POLITECNICO
MILANO 1863

AUTOMATION AND CONTROL LABORATORY TEAM 15 – WAREHOUSE

PROF. G. CAZZULANI

a.a. 2022/2023

Barone Lorenzo

Bartoli Matteo

Brunacci Brigida

Guida Mario

Contents

1 Introduction

The concept of storage has a rich history dating back to early human civilization. However, the term "warehousing" emerged relatively recently after a long period of evolution. With the rapid advancement of logistics and research, the world now discusses the concept of "automated warehouses," which is the focus of this project.

The prominence of logistic companies has greatly increased the visibility of warehouse automation and its transformative impact on the storage, tracking, and movement of products.

In simple terms, warehouse automation involves the use of specialized equipment, storage and retrieval systems to replace tasks previously performed by unskilled and semi-skilled labour. As the name suggests, it automates repetitive processes that once required human intervention. Through the integration of robotics, automated systems can unload products from trailers or manufacturing lines, as well as store and retrieve items on demand, all without direct involvement from human workers.

Implementing automation for product storage and retrieval offers numerous advantages. Manufacturers of automated warehouses emphasize these benefits, which contribute to improved efficiency and cost savings. Let's delve into the advantages of implementing automation in storage systems:

One major advantage is the ability to achieve high-density storage with high selectivity. The control algorithm employed in automated systems enables the storage of a large number of products while maintaining the flexibility to access individual items easily.

Reduced operational costs are another key benefit. Automation optimizes various aspects of operations, addressing one of the primary goals of all industries. By streamlining processes and minimizing manual labour, businesses can achieve cost savings in their overall operations.

Automation also leads to faster load and unload times, enhancing overall throughput. With automated storage systems, forklift operators no longer need to spend time searching for available pallet spaces. Instead, the storage system takes care of the placement, allowing for quicker and more efficient loading and unloading processes.

Precise storage is ensured through electronic control. The alignment precision of items in automated storage systems is significantly increased, eliminating errors and minimizing the risk of misplaced or damaged products.

One notable advantage is the creation of a more economical storage space. Automated systems can make better use of vertical space, reducing the required footprint compared to traditional horizontal storage. This space-saving capability results in cost savings. Additionally, in temperature-controlled environments like cool rooms and freezers, automation helps lower refrigeration costs. Moreover, labour costs are reduced as workers no longer need to physically move to the package location to pick up orders. Instead, they can send orders and continue working while waiting for the products to be retrieved.

Safety is also enhanced in automated storage systems. The inclusion of safety features significantly reduces the risk of warehouse incidents and minimizes damage to the structure, ensuring a safer working environment for employees.

Ultimately, the implementation of automated storage systems provides a greater return on investment. With cost-effective solutions that offer high selectivity and efficient storage of more pallets, businesses can achieve improved productivity and profitability. The use of high-quality materials ensures the durability and longevity of the automated storage system, making it a reliable and long-lasting solution for any warehouse or storage facility.

In summary, the advantages of implementing automation in storage and retrieval processes include high-density storage, reduced operational costs, faster load and unload times, more accurate storage, economical use of space, safer working environments, and a greater return on investment. These benefits contribute to enhanced efficiency, cost savings, and improved overall performance in warehouse operations.

2 Project description

2.1 Physical Model

The goal of this project is to control a warehouse composed of a 3x3 matrix of cells and a carrier moving along a matrix plane (X, Z axis), which can be pushed in or out with respect to a cell (Y axis) to store/retrieve a package.

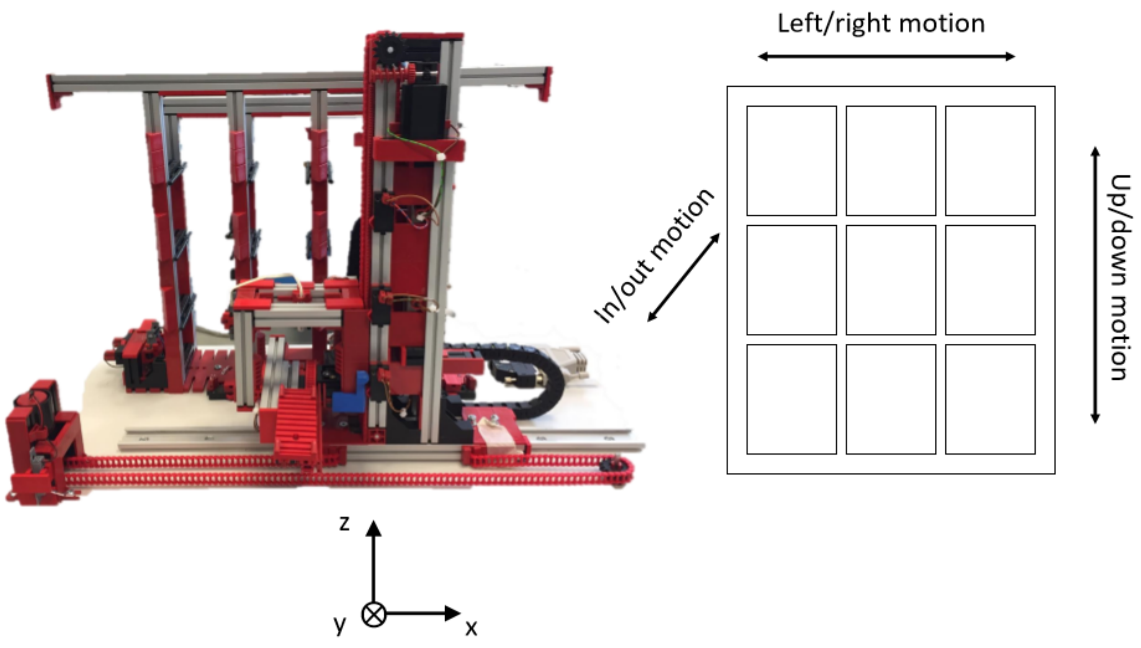


Figure 1: front view (left) and schematic representation (right) of the warehouse

The system has on/off motors which allow movement in all directions. The current position is detected through sensors, 6 for position along Z axis(7, 8, 9, 10, 11, 12) and 3 for both X (1, 2, 3) and Y positions (4, 5, 6) as depicted in Figure 3: sensors' position along x (left), along y (centre) and along z (right)

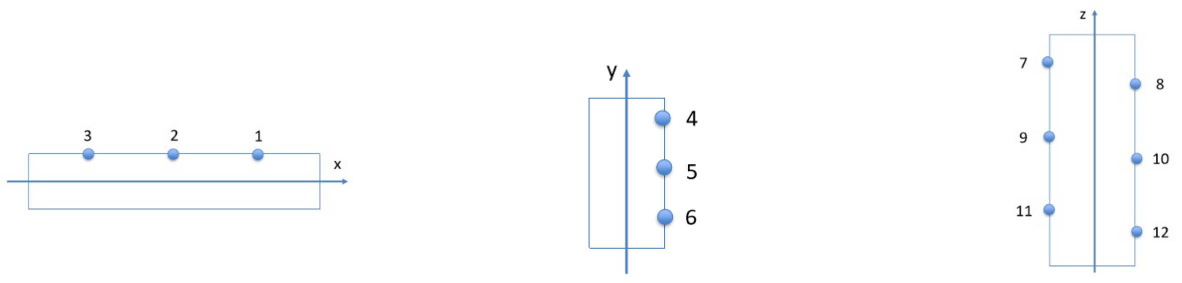


Figure 2: sensors' position along x (left), along y (centre) and along z (right)

2.2 Project Objective

The objective of this project is to automate the operations of the warehouse.

In this subchapter, we will focus on identifying the key objectives taking into account principal benefits of an automated warehouse as in Chapter 1.

Firstly, we will conduct research and assessment to anticipate and address potential challenges that may arise throughout the project. The representation of the reasoning followed is illustrated in Figure 3.

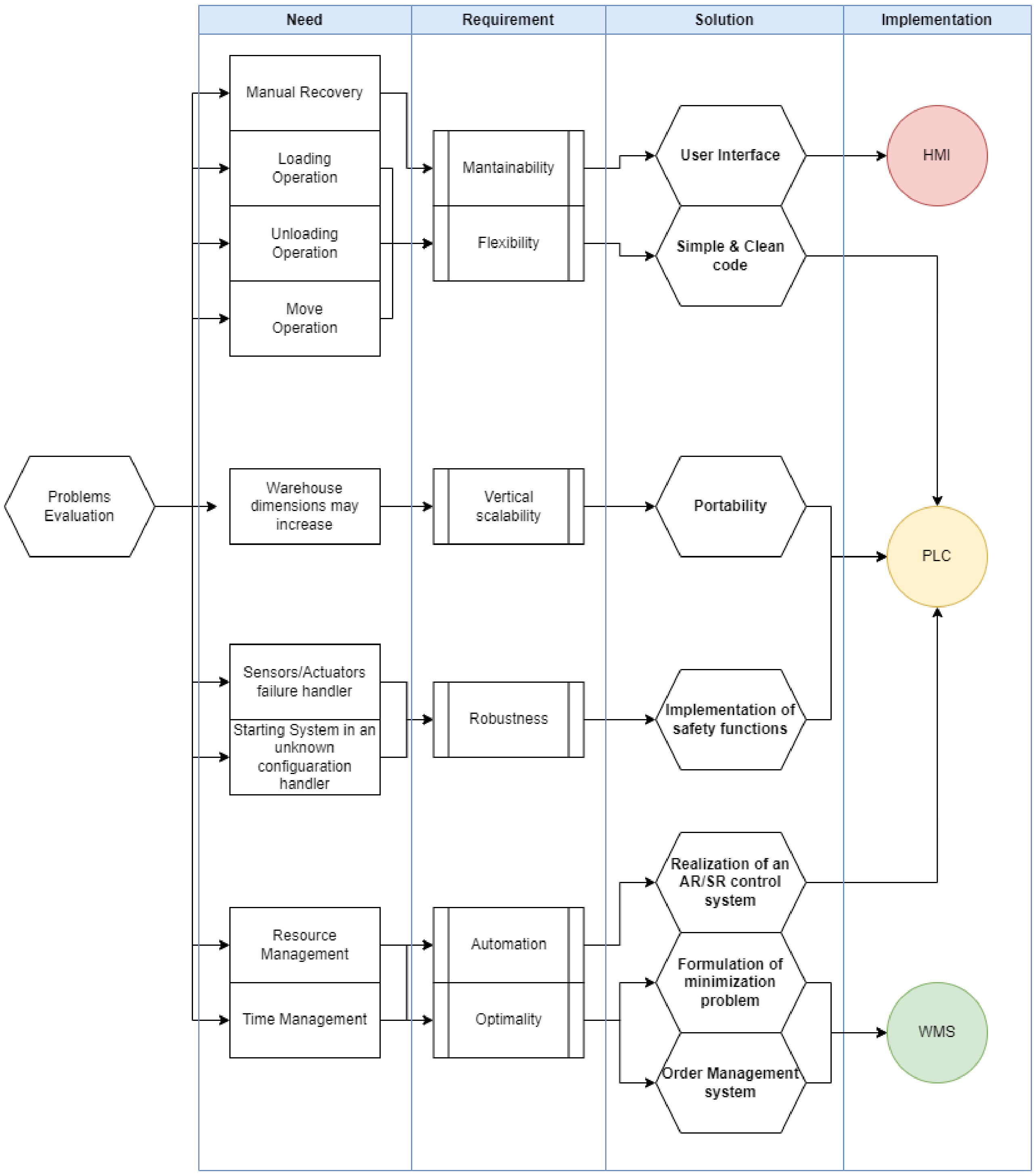


Figure 3: Schematic of the reasoning flow followed to develop the solution

Upon initial analysis, the operations' commands handled by the Warehouse Management System (WMS), such as storing and retrieving boxes, are defined to establish the foundation for determining the best strategic approach. Once the objectives and main tasks of the system are identified, a high-level block diagram is developed, as depicted in Figure 4, outlining the overall system architecture.

To effectively manage the sequence of commands, a high-level control strategy is also essential. As a result, an external sequence optimizer will be developed to accommodate various optimization criteria. This optimizer will ensure the proper sequencing of tasks, taking into account factors such as order priority, inventory availability, and warehouse capacity.

By integrating the WMS, the project aims to streamline box storage and retrieval processes, optimize resource allocation, and improve overall operational efficiency. The developed block diagram and control strategy will provide a structured framework for orchestrating warehouse operations and achieving the desired optimization outcomes.

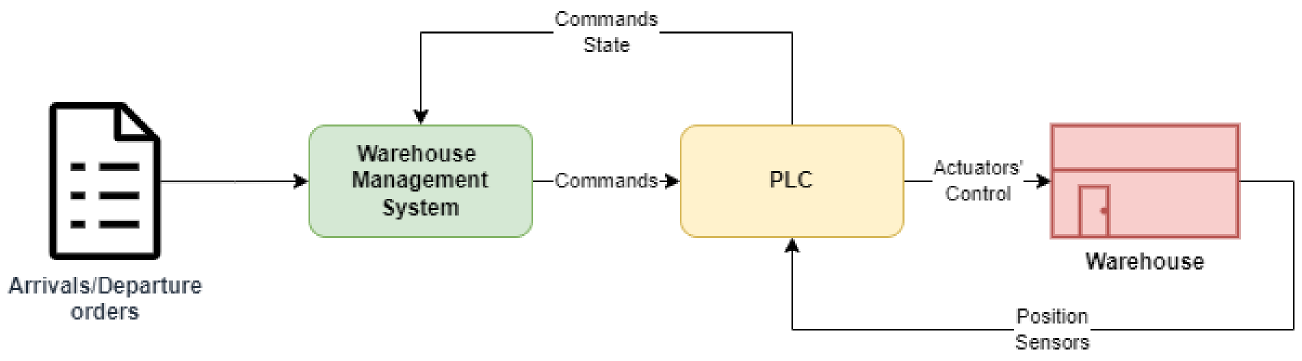


Figure 4: High-level bock diagram of the system

2.3 Hardware

This project utilizes the PM554-T PLC manufactured by ABB for controlling the storage warehouse operations. To interact with the environment, the PLC leverages an I/O interface, specifically the DC532 module from ABB. The DC532 module consists of 32 terminals, with 16 terminals designated as digital inputs (I0-I15). The remaining 16 terminals can flexibly function as either digital inputs or outputs (C16-C31) based on the specific requirements of the system.

Figure 5: I/O interface of the PLC

For what concerns the mapping between physical inputs (sensors signals and buttons) and PLC variables the following conventions are adopted:

No.	Variables	Mapping
1	XposRight	I0
2	XposMid	I1
3	XposLeft	I2
4	YposIn	I3
5	YposMid	I4
6	YposOut	I5
7	Zpos3Above	I6
8	Zpos3Below	I7
9	Zpos2Above	I8
10	Zpos2Below	I9
11	Zpos1Above	I10
12	Zpos1Below	I11
13	FeederOccupied	I12
14	Handkey1	I13
15	Handkey2	I14

Table 1: mapping of physical inputs and PLC variables

Where the numbers of the first column are associated with the conventions of Figures 6-7. The following figures show where the sensors are located in the real system. Moreover, the sensors with their associated number are highlighted.

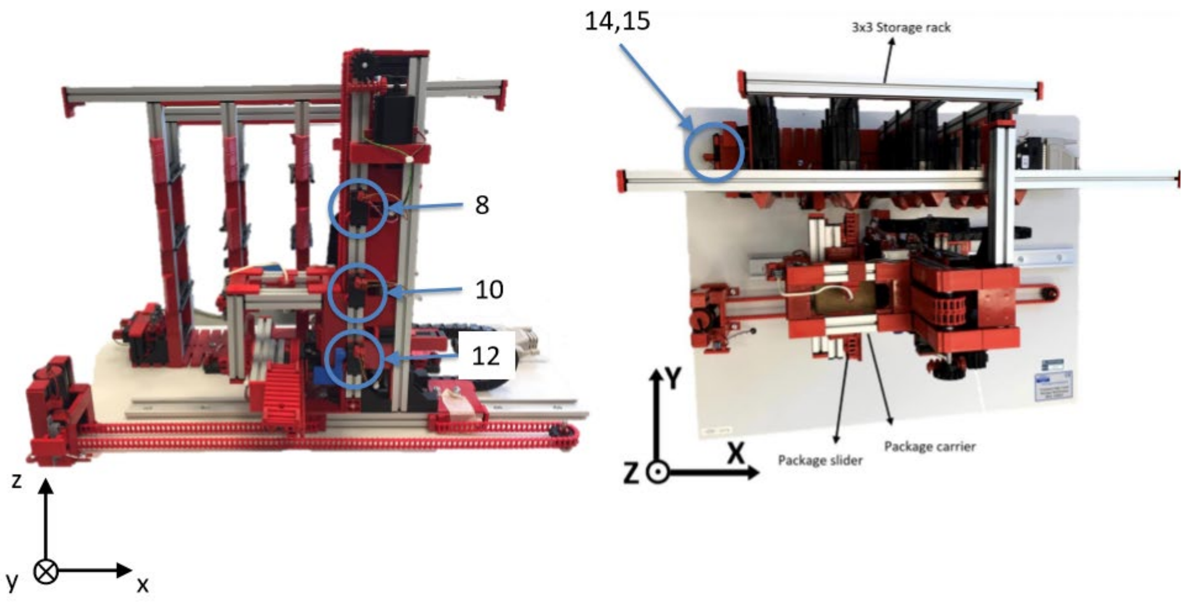


Figure 6: front view (left image) and view from above (right image) of the warehouse with highlighted sensors' positions

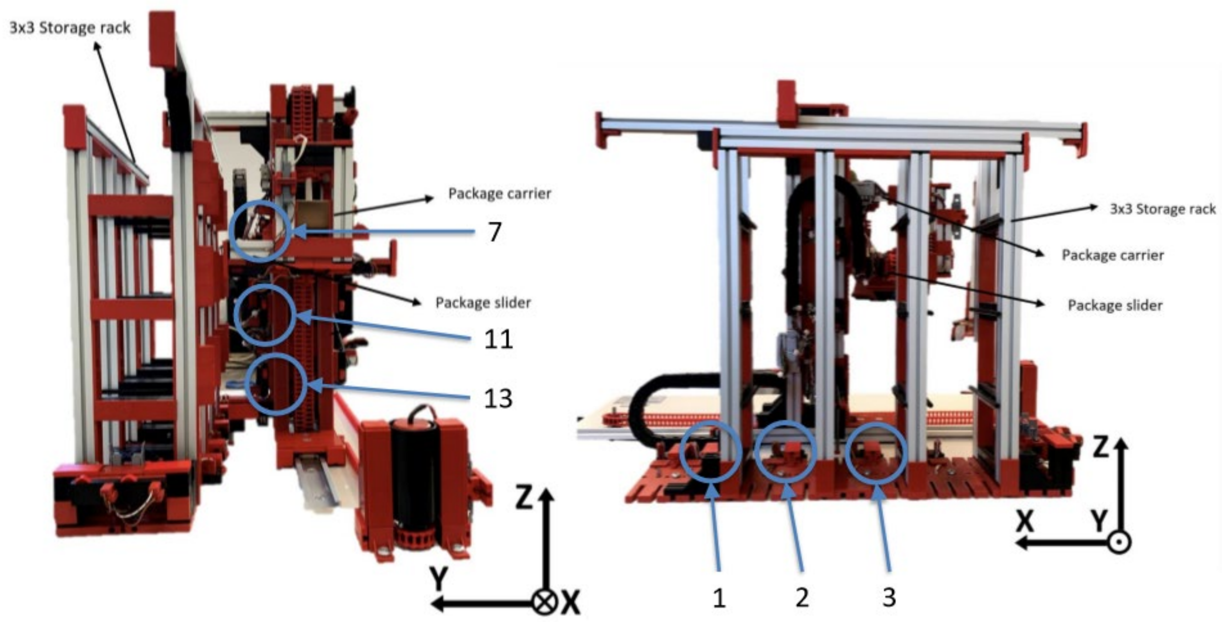


Figure 7: side view (left image) and back view (right image) of the warehouse with highlighted sensors' positions

Handkey1 and Handkey2 are buttons which can be used for security functions or for other purposes. The application in this project will be better explained in the following chapters.

It is important to note that all the switches are normally closed this means that the associated variable is normally TRUE and becomes FALSE when the switch is pressed (in the case of a sensor for example when it detects the presence of the feeder in front of it, its value passes from TRUE to FALSE).

Let's analyse now the outputs mapping. The convention is the following:

Variables	Description of signal	Mapping
XRight	Activate X motor to the right	C24
XLeft	Activate X motor to the left	C25
YOut	Activate Y motor out	C27
Yin	Activate Y motor in	C26
ZUp	Activate Z motor up	C28
ZDown	Activate Z motor down	C29
Led1	Switch Led 1 on	C30
Led2	Switch Led 2 on	C31

Table 2: system outputs mapping

It is clear, from the table, which motor or which led is controlled by the output of the PLC. Led1 and Led2 can be used to notify dangerous situations or the need for maintenance, their usage in this project will be explained in the next chapters.

2.4 Software

The overall software architecture involved in this project is shown in 8

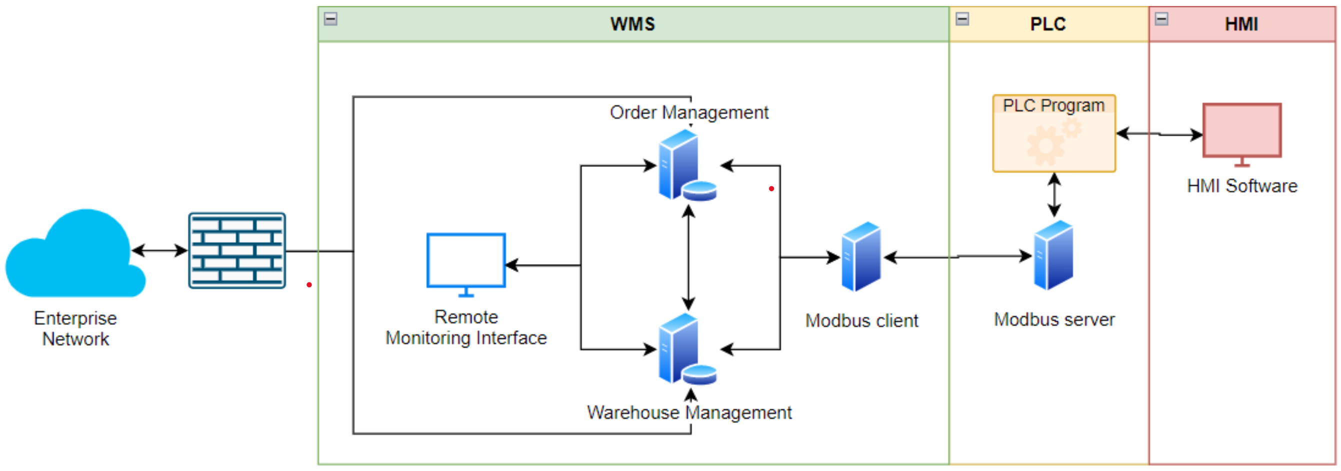


Figure 8: Software Architecture

The software architecture of the automated warehouse system consists of three core components.

- HMI (Human-Machine Interface): The HMI component is connected to the PLC (Programmable Logic Controller). It serves as the interface through which operators interact with the system. The HMI allows operators to monitor and control various aspects of the warehouse operations, such as checking system status, initiating commands, and receiving real-time feedback. It provides a user-friendly interface for efficient interaction with the automated system.

- PLC (Programmable Logic Controller): The PLC component acts as the central control unit of the warehouse system. It collects and executes the PLC program, which contains the logic and instructions for controlling the warehouse operations. Additionally, the PLC serves as a Modbus server, enabling communication with other devices in the system. It facilitates data exchange and coordination between the HMI, WMS, and other connected devices.

- WMS (Warehouse Management System): The WMS component serves as the brain of the automated warehouse system. It handles various critical functions necessary for efficient warehouse operations. The main components of the WMS include:

   a) Order Management: The WMS manages the processing and fulfilment of orders. It receives incoming orders from external sources, such as customers or an ERP system, and handles order allocation, prioritization, and tracking. The WMS ensures that orders are processed accurately and on time to meet customer demands.
   b) Warehouse Management: The WMS oversees the overall management of the warehouse. It optimizes inventory placement, storage, and retrieval processes to maximize space utilization and minimize operational inefficiencies. The WMS tracks inventory levels, manages stock movements, and provides real-time visibility into the status and location of items within the warehouse.
   c) Modbus Client: The WMS functions as a Modbus client, allowing it to communicate with the PLC's Modbus server. This communication enables a seamless exchange of data and instructions between the WMS and the PLC. The WMS sends commands and instructions to the PLC to control the automated equipment. It receives feedback and status updates from the PLC, enabling the WMS to monitor and coordinate warehouse operations effectively.

Together, these three core components work in harmony to ensure smooth and efficient warehouse operations. The HMI provides the interface for human interaction, the PLC serves as the control unit and communication gateway, and the WMS acts as the intelligent system managing orders, optimizing warehouse processes, and facilitating communication with the PLC.

3 Modelling

While developing a solution to a problem, the first thing to do is to analyse in detail the system and extract from this analysis a rigorous model of the system.
In this chapter, a formal model for the entire system is rigorously defined. First, a detailed description of the PLC states is given using deterministic Finite State Automata formalism, used also to model *move to target* function. In the end, a dissertation on main operations is provided together.

3.1 PLC Model

The PLC state modelling is a crucial aspect of designing the control system. It provides a systematic and structured representation of the system's behaviour, allowing for a comprehensive understanding of state transitions and control logic. In this chapter, we will focus on the PLC state modelization for the automated warehouse and explore the main features and advantages of the model.

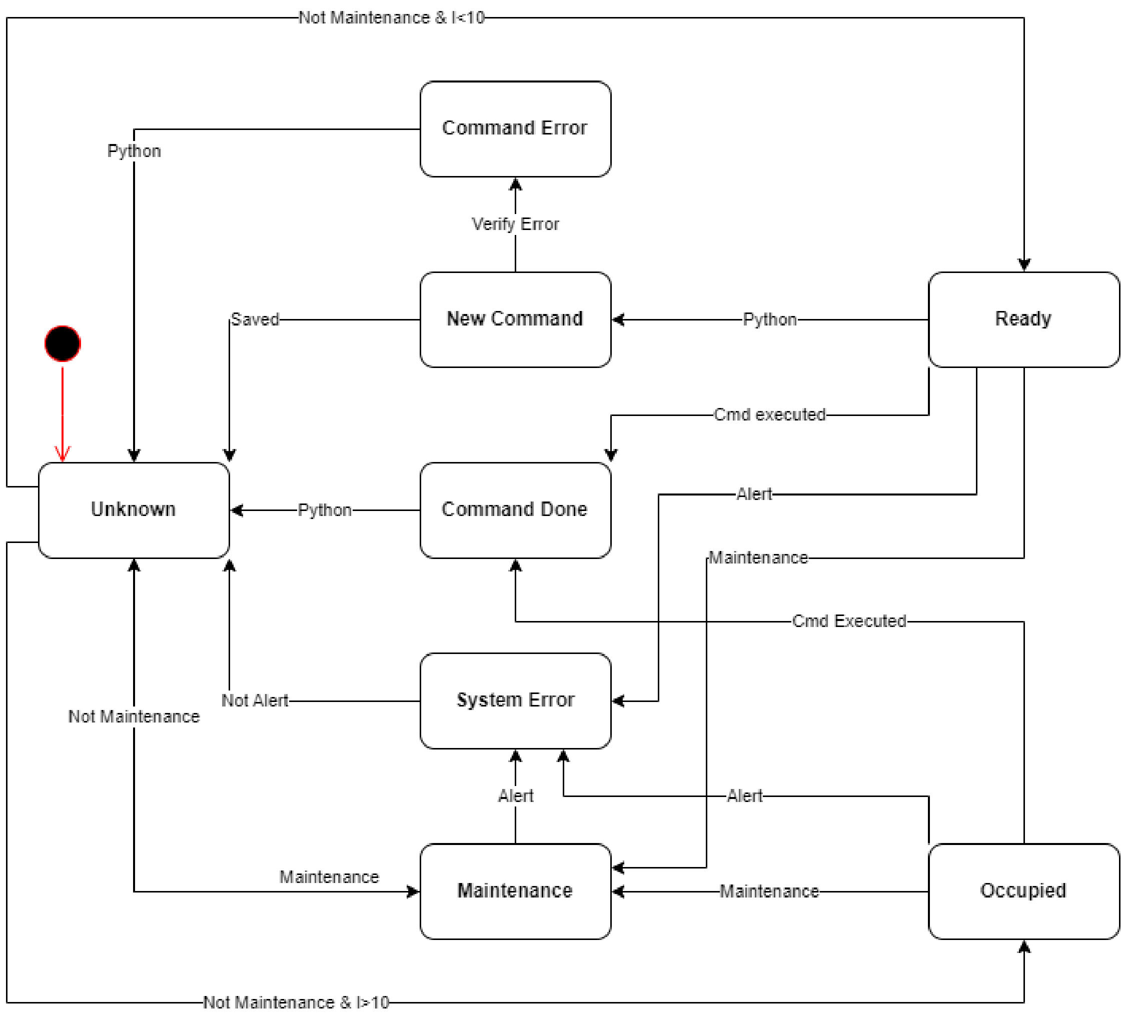


Figure 9: FS automata describing PLC states

The PLC includes eight states that govern the behavior of the warehouse system: Unknown, Ready, Occupied, Maintenance, System Error, New Command, Command Done, and Command Error. Let's examine each state and its characteristics:

1. **Unknown State**:
   The Unknown state represents the initial state of the warehouse system when there is no prior information about its current state. In this state, the system evaluates certain conditions to determine the next state. If the maintenance mode is activated, indicating that maintenance operations are required, the system transitions to the Maintenance state. Alternatively, if the command count exceeds or equals 10, indicating that the warehouse is occupied, the system transitions to the Occupied state. If none of these conditions are met, the system transitions to the Ready state, indicating its readiness to receive new commands.

2. **Ready State**:
   The Ready state signifies that the warehouse system is prepared and ready to receive new commands and execute operations. In this state, the system evaluates various conditions to determine the next course of action. If an alert is triggered, indicating a system error, the system transitions to the System Error state to handle the error condition. Similarly, if the maintenance mode is activated, the system transitions to the Maintenance state to perform necessary maintenance actions. Upon completion of a command execution (cmd_run_done), the system transitions to the Command Done state. If the command count exceeds or equals 10, suggesting that the warehouse is currently occupied, the system transitions to the Occupied state. In case a command submission is requested (submit), the system executes the cmd_submit() function while remaining in the Ready state. Additionally, if the start signal is received and there are commands in the queue (cmd_count > 0), the system executes the cmd_run() function, maintaining the Ready state. If none of these conditions are met, the system shuts down the motors and remains in the Ready state until further input.

3. **Occupied State**:
   The Occupied state indicates that the warehouse is currently engaged in ongoing operations. When in this state, the system monitors certain conditions to determine the subsequent state. If an alert is triggered, indicating a system error, the system transitions to the System Error state to handle the error condition appropriately. Similarly, if the maintenance mode is activated, the system transitions to the Maintenance state to carry out necessary maintenance tasks. Upon completion of a command execution (cmd_run_done), the system transitions to the Command Done state. If the command count falls below 10, implying that the warehouse is no longer occupied, the system transitions back to the Ready state, indicating its readiness for new commands. In case a command submission is requested (submit), the system executes the cmd_submit() function while remaining in the Occupied state. Furthermore, if the start signal is received and there are commands in the queue (cmd_count > 0), the system executes the cmd_run() function, maintaining the Occupied state. If none of these conditions are met, the system shuts down the motors and remains in the Occupied state until further input.

4. **Maintenance State**:
   The Maintenance state represents the mode in which the warehouse system undergoes maintenance operations. When in this state, the system evaluates specific conditions to determine

subsequent actions. If an alert is triggered, indicating a system error, the system transitions to the System Error state for appropriate error handling. Similarly, if the maintenance mode is deactivated, the system transitions back to the Unknown state, indicating a return to regular operation. Upon completion of a command execution (cmd_run_done), the system transitions to the Command Done state. If none of these conditions is met, the system remains in the Maintenance state, performing maintenance actions while ensuring the safety and reliability of the warehouse system.

5. **System Error State**:
   The System Error state indicates the occurrence of a system error and triggers appropriate actions such as shutting down motors and handling alerts. It helps in maintaining system integrity, preventing potential damages, and enabling the timely resolution of errors. In this state, the system evaluates conditions to determine the subsequent state. If the alert is no longer active, indicating the resolution of the error, the system transitions back to the Unknown state. If the maintenance mode is activated, the system transitions to the Maintenance state to address the error condition and perform necessary actions.

6. **New Command State**:
   The New Command state handles the processing of new commands received by the system. It performs actions such as command deletion, validation, addition, and error handling. This state ensures the accurate execution of commands and proper command queue management. If a command with the action "DELETE_INDEX" is received, the system executes the cmd_delete() function and transitions to the Command Done state if the delete ID is found. Otherwise, it transitions to the Command Error state. For other command actions, the system checks for validity and adds the command if both the action and targets are valid. If the validation is successful, the system transitions back to the Unknown state. Otherwise, it transitions to the Command Error state.

7. **Command Done State**:
   The Command Done state is responsible for handling the completion of commands. It initiates actions such as shutting down motors, updating command status, and preparing for the execution of the next command. This state ensures the systematic and efficient execution of commands. If a command execution is completed (cmd_run_done), the system executes the cmd_done() function, writes the executed command, and resets the cmd_run_done flag. If the modbus_check_manual condition is true, indicating manual intervention, the system transitions back to the Unknown state.

8. **Command Error State**:
   The Command Error state represents a state in which an error occurs during command processing. This state is triggered when a command received by the system cannot be executed successfully or encounters an error during validation or execution. It allows for appropriate error handling and can trigger actions such as updating command status as an error, notifying operators, and taking corrective measures. In this state, the system shuts down motors and evaluates conditions to determine subsequent actions. If the cmd_run_done flag is true, indicating the completion of command execution, the system transitions to the Command Done state, updating the command

status as done. If the modbus_check_manual condition is true, the system transitions back to the Unknown state.

In the PLC state modelization can be found several key features and advantages of the automated warehouse as in Chapter 1:

- Control and Coordination: The state modelization allows for effective control and coordination of various warehouse operations by defining the states, transitions, and actions to be taken in each state.
- Error Handling: The system can quickly respond to system errors and alerts by transitioning to the appropriate error-handling state, ensuring prompt resolution and minimizing downtime. The System Error and Command Error states specifically focus on handling errors encountered during system operation and command processing.
- Flexibility and Adaptability: The model allows for easy modification and addition of new states and transitions as per evolving requirements, ensuring the system remains flexible and adaptable to future changes.
- Efficient Resource Utilization: By modelling states such as Occupied and Ready, the system can optimize resource utilization, ensuring efficient workflow management and minimizing idle time.
- Maintenance Support: The model includes a dedicated Maintenance state, enabling the system to prioritize and perform necessary movements.

3.2 Operations' Model

Each operation can initiate from any known position and can be performed in any order, provided they are logically coherent with each other.

Ultimately, all operations conclude with an empty feeder and motors turned off.

The "move2target" (m2T) function is a crucial component in the PLC code that is used as the basis for the three operations: load, unload and move. Once provided the target on the upper level, it handles the actual movement of the shuttle to such target position within the automated warehouse system. Let's discuss the modelling and states of the "move2target" function.

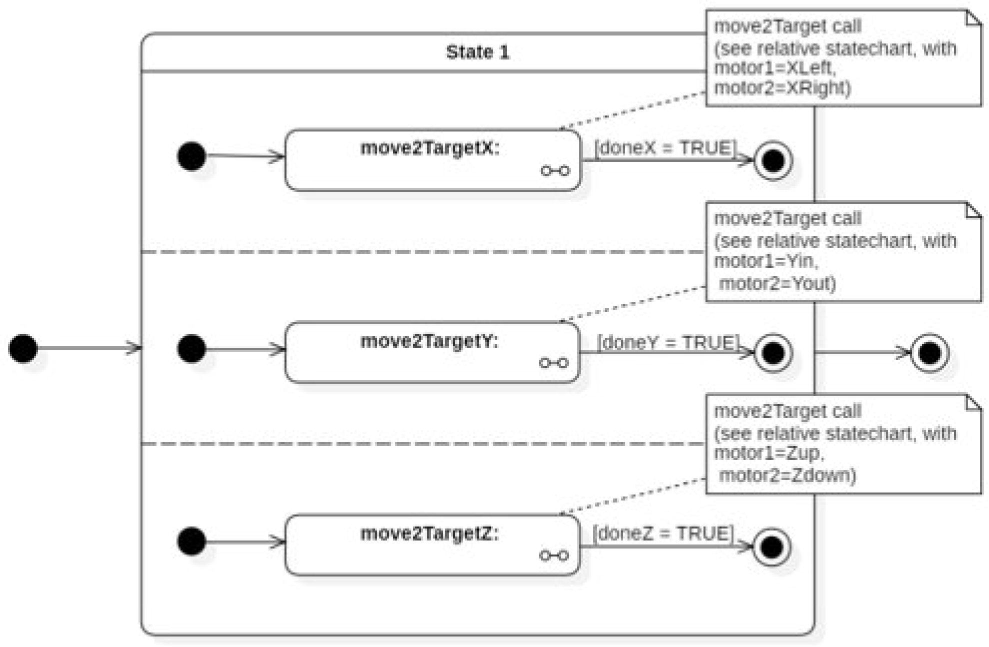

State 1
move2TargetX:
[doneX = TRUE]
move2Target call (see relative statechart, with motor1=XLeft, motor2=XRight)
move2TargetY:
[doneY = TRUE]
move2Target call (see relative statechart, with motor1=Yin, motor2=Yout)
move2TargetZ:
[doneZ = TRUE]
move2Target call (see relative statechart, with motor1=Zup, motor2=Zdown)

Figure 10: FS automata describing move2Target

The "move2target" function can be modelled as a state machine with the following states:

- Initialization:
  The function is initialized with the target coordinates (x, y, z) received as input.
- Moving to Target:
  In this state, the function calculates the necessary movements and the control signals to send on X,Y and Z motors in order to guide the shuttle towards the target position accordingly.
  The function continuously checks the current position of the shuttle and compares it with the target position to determine the progress of the movement.
- Arrival at Target:
  Once the shuttle reaches the target position, the function sets a flag named m2T.done, so that the upper-level function can effectively detect the requested task is completed.
  Additional actions or checks can be performed at this point, depending on the specific requirements of the operation calling the "move2target" function (read later for further discussion on this).

The states and transitions described above demonstrate a simplified model of the "move2target" function. The actual implementation involves more complex logic and subroutines to ensure precise positioning and movement control and to prevent the effects of improper sensors' and motors' behaviour. The variables pos_current, pos_last and last_move are used to store and track the shuttle's motion based on the logic shown in the following state chart, and to cover scenarios of hardware malfunction: if the motor on one of the axes surpasses the intended destination by remaining operational beyond the required duration, the code detects a null current position (as no sensors would be in a pressed state) but at the same time registers the passage through the target position with the variable pos_last, so it performs a correction by reversing the ongoing movement, saved in last_move; if the sensor corresponding to the target position gives no feedback on the transit of the shuttle, the same correction will be done once one of the following position sensors is triggered, and in case the error is due to a definitive fault of the sensor rather than a glitch, namely the passage of the shuttle on the target position keeps being not read, the procedure is killed after a timer expiration leading to a system error.

In the context of the load, unload and move operations, the "move2target" function is utilized to handle the movement of the shuttle between various positions within the automated warehouse system. By incorporating this function into the state-based modelling of the operations, the PLC code can effectively control and coordinate the movements required to perform the relative command following safety principles; for instance, when performing a load command, the coordinates given as input to the m2t function are the ones related to the current position but with y coordinate set to mid position, so that in the subsequent calls of the m2t function for moving the shuttle on x and z axes the shuttle doesn't hit the warehouse structure.

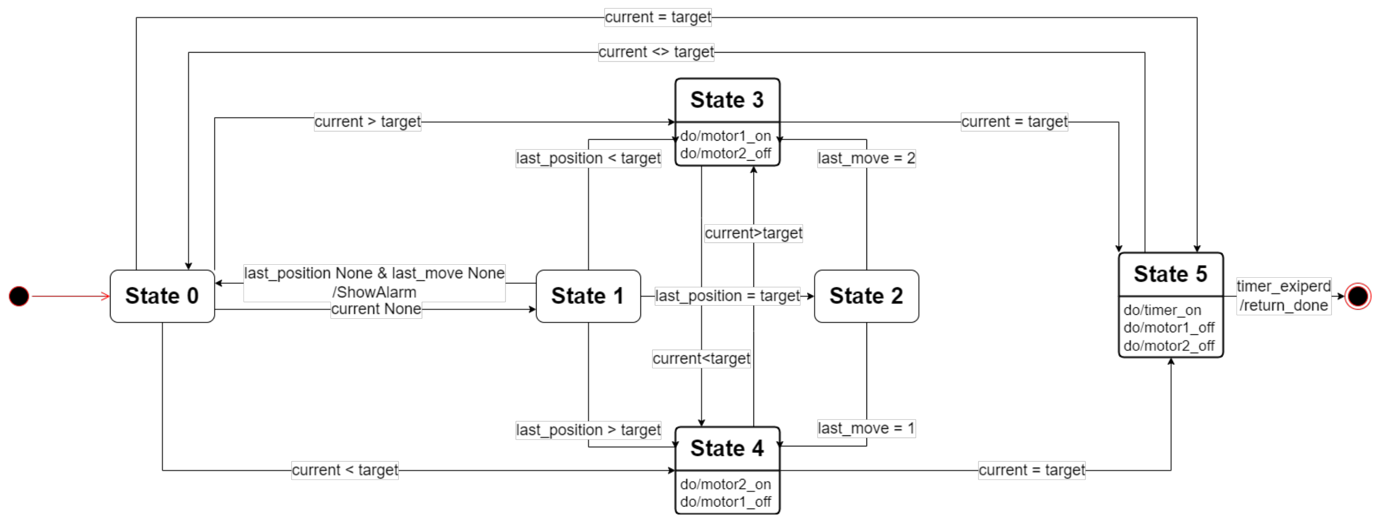


Figure 11: FS automata describing MovetoTarget states

The previously defined basic operation fulfils all the required functionalities of the system that lead to the identification of the following main operations: Load, Unload and Move.

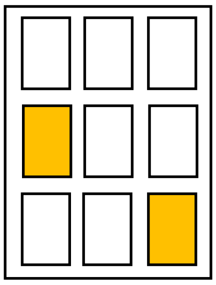
Each operation is implemented as a state machine too, where different states represent specific steps and actions. By using a state-based approach, the code ensures precise positioning, checks for presence, and includes error-handling states. More precisely it contains conditional statements to handle properly several possible undesired events, like temporary or definitive sensor faults (as it is clear, not all cases can be covered as they would need, given their nature, a human intervention) but also to reach a higher optimality level, like avoiding unnecessary movements.

An example can be seen performing an unload-load command combination, a case in which the program prevents the shuttle to move in the default mid position after the first command, as not needed in the scope.

In addition, the code's modular structure and conditional branching allow for flexibility, maintenance, and easy modification if needed.

Let's dive into the technical details of each operation:

- LOAD



The load operation starts with the state value set to 0. As the operation progresses, the state value is incremented, indicating the transition to the next state.

The steps followed by this operation are positioning the shuttle at the current mid-position, moving the shuttle to the home position, moving the home shuttle out, waiting for the item placing signalled by the user confirmation (HandKey2 or CONTAINER CHECK button in the HMI). As mentioned, safety controls are performed during the procedure: container_check is a function which reads the value of the variable related to the feeder sensor to check the presence of the parcel in the shuttle; it is executed after the user signalization in the home position, and in case it returns a negative result the last two steps of the command are repeated to allow the user placing the container again.

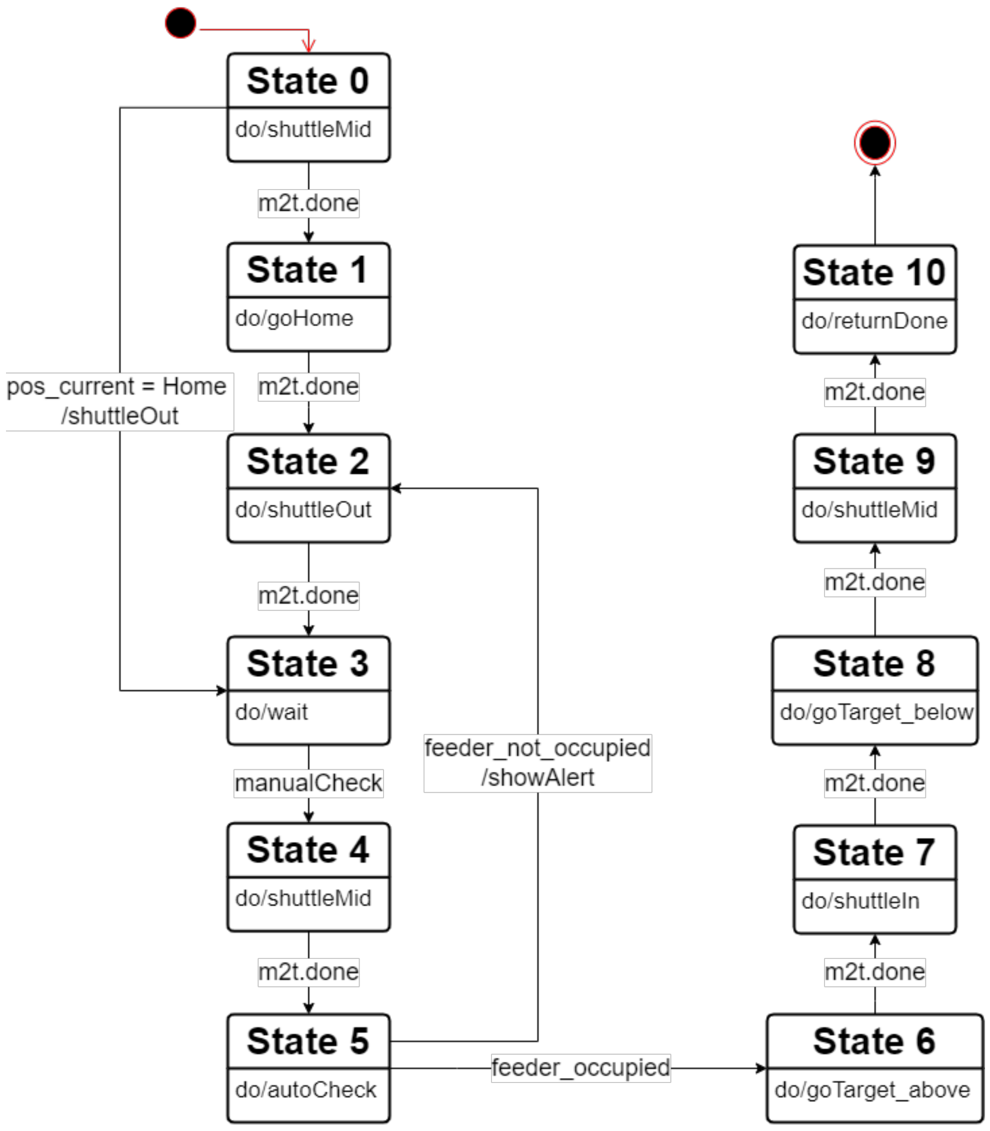
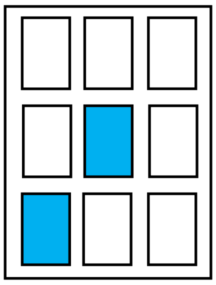


Figure 12: FS automata describing Load Command

- UNLOAD



Similarly to the load operation, the unload one is implemented as a state machine with different states, it relies on iterative calls of the function move2target and it exploits, in the same way, the function container_check to ensure that the shuttle is empty before taking the container and that the parcel is successfully unloaded (one possible scenario is having a slot registered as occupied but being physically empty); the negative result, in this case, leads to an error displayed on the HMI screen so to inform correctively the user.
With the aim of brevity, the specific steps won't be described, but they are visible in the following state chart.

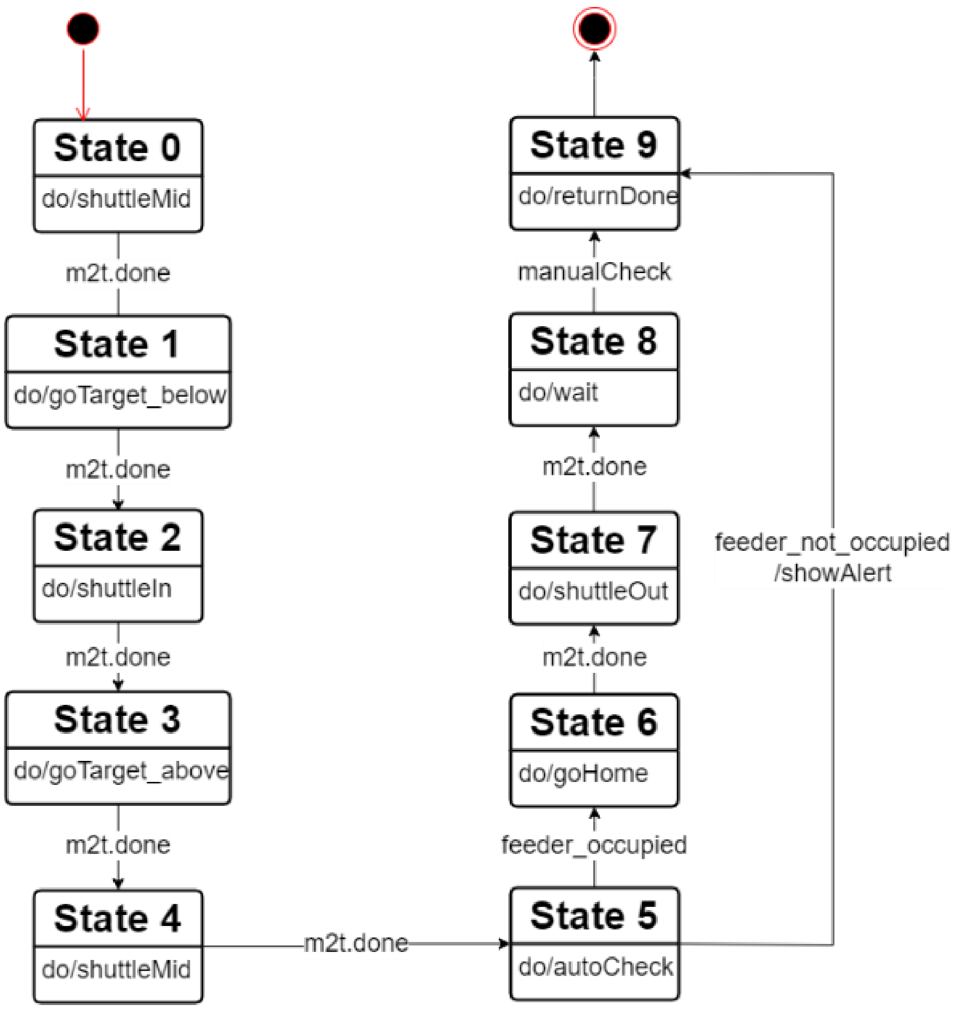
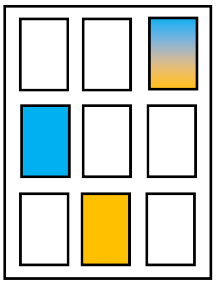


Figure 13: FS automata describing Unload command

- MOVE

The move command is implemented as an unload-load combination, devoided of the intermediate steps for going through the home position; through such command is possible indeed to take an already placed container from its position and to place it into another slot.
Accordingly, it executes the relative checks on the container presence.



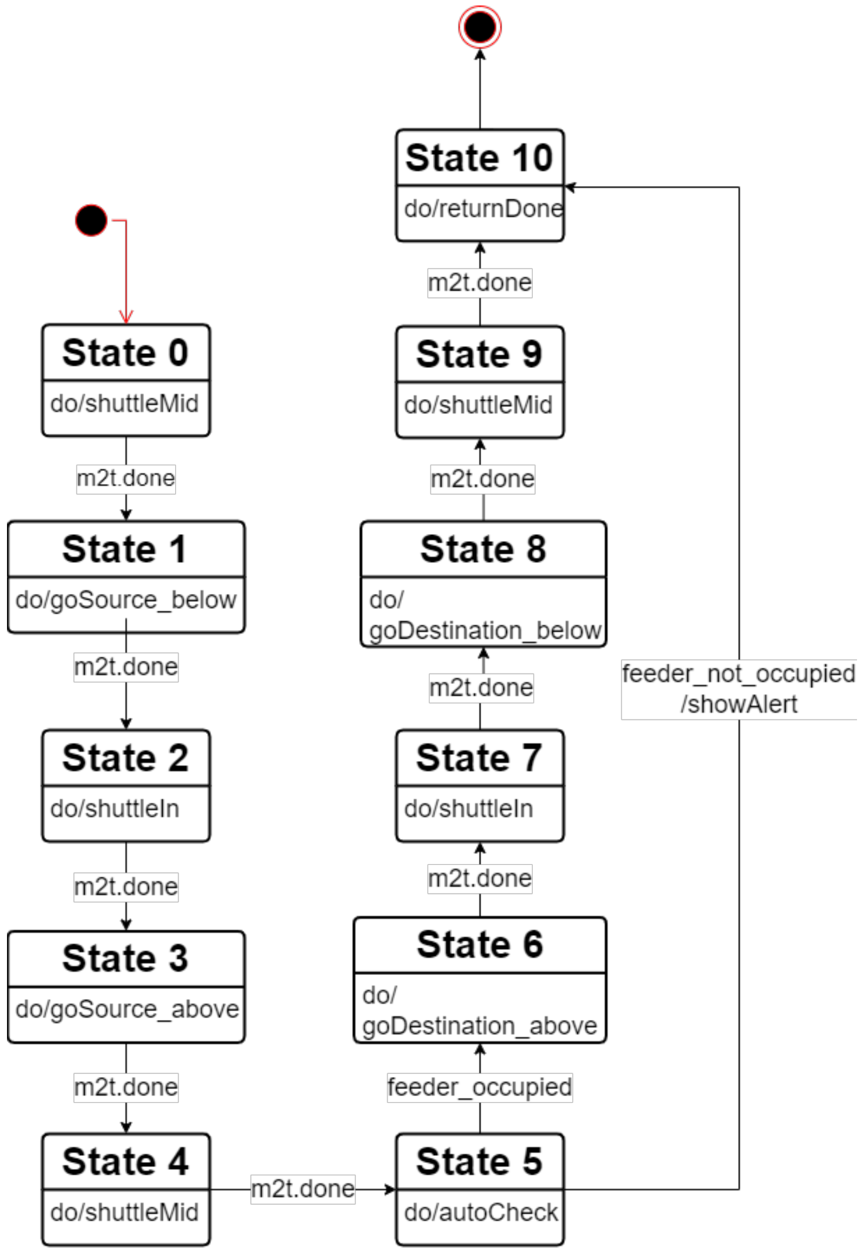


Figure 14: FS automata describing Move command

4 Control

In this chapter the implementation of the control logic in the software environment is discussed.

The overall program is divided into 3 sub-programs, each of them working on different level as shown in Figure :

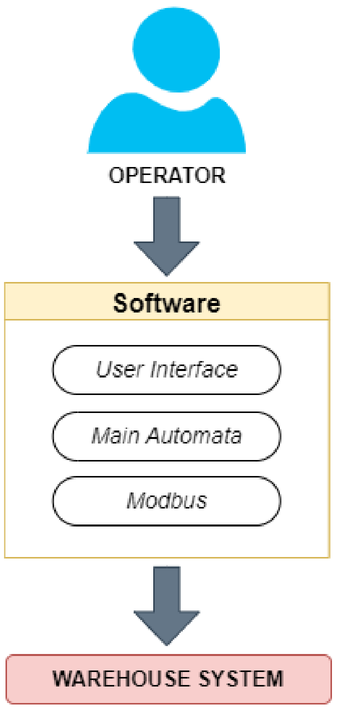


Figure 15: Control program hierarchy

The Warehouse management system interacts with the plc trough registers and modbus communication as we'll better see in chapter 5.

The middle level program, *Main automata,* aims to manage the basic operations in order to make the system perform correctly the tasks given by the user.

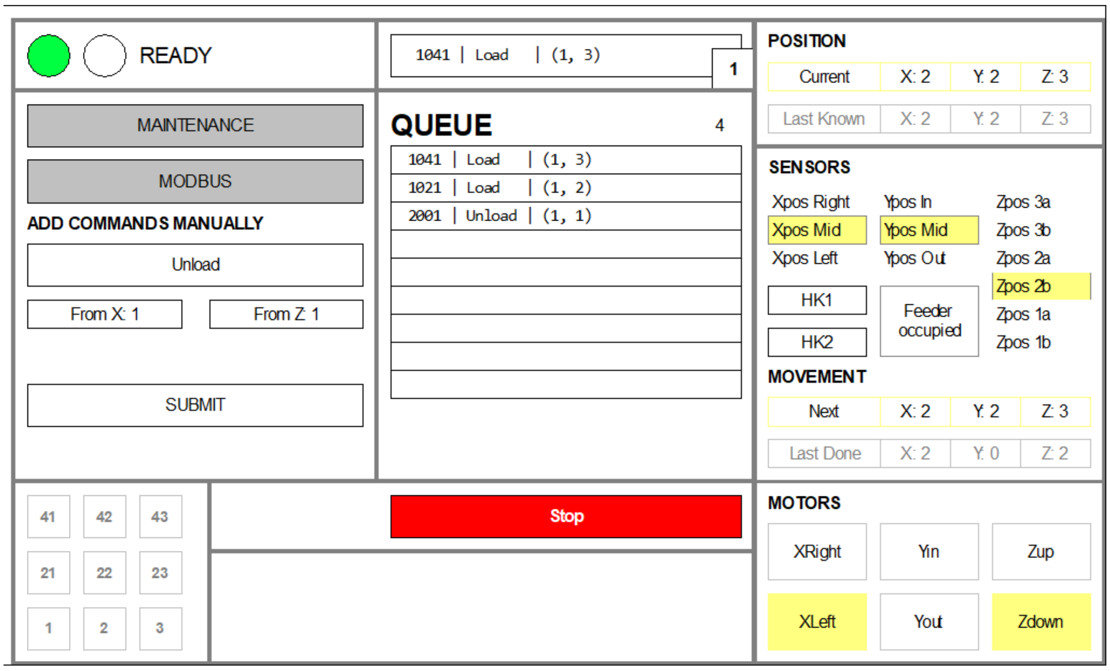
Finally, the high-level program, *UserInterface,* is the one that directly interface the software with the user, through a visual HMI.

Each of these programs work together with various safety function, described in Chapter Safety Functions4.2 that ensure the warehouse works safely and correctly.

4.1 User Interface

The *UserInterface* program aims to directly interface the user with the software. Through this program, the operator can add one or more tasks to the list of the operations that will be performed by the warehouse, also in runtime.

The user can set a command through a visual HMI:


READY

```
1041 | Load   | (1, 3)   1
```

QUEUE 4

```
1041 | Load   | (1, 3)
1021 | Load   | (1, 2)
2001 | Unload | (1, 1)
```

POSITION

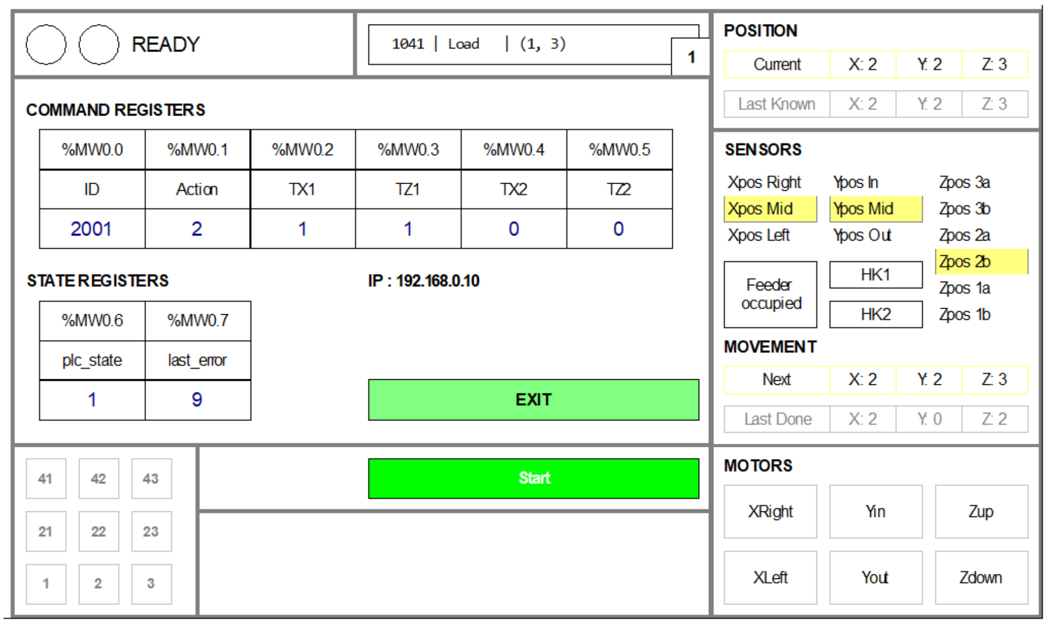
Current	X: 2	Y: 2	Z: 3
Last Known	X: 2	Y: 2	Z: 3

MOVEMENT

Next	X: 2	Y: 2	Z: 3
Last Done	X: 2	Y: 0	Z: 2

Figure 16: Human Machine Interface

On the left section is placed a tab reporting the current system state (in the picture above corresponding to READY), the buttons for switching to the maintenance and to the Modbus panels and the buttons for choosing and setting a command to add to the list; on the central section, on the top, is reported the first command of the list, with a number showing in which state of the command the machine is at the moment, and under that the command queue composed of the commands received either by the user through the HMI and the warehouse system (python program), followed by the start button which activate or deactivate the autorun of the commands; on the right part of the HMI is present the info related to the shuttle motion tracking, the sensors' and motors' state and the buttons corresponding to the hand keys; the bottom part is reserved for reporting the list of the errors occurred.


READY

```
1041 | Load   | (1, 3)   1
```

COMMAND REGISTERS

%MW0.0	%MW0.1	%MW0.2	%MW0.3	%MW0.4	%MW0.5
ID	Action	TX1	TZ1	TX2	TZ2
2001	2	1	1	0	0

STATE REGISTERS

%MW0.6	%MW0.7
plc_state	last_error
1	9

IP : 192.168.0.10

POSITION

Current	X: 2	Y: 2	Z: 3
Last Known	X: 2	Y: 2	Z: 3

MOVEMENT

Next	X: 2	Y: 2	Z: 3
Last Done	X: 2	Y: 0	Z: 2

Figure 17 :HMI modbus

The image above shows the Modbus panel, where is possible for the user to read the values currently written in the Modbus memories. Moreover, a button called MANUAL MODBUS CHECK for resetting such memories to the UNKNOWN state is placed.

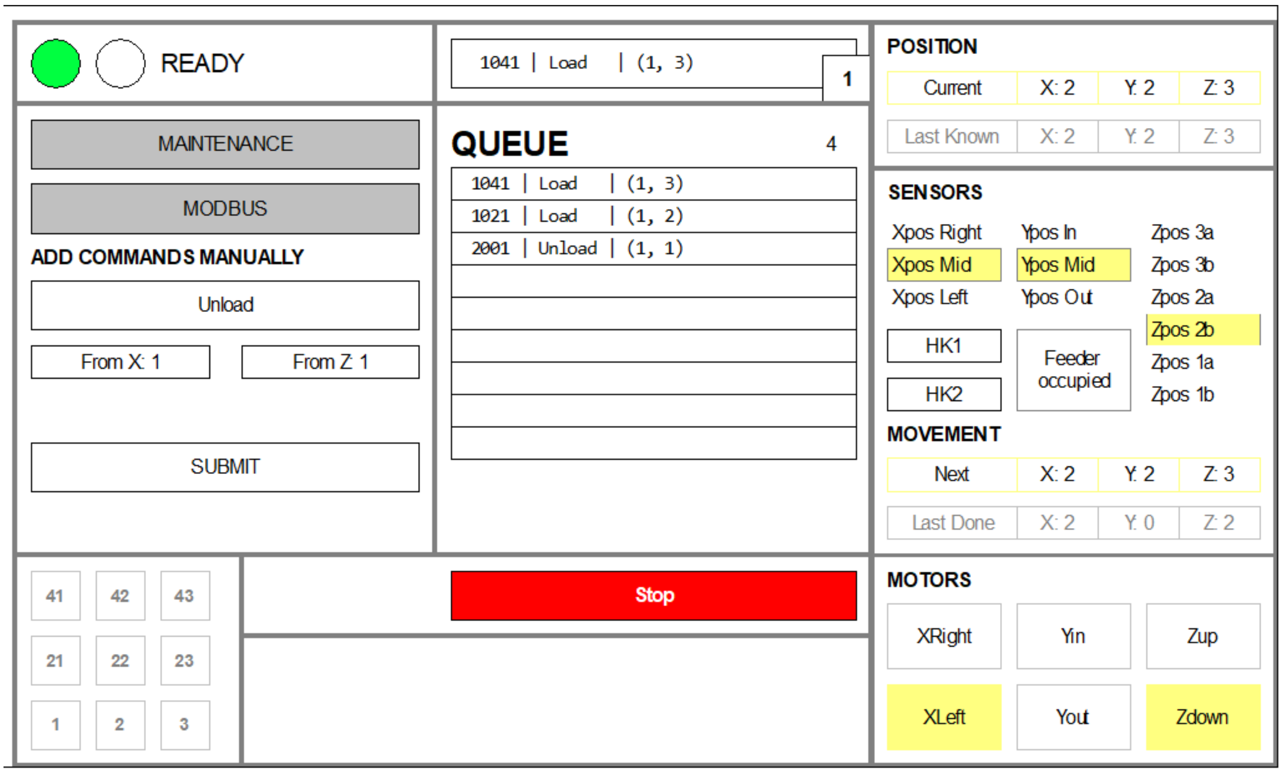
The type of the new command that the operator wants to add can be chosen by pressing the buttons inside which the name of the command is written. Only one type can be specified at a time.

Then, for each type of command except for the go home, a cell has to be specified.

Once all the needed parameters corresponding to a specified command have been correctly chosen, the command can be written by pressing the SUBMIT button.

A limit on the maximum number of commands has to be decided, equal to the dimension of the coms[] vector. Due to the dimension of the warehouse a maximum of 10 commands is set since it is sufficient to load or unload all the cells.

To delete a command with the specified ID, press the button DELETE.



```
1041 | Load   | (1, 3)
```

```
1041 | Load   | (1, 3)
1021 | Load   | (1, 2)
2001 | Unload | (1, 1)
```

Figure 18: Human Machine Interface

4.2 Safety Functions

To ensure the correct and safe operation of the automated warehouse system, it is crucial to implement safety measures and alarms. By implementing the alert_manager that calls the alert_add_msg function block to add alert messages to the system, the warehouse system can effectively detect and address potential hazards or malfunctions promptly. This subchapter discusses the safety measures implemented in the system.

By implementing these safety measures and alarms, the automated warehouse system enhances operational safety, reduces the risk of accidents, and allows for prompt identification and resolution of potential issues. Operators and maintenance personnel can rely on the system's alerts to take appropriate actions, ensuring the correct and safe functioning of the warehouse operations.

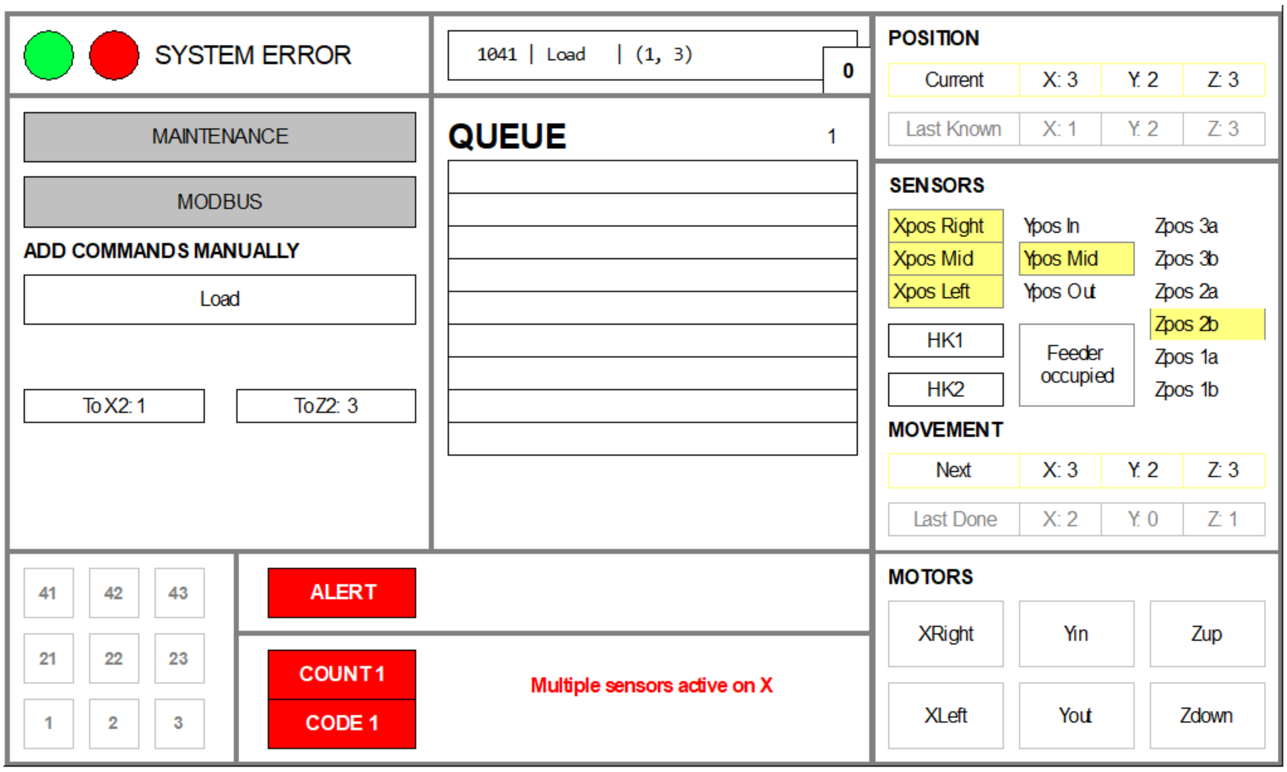
4.2.1 Alert Manager

This program aims to recognize all the possible physical malfunctions of the warehouse system. If a malfunction is detected, an alarm is generated for the user and the system is blocked.

Here's a breakdown of the key features in the code and possible malfunctioning of the system:

- Timeout Display: The timeout_display function is used to set a timeout for displaying alerts. This ensures that alerts are displayed for a specific duration before they are cleared.
- Alert Initialization: The code includes an initialization section where the alert_display_fix flag is checked. If the timeout for displaying alerts has expired (timeout_display.Q is true), the alert_display_fix flag is set to false, allowing new alerts to be shown. Additionally, if alert_display_fix is not set, indicating that alerts can be displayed, the alert, alert_count, alert_code, and alert_string variables are reset, preparing the system for new alerts.
- Multiple Sensors Active in the Same Axis (Alert codes 1, 2, 3): The code includes a loop that iterates over each axis and checks the number of active sensors for each one. If more than one sensor is active, indicating a potential issue, an alert message is generated using the CONCAT function and added to the system using the alert_add function. The corresponding alert code is set based on the axis number.
- No Position Reference (Alert codes 4, 5, 6): Another loop is used to check the presence of position references on each axis. If the current position (pos_current[i]) is zero and either the last position (pos_last[i]) or the last move (last_move[i]) is also zero, indicating a missing reference, an alert message is generated and added to the system.
- No Sensor Trigger for a Long Time While Motor is Active (Alert codes 7, 8, 9): The code includes timeout functions (timeout_XnoMove, timeout_YnoMove, timeout_ZnoMove) that monitor the duration of sensor inactivity while the corresponding motor is active. If the timeout expires (timeout_XnoMove.Q, timeout_YnoMove.Q, timeout_ZnoMove.Q are true), indicating a potential motor failure, an alert is added to the system using the alert_add function. The motor is also shut down to prevent further issues.
- Command Errors (Alert codes 11, 12, 13): The code includes checks for command errors such as invalid targets, actions, or IDs. If any of these errors are detected (valid_targets, valid_action, delete_id_found flags are false), the corresponding alert message is added to the system. These alerts are marked as fixable (alert_dipslay_fix := TRUE) to indicate that they can be addressed.
- Writing the Last Alert Code to Modbus Register: if there is an active alert, the last alert code (alert_code) is written to a specific Modbus register (%MW0.7). This allows external systems to access and process the alert information.

Furthermore, the system can be blocked manually by pressing the button **HandKey1.** This is useful to the operator to stop the system if he detects any unusual behaviour or for safety reasons.



```
1041 | Load     | (1, 3)
0
```

POSITION

Current	X: 3	Y: 2	Z: 3
Last Known	X: 1	Y: 2	Z: 3

MOVEMENT

Next	X: 3	Y: 2	Z: 3
Last Done	X: 2	Y: 0	Z: 1

Multiple sensors active on X

Figure 19: HMI in system error

When the system is blocked automatically **led1** and **led2** is turned on blinking to give a visual feedback to the operator.

Once the system is blocked, it can be restarted by pressing the *START* button in the *Alarms* visualizer once the fault is resolved. This happens usually after some hardware fixes to the defective components has been performed.

4.2.2 Maintenance Mode

Maintenance Mode is a mode thought to be used in emergency cases when the feeder must be moved directly by the operator.

When "Maintenance" button is pressed in the "Joystick" HMI, the system enters in the *Maintenance Mode* state. Being in this state it is possible to activate all the motors "manually" or to correct in the SET SLOT STATES section the state of the warehouse slots locally stored. As soon as this mode is activated, all motors will be turned off.

During the time the warehouse is in *Maintenance* state, the motors will only be able to move controlled by the "Joystick" HMI.

This mode is intended as a security function, in case there is a need to control the system "manually"(for example when there is the need of understanding what sensor or actuator is broken), but still it has to be considered as a critical mode, since it is possible to implement behaviours harmful for the system.

Direction buttons in the "joystick" HMI are disabled if the system is not in *MaintenanceMode* state.

The go home button, which initiates the procedure for moving in an automatic way the shuttle to the home position, is not enabled if system errors are being signaled.

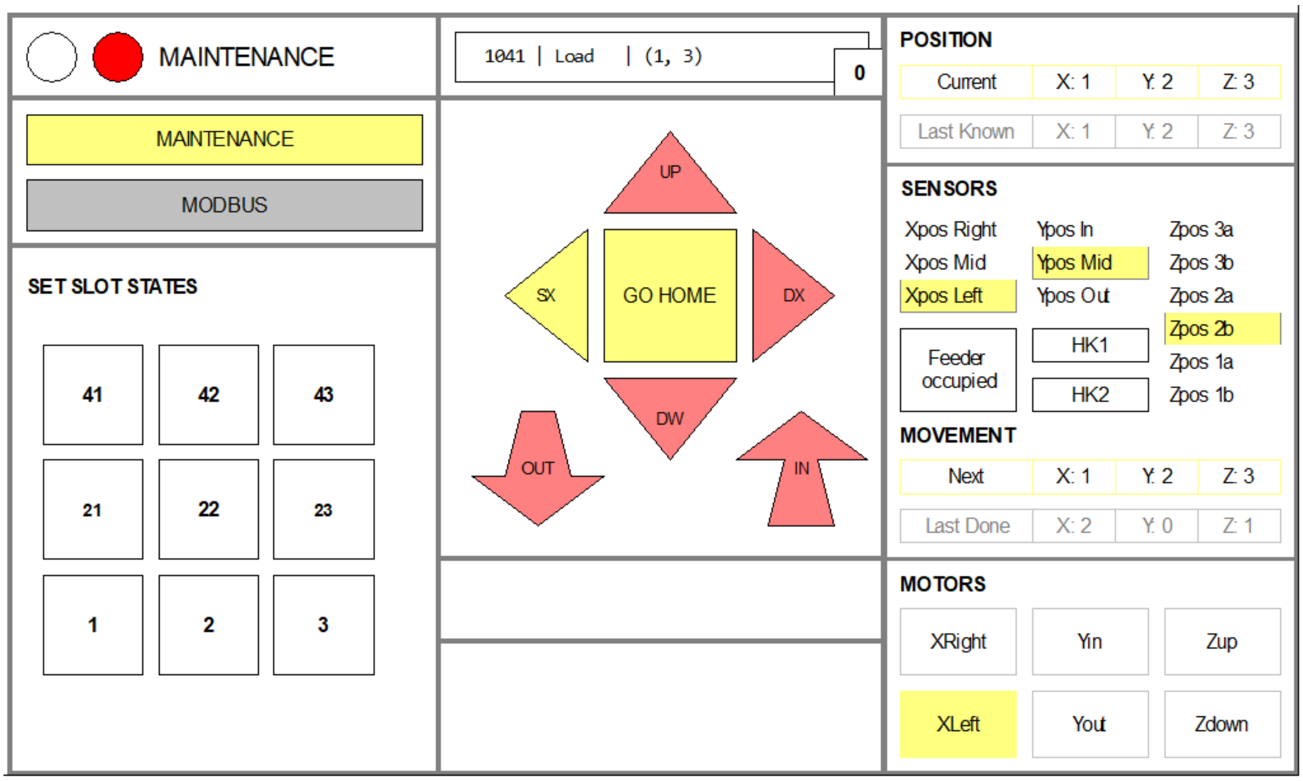


Figure 20: HMI for the manual mode

5 Communication

Modbus with TCP/IP client-server is a powerful communication protocol widely used in industrial automation systems to facilitate seamless data exchange between devices and control systems. This protocol leverages the TCP/IP networking stack to enable reliable and efficient communication over Ethernet networks. In this introduction, we will explore how Modbus with TCP/IP client-server works and the key components involved in its operation.

At its core, Modbus with TCP/IP client-server follows a client-server model, where communication is established between a client device or software application and a server device. The client initiates requests to read or write data from the server, and the server responds to these requests by providing the requested data or executing the specified actions.

The communication between the client and server in Modbus with TCP/IP occurs using the Transmission Control Protocol (TCP) as the transport layer and the Internet Protocol (IP) as the network layer. This combination ensures reliable, connection-oriented communication over Ethernet networks.

To establish communication, the client device initiates a TCP connection with the server device using the server's IP address and a designated port number. Once the connection is established, the client sends Modbus requests to the server, specifying the type of operation (read or write) and the data address or register to be accessed.

The Modbus protocol specifies different function codes that determine the type of request being made. For example, function code 03 is used for reading holding registers, while function code 06 is used for writing to a single register. The server processes the client's requests and responds accordingly, providing the requested data or confirming the successful execution of the write operation.

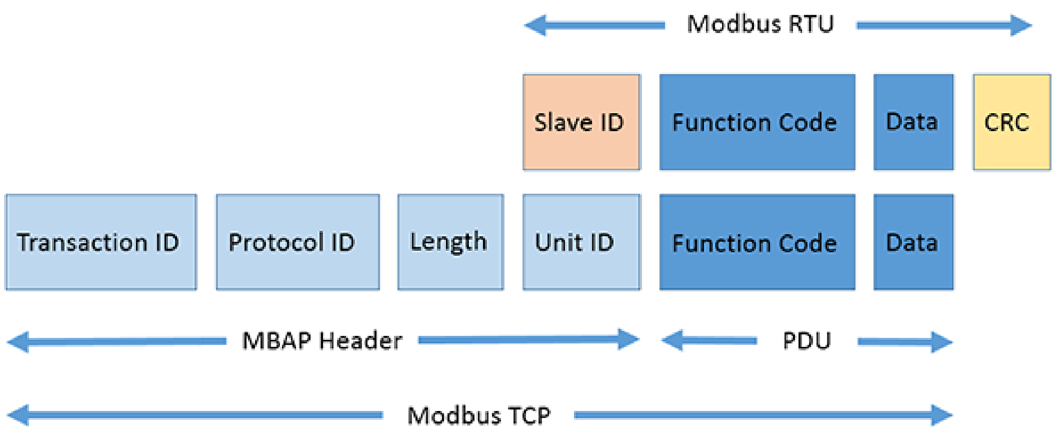


Figure 221: Structure of the RTU packet (serial) and the TCP one

The data in Modbus with TCP/IP client-server communication is typically organized into discrete registers or coils for digital data and holding registers for analog or numerical data. Each register is assigned a unique address, and the client and server devices use these addresses to identify and access the desired data.

One of the advantages of Modbus with TCP/IP client-server communication is its ability to handle multiple simultaneous client connections to a server. This allows multiple clients to interact with the same server, making it ideal for applications where data needs to be accessed by multiple devices or software applications.

In summary, Modbus with TCP/IP client-server communication follows a client-server model where the client initiates requests to read or write data from the server using the Modbus protocol over a TCP/IP connection. The TCP/IP stack ensures reliable communication, while the Modbus protocol defines the structure and function codes for data access. This approach enables seamless integration, efficient data exchange, and real-time control in industrial automation systems.

5.1 PLC

We will delve into the communication aspect of the Programmable Logic Controller (PLC) and how registers 1 to 8 are utilized to facilitate communication with Python scripts, enabling the execution of commands through the Modbus protocol. Additionally, we will focus on register 6 (%MW0.6), which serves as the reference for the machine state.

1.PLC Registers for Communication: Registers 1 to 8 within the PLC's memory structure are designated for communication purposes. These registers act as a dedicated interface, allowing seamless data exchange between the PLC and Python scripts running on external devices or computers with Modbus capabilities.

Through the utilization of Modbus commands, Python scripts can read from or write to these registers, enabling control and monitoring of the PLC-controlled system. The registers serve to bridge the communication gap between the PLC and Python, facilitating the exchange of information and commands.

2.Execution of Modbus Commands: The Modbus protocol provides a range of commands that facilitate the interaction between the PLC and external devices. With registers 1 to 8 acting as the communication interface, Python scripts can utilize Modbus commands to execute various operations on the PLC.

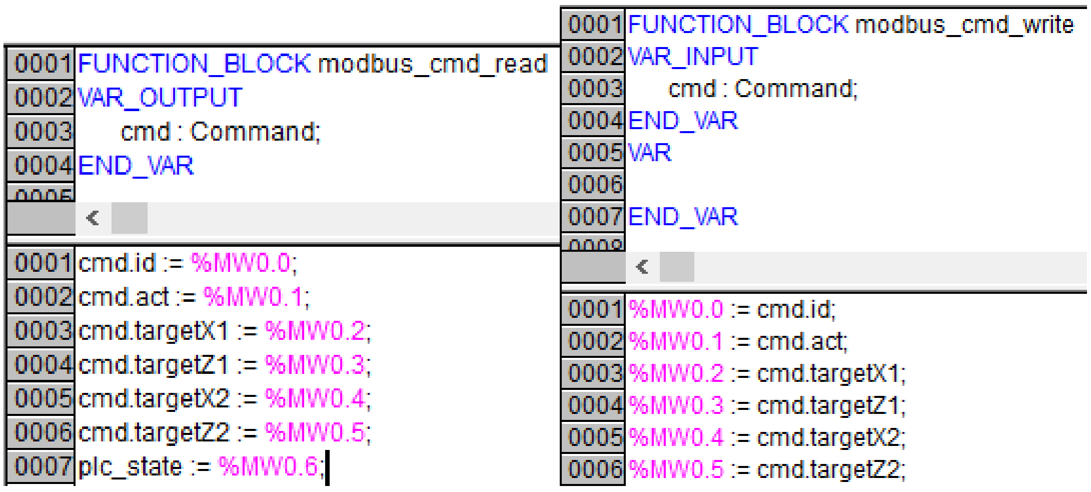
For instance, Python scripts can send Modbus commands to read specific data from the PLC's registers, retrieve sensor readings, or access important system parameters. Similarly, Python can transmit commands to write data to the registers, allowing for control of actuators, adjusting setpoints, or triggering specific actions within the PLC-controlled system.

3.Machine State: Register 6 (%MW0.6): Register 6, identified as %MW0.6, is specifically assigned to represent the machine state within the PLC. This register provides valuable information about the current operational state of the machine or a particular process within the system.

The machine state can be defined based on the requirements of the application. Examples of machine states include "running," "idle," "error," "maintenance," or any other relevant states that indicate the operational condition of the system.

Python scripts or external devices can read the value stored in register 6 to determine the current machine state. This information can be utilized to make decisions and trigger specific actions based on the observed state. For instance, if the machine state indicates an error, appropriate actions can be taken to stop the process, alert operators, or initiate error recovery procedures.

In this subchapter, we explored the communication aspect of the PLC, focusing on registers 1 to 8 used for communication with Python scripts and executing Modbus commands. These registers serve as a dedicated interface, enabling seamless data exchange between the PLC and external devices. Additionally, we discussed register 6 (%MW0.6), which represents the machine state, providing valuable insights into the current operational state of the system. Leveraging these registers and the Modbus protocol, communication and control of the PLC-controlled system can be efficiently achieved.



```
FUNCTION_BLOCK modbus_cmd_read
VAR_OUTPUT
	cmd : Command;
END_VAR
```

```
cmd.id := %MW0.0;
cmd.act := %MW0.1;
cmd.targetX1 := %MW0.2;
cmd.targetZ1 := %MW0.3;
cmd.targetX2 := %MW0.4;
cmd.targetZ2 := %MW0.5;
plc_state := %MW0.6;
```

```
FUNCTION_BLOCK modbus_cmd_write
VAR_INPUT
	cmd : Command;
END_VAR
VAR

END_VAR
```

```
%MW0.0 := cmd.id;
%MW0.1 := cmd.act;
%MW0.2 := cmd.targetX1;
%MW0.3 := cmd.targetZ1;
%MW0.4 := cmd.targetX2;
%MW0.5 := cmd.targetZ2;
```

Figure 232: PLC Registers

5.2 Python PLC library

The PLC library is a tool created to handle communication between the PLC and high-level Python programs. This library is specifically designed to integrate with our PLC program.

With its specialized classes, offers a robust framework for managing and executing commands, facilitating Modbus communication, and providing a simulated environment for testing and analysis.

There are four key classes: Command, ComunicationModule, CommandHandler and Simulator. For each of those classes, the specifications with purpose, attributes and methods are explained below.

5.2.1 Command

At the core of this library is the Command class, which provides a structured and organized way to represent commands, with all their attributes like action type, targets, and identifiers.

It ensures data consistency and integrity by validating inputs and applying appropriate attribute assignments. It allows commands to be easily created, modified, and accessed within a system.

5.2.1.1 Attributes

- **action**: A string representing the type of action (e.g., "Load", "Unload", "Move", "Delete").
- **id**: An integer representing the unique identifier for the command.
- **target_from**: A tuple (x, y) representing the starting position or source of the action.
- **target_to**: A tuple (x, y) representing the ending position or destination of the action.
- **action_multiplier**: A constant used to generate unique identifiers based on the action type.

5.2.1.2 Methods

- **__init__(action: str | int, target_from=(0, 0), target_to=(0, 0), slot_id=0, delete_id=0):** Initializes a new Command object with the provided parameters. It handles the validation and assignment of attributes based on the given inputs.
- **get_data() -> List[int]:** Returns a list of integers representing the command's registers that will be shared using the Modbus protocol.
- **get_slot_id(self) -> int:** Returns the slot ID portion of the command's unique identifier.
- **to_string(self) -> str:** Returns a human-readable string representation of the command, summarizing the unique identifier, action type, and relevant target positions.

5.2.2 Communication Module

The CommunicationModule class serves as a fundamental building block, providing a reliable means to establish communication between our implemented PLCs and external systems.

Relying on the pyModbusTCP library it enhances the capability of a Modbus client handling the connection to the host via a specified port, offering methods for reading and writing the Modbus registers of commands, plc state, and error codes.

5.2.2.1 Attributes

- **client**: An instance of the ModbusClient class used for communication with the PLC.
- **host**: A string representing the host IP address.
- **port**: An integer representing the port number used for communication (502 is the Modbus TCP/IP standard port).
- **command**: An instance of the Command class representing the current command.
- **state**: An integer representing the state of the communication module.
- **error_code**: An integer representing the error code received from the PLC.
- STATE_ADDRESS: An integer representing the Modbus register address for the state.
- ERROR_CODE_ADDRESS: An integer representing the Modbus register address for the error code.
- Other constants mapping the enumeration of the PLC main automata states.

5.2.2.2 Methods

- **connect():** Establishes a connection with the Modbus server hosted by the PLC. Raises a \`ConnectionError\` if the connection fails.
- **read_registers(address, count) -> List[int]:** Reads Modbus registers starting from the specified address and returns the register values as a list of integers. Returns \`None\` if the read operation fails after multiple attempts.
- **write_registers(address, registers) -> bool:** Writes the given list of registers to the specified Modbus address. Returns \`True\` if the write operation is successful or \`False\` if the write operation fails after multiple attempts.
- **update() -> boo**l: Reads Modbus registers using the \`read_registers()\` method and updates the module's state and error code attributes accordingly. If the registers are empty, a message is printed. If the state indicates a new command or a command completion the appropriate registers are read and converted to a \`Command\` object using the \`registers_to_command()\` method. The method returns \`True\` if the update is successful.
- **set_state(state: int) -> bool:** Sets the state of the PLC to the specified value by writing to the corresponding Modbus register. Returns \`True\` if the write operation is successful and the update is performed.
- **send_command_new(command: Command) -> bool:** Sends a new command to the PLC by writing the command's data along with the NEW_COMMAND state to the Modbus registers. Returns \`True\` if the write operation is successful.
- **send_command_done(command: Command) -> bool:** Sends a command completion signal to the Modbus client by writing the command's data along with the COMMAND_DONE state to the Modbus registers. Returns \`True\` if the write operation is successful. (Used by Simulator).
- **send_command_error(command: Command) -> bool:** Sends a command error signal to the Modbus client by writing the command's data along with the COMMAND_ERROR state to the Modbus registers. Returns \`True\` if the write operation is successful. (Used by Simulator).

- **registers_to_command(registers: list) -> Command:** Converts a list of Modbus registers to a `Command` object. Prints an error message and returns `None` if the number of registers is insufficient or an invalid command is encountered.

5.2.3 Command Handler

The CommandHandler class extends the functionalities of the CommunicationModule, acting as a central hub for managing commands and their execution within the system.

It offers a comprehensive set of methods for adding, deleting, and verifying commands, ensuring their validity and integrity before execution. Additionally, the CommandHandler class monitors the state of the PLC and handles the commands stream accordingly.

It saves in temporary lists the commands to send, the sent ones and the commands executed allowing the asynchronous management of commands with respect to the PLC state. It also provides functionality for persisting data, saving and loading those lists from a JSON file.

5.2.3.1 Attributes

This class in addition to the attributes inherited from the ComunicationModule has:

- **name**: A string representing the name of the command handler.
- **targets**: A list of tuples representing the valid targets (x, z) coordinates that the command handler can operate on.
- **file_path**: A `Path` object representing the file path of the JSON file used to save and load command data.
- **commands**: A list of `Command` objects representing the commands to be sent.
- **commands_sent**: A list of `Command` objects representing the commands that are in the PLC queue.
- **commands_done**: A list of `Command` objects representing the commands that have been successfully executed by PLC.
- **commands_error**: A list of `Command` objects representing the commands that encountered an error during execution.
- **history**: A list of `Command` objects representing the historical record of all sent commands.
- **polling_time**: A float representing the time interval (in seconds) between consecutive checks of the PLC’s state.

5.2.3.2 Methods

- **verify_target(target: tuple) -> bool:** Verifies if a given target (x, z) coordinate is valid based on the `targets` attribute. Returns `True` if the target is valid, `False` otherwise.
- **verify_command(command: Command) -> bool:** Verifies if a given `Command` object is valid. Checks the action and target coordinates against the `targets` attribute and performs additional checks for the "Delete" action. Returns `True` if the command is valid, `False` otherwise.
- **add(command: Command) -> bool:** Adds a `Command` object to the `commands` and `history` lists if the command passes the verification process. Returns `True` if the command is successfully added to the `command list`, `False` otherwise.
- **add_load(target_to: tuple, slot_id: int) -> Command:** Creates a "Load" command with the specified target and slot ID and adds it to the command handler using the `add()` method. Returns the created command if it is successfully added, `None` otherwise.
- **add_unload(target_from: tuple, slot_id: int) -> Command:** Creates an "Unload" command with the specified target and slot ID and adds it to the command handler using the `add()` method. Returns the created command if it is successfully added, `None` otherwise.

- **add_move(target_from: tuple, target_to: tuple, slot_id: int) -> Command:** Creates a "Move" command with the specified source and destination targets and slot ID and adds it to the command handler using the `add()` method. Returns the created command if it is successfully added, `None` otherwise.
- **delete(delete_id) -> Command:** Deletes a command from the `commands` list or the `commands_sent` list if the command is in the sent state. Returns the deleted command if it is successfully removed, `None` otherwise.
- **check() -> bool:** It is the central function of the Modbus client command handler. It performs various tasks accordingly to the updated PLC state, including sending new commands, managing completed or erroneous commands, and providing informative output displaying essential information such as the ongoing command, any encountered errors, and the status of the PLC. The method returns True if the check is successful and False otherwise.
- **run():** Implements the main loop of the command handler. It repeatedly calls the `check()` method, saves the command data, and sleeps for the specified `polling_time` between iterations. It can be interrupted by a `KeyboardInterrupt` to stop the execution of the command handler.
- **start():** Starts the execution of the command handler. It displays a message indicating the start of the command handler and calls the `run()` method.
- **stop():** Stops the execution of the command handler. It saves the command data and displays a message indicating the stop of the command handler.

5.2.3.3 File Handling

The `CommandHandler` class saves and loads command data to/from a JSON file using the `save()` method. The file path is specified by the `file_path` attribute. The JSON file has the following structure:

```
{
  "commands": [list of command data],
  "commands_sent": [list of command data],
  "commands_done": [list of command data],
  "commands_error": [list of command data],
  "history": [list of command data]
}
```

5.2.4 Simulator

The Simulator class within the library provides a simulated environment for the CommandHandler and implements the server side of the Modbus protocol allowing to test and analyze the behaviour of the system without physical connections to the PLC.

It implements the functionalities that are in our PLC program updating its state according to the main automata logics and emulating the execution of the commands setting consecutive timers. The Simulator class also facilitates the persistence of command data, ensuring that test cases and results can be stored and reused for future analysis and debugging.

5.2.4.1 Attributes

- **targets (list):** A list of tuples representing the valid target coordinates of the slots that it handles.
- **name (str):** The name of the simulator.
- **server:** An instance of the Modbus server used for communication.
- **data_bank:** An instance of the data bank to manage registers.
- **commands (list):** A list of commands to be executed.
- **commands_done (list):** A list of commands that have been executed successfully.
- **commands_error (list):** A list of commands that encountered errors.
- **polling_time (float):** The time interval for polling and updating the state.
- **polling_timer:** A timer to control the polling interval.
- **execution_time (float):** The execution time for each command.
- **execution_timer:** A timer to control the execution time of each command.
- **idle (bool):** A flag indicating if the simulator is idle or executing a command.
- **file_path (Path):** The file path to save the simulator's command data.

5.2.4.2 Methods

- **__init__(host, name: str, targets: list, polling_time: float = 0.5, execution_time: float = 10):** Initializes the simulator by setting up the necessary parameters, Modbus server, data bank, and loading command data if available. It allows for customization of the simulator's behaviour and configuration.
- **save():** Is responsible for saving the simulator's command data to a JSON file. This allows for the persistence of the command data, ensuring that it can be loaded and reused in future simulations.
- **initialize_server():** Initializes the Modbus server instance for communication. It sets up the server with the specified host address and default Modbus port, allowing the simulator to send and receive Modbus messages.
- **initialize_registers():** Initialize the registers used by the simulator and initialize them at 0.
- **verify_target(target: tuple):** Verifies if a given target coordinate is valid based on the specified targets. It checks if the target coordinate exists in the list of valid targets, ensuring that the simulator operates within defined boundaries.
- **verify_command(command: Command):** Examines a command and verifies its validity. It checks the action and associated targets of the command to ensure they meet the requirements and constraints of the simulation. This prevents the execution of invalid commands.
- **execute_next_command():** executes the next command in the commands list. It moves the command to the commands_done list after successful execution, keeping track of completed commands. This method facilitates the step-by-step execution of commands in a controlled manner.
- **main_automata():** implements the main logic of the simulator. It updates data, checks the state of the simulator, and executes commands based on the current state. This method ensures that the simulator's behaviour and decision-making align seamlessly with the developed PLC logic.

- **run():** is the main loop of the simulator. It periodically invokes the main automata, updating the state, executing commands, and saving command data during each iteration. This ensures that the simulation progresses and remains synchronized with the desired timing.
- **start():** initiates the simulator by initializing the server, registers, and starting the main loop. It serves as the entry point for running the simulation and ensures that all necessary components are set up correctly.
- **stop():** stops the simulator by cancelling timers, saving the command data, and stopping the Modbus server. It provides a controlled way to terminate the simulation, ensuring that all resources are properly cleaned up and data is saved for future use.

6 Database

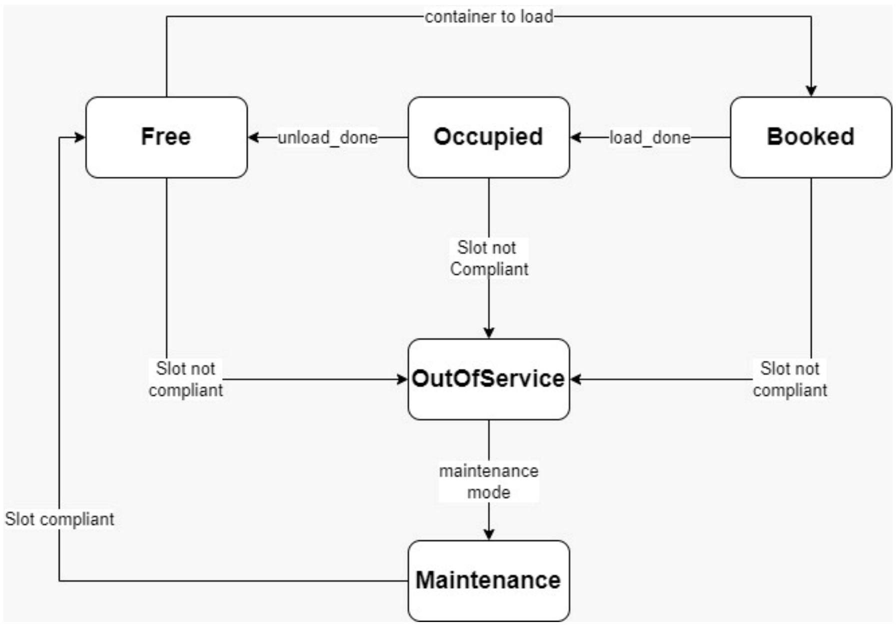
In the following, the database system that has been adopted in order to allow the horizontal scalability of the system is presented.

Given the advantage of the high-level programming of the Warehouse Manager, it was possible to make the database modeling more detailed and complex. It has been divided into several Excel sheets grouped by pertinence within the client's memory.

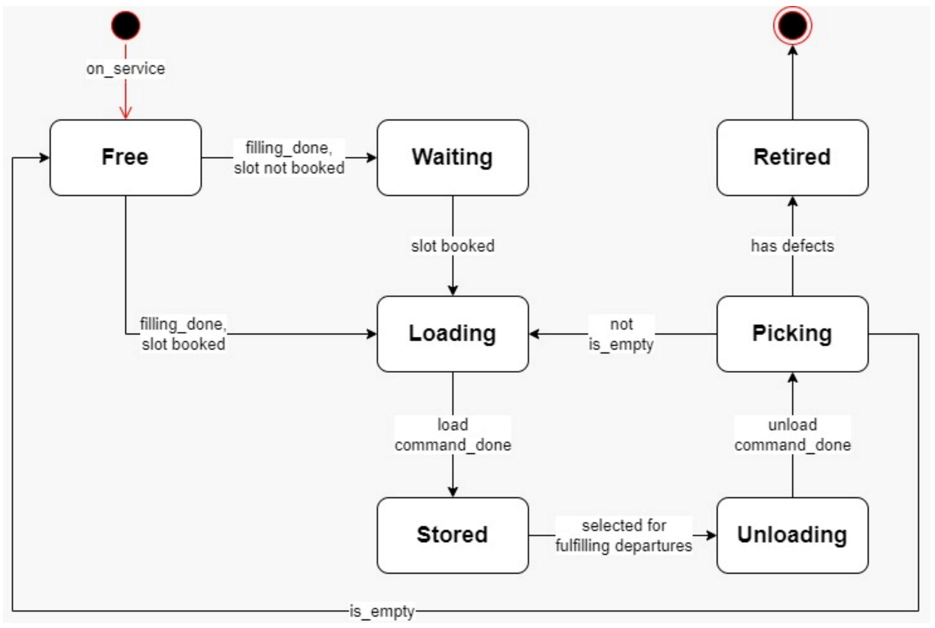
The database consists of three main files: "Components," "Items," and "Orders."

Components models the classes of slots and containers with their attributes, specifically:

"Slots" associates each slot with unique attributes such as "ID" (a unique code allowing up to 5000 allocations), "state" (describing its status as free, occupied, booked, outofservice, or maintenance), "area" (dividing the warehouse into areas managed by different PLCs - Modbus servers), "targetX/Z" (the coordinates of the slot within the warehouse), "path weight" (the length of the path to reach the slot from the loading and unloading position home), "loads/items count" (counters for warehouse performance analysis), "type" (referring to the slot type, i.e. small, medium, large), "width/depth/height/volume" (specifying the dimensions of the slot), and "items" (a list of elements stored in the slot).



"Containers" similarly associates each container with attributes such as "ID" (up to 5000 allocations), "state" (describing its status as free, stored, loading, unloading, picking, retired, or waiting), "slot" (the ID of the slot containing the container), "priority" (managing the loading/unloading phases in an orderly manner), "loads/items count" (counters for warehouse performance analysis), "type/width/depth/height/volume" (similar attributes to those used for slots), "filled/filling" (total occupied volume and percentage), "item types" (types of elements within the container), and "items" (a list of elements stored in the container).



Items models the types of products and their attributes and relationships, specifically:

“Items” associates each element with various information, including "Serial Number," "packaging type/quantity," "description," "supplier," "lead time," "lot quantity", "length/width/height/volume/weight, " marketing information such as "price buy/price sell/discount/price sell net," "safety stock/reorder point" for efficient goods replenishment management, "demand average/variability" (parameters used by the optimization algorithm) , and "stored quantity" (total quantity stored).

"Relations" section represents the correlation matrix of the products, it’s an index of products that are frequently ordered together.

Orders models the types of incoming and outgoing orders from the warehouse, specifically:

"Arrivals" document records incoming orders and associates each order with attributes such as "ID," "State" (used by the warehouse manager to manage the load, which can be scheduled, dispatched, arrived, dispatching, or cancelled), "arrival/dispatch time," "supplier," and "items" (products and quantities within the order).

Similarly, the "departures" sheet records outgoing orders and associates each order with attributes such as "ID," "State" (used by the warehouse manager to manage the unloading, which can be placed, delivered, fulfilling, or cancelled), "placed/deadline", "items" (products and quantities within the order), and "address" (destination of the order).

7 Optimization algorithm

Once the incoming orders have been computed, it is necessary to organize their loading into the warehouse. To accomplish this, a simple optimization algorithm has been developed to fill the slots in the best possible way.

The problem at hand involves selecting the optimal group of elements from a given set. The objective function aims to maximize the correlation between the selected elements of the group while also considering their demand and the quantity already stored.

7.1 Algorithm description:

The parameters of the problem include a set of elements, their corresponding demand values, the quantity already stored for each element, and a correlation matrix capturing the pairwise correlations between the elements.

$Input\ Set$: $S = \{1, \dots, n\}$

$Demand$: $d_i \ \ \forall\, i \in S$

$Stored\ Quantity$: $q_i \ \ \forall\, i \in S$

$$Correlation\ Matrix: \quad c_{ij} = \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix}$$

The decision variable is a binary variable that indicates whether an element is included in the optimal group.

$$Decision\ variable: \quad x_i \begin{cases} 1 & if\ x_i\ is\ in\ the\ optimal\ group \\ 0 & if\ x_i\ is\ not\ in\ the\ optimal\ group \end{cases} \ \forall\, i \in S$$

The objective function aims to maximize the sum of correlations between the elements loaded into the slot, aiming to reduce the number of slots to be unloaded to fulfil an outgoing order. Additionally, the demand for the elements is maximized to encourage the selection of highly requested items. However, the quantity already present in the warehouse is also taken into consideration to avoid overloading the warehouse with a certain item.

$$Objective\ function: \quad \max_x \left(\sum_{i \in S} \sum_{j \in S} x_i \cdot x_j \cdot c_{ij} \cdot d_i \,/\, q_i \right)$$

The constraint ensures that the group size is as specified, chosen to strike a balance between product heterogeneity and the container's capacity to fulfil as many unloading requests as possible. It can be adjusted after optimization is conducted in a real-world scenario.

$$Group\ size\ constraint: \sum_{i \in S} x_i = 4$$

The algorithm utilizes the MIP (Mixed Integer Programming) library in Python to formulate and solve the optimization problem.

8 Warehouse management system

The warehouse manager is the high-level program designed to effectively manage the flow of goods within the warehouse. It is a software that incorporates an optimization algorithm to determine the best way to store incoming products, considering factors such as space availability, product types, and specific warehouse requirements.

The warehouse manager also handles various components of the warehouse, such as shelving, storage areas, handling equipment, and machinery. Through specific commands, the program can coordinate operations within the warehouse, for example, by assigning storage positions, planning product movement, or managing picking activities.

Additionally, the warehouse manager is responsible for fulfilling outgoing orders. By using optimized routing algorithms, the program can determine the most efficient path to retrieve the requested products and organize shipments to minimize order preparation times and errors.

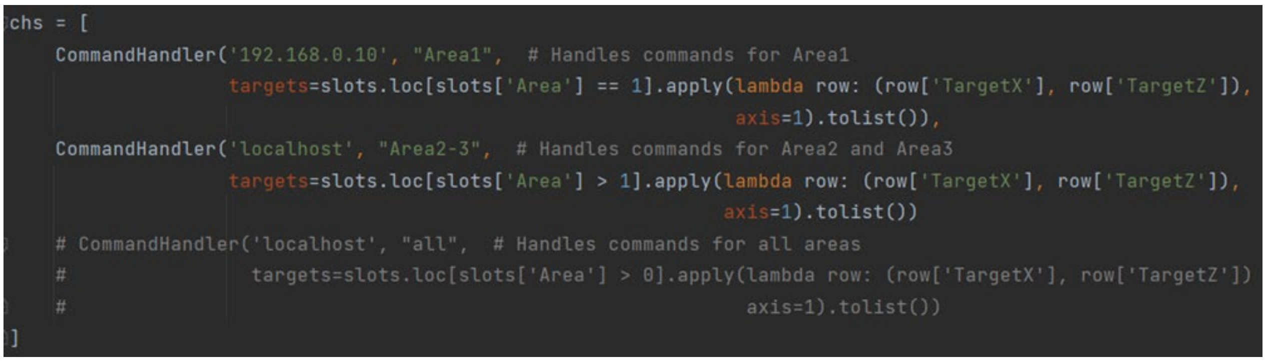
It's important to highlight that the warehouse manager is a highly flexible and customizable program since the needs and policies of warehouses can vary significantly. Therefore, the software can be continuously developed to adapt to the specific requirements of the given warehouse, improving operational efficiency and optimizing the flow of goods.

8.1 Initialization

At the beginning, the database tables are imported and stored in pandas DataFrames for efficient management within the program and to facilitate searches and assignments when working with a large amount of data.

Afterwards, the optimization algorithm "optimize_group" is imported, and the commandHandler class and Command class are imported from the custom-made PLC library. These classes will facilitate the management of commands between the various PLCs in the system and the program itself.

It is important to note the ease with which warehouse areas managed by different PLCs can be added by simply adding the specific commandHandler to the list of command handlers. The commandHandler should be initialized with the host, the name of the area, and the targets of the warehouse it manages. This flexibility allows for seamless integration of additional areas and PLCs into the system.



```
chs = [
    CommandHandler('192.168.0.10', "Area1",  # Handles commands for Area1
                   targets=slots.loc[slots['Area'] == 1].apply(lambda row: (row['TargetX'], row['TargetZ']),
                                                               axis=1).tolist()),
    CommandHandler('localhost', "Area2-3",  # Handles commands for Area2 and Area3
                   targets=slots.loc[slots['Area'] > 1].apply(lambda row: (row['TargetX'], row['TargetZ']),
                                                              axis=1).tolist())
    # CommandHandler('localhost', "all",  # Handles commands for all areas
    #                targets=slots.loc[slots['Area'] > 0].apply(lambda row: (row['TargetX'], row['TargetZ'])
    #                                                           axis=1).tolist())
]
```

8.2 Main Loop

The program is designed to be cyclic in order to allocate resources to the various tasks it needs to perform and to prioritize each of these tasks appropriately.

By utilizing a continuous loop and dedicating resources to each task, the program can operate autonomously and responsively. It ensures that warehouse operations are efficiently managed, and tasks are assigned and completed with the appropriate priority.

The cyclic structure also facilitates the continuous updating of the warehouse's status and enables real-time decision-making based on the most up-to-date information.

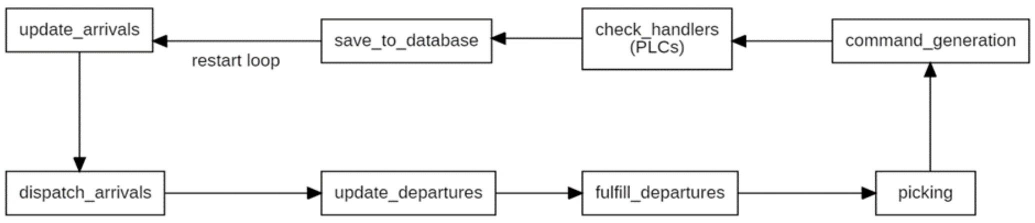


Figure 243: Main loop

8.3 Update arrivals

In the WarehouseManager, we have defined different states to track the progress of incoming orders.

When an order is first placed and sent to the supplier, it is marked as "Scheduled." This indicates that the order has been scheduled for delivery but has not yet arrived in the warehouse.

Once the courier arrives at our warehouse with the order, we update the status to "Arrived," signifying that the physical arrival is present in the facility, and it is ready for dispatching.

The state is "Dispatching" when the cargo is accepted by the dispatching algorithm. This means that the system is preparing to allocate the arrival to its designated location within the warehouse.

Finally, when the arrival is marked as "Dispatched."

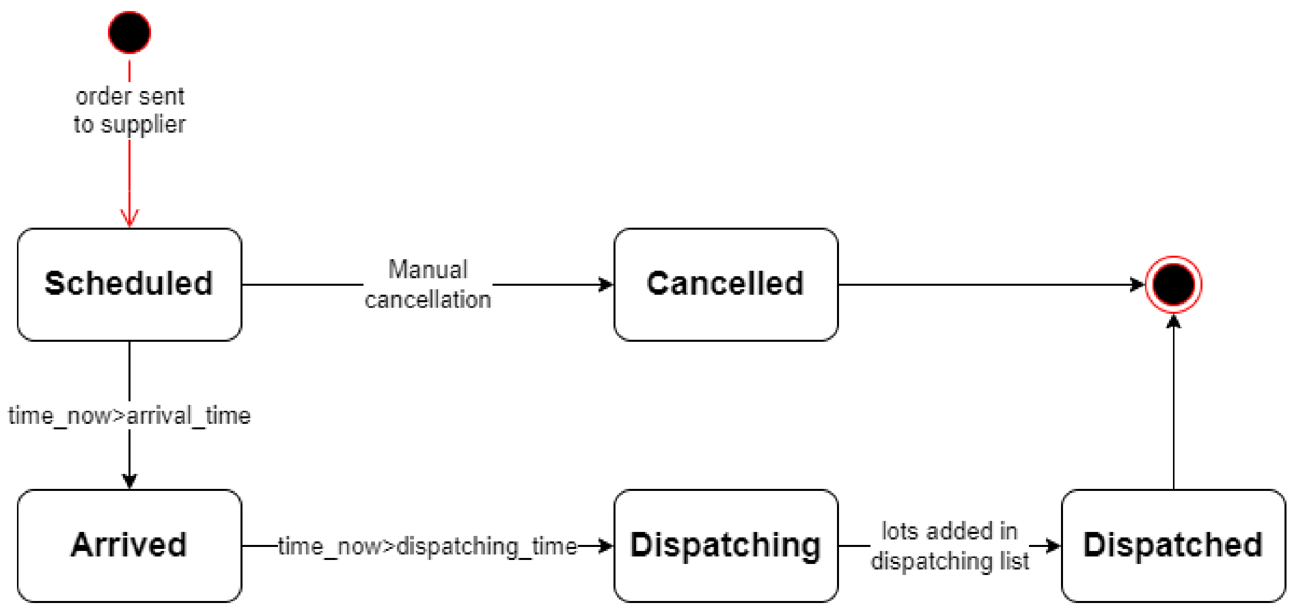


Figure 254: Arrivals

In particular, the function, `update_arrivals()`, update the state of arrivals and maintains a separate list, `dispatching_lots`, that tracks the lots currently being dispatched.

1. The function first checks for arrivals that are scheduled and have an arrival time in the past. These arrivals are considered "late," and their state is updated from 'Scheduled' to 'Arrived'.

2. Next, it checks for arrivals that have already arrived (state is 'Arrived') and have a dispatch time in the past. These arrivals are considered ready for dispatching, so their state is updated to 'Dispatching'.

3. The function then looks for arrivals in the 'Dispatching' state. It iterates over the cargo items within these arrivals and checks each lot within the cargo.

4. For each lot, it checks if the lot's name already exists in the `dispatching_lots` list. If not, the lot is added to the list. If the lot already exists, it updates the quantity of the existing lot by adding the quantity of the new lot.

5. After updating the `dispatching_lots` list based on the lots in the 'Dispatching' arrivals, the state of these arrivals is updated to 'Dispatched'.

8.4 Dispatch Arrivals

The `dispatch_arrivals` function is a crucial part of the WarehouseManage responsible for handling the dispatching process of arrived lots. Its purpose is to efficiently subdivide the lots into containers and book slots for those containers using optimization algorithms.

8.4.1 Selecting Four Best Items:

Select up to four lots from the `dispatching_lots` list. The selection is based on an optimization algorithm called `optimize_group`, which considers the `items` and `relations` data to determine the best combination of items to load together in a container.

8.4.2 Selecting a Suitable Container:

Once the four best items are determined, the function calls the `select_container` function to choose a suitable container for loading the selected items.

Specifically, the `select_container` function iterates through containers, starting with an estimated item quantity that can fit in each container. It checks if there is a 'Free' container compatible with the items dimenstions. If a matching container is found, its index is returned. If no suitable container is found, it gradually reduces the item quantity until an adequate container is found or the item quantity becomes too small.

The initial value of this quantity can be setted as input, and his value optimized per privilegiare una distribuzione uniforme della merce nel magazzino

8.4.3 Filling the Container:

After selecting a container, the function proceeds to fill it with the selected items. This is done by invoking the `fill_with_many_items` function, which adds the items to the container identified by the `container_index`. The function also specifies a filling percentage (95) indicating the desired fill level of the container.

The filling system prioritizes an even quantity of products within the same container to improve product search during the picking phase and because the grouping algorithm ensures a strong relationship between items.

Once the container is filled, its state, priority, and item count are updated in the `containers` database.

8.4.4 Booking a Slot:

The next step involves reserving a slot for the filled container by invoking the `book_free_slot` function. This function receives the container as an input and employs specific booking algorithms to search for an available slot within the warehouse. It focuses on slots of the same type as the container being loaded.

The algorithm aims to find a slot with the minimum path weight that is also sufficiently distant from similar containers carrying the same type of items. To achieve this, the algorithm establishes a range of prohibited path weights close to those of similar containers. This range gradually decreases if no suitable slots are found for loading.

The initial value of this range can be set and optimized according to the specifications of the warehouse. If no slot is available, the container's status is designated as 'Waiting,' and the function concludes.

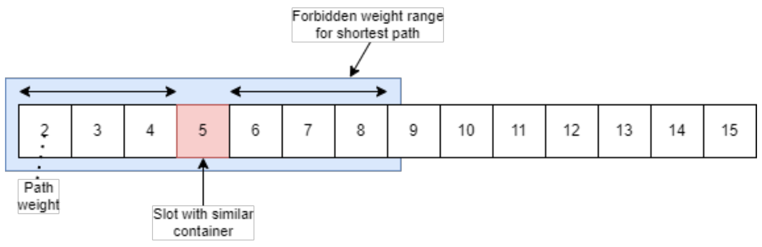

Forbidden weight range for shortest path
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Path weight
Slot with similar container

Figure 266: Path Weight

8.5 Update Departures

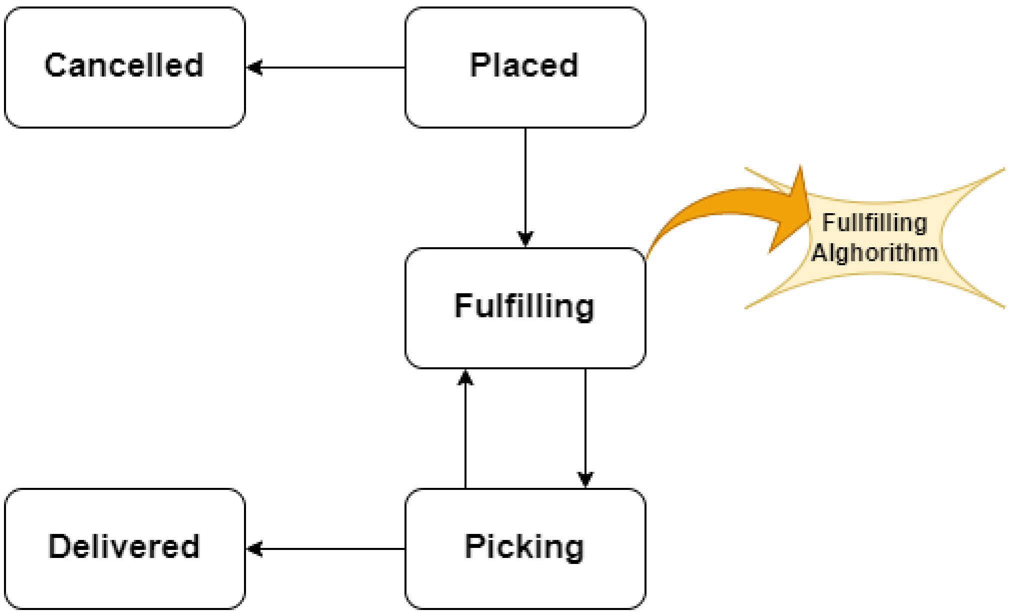


Figure 275: Departures

8.6 Fulfill Departures

"Dispatch Arrivals" is a function that allows for managing outgoing orders from the automated warehouse by reserving the unload of the container that can unload the maximum number of items required in the first 20 incoming orders. This optimization enhances the dispatching process.

It also checks if there is an empty container available that can be chosen to fulfill the orders, giving it priority over others. This is done to free up partially filled slots in the warehouse as soon as possible.

8.7 Picking

"Picking" is responsible for removing items from the containers and filling orders. It updates the status of the container to "Loading" if it still contains other items, or sets it to "Free" if it is empty.

The status of departures is updated with "Delivered" for completed orders, while orders that still require additional items remain in the picking stage.

These remaining items will be requested in the next iterations of the main process.

8.8 Command Generation

The commands to load and unload are generated all at once in the program, following the priority set in the container's "Priority" attribute. This priority attribute is determined by other previous functions, ensuring the correct management of the container flow within the system.

These commands are temporarily stored in a support list and then added to the respective Handlers' command queues. It is the responsibility of this class to send the commands to the PLC at the appropriate time.

8.9 Check Handlers (PLCs)

"Check Handlers" is responsible for invoking the "check" function in all commandHandlers to integrate the functionalities described in the PLC library session.

It updates the status of controlled PLCs and their executed commands, then if there are any new commands available, the commandHandler sends them to the PLCs.

8.10 File Management

The Warehouse Manager, along with the supporting PLC library, saves the execution context in temporary files at each cycle.

This allows for the interruption of the execution at any time to manually check or update the database and resume from where it left off.

This improvement enhances the debugging phases of simulations and prevents data loss in real-world use cases.

9 Validation

The validation step plays a crucial role in the successful implementation of an automated warehouse project. Simulations and tests provide valuable insights into the system's behaviour, performance, and reliability. By using simulations, developers can thoroughly assess the system's performance in a controlled environment, while performance tests verify the accuracy and efficiency of executing commands. Through comprehensive validations, the automated warehouse project can be refined and optimized, ensuring a reliable and effective solution for efficient product storage and retrieval.

9.1 Simulation

Simulation plays a crucial role in validating the automated warehouse project. By creating a virtual environment that closely resembles the real-world system, simulations allow for comprehensive testing and evaluation of the system's performance. The setup of simulation '*plc_simulator_all*' has been written in Python.

In the provided code, a simulator object is created using the PLC simulator module. The simulator is configured with the desired parameters, including the number of x-axis and z-axis targets, starting positions, polling time, and execution time. This simulation environment allows for testing and validating the behaviour and functionality of the automated warehouse system.

During the simulation, various scenarios can be tested to evaluate the system's performance. Metrics such as throughput, response time, and resource utilization can be analyzed to assess the system's efficiency and identify potential areas for improvement. Simulations provide a controlled environment for validating the automated warehouse project before its actual implementation, enabling developers and stakeholders to gain insights into the system's behaviour and make informed decisions based on the simulation results.

9.2 Test performance

Testing the performance of the automated warehouse system is crucial to ensure its reliability and accuracy in executing commands. The code '*plc_send_commands*' demonstrates tests performed using the CommandHandler module.

In the code, a CommandHandler object is created to perform a series of commands in the automated warehouse system. The commands include loading, unloading, and moving items to specific targets. By executing these commands, the system's ability to accurately carry out the requested actions can be evaluated.

During the performance test, the system's response time, accuracy in executing commands, and overall reliability can be assessed. This helps identify any potential issues or areas for improvement, ensuring that the automated warehouse system meets the required performance standards.

10 Discussions

The discussions chapter delves into potential future evolutions of the programmed warehouse, exploring avenues for further enhancements and scalability. This chapter examines two key areas: SCADA integration and the possibility of handling multiple warehouses, specifically focusing on vertical scalability.

10.1 SCADA integration

Integrating a Supervisory Control and Data Acquisition (SCADA) system into the programmed warehouse can offer significant advantages in terms of monitoring, control, and data management. SCADA systems provide a centralized platform for real-time visualization, remote monitoring, and efficient control of industrial processes. By integrating SCADA functionality into the warehouse system, the following benefits can be achieved:

- Enhanced Monitoring: SCADA systems enable real-time monitoring of various parameters such as inventory levels, equipment status, and performance metrics. This information can be presented through intuitive graphical interfaces, allowing operators to identify bottlenecks, track system efficiency, and make informed decisions for optimization.

- Remote Access and Control: SCADA integration enables remote access to the warehouse system, allowing operators to monitor and control operations from a centralized location. This capability facilitates efficient troubleshooting, maintenance, and rapid response to operational issues, even from off-site locations.

- Data Analytics: SCADA systems offer advanced data collection, storage, and analysis capabilities. By leveraging historical data and applying analytics algorithms, valuable insights can be gained to optimize warehouse operations, predict maintenance requirements, and drive continuous improvement.

- Alarm and Event Management: SCADA systems provide sophisticated alarm and event management features, enabling timely notifications and alerts for abnormal conditions or system failures. This proactive approach helps minimize downtime, improve system reliability, and ensure timely intervention when issues arise.

Integrating SCADA functionality into the programmed warehouse enhances its operational efficiency, reliability, and data-driven decision-making capabilities. By harnessing the power of real-time monitoring, remote access, and advanced analytics, the warehouse system can further optimize its performance and adapt to changing requirements.

10.2 Multi warehouse management

As the demand for automated warehousing solutions grows, the ability to handle multiple warehouses becomes increasingly relevant. Vertical scalability refers to the capability of the programmed warehouse system to expand and manage multiple warehouse facilities simultaneously. Here are some key considerations:

- Centralized Control: To handle multiple warehouses efficiently, a centralized control system can be implemented. This control system would oversee and coordinate operations across multiple warehouse locations, ensuring synchronized and optimized performance.

- Communication and Networking: Robust communication and networking infrastructure would be essential for seamless data exchange and real-time coordination between the central control system and individual warehouse units. Technologies such as industrial Ethernet, wireless communication, and secure network protocols would facilitate reliable and efficient data transfer.

- Resource Allocation and Optimization: With multiple warehouses, resource allocation and optimization become crucial. Advanced algorithms and decision-making mechanisms can be implemented to allocate storage space, manage inventory, and optimize material flow across different warehouse units, ensuring efficient resource utilization and minimizing operational costs.

- Scalable Architecture: The programmed warehouse system should be designed with scalability in mind, accommodating the addition of new warehouse units without significant architectural modifications. Scalable architecture allows for seamless integration of additional warehouses while maintaining system stability and performance.

By enabling the programmed warehouse system to handle multiple warehouses, vertical scalability provides the flexibility to adapt to expanding business needs, accommodate growth, and optimize operations across multiple locations.

11 References

- https://www.youtube.com/watch?v=vbPc4SruY5s&ab_channel=AverageEngineer
- https://www.netsuite.com/portal/resource/articles/inventory-management/warehouse-automation.shtml
- https://new.abb.com/plc/automationbuilder