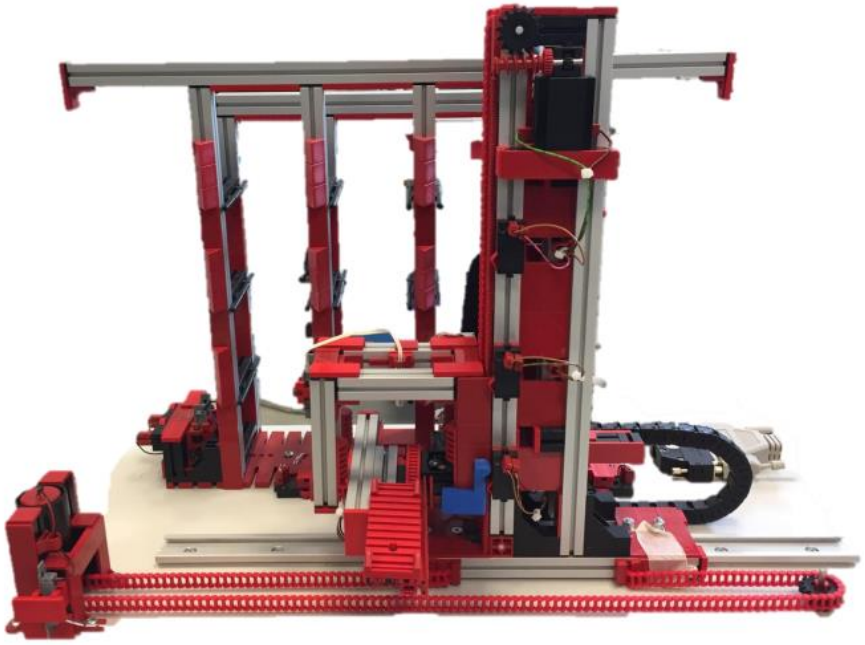
POLITECNICO
MILANO 1863

Report - Automation and Control Engineering Laboratory

AUTOMATION AND CONTROL OF AN INDUSTRIAL STORAGE WAREHOUSE



Mauro Balzarotti	**10493129**
Matteo Bergamaschi	**10504968**
Raffaele Giuseppe Cestari	**10490779**
Matteo Colombo	**10454300**
Sonia De Santis	**10525342**

Table of Contents

1. Problem statement and system description

The assigned project aims to the automation and control of a prototype of an industrial automatic storage warehouse. The structure of the warehouse model is constructed by nine cells, which can be identified considering the $x - z$ plane, as shown in Figure 1:

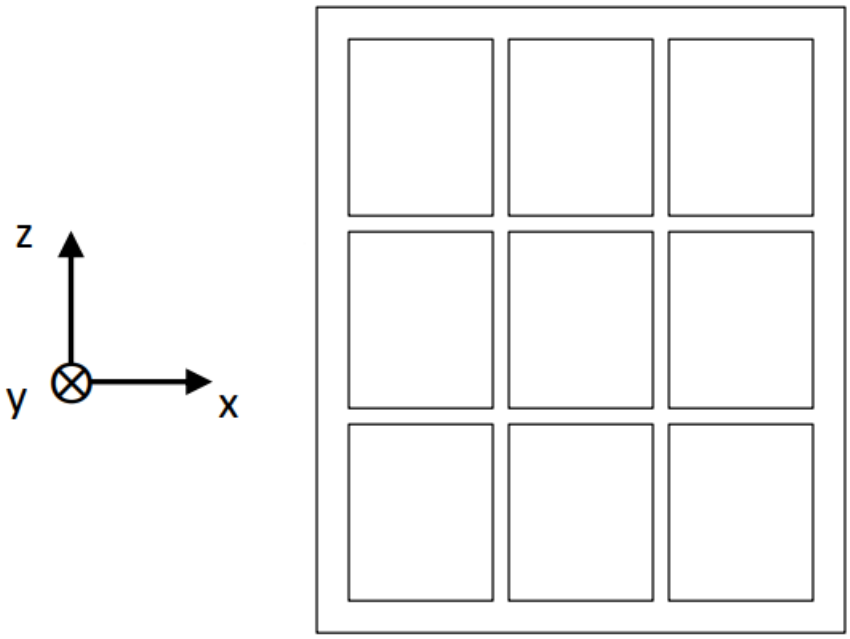


Figure 1 Warehouse X-Z plane

The plant is controlled by a Programmable Logic Controller (PLC) of type PM554 equipped with the I/O module DC532 (ABB Group).
The core element of the system is the cart, responsible for moving packages around the warehouse and equipped with motors enabling movement along X, Y and Z axes, guaranteeing the system to have 3 degrees of freedom: the translational movements along these three directions. Each action performed by the plant will be a combination of actions along these directions.
During its movement, the cart activates 12 sensors distributed along the three axes:

- X-axis is provided with 3 sensors.
- Y-axis is provided with 3 sensors.
- Z-axis is provided with 6 sensors.

The sensors' distribution is represented in Figure 2.

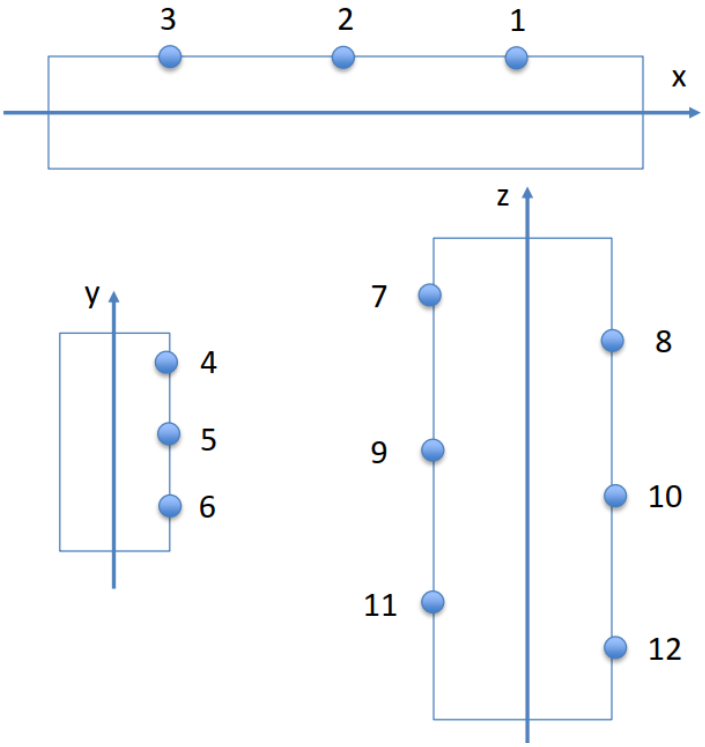


Figure 2 Sensor distribution

Each sensor corresponds to a Boolean variable, which assumes the TRUE value whenever the sensor is not active, while it becomes FALSE as soon as the presence of the cart activates the sensor; this means that sensors convention is *normally closed.*

To control the cart, it is possible to control the applied voltage on each motor, one for each axis, imposing the supply or the removal of the voltage on the desired motor. Whenever the voltage is supplied to the motor, a Boolean variable describing the state of the motor is correspondingly triggered. The motor is active if its associated Boolean variable is set to TRUE; it is off otherwise.

The overall input variables associated to both motors and sensors are captured in the following Table 1:

1	XposRight	Input
2	XposMid	Input
3	XposLeft	Input
4	YposIn	Input
5	YposMid	Input
6	YposOut	Input
7	Zpos3Above	Input
8	Zpos3Below	Input
9	Zpos2Above	Input
10	Zpos2Below	Input
11	Zpos1Above	Input
12	Zpos1Below	Input
13	FeederOccupied	Input
14	Handkey1	Input
15	Handkey2	Input
16	XRight	Output
17	XLeft	Output
18	YOut	Output
19	YIn	Output

Table 1 Sensor enumeration

As it can be inspected by looking at the above Table 1, there are further inputs variable linking the hardware system with the software implementation: FeederOccupied, Handkey1 and Handkey2. The three inputs represent respectively the sensor evaluating whether the package is on the cart or not, and two additional handkeys representing buttons which the user can use to interact with the system.

After having understood all the low-level functioning and the available components, before starting developing the implementation of the control strategies to manipulate the automatic storage warehouse, models of the system's structure and behaviour have been developed according to the Finite State Automata theory, which will be described in detail in the following section.

2. Modelling

The first thing to be noticed is that the warehouse system, being easily described as a discrete event system, can be intrinsically represented as a Finite State Machine, therefore as an automaton or as a synchronous composition of multiple sub-automata.

The structure of the warehouse has been decomposed along with the three degrees of freedom available, that are respectively X, Y and Z. Therefore, the automata modelling phase has been organized in modelling each axis as a combination of two automata, one representing the motor moving along that axis, and the other representing the set of sensors distributed along the same axis.

The software used to study the automaton models and to compute the synchronous compositions is Supremica 2.6.

2.1. MotorX automaton

In the following picture the MotorX automaton is shown:

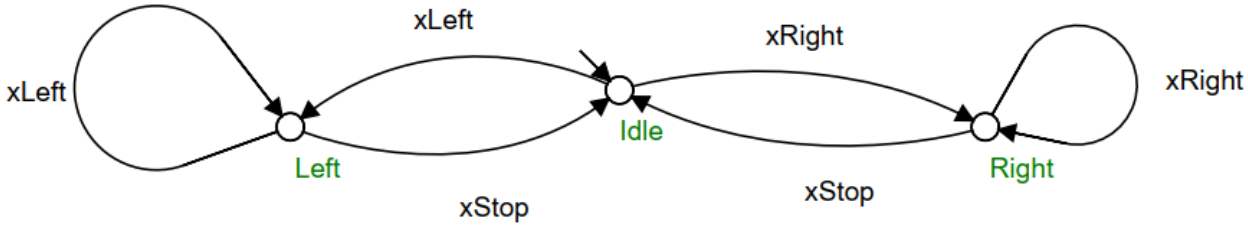


Figure 3 MotorX automaton

Three states can be recognized, *Left*, *Idle*, and *Right*, representing respectively the condition of moving left, being idle (off) and moving right.

Its alphabet is $\Sigma_{\mathrm{MotorX}} = \{\mathrm{x}_{\mathrm{Left}}, \mathrm{x}_{\mathrm{Stop}}, \mathrm{x}_{\mathrm{Right}}\}$.

2.2. SensorX automaton

Here it is shown the automaton representing the combination of sensors along the x-axis.

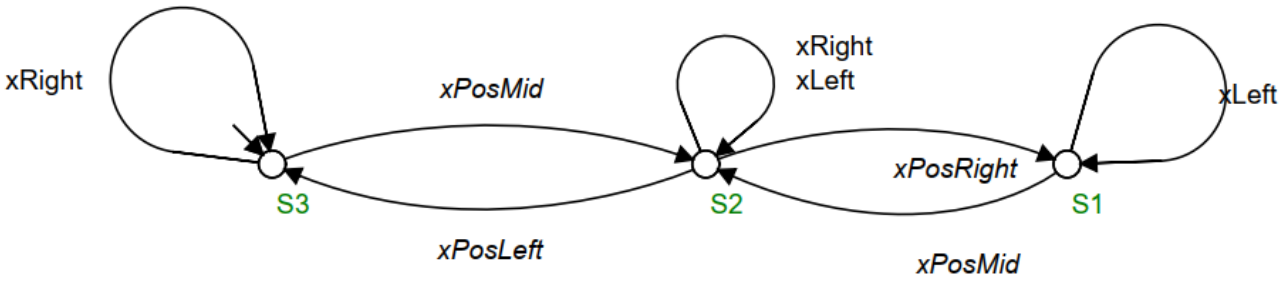


Figure 4 SensorX automaton

Its alphabet is $\Sigma_{\text{sensorsX}} = \{\text{x}_{\text{Left}}, \text{x}_{\text{Right}}, \text{x}_{\text{PosMid}}, \text{x}_{\text{PosLeft}}, \text{x}_{\text{PosRight}}\}$.

It is remarkable the presence of $\text{x}_{\text{Left}}, \text{x}_{\text{Right}}$ events in the alphabet of the sensorsX automaton, this represents a physical choice. The only events making the sensorX automaton changing its states are the triggers of the sensors, while motor events can be accepted but they do not lead directly to the change in sensorX state. The presence of $\text{x}_{\text{Left}}, \text{x}_{\text{Right}}$ in sensorX automaton will also affect the synchronous composition, generating the overall x-axis automaton, shown in Figure 5.

2.3. MotorX || sensorsX

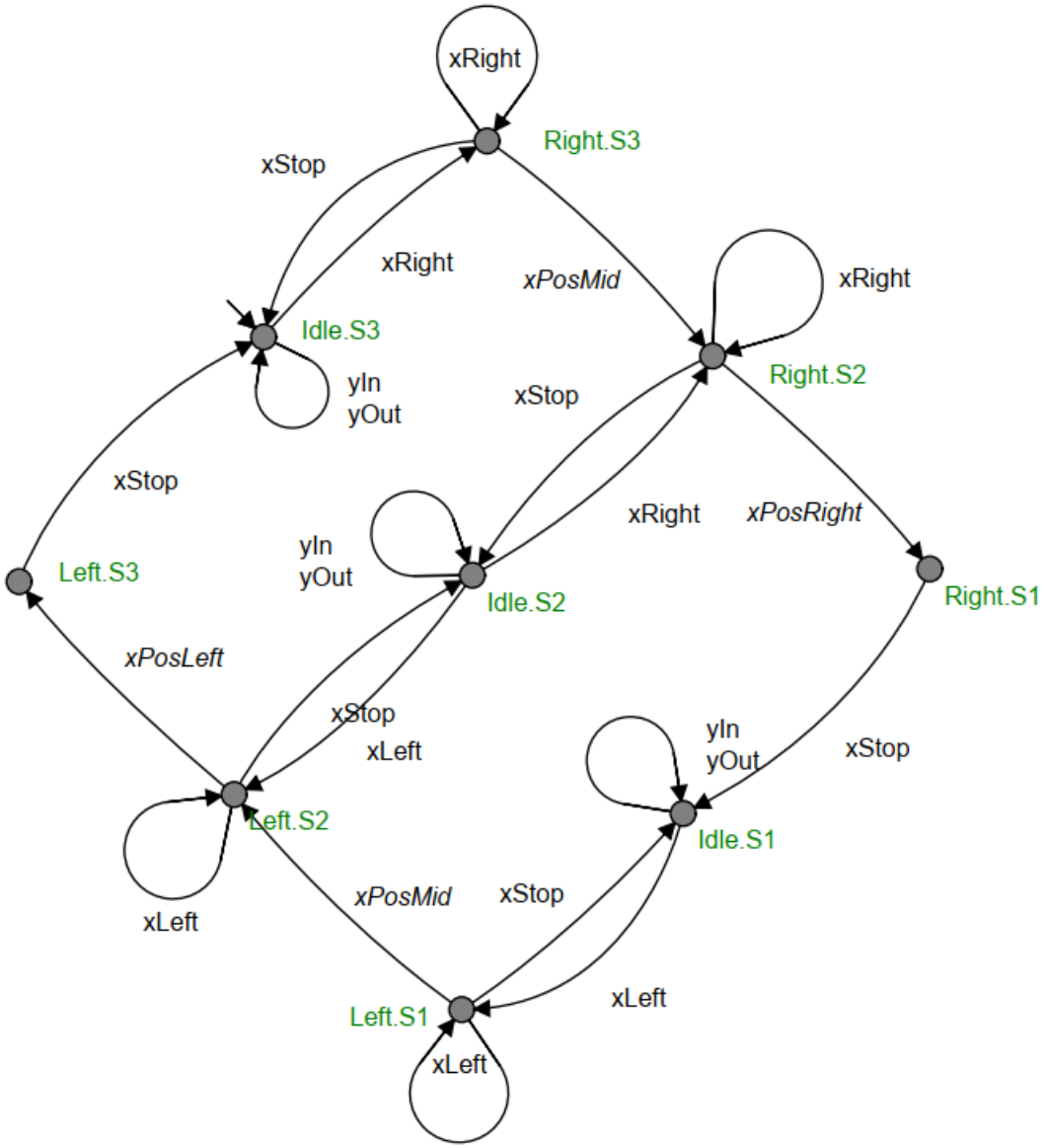


Figure 5 MotorX || SensorX automaton

The same strategies and core concepts are repeated for both the y-axis and z-axis.

For sake of conciseness, the automata representing the y-axis will not be shown here because they resemble the same behaviour of the x-axis automata, it can be found in Appendix 1. Here it will be presented the z-axis automata because they present a different configuration (due to a higher number of sensors distributed along the z-axis).

2.4. MotorZ automaton

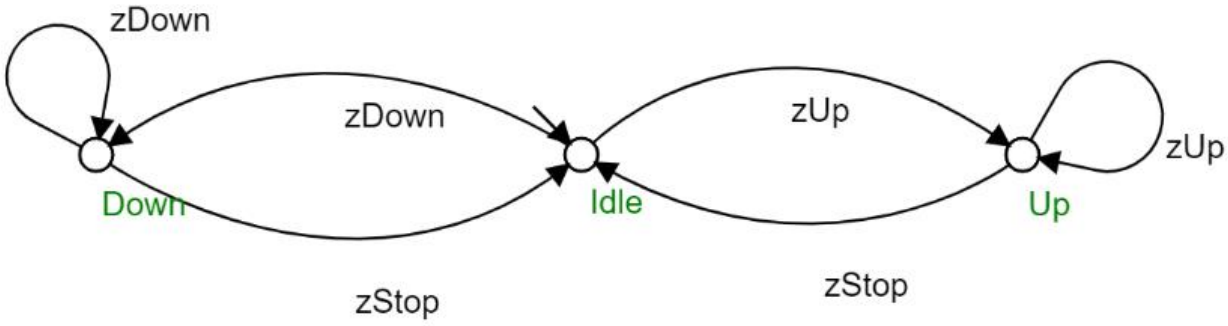


Figure 6 MotorZ automaton

$$\Sigma_{\text{MotorZ}} = \{z_{\text{Down}}, z_{\text{Stop}}, z_{\text{Up}}\}$$

2.5. SensorsZ automaton

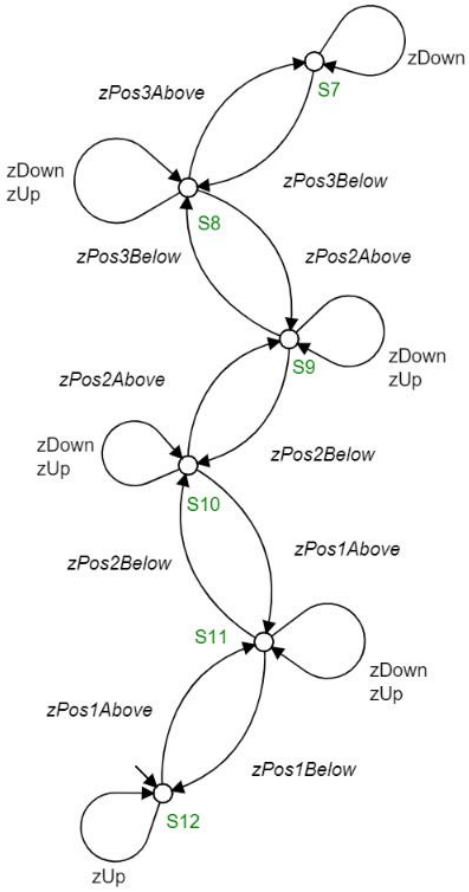


Figure 7 SensorZ automaton

$$\Sigma_{\text{SensorsZ}} = \begin{Bmatrix} z_{\text{Pos1Above}}, z_{\text{Pos1Below}}, z_{\text{Pos2Above}}, z_{\text{Pos2Below}}, \\ z_{\text{Pos3Above}}, z_{\text{Pos3Below}}, z_{\text{Down}}, z_{\text{Stop}}, z_{\text{Up}} \end{Bmatrix}$$

2.6. MotorZ || sensorsZ

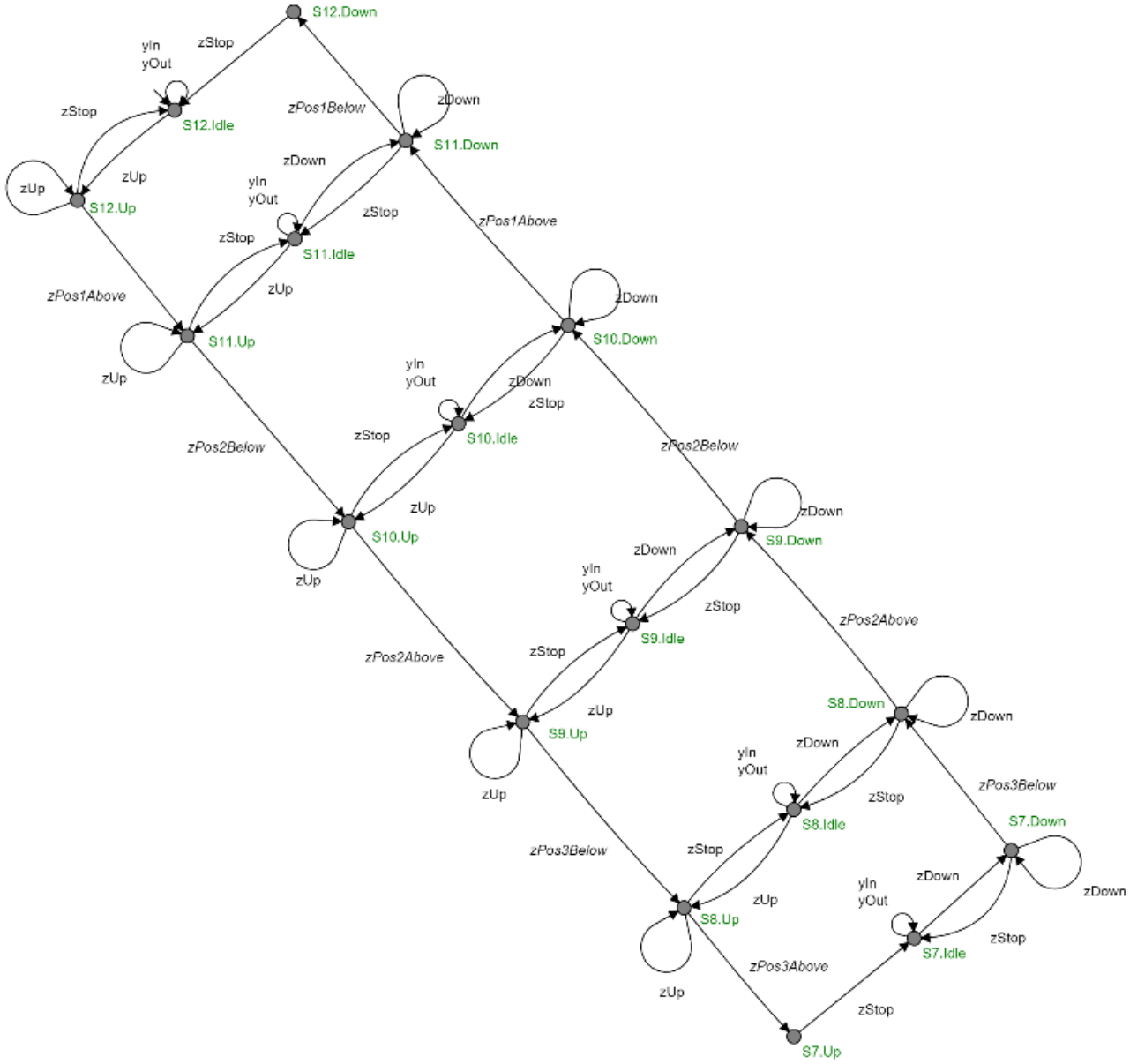


Figure 8 MotorZ || SensorZ automaton

It is important to notice that the above parallel compositions have been refined removing those transitions not representing physical state changes, not corresponding to a feasible evolution of the system in practice. Some examples in this sense are changing state (meaning sensor triggered) when motor is in *idle state*, sensors activation in opposite direction with respect to the direction of movement.

For sake of visualization reasons, we choose to avoid showing the full synchronous composition concerning the automata modelling the intended behaviour along the three axes. The overall automata would have a very large number of states because the alphabets of the sub-automata overlap only partially, moreover it would not be easily interpretable.

However, some insights on the behaviour obtained considering the full model are the following:

The full automata model can be organized as a *three-layer automaton*, where each layer represents a position along the Y-axis. Once the system is in the layer corresponding to *YposMid*, the automaton shows the possibility of selecting movements along both X and Z directions, without any sort of constraint. Moreover, within the *YposIn* layer, it is clear how it is possible to perform movements only along Z-axis. Indeed, that position corresponds to the cart being within the cell where the package must be stored, hence can be performed only upward or downward movements to load/unload the package.

Finally, an important convention is where to set the initial configuration of the cart. The Home position chosen is the one corresponding to the following configuration of sensors: $\{\mathrm{XposLeft}, \mathrm{YposOut}, \mathrm{Zpos1Below}\}$.

3. User Interface

A graphical interface has been designed with the purpose of both interacting with the system and monitoring the evolution of the system through a graphical representation of its states. User interface is equipped with two separate visualization interfaces, the *screen* and the *HMI*.

3.1. Screen

It is a visualization interface to allow programmers to monitor the position of the cart along both the X-Z plane, but also along the Y-axis, providing a full tri-dimensional monitoring interface returning the current state of the system. It is also equipped with an interface showing which motor is currently active. The Screen interface not only has a monitoring role but also allows to simulate the behaviour of the warehouse in absence of the direct connection with the real plant. To do so, it is equipped with an equivalent set of all the sensors along the X, Y and Z axes, allowing the programmers to simulate the functioning of their implemented methods. It is also available a further button FeederOccupied, allowing to simulate the presence of the package on the cart as well. In Figure 9 a first snapshot of the Screen interface.

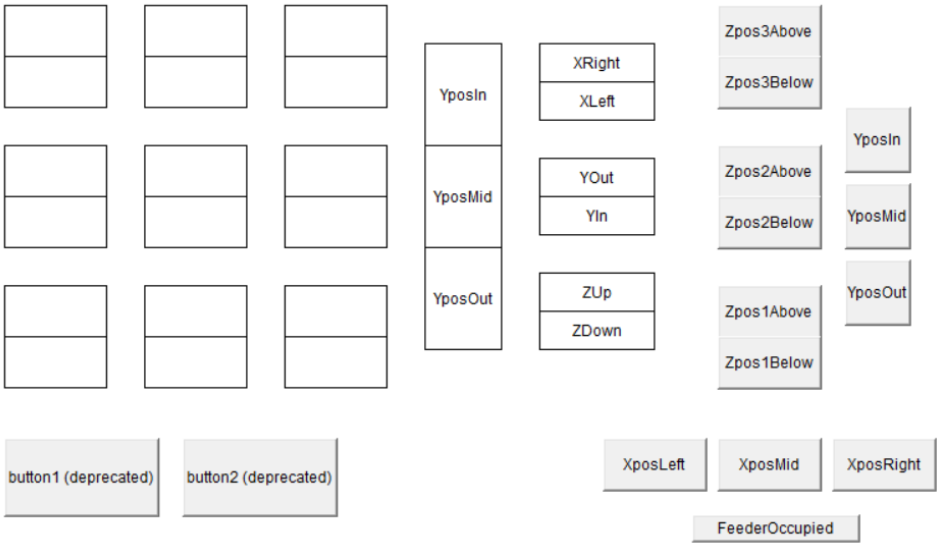


Figure 9 Screen interface

In Figure 10 it will be shown a configuration of the Screen Interface when performing the simulation.

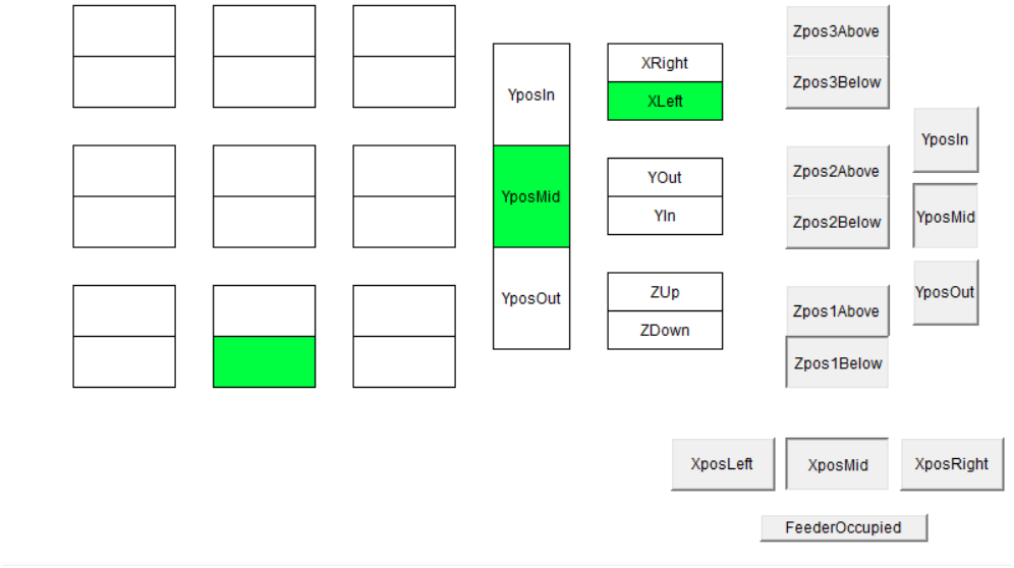


Figure 10 Screen interface in simulation mode

The green highlighted cells represent the condition in which the warehouse is. As an example, it will be explained the above configuration: on the left of the figure, in the first row, second column of the matrix of cells, it is showed the current position of the cart. In detail it is in $[\mathrm{XposMid}, \mathrm{Zpos1Below}]$. To check in the *depth* direction, Y-axis, we must look at the YposMid cell, which indeed is coloured in green. Then, on the right of it we can see the motor column status, from here we can identify the XLeft motor as active, therefore we can foresee that the sensor configuration will soon change from XposMid to XposLeft. Eventually, on the right we can see the provided buttons, each one simulating the trigger/un-trigger of each of the sensor which each button represents. To continue the simulation, if we are not directly connected to the true plant, we must manually toggle those buttons to mimic the behaviour of the physical sensors.

3.2. HMI

The HMI interface has the purpose of simulating the effective user interface that will be assigned to the users working with the warehouse. It is designed with 9 cells, each one representing the corresponding physical cell available in the warehouse. The colour of each cell is by default WHITE, as the cell is empty (no package has been stored yet in the cell). The same cell becomes BLUE whenever the program saves the information of the package being stored in the cell. The HMI interface can be in two configurations, depending on whether login has been done or not. In the following figure the setup of the pre-login condition.

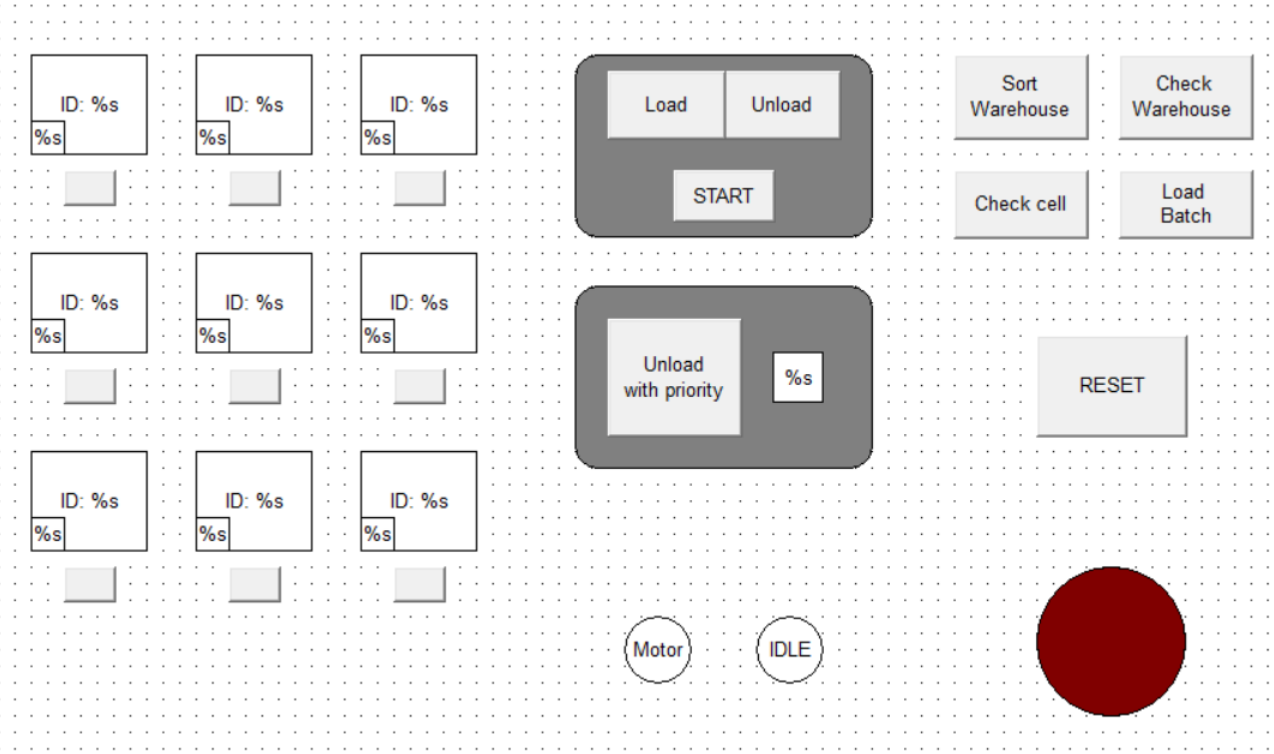


Figure 11 HMI interface

A view of the HMI interface while running the PLC can be found in Appendix 2.

Now it will be given a brief explanation on the content of the interface:
under each cell is available a small button, meant to be used by the user to perform cell selection, then, once the desired cell has been selected, the user can perform load and unload operations on the very same cell. The implementation details will be provided in the next sections of the report.

The interface also provides the following buttons:

- **START**: it starts the execution of non-automatic procedures such as load and unload, it is required to be pushed after having selected the desired cell and the desired command.
- **RESET**: it performs the physical reset of the position of the cart, bringing it back to the Home position $\{\mathrm{XposLeft}, \mathrm{YposOut}, \mathrm{Zpos1Below}\}$. See Reset().
- **Sort warehouse**: re-ordering of the packages stored in the warehouse according to an optimization algorithm, minimizing the required time to pick-up high priority packages. Further details will be given in implementation section Reorganise().
- **Check warehouse**: it scans all the cells of the warehouse and checks whether a package is still present or not. See Total_Check().

- **Check cell**: checks if the selected cell contains or not a package, evaluating the FeederOccupied sensor.
- **Emergency Button**: The great red button, in the right side of the interface, is the Emergency Button. At any time, the user can press that button and its effect on the warehouse will be of stopping immediately the cart, independently from the task being currently performed, switching off all the motors. It also deactivates alarms if triggered.
- **Motor**: alarm light, informing if any of the motors X, Y or Z, going in any direction (as mentioned previously, each motor rotates clockwise or counterclockwise, imposing translational movement in a specific way.) has been stuck in the same condition for a certain amount of time; the time considered depends on the application, for simulation purposes it has been imposed to $10s$. A reasonable choice for a warehouse with hundreds of cells might be 300s, this number will be dependent on the dimensions and on the pre-existing security features of the motors. Further details will be given later.
- **Idle**: alarm light, informing if any of the sensor detects the cart being in the same position for a certain amount of time; the time considered depends on the application, for simulation purposes it has been imposed to $10s$. The highlighted phenomenon may represent the permanence of the cart in the same position for an extremely large amount of time, reflecting the fact of the warehouse not performing any task.

A clear depiction on how the main process selects the various features can be found in Appendix 3.

4. Load/unload

Before starting to implement flexible functions, the first approach to the problem was to manually implement a sequence of actions to achieve the goal of loading and unloading the package in a desired cell. This step is not discussed in depth and it is just mentioned before entering in the implementation details. The trial lacked generality, however it had the relevant goal of increasing the team expertise in the management of the plant functionalities and understanding the PLC logic, with its differences from the standard implementation rules. Just to mention some of them, one can think about the behaviour of the PLC which scans throughout the code multiple times, at a given frequency, intrinsically implementing a cyclic behaviour. This fact is a meaningful difference with respect to traditional programming, indeed it is suggested to avoid using *while and for loops*, because their duration may exceed the time allocated by the PLC to perform a scan throughout the code, leading to compiling errors. Thanks to the first approach to the PLC coding, we have been able in understanding these features and orient our coding strategies to keep them into account.

To present all the implemented functionalities, the idea is to follow the logical path throughout the development, way through which the team struggled during the development of the project.

The first task that has been tackled was of building the implementation for reaching the goal of loading and unloading a package in a desired cell. To achieve this objective, many functionalities have been implemented. In the following key points behind the implementation of those functionalities will be presented.

4.1. MoveXYZ()

First, the function *move()* has been developed, it allows to move the cart from its current position to the desired one, by performing a comparison between the actual and the desired position, having the information coming from the sensors.

moveXYZ(axis: STRING; **pose**: ARRAY [1..3] OF INT; **des_pos**: INT**): BOOL**

Inputs:

- **Axis**: desired movement axis
- **Pose:** actual position of the cart
- **Des_pos**: desired position of the cart

Outputs:

- Returns a **Boolean,** TRUE if the desired position has been reached.

The idea with which this function has been developed, is to be as general as possible, defining within the set of the inputs the axis along which the movement should be performed, defining it as a string. Then, the second input is an array, pose, provided as output of another function which will be immediately described in the following, it describes through an array of integers the current position of the cart. The third input is the desired position, defined as an integer, to be reached with the action of movement. It is enough to specify the desired position as an integer because the axis has already been selected, hence the problem is mono-dimensional.

4.2. Current_pose()

It is one of the core functions, together with MoveXYZ() of the entire implementation, being called multiple times, and providing as output the position of the cart as an array of integers, once evaluating the sensors signals.

Current_pose(): ARRAY [1..3] OF INT

Inputs:

Outputs:

- Returns a **Boolean,** TRUE if the desired position has been reached.

The two functions above explained are the key components for what concerns the movement operation, which is fundamental in any more complex task.

4.3. Deposit_package() and Extract_package()

Now, other two key complementary functions are presented which have been used in order to implement the basic operations of loading and unloading a package, once the desired cell have been reached. Deposit_package() and Extract_package() are respectively the two functions implementing the sequence of actions necessary to perform the deposit operation and the pick-up operation of a package, according to the physical constraints of the warehouse at hand. In particular, in order to perform the drop operation, once the desired cell has been reached, the sequence of actions is the following:

1) The cart enters the cell from the *above* position.
2) The cart moves along Z-axis and reaches the *below* position, in the same cell. This action is such that the package is left on the current cell.
3) The cart exits the cell, without the package.

The pick-up operation, on the other hand, is symmetric, as it is clear looking at the following sequence:

1) The cart enters the cell from the *below* position.
2) The cart moves along Z-axis and reaches the *above* position, in the same cell. This action is such that the package is picked-up in the current cell.
3) The cart exits the cell, with the package onboard.

Deposit_package(): BOOL

Inputs:

Outputs:

- Returns a **Boolean,** TRUE if the operation is completed.

Extract_package(): BOOL

Inputs:

Outputs:

- Returns a **Boolean,** TRUE if the operation is completed.

Thanks to the above set of functions, we have been able to implement the load/unload task with success, in different cells with repeatability.

4.4. Deposit _ladder()

In order to better grasp the full potential of the programming languages available from the (International Electrotechnical Commission), it was decided to rewrite the *Deposit_package* function using the ladder language. The same logic used for structured text has been implemented. For this function, as can be seen in Appendix , the ladder language is simpler, more intuitive and ensures ease of debugging. For these reasons it is a language widely used in the industrial sector.

It must be noted that the great accessibility of this language comes at the price of severe limitations trying to implement more complex functions. The more problematic cases would be impossible to program and the feasible but complex ones would have to deal with blocking signals and verbose implementation.

4.5. Reset()

Before moving to more complex tasks, it is then decided to implement another fundamental function, the *Reset()* function. Differently from the previously mentioned ones, it has been implemented as a function block and not a simple function, because function blocks as it is specified in CodeSys and ABB documentation, do not reinitialize the internal variables on each cycle, a feature which is required in order to track whether the reset functionality has already been called or not.

Reset(disable: BOOL): **BOOL**

Inputs:

- **Disable:** reset the output of the function to FALSE once it has been executed.

Outputs:

- Returns a **Boolean,** TRUE if the operation is completed.

In order to avoid in forbidding a successive invocation of Reset(), it has been used the additional input *disable*, which allows to reset the function output to FALSE. If it were not implemented in this way, the effect would result in the possibility of properly invoking the function only once.

With this final function, the set of tools necessary to implement low-level task has been prepared. In the following section more complex goals will be presented, whose implementation has been constructed basing on the above implemented functions, without re-inventing the wheel but instead exploiting the basic concept of software engineering of modularization and re-usability of code, which at the same time guarantee robustness to code and a lower time expense in proceeding in the development of code.

5. Automatic Load/Unload and Cell Status Check

5.1. Data types

Two data types have been introduced in order to better describe some components of the warehouse: Package and Cell_state.

1. **Package**: struct containing the information characterizing a package entity. Its fields are:
   - **Id**: integer number different from 0 identifying the package.
   - **Priority**: integer number greater than 0, characterizing the level of priority of the package. The lower this value, the higher the priority assigned to the package (e.g. 1 = high priority, 9 = low priority). Packages with high priority are the ones that should be unloaded first from the cells of the warehouse. Note that no upper bound is defined for this quantity. In this project priority values belongs to the range [1,9] because of the dimension of the warehouse, but the problem can be easily scaled for larger warehouses. More than one package can have the same priority value.
   - **Weight**: integer number characterizing the weight in kg of the package.

   Each package handled during the operations performed in the warehouse is identified by the latter data type.

2. **Cell_state**: struct containing the information characterizing the state of each cell of the warehouse. Its fields are:
   - **Occupied**: Boolean variable set to TRUE if a package is present in the considered cell, otherwise set to FALSE.
   - **Package**: struct identifying the package stored in the considered cell (see datatype Package described above).
   - **timerCell**: timer starting once a package is loaded in the considered cell. This information is useful in order to know how much time packages are stored in the warehouse.
   - **Expired**: Boolean variable set to TRUE if the package in the cell is stored for too long, reaching its expiration date (needed in case of perishable products). Otherwise, it is set to FALSE.

When a package is loaded in a cell, its state is updated with the package information, the occupied field is set to TRUE and the timer associated with the cell is initialized. On the other hand, when a package is removed from a cell, its state variables are set to 0.
The timerCell field is used as a value related to the specific product and it will just show the operator which packages are being stored from an excessive amount of time, the operator will than be able to unload the older packages first. An alternative usage of this value could be to take into account the expiration date of certain products and to develop functions able to better manage this type or goods, for example an automatic removal procedure of expired products and a stronger constraint inside the cost function to remove the packages.

A matrix of size 3x3 of Cell_state elements is introduced as a global variable, in order to describe the whole state of the warehouse. Its name is W_state.

```
W_state = [1..3, 1..3] of Cell_state
```

5.2. Load_Unload()

The successive development phases involved the implementation of more complex functions, each one based on the invocation of low-level functions already explained in the previous section. In the following section will be presented at first an automatic version of the *Load/Unload* function. The operation has been implemented as a function block, with the following specifications:

Load_Unload(disable: BOOL;pose: ARRAY [1..3] OF INT; cell: ARRAY[1..2] OF INT;carica: BOOL;): **BOOL**

Inputs:

- **Disable:** reset the output of the function to FALSE once it has been executed.
- **Pose:** actual position of the cart.
- **Cell:** actual cell of the cart (X-Z plane)
- **Carica:** TRUE if *load mode*, FALSE if *unload mode*

Outputs:

- Returns FALSE when the operation is completed.

The *disable* Boolean input has the same functioning previously explained in the *Reset()* function, allowing to call the same function multiple times. The function, in order to know which set of actions should be implemented, takes as input the current cell position. Then, finally takes a Boolean, *carica*, which specifies whether the execution that should be performed is devoted to loading or unloading the package.

5.3. Check_package()

After having implemented the functionality of automatically loading and unloading a package in the warehouse, the need of monitoring the status of the warehouse rose. The next function implemented is *Check_package()*, a function block with the purpose of monitoring the status of the selected cell, understanding whether the package is present or not in the given cell and updating correspondingly the cell status in an ad-hoc defined global matrix, representing the overall status of the warehouse. To sense the presence of the package, a lift movement is done, to trigger the FeederOccupied sensor. The function block output is the updated status of the cell state matrix. In the following the implementation details.

Check_package(disable: BOOL**): BOOL**

Inputs:

- **Disable:** reset the output of the function to FALSE once it has been executed.

Outputs:

- Updated Cell-state Matrix
- Boolean, TRUE when operation is completed.

Check whether the package is present or not in the current cell and updates subsequently the information within the global matrix of cell state. To check if the package is there, it performs a lift movement to sense FeederOccupied.

5.4. Position_CP()

A propedeutic function thought to prepare the warehouse position with the purpose of calling correctly *Check_package()*, is *Position_CP()*. Differently from *Load_Unload()* and *Check_package(),* it has been implemented as a function. The peculiarity of this function is the possibility of selecting the *mode*, indeed if the Mode field is set on TRUE, then the z position is set on *below*, otherwise is set on *above*. This depends on the configuration required, because as it has been shown in sensor configuration section, Z position for each cell can be *below* or *above*.

Here below is presented the structure of the *Position_CP()* function:

1. **Position_CP(**x: INT,z: INT, pos: ARRAY [1..3] OF INT,mode: BOOL**): BOOL**
    - **X:** cell x-coordinate
    - **Z:** cell z-coordinate
    - **Pos:** actual position of the cart
    - **Mode:** default TRUE, sets **z** on *below*, if FALSE sets **z** on *above*
      Returns TRUE once the operation is completed.
      Propedeutic function for the correct usage of *Check_package()*.

6. Alarm Implementation

This section motivation derives from a practical use case. One operator in the logistic domain, working with a true automatic warehouse has been interviewed. Hence, from the empirical needs of true users, some functions have been derived, which have multiple goals:

- Implementing safety.
- Highlighting faults.
- Preventing expiration of products stored for an excessively long time in the warehouse.

6.1. Safety analysis:

Safety is fundamental in a work environment and for this reason, a whole section is devoted to this topic (Ericson, 2005).

The risks that may arise in a real warehouse, characterized by the same configuration of the model present in the laboratory, have been studied.

This analysis allows to study the possible hazards of the system to reduce their probability of occurrence and/or their effect, and how to define proper countermeasures.

The preliminary step is to do a Functional Analysis to identify the functions that the system must perform, independently from how they are implemented. The representation by means of function branches shown in helps to clarify which are the main operations and represents the links between functions and subfunction. Note that the Automated storage and retrieval of boxes is the primary function.

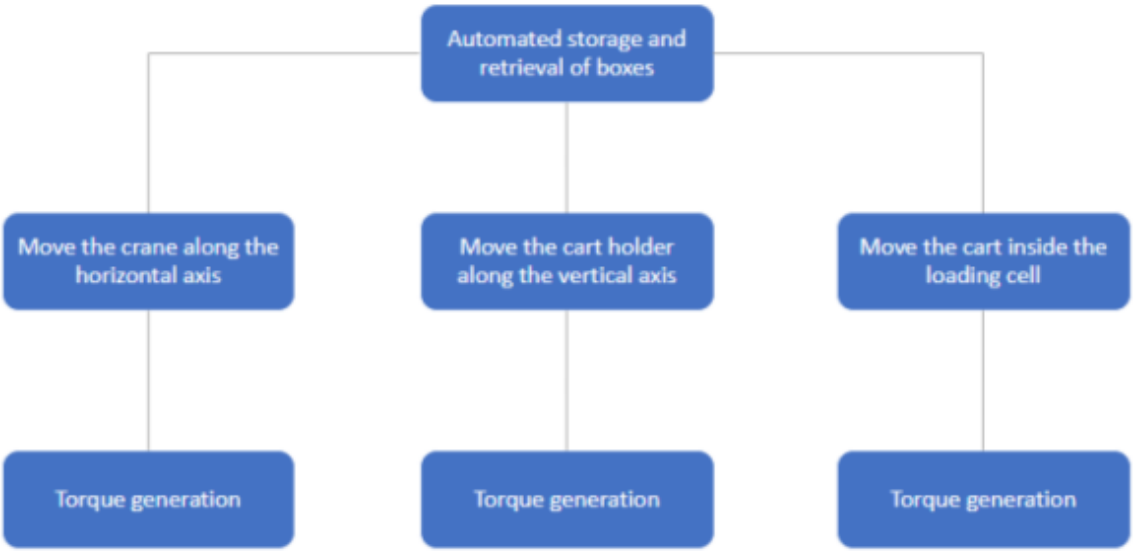


Figure 12 Functional analysis

After that, the operating conditions have been identified: in this case it is necessary to analyse only the package handling.

The next step is to do the Preliminary Hazard Analysis (PHA). It is a deductive approach, even if qualitative in nature, to study system safety because it starts from the system hazards caused by a loss of functionality.

The PHA has different advantages:

1. Allows determining most hazards.
2. Allows determining the associated risk.
3. Reduce severity and/or probability of critical hazards.
4. Reduces the project design time.
5. It is a great starting point for more detailed analysis.

The objectives of PHA are:

- Describe the hazards in the system, their origin and the possible consequences.

OPERATING MODE	SOURCE	PHENOMENA	EFFECT
Package handling	Malfunction of the motors	Collision	Damage of main structure. Damage of rack structure. Damage of product.
Package handling	Malfunction of the sensors	Collision	Damage of main structure. Damage of rack structure. Damage of product.

Table 2 Hazard description

- Define the consequences and risks associated with the hazards and possible related accidents. The Hazard's severity and the probability have been supposed.

  The following components were analysed separately: Rack structure, Main structure and Product.

TARGET	RACK STRUCTURE (R)
SEVERITY OF CONSEQUENCES	
CATASTROPHIC	Total breakdown
CRITICAL	Permanent deformations
MARGINAL	Slight deterioration
NEGLIGIBLE	Superficial scratch

Table 3 Hazard's severity of the rack structure

TARGET	MAIN STRUCTURE (M)
SEVERITY OF CONSEQUENCES	
CATASTROPHIC	Total breakdown
CRITICAL	Permanent deformations
MARGINAL	Slight deterioration
NEGLIGIBLE	Superficial scratch

Table 4 Hazard's severity of the main structure

TARGET	PRODUCT (P)
SEVERITY OF CONSEQUENCES	
CATASTROPHIC	Total damage
CRITICAL	Functional or physical alterations
MARGINAL	Appearance alterations
NEGLIGIBLE	No-effects collisions

Table 5 Hazard's severity of the product

After that, a Risk assessment matrix has been computed for each component to set a risk code for each hazard.

SEVERITY OF CONSEQUENCES	PROBABILITY OF MISHAP				
	E - IMPROBABLE	D - REMOTE	C - OCCASIONAL	B - PROBABLE	A - FREQUENT
I - CATASTROPHIC	2	3	3	3	3
II - CRITICAL	2	2	3	3	3
III - MARGINAL	1	1	2	2	2
IV - NEGLIGIBLE	1	1	1	1	1

Table 6 Rank structure risk assessment matrix

SEVERITY OF CONSEQUENCES	PROBABILITY OF MISHAP				
	E - IMPROBABLE	D - REMOTE	C - OCCASIONAL	B - PROBABLE	A - FREQUENT
I - CATASTROPHIC	2	2	3	3	3
II - CRITICAL	1	2	3	3	3
III - MARGINAL	1	1	1	2	2
IV - NEGLIGIBLE	1	1	1	1	2

Table 7 Main structure risk assessment matrix

SEVERITY OF CONSEQUENCES	PROBABILITY OF MISHAP				
	E - IMPROBABLE	D - REMOTE	C - OCCASIONAL	B - PROBABLE	A - FREQUENT
I - CATASTROPHIC	2	2	3	3	3
II - CRITICAL	1	2	2	3	3
III - MARGINAL	1	1	2	2	2
IV - NEGLIGIBLE	1	1	1	1	2

Table 8 Product risk assessment matrix

Legend: 3 = Imperative to suppress risk to lower levels; 2 = Operation requires written. Time-limited waiver, endorsed by management; 1 = operation permissible.

- Determine the corrective actions to modify, control or eliminate the dangerous situations.

Probability interval: 10 years	RISK BEFORE				COUNTERMEASURES	RISK AFTER		
HAZARD	Target	Severity	Probability	Risk Code		Severity	Probability	Risk Code
STRUCTURAL COLLISION BETWEEN THE MAIN STRUCTURE AND THE INTERNAL RANK STRUCTURE	R	II	D	2	While Yposin sensor is active, avoid X axis movement and time Z axis movement. Add motor alarms. Reduce motor speed. Add acoustic alarm.	III	E	1
	M	II	D	2		II	E	1
	P	I	B	3		II	D	2
STRUCTURAL COLLISION BETWEEN THE MAIN STRUCTURE AND THE EXTERNAL RANK STRUCTURE	R	II	D	2	Add motor alarm on Y axis motor. Reduce motor speed. Add acoustic alarm.	III	E	1
	M	II	D	2		II	E	1
	P	III	D	3		IV	E	1
COLLISION BETWEEN THE MAIN STRUCTURE AND THE PACKAGE	R	II	D	2	Increase the "feeder occupied" sensor sensitivity. Add sensors to check the presence of the package in the cells. "Check-package" function. Add acoustic alarm.	II	E	1
	M	II	D	2		II	E	1
	P	I	C	3		I	E	2
Motor overheating	M	II	C	3	Change Motor. Add motor alarms. Add acoustic alarm.	II	E	1
	P	II	C	2		II	E	1

Table 9 PHA

Thanks to this analysis, alarm/safety functions have been implemented on the PLC:

- ***LedBlinkingMotor.***
- ***LedBlinking.***
- ***Check_package.***
- ***Total_Check.***

The PHA is only a preliminary analysis and therefore it has some limits:

- It is difficult to consider all the hazards in the system.
- The description of the system failures and related consequences is not very precise yet and hence not reliable.
- It does not consider multiple failures and their combined effect on the system.

To further assess the safety of the warehouse, more in-depth analysis should be made such as: FMEA, FTA, RBD, ET…

6.2. LedBlinkingMotor()

The highest priority issue is for sure safety; hence the *LedBlinkingMotor()* program has been developed. It has been designed with the purpose of highlighting the presence of some motors for an excessive amount of time in the same condition of movement. When a motor is turned on, a timer corresponding to the configuration being assumed is fired. Whenever the motor changes its configuration, meaning changes direction of movement or is switched off, then the timer is reset. If the timer is not switched off, means that the motor is remaining in the current set-up. When the timer expires, means that the time is up and the fact of the motor remaining in the same position for an excessively large time is assumed to be harmful, therefore it is raised an alarm. On the *Monitoring Screen,* presented previously*,* a led is switched on. A practical example of the above mentioned dangerous behaviour, is the cart being stuck in an incorrect position, where no positioning sensors are able to detect the cart, this condition may be reached after a wrong sequence of commands, however is important to detect this failure in order to both prevent damages to the plant, indeed the continuous working of the motor may lead to overheating of the electrical moto and wearing of mechanical components, but also may put in danger operators working nearby the cart.

Here is shown the structure of the program.

```
LedBlinkingMotor(): XRSignal: BOOL:= FALSE; XLSignal: BOOL:= FALSE; YISignal: BOOL:= FALSE; YOSignal: BOOL:= FALSE; ZUSignal: BOOL:= FALSE; ZDSignal: BOOL:= FALSE;

Alarm program evaluating if the motors are active for too long (default 10 s)
Motor_bool: default FALSE, becomes TRUE if the motor configuration has not changed from the starting of the timer.
The anomaly is then shown in the interface.
```

6.3. LedBlinking()

A similar function, being implemented analogously is LedBlinking(). It has been implemented as its twin LedBlinkingMotor() as a program. It has the purpose of monitoring if the cart stays still in the same position for an excessively long amount of time. If the cart is not moving while the automatic warehouse is active, means that it is not managing the warehouse correctly. This phenomenon may be caused by a variety of reasons and one of them, experienced by operators in the sector, is reported in the following. Let us consider that the user may define a long task to be managed by the warehouse, that for example may take several minutes. Perhaps, in the meanwhile, the operator proceeds in accomplishing other jobs. However, while the operator is performing some other tasks, the warehouse has eventually fulfilled the original task. Then, the

warehouse stays still until it receives a new order, which may take some time since the operator is engaged. Here it comes the purpose of LedBlinking(), indeed, if the cart stays still for an excessively long time, an alarm is raised on the Screen. Differently from the previous program, this is not safety critical, because the idle state is not dangerous. The warehouse is simply waiting for some new commands. Here is shown the structure of the program. This function, however, also performs a control function because it can allow to identify a sensor malfunction.

LedBlinking(): sensors_bool:BOOL
Alarm program evaluating if the cart is idle for too long (default 10 s) in a different position from the initial one.
sensors_bool: default FALSE, becomes TRUE if at least one sensor detects that the position has not changed from the starting of the timer.
The anomaly is then shown in the interface.

6.4. LedBlinking_CFC()

In order to further develop our understanding of the potential given by this software, as previously mentioned in chapter 4.4 we developed and integrated the LedBlinking function using the CFC language.

The CFC (Continuous Function Chart) language is not one of the standardized ones described in Part 2 of (International Electrotechnical Commission), it is a superset of the FBD (Function Block Diagram) language and has several advantages: it is remarkably easy to understand and it allows to program feedback loops without interim variables, in case of an increase in complexity it is possible to split the code into smaller, more manageable blocks thanks to its hierarchical design capabilities.
It's also worth noting that its structure is very similar to Simulink, a well-known graphical language.

The LedBlinking function is surprisingly compact and straightforward to develop in CFC language, see Appendix . Thanks to the pre-built block library it was not even necessary to code the timer block. The code can be simplified even more by assigning a parameter to the timer .PT value as we choose the same one for all the sensors.

This language is way more powerful than the ladder diagram previously tested, though its flexibility cannot be compared to the structured text it clearly shows the reasons behind the industrial relevance of graphical languages.
It must be noted that all the tested languages interacted flawlessly one another thanks to the standardized approach by which they were created.

6.5. Alarm Configuration

In this section the functionalities concerning Alarm System Configuration offered by ABB Automation Builder software are exploited. The goal of this section is to introduce the reader to the way how the above-mentioned programs *LedBlinking()* and *LedBlinkingMotor()* programs have been merged with the rich architecture offered by the software platform. The alarm system allows to detect critical process states, to record them and to visualize them for the user with the aid of a visualization element. The alarm handling can be done directly in Automation Builder or alternatively in the PLC.
The configured alarms are natural extension of the programs previously presented. Their scope is briefly recalled to the reader: from the need of monitoring the status of both motor and cart position, two alarm

programs have been developed. The first measures how much time is spent by a motor remaining in the same configuration (for example the motor of the x-axis being turned on towards right), if an excessively large amount of time passes, then an alarm is triggered. This has high importance because of safety issues, if the motor is turned on for a large amount of time might be clue of the cart being stuck in some condition, it might lead to motor overheating and damages. The second alarm system measures if the cart remains for an excessively large amount of time in the same position which is not the initial one, this might represent the fact that the cart is stuck in some specific position or that the warehouse is not receiving any command by the user, leading to wastefulness of resources.
After this introduction, it is now shown how the configuration has been set.
First, the procedure requires the introduction of the desired alarm names and type. The used ones are summarized in the following figure.

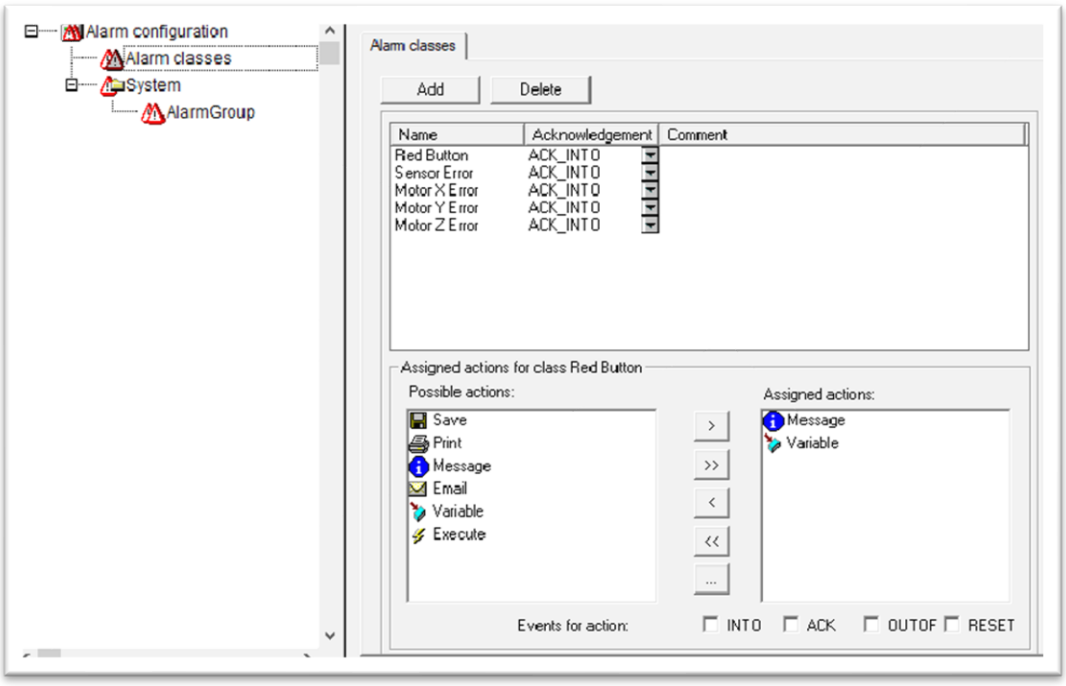


Figure 13 Alarm Configuration 1

The alarm name coincides with the definition of *Alarm Classes*. As can be seen, the used classes are:

1. **Red Button**: highest priority alarm, it is triggered manually by the operator. Once fired, it immediately stops any operation.
2. **Sensor Error**: connected to the output of the program *LedBlinking()*, it alerts when the cart remains in the same position for an excessively large amount of time.
3. **Motor X Error**: alerts if the motor of the X-axis remains in the same ON configuration for an excessively large amount of time.
4. **Motor Y Error**: alerts if the motor of the Y-axis remains in the same ON configuration for an excessively large amount of time.
5. **Motor Z Error**: alerts if the motor of the Z-axis remains in the same ON configuration for an excessively large amount of time.

The acknowledgment type of the alarms generated has chosen to be ACK_INTO. This type, according to the official documentation, corresponds to an alarm which must be acknowledged by the user. Now, looking at the lower part of the figure above, it can be easily seen the set of possible actions assigned to a certain alarm class. They correspond to the set of reactions the system generates once the alarm fires. In our implementation it has been used *message* and *variable assignment*, however, there exist other

functionalities such as e-mail sending or saving of the alarm record. The successive step of the alarm configuration is the assignment of variables to which have been associated the alarm classes previously. This is needed to interact with PLC variables to trigger the executions of the alarms.

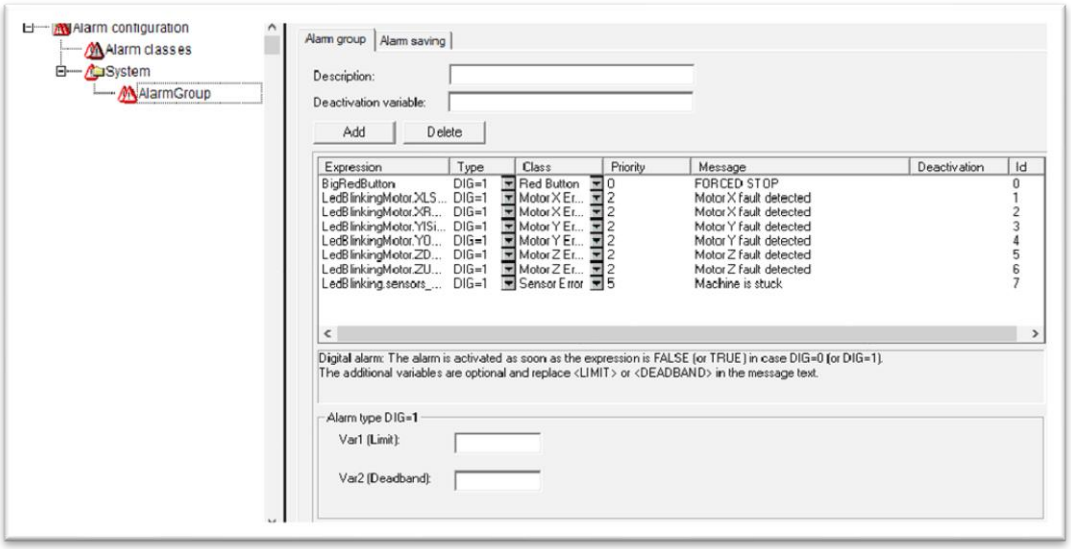


Figure 14 Alarm Configuration 2

In the above figure, can be seen several aspects of the alarm implementation. The second column, *Type*, represents the type of the *digital alarm*. *DIG = 1* is the type of our defined alarms, in detail, it corresponds to the fact that the alarm is activated as soon as the expression is TRUE. The fourth column named *Priority* instead defines the priority level of the alarm. A lower number corresponds to a higher priority. Maximum priority is 0 and it has been assigned to the Red Button. Priority also defines the order of visualizations of alarms in case of multiple simultaneous alarm occurrence. In our implementation Motor Alarms has higher priority with respect to Sensor Alarms, this because motor faults are considered safety critical, differently from sensors. Finally, to collect the chronological occurrence of several faults, it has been developed a Fault Table, in which error messages generated by the alarm occurrences are collected. In the figure below it is shown an example of fault table content. Once the events generating the alarms have been solved is possible to reset alarm to initial state through the RESET command.

	Date	Time	Class	Message
0	24-05-2021	15:25:43	Motor X Error	Motor X fault detected
1	24-05-2021	15:24:37	Sensor Error	Machine is stuck
2	24-05-2021	15:24:05	Motor Z Error	Motor Z fault detected
3	24-05-2021	15:24:05	Motor Y Error	Motor Y fault detected
4	24-05-2021	15:24:05	Motor X Error	Motor X fault detected

Figure 15 Fault Table

6.6. expiringCell()

A third similar function is *expiringCell()*. It has been implemented as a program. The idea is of monitoring the amount of time spent by each package in the warehouse since it has been stored. Also, this function comes from the empirical user need of having a tool allowing to avoid having some packages stored in the warehouse for an excessively large amount of time. Let us suppose to have a package containing perishable material. Let us consider that the considered package has a low-level of priority and therefore is stored in a far away position within the warehouse. Let us furtherly assume that the content of the same package is already stored in other packages, which are stored in nearer position with respect to the Home position, therefore, any task requiring in picking-up the specific material according to a suitable optimization decision process, would never pick up the package considered. However, this might be a problem since the content of the package is perishable, therefore it is needed to consider the time spent in the warehouse for each package stored.

A propaedeutic software component that must be presented to better explain the functioning of the program at hand, is an ad-hoc defined data type: the *struct Cell_state**.***

It is defined as the following:

```
TYPE Cell_state :
STRUCT
      occupied:BOOL;
      id:INT;
      weight:INT;
      timerCell: TON;
      expired: BOOL:=FALSE;
END_STRUCT
END_TYPE
```

For *expiringCell() program* the interesting field of the presented struct is *timerCell*. The considered timer is triggered whenever the package is stored in the considered cell, information which is retrieved by looking at the field *occupied* of the struct.

expiringCell() checks continuously the struct of each cell, to evaluate the timer. If the timer of a specific cell is expired, then the corresponding field within the struct, *expired* is set to TRUE. This will have corresponding effect on the interface, in which will be highlighted those cells for which the field *expired* is TRUE. Below it is provided a snippet of the implementation in pseudocode.

```
Running over rows
running over columns
IF cell[row,column].occupied = TRUE THEN
cell[row,column].timerCell(ON); (*if occupied switch on the timer*)
END_IF
IF cell[row,column].occupied = FALSE THEN
```

```
cell[row,column].timerCell(OFF); (*if not occupied switch off the timer*)
cell[row,column].expired:=FALSE; (*package has been removed*)
END_IF
IF cell[row,column].occupied = TRUE AND  cell[row,column].timerCell→expired THEN
cell[row,column].expired:=TRUE; (*package is expired*)
END_IF
END_FOR
END_FOR;
```

7. Advanced functions

7.1. State of the art

Now that basic functions are defined and safety measures are implemented, the team's objective is to evolve the basic warehouse functionalities by introducing new advanced functions. The idea behind these new functions is to not restrict ourselves to the case of the 9 packages warehouse model present in the laboratory, but, instead, to provide scalable solutions, which can be theoretically used also for larger systems. Hence, the logic of the applications is kept the most general possible.

The new applications and functions which will be described in the following paragraphs, require to move the packages from one cell to the other according to a certain logic (this is required for example in Reorganise() function block, described later). Before implementing the algorithms, a research was performed to see how the objectives of these functions are achieved in real warehouses.

According to logistics theory, when a function requires handling of multiple packages, it also involves the solution of a routing problem. This means that, if a sorting operation of the warehouse is performed (this is the purpose of the Reorganise() function), when moving packages from one cell to the other, the path followed by the cart to complete the operation should be minimized.

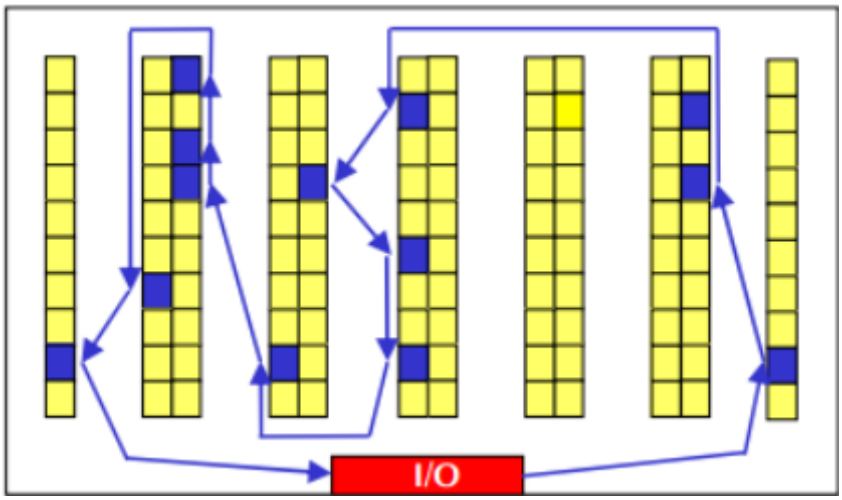


Figure 16 TSP applied to a warehouse

The cornerstone for this kind of routing problems is the "Travelling Salesman Problem" (TSP). The TSP is one of the basic case studies of computational complexity theory. The typical representation of this problem is finding the shortest path that a salesman should follow in order to visit all the cities where its clients are and then return to the starting city. This also represents the basic routing problem of modern automatic warehouses: start from a "home" state, visit a certain number of cells, perform the needed operation on each cell and then return to the "home" position. This problem is modelled through a graph G = (N, E), where N is the set of nodes and E is the set of edges of the graph; each node represents a cell, while each edge connects any two cells in the graph. Every cell is connected to all other cells. If the number of nodes is N, then the number of edges is $(N-1)!$. The nodes of the warehouse graph representation are shown in Figure 16, while, for the sake of clarity, only the edges from (2, 2) are being displayed. As mentioned before, every node can be reached from any other node.

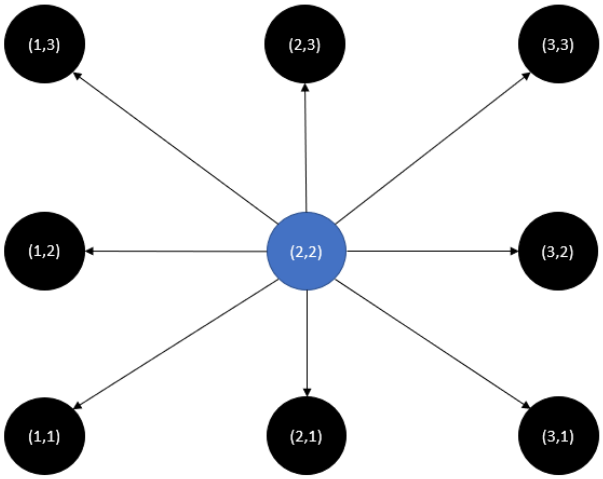


Figure 17 All the nodes in the warehouse graph and the edges from (2, 2)

Considering any two cells of the warehouse, for example (1,2) and (3,3), the time required to reach (1,2) from (3,3) is different from the one required to reach (3,3) from (1,2), because of the motors and the warehouse design. This means that the warehouse requires the solution of an asymmetric TSP. Hence, when traversing the warehouse graph, every edge has a well-defined direction.

Given this problem, its exact solution requires the enumeration of all possible paths to reach the desired node while passing through target nodes, and the selection of the most convenient one, according to the chosen cost metric. This type of solution has $O(N!)$ complexity, which means that it is unfeasible for most general warehouses, with a huge number of cells.

To effectively solve this problem, in a reasonable time, there can be three main strategies:

1. Restrict the problem to a particular case (like a subset of the original problem) where an optimal solution exists for sure.
2. Use heuristic algorithms, which are computationally more agile, but only return the most likely optimal solution.
3. Find the optimal exact solution, but only if the graph of the problem has a small number of nodes and the problem can be solved relatively quickly.

Several algorithms able to solve this problem in different ways and with different complexity exist in literature, like the Bellman-Held-Karp algorithm (based on the Bellmann optimality criterion), or the nearest neighbour algorithm (which is a heuristic algorithm, but only works well under very strict requirements).

Considering this brief overview of the TSP problems (many more algorithms for its solution exist and can be found in literature), the team decided to use an approach based on pre-defined policies in order to implement the functions which require complex routing. Writing and solving a TSP solution algorithm for a small warehouse model would be very resource consuming and pointless, since it could hardly be generalized for larger warehouse. Even if it could be generalized, the high number of cells in a real warehouse would increase the computational complexity of the algorithm way too much. Using pre – defined routing policies allows for better path stability and predictability, easier implementation and increases ease of implementation for storage policies, like those used in the Reorganise_batch() algorithm described below. It is to be noticed that, given the small size of the given warehouse model, not having optimal pathing for our function does not impact the warehouse performance in a significant way. This also makes pre-defined

policies the best choice for the considered model. A brief overview of the main characteristics of pre-defined policies versus the TSP problem can be found in the table below:

	Order by order	Pre-defined policies
	Choose best path for every order (use TSP)	Common routing logic applied for all orders, regardless of minimal path computation
Pros	Minimal path always achieved	Better routing stability (and predictability). Easily appliable to storage policies
Cons	Computational complexity exponentially grows with order size	Non – optimal path

Table 10 TSP vs pre-defined policies

After having implemented the basis movement functionalities of the warehouse and the standard safety measures, it has been decided to improve the above-mentioned functions to enrich the available toolbox. The goal of these functions is to increase the automation level of the warehouse user-side, such that he/she must only perform simple tasks e.g., load and unload packages on the cart. In this section several points will be covered. But first it will be discussed the assumptions that were taken into account for the rationale behind these functions.

7.2. Case study

In order to develop more realistic design choices, there were considered different kinds of possible customers, differentiated by their needs.

Based on the product variety:

- One product typology
- Low product variety (no doubles)
- High product variety

The fist category of customers will have only one product in their warehouses but it might have different property which will differentiate the stored lots, for example: expiration date, shipment origin and destination or environmental variables that might affect the quality.
The second customer will have different products to be stored, without the need to account for doubles of the same typology, this might lead to the easiest management logic since every lot could have a unique identifier, removing the need to differentiate between typology and multiplicity.
The last customer category is the chosen one. It will be considered a customer whose needs are of storing packages of different products, they might have doubles and the software may take into account also other properties in order to better place the lot in the warehouse.

Furthermore, it was defined a more detailed classification based on the loading procedure that the client might require.

1. Load batch of packages and reorganise the warehouse during shifts or night-time.

   Pros:

   - Packages can be placed immediately in the warehouse.
   - No queue.

   Cons:

   - During working hours the warehouse will be only partially optimized, this can be critical in case of an high flow of products.
   - Reorganising the warehouse might take more time than available.
   - Warehouse will need at least one free cell to use as a buffer.

2. Before loading the batch of packages the warehouse will be reorganised taking into account the optimal location of the new packages.

   Pros:

   - Warehouse will always be optimally sorted.

   Cons:

   - Long waiting times before loading the batch might lead to an increased queue in the meantime.
   - If there is a package with high priority in input almost all of the packages must be moved.

It was chosen the former management logic as it will allow for a more realistic and flexible scenario.

The customer will also need:

- A function to inform the user that a package is present in the warehouse from longer than a fixed amount of time dependant on the single lot.
- Each package will have a priority value, which might result from the optimization of a cost function based on the need of the specific customer, in the case here considered it will represent a simple time minimization.
- It will be present also a weight value that might be combined with the travel time in order to setup a basic extension of the optimization capabilities of the warehouse.
- The managed products might have a critical weight which will make them dangerous to be placed in the upper levels of the warehouse. The developed algorithms will impose a soft constraint on this factor by trying to place packages with similar priority on the lower available level before.

7.3. Total_Check()

The purpose of this function is verifying whether a package is present or not in each cell of the warehouse. This function is needed to provide robustness to the system. *Total_Check()* is a flexible function which is used to keep the status of our system updated and helps preventing unexpected behaviours. Since the only sensor to detect the presence of a package is fixed on the cart structure, we have no way of keeping track of the internal warehouse state in real time. For instance, if an operator were to manually unload a package, or some packages were manually moved during a fault state, when the warehouse is shut down, the user

interface would incorrectly display the occupancy state of some cell, due to the lack of internal cell sensor. By running the *Total_check() PRG* before complex automated procedures, like the one described in the following paragraphs, we ensure that the internal warehouse status stored in the memory of our program is correct, and that such operations will be carried out successfully, without unexpected behaviours.

To check every single cell of our warehouse, the same logic of *Check_cell() (FB)* was used: we use the cart sensor to see if a package is stored in a cell, if the output of FeederOccupied is True, the cell is marked as true, else the cell status is marked as false, and the ID field of the cell currently being checked is emptied. This procedure is iterated in every single cell: The presence check for the package must be carried out for each cell. This process can be time consuming, so a proper logic was implemented to ensure the minimal execution time is achieved: the cart starts from the default position after a reset command is issued and visits all cell in the column. The cart only moves form a cell to the adjacent one vertically. Once the whole column has been cleared, the cart is moved to the closest horizontal adjacent cell and repeats the check for all the cells in the column. The warehouse is therefore checked column by column, until all cells have been cleared and their status is updated in the global variables.

Total_check() is a PRG, so no input variable is required. It is executed if the Check warehouse is pressed in the HMI, which causes the main program to enter the case where *Total_check()* is located. After the whole program is executed, the reset command is issued, so that the warehouse returns to its default state and is ready to execute the next operation issued by the user.

7.4. Sort()

The function Sort() has the main goal of sorting an array of packages by priority.

Sort(Packages_array: array[1..3, 1..9] of int**): array[1..3,1..9] of int**

Inputs:

- Packages_array: matrix 3x9 containing the packages id, priority values and weights.

Outputs:

- **Sort:** matrix 3x9 containing the packages id, priority values and weights, sorted by priority.

This function is a fundamental block of other functions described later (see Gen_w_desired() and Gen_w_desired_batch()).

It takes as input a matrix with 3 rows and 9 columns. Each column j represents a package, denoted by an id number reported in (1,j), a priority value reported in (2,j) and a weight value reported in (3,j).

The input matrix has 9 columns, but some of them may be filled with zeros: it means that no package is reported in that column.

For example, it is needed to extract the three values in position (1,3), (2,3) and (3,3) of the input matrix to know the features of the package in column 3.

The function returns another matrix reporting the same information but ordered with respect to the priority values present in the input matrix.

Since the priority scale is descending (e.g. 9 = low priority, 1 = high priority), the fact that the output is ordered with decreasing priority means that the values on the second row of the output matrix will be ordered from the smaller one to the larger one. The information of each package is carried out in the ordered matrix. Columns filled with zeros are reported after the ordered ones.

For example, given the following input:

$$input\_matrix = \begin{bmatrix} 3 & 0 & 5 & 7 & 0 & 0 & 4 & 0 & 0 \\ 2 & 0 & 6 & 5 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

The output is:

$$output\_matrix = \begin{bmatrix} 4 & 3 & 7 & 5 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 5 & 6 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

7.5. Lower_empty_cell()

The function Lower_empty_cell() returns the X and Z coordinates of the empty cell characterized by lowest Z coordinate, if present. If the warehouse is completely full, this function returns an array of two elements equal to 0.

Lower_empty_cell: array[1..2] of int

Outputs:

- **Lower_empty_cell:** array of two elements reporting the X and Z coordinate of the empty cell with smallest Z coordinate.

If two or more cells are empty and with the same Z coordinate, the one with smallest X coordinate (thus the closest to the home position) is selected as output.

This function has been implemented with the intention of using its output in other applications: one may decide to load new packages in the empty cells of the warehouse with smaller Z coordinate (see Case study).

This function is not used in practice: a different logic is used to decide where to load packages in the warehouse (see Reorganise_batch()). In any case, this function returns an information which can be easily used if one decides to implement a different logic to load new packages.

7.6. Closer_empty_cell()

The purpose of Closer_empty_cell() is returning the closest empty cell with respect to a given one. It takes as input an array of three elements which identify one cell of the warehouse.

Closer_empty_cell(pose: array[1..3] of int**): array[1..2] of int**

Inputs:

- **Pose:** array composed by three integer numbers which identify a cell of the warehouse.

Outputs:

- **Closer_empty_cell:** array composed by two integer numbers which identify the closest empty cell with respect to the input cell.

Note that to identify one of the nine cells in the warehouse, two integers are needed: position along X axis and position along Z axis. The function Closer_empty_cell takes instead an array of three numbers as input for practical reasons: the function Current_pose() returns the position of the cart as an array of three elements, therefore its output can be easily used as input of this function.

The output of Closer_empty_cell is an array of two elements: position along X axis (integer number between 1 and 3) and position along Z axis (integer number between 1 and 3).

The returned cell can be used in other routines of the implemented algorithms, for example to perform loading or unloading operations.

Distance between cells is evaluated according to Pythagoras' theorem and evaluating the following rules:

- In case two empty cells are present with the same distance with respect to the input cell, one along X direction and the other one along Z direction, then the empty cell along X direction is considered the closest one.
- In case two empty cells are present with the same distance with respect to the input cell along the same X axis, the one characterized by smaller X coordinate is selected as the closest one.
- In case two empty cells are present with the same distance with respect to the input cell along the same Z axis, the one characterized by smaller Z coordinate is selected as the closest one.

In Figure 18 an example of the functioning of Closer_empty_cell() is shown: the input cell is the green one, the occupied cells are the blue ones, and the output cell is the orange one; in the figure three orange cells are shown numbered from the first choice to the third one, to represent which output the function would return if the cells characterized by smaller numbers were occupied.

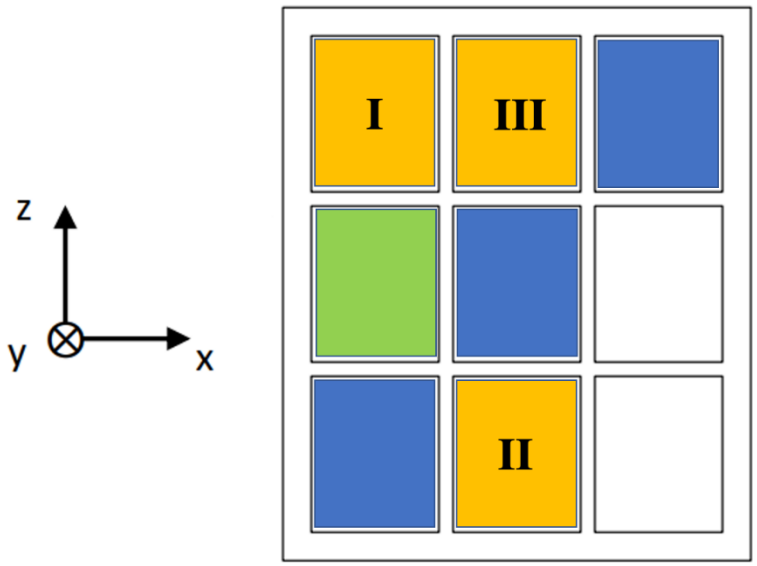


Figure 18 Description of Closer_empty_cell() functioning

7.7. Gen_w_desired()

Each package stored in the warehouse is provided with a priority value. Packages with higher priority need to be unloaded sooner than the ones with lower priority. Therefore, an optimized configuration of the warehouse is the one where packages with higher priority are stored closer to the home position, in order to make the time required to unload them shorter.

The purpose of Gen_w_desired() is creating an optimized warehouse configuration.

The function computes and returns the optimized warehouse state based on the actual packages stored in the warehouse.

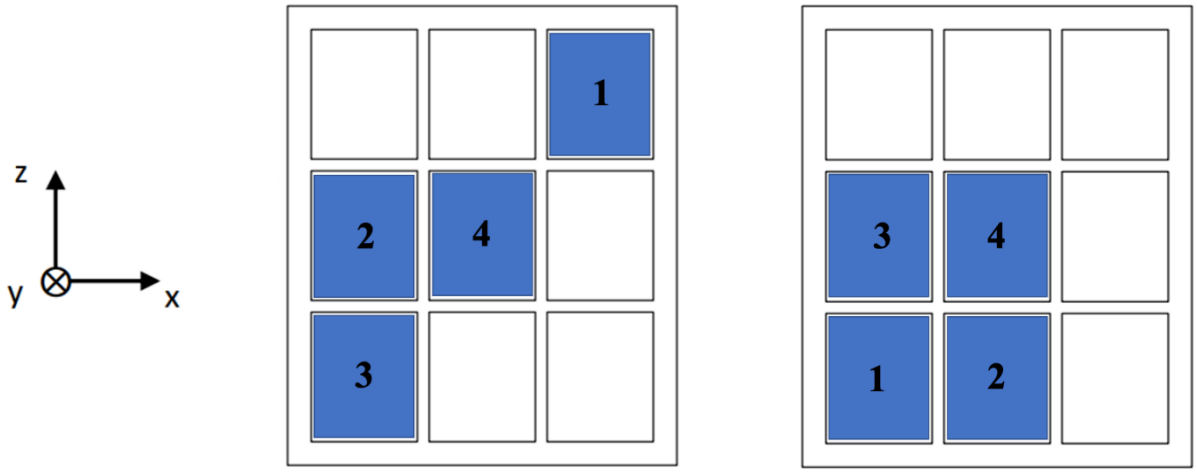
Gen_w_desired(): array[1..3,1..3] of Cell_state

Outputs:

- **W_desired:** 3x3 matrix representing the desired configuration of the warehouse, based on the packages stored.

The logic used to implement this objective features a preliminary part, where the packages stored in the warehouse are sorted with decreasing priority. This operation is achieved by calling the function Sort() giving as input an array containing the packages stored in the warehouse and obtaining as output an array containing the same packages but sorted by priority. Then, the desired state of the warehouse is obtained by placing the packages in the ordered array one by one in the closest position with respect to the home position.

In Figure 19 an example of the Gen_w_desired() functioning is reported: the actual state of the warehouse is shown in a), while in b) the output of the function shows how the packages should be stored in an optimized configuration.



a) *b)*

Figure 19 Description of Gen_w_desired() functioning

7.8. Reorganise()

As already mentioned, operations of loading new packages and unloading the stored ones happen during the day. The night-time can be exploited to reposition the packages in a strategic way, optimizing the unloading operations of the next day by making them shorter in time.

This objective has been implemented in Reorganise() function block, whose purpose is eventually rearranging the packages positions in the warehouse to prioritize some packages with respect to others.

Reorganise(W_state_desired: array[1..3,1..3] of Cell_state, pose: array[1..3] of int, Disable: bool**): bool**

Inputs:

- **W_state_desired:** 3x3 matrix representing the desired configuration of the warehouse.
- **Pose:** array composed by three integer numbers which identify a cell of the warehouse.
- **Disable:** reset the output of the function to FALSE once it has been executed.

Outputs:

- **Reorganise:** Boolean set to True when operation is completed.

Before calling this function block, a desired configuration of the warehouse must be computed, based on the packages actually stored in the cells (see function Gen_w_desired()). This desired configuration is an input of Reorganise() and prescribes where the packages should be in order to optimize the future unloading operations.

Once the Reorganise() function block is running, first of all the presence of an empty cell in the warehouse is checked: at least one empty cell used as a buffer is needed in order to perform the re-ordering of the packages. Therefore, if no empty cell is present, the function block terminates without performing any action. A message is displayed in the user interface to notify the user about the impossibility of performing a re-arrangement of the warehouse.

If an empty cell is present, it is possible to go on with the re-ordering operation. The actual state of each cell of the warehouse is compared with the desired one: if there is no mismatch between the two of them, no action is performed, otherwise the position of the packages is changed in order to achieve the desired state.

The whole re-ordering process ends when all the cell states correspond to the desired ones.

The algorithm used to implement the Reorganise() function block is shown in Figure 27.

7.9. Gen_w_desired_batch()

During the day new packages arrive and must be stored in the warehouse.

The packages, eventually grouped in a batch outside the warehouse, should be stored as soon as possible, in order to free the space where operators work.

The logic used to store the packages in the batch is not casual: the optimized configuration of the warehouse, depending on both the priority of the packages already stored and the priority of the ones waiting to be stored, can be exploited.

The purpose of Gen_w_desired_batch() is that of creating a desired state taking into account packages in the warehouse and in the batch. The implementation is the same as the one of Gen_w_desired(). The only difference is that packages present in the warehouse and also in the batch are sorted with decreasing priority order and then placed in the desired configuration.

`Gen_w_desired_batch(``Batch: array[1..9] of package`**`): array[1..3,1..3] of Cell_state`**

`Inputs:`

- **`Batch:`** `array of 9 elements containing the packages to be stored in the warehouse.`

`Outputs:`

- **`W_desired_batch:`** `3x3 matrix representing the desired configuration of the warehouse, based on the packages already stored and the ones in the batch.`

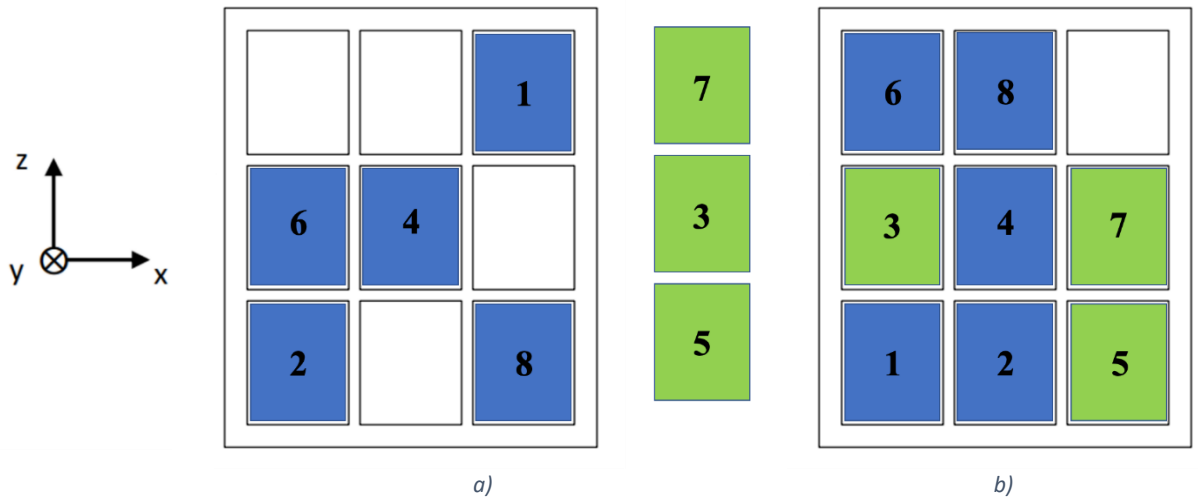


Figure 20 Description of Gen_w_desired_batch() functioning

In Figure 20 an example of the Gen_w_desired_batch() functioning is reported; blue packages are the ones present in the warehouse state, while the green ones are packages coming from a batch. The numbers in the cells represent the priority values of the packages. The actual state of the warehouse and the batch of packages is shown in (a), while in (b) the output of the function shows how the packages should be stored in an optimized configuration.

7.10. Reorganise_batch()

The function block Reorganise_batch() has the purpose of storing new packages in the warehouse according to a specific logic.

Reorganise_batch(Batch: array[1..9] of package, W_state_desired: array[1..3,1..3] of Cell_state, Pose: array[1..3] of int, Disable: bool**): bool**

Inputs:

- **Batch:** array of 9 elements containing the packages to be stored in the warehouse.
- **W_state_desired:** 3x3 matrix representing the desired configuration of the warehouse.
- **Pose:** array composed by three integer numbers which identify a cell of the warehouse.
- **Disable:** reset the output of the function to FALSE once it has been executed.

Outputs:

- **Reorganise_batch:** Boolean set to True when operation is completed.

For obvious reasons it is not possible storing new packages if the warehouse is already full. Therefore, in the following routine, before storing any package, the presence of an empty cell is always checked. In the case an empty cell is not present, the loading operation terminates, and a message is displayed in the user interface to notify the user about the impossibility to go on with the operations.

Each package of the batch is stored, if possible, in the position prescribed by the output of Gen_w_desired_batch(). Note that this is not always possible, because the desired state of the warehouse is computed depending on the packages already stored in the warehouse also. These packages may not be in the right position according to the desired one, hence they may occupy a cell assigned to a package in the batch. Therefore, if the package cannot be put in the desired cell, it is put in the closest cell possible with respect to the assigned one. This happens because, when the re-ordering operation of the warehouse is performed by night (see Reorganise()) two cases will arise:

1. The package which was previously in the batch is already positioned in the desired cell.
2. The package which was previously in the batch is not positioned in the desired cell but it is close to it (the distance between the desired position and the actual one also depends on the state of the warehouse when the package was stored: note that in an almost full warehouse it is more difficult to put the package close to desired position, while in an almost empty warehouse it is more likely that the package is close to the assigned position).

In Figure 21, the final configuration of the warehouse after packages from a batch are loaded is shown; blue cells are the ones already storing a package, while green cells are storing the packages just loaded from the batch. The number in each cell indicates the priority value of the package. This final configuration of the warehouse depends on the batch and the desired configuration shown in Figure 20.

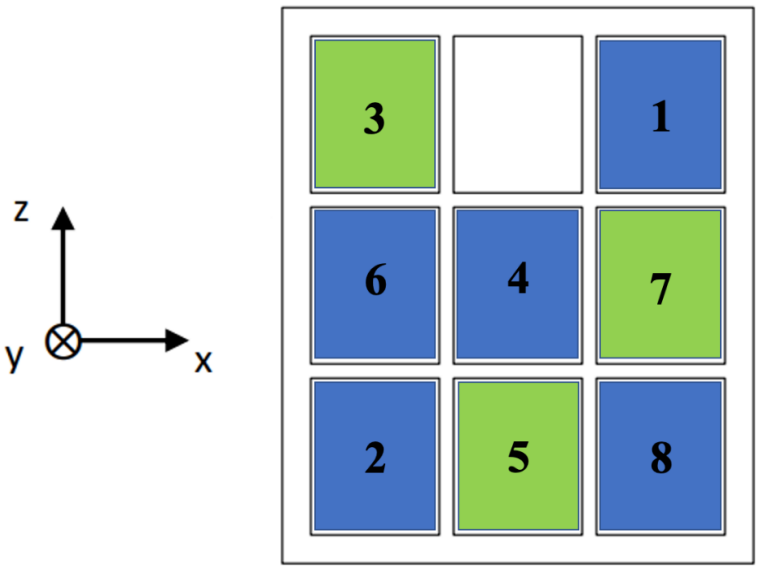


Figure 21 Description of Reorganise() functioning

7.11. Unload_priority()

When performing unloading operations, packages with higher priority are assumed to be the first ones the operator needs to take out from the warehouse.

Therefore, the function block Unload_priority() can be used to decide which package to unload, based on the priority of the stored packages.

`Unload_priority(``Priority: int, Pose: array[1..3] of int, Disable: bool`**`): bool`**

`Inputs:`

- **`Priority:`** `priority value of the package to be unloaded, it is an integer number.`
- **`Pose:`** `array composed by three integer numbers which identify a cell of the warehouse.`
- **`Disable:`** `reset the output of the function to FALSE once it has been executed.`

`Outputs:`

- **`Unload_priority:`** `Boolean set to True when operation is completed.`

The user can insert an integer number, which is the priority value of the desired package (note that the priority value of each package stored in the warehouse can be visualized from the user interface). Then, three cases arise:

1. One package is present with the inserted priority value: that package is unloaded.
2. More than one package is present with the inserted priority value: the closer one with respect to the home position is unloaded.
3. No package is present with the inserted priority value: no action is performed.

8. Conclusions and Future Work

From the programming point of view, since this project represents the first experience in PLC programming it has been developed in just one programming language: Structured Text. Only secondly, we decided to try with other PLC programming languages, to gain more confidence with the available tools. We understand that many functions could be written more effectively with other languages, such as the ones considered in a second time. However, this laboratory represented a springboard to possible new PLC programming experiences, particularly in real working environments. Throughout the development of our project, we had the possibility to interact with a real user of an automated warehouse system, this gave us the chance to think about developing new functionalities based on the practical needs of the user interviewed. Thanks to this, more importance has been given to some aspects, which in practice are fundamental, which otherwise would be considered only as marginal. From this derives the awareness of the importance of understanding the real requirements of an application domain, which are unavoidable when programming a software in industrial deployment. In this laboratory experience we tackled against several issues due to the intrinsic nature of the problem at hand. In this course, differently from the *traditional ones*, there is a significant practical component. This involves some physical experience about the hardware which was not developed yet by the team. In detail in many situations, we were unable (at least initially) to recognize if the problems highlight were coming from hardware side or from the software one. These events guided us in awareness that the hardware component is of paramount importance in the overall functioning of the physical system. The choice of this project has been driven by the curiosity on this topic: PLC programming of automated systems. Being at the end of our study in Automation and Control Engineering, we thought that choosing this kind of project would be of great interest because we together recognized the importance of the application in industrial environments, moreover many of us did not experienced before anything similar. As possible future developments we highlight the possibility of changing the modeling representation used in design phase.

In this project Finite State Automata have been used, however they have the intrinsic limitation of not being able in modeling timing constraints. To provide a representation of the automated warehouse such that also the temporal component is kept into account, we suggest selecting a different modeling technique which is the Timed Petri Nets. Thanks to this choice, the developer would be able in defining temporal constraints in the executions of the actions of the warehouse, also helping the possible more detailed development of optimization algorithms. For what concerns the development of advanced functions, the dependency on the client choice is heavy. It affects the logic with which programming has been developed. A different client definition would cause in having different choices, for example, if the client is a food store, then it must for sure considered the conservation issues of the packages, differently from the case of a standard Amazon-like client. However, programming has been developed according to modularization. This ensures that eventual new works, although deployed on different structures, must not be completely developed from scratch but can be re-used some core modules written in Structured Text. Considering advanced functions sub-modules here the most important are recalled *closer empty cell* (computing the nearest cell to the one considered), *lower empty cell* (computing the closest cell to home) and *sort* (re-ordering the packages according to a defined priority level). These three functions (and many others) can be re-used in other applications as discussed previously.

A further possible extension of the work might include the modelling of an expansion of the warehouse including more layers in height, in width or in depth, or the management of multiple warehouses simultaneously. This must be done according to more complex functions which are out of the scope of this work, however the basic programming logics developed can be re-used also in this case.

A useful feature for the operator, easy to implement, might be the ability to program macro-operations and to display the estimated time of arrival of the global and partial operations. This time could be computed by knowing more parameters of the control action (such as motor velocity) or estimated by means of a suitable

algorithm.
Finally, in the last section of the conclusion, we show a possible future development of the safety analysis section.
As already described in the section dedicated to safety analysis, different techniques can be used to identify and assess the various failures (Ericson, 2005). Some of the most common and complete ones are:

- FMEA (Failure Modes and Effects Analysis) is an inductive method for studying in systematic way the failure modes of the system components, their causes, and their effects on the system itself.

  Advantages:

  1. The comprehensive approach allows considering all possible component failures in all the operating modes.
  2. The analysis allows specifying the criticality of the failure modes, the possible need for recovery actions and their effect.
  3. Very useful at the beginning of the safety analysis work (even if it should come after a PHA).
  4. Facilitates the cooperation between the project designer and the safety analyst.
  5. The comprehensive method leads to a deep knowledge of the system, components, and their interaction.

- FT (Fault Trees) is a deductive method to identify the possible combinations of events that produce failure and then continue the analysis up to identification of the failure modes of the components starting from a failure mode of the system.

  Advantages:

  1. Possible to take account of logical relations.
  2. Allow a probabilistic analysis.
  3. Allow immediate propagation of the top event severity to the basic events. This will identify the most critical basic events.
  4. Provide a clear and self-explanatory documentation on the propagation of faults in the system.

If you want to create a warehouse that reflects the structure of the model, it is suggested to use both techniques in a complementary way because they capture different aspects of this problem, and this involves better identification and prevention of failures which is fundamental for a warehouse.

Concluding, the project has been carried out effectively and all the predefined goals have been achieved. The team is satisfied of the accomplished results. Practical proofs have been obtained by recordings on the warehouse.

Appendix 1

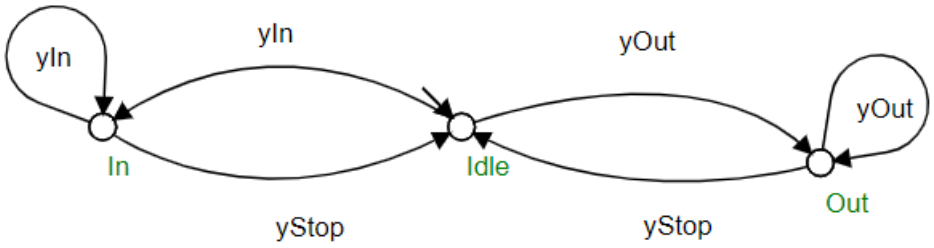


Figure 22 MotorY automaton

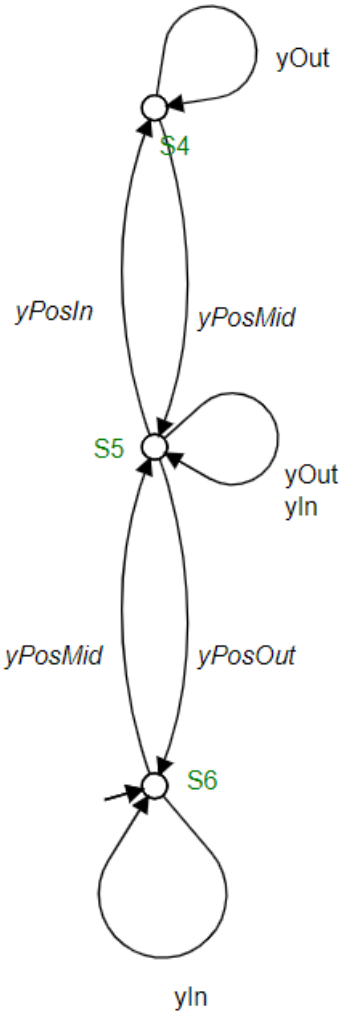


Figure 23 SensorY automaton

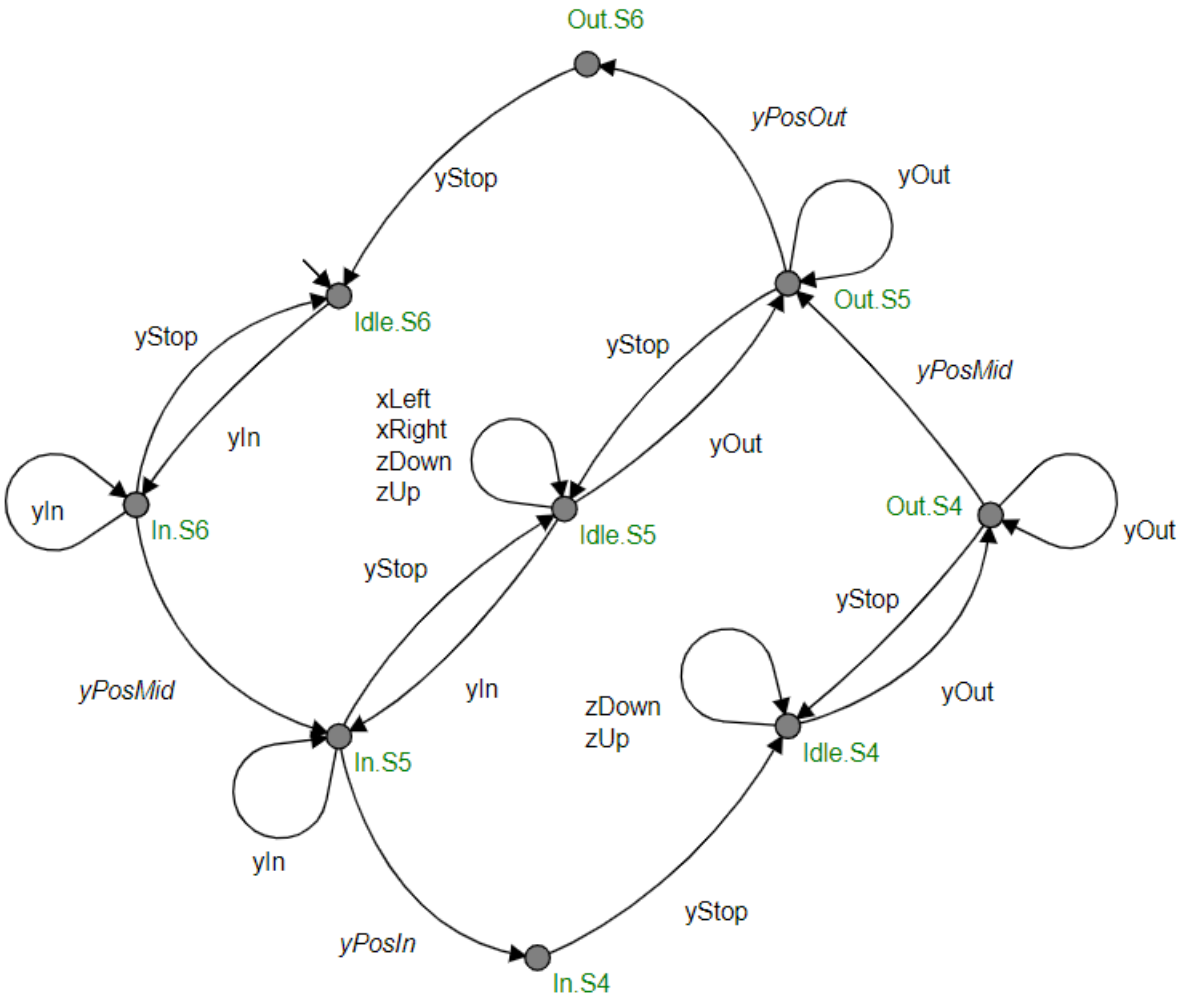


Figure 24 MotorY || SensorY automaton

Appendix 2

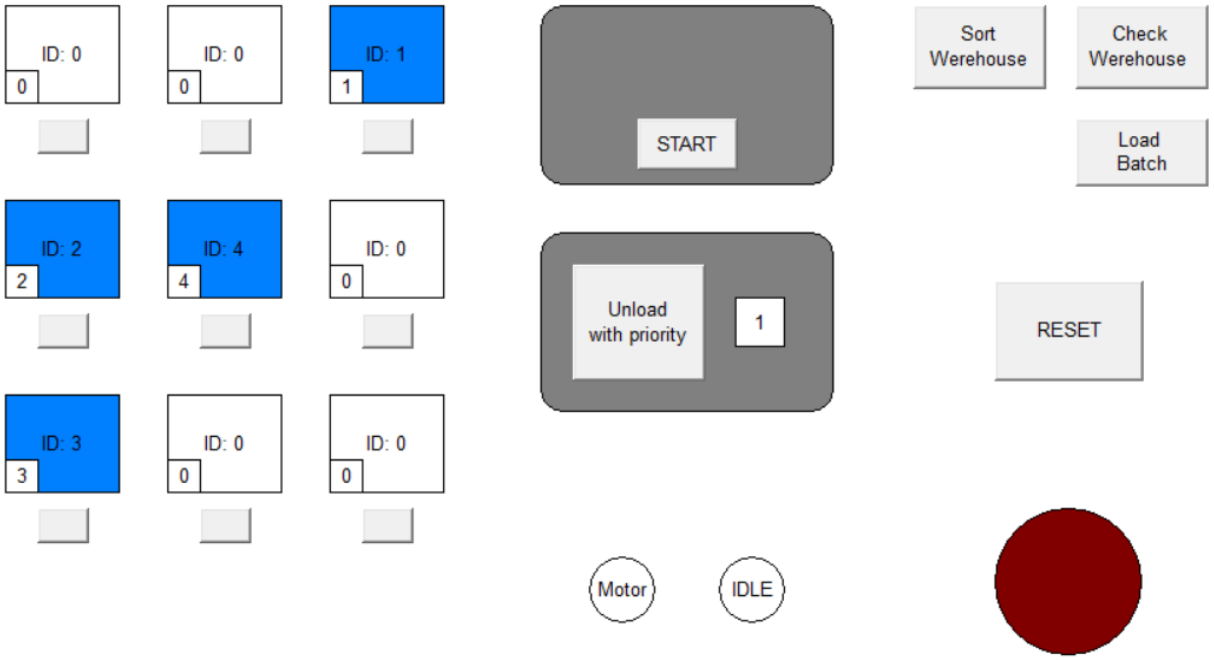


Figure 25 HMI interface while running

Appendix 3

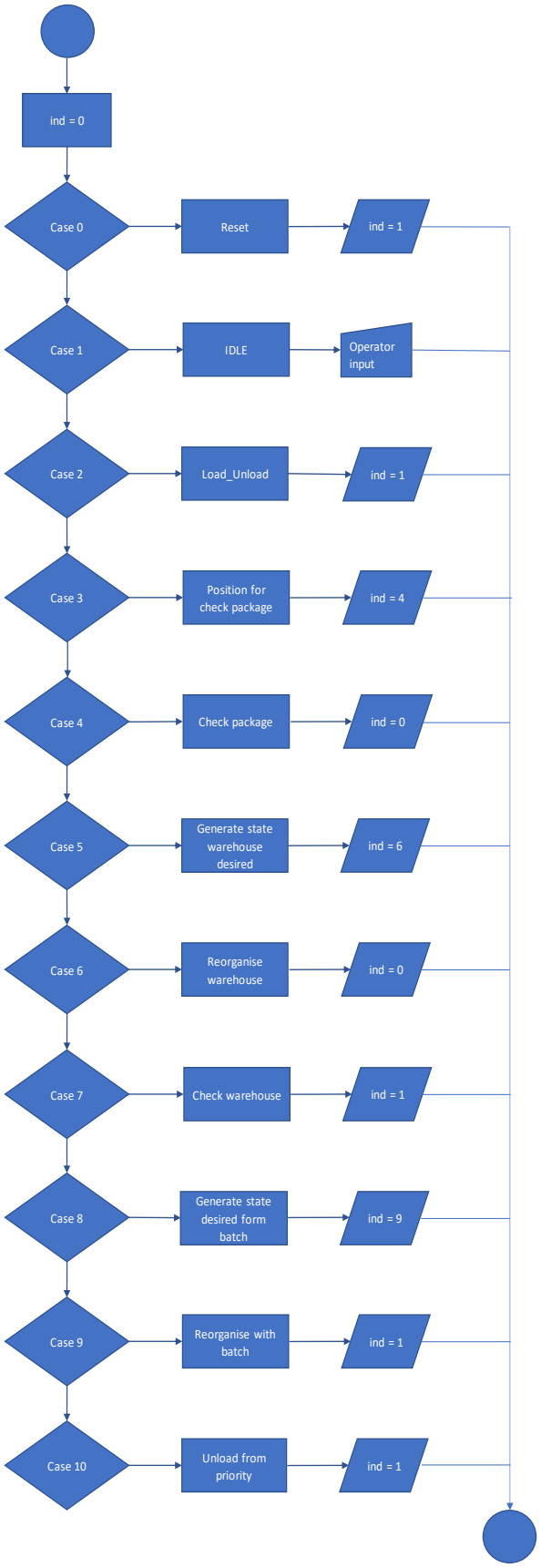


Figure 26 Flowchart of the code used to select a functionality

Appendix 4

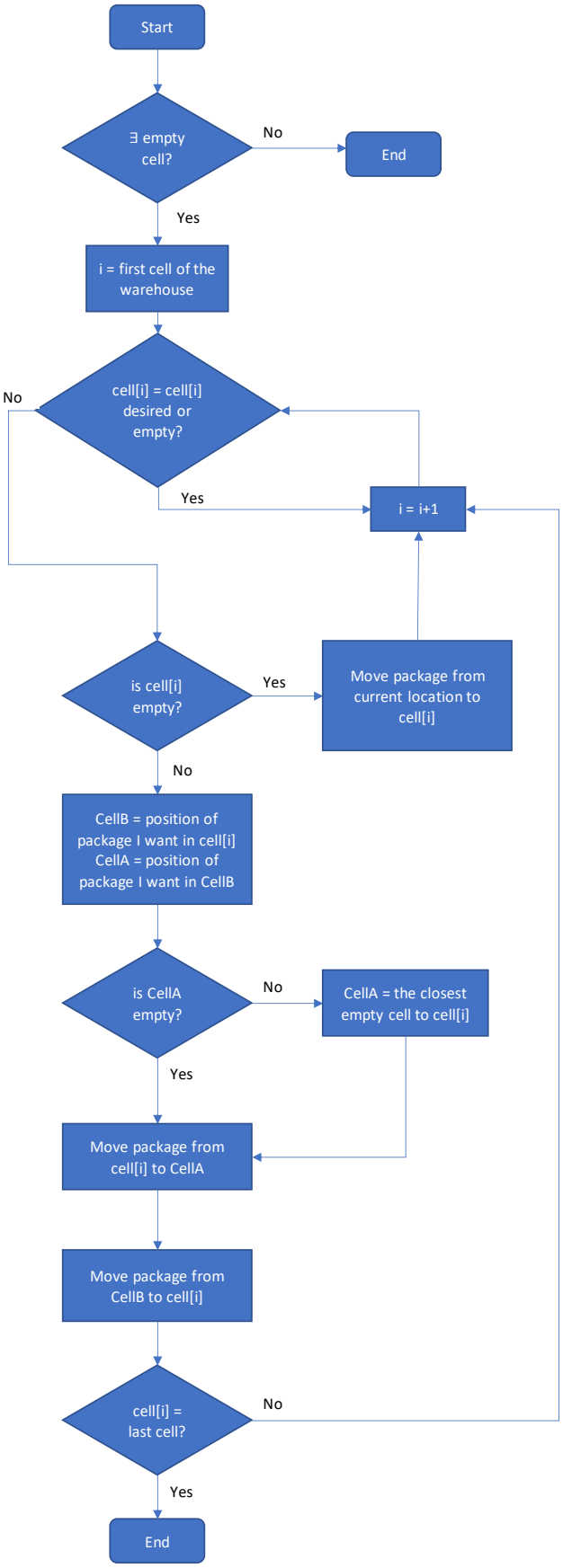


Figure 27 Flowchart of Reorganise()

Appendix 5

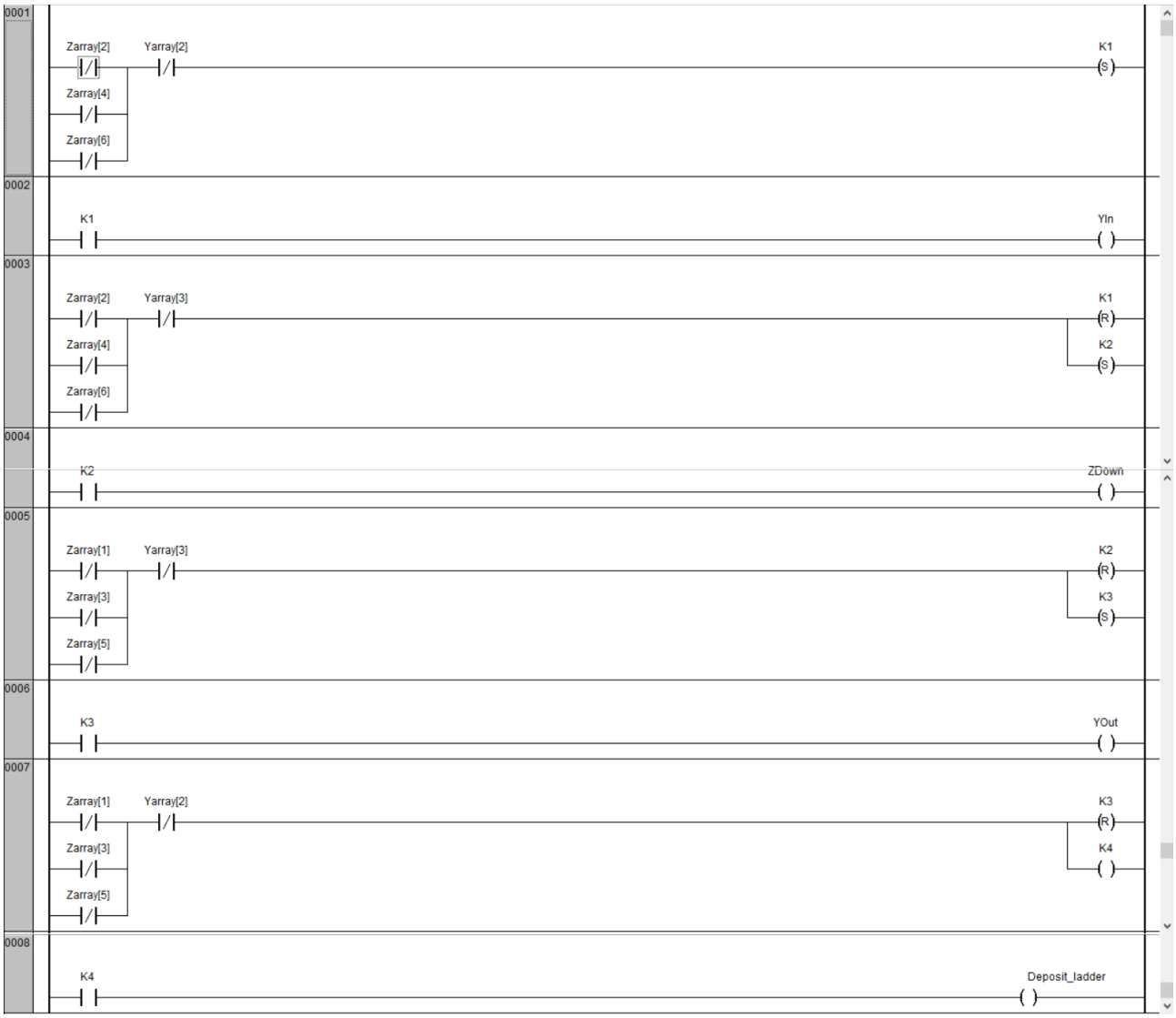


Figure 28 Deposit_ladder()

Appendix 6

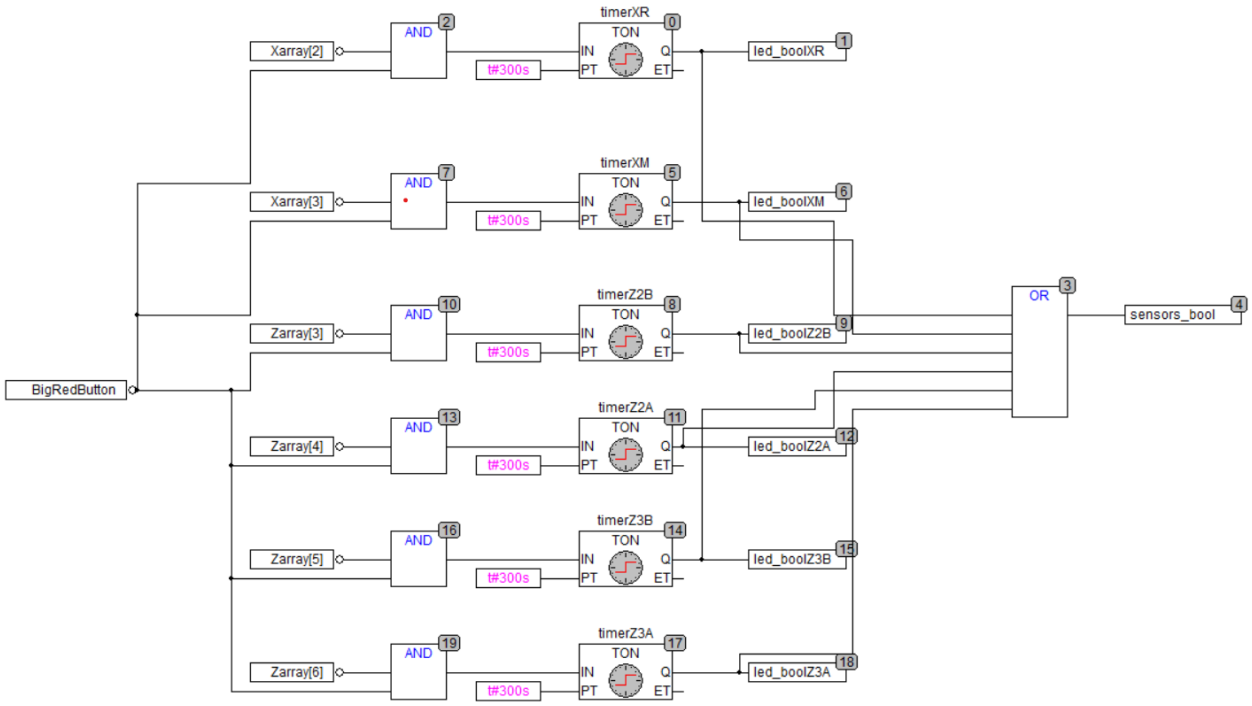


Figure 29 LedBlinking_CFC()

List of Figures

List of Tables

Bibliography

ABB Group. (n.d.). *INSTRUCTIONS FOR USE - PLC Automation - AC500 Automation Builder 2.4.0 AC500 V2.* Retrieved from https://global.abb/group/en

Ericson, C. (2005). *Hazard Analysis Techniques for System Safety.* Wiley Interscience.

International Electrotechnical Commission. (n.d.). *IEC 61131.*

L.H. Chiang, E. R. (2001). *Fault detection and diagnosis in industrial systems .* Springer.