the SICK Safety Designer software, correlating signals from the motor safety encoders, laser scanners (Figure 1.20), internal software bits and other hardware inputs. SICK's safety laser scanners and the Flexi Soft Safety

Controller with EFI-Pro Gateway, a safe communication protocol, are the basis for an integrated, standards-compliant safety design.
Encoder information are used to dynamically change the field size on the safety laser scanners via EFI-Pro communication according to speed and direction [11].

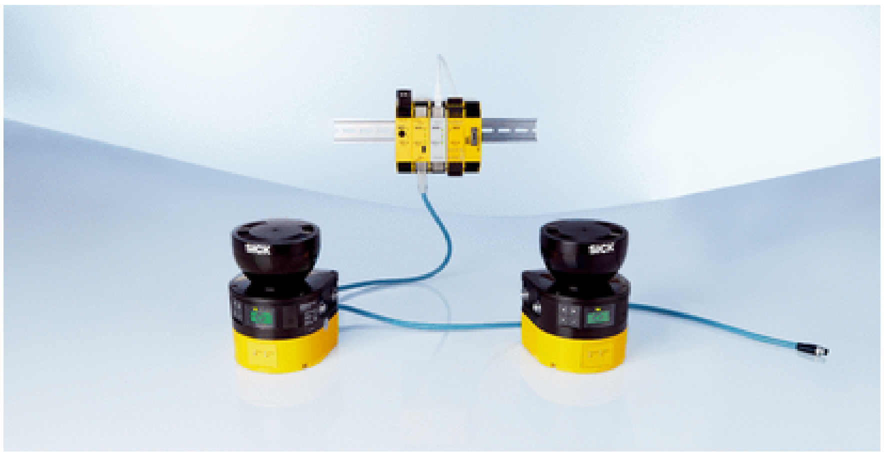


Figure 1.20: SICK Safety PLC and Laser Scanners

SICK Flexi Soft allows the application of complex algorithms and logic so that the LGVs can be configured to work completely independently.

If the safety laser scanner detects an obstruction in its warning fields, the safety controller instructs the vehicle control system to slow down and then speed up again once the field is clear, allowing efficient and productive operation.
If the inner protective field is breached, the Flexi Soft ensures that all drives are stopped immediately and are prevented from starting up again until it is safe to do so. The design of the protective fields is carried out taking into account the specifications of the laser and encoders to provide an accurate value that guarantees vehicle's safety.

Starting with the stopping distance (which includes the vehicle's braking distance, the distance travelled during the response time of the safety laser scanner and the response time of the safety control system), the other factors to be added are: a general safety supplement = 100 mm, a supplement for any reflection related measurement error of the safety laser scanner, a supplement for any lack of ground clearance of the vehicle, a supplement for the reduction in the breaking performance of the vehicle as defined in vehicle documentation.

It is advantageous to keep this summed value as small as possible so that: Industrial autonomous vehicles can work in closer proximity to each other, they do not need large clearance space around them, objects and people are less likely to cause stoppages or slow downs of the vehicle, the vehicles can move much faster with smaller fields and increase availability.

The Safety areas can be defined as function of the installation path, in some cases the LGV has to deal with situation in witch the laser scanners will stop the vehicle without obstacle in front of it. For instance in some case of picking/placing the forks will cover the sensor, shutting down the vehicle and compromising the achievement of the target.
In this case the Waning and Protective fields are imposed to be almost null (Picking and Placing Area definition), and the AGVs can continue its mission. An accurate design of all the Safe Areas and Safety definitions is carried out to ensure a safe and correct behaviour of the AGV.

The three Laser scanners in the Skilled AGV are connected via Ethernet/IP to the CPU of the safety PLC (which contains the Flexi soft program with area definitions). Other safety inputs enter the PLC and are, for instance: the brakes, the reset power button, the enable power supply, the battery charge, the automatic/manual selector and the chain tension sensor signal.

The Gateway Modbus (the grey module in figure 1.19) is used to exchange signals with Modbus communication between the Siemens PC and the Safety PLC.
It also functions as bridge to allow the Siemens PLC to relate with the device under the Safety PLC in the AGV block scheme (Figure 1.16) like Battery Pack, HMI, TIM Laser and Radio Client.

1.2.5.3 PC Siemens

As in the Stationary, the LGV Siemens PC has an image, and contains the software of the PLC an the LGV's Firmware.

Figure 1.21: PC Embedded Siemens

A firmware is a "software" that provides basic machine instructions that allow the hardware to function and communicate with other software running on a device. Firmware provides low-level control for a device's hardware, so it is not generally designed to be user friendly. It is used to run user programs on the device and can be thought of as the software that enables the hardware to run. Firmware may be written into read-only memory (ROM), erasable programmable read-only memory (EPROM) or flash memory [13].

The Siemens PC block is one of the main components of the AGV block scheme, as it communicates with everything and relates to every device. The schematic block is divided into four ports that have different communication protocols to enable the correct exchange of information.
The PC is in charge of handling all tasks and cases, the software in it can interact with external inputs and then provide input signals to devices to perform specific tasks.

The Siemens PC is the one that translates the missions received from the Stationary and Router via Radio Client into commands to be given to the steering and traction motors to reach the destination. This is a CanOpen communication and is connected to the Can port of the Siemens PC.

The Safety Wirless Control block allows the communication with the joystick for manual navigation. CanOpen communication allows to receive the signal from the joystick and let them arrive to the PLC

The devices, as shown in the block diagram in figure 1.16, are connected, from the hardware point of view, in serial mode starting from the traction drive and ending after the steering encoder with a terminal resistance but the Can/ID communication is in parallel mode.

The Input/Output module is one of the main modules of the PC. Through it, the PC can communicate with all devices by receiving and providing information or tasks. The communication is governed by the Profinet communication protocol. Based on the input provided by the vehicle status, the PC gives the commands for the flashing of the AGV's led blinking, which is used to give a rapid information about the vehicle's status.

Chapter 2

Project Description

2.1 New Design for LGV

A third-party component was installed to convert the AGV from laser triangulation to natural navigation.

The complete package consists of BlueBotics AntLite^{+}, an on-board navigation system that allows each AGV to navigate autonomously in the environment, and AntServer, the fleet management software for the installation site that performs all high-level tasks, such as mission scheduling and deadlock-free traffic control. AntLab is the software used for vehicle and installation configuration [17].

2.2 BlueBotics

BLUEBOTICS
Your Vehicle Navigation Partner

BlueBotics is a Swedish company that provide natural feature navigation solution to AGVs. Its Autonomous Navigation Technology (ANT) was created in order to make AGVs easier to install and operate. ANT natural feature navigation (sometimes called ‘natural navigation’, ’free navigation’ or ‘SLAM navigation’) uses a vehicle’s existing safety laser scanners to firstly identify and then match permanent features in the environment, such as walls, pillars, and machines. This complete natural feature navigation solution calculates the vehicle’s position (localization), controls the vehicle motion, and interfaces directly with the vehicle's safety laser scanners [15].

ANT uses laser scanner data and odometry to provide a robust localization of the vehicle in the map, using permanent structures (features) in the environment as references. The accuracy is very high, varying in ± 1 cm and ± 1°.

ANT is compatible with all AGVs types including tricycle, differential, Ackermann model based and omnidirectional.
The obstacle avoidance allows the dynamic navigation of the vehicle around blockages without stopping, except for "too far from the path" error.

The transfer of mission data from computer to vehicle happens once (instead of continuously sending commands from the server to the vehicle), reducing the network requirements.

The overall structuring of ANT technology is represented in figure 2.1.

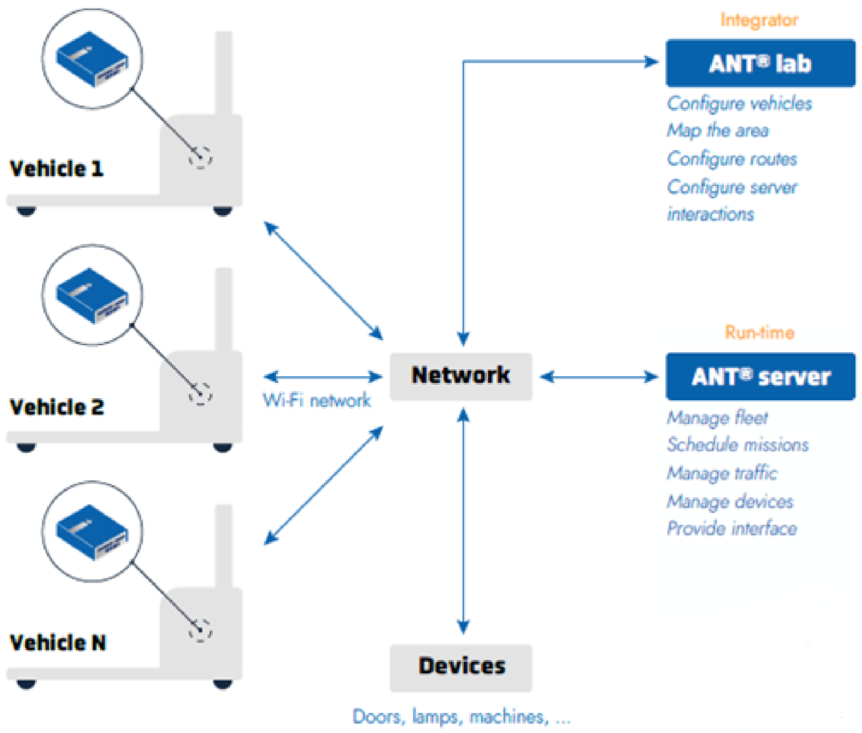


Figure 2.1: ANT Structure

The three main components of the Autonomous Navigation System technology are: AntLite^{+}, AntServer and AntLab.

- AntLite$^+$ is the onboard navigation system, that allows each AGV to navigate autonomously in the environment so both localization and navigation calculations are embedded in the AGV. The network connection is only required for mission dispatch, monitoring, traffic in sensitive areas and connect devices, but not for localization and navigation. Each AGV is equipped with its own AntLite$^+$, and one instance of AntServer is installed.

- AntServer allows the fleet management, the selection of the right vehicle for each mission and the coordination of vehicles in path intercepts (waiting points). It performs all high-level tasks such as mission's scheduling, deadlock-free traffic control, battery charge management, interface with external software and external device controllers. It also allows simulation of vehicles and missions.
  The dedicated AntServer's API allows the mission managing according to the company's management system [16].

- AntLab is the software used for configuration of both vehicles and installation. It allows to perform vehicle configuration and calibration, the environment mapping for localization, the drawing of the routes, the definition of actions, the configuration of devices such as chargers, doors and lifts using AntServer. AntLab is used to set up a fleet of AGVs equipped with AntLite$^+$ in an industrial environment, but the continuous running of AntLab is not necessary to enable the operation of the AGVs [17].

2.3 Hardware design

The first thing to consider in the new Skilled AGV design was the definition of the appropriate hardware structure to provide the right communication between all components and its ability to provide the same actions of actual Skilled AGV. As stated in the AntLite$^+$ manual in the configuration section [17], a PLC is required to perform all the tasks that the actual Skilled AGV can perform.

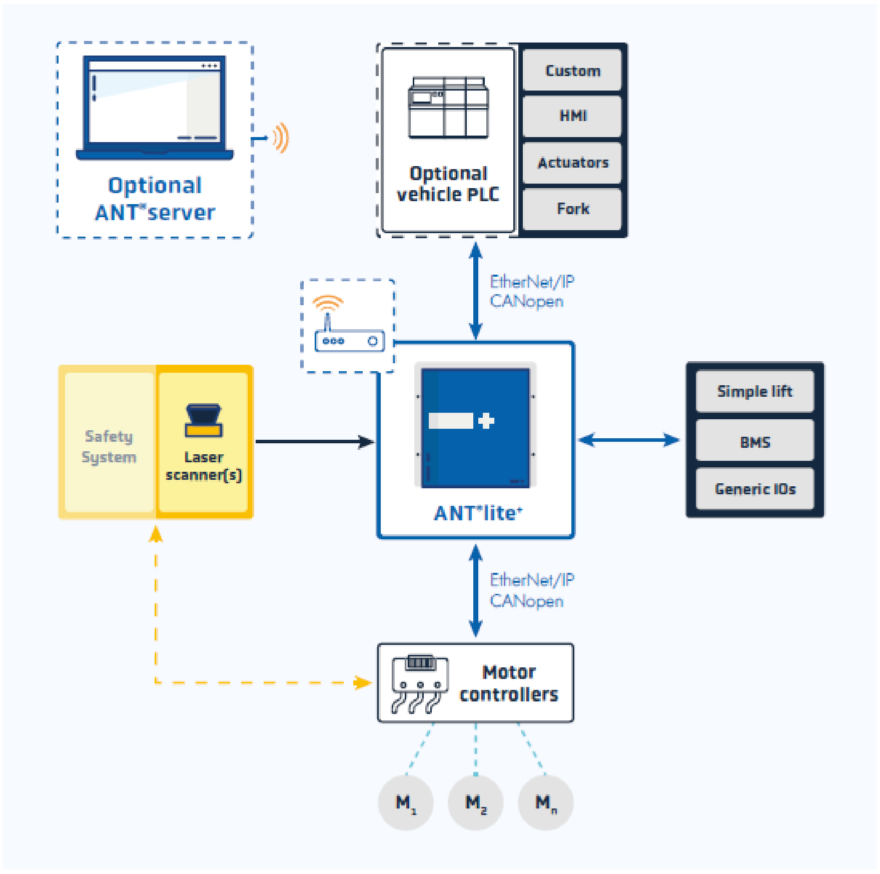


Figure 2.2: ANT and AGV devices

As a new project, all the possible design ways was considered, starting from the PLC supplier. In industrial automation, the two main PLC manufacturers are Allen Bradley (Rockwell Automation) and Siemens. The two PLCs differs in many aspects, from the programming language to the used communication protocol.

The comparison between the adaptability of the Allen Bradley PLC and the Siemens PLC in the project was the first step in the project planning.

The main differences between Rockwell and Siemens PLC are [19]:

1. Rockwell and Siemens PLC have similar speeds, reliability and has almost the same number of output.

2. Siemens does not require a rack or Siemens power supply for the rack, any external 24VDC works, instead Allen Bradley requires both an Allen Bradley Rack and a Allen Bradley power supply.

3. Rockwell PLC is considered more intuitive and user-friendly to program than Siemens controllers.

4. The main difference is them communication protocols. Allen-Bradley controllers support North American protocols such as DeviceNet, ControlNet and Ethernet IP. Siemens uses European protocols such as Profinet-Profibus, ASI, MODBUS or MODBUS TCP/IP.

During the definition of the new Skilled AGV hardware structure, the most relevant difference between the two hardware schemes was related to the PLC communication protocol.

This was a crucial point because AntLite$^+$ supports the Ethernet/IP protocol to communicate with Laser Scanners (needed also in the Safety PLC) so, if the Siemens PLC has been chosen, a Gateway Converter is required to allow the correct communication.

Initially the Rockwell PLC was preferred, as it could support direct communication with AntLite$^+$ and Laser Scanner in Ethernet/IP. A solution to make the Siemens PLC supported as well was found in the PN/MF Bus Coupler (defined in Chapter 2 Section 2.5.3), which allows a proper Profinet and Ethernet/IP conversion.

The two hardware projects, based on the PLC choice, were identical except for this coupling device. Euroimpianti's choice was to proceed with the Siemens PLC for standardization as they already use it for them products.

The block scheme of the hardware design is reported in figure 2.3. As can be seen both Stationary and AGV structuring changes from the one of the LGV described in Chapter 1, Section 1.2.3.

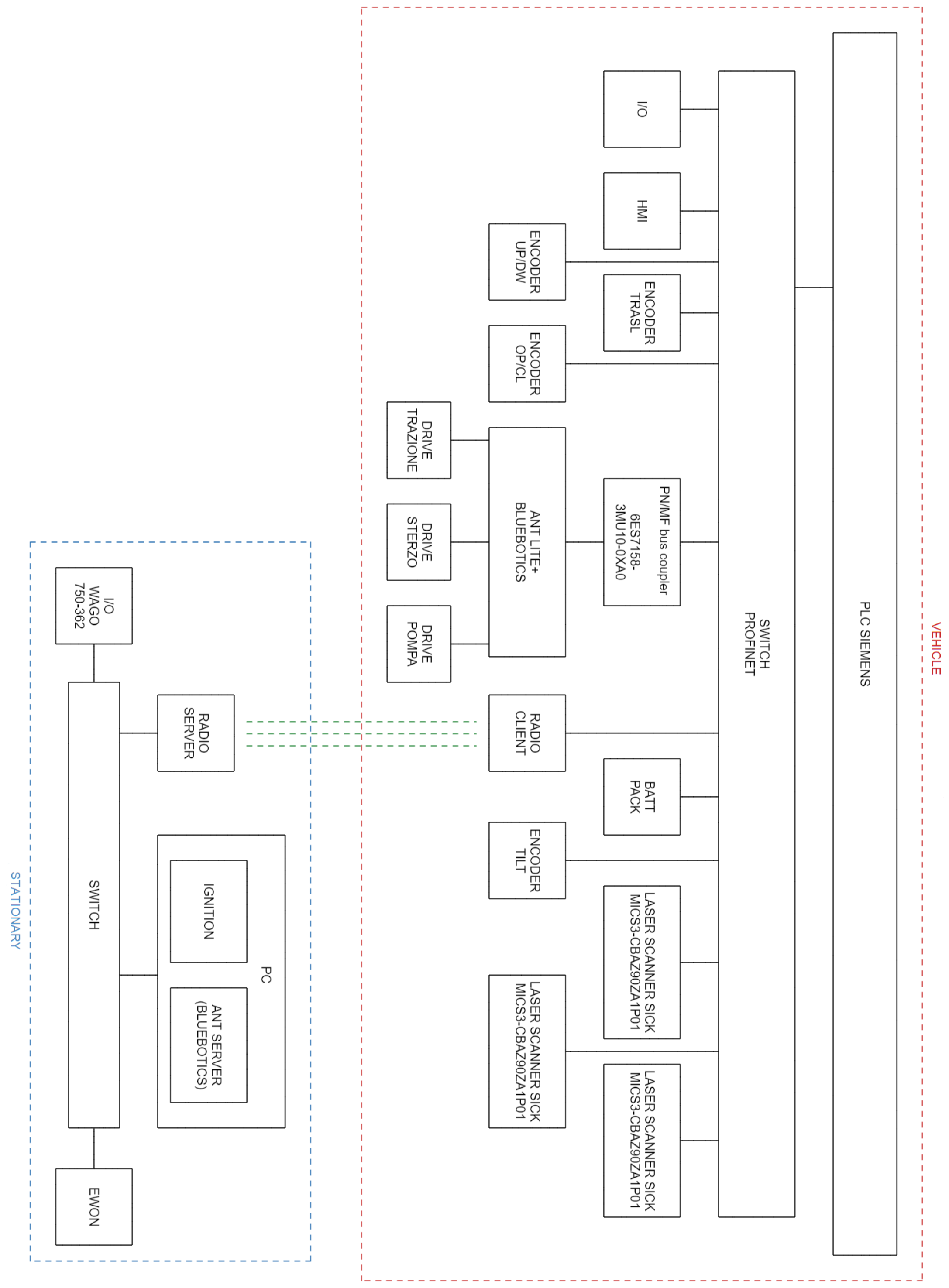
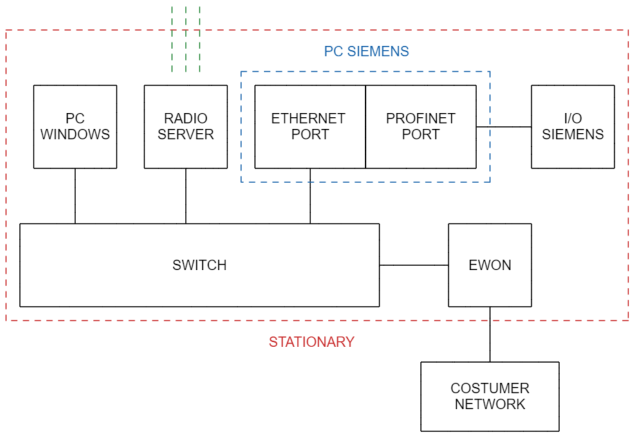


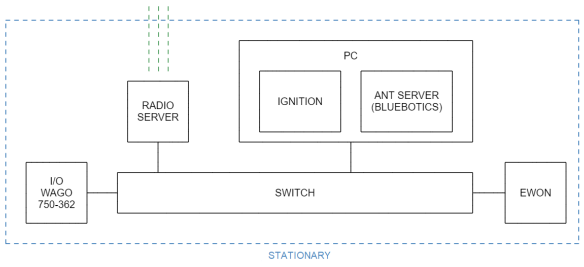
Figure 2.3: New Skilled AGV block scheme

2.4 New Stationary

The figure reported below represent the comparison between the actual Stationary structure (figure 2.4a) and the new designed one (figure 2.4b).



(a) Actual Stationary design



(b) New Stationary design

The new Stationary differs from the actual Skilled Stationary, but the basic idea is the same. The PC Windows has in it the AntServer provided by BlueBotics, which takes care of mission scheduling, traffic management and exchanging information with the new AntLite$^+$.

The radio-server is still present and it is required for the Client/Server hierarchy messaging, as defined in Chapter 1, Section 1.2.4.3.
Ignition is a SCADA program that, regardless of brand, model or platform, is able to dialogue with any equipment in the plant.
It can communicate with the external environment with the I/O modules and Modbus communication. In the new Stationary the I/O module is connect directly to the switch and the PC Siemens is not needed anymore.

Therefore Ignition, installed in the windows PC, will replace the Stationary's PC Siemens to schedule missions. It is also used to define the human interface and exchange information with AntServer.

2.4.1 AntLab

AntLab is the interface software developed by BlueBotics, which replaces the CAD file definition in the actual Stationary programming. The basic idea is quite similar: as before, the map of the environment is used to define the path that the AGVs must follow in order to move correctly in space and achieve target missions. This configuration software is used for [17]:

- connect to one vehicle at a time, control it and modify its properties,
- map the environment and edit the map,
- define the routes and the actions that the vehicle will follow along the path,
- test/debug the routes, paths and actions using troubleshooting tools.

AntLab files are called projects. Generally, an AntLab project corresponds to an installation with the required number of vehicles. It contains all necessary data: vehicle list and properties, maps, routes and actions.

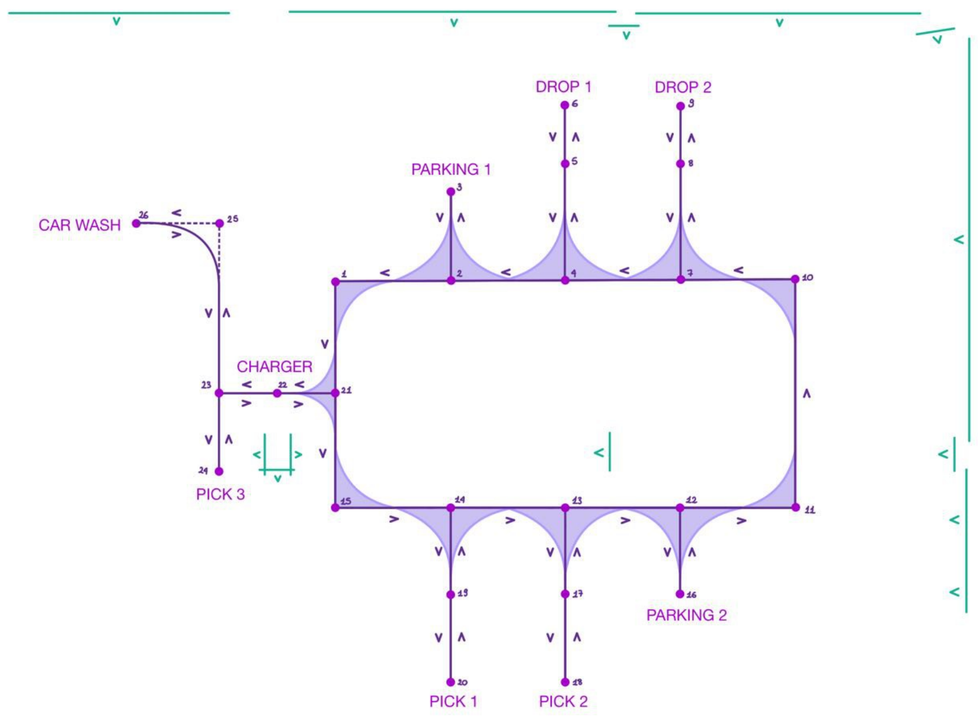


Figure 2.5: AntLab project example

Figure 2.5 shows a sketch of an AntLab project. The allowed paths are still defined with the node-line method and each node can be associated with an action like: charging, parking, picking and placing, car wash (special station where AGVs can leave the load in case the delivery point is not empty or if the mission was aborted during the transfer) and many others.

Green lines, on the other hand, represents the environment features, those objects in the environment that are fixed and which BlueBotics technology uses for its localisation.

Each path line is coupled with and arrow that forced the only possible direction of the AGV in that segment. The areas in light purple are those that allow the AGV to perform the curve. They are defined by AntLab based on the calculation of the AGV's speed in that section. Speed and curvature radius bounds can also be set manually by the developer.

The allowed path design can be done starting from the Laser Scanners data of the AGV moving in the environment or from a CAD file of the plant. The paths are then defined in the AntLab designer. As explained in Chapter 3, Section 3.1.1 Once everything has been set up and configured, all this information are stored in the vehicle, which means that AntLab does not have to be opened or the vehicle connected in order to have a running vehicle.

The vehicle is insert in the project with a firmware called bootfile.
There exist a bootfile for each type of vehicle and contains all the common parameters for this type like: motors, kinematics, laser types, maximum speeds and controller gains.

Added to these parameters are the calibration parameters of the individual vehicles, which are unique and take into account the small mechanical differences in a fleet of vehicles. Typical vehicle calibration parameters are wheel radius, laser position and odometry quality.

These values are stored in the flash memory of AntLite$^+$ and can be erased via a "Reset to factory default".

The vehicle parameters overwrite the default platform parameters and should only be calibration parameters and not critical parameters to ensure that all vehicles in a fleet behave the same way.
When connecting to a vehicle from an AntLab project, an automatic check is performed to ensure that the vehicle and project parameters are consistent.

2.4.2 AntServer

AntServer is a program designed to run from a computer or server at the installation site, it is an advanced mission and fleet management software for AGVs, regardless of vehicle type and kinematics.

AntServer is coupled with AntServer API, a REST-based programming interface to connect AntServer to other software systems. It allows full integration with an existing Warehouse Management System (WMS) or the design of specific user interfaces. Every possible interaction with AntServer is available via the AntServer API, from mission creation to alarm monitoring.

All communication between AntServer and the vehicles are managed through a wireless network. This is only needed when: the vehicle has been assigned a mission and starts it, the vehicle has finished a mission and notify it, the vehicle will pass a crossing soon and needs an acknowledgement from AntServer and when the vehicle needs to communicate with external hardware [21]. The AntServer schematic functionality is shown in Figure 2.6.

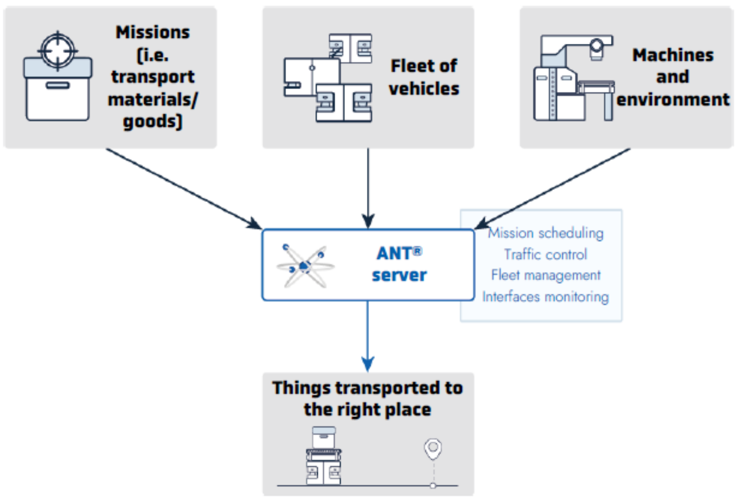


Figure 2.6: AntServer functionality

This means that the vehicle can drive autonomously most of the time and exchange information just when required, thanks to the navigation algorithm directly integrated in AntLite$^+$. This results in minimal data exchanged, which can be done over low bandwidth and so vehicles are always able to have a connection, even if weak.

Through its interfaces, AntServer constantly receives mission requests from external sources such as WMS, operator and machines. Generally, a mission defines a task, but not the vehicle that has to perform it, so AntServer optimally assigns missions to vehicles considering mission priority, vehicle availability, their position on the map, mission deadline, payload/vehicle type and mission.

Vehicle traffic management is based on vehicle map and space sharing. They must follow a common set of traffic rules to avoid collisions and ensure smooth traffic without deadlocks.
AntServer managing is based on the following concepts [21]:

- AntServer processes each path intersection as a single access resource, so whenever a vehicle reaches an intersection it automatically asks AntServer if the access is granted. On the other hand, when it leaves, it asks AntServer to clear the intersection. In this way only one vehicle at a time can access in the intersection;

- AntServer can count the number of vehicles allowed in each region of the graph. Since it supervises the entire fleet at an higher lever, it can manage the correct access to cells;

- In some cases is necessary to manually define additional traffic rules. This usually involves limiting an area to only one vehicle, because there is not enough space or because the complexity of the graph can lead to complex traffic situations. For this reason, a region can be manually restricted to the access of only one vehicle at a time.

In general, vehicles in a plant have a behaviour that changes according to specific positions. For example, a forklift must move its forks when picking up a pallet, slow down in a dangerous area or emit a sound when driving backwards.

These behaviours are handled in AntServer with the concept of Actions. Some are implicit, such as slowing down in a curve, and are handled automatically by AntLite$^+$. Others are explicitly defined by the integrator to serve a purpose, such as pick up a payload.
The configuration of an installation and the scheduling of actions are carried out with AntLab and stored in a project file. When the project is ready, it can be transferred to AntServer so AntLab is not used for run-time, as long as there is no need to modify the project.

An important aspect of actions in AntServer is that the logic is not relate to time constraints, but to position constraints: a vehicle will activate its actuators at a specific position because there is an action to be perform there. This allows a clear process to be defined for installation.

2.4.3 Ignition

Ignition is an Integrated Software Platform for SCADA systems, released by Inductive Automation, based on SQL Database-centred architecture. SQL stands for Structured Query Language and is a specific language used for data management in a relational database management system. Ignition platform has three main components:
the Ignition Gateway, the Designer, and the Runtime clients.

- The Ignition Gateway is the main software service that manages everything in Ignition. It is a single application running as a web server and accessible through a web browser. It connects to data and PLCs, executes modules, and communicates with clients. When the Gateway server is running, it can connect to a device, to a database, launch the Designer and launch a Vision client or Perspective session as represent in Figure 2.7.
- The Designer brings all data, systems and developers together in one simple integrated development environment, designed specifically to realise industrial applications quickly and efficiently.
- The web-launched Vision Clients in Ignition are the "runtimes" of the Vision module. One or more Clients can be launched to display the projects created in the Designer [22].

Ignition is a server software that acts as a hub for everything in the plant for total system integration, regardless of brand, model or platform.

Any type of industrial application can be created for desktops, industrial displays and mobile screens as Ignition Designer combines a rich component library, powerful design and scripting tools.

The development power of Ignition is extensible through the addition of fully integrated software modules. Each module provides distinct functionality such as Real-Time Status control, alarms, data acquisition, scripting and scheduling.

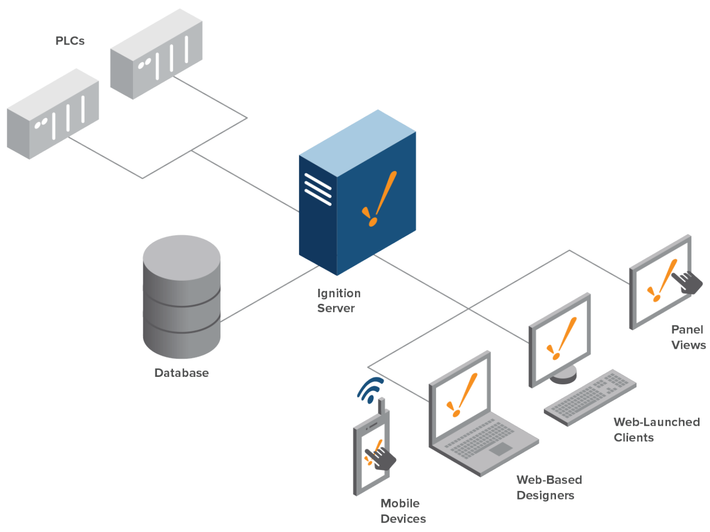


Figure 2.7: Ignition structuring

Ignition is programmed to optimise the flow of data and to display actual tag values in real time. It is also possible to start and stop processes, monitor multiple data points in multiple locations and check the status of the entire plant at any time [23].

Ignition Designer provides an environment to create optimized Human-Machine Interfaces (HMI), creating an application that displays HMIs with historical trends, alarms, and all crucial aspects of the installation required by the operator.

2.4.3.1 Ignition and Bluebotics

AntServer API provides a convenient way to interface AntServer with Ignition to provide the fleet management with a production software or a warehouse management system [24]. AntServer API is based on REST (Representational State Transfer).

A REST API, also known as a RESTful API, is an application programming interface conforming to the constraints of the REST architectural style, which allows interaction with RESTful web services. APIs (an acronym for Application Programming Interfaces) are a set of definitions and protocols by which application software is realised and integrated. They can be regarded as a contract between an information provider and the user receiving that information: the API establishes the content requested by the consumer (the call) and the content requested by the producer (the response). The API then acts as an intermediary between users or clients and the web resources or services.

REST, instead, is a set of architectural constraints, not a protocol or a standard [25] that allows clients to access resources provided by the system via URIs (Uniform Resource Identifiers) and the following four HTTP methods:

- GET: only retrieve the representation/information of the resource and do not modify it in any way.
- PUT: used to update an existing resource. If the resource does not exist, then API may decide whether or not to create a new resource.
- POST: used to create a new resource into the collection of resources.
- DELETE: APIs deletes resources identified by the request URI. [26].

When a client request is sent via a RESTful API, it transfers a representative state of the resource to the requestor. The information, or representation, can be delivered in one of the allowed formats via HTTP like: JSON (Javascript Object Notation), HTML, XLT, Python, PHP or plain text.

JSON (JavaScript Object Notation) is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. Although originally derived from the JavaScript scripting language, JSON is a language-independent data format [27].

The JSON format is employed by AntServer because it is language-independent and easily readable. In particular AntServer will always act as a slave, responding to requests from the client. This means that AntServer will never send data without an explicit request from the client. To monitor the status of objects in AntServer, the client must poll the information regularly.

Going into detail, AntServer sends a response to each request and the returned JSON data is encapsulated in the following envelope [21]:

```
{
"payload": { ... },
"resultinfo": [...],
"retcode": 0
}
```

The *payload* field contains the requested information and is specific to each command.
The *resultinfo* array contains information about the items specified in the request.
The *retcode* field is 0 when the command is successful. In case of error, a non-zero value is returned with additional information [21].

The main requests used with AntServer API [21]:

- Login: URL: . . . /wms/monitor/session/login.
  The aim is to send identifier and password to AntServer, to establish a secured connection, which responds with a session token. This session token must be used with all other requests to ensure safe transactions.

- Vehicles: URL: . . . /wms/rest/vehicles
  Vehicles requests allow to insert or extract a vehicle (POST) and monitor the status of the vehicles (GET).

- Missions: URL: . . . /wms/rest/missions
  Mission requests allow to create new missions (POST), monitor current missions (GET) and cancel missions (DELETE).

- Alarms: URL: .../wms/rest/alarms
  Alarms requests allow to get the list and status of alarms in the installation (GET).

- Devices: URL: .../wms/rest/devices
  Devices request allow to get the status of devices in the installation (GET) and to send commands to these devices (POST) [21].

The main parts of the workflow with AntServer API are [21]:

- Managing the connection
- Sending commands
- Monitoring the status of each component of the installation by exchange data
- Create missions and get their status using their ID.

A schematic representation is reported in Figure 2.8

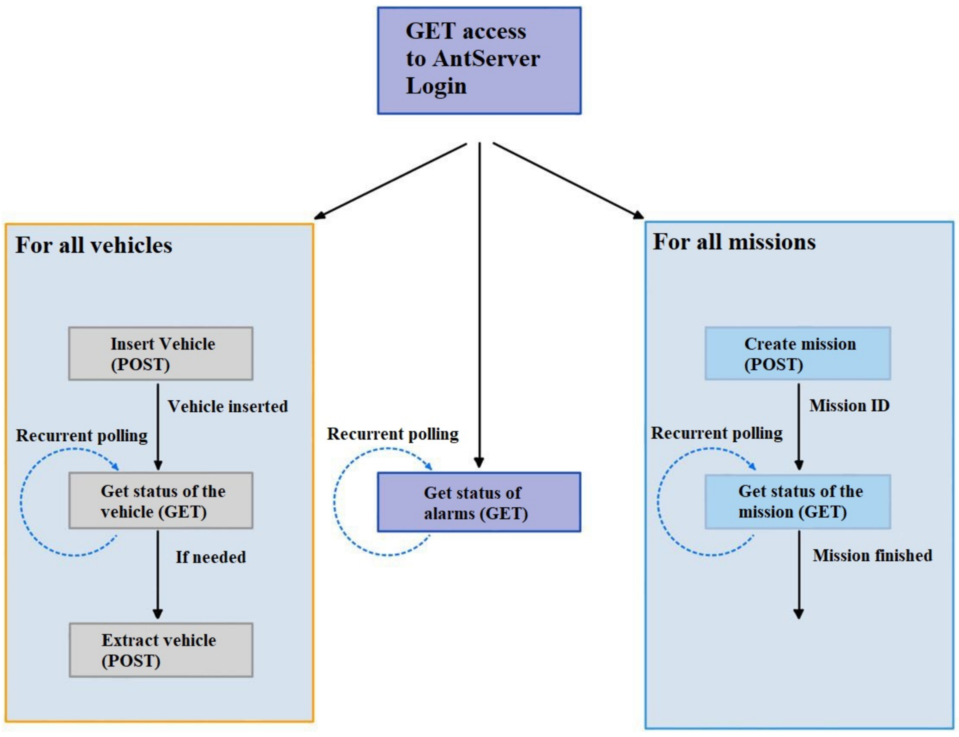


Figure 2.8: AntServer workflow

2.5 New AGV

The new AGV hardware design is reported in Figure 2.9b, and can be compared with the actual Skilled LGV shown in Figure 2.9a.

The differences between the current Skilled LGV and the new designed one are many. Firstly, the Safety PLC has been removed and encoded in the Siemens PLC, which will contain both safety and non-safety modules.
Secondly, a Profinet switch has been used to make everything communicate. In Profinet Switch, each switch port is identified by a port address and therefore everything can communicate correctly and directionally.

Profinet switches have them own General Station Description (GSD) file. The Controller recognizes them as IO devices, and the GSD file defines
Profinet-related diagnostics information. Most of the Ethernet switches are suitable for Profinet [28].

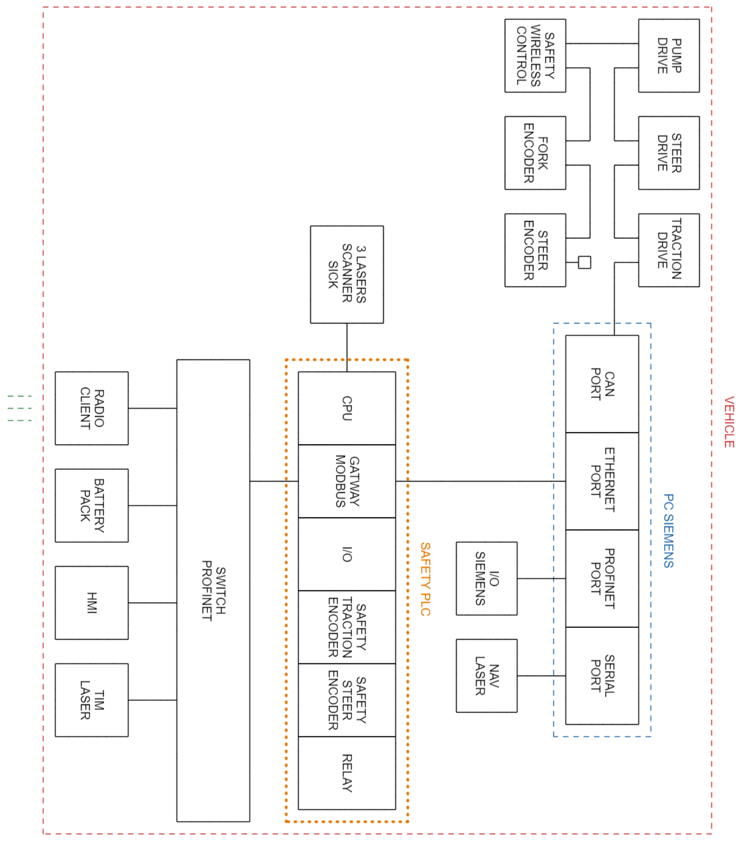
This switch is needed because many devices has to communicate both with the additional Siemens PLC and with AntLite$^+$, for instance Laser Scanners. The model of this Lasers is the one that is supported by BlueBotics technology and has a Profinet and Profisafe Communication so can be used for Safety purposes.

To enable BlueBotics AntLite$^+$, whose communication protocol is Ethenet/IP, to exchange information with Sick Laser Scanners (Profinet and Profisafe communication protocols), the PN/MF bus coupler has been added. This allows easy and fast data exchange between a Profinet and an Ethernet/IP controller.

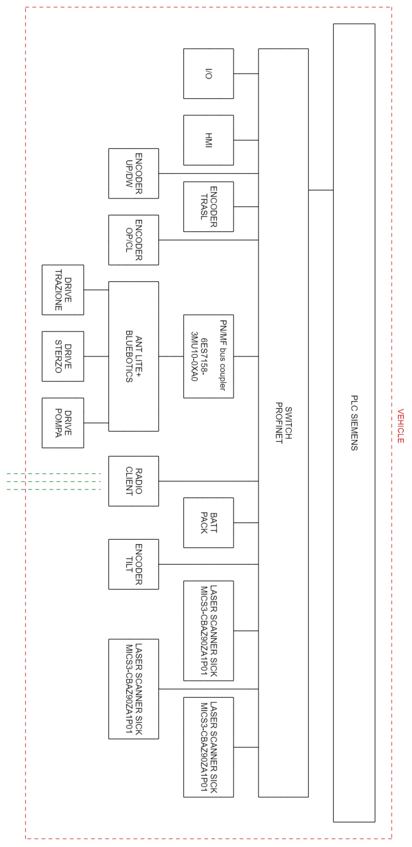
All other components in the block scheme are connected to the switch so that the PLC can manage them and are: I/O modules, battery pack (can be also managed by ANT technology to send vehicles into charging position when needed), HMI and encoders.

Encoders connected to the switch controls different fork movements:

- Encoder up/down: positioned to control the upward and downward movement of the forks;
- Encoder trasl: positioned to control the translation of the forks along the horizontal axis, so the right/left movement;
- Encoder op/cl: positioned to control the relative opening and closing of the forks in order to pick up and position pallets of different sizes;
- Encoder tilt: positioned to control the angle of the forks to allow picking and positioning if the pallet is not perfectly positioned horizontally.



(a) Actual LGV design



(b) New AGV design

2.5.1 AntLite$^+$

AntLite$^+$ is designed to be installed in a wide range of vehicles, for this reason it can support various architectures. In the configuration defined in this project AntLite$^+$ is connected to both Motor Controllers and the AGV devices, in this way the Siemens PLC can be seen as an additional PLC that provides the Safety system and the other tools not managed by AntLite$^+$ [17].

The connected motor controllers (Traction and Steering drives in Figure 2.9b) have two functions: to receive motor commands from AntLite$^+$ and drive the motors accordingly, and send odometry feedback information to AntLite$^+$ as an input to the localization algorithm. Direct communication between AntLite$^+$ and the motor controllers is based on the CAN bus or on EtherNet/IP.

The connected Laser scanners are essential for the localization of the AGV, as they provide an image of the environment, and for navigation, whose outer data are used by the ANT navigation algorithm to slow down around obstacles and to stop before triggering safety mechanisms. This allows smooth vehicle operation in a dynamic environment.

The laser scanners can be used to track an object and adjust the vehicle trajectory to adapt to the object's position. A Laser Scanner can be used to perform various tasks at the same time. So, depending on the needs for the application, several Laser Scanners can be used and combined.
The new Skilled AGV will be equipped with three Safety Laser Scanners near floor height, used for localization, navigation and Safety purposes [17]. To provide high-precision also the odometry is used: this algorithm uses the information from the wheel encoders to calculate the vehicle's movement. Depending on the kinematics, AntLite$^+$ uses wheel speed and/or steering angle [17].

AntLite$^+$ communicates with the added PLC to provide additional functionalities. There are two different ways to connect the PLC to AntLite$^+$: CANopen and EtherNet/IP. Therefore an EtherNet/IP bridge is required to use a PLC with Profinet communication.

PLC functionality is associated with nodes in the AntLab project, as *PLC Action*, and then is triggered by AntLite$^+$. An action is a mechanism used to trigger an operation, with the possibility to determine when the action terminates, whether it terminated successfully or failed, and to abort an action in progress. The commander provides one or more arguments and an execute bit to perform the Action. The executor answers with a success or failure bit and, optionally, a results byte [21].

2.5.2 Laser Scanners

The chosen model for Laser Scanners is the one that is supported by BlueBotics technology and has a Profinet and Profisafe Communication so can be used for Safety purposes. The only type of Scanners that have all this characteristics is the SICK product MicroScan3 Laser Scanner that support both Profinet/Profisafe and EtherNet/IP communication protocols.

Laser scanners have multiple purposes in an AGV:

- Localization: The laser scanner provides an image of the environment to AntLite$^+$ for localization of the AGV in the map.

- Navigation: The laser is used by ANT navigation algorithm for slowing down around obstacles and for stopping before triggering safety mechanisms. This is what allows smooth operation of the vehicle in a dynamic environment.

- Safety: If the laser is safety-rated, well configured and well positioned, it can provide safety functions of the AGV.

- Tracking: The laser scanners can be used to track an object and adjust the vehicle trajectory to adapt to the object's position. [17].

A Laser Scanner can be used to perform various tasks at the same time. So, depending on the needs for the application, several laser scanners can be used and combined.

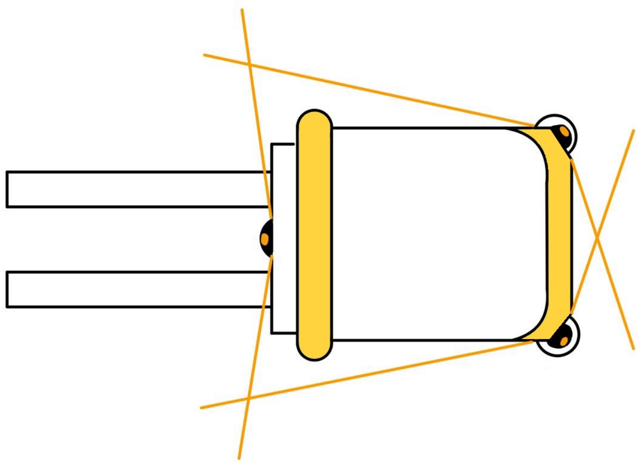


Figure 2.10: Lasers configuration

Figure 2.10 shows a sketch of the Laser Scanners positioning in the new Skilled AGV and the area covered around the vehicle.

To guarantee optimal performance, three main aspects has to be taken into account: maximising the Laser Scanner's field of view, controlling the consistency of the Scanner's height and ensuring that the Laser Scanner is horizontal.
Laser Scanners have a wide field of view, to optimize them usage an optimal positioning in the vehicle has been designed requiring the same brand and model for each Laser Scanners as AntLte$^+$ will merge scanner's data. Laser scanners are must have a relative height difference less than $\pm 2cm$. Laser scanner horizontality is a critical aspect for AntLite$^+$ as several problems can arise if they are not well set. For instance the scanners will see the ground, or above a needed feature for localization, or see a feature different from the one expected leading to a false match and an inaccurate laser distance [17].
The vehicle's mechanical stability is important to ensure that the Laser Scanners do not vibrate or shift over time.

The chosen Sick MicroScan3 has the follow characteristics [17]:

manufacturer	model	Measuring range	Protective range	FoV
SICK	microScan3	64m	9m	275°

The measuring range is the maximum distance from which the Safety Laser Scanner can detect an object. The protective range is the measurement of the Protective Field cover distance, i.e. the area in which, if an object is detected, the vehicle is forced to stop. A warning field can be defined around this area to trigger a warning signal [18]. The composition of those fields will define the vehicle hull shapes as reported in Chapter 3, Section 3.1.2.

Laser Scanners data are used from ANT technology, as the chosen Scanners are Safety Scanners, to set the correct Safety Area around the vehicle according to its position in the environment and the situation in witch the AGV is operating. For instance the docking hull corresponds to a small area around the vehicle, in this configuration the vehicle will not be able to travel at high speed but it will be used in picking/placing actions.

Standstill hull is used when the vehicle is not moving, this means that if something is present in the hull, the vehicle will not start.

The hull shape selected when the AGV is moving is defined according to the vehicle speed and direction. If the vehicle is moving forward or backward at low speed the hull will be the same as the standstill, but will be greater in the direction of movement.

If it moves faster, the shape in the moving direction will be greater, so that it can start to stop the vehicle when it detect objects in the hull boundary [17].

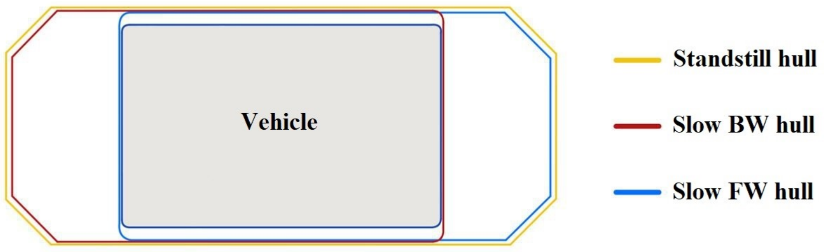


Figure 2.11: Vehicle Hulls

During normal operation, if an object is detected inside the hull, the laser safety zone will trigger an emergency stop signal to stop the vehicle as fast as possible to avoid collisions.

2.5.3 PN/MF Coupler

The PN/MF Coupler is used to connect an EtherNet/IP network to a PROFINET sub-network or to interconnect two PROFINET sub-networks.
It also offers deterministic data exchange between PROFINET and EtherNet/IP controllers.

The PN/MF Coupler has two Ethernet interfaces:

- X1 for EtherNet/IP or Profinet IO
- X2 for Profinet IO

During its configuration, Siemens PLC creates two IO devices with their respective subnets from the PN/MF coupler. [29].

The new AGV Skilled contains a Siemens PLC and the PN/MF coupler (ProfiNet MultiField bus coupler) that is required to communicate with AntLite$^+$. The configuration of the PN/MF coupler in Siemens PLC starts with the installation of its GSDML file, a readable ASCII text file that contains both general and device-specific specifications for communication (Communication Feature List), and network configuration.

This is followed by the adding of the I/O modules, as required in the PN/MF coupler documentation, and the definition of the PLC input structure that contains all input bits and relative addresses. For instance [17]:

Input for PLC	bit	address
PosX	32	%ID0
PosY	32	%ID4
PosTheta	16	%IW8

Table 2.1: Examples of PLC inputs

The device has to be configured also in AntLab by set the Siemens PN/MF as an Ethernet/IP PLC.

2.5.4 PLC Siemens

The Siemens PLC has a multiple role in this new AGV project. Its configuration includes safety and non-safety modules, so that it can provide both functionalities.

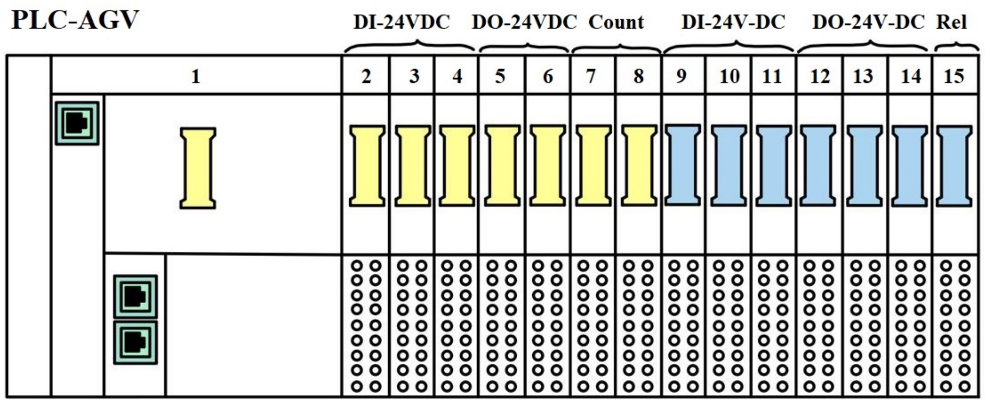


Figure 2.12: PLC modules configuration

The Figure 2.12 represents the configuration defined in TIAPortal to develop the address assignment of the input/output signals that will be called up in the programming environment. The modules in yellow identify the safety modules that use the PROFIsafe communication protocol.

Profinet is an open industrial Ethernet standard that offers real-time data exchange while maintaining openness for flexible plant and machine concepts. In addition to the standard features of each Profinet device, optional features such as PROFIsafe provide additional functionality.
PROFIsafe extends the standard Profinet communication protocol to meet the unique requirements of functional safety information, necessary to comply with stringent safety standards. PROFIsafe takes care of the functional safety part of communications. It ensures the integrity of failsafe signals transmitted between safety devices and a safety controller meeting the relevant safety standards for industrial networks [30].

The first three modules in the Siemens PLC configuration (Figure 2.12) are Safety DI-24VDC, i.e. Safety Digital Input modules with 24V DC voltage and Direct Current.
The next two, enumerate with 5 and 6 are Safety DO-24DC, which stands for Safety Digital Output - 24V Direct Current.
The last two safety modules, enumerated with 7 and 8, are the Safety Count

modules needed to count various things and evaluate the state evolution of certain variables and are used for encoders information. The counter limits can be set to provide information on the evolution of variables, such as a warning or error message [31].

From the 9th to the 15th modules are light blue, which means they are non-safety. These are the Digital Input, Digital Output and Relay modules. The relay outputs consist of a contact that opens or closes depending on the processing of the entered program.

2.5.4.1 Safety

Safety modules are necessary to define the Safe behaviour of Skilled AGVs, the priority is to ensure the safety of workers and environment when AGVs are operating.

The Safe behaviour of an Automated Guided Vehicle is defined in:

- UNI EN ISO 3691-4:2020 *'Industrial trucks - Safety requirements and verification - Part 4: Driverless industrial trucks and their systems'* containing requirements for the design and production of trucks and for the preparation of operating and load transfer zones
- UNI EN 1175:2020 *'Safety of industrial trucks - Electrical/electronic requirements'* concerning the electrical systems of industrial trucks, including self-driving trucks [32].

The UNI EN ISO 3691-4:2020 standard provides guidance on the safety measures that must be present on trucks to enable their safe operation, including devices for detecting people in the path.
A very important part of the standard is the definition of the requirements for the preparation of operating areas, including signs and markings on the ground, perimeter guards, sensitive devices to protect openings for forklift access, and minimum spaces for the passage of workers.

To guarantee a Safe AGV behaviour, a Safety functionality has to be composed of three subfunctions:

- Detect: for instance with position sensors and light curtains
- Evaluate: for instance with fail-safe control and modular safety systems
- React: for instance with drives and motor management systems

Together, these subfunctions must create a safe and effective chain.

PLC Safety functions are integrated in SINAMICS, the Digital base AC and DC drive system used to control motion, speed, and torque. Safety functions in SINAMICS drives can be divided in four categories [33]:

- Functions to safely stop the drive
- Functions for safe brake management
- Functions for safely monitoring drive motion
- Functions for safely monitoring the position of a drive

Safety functionality is implemented principally through safety functions in the software. Safety functions are executed by the SIMATIC Safety system in order to bring the system to a safe state or maintain it in a safe state in case of a dangerous event.

Safety functions are contained mainly: in the safety-related user program (safety program), in the CPU and in the fail-safe inputs and outputs (I/O).
The I/O safety modules ensure the safe processing of field information from sensors (e.g. emergency STOP pushbutton, light barriers) and actuators (e.g. for motor control). They have all of the required hardware and software components for safe processing.

The programming of fail-safe FBs and FCs in the safety program can be done by creating fail-safe DBs.
Safety checks are automatically performed and additional fail-safe blocks, for error detection and fault reaction, are inserted when the safety program is compiled. This ensures that failures and errors are detected and appropriate reactions are triggered to maintain the system in the safe state or bring it to a safe state.
In addition to the safety program, a standard user program can be run on the CPU. It can coexist with a safety program in an CPU because the unintentional influencing of the safety-related data of the safety program is uncovered by the standard user program.
Data can be exchanged between the safety program and the standard user program in the CPU by means of bit memory or data of a standard DB or by accessing the process image input and output [34].

2.5.4.2 Non-safety

The non-safety actions of the AGV includes all the function that are not defined as Safe and do not require ProfiSafe communication. All the AntLab *PLC actions* are coded in this environment and define all the action that are external to the AntLab programmed actions.
The Safety variables can be used in the Non-Safety programming environment but cannot be changed or assigned a value. For instance the blinking light, that provide visual information on the correct, warning or error status of the AGV, is coded in the Non-Safety environment using the Safety AGV state variables and the TON-TOFF function blocks. The use of a PLC in an AGV driven by AntLite$^+$ can have multiple objectives:

- Additional functionality: a PLC integrated in the vehicle allows additional vehicle functionality to be executed from commands sent by AntLite$^+$
- Digital IOs: AntLite$^+$ can interact directly with a selected number of external PLC I/Os, so as to have access to devices such as proximity sensors, lifts, lamps and many others.
- Connection to motor controllers: if motor controllers are not directly compatible with AntLite$^+$, they can be connected to the PLC. The PLC can then ensure data exchange between AntLite$^+$ and the motor controllers [17]. In Skilled AGVs these are directly connected to both AntLite$^+$ and the Siemens PLC via the Ethernet switch.

PLC actions are triggered from AntLite$^+$, to use functionalities implemented in the PLC. The PLC Actions are specified by their own number and arguments. Their function is determined only by the implementation in the PLC. For example, action 0 can be used to move the fork at a certain height and action 1 can be used to move a conveyor at a certain speed [17].

Chapter 3

Project Develop

3.1 AntLab programming

3.1.1 Map definition

The environment features representation and the definition of allowed AGVs path takes place in the AntLab design environment. This can be done in two ways:

1. By moving the vehicle around the environment in manual mode, taking notes on the positions of the reflectors and features position and then defining the paths

2. By the import of a CAD file representing the environment's plant and then define the allowed AGV paths considering the plant scaling factor.

AntLite$^+$ provides Natural Feature Localization, this means that, in most cases, the features present in the environment are sufficient to build an accurate map and navigate within it. In the first case, during the mapping with vehicle manual navigation in the environment, AntLite$^+$ will extract segments and reflectors.
Segments are the representation of a straight object seen by lasers (like walls and boxes, those should be long-term static objects as they will provide tools for AGV localization).
Reflectors, instead, represent high reflectivity locations added manually in case there are not enough references for localization [17].

In the second case, a CAD file of the warehouse layout is imported into AntLab designer, this is not used for localization but only for representation.
CAD file must be an SVG file or a PNG file, then lines, polylines, arcs and

circles that were set as visible in AutoCAD are imported. The scale ratio must be adapted to the extracted map so that the overwritten path are coherent with warehouse dimensions.

Once the map of the environment has been defined or imported, routes and actions must be designed.
Routes are composed of a set of nodes connected by links. Nodes represent the vehicle passing points and are represented by a number (identification), an arrow (orientation of the vehicle when snapped on the node) and a radius (how sharp the trajectory curvature is).
The node radius parameter, that describes the curve radius of the line generated on the node, can be set as Manual or Automatic.

An AGV Action can be assigned to nodes and can start on transit, on site or on departure.
For example, if the AGV heads to a node whose assigned action is to pick up a pallet at a height of 2 meters, it is convenient to start lifting the forks on transit. A sequence of actions can then be assigned to the node:

- on transit action: impose velocity reduction
- on transit action: when the AGV is, for instance, $3m$ from the node, start to raise the forks
- on site action: check on the correct forks height, if it is not stop and wait for it, in case send and error signal
- on site action: reach the node position to pick up the pallet, wait for a signal from the presence sensor
- on site action: once the pallet has been pick, go back and remove it from the shelf
- on transit action: command the forks to reach the “travelling forks height”
- on transit action: remove the velocity limits. [21].

The AGV path following is performed by considering the vehicle center as reference point, both in curvature and straight line trajectory sections.

In following a path, AntLite$^+$ uses two controllers, these are tuned to ensure that the vehicle follows its predefined path as well as possible and in case of errors will make adjustments in the motors commands to return to the path.

Distance controller is used to minimize the distance between the vehicle position and the path, its value will mostly affect how the vehicle performs on straight lines.
Heading controller is used to minimize the difference between the vehicle heading and the vehicle path direction (tangential to the closest path point in a curve) and will affect how the vehicle performs in curves and while exiting a curve.

Non-omnidirectional vehicle is assumed to be able to drive forward and backward along a link following the direction in which it is oriented. If it is too far from the required orientation it will send and error massage of the type “Too far from the path”.

The map files created in AntLab is sent to each vehicle that operate in the environment to have consistent behaviour [21].

3.1.2 Vehicle definition and calibration

A bootfile exists for each type of vehicle and it contains both common and individual calibration parameters. When the vehicle is connected to the AntLab project, an automatic check for parameter consistency is performed [21].

One of the main aspects of vehicle calibration is Laser Scanners calibration. The chosen laser model has to be insert and then Lasers position, roll and pan angle are automatically calculated during the calibration process. Each laser scanner has it own IP address so that data can be classified according to its value.

In case the laser scanner detect unwanted points of the vehicle, presence of red points inside the vehicle shape in AntLab, redefine the laser scanner properties angle interval to discard those values.

The laser scanner data are necessary for AGV navigation in the environment and a set of safety hull shapes is defined to use them rapidly. Each of these hulls corresponds to a combination of laser scanners safety fields and is used for a particular set of situation, as define in Chapter 2, Section 2.5.2.

The main purpose of the hull is to avoid AGV collisions. Hulls shape definition can be done in an Excel file and then imported in AntLab [17].

AntLite can work in Obstacle Avoidance mode, this will allow the vehicle to go outside of the predefined path to avoid an obstacle while staying as close as possible to the path [17].

3.2 Programming Ignition

The communication with the AntServer is implemented with REST API, composed by URL and JSON, as described in Chapter 2, Section 2.4.3.1.
Ignition's JSON handling requires an external Python library, which must be imported in order to use the defined functions.

Once this is done, it is possible to import the two methods **request** and **json** in the programming environment. Those contains the functions needed to communicate with AntServer and from this with AntLab and AntLite$^+$.

3.2.1 JSON

JSON (JavaScript Object Notation) is a standard text-based format for representing structured data based on JavaScript object syntax. It is completely language independent and is commonly used for transmitting data.

JSON exists as a string and this format is useful for transmitting data over a network, but it must be converted into a native JavaScript object to access the data. JavaScript provides a global JSON object that has methods available for converting between the two formats.
JSON string, similar to JavaScript object literal format, can be stored in a file that's similar to text file with .json extension. So JSON include the same basic data types as JavaScript object: strings, numbers, arrays, booleans, and other object literals. This allows a hierarchy structuring of data.

An object is an unordered set of name/value pairs, it begins with "{" left brace and ends with "}" right brace. Each name is followed by ":" colon and the name/value pairs are separated by "," comma.

To access the data inside, the dot/bracket notation is necessary and to access data further down the hierarchy, a chain of the required property names and array indexes together is necessary.
A JSON file example is reported in Figure 3.2.2

3.2.2 Ant Server communication

As reported in Chapter 2, Section 2.4.3.1 BlueBotics AntServer communication is done with JSON. Once the Ignition setup has been update with the new required libraries for JSON understanding and managing, the information contained in the required JSON response can be understand.

AntServer's JSON messages are as numerous as AntServer has functionalities. To ensure fast message conversion and a readable programming structure, an object-oriented programming structure was chosen to contain and classify all necessary methods.
In Ignition coding environment the main developed objects are: Server, Mission, Vehicle, Map and BlueBotics object that calls all the others and provide AntServer communication.

The BlueBotics object contains the main methods to allow the proper manage of the new Skilled AGV technology.

The applied HTTP methods to relate with AntServer are:

- POST: Required the URL to send the request to AntServer with the JSON file with the required parameters.
- GET: Required URL for send the request and then receive the AntServer response. As it is only a requirement of information without modification, the JSON file with parameters is not necessary.
- DELETE: Requires just the API URL to communicate with AntServer

In each method case the base idea is the same and it is structured as:

1. Import the request libraries for handle JSON file format.
2. Try/except block to verify if the Login has been successful. This is done by the verification of the sessiontoken length, returned from AntServer Login.
3. Composition of the **api_url** string with the acquired sessiontoken.
4. Composition of the string that will compose the JSON structure with the required parameters (if necessary) and its conversion from string to **.json** format.
5. The sending of the request to AntServer with the proper HTTP method. The response, if the communication holds correctly, is saved in a proper variable to then extract the required information.

The BlueBotics object has all the methods required to allows the message exchanging with AntServer.

Follows the definition of the BlueBotics object and a method example for each of the applied HTTP method.

```
    import requests
    import json
```

In the first part of the program, the import of the two methods necessary for reading JSON and send requests take place.

```
class BlueBotics:
  def __init__(self,ip = "localhost",port = "8081",user="admin",
   psw="123456"):
    self.ip = ip
    self.port = port
    self.user = user
    self.psw = psw
```

Then follows the definition of the BlueBotics object, whose constructor section contains the variables: IP, port (to complete the IP addressing), username and password. Used to enable the communication with AntServer and then with the AGV.

The first method of the BlueBotics object is the **Login** and it will return a session token that is required by all other methods to communicate with AntServer.

```
  def Login(self):
    #INPUT
      #[STRING] username: Name of a user as defined in the system
      #[STRING] pwd: Password
    #OUTPUT
      #[INT] err: Error status
      #[STRING] sessiontoken: token received from ANTServer
      #[STRING] username: Name of a user as defined in the system
```

In the first part of the method the complete description of the involved and necessary parameters is done to make the code section easily readable and understandable.

```
    api_url = "http://" + self.ip + ":" + self.port + "/wms/
   monitor/session/login?username=" + self.user + "&pwd=" + self.
   psw
```

The Login method requires username and password to access AntServer, and those are defined in the BlueBotics object constructor as **self.user**, **self.psw**.
In the same section are reported the other variables required to build the **api_url** to communicate with AntServer, like **self.ip** and **self.port**. All those variables can can have a default value, reported in AntServer manual, to access the device.

```
    response = requests.get(api_url)
    jsonfile = response.json()
```

Once the **api_url**, an address that allows to access an API and its features, is defined the GET request is sent to AntServer with the method **requests.get(api_url)**. AntServer response will be contained in a Response object called **response**.

The method **response.json()** returns a JSON object of the result, if the result was written in JSON format, otherwise it opens a PopUp error window.

```
    self.retcode = (jsonfile["retcode"])
    if self.retcode == 0:
      self.sessiontoken = (jsonfile["payload"]["sessiontoken"]).
  encode()
      self.username = (jsonfile["payload"]["username"]).encode()
    else:
      system.perspective.openPopup('1', 'Error', params = {'
  textLabel':'retcode = ' + self.retcode + '!'},showCloseIcon =
  False)

```

Once the **jsonfile** variable has been created the access to all the parameters contained in the JSON format are available using the square brackets as in dictionaries programming structures.
In this way is easy to verify the retcode value: if it is zero, this means that communication with AntServer has been successful, otherwise an error has occurred and it can be identified from the retcode value.

If the communication holds correctly, then the session token can be acquired as it is contained in the payload in **jsonfile**.
The **encode()** method returns an encoded version of the given string. This is necessary cause the parameters in the **jsonfile** are encoded in the Unicode format that add format characters to the string. In the same way the username is saved in the **self.username** constructor variable. If the retcode is different from zero a PopUp window is displayed with an error message.

An example of REST DELETE method is the **CancelOneMission** method contained in the BlueBlotics object. It allows to delete a specific mission in the mission scheduler.

```
    def CancelOneMission(self,missionID):

  #INPUT
    #[INT] missionID: Id of the mission

  #OUTPUT
    #[STRING] missionid: Mission id.
    #[BOOL] cancelled: true, The mission is cancelled.
              #false, The mission is not cancelled.
```

The **CancelOneMission** method requires as input variable the **missionID** to identify the mission to be deleted.
As output it returns the ID of the deleted mission and the Boolean variable to check if the cancellation happen correctly or not.

```
  try:
    if len(self.sessiontoken)<=0:
      self.Login()
  except:
    self.Login()
```

In the first part of the program the try/except block verifies if the **Login** happen by checking the lenght of the **sessiontoken** variable. If it is zero then the Login is performed.

```
  api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/
missions/" + str(missionID) + "?&sessiontoken=" + self.
sessiontoken
  response = requests.delete(api_url)
  jsonfile = response.json()
```

Then the definition of the **api_url** is done using the ip, port and session token defined in the BlueBotics object constructor and the missionID obtained as input to the method. The request to AntServer is performed with **requests.delete(api_url)** as it is an HTTP DELETE method and it do not required a JSON file with additional parameters. The only required parameter is the missionID that is insert in the **api_url**.

As in the previous method the **responce.json** returns the JSON object from the JSON file so that the response parameters information are accessible.

```
 self.retcode = (jsonfile["retcode"])
 if self.retcode == 0:
   missID = jsonfile["payload"]["missionid"].encode()
   cancelled = jsonfile["payload"]["cancelled"]
 else:
     system.perspective.openPopup('1', 'Error', params = {'textLabel':'retcode = ' + self.retcode + '!'},showCloseIcon = False)

 return missID,cancelled
```

In the last part of the method the retcode is verified to be zero, meaning that the communication and the function holds. If this is true **missID** and **cancelled** response values are extracted from the **jsonfile** to be returned as output. Those represent the deleted mission ID and a boolean variable to check if the cancellation happened correctly.

The **CreateMission** method is an example of HTTP POST method.
This initiates the creation of a CreateMission message from the warehouse management software, which is then sent to AntServer to communicate with the vehicle and successfully complete the task.
The scheduler will select the best vehicle for the required action.

```
 def CreateMission(self,missType,fromNode,toNode,priority=1,deadline="",dispatchtime="",vehicle="",cardinality=1,payload="Something"):
 #INPUT
   #[INT] missiontype (mand.):
     #0: Transport from node to station.
     #1: Move to station.
     #2: Waiting lane.
     #7: Transport to node.
     #8: Move to node.
     #9: Transport from station to station.
     #10: Move a specific vehicle to a node.
     #12: Move to a loop
   #[STRING] fromnode: Name of the pickup node: <Node> or station (depending on the type of mission)
   #[STRING] tonode: Name of the delivery node or station
```

```
    (depending on the type of mission)
      #[INT] cardinality: number of missions to create
            #[INT / STRING] priority (Another number or text is
   not interpreted. Mission priority is set to Medium default
   value):
        #0 or Low,
        #1 or Medium,
        #2 or High
      #[TIME] deadline (opt.): The absolute date from which the
   mission can be assigned to a vehicle
      #[TIME] dispatchtime (opt.): The absolute date when the
   mission is started
      #[STRING] vehicle: Name of the vehicle that you want to
   assign
      #[STRING] payload: Name of the payload
    #OUTPUT
      #[STRING ARRAY] rejectedmissions: Array of strings
   containing ids of rejected missions
      #[STRING ARRAY] pendingmissions: Array of strings
   containing ids of pending missions
      #[STRING ARRAY] acceptedmissions: Array of strings
   containing ids of accepted mission.
```

The **CreateMission** method requires: the mission type, the departure and arrival node and other variables whose value can be assigned explicitly or by default [21].
The **missionType** description table is reported in Table 3.1. The value N/A means that the starting point of the vehicle is its real-time position when it receives the mission.

ID	Description	fromnode	tonode
0	From a source node to a target station	Node name	Station name
1	Move to a station	N/A	Station name
2	Waiting lane	N/A	Station name
7	From a source node to a target node	Node name	Node name
8	Move to a node	N/A	Node name
9	From source station to target station	Station name	Station name
10	Move a specific vehicle to a node	N/A	Node name
12	Move to a loop	N/A	Loop name

Table 3.1: MissionTypes

The Response from AntServer returns three different arrays containing the IDs of the missions created, which have the respective characteristic:

- Rejected: missions that cannot be executed by AntServer. For example no route found between the source node and the destination
- Pending: missions that have been received by AntServer but that have not been accepted or rejected yet
- Accepted: missions that AntServer is going to execute

As far as the missions is concerned, not all fields are mandatory depending on mission type and mission scheduler.

Entering in code details:

```
  try:
    if len(self.sessiontoken)<=0:
      self.Login()
  except:
    self.Login()
```

The code starts with a **try/except** block to verify if the Login happened, otherwise it is performed in the **except** part of the code.

```
  api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/
missions?&sessiontoken=" + self.sessiontoken
  myobj = '{"missionrequest":'
  myobj = myobj + '{"requestor": "' + self.username + '",'
  myobj = myobj + '"missiontype": "' + str(missType) + '",'
  myobj = myobj + '"fromnode": "' + str(fromNode) + '",'
  myobj = myobj + '"tonode": "' + str(toNode) + '",'
  myobj = myobj + '"cardinality": "' + str(cardinality) + '",'
  myobj = myobj + '"priority": "' + str(priority) + '",'
  if deadline != "":
    myobj = myobj + '"deadline": "' + str(deadline) + '",'
  if dispatchtime != "":
    myobj = myobj + '"dispatchtime": "' + str(dispatchtime) + '
",'
  myobj = myobj + '"parameters": {'
  myobj = myobj + '"value": {'
  if vehicle != "":
    myobj = myobj + '"vehicle": "' + str(vehicle) + '",'
  myobj = myobj + '"payload": "' + str(payload) + '"},'
  myobj = myobj + '"desc": "Mission extension",'
  myobj = myobj + '"type": "org.json.JSONObject",'
  myobj = myobj + '"name": "parameters"}'
  myobj = myobj + '}}'

  myobj = json.loads(myobj)
```

Follows the definition of the required **api_url** to communicate with AntServer, and the definition of the JSON parameters file required for the mission creation. Firstly the string variable **myobj** is composed with the parameters structuring required to develop a correct AntServer communication.
The **json.loads()** method accepts as input a valid string and converts it to a JSON file.

```
    response = requests.post(api_url, json = myobj)
    jsonfile = response.json()
```

The request is sent to the AntServer with the
requests.post(api_url, json = myobj) method that requires the composed api_url and the JSON file with the required parameters.
An example of AntServer CreateMission Request and Response is reported.

Request example: Create a mission rule
Method: POST;
URL: `http://localhost:8081/wms/rest/..`
`../missions?&sessiontoken=uAzDPqMRPfEhg05X3ajXsw%3D%3D`

```
{
    "missionrequest": {
        "requestor": "Requestor-Username",
        "missiontype": "0",
        "fromnode": "S1-Pick",
        "tonode": "Station 1",
        "cardinality": "1",
        "priority": 2,
        "deadline": "2015-02-25T12:27:41.043Z",
        "dispatchtime": "2015-02-25T12:27:41.043Z",
        "parameters": {
            "value": {
                "payload": "Something"
                "value": {
                    "payload": "Something",
                    "maxwaitingvehicles": "3",
                    "maxduration": "3"
                },
                "desc": "Mission extension",
                "type": "org.json.JSONObject",
                "name": "parameters"
            }
        }
    }
}
```

This is an example of the JSON file sent to AntServer to create a new mission. The server response structure is, according to BlueBotics documentation:

Response example:

```
{
    "payload": {
        "rejectedmissions": [],
        "pendingmissions": [
            "1"
            ],
        "acceptedmissions": []
        },
    "retcode": 0
}
```

Then the response, that can be read with the **response.json()** method, is analysed to verify the correctness of the mission creation:

```
      self.retcode = (jsonfile["retcode"])
    if self.retcode == 0:
      answerMissJSON = (jsonfile["payload"])
      missID = MissionID(answerMissJSON)
    else:
        system.perspective.openPopup('1', 'Error', params = {'
   textLabel':'retcode = ' + self.retcode + '!'},showCloseIcon =
   False)

    return missID
```

The variable **self.retcode** is verified to be zero to check the correctness of the communication with AntServer. Therefore, an **if/else** block structure is added and, in case of an error, the error message is displayed with a PopUp window in the HMI.

The variable **missID** is a MissionID object with the chronology of the missions [21].

```python
class MissionID:
  def __init__(self,answerMissJSON):
      rejMiss = answerMissJSON["rejectedmissions"]
    penMiss = answerMissJSON["pendingmissions"]
    acpMiss = answerMissJSON["acceptedmissions"]

    self.rejectedmissions = []
    self.pendingmissions  = []
    self.acceptedmissions  = []

    for rejected in rejMiss:
      self.rejectedmissions.append(rejected.encode())
    for pending in penMiss:
      self.pendingmissions.append(pending.encode())
    for accepted in acpMiss:
      self.acceptedmissions.append(accepted.encode())
```

The **MissionID(answerMissJSON)** class is the one that receive as input the **payload** section of the AntServer responce and the MissionID object that has in it all the mission rejected, pending and accepted in order.
In case missions are not created or are rejected, a reason will be given in the reply with the retcode value.

This structure allows to easily identify if the last sended mission has been accepted, rejected or insert with the pending ones.

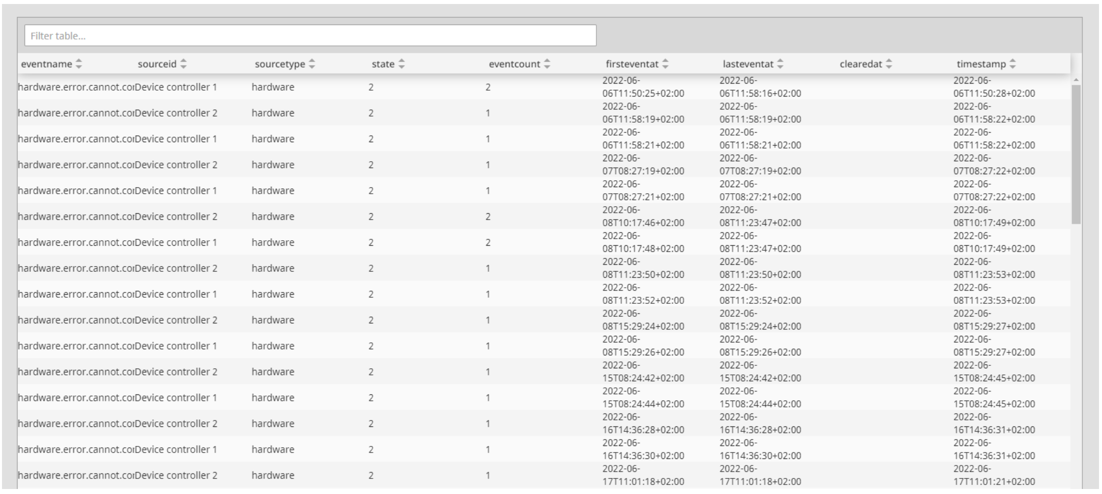
3.2.3 Perspective HMI programming

HMI program definition was done in Perspective, a subsection of Ignition programming that allows the HMI to be visualised in a web page and then in screens of different sizes like PCs, tablets and smartphones.
This programming section defines the exchange of information and tasks between the operator and the overall AGVs system.

The web page sections report the following windows:

- Alarm Section: Returns a table with the active Alarms and a table with them historical overview. The Figure 3.1 report an example of error message due to the non-connection of the Device. The **GetAllAlarms** method of the BlueBotics object allows to get all alarms information, the code can be found in Appendix A.

  - eventname: Name of the alarm
  - sourceid: Name of the event source
  - sourcetype: Type of the event source
  - state: State of the alarm Active, Acknowledged, Closed or Deleted
  - eventcount: Number of times the alarm occurred
  - firsteventat and lasteventat : Date of the first and last occurrence
  - cleardat: Date the alarm was cleared by the operator
  - timestamp: Date of the last update

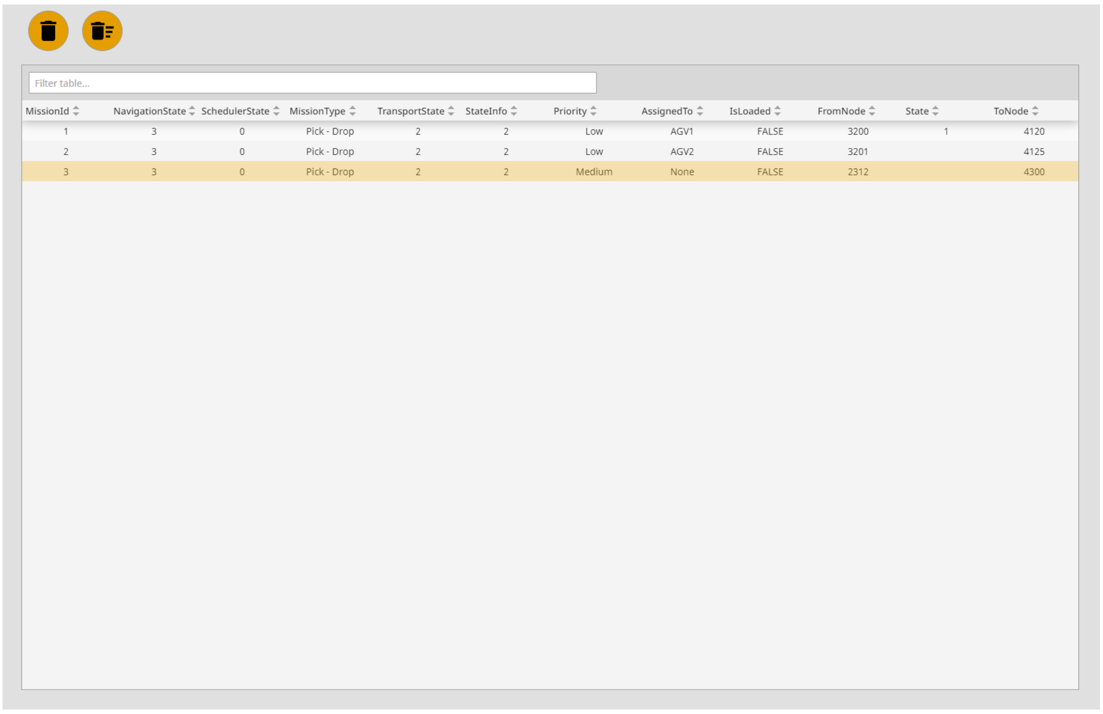


Filter table...

eventname	sourceid	sourcetype	state	eventcount	firsteventat	lasteventat	clearedat	timestamp
hardware.error.cannot.co	Device controller 1	hardware	2	2	2022-06-06T11:50:25+02:00	2022-06-06T11:58:16+02:00		2022-06-06T11:50:28+02:00
hardware.error.cannot.co	Device controller 2	hardware	2	1	2022-06-06T11:58:19+02:00	2022-06-06T11:58:19+02:00		2022-06-06T11:58:22+02:00
hardware.error.cannot.co	Device controller 1	hardware	2	1	2022-06-06T11:58:21+02:00	2022-06-06T11:58:21+02:00		2022-06-06T11:58:22+02:00
hardware.error.cannot.co	Device controller 2	hardware	2	1	2022-06-07T08:27:19+02:00	2022-06-07T08:27:19+02:00		2022-06-07T08:27:22+02:00
hardware.error.cannot.co	Device controller 1	hardware	2	1	2022-06-07T08:27:21+02:00	2022-06-07T08:27:21+02:00		2022-06-07T08:27:22+02:00
hardware.error.cannot.co	Device controller 2	hardware	2	2	2022-06-08T10:17:46+02:00	2022-06-08T11:23:47+02:00		2022-06-08T10:17:49+02:00
hardware.error.cannot.co	Device controller 1	hardware	2	2	2022-06-08T10:17:48+02:00	2022-06-08T11:23:47+02:00		2022-06-08T10:17:49+02:00
hardware.error.cannot.co	Device controller 2	hardware	2	1	2022-06-08T11:23:50+02:00	2022-06-08T11:23:50+02:00		2022-06-08T11:23:53+02:00
hardware.error.cannot.co	Device controller 1	hardware	2	1	2022-06-08T11:23:52+02:00	2022-06-08T11:23:52+02:00		2022-06-08T11:23:53+02:00
hardware.error.cannot.co	Device controller 2	hardware	2	1	2022-06-08T15:29:24+02:00	2022-06-08T15:29:24+02:00		2022-06-08T15:29:27+02:00
hardware.error.cannot.co	Device controller 1	hardware	2	1	2022-06-08T15:29:26+02:00	2022-06-08T15:29:26+02:00		2022-06-08T15:29:27+02:00
hardware.error.cannot.co	Device controller 2	hardware	2	1	2022-06-15T08:24:42+02:00	2022-06-15T08:24:42+02:00		2022-06-15T08:24:45+02:00
hardware.error.cannot.co	Device controller 1	hardware	2	1	2022-06-15T08:24:44+02:00	2022-06-15T08:24:44+02:00		2022-06-15T08:24:45+02:00
hardware.error.cannot.co	Device controller 2	hardware	2	1	2022-06-16T14:36:28+02:00	2022-06-16T14:36:28+02:00		2022-06-16T14:36:31+02:00
hardware.error.cannot.co	Device controller 1	hardware	2	1	2022-06-16T14:36:30+02:00	2022-06-16T14:36:30+02:00		2022-06-16T14:36:31+02:00
hardware.error.cannot.co	Device controller 2	hardware	2	1	2022-06-17T11:01:18+02:00	2022-06-17T11:01:18+02:00		2022-06-17T11:01:21+02:00

Figure 3.1: HMI logs table example

- Mission Section: Returns a table of the active missions and the possibility to delete the selected one (**GetOneMissions()** method) or all missions (**GetAllMissions()** and **GetAllMissionsTable()** methods that provide the GET method to receive information from AntServer and the creation of the HMI table). The example in Figure 3.2 shows the mission table where the main parameters for each mission are reported.



MissionId	NavigationState	SchedulerState	MissionType	TransportState	StateInfo	Priority	AssignedTo	IsLoaded	FromNode	State	ToNode
1	3	0	Pick - Drop	2	2	Low	AGV1	FALSE	3200	1	4120
2	3	0	Pick - Drop	2	2	Low	AGV2	FALSE	3201		4125
3	3	0	Pick - Drop	2	2	Medium	None	FALSE	2312		4300

Figure 3.2: HMI mission table example

  - MissionID: Unique ID of the mission
  - NavigationState: State of the mission so, for instance, if it has been accepted or rejected
  - SchedulerState: Reason why the mission is not assigned to a vehicle, 0 if the assignment happened correctly
  - MissionType: is the type of the required mission and is the one defined in the POST method CreateMission JSON file (Table 3.1).
  - TransportState: State of the transport, for instance if it is transported, selected or terminated
  - StateInfo: Vehicle state error, for instance if the station or the destination are unknown
  - Priority: Urgency level of the mission

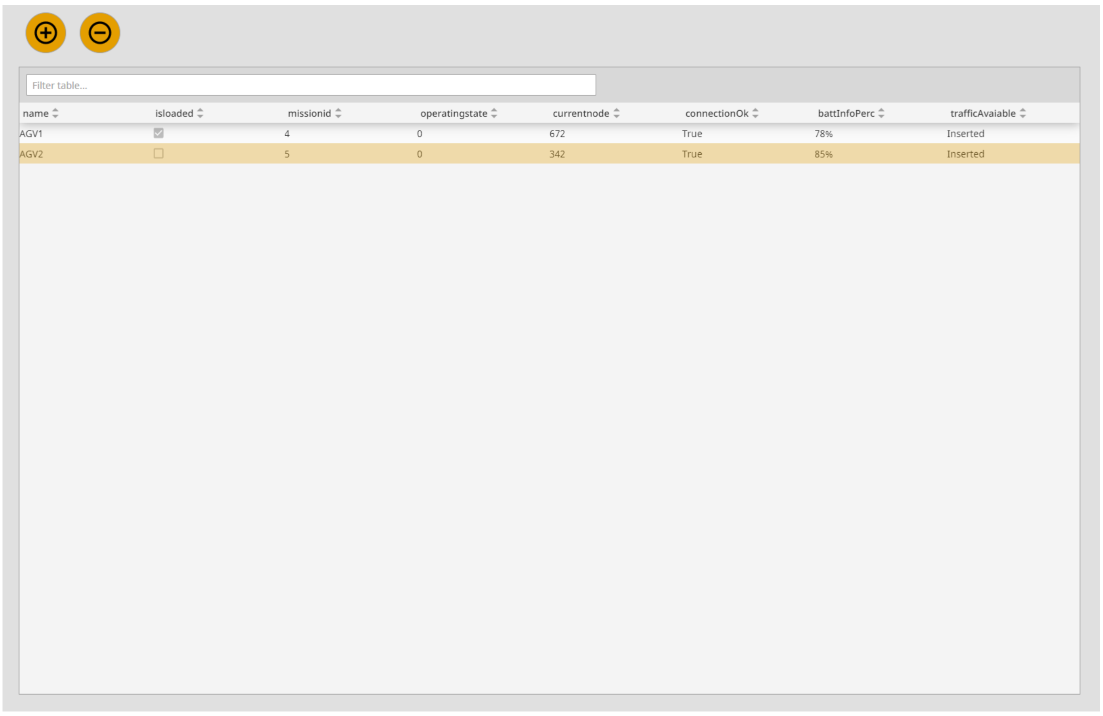
- AssignedTo: Name of the assigned vehicle
- isLoaded: If the payload is loaded on the assigned vehicle
- FromNode: Name of the pickup node: ¡Station.Node¿
- State: Mission state, for instance if it was accepted, rejected or cancelled
- ToNode: Name of the delivery node: ¡Station.Node¿

Details of number's value information is reported in the respective method comment section in Appendix A.
The Figure 3.2 shows just three missions as example, in real installation there will be a long set of missions to be gradually send and complete.

- Vehicle Section: Returns a table with the main AGV's information. In this section it is also possible to insert/remove a vehicle from the ANT view using the two icons on the top left of the HMI window.
  The parameters are asked to AntServer with the defined methods **GetAllVehicle()**, **GetOneVehicle()** that returns vehicle's information.

  - name: Name of the vehicle
  - isloaded: define if the payload is loaded on the vehicle or not
  - missionid: The ID of the assigned mission
  - operatingState: returns state of the vehicle, for instance if it is ready, if a mission has been assigned or not
  - currentnode: ID of the node in which the vehicle is (or the last crossed one)
  - connectionOK: defines if the connection between the server and the vehicle is established or not
  - battInfoPerc: Vehicle battery charge percentage
  - trafficAvailable: Defines if the vehicle is available or not for mission

  The shown parameters are just a sub-set of the parameters that the **GetAllVehicles()** method returns. The highlight parameters are those that are more important to be visualized from operators for control the AGVs. The example Figure 3.3 shows just two AGVs but in real installation there will be many vehicles operating simultaneously.



name	isloaded	missionid	operatingstate	currentnode	connectionOk	battInfoPerc	trafficAvaiable
AGV1	☑	4	0	672	True	78%	Inserted
AGV2	☐	5	0	342	True	85%	Inserted

Figure 3.3: HMI vehicles table example

- Map Section: Returns a representation of the AntLab environment with the evolution of the vehicles position.
  To receive all the map information from AntServer the BlueBotics object method **GetMap()** is used. This command provides all the available information about maps like node position, links, feature position and virtual walls.
  This information is used to draw the environment with the **MapRenderer** object, which uses the defined object to draw nodes, links, vehicles to create a virtual scheme of the environment. As can be seen in Figure 3.4 in the upper section the commands to zooming in and translate in the overall map can be found. The drawed vehicles position, as it is received from AntServer, it is almost real-time updated.

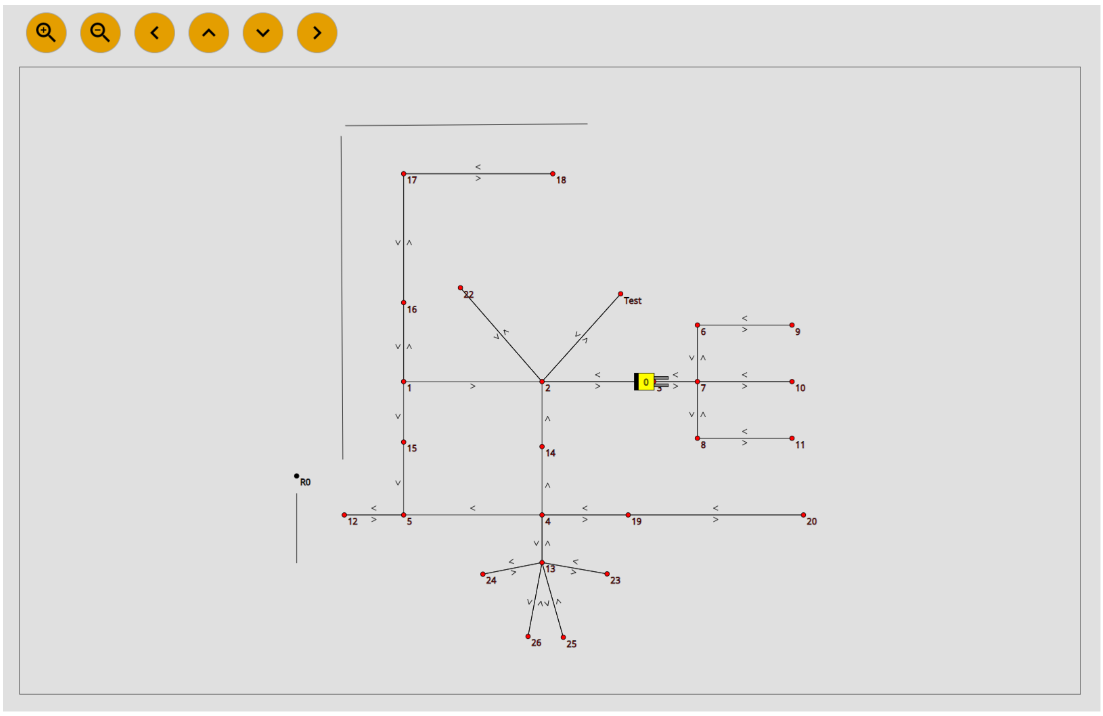


Figure 3.4: HMI map view example

Entering in the details of the Map Section program.
The basic principle of Ignition programming is the cyclic request to the vehicle and server for status and mission information. In this way, alarms can be acquired quickly (the request time is shorter) and other basic information is requested over a longer period of time, based on information importance.
The Map drawing require all the information about the allowed path and the Actions to be performed by the AGV at certain nodes. To this end, the objects that allow drawing have been defined, and those are: Line, Arrow, Text, Circle, Vehicle, Node and MapRenderer.

The **GetMap** method is an HTTP-GET method and is used to ask and then receive all the available information about the map like node position, links, feature position and virtual walls as reported in the first commented section of the method code.

```
def GetMap(self,levelID):
  #INPUT
    #[INT] levelID: Map id.
  #OUTPUT
    #[STRING] alias: Map name.
    #[STRING] description: Map description.
    #[INT] id: Map id.
    #[layers: Always two layers: one for localization information
   , the other for navigation.
      #[0]: Localization information
        #defaultstyleid:
          #[STRING] lines: localization.
          #[STRING] points: localization.
        #[STRING] desc: Segments and reflectors.
        #lines: Array of localization segments.
          #[INT ARRAY] coord: Localization segments [x1,y1,x2,y2]
        #[STRING] name: localization.
        #points: Array of reflectors
          #[FLOAT ARRAY] coord: Reflector position and covariance
    [x,y,cov].
      #[1] Navigation information
        #defaultstyleid
          #[STRING] lines: navigation
          #[STRING] points: navigation
        #[STRING] desc: Nodes, links and virtual walls
        #lines: Array of localization segments.
          #[INT ARRAY] coord: Navigation segments [x1,y1,x2,y2].
          #[STRING] styleid: Ivirtual-wall, The navigation
   segment is a virtual wall.
          #Empty field, It is a simple navigation node link.
        #[STRING] name: navigation
        #symbols: Array of navigation node position.
          #[FLOAT ARRAY] coord: Navigation node position [x,y,0].
          #[STRING] name: Navigation node name
    #[INT] level: Level id.
    #origin
      #[FLOAT ARRAY] offset: [x,y] offset.
      #[FLOAT] orientation: Map orientation.
```

```
#[STRING] group: Level id.
#[FLOAT ARRAY] offset: [x,y] offset.
#[FLOAT] orientation: Map orientation.
```

The code structure follows the general structure defined in the Section 3.2.2 and, as it is a GET method, is not required the JSON parameters file.

```
  try:
    if len(self.sessiontoken)<=0:
      self.Login()
  except:
    self.Login()
```

As in the other method, in the first part the **SessionToken** length is verified to check if the Login happened properly, otherwise it is performed.

```
  api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/maps/level/" + str(levelID) + "/data?&sessiontoken=" + self.sessiontoken
  response = requests.get(api_url)
  jsonfile = response.json()
```

The **api_url** is defined to allow the AntServer communication and the response is saved in the **jsonfile** variable.

```
  self.retcode = (jsonfile["retcode"])
  if self.retcode == 0:
    fileJSON = jsonfile["payload"]["data"]
    Map = []
    for data in fileJSON:
      CreateMap = MapDraw.MapRenderer(data, "black", 1, 8, "red", "black")
      pathLayout = "C:\Users\...\Desktop\layout.svg"
      system.file.writeFile(pathLayout, CreateMap.render())
  else:
    system.perspective.openPopup('1', 'Error', params = {'textLabel':'retcode = ' + self.retcode + '!'},showCloseIcon = False)
  return
```

After the retcode zero value verification data contained in the payload, one of the parameters in the JSON file, are send as input to the MapRenderer object contained in MapDraw.
MapRenderer object analyze data and define the .svg file containing the infor-

mation to draw the environment. The variables needed to define the object are: the .json file of the map information, line color, line width value, node diameter, node color and text color.
CreateMap is a MapRenderer object and the application of the **CreateMap.render()** method will define the string that will contain all the information required to built the svg file of the image. The object code definition can be found in Appendix A.
The svg file will contain information about the coordinates of the lines ((x, y) coordinates of two boundary points) and points, segments and reflectors, the orientation in each segment, the node ID and the node name (to identify the chosen action in special nodes) and the vehicles position.
The compilation of all this information allows the dynamic visualization of the environment with the AGVs in operation and the possibility of zooming and translating in the map.

3.3 Programming Siemens PLC

3.3.1 Safety

As reported in Chapter 2, Section 2.5.4.1 the Safety program uses the Safety block functions. These are easily recognisable by the yellow filling in the names of functions and variables.
The Laser Scanners outer data managing is one of the main aspects of the AGV's Safe behaviour.

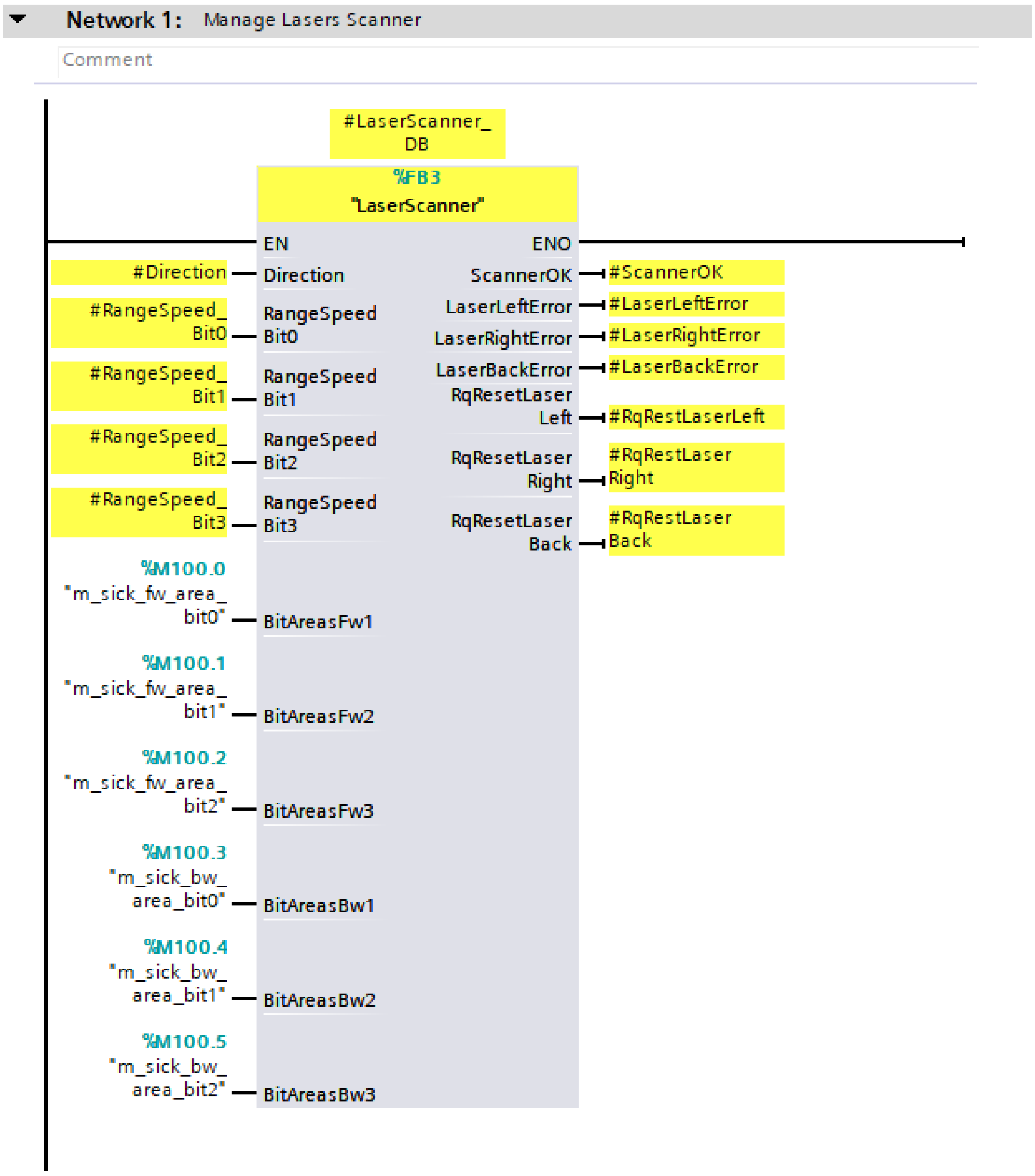


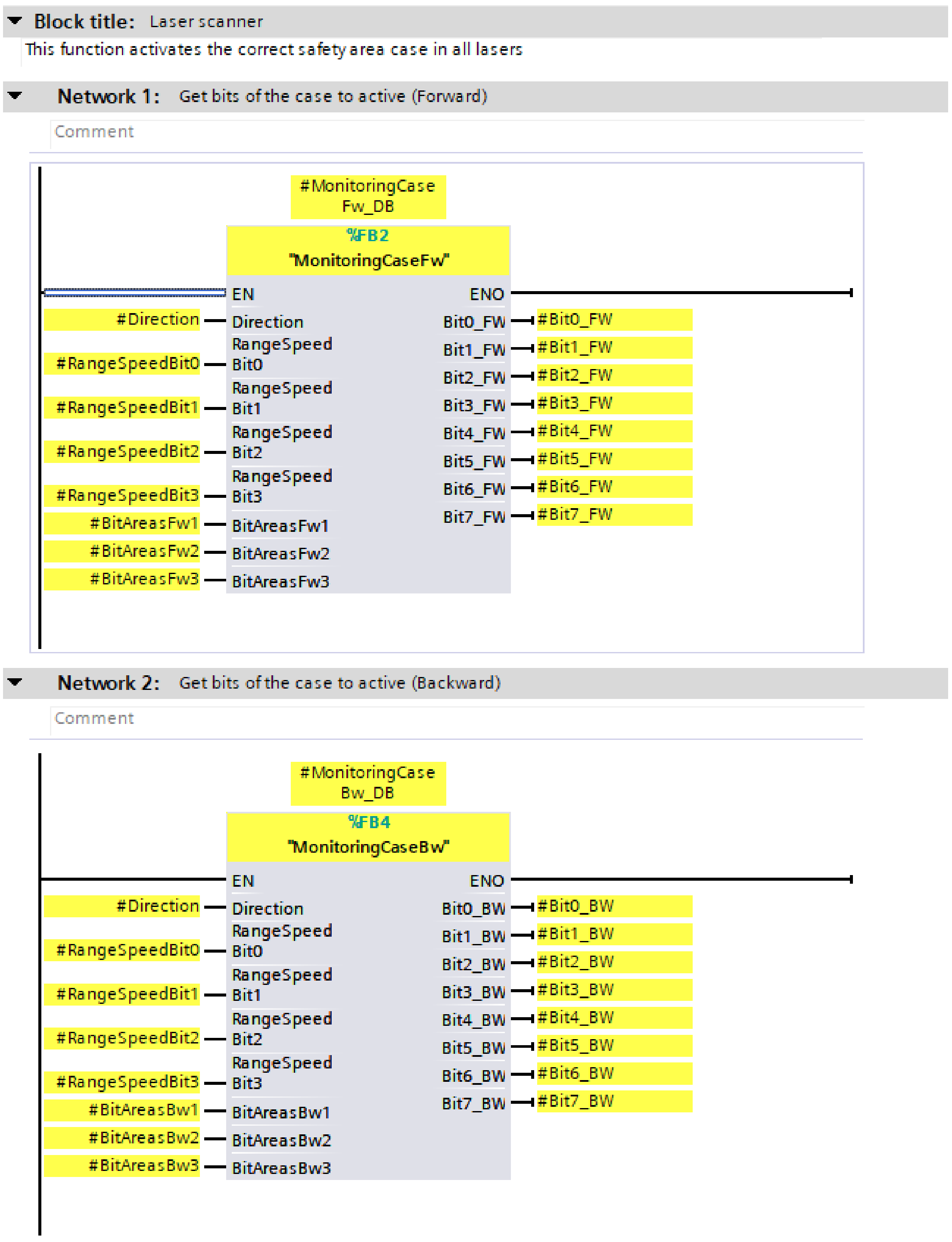
Figure 3.5: LaserScanner Function Block

The Figure 3.5 report the **LaserScanner** block function that set, in function of AGV velocity and bit information from the PLC, the safety area to be used.
It is a Function Block (FB) and then, whenever it is called, it creates a DB instance data block to store variables value. The input to this Function Block are the AGV direction, the RangeSpeeds (defined in 4 bits) and then the Merker variables (bits for the temporally store of variables or intermediates calculus results) that encode information received from the PLC to force specific safety areas in special sections of the environment.
The outer data are: the respective laser field for the chosen safety area and the state of the laser scanners like the reset request and if they are in error. The **ScannerOK** is a boolean control variable that is true if the Scanner didn't see object, so when the space around is free from obstacles.
The **#LaserError** is true if the Laser is in error and this can happen if the configuration is missing or if the safety area process fails and two areas are send to the Laser Scanner (only one is allowed). Laser Scanners must always have a one and only active safety area.
The other three variables, as many as Laser Scanners, are the Reset Request of each laser to enable correct operation from reset onwards.

The **LaserScanner** block function is composed of two block functions reported in Figure 3.6 and those are: **MonitoringCaseFw** and **MonitoringCaseBw**. This functions received as inputs the **LaserScanner** function inputs and send as output the bit composition that will activate the respective Safety Area for vehicle front and back.

The structure inside the two functions is almost identical, they just differs in variables combination to identify the proper area in those conditions.

The input variable are the **#Direction**, the **#RangeSpeed** (defined in 4 Bits) and the **#BitArea** FW and BW (each defined in 3 bits) from the PLC. The comparison between the input values allows the definition of a variable for each of the 8 bits in vehicle's Forward and Backward movement.


Block title: Laser scanner

This function activates the correct safety area case in all lasers

Network 1: Get bits of the case to active (Forward)

Comment

Network 2: Get bits of the case to active (Backward)

Comment

Figure 3.6: LaserScanner-Monitoring case

Figure 3.7 and in Figure 3.8 shows the code section to define the outer bit variables. Based on **#RangeSpeedBit**, **#Direction** and **#BitAreas** variables the different motion cases are identified.
The first step in Network 1 report the identification of **#RangeSpeedBit** and,

if one of the first two bits has value one and the other is zero the variable **#Speed1or2** is initialize ad set to True. This initial step allows to reduce the number of code lines.

Analysing the Network 2, that report the **Case1** identification, if the **#Speed1or2** is true; **#RangeSpeedBit2**, **#RangeSpeedBit3** are both zeros, and **#BitAreasFw1**, **#BitAreasFw2**, **#BitAreasFw3** are all zeros then the outer value is **#CaseFw1** set as out and then **Case1** is identified. The **#Case** variables are the identified Area based on bit composition. All those cases correspond to an area, there will be up to 14/16 different areas.

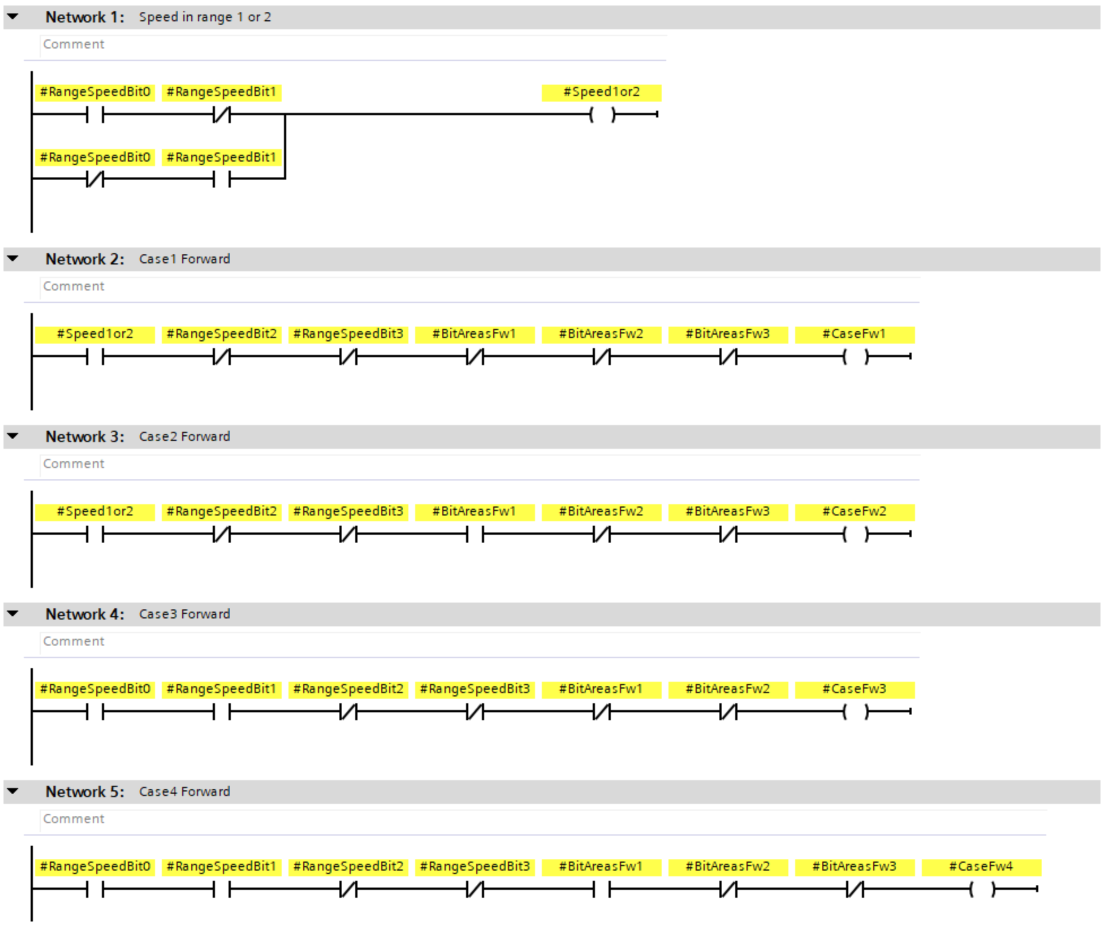


Figure 3.7: CaseFW definiton

Similar reasoning can be used to understand other networks:

- **#CaseFw2**: if **#Speed1or2**, **#BitAreasFw1** are 1 and the others (**#RangeSpeedBit2**, **#RangeSpeedBit3**, **#BitAreasFw2** and **#BitAreasFw3**) are zero.
- **#CaseFw3**: if both **#RangeSpeedBit0** and **#RangeSpeedBit1** are 1 and others (**#RangeSpeedBit2**, **#RangeSpeedBit3**, **#BitAreasFw1**, **#BitAreasFw2**) are zero,
- **#CaseFw4**: if **#RangeSpeedBit0**, **#RangeSpeedBit1** and **#BitAreasFw1** are 1 and others (**#RangeSpeedBit2**, **#RangeSpeedBit3**, **#BitAreasFw2** and **#BitAreasFw3**) are zero.

Other cases are built in a similar way, at the end of bits value comparison there will be the comparison of all those cases to end up with a value for each **#BitN_FW** value, with N ranging in [0,7].
As well **#BitN_BW** value, with N ranging in [0,7] are defined in **MonitoringCaseBW** Function Block.

The figure 3.8 report two examples in value assignment, respectively to **#Bit0_FW** and **#Bit1_FW**.
This is due to the fact that all the possible area cases (for both backward and forward) has to be defined in 8 bits for forward and 8 for backward.
The first assigned value **#Bit0_FW** is set to one if one of those reported cases hold, so if **#CaseFw1**, **#CaseFw3**, **#CaseFw5**, **#CaseFw7**, **#CaseFw9**, **#CaseFw11** or **#CaseFw13** holds.
The second one instead when one of **#CaseFw2**, **#CaseFw3**, **#CaseFw6**, **#CaseFw7**, **#CaseFw10**, **#CaseFw11** or **#CaseFw14** holds.

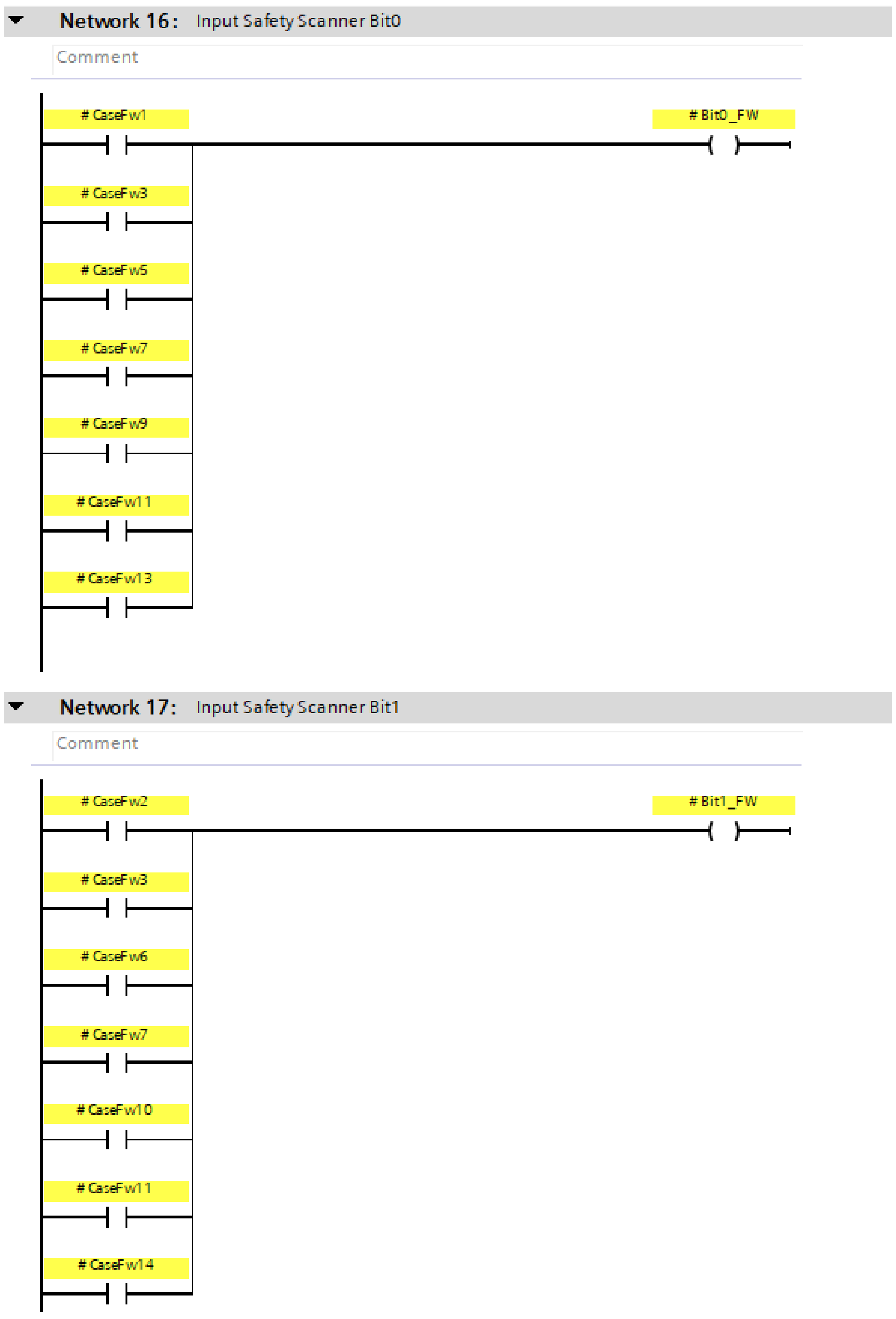


Figure 3.8: Bit_FW definition

The **#BitN** values are then assigned to the LaserScenners varibales **Q_LaserScanner.SetMonitorCaseNoTable_N**. Those variable are part of the Laser structure so its a direct assignment of laser activation. In a way similar all the other **Bit_FW** and **Bit_BW** are filled and then the Laser Scanner Monitoring areas are assigned as reported in Figure 3.9.

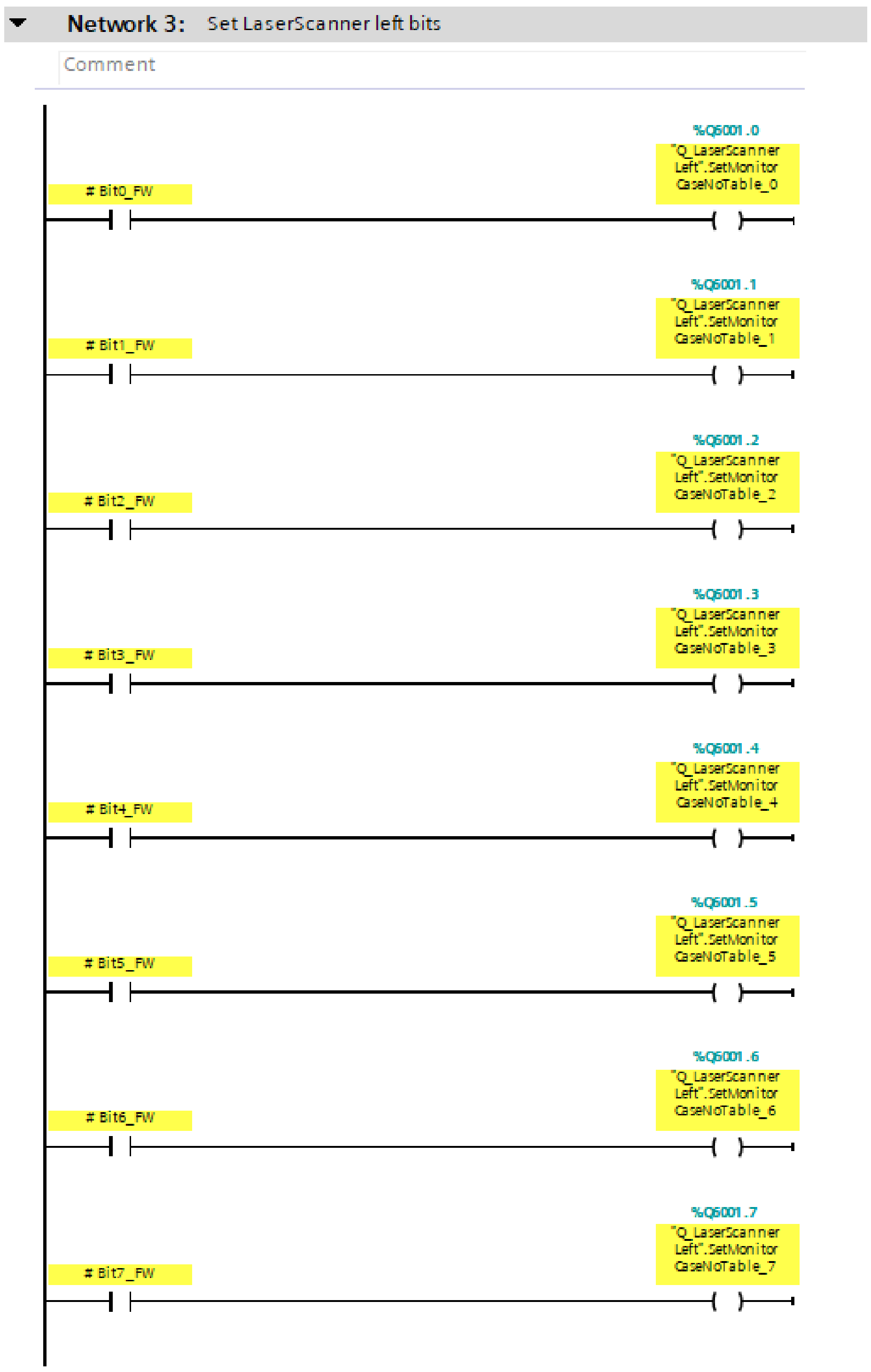

Network 3: Set LaserScanner left bits
Comment
%Q6001.0
"Q_LaserScanner Left".SetMonitorCaseNoTable_0
#Bit0_FW
%Q6001.1
"Q_LaserScanner Left".SetMonitorCaseNoTable_1
#Bit1_FW
%Q6001.2
"Q_LaserScanner Left".SetMonitorCaseNoTable_2
#Bit2_FW
%Q6001.3
"Q_LaserScanner Left".SetMonitorCaseNoTable_3
#Bit3_FW
%Q6001.4
"Q_LaserScanner Left".SetMonitorCaseNoTable_4
#Bit4_FW
%Q6001.5
"Q_LaserScanner Left".SetMonitorCaseNoTable_5
#Bit5_FW
%Q6001.6
"Q_LaserScanner Left".SetMonitorCaseNoTable_6
#Bit6_FW
%Q6001.7
"Q_LaserScanner Left".SetMonitorCaseNoTable_7
#Bit7_FW

Figure 3.9: Left Laser Scanner areas setting

3.3.2 Non Safety

The Non Safety functionalities defined in Siemens PLC are those not related to a Safe behaviour of the Skilled AGV. Those can use Safety variables (identified by the yellow color) but not modified them value, as this is allowed only from safety variables and by using safety function blocks.
The Figure 3.10 report an example of Non-Safe behaviour definition. The **enable_power** variable is defined from the Safety variables about power button pushing and laser scanner status. In particular, a temporal variable is defined as **#t_temp1** and its value is one if **i_power_pushbutt** is one, meaning that the power on button has been pressed, and one of the two case holds:

- The three laser Scanner status, **I_LaserScanner.StatusSafeCutOffPath1**, for the Right, Left and Back Laser are true, and so the Laser scanner field of view in the starting are case is free from obstacles. Those values are False when the Laser Scanner detect obstacles in the Protective field (the red one in laser scanners field image in Chapter 1 Section 1.2.5.1)

- The **Main_Safety_RTG1_DB.BypassOn** is true, meaning that the variable **BypassOn** inside the DB MainSafety is true and then the manual choice to turn it on, despite of some error, is done. The **BypassOn** is the variable associated to the operator button in the joystick for move the vehicle with small velocity in manual mode without safety area.

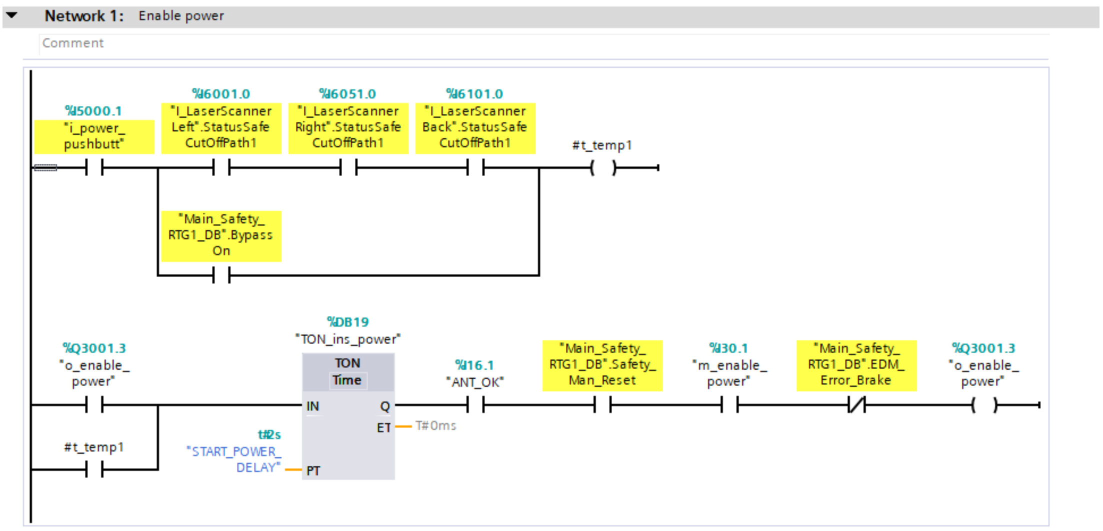


Figure 3.10: Enable power definition

This variable value (**#t_temp1**)is the used in **o_enable_power** value assignment. The first condition requires that one of the two variable, **o_enable_power** or **#t_temp1**, is true and this value is true for 2 seconds (**START_POWER_DELAY**).
This structuring allows the enable power variable to keep the value 1 once it is set, cause for sure the **o_enable_power** will be one for two second once it is set as one.

The other requests that has to be true are:

- **ANT_OK** so the ANT exchanging information holds properly
- **Main_Safety_RTG1_DB.Safety_Man_Reset** that represent the safety condition for set the AGV in manual mode
- **m_enable_power** represent the monitoring on safety devices like Lasers, chain, bypass button and encoders
- NOT **DB.EDM_Error_Brake** so the brakes are working properly

then **o_enable_power** is true so the power is sent to all the drives, otherwise is false and the vehicle power-on is not allowed.

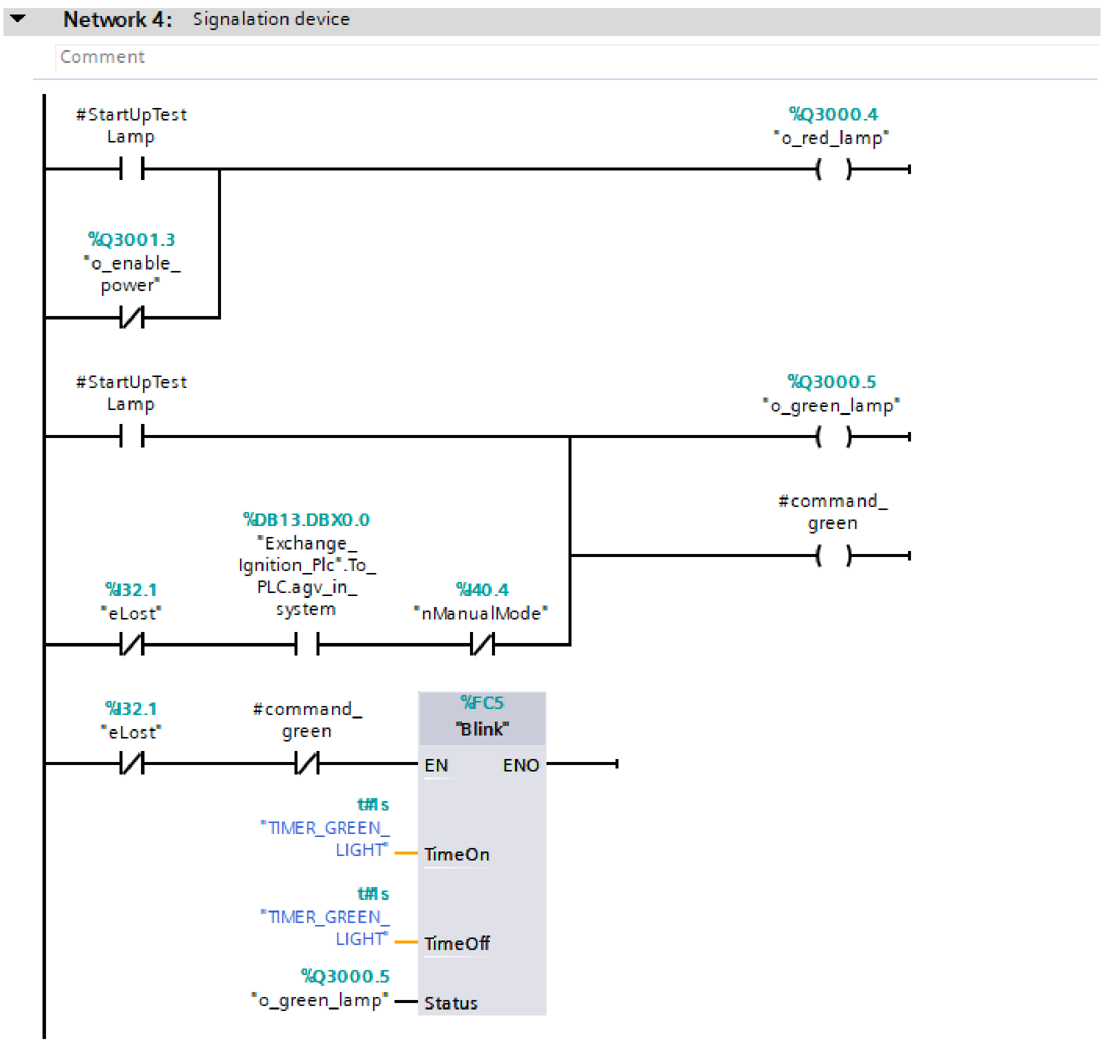


Figure 3.11: Red and Green light blinking cases

Figure 3.11 defines the green and red light blinking or not based on the state of the vehicle. This, clearly, depends also on the **o_enalbe_power** value. The **#StartUpTestLamp** is a boolean variable that is true in the time interval that follows the power on of the vehicle.
The red lamp will blink in case of Start up test or in case the **o_enalbe_power** is false, so when the vehicle, for some reason, has the power disabled.

The green lamp will be a static light in case of StartUp Test and in case the vehicle is not Lost, the AGV is insert in the system
(**Exchange_Ignition_Plc.To_PLC.agv_in_system** is the variable that is set to true when the vehicle is insert and is contained in the DB encharged in the exchange between PLC and Ignition) and it is not in ManualMode. When all this condition holds also the **#command_green** variable is initialized ad true. This variable is used for the green light blinking, if the vehicle is not lost and

the **#command_green** is false (meaning that the vehicle is not instert or is in manual mode) to provide the visual information of no-lost vehicle.

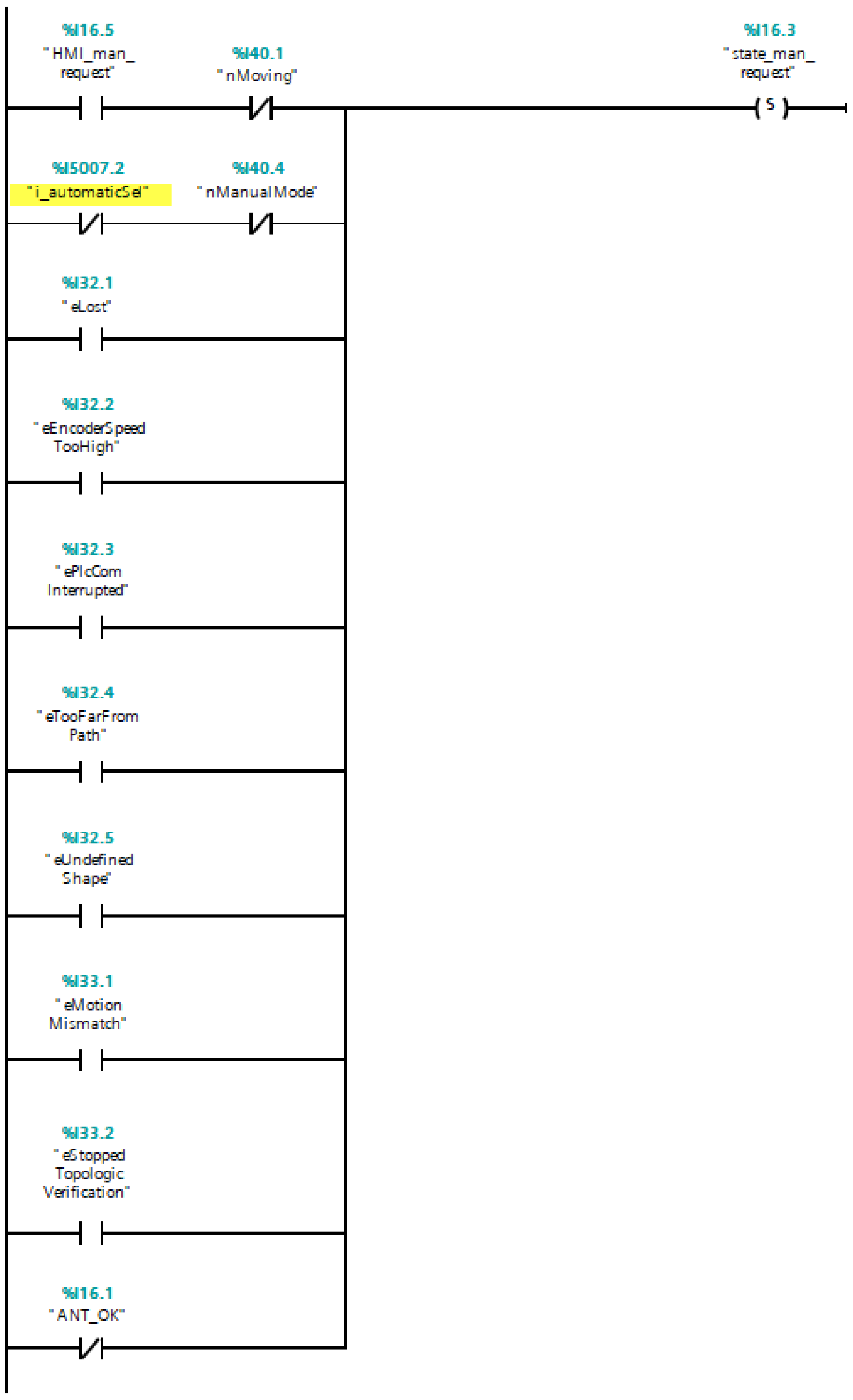


Figure 3.12: Set Manual request

Figure 3.12 report the code section to set the **state_man_request** so the raising of the request of manual setting. The outer coil with the S inside is the SET variable and, when it is activated, the associated bit will take value 1. In the case, the **state_man_request** will be set to one in case the:

- **HMI_man_request** and **NOT nMoving**: The manual request is received from the HMI (**HMI_man_request** is set to one) and the vehicle is not moving
- **i_automaticSel** and **NOT nManualMode**: The Safety automaticSelector is not true and the vehcile is not already in manual mode, so the raising of the request is performed
- **eLost**: The vehicle is Lost
- **eEncoderSpeedTooHigh**: The speed in the encoder is too high
- **ePlcComInterrupted**: The PLC communication has been interrupted
- **eTooFarFromPath**: The AGV is "too far from the path"
- **eUndefinedShape**: No shape defined for the requested safety case
- **eMotionMismatch**: The vehicle does not correctly follow its motion command
- **eStoppedTopologicVerification**: Important localization features are not visible
- **NOT ANT_OK**: error in agv insertion

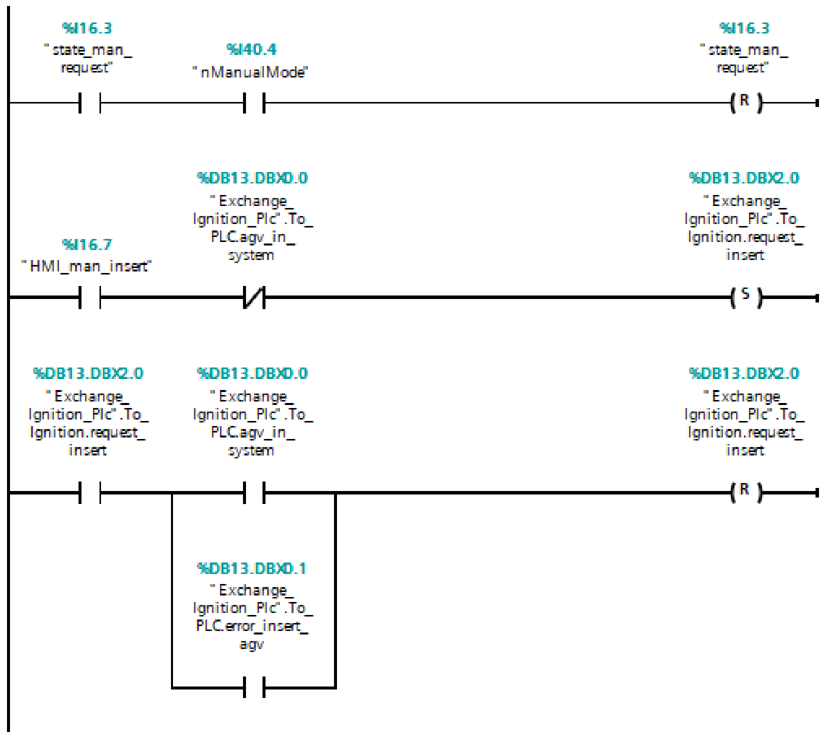

%I16.3
"state_man_request"
%I40.4
"nManualMode"
%I16.3
"state_man_request"
R
%DB13.DBX0.0
"Exchange_Ignition_Plc".To_PLC.agv_in_system
%DB13.DBX2.0
"Exchange_Ignition_Plc".To_Ignition.request_insert
%I16.7
"HMI_man_insert"
S
%DB13.DBX2.0
"Exchange_Ignition_Plc".To_Ignition.request_insert
%DB13.DBX0.0
"Exchange_Ignition_Plc".To_PLC.agv_in_system
%DB13.DBX2.0
"Exchange_Ignition_Plc".To_Ignition.request_insert
R
%DB13.DBX0.1
"Exchange_Ignition_Plc".To_PLC.error_insert_agv

Figure 3.13: Set and Reset Insertion vehicle request

Figure 3.13 represent the code section that provide the managing of vehicle extraction and insert request. In the firs row the **state_man_request** is reset to zero in case the **state_man_request** holds and the vehicle is in Manual mode, meaning that the request has been satisfied. Then the setting of the insertion request is done if the insertion command is received by the HMI (**HMI_man_insert** is true) and the AGV is not in system (**Exchange_Ignition_Plc.To_PLCagv_in_system** is false).

The insertion request is reset, so turning back to zero the bit value, when there's the **Ignition.request_insert** true and one of the two conditions holds: the AGV is in system or there's an error in insert the vehicle.

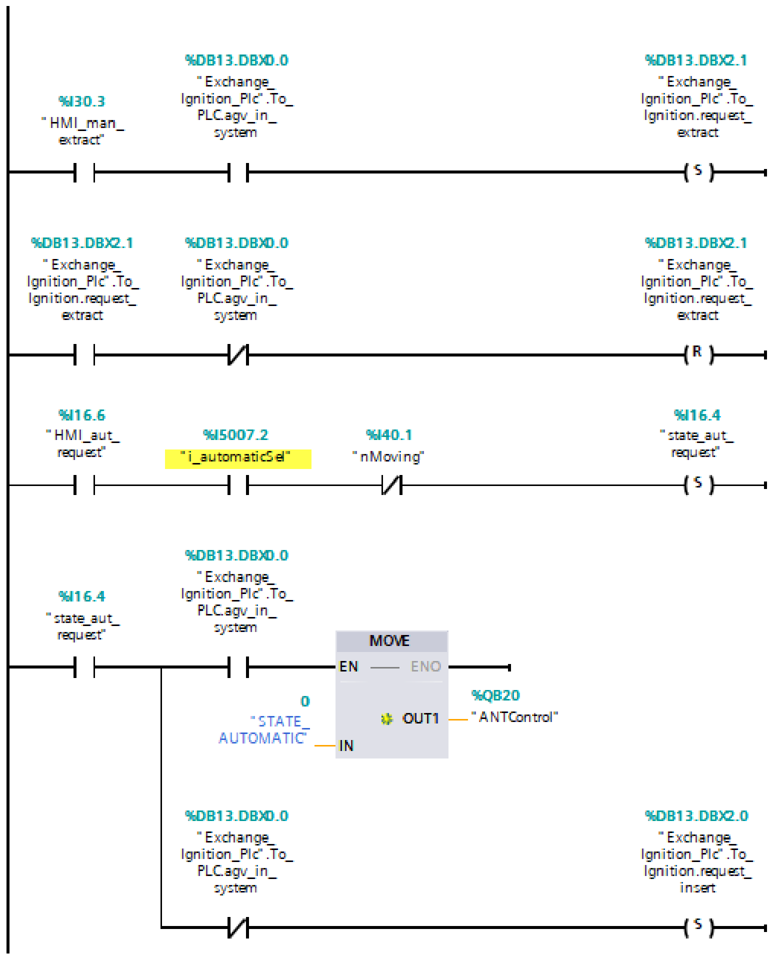


Figure 3.14: Set and Reset extraction vehicle request and set automatic vehicle request

Follows (Figure 3.14) the extraction request setting when there's the **HMI_man_extract** request and the AGV is in system. Then the reset of the extract request when the Ignition request of extraction is true and the vehicle is no more in the system, meaning it has been extracted.
The automatic state request is set to one if the request automatic value in the HMI is true, the Safety automatic/manual selector is in automatic mode and the vehicle is not moving.
If the request holds and the AGV is in system, then a true value is send as input to the MOVE block function.
A MOVE function is used to output a specific value when a defined input is True. To achieve this, an EN expression is used with the MOVE function. The EN expression uses a Boolean input from a direct input variable (agv_in_system). The other has a constant value, "STATE AUTOMATIC". When the direct input variable is in State 1 (True), the constant value is moved to the output (so the output is "STATE AUTOMATIC"). When the input variable is in State 0 (false),

the output is NON-"STATE AUTOMATIC". This is the block scheme version of an if-then code structuring in ST program.
If the **state_aut_request** is true and the AGV is not in system than the insertion request is raised.

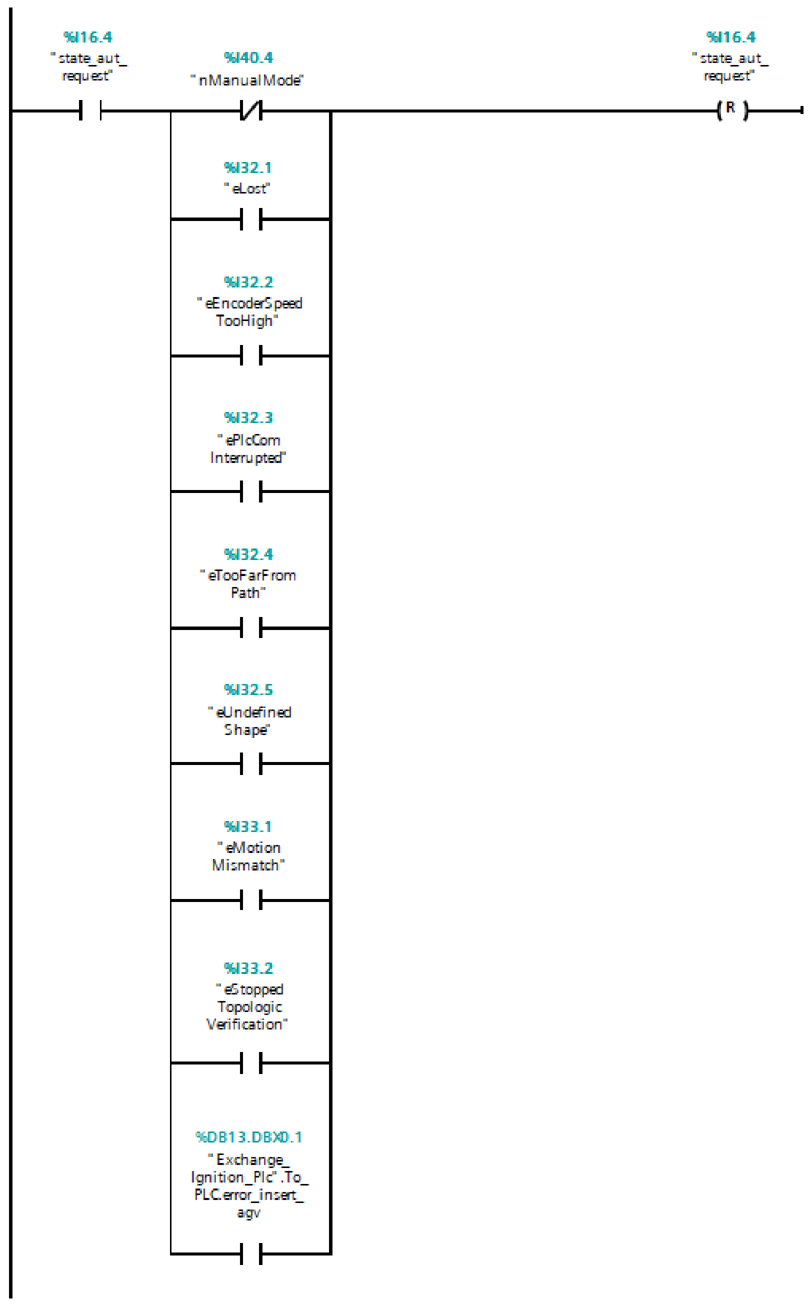


Figure 3.15: Reset of vehicle automatic request

Figure 3.15 is the coding section that resets the **state_aut_request**. This holds if the **state_aut_request** has been raised and one of the following conditions holds:

- NOT **nManualMode**: The vehicle is not in manual mode
- **eLost**: The vehicle is lost (in automatic mode)
- **eEncoderSpeedTooHigh**: The speed of a wheel was too high
- **ePlcComInterrupted**: The communication with the PLC has been interrupted
- **eTooFarFromPath**: The vehicle is too far from the path (in automatic mode)
- **eUndefinedShape**: No shape defined for the requested safety case
- **eMotionMismatch**: The vehicle does not correctly follow its motion command
- **eStoppedTopologicVerification**: Important localization features are not visible
- **ExchangeIgnition_Plc.To_PLC.error_insert_agv**: error in agv insertion

This means that with this condition the vehicle cannot be set in automatic mode so the set request is reset to zero. The main parts of the PLC code are structured in Ladder to allow a easy readable environment for testers that are not so confident with the structured test. Clearly the same target could have been reached with SF function blocks.

Chapter 4

Considerations and results

4.1 Final considerations

The new Skilled AGV project will require an year to be fully implemented and used. The explained project requires 6 months, it is not completed but it touched most of the required step for a new Skilled AGV definition.
In the first part, the studying of BlueBotics AntLite$^+$ and AntServer manual has been done to understand them technology base idea for controlling and developing.

BlueBotics technical specification has an important role in the study cause the coupling between the allowed devices (from laser scanner to traction and steering drives) and the one supported by the actual Skilled AGV was necessary for the project developing.

After that, the reasoning about the managing of all the AGV actions and moving has been done. From the previous experiences in AGV installations most of the possible cases problem solution has been thought.

Starting from the PLC choice two hardware main block project has been defined and firstly the Rockwell PLC was preferable. This was due to the Ethernet/IP communication protocol applied from both Rockwell and AntLite$^+$.
With the PN/MF coupler the two alternatives were very similar from the hardware point of view, then the Siemens PLC has been chosen for standardization reasons as Euroimpianti uses Siemens PLC for most of them products.

The Software definition has faced all the requirements of a complete and correct AGV navigation and acting.

The conversion from Laser Triangulation to Natural Navigation takes many changes, allowing to reach still un-reached target.

Laser Guided Vehicles (LGV) installation requires the design of reflectors position as, at each time instant, the vehicle must be able to see three of them to perform its localization. This navigation technique is accurate and allows safe high speed navigation but requires the periodical reflectors cleaning from dirty that compromise the lasers reflection and then with AGV localization. An extra attention is reserved to the loads position that may not allow AGV to see the reflectors.
The path modification will require the new design of the allowed path and the new vehicle's installation by moving it in the environment in manual mode to understand the new reflectors position.

AGVs that navigate with natural navigation are an innovative technology, and the automation world is moving to this direction. The installation is easier then LGV as requires reflector positioning just in some conditions where the environment do not provide enough features for AGV localization.
The AGV installation still requires the AGV to be moved around in manual mode to learn the environment, then the drawing of the allowed path is done based on laser scanners data of the environment features like walls, static objects and reflectors.
This makes the technology very flexible and the application is allowed in almost any environment. The cons of this technology is the possibility that the vehicle could not find itself in very chaotic environment.
In some cases the switch from Natural Navigation to Laser Triangulation Navigation could be required. The new Skilled AGV with BlueBotics technology can deal with it and allows the correct safe behaviour of the AGV in almost any possible environment.

Natural navigation technology allows the AGV to navigate and localize in a various type of environments, allowing the new Skilled AGV to be more adaptable to the installation environment conditions.
The installation will be faster and an higher number of AGVs can be involved in the same project thanks to BlueBotics technology traffic managing.

The implementation of the BlueBotics technology will make the Stationary lighten and optimized the used software. The router is constantly updated and the human interface is dynamic and easy to use as it is intuitive and readable.

This makes the new AGV more stable, easier and optimized.

The object oriented developed program structuring is less demanding in memory and variable refreshing so the performance increases. The developed code is generic and adaptable in any kind of situation and any vehicle type with little changes. The software modification are easy as it is well commented and readable.

Ignition can interface with any kind of database and warehouse managing software. This makes Skilled AGV product more vendible and updated.

Natural navigation is the future in autonomous navigation vehicles so this new project will be the base for all futures Skilled AGVs.

Appendix A

Ignition code

A.1 Ignition-AntServer communication

```python
import requests
import json

class Server:
  def __init__(self,serverJSON):

    self.servPause = serverJSON["antServerPause"]
    self.pauseButVisible = serverJSON["displayPauseButton"]
    licenseValidity = serverJSON["licenseValidity"]
    self.nameLic = []
    self.timeleftLic = []

    for license in licenseValidity:
      self.nameLic.append(license["name"])
      self.timeleftLic.append(license["timeleft"])

    self.licErrorMsg = (serverJSON["licenseErrorMsg"]).encode()
    self.licErrorCode = serverJSON["licenseErrorCode"]


class BlueBotics:
  def __init__(self,ip = "localhost",port = "8081",user="admin",
   psw="123456"):
    self.ip = ip
    self.port = port
    self.user = user
    self.psw = psw

  def GetAllMissions(self):

    #INPUT

    #OUTPUT
      #[STRING] missionid: Unique id of the mission.
      #[TIME] dispatchtime: Absolute date when the mission is started
      #[INT] timetodestination: Estimated time to end the mission
      #[INT] navigationstate
        #0: Received.
        #1: Accepted (=planned).
        #2: Rejected.
        #3: Started (=running).
        #4: Terminated (=successful).
        #5: Cancelled.
      #[INT] schedulerstate: Reason why the mission is not assigned to a vehicle:
          #0: A vehicle has been assigned to the mission.
          #1: There is no vehicle in state ready to get a mission.
          #2: There are vehicles available to get mission but they cannot reach the mission start node.
          #3: There are no vehicle of type compatible with the mission.
          #4: The vehicle associated with this linked mission is not available.
          #5: For mission of type deadline, the deadline is not reached.
          #6: For mission of type priority, the priority is too low
      #[FLOAT] totalmissiontime: Estimated or effective time to execute the mission
      #[TIME] arrivingtime: Estimated or effective end time of the mission.
      #[INT] missiontype
        #0: Transport from node to station.
        #1: Move to station.
        #2: Waiting lane.
        #7: Transport to node.
        #8: Move to node.
        #9: Transport from station to station.
        #10: Move a specific vehicle to a node.
        #12: Move to a loop.
      #[BIGINT] groupid Id of a group of missions when generated
   at the same time by a mission rule
      #[INT] transportstate
        #0: New.
        #1: Accepted.
        #2: Rejected.
        #3: Assigned.
        #4: Moving.
        #5: Transporting to selector.
        #6: Selecting delivery from start.
        #7: Delivering.
        #8: Terminated.
        #9: Cancelled.
        #10: Error.
        #11: Cancelling.
        #12: Selecting pick up node.
        #13: Selecting delivery from selector.
        #14: Moving to departure selector
      #[BIGINT] missionrule: Id of the mission rule which created
    the mission. Empty if the mission has not been created by a
   mission rule.
      #[INT] stateinfo
        #0: Undefined.
        #1: Unknown loop.
        #2: No insertion node into loop.
        #3: Unknown destination.
        #4: Transport from unknown location.
        #5: Transport from unknown station.
        #6: No node available in pick up station.
        #7: Unknown station.
        #8: No node available in drop off station.
        #9: Unparsable mission request.
        #10: Timetables are closed.
        #11: Deadline is in the past.
      #[INT/STRING] priority
        #0: No priority.
        #1 or Low,
        #2 or Medium,
        #3 or High.
      #[STRING] assignedto: Name of the assigned vehicle.
      #[STRING] payloadstatus
        #Waiting (paylolad is ready to be picked up).
        #PickedUp.
        #Delivered.
```

```
        #Withdrawn (the payload is discarded from the system).
      #[BOOL] isloaded: true, Payload is loaded on the assigned vehicle.
              #false: Payload is not loaded on the assigned vehicle.
      #[STRING] payload: Name of the payload.
      #[BOOL] istoday: true, The mission deadline is before 23:59 PM.
             #false: Otherwise
      #[STRING] fromnode: Name of the pickup node: <Station.Node>.
      #[INT] state
        #0: Pending.
        #1: Dispatched to Scheduler (but answer not received yet)
        #2: Accepted by Scheduler.
        #3: Rejected by Scheduler.
        #4: Missed (the mission is canceled because its deadline is missed).
        #5: Cancelling.
        #6: Cancelled.
      #[TIME] deadline: The absolute date from which the mission can be assigned to a vehicle.
      #[STRING] tonode: Name of the delivery node: <Station.Node>
      #[BOOL] askedforcancellation
        #true: The mission is asked to be cancelled at the next monitorCancellation action.
        #false: Otherwise.

    try:
      if len(self.sessiontoken)<=0:
        self.Login()
    except:
      self.Login()

    api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/missions?&sessiontoken=" + self.sessiontoken
    response = requests.get(api_url)
    jsonfile = response.json()

    self.retcode = (jsonfile["retcode"])
    if self.retcode == 0:
      missionsJSON = (jsonfile["payload"]["missions"])
      Missions = []
      for missJSON in missionsJSON:
        Missions.append(MissionObj(missJSON))
    else:
        system.perspective.openPopup('1', 'Error', params = {'textLabel':'retcode = ' + self.retcode + '!'},showCloseIcon = False)

    return Missions

def GetAllMissionsTable(self,path="[default]JSON/GetAllMissions/AllMissions"):

    #INPUT
      #[STRING] path: path of the tag where the function writes the return value

    #OUTPUT
      #THE SAME OF "GETALLMISSIONS()"

    Missions = self.GetAllMissions()

    # First create a list that contains the headers
    headers = ["MissionId", "DispatchTime", "TimeToDest", "NavigationState", "SchedulerState", "TotalMissTime", "ArrivingTime",\
           "MissionType", "GroupId", "TransportState", \
           "MissionRule", "StateInfo", "Priority", "AssignedTo",\
           "PayLoadStatus", "IsLoaded", "Payload", \
           "IsToday", "FromNode", "State", "Deadline", "ToNode", "AskedForCanc"]

    # Then create an empty list, this will house our data.
    data = []

    for mission in Missions:

      # Then add each row to the list. Note that each row is also a list object.
      data.append([mission.missionid, mission.dispatchtime, mission.timetodestination, mission.navigationstate, \
    mission.schedulerstate, mission.totalmissiontime, \
    mission.arrivingtime, mission.missiontype, mission.groupid, mission.transportstate, mission.missionrule,mission.stateinfo,
    mission.priority,mission.assignedto, mission.payloadstatus, mission.isloaded, mission.payload, mission.istoday, \
    mission.fromnode, mission.state, mission.deadline, \
    mission.tonode, mission.askedforcanc])

    # Finally, both the headers and data lists are used in the
   function to create a Dataset object
    TableMissions = system.dataset.toDataSet(headers, data)

    TagPath = [path]
    TagValue = [TableMissions]
    system.tag.writeBlocking(TagPath,TagValue)

  def GetOneMission(self,missionID):

    #INPUT
      #[INT] missionID: Id of the mission

    #OUTPUT
      #THE SAME OF "GETALLMISSIONS()"

    try:
      if len(self.sessiontoken)<=0:
        self.Login()
    except:
      self.Login()

    api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/
   missions/" + str(missionID) + "?&sessiontoken=" + self.
   sessiontoken
    response = requests.get(api_url)
    jsonfile = response.json()

    self.retcode = (jsonfile["retcode"])
    if self.retcode == 0:
      missJSON = (jsonfile["payload"]["missions"])
      Mission = MissionObj(missJSON[0])
    else:
        system.perspective.openPopup('1', 'Error', params = {'
   textLabel':'retcode = ' + self.retcode + '!'},showCloseIcon =
   False)

    return Mission
```

```
def CancelAllMissions(self):

  #INPUT

  #OUTPUT
    #[STRING ARRAY] cancelled: List of the cancelled mission
        ids.

  try:
    if len(self.sessiontoken)<=0:
      self.Login()
  except:
    self.Login()

  api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/
 missions?&sessiontoken=" + self.sessiontoken
  response = requests.delete(api_url)
  jsonfile = response.json()

  self.retcode = (jsonfile["retcode"])
  if self.retcode == 0:
    cancelled = jsonfile["payload"]["cancelled"]
  else:
      system.perspective.openPopup('1', 'Error', params = {'
 textLabel':'retcode = ' + self.retcode + '!'},showCloseIcon =
 False)

  return cancelled

def CancelOneMission(self,missionID):

  #INPUT
    #[INT] missionID: Id of the mission

  #OUTPUT
    #[STRING] missionid: Mission id.
    #[BOOL] cancelled: true, The mission is cancelled.
              #false, The mission is not cancelled.

  try:
    if len(self.sessiontoken)<=0:
      self.Login()
  except:
    self.Login()

  api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/missions/" + str(missionID) + "?&sessiontoken=" + self.sessiontoken
  response = requests.delete(api_url)
  jsonfile = response.json()

  self.retcode = (jsonfile["retcode"])
  if self.retcode == 0:
    missID = jsonfile["payload"]["missionid"].encode()
    cancelled = jsonfile["payload"]["cancelled"]
  else:
      system.perspective.openPopup('1', 'Error', params = {'textLabel':'retcode = ' + self.retcode + '!'},showCloseIcon = False)

  return missID,cancelled

def InsertVehicle(self,vehicleName, nodeId, forceInsertion="True"):

  #INPUT
    #[STRING] vehicleName: name of the vehicle that you want to insert
    #[STRING] nodeId: Id or alias of the node where the vehicle must be inserted.
    #[BOOL] forceInsertion: Optional argument to force the insertion even if the vehicle is already inserted
                #in another ANT server (the vehicle will be set in error in the other ANT server).

  #OUTPUT
    #[VehicleOBJ]: Vehicle object
    #[STRING] warningMessage: Warning message related to a problem to be noticed but which does not prevent insertion.
    #[INT] warningCode
      #6: License check failed (no license for this vehicle). Error temporarily ignored using license xxx.
      #7: License check failed (vehicle bootfile must be upgraded). Error temporarily ignored using license xxx.
      #8: License check failed (license file xxx must be upgraded). Error temporarily ignored using license xxx.
    #[STRING ARRAY] warningParameters Parameters related to the warning message (vehicle name, license information, etc.)

  try:
    if len(self.sessiontoken)<=0:
      self.Login()
  except:
    self.Login()

  api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/vehicles/" + str(vehicleName) + "/command?&sessiontoken=" + self.sessiontoken
  myobj = '{"command":'
  myobj = myobj + '{"name": "insert",'
  myobj = myobj + '"args":{'
  myobj = myobj + '"nodeId": "' + str(nodeId) + '",'
  myobj = myobj + '"forceInsertion": "' + str(forceInsertion) + '"'
  myobj = myobj + '}}}'

  myobj = json.loads(myobj)
  response = requests.post(api_url, json = myobj)
  jsonfile = response.json()

  self.retcode = (jsonfile["retcode"])
  if self.retcode == 0:
    vehicleJSON = (jsonfile["payload"]["vehicle"])
    vehicle = VehicleObj(vehicleJSON)
    warningMessage = \
      (jsonfile["payload"]["warningMessage"]).encode()
    warningCode = (jsonfile["payload"]["warningCode"])
    warningParameters = \
      (jsonfile["payload"]["warningParameters"])

  return self.retcode,vehicle,warningMessage,warningCode,warningParameters

def ExtractVehicle(self,vehicleName):

  #INPUT
    #[STRING] vehicleName: name of the vehicle that you want to insert

  #OUTPUT
    #[VehicleOBJ]: Vehicle object

  try:
    if len(self.sessiontoken)<=0:
      self.Login()
  except:
    self.Login()

  api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/vehicles/" + str(vehicleName) + "/command?&sessiontoken=" + self.sessiontoken
  myobj = '{"command":'
  myobj = myobj + '{"name": "extract",'
  myobj = myobj + '"args":{}'
  myobj = myobj + '}}'

  myobj = json.loads(myobj)
  response = requests.post(api_url, json = myobj)
  jsonfile = response.json()

  self.retcode = (jsonfile["retcode"])
  if self.retcode == 0:
    vehicleJSON = (jsonfile["payload"]["vehicle"])
    vehicle = VehicleObj(vehicleJSON)

  return self.retcode, vehicle

def GetAllVehicles(self):

  #INPUT

  #OUTPUT
    #[STRING] name: Name of the vehicle.
    #[BOOL] isloaded: true, A payload is loaded on the vehicle.
             #false: A payload is not loaded on the vehicle.
    #[STRING] missionid: The id of the assigned mission
    #[STRING] payload The type of payload.
    #[INT] operatingstate
      #0: Not inserted
      #1: Ready
      #2: Assigned to a mission
      #3: Not available
      #4: Paused
      #5: Asleep (in a loop)
      #6: In error
    #[TIME] timestamp: Time of the last received data.
    #[INT ARRAY] path: List of nodes names forming the path of the current mission.
    #[STRING] nameAct: Name of the action that the vehicle is currently executing.
    #argsAct: Details about the current action (for example the node or device id) depending on its state.
    #[STRING] sourceidAct: vehicle name.
    #[STRING] sourcetypeAct: Always "vehicle"
    #[FLOAT] coord: Vehicle position [x, y, z].
    #[FLOAT] course: Orientation of the vehicle.
    #[STRING] map: Name of the current map
    #[INT] group: Floor level.
    #[INT] currentnode: Id of the current node (the last crossed node if between two nodes).
    #[STRING ARRAY] missProgress:  [Time to destination [s], Arrival time [time], Elapsed time [s] and progress [%]].
    #[STRING ARRAY] missInfo: [start node alias or current node alias, target node  alias, final destination alias].
    #[STRING ARRAY] connectionOk: true, A connection between the server and the vehicle is established,
            #false: No connection has been established or the connection has been lost.
    #[STRING ARRAY] battInfo: [Percentage [%], voltage [V]].
    #[STRING ARRAY] battMaxTemp: Maximum temperature in the battery (in C)
    #[STRING ARRAY] errors: [List of current vehicle errors if any]
    #[STRING ARRAY] errorBits: [List of current errors as a set of 8x32bit integers: Critical0, Critical1, Error0, Error1, Warning0, Warning1, Notice0, Notice1
          #Each integer is a set of bits representing different types of alarms. They are described in the ANT lite User Manual.]
    #[STRING ARRAY] vehicleType: Type of vehicle as defined by the ANT firmware.
    #[STRING ARRAY] vehicleShape: List of (x,y) coordinates of polygon representing the vehicle shape + the safety areas
    #[STRING ARRAY] bodyShape: List of (x,y) coordinates of the polygon describing the physical vehicle shape.
    #[STRING ARRAY] messages:[List of messages coming from ANT]
    #[STRING] trafficAvaiable: Free or empty field: The vehicle is available for mission or not.
    #[STRING] trafficMissID: Reserved mission id.
    #[STRING] trafficInSystem: Inserted or Extracted.
```

```
      #[STRING] trafficCellID: Reservation cell id.
      #[STRING] trafficCurrAction: Current static action:
                #charging, parking: Not moving.
                #enter cell, entering cell granted: Moving.
      #[STRING] trafficStartNode: Start node id.
      #[STRING] trafficTargetNode: Target node id (filled when the vehicle is moving).
      #[STRING] trafficCellPath: Cell path: e:[ 1 7 ] a:[ 1 7 ] (filled when the vehicle is moving).
      #[STRING] trafficTimeDest: Time to destination.
                #0.0: The vehicle is not moving.
      #[STRING] trafficCurrNode: Current node id.
                #0: The vehicle is moving
      #[STRING] vehicleState: String to indicate the current vehicle state: charging, parking, running a mission, paused, asleep, etc.
      #[STRING] vehicleBlocked: true: Missions are blocked.
              #false: New missions are accepted.
      #[STRING] lockUUID: Unique ID of the application which is already connected to the vehicle.
      #[STRING] lockApp:Name of application (antlab or antserver) which is already connected to the vehicle.
      #[STRING] lockPC: Name of the PC which has an application already connected to the vehicle.
      #[STRING] lockUser: Name of the user which has an application already connected to the vehicle.
      #[BOOL] antServerPause: true: The server is paused.
                  #false: Otherwise.

    try:
      if len(self.sessiontoken)<=0:
        self.Login()
    except:
      self.Login()

    api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/vehicles?&sessiontoken=" + self.sessiontoken
    response = requests.get(api_url)
    jsonfile = response.json()
    print(jsonfile)

    self.retcode = (jsonfile["retcode"])
    if self.retcode == 0:
      payloadJSON = (jsonfile["payload"])
      vehiclesJSON = (jsonfile["payload"]["vehicles"])
      try:
        antServerPause = (jsonfile["payload"]["antServerPause"])
      except:
        antServerPause = None
      Vehicle = []
      for vehiJSON in vehiclesJSON:
        Vehicle.append(VehicleObj(vehiJSON))
    else:
        system.perspective.openPopup('1', 'Error', params = {'textLabel':'retcode = ' + self.retcode + '!'},showCloseIcon = False)

    return Vehicle,antServerPause

  def GetAllVehiclesTable(self,path="[default]JSON/GetAllVehicle/AllVehicles"):

    #INPUT
      #[STRING] path: path of the tag where the function writes the return value

    #OUTPUT
      #THE SAME OF "GETALLVEHICLES()"

    Vehicles,antServerPause = self.GetAllVehicles()

    # First create a list that contains the headers
    headers = ["name", "isloaded", "missionid", "payload", "operatingstate", "timestamp", "path", "nameAct", "argsAct",
    "sourceidAct", "sourcetypeAct", "coord", "course", "map",
    "group", "currentnode", "missProgTDest", "missProgTArr",
    "missProgTElaps", "missProgPerc", "missInfoNode",
    "missInfoTargNode", "missInfoFinalNode", "connectionOk",
    "battInfoPerc","battInfoVolt", "bbattMaxTemp", "errors",
    "errorBitsC0", "errorBitsC1", "errorBitsE0", "errorBitsE1",
    "errorBitsW0", "errorBitsW1", "errorBitsN0", "errorBitsN1",
    "vehicleType", "vehicleShape", "bodyShape", "messages",
    "trafficAvaiable", "trafficMissID", "trafficInSystem",
    "trafficCellID", "trafficCurrAction", "trafficStartNode",
    "trafficTargetNode", "trafficCellPath", "trafficTimeDest",
    "trafficCurrNode", "vehicleState", "vehicleBlocked",
    "lockUUID", "lockApp", "lockPC", "lockUser","antServerPause"]
    # Then create an empty list, this will house our data.
    data = []

    for vehicle in Vehicles:

      # Then add each row to the list. Note that each row is also a list object.
      data.append([vehicle.name, vehicle.isloaded, \
   vehicle.missionid, vehicle.payload, vehicle.operatingstate, vehicle.timestamp, vehicle.path, vehicle.nameAct, \
   vehicle.argsAct, vehicle.sourceidAct, vehicle.sourcetypeAct, str(vehicle.coord), vehicle.course, vehicle.map,vehicle.group, vehicle.currentnode, vehicle.missProgTDest, \
   vehicle.missProgTArr, vehicle.missProgTElaps, \
   vehicle.missProgPerc, vehicle.missInfoNode, \
   vehicle.missInfoTargNode, vehicle.missInfoFinalNode, \
   vehicle.connectionOk, vehicle.battInfoPerc, \
   vehicle.battInfoVolt, vehicle.battMaxTemp, vehicle.errors, \
   vehicle.errorBitsC0, vehicle.errorBitsC1, vehicle.errorBitsE0, vehicle.errorBitsE1, vehicle.errorBitsW0,vehicle.errorBitsW1, vehicle.errorBitsN0, vehicle.errorBitsN1,vehicle.vehicleType, vehicle.vehicleShape, vehicle.bodyShape, vehicle.messages, vehicle.trafficAvaiable, vehicle.trafficMissID, \
   vehicle.trafficInSystem, vehicle.trafficCellID, \
   vehicle.trafficCurrAction, vehicle.trafficStartNode, \
   vehicle.trafficTargetNode, vehicle.trafficCellPath, \
   vehicle.trafficTimeDest, vehicle.trafficCurrNode, \
   vehicle.vehicleState, vehicle.vehicleBlocked,vehicle.lockUUID, vehicle.lockApp, vehicle.lockPC, vehicle.lockUser, antServerPause])

    # Finally, both the headers and data lists are used in the function to create a Dataset object
    TableVehicles = system.dataset.toDataSet(headers, data)

    TagPath = [path]
    TagValue = [TableVehicles]
    system.tag.writeBlocking(TagPath,TagValue)
```

```python
def GetOneVehicle(self,vehicleName):

    #INPUT
      #[INT] AutoWrite:
         #1 : Update the missions table of the interface
         #0 : Return dataset
      #[STRING] vehicleName: name of the vehicle

    #OUTPUT
      #THE SAME OF "GETALLVEHICLES()"

    try:
      if len(self.sessiontoken)<=0:
        self.Login()
    except:
      self.Login()

    api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/vehicles/" + str(vehicleName) + "/info?&sessiontoken=" + self.sessiontoken
    #api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/vehicles/AGV1?&sessiontoken=" + self.sessiontoken
    response = requests.get(api_url)
    jsonfile = response.json()

    self.retcode = (jsonfile["retcode"])
    if self.retcode == 0:
      payloadJSON = (jsonfile["payload"])
      vehiclesJSON = (jsonfile["payload"]["vehicles"])
      try:
        antServerPause = (jsonfile["payload"]["antServerPause"])
      except:
        antServerPause = None
      Vehicle = VehicleObj(vehiclesJSON[0])
    else:
        system.perspective.openPopup('1', 'Error', params = {'textLabel':'retcode = ' + self.retcode + '!'},showCloseIcon = False)

    return Vehicle,antServerPause




```

```
def GetAllAlarms(self):

  #INPUT

  #OUTPUT
    #[STRING] uuid: Unique id of the alarm
    #[STRING] eventname: Name of the alarm.
    #[STRING] sourceid: Name of the event source
    #[STRING] sourcetype: Type of the event source
    #[INT] state:
      #0: Active.
      #1: Acknowledged.
      #2: Closed.
      #3: Deleted.
    #[INT] eventcount: Number of times the alarm occurred.
    #[TIME] firsteventat: Date of the first occurrence.
    #[TIME] lasteventat: Date of the last occurrence
    #[TIME] clearedat: Date the alarm was cleared by the operator.
    #[TIME] timestamp: Date of the last update

  try:
    if len(self.sessiontoken)<=0:
      self.Login()
  except:
    self.Login()
  api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/alarms?&sessiontoken=" + self.sessiontoken
  response = requests.get(api_url)
  jsonfile = response.json()

  self.retcode = (jsonfile["retcode"])
  if self.retcode == 0:
    alarmsJSON = (jsonfile["payload"]["alarms"])
    Alarms = []
    for alarmJSON in alarmsJSON:
      Alarms.append(AlarmObj(alarmJSON))
  else:
      system.perspective.openPopup('1', 'Error', params = {'textLabel':'retcode = ' + self.retcode + '!'},showCloseIcon = False)

  return Alarms

```

```
def GetAllAlarmsTable(self,path1="[default]JSON/GetAllAlarms/
 AllAlarmsActive",path2="[default]JSON/GetAllAlarms/
 AllAlarmsLogs"):

  #INPUT
    #[STRING] path: path of the tag where the function writes
 the return value

  #OUTPUT
    #THE SAME OF "GETALLALARMS()"

  Alarms = self.GetAllAlarms()

  # First create a list that contains the headers
  headers = ["uuid", "eventname", "sourceid", "sourcetype", "
 state", "eventcount", "firsteventat", "lasteventat", "
 clearedat", "timestamp"]

  # Then create an empty list, this will house our data.
  dataActive = []
  dataLog = []

  for alarm in Alarms:

    # Then add each row to the list. Note that each row is also
  a list object.
    if alarm.state == 0:
      dataActive.append([alarm.uuid, alarm.eventname, \
  alarm.sourceid, alarm.sourcetype, alarm.state, \
  alarm.eventcount, alarm.firsteventat, alarm.lasteventat,
 alarm.clearedat, alarm.timestamp])

    dataLog.append([alarm.uuid, alarm.eventname, alarm.sourceid
 , alarm.sourcetype, alarm.state, alarm.eventcount, \
 alarm.firsteventat, alarm.lasteventat, alarm.clearedat, \
 alarm.timestamp])

  # Finally, both the headers and data lists are used in the
 function to create a Dataset object
  TableAlarmsActive = \
system.dataset.toDataSet(headers, dataActive)
  TableAlarmsLog = system.dataset.toDataSet(headers, dataLog)

  TagPath = [path1]
  TagValue = [TableAlarmsActive]
  system.tag.writeBlocking(TagPath,TagValue)

  TagPath = [path2]
  TagValue = [TableAlarmsLog]
  system.tag.writeBlocking(TagPath,TagValue)

def GetMap(self,levelID):

  #INPUT
    #[INT] levelID: Map id.

  #OUTPUT
    #[STRING] alias: Map name.
    #[STRING] description: Map description.
    #[INT] id: Map id.
    #[layers: Always two layers: one for localization information, the other for navigation.
      #[0]: Localization information
        #defaultstyleid
          #[STRING] lines: localization.
          #[STRING] points: localization.
        #[STRING] desc: Segments and reflectors.
        #lines: Array of localization segments.
          #[INT ARRAY] coord: Localization segments [x1,y1,x2,y2].
        #[STRING] name: localization.
        #points: Array of reflectors
          #[FLOAT ARRAY] coord: Reflector position and covariance [x, y, cov].
      #[1] Navigation information
        #defaultstyleid
          #[STRING] lines: navigation
          #[STRING] points: navigation
        #[STRING] desc: Nodes, links and virtual walls
        #lines: Array of localization segments.
          #[INT ARRAY] coord: Navigation segments [x1,y1,x2,y2].
          #[STRING] styleid: Ivirtual-wall, The navigation segment is a virtual wall.
                    #Empty field, It is a simple navigation node link.
        #[STRING] name: navigation
        #symbols: Array of navigation node position.
          #[FLOAT ARRAY] coord:navigation node position [x,y,0]
          #[STRING] name: Navigation node name
    #[INT] level: Level id.
    #origin
      #[FLOAT ARRAY] offset: [x,y] offset.
      #[FLOAT] orientation: Map orientation.
  #[STRING] group: Level id.
  #[FLOAT ARRAY] offset: [x,y] offset.
  #[FLOAT] orientation: Map orientation.

  try:
    if len(self.sessiontoken)<=0:
      self.Login()
  except:
    self.Login()

  api_url = "http://" + self.ip + ":" + self.port + "/wms/rest/maps/level/" + str(levelID) + "/data?&sessiontoken=" + self.sessiontoken
  response = requests.get(api_url)
  jsonfile = response.json()
  print(jsonfile)

  self.retcode = (jsonfile["retcode"])
  if self.retcode == 0:
    fileJSON = jsonfile["payload"]["data"]

    Map = []
    for data in fileJSON:
      #Map.append(MapObj(data))
      CreateMap = MapDraw.MapRenderer(data, "black", 1, 8, \
  "red", "black")
      pathLayout = "C:\Users\...\Desktop\layout.svg"
      system.file.writeFile(pathLayout, CreateMap.render())
  else:
    system.perspective.openPopup('1', 'Error', params = {'textLabel':'retcode = ' + self.retcode + '!'},showCloseIcon = False)

  return
```

A.2 Ignition Perspective

```

class MapRenderer:
     def __init__(self, json, line_color, line_stroke_width,
    node_diameter, node_color, text_color):
         self.json = json
         self.line_color = line_color
         self.line_stroke_width = line_stroke_width
         self.node_diameter = node_diameter
         self.node_color = node_color
         self.text_color = text_color
         self.layers = self.get_layers()
         self.arrowDist = 10

         self.xmin = 0
         self.xmax = 0
         self.ymin = 0
         self.ymax = 0

         for layer in self.layers:
           for l in layer["lines"]:
               x1 = l["coord"][0]*100
               y1 = -l["coord"][1]*100
               x2 = l["coord"][2]*100
               y2 = -l["coord"][3]*100

               if self.xmin == 0 and self.xmax == 0 \
              and self.ymin == 0 and self.ymax == 0:
                   self.xmin = x1
                   self.xmax = x1
                   self.ymin = y1
                   self.ymax = y1

               if x1 < self.xmin:
                   self.xmin = x1
               elif x1 > self.xmax:
                   self.xmax = x1
               if x2 < self.xmin:
                   self.xmin = x2
               elif x2 > self.xmax:
                   self.xmax = x2

               if y1 < self.ymin:
                   self.ymin = y1
               elif y1 > self.ymax:
                   self.ymax = y1
               if y2 < self.ymin:
                   self.ymin = y2
               elif y2 > self.ymax:
                   self.ymax = y2

         self.xmin = self.xmin - 100
         self.xmax = self.xmax + 100
         self.ymin = self.ymin - 100
         self.ymax = self.ymax + 100
         self.svgObj = svg(self.xmin,self.ymin,self.xmax-self.xmin
    ,self.ymax-self.ymin)

     def get_layers(self):
         layers = self.json["data"]["layers"]
         return layers

     def render(self):

         for l in self.layers[0]["lines"]:

        x1 = l["coord"][0]*100
        y1 = -l["coord"][1]*100
        x2 = l["coord"][2]*100
        y2 = -l["coord"][3]*100

        self.svgObj.add(Line(
        x1 = x1,
        y1 = y1,
        x2 = x2,
        y2 = y2,
        stroke = self.line_color,
        stroke_width = self.line_stroke_width,
        id = "Line"))

         for l in self.layers[1]["lines"]:

        x1 = l["coord"][0]*100
        y1 = -l["coord"][1]*100
        x2 = l["coord"][2]*100
        y2 = -l["coord"][3]*100

```

```
        xmid=(x1+x2)/2
        ymid=(y1+y2)/2
        xdiff=abs(x2-x1)
        ydiff=abs(y2-y1)
        if x2-x1 == 0:
            m = 1

            if y1-y2 > 0:
                angleArrow=math.pi/2
                xc=xmid-self.arrowDist
                yc=ymid
            else:
                angleArrow=3*math.pi/2
                xc=xmid+self.arrowDist
                yc=ymid
        elif y2-y1 == 0:
            m = 0

            if x2-x1>0:
                angleArrow=0
                xc=xmid
                yc=ymid
            else:
                angleArrow=math.pi
                xc=xmid
                yc=ymid-2*self.arrowDist
        else:
         m=(y2-y1)/(x2-x1)
         angleArrow = math.atan(m)
         angle=math.pi/2 - math.atan(m)
         print(m)
         print(math.degrees(angle))

         if angle < math.pi/2:
                   if x2-x1>0:
                       xc=xmid-self.arrowDist*math.cos(angle)
                       if angle > math.pi/4:
                         yc=ymid-0.1*(self.arrowDist)*\
                         math.sin(angle)
                       else:
                         yc=ymid-0.6*(self.arrowDist)*\
                         math.sin(angle)
                   else:
                       xc=xmid+self.arrowDist*math.cos(angle)
```

```
                      yc=ymid-2*(self.arrowDist)*math.sin(angle)
                      angleArrow=angleArrow+math.pi
              elif angle > math.pi/2:
                  if x2-x1>0:
                      xc=xmid-self.arrowDist*math.cos(angle)
                      if angle > 3*math.pi/4:
                        yc=ymid-0.6*(self.arrowDist)*\
                        math.sin(angle)
                      else:
                        yc=ymid-0.1*(self.arrowDist)*\
                        math.sin(angle)
                  else:
                      xc=xmid+self.arrowDist*math.cos(angle)
                      yc=ymid-2*(self.arrowDist)*math.sin(angle)
                      angleArrow=angleArrow+math.pi

       self.svgObj.add(Line(
       x1 = x1,
       y1 = y1,
       x2 = x2,
       y2 = y2,
       stroke = self.line_color,
       stroke_width = self.line_stroke_width,
       id = "Line"))
       self.svgObj.add(Arrow(
       xc = xc,
       yc = yc,
       angle=math.degrees(angleArrow),
       stroke = self.line_color,
       stroke_width = self.line_stroke_width,
       id = "Arrow"))

        for layer in self.layers:
            for n in layer["symbols"]:
                self.svgObj.add(Node(
                x = n["coord"][0]*100,
                y = -n["coord"][1]*100,
                d = self.node_diameter,
                name = n["name"],
                angle = 0,
                fill = self.node_color,
                stroke = self.line_color,
                stroke_width = self.line_stroke_width,
                font_size=13,
                id = "Node: " + str(n["id"])
            ))

            for i,n in enumerate(layer["points"]):
                self.svgObj.add(Node(
                x = n["coord"][0]*100,
                y = -n["coord"][1]*100,
                d = self.node_diameter,
                name = "R" + str(i),
                angle = 0,
                fill = "black",
                stroke = self.line_color,
                stroke_width = self.line_stroke_width,
                font_size=13,
                id = "Reflector"
            ))

        t_v = Vehicle(x=0, y=0, width=60, height=30, angle=0, id=
    "AGV0", name="0",font_size=15,fill="yellow",stroke=self.
    line_color,stroke_width=self.line_stroke_width)
        self.svgObj.add(t_v)

        out = str(self.svgObj)
        return out
```

References

[1] *Skilled Group website*, `https://www.skilledgroup.com/`

[2] *Holonomy in mobile robots, J.A. Batlle, A. Barjau*

[3] *Mobile Robot Kinematics, Roland Siegwart, Margarita Chli, Martin Rufli*

[4] *Rockwell documentation on Ethernet/IP*

[5] `https://simple.wikipedia.org/wiki/Ethernet_switch`

[6] `https://it.wikipedia.org/wiki/Sistema_client/server`

[7] *Ewon documentation*

[8] *The perfect size of the protective field on an industrial autonomous vehicle, from SICK wbsite*

[9] *Huffman Engineering website*

[10] *William Goble, Learn to Trust Safety PLCs*

[11] *A Safe Starting Point for AGV Navigation, SICK website*

[12] *Modbus Gateways, Control Solutions website*

[13] *Firmware definition, TechTarget*

[14] *NAV-LOC Laser positioning system, SICK documentation*

[15] *BlueBotics website*, `https://bluebotics.com`

[16] *BlueBotics ANT navigation documentation*

[17] *BlueBotics AntLite$^+$ manual*

[18] *SICK Laser Scanners manual*

[19] *Allen Bradley vs Siemens PLC, DO supply*

[20] *Profinet explanation, Profinet website*

[21] *BlueBotics AntServer manual*

[22] *Ignition documentation*

[23] *Ignition website*

[24] *BlueBotics documentation, How an Automobile Parts Producer Used Nipper AGVs to Meet its Resource Optimization Goals*

[25] *Api REST definition* `https://www.redhat.com/it/topics/api/what-is-a-rest-api`

[26] `https://restfulapi.net/http-methods/`

[27] `https://www.json.org/json-en.html`

[28] *PROFINET Switch vs Ethernet Switch, PROFINET website*

[29] *Simatic PN/MF coupler documentation*

[30] *The Difference Between PROFINET and PROFIsafe, PROFINET website*

[31] *Counting and Measuring with the Counter Module, Siemens documentation*

[32] `https://quadrasrl.net/nuova-norma-en-iso-3691-4-2020-agv-e-lgv/`

[33] *Safety Siemens documentation*

[34] *Industrial Software SIMATIC Safety: Configuring and Programming*

[35] `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON`

Ringraziamenti

Ringrazio il professor Angelo Cenedese per avermi dato l'opportunità di seguire un progetto complesso in una realtà importante e per l'aiuto fornitomi nella stesura della presente tesi.

Ringrazio il team di Euroimpianti ed in particolare Michele Sanvido per tutto quello che mi ha insegnato durante questo nostro progetto.

E' stato un viaggio molto travagliato, impegnativo, pieno di lacrime, quaderni di appunti, schemi ed esercizi. Non avrei mai raggiunto questo obbiettivo senza le persone che mi sono state vicine in questi anni.

Ringrazio innanzitutto la mia famiglia, grazie a mamma, papà e Nicole per avermi supportato e sopportato. Nulla di tutto ciò sarebbe stato possibile senza di voi.
Ringrazio zia Maria Grazia per l'in bocca al lupo puntuale prima di ogni esame e per le (tante) candele accese affinchè tutto andasse per il meglio.

Ringrazio i miei amici di una vita, Gioele, Mara e Davide perchè nonstante tutto mi sono sempre stati vicini e sanno quanto sono importanti per me.

Ai miei Uni-Friends per le chiacchierate fino a tarda notte, per il sostegno e l'aiuto reciproco che ci siamo dati in tutti questi anni. Ai miei compagni di viaggio dal primo giorno Mattia e Giovanni, insieme siamo sopravvisuti ai traumi dei primi anni. Ai miei compagni pendolari e alle due ore di treno giornaliere ad invadere l'ultima carrozza con i nostri discorsi. In molti saranno felici di non sentirci più.

Ringrazio Christian in modo particolare, per tutti i ”ti chiamo 15 minuti che poi devo studiare” e alla fine stavamo al telefono ore. Sei uno dei regali migliori che UniPd mi abbia dato.
L’università ci ha tolto tanto, ci ha messo alla prova e molte battaglie le ha pure vinte. Ma devo dire che rifarei tutto anche solo per incontrarvi.

Ringrazio il calcio e tutte le compagne con cui ho condiviso lo spogliatoio in questi anni. Con voi ho imparato l’importanza del lavoro di squadra.
In modo particolare ringrazio Sara per tutti i consigli e per tutto quello che abbiamo condiviso l’ultimo anno, il mio supporto lo avrai sempre.

Infine ringrazio me stessa, per la tenacia, i viaggi in pulmino passati a ripassare, a tutte le rinuncie per riuscire a portare a termine il mio obbiettivo dal primo giorno del primo anno, *Tra 5 anni, quando mi laureerò*, non ci credeva quasi nessuno, ma così è stato.