Politecnico di Milano

Automation and Control Engineering

Automation of an Elevator Plant

Automation and Control Laboratory

Academic Year: 2018 - 2019

Professors:

Facchinetti Alan

Tarsitano Davide

Students:

Maicol Dolci 900035

Riccardo Giani 914931

Eleonora Mercalli 898889

Alessandro Perinetti 898890

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Task definition

This project aims to realize a control system for a real plant based on discrete events. In our case it is an elevator plant. This report starts by explaining the basic functions implemented on it and then, step by step, the difficulty of the work will increase.

The control system is based on a PLC coded through CoDeSys, which is the software that was used to build the program.

The first task to deal with is the single call management of the elevator. The increasing difficulty consists in the addition of different features, such as the storage of multiple calls and its relative management.
An optimization approach was used with the purpose of minimizing the path covered by the elevator depending on the calls list.

Moreover it was built a fault detection system able to observe possible faults during the execution of the plant.
In conclusion, additional features were added such as a weight detection sensor and a HMI interface to directly control the system through an LCD display.

1.2 Work organization

To carry out the project, we started from the basic function of an elevator: move from floor i to floor j. Once this task was performed, a further step leading to an increase of complexity was taken.

To define the solution of the problems a «Waterfall model» was followed. The Waterfall model is a sequential approach, where each fundamental activity of a process is represented as a separate phase, arranged in linear order.

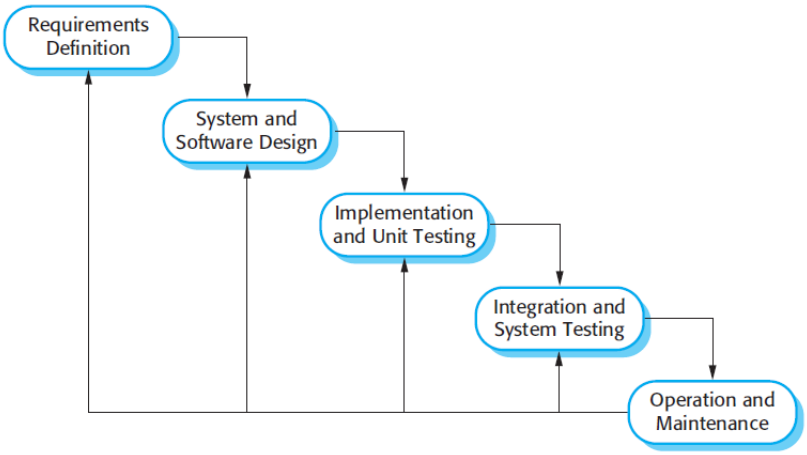


Figure 1.1: Waterfall model

The phases of the waterfall model are: Requirements, Modeling, Implementation, Testing and Maintenance. All of the activities must be planned and scheduled before starting working on them.
The software development is not a simple linear process, but it involves feedback from one phase to another one. The clear definition of the requirements acquires a fundamental role in the implementation of the model with the purpose to avoid ambiguous situations.

The definition of macro-functions helped to speed the debugging process up when some errors were detected.

Chapter 2

Materials and software

2.1 The elevator: I/O

General overview

A didactic elevator was used to carry out our project. The model is «Didactic lift ASC89» and its data-sheet will be shown in Appendix A.

Figure 2.1: Didactic elevator

In order to make the elevator work properly, a grid supply and a PLC system are required. The model of our PLC is PM571, provided by ABB.

The elevator is composed by a cabin that can be moved among four floors thanks to an electric motor located on the top of the structure. At each floor a door is present and it can be opened or closed through an input given at the related electric motor.

Figure 2.2: Cabin panel

The panel that simulates the interior of the cabin has five buttons, four to choose the desired floor and one to stop the movement of the elevator.
Moreover, it is present a lever that can be switched to simulate the presence of an obstacle during the closure of the door. This lever operates on each door of the plant.

At the ground floor and at the 3rd one, it is possible to make just one type of call: for the former "go upwards" while for the latter "go downwards", since they are the extremities of our model. At the 1st and 2nd floor, both choices are available.

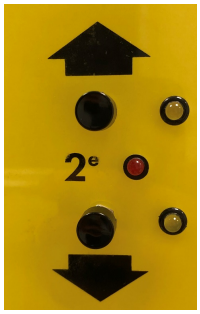


Figure 2.3: Floor buttons

In the system many optical sensors can be found. Some of them are used to check the presence of the cabin at the floor, others to verify that the doors are completely closed or opened.
The system is also provided with LEDs. The lights, that define if the elevator is present at the floor, are placed respectively on the cabin panel and at the side of the related number of floor. Moreover for each external call button there is a LED that turns on every time a request is made using it.

In order to have an easy overview of the inputs and outputs of the system, two summary tables with a brief explanation are shown below:

OUTPUTS	
Name	**Description**
CabGoUp	set *TRUE* to activate the engine that moves the cabin upwards
CabGoDown	set *TRUE* to activate the engine that moves the cabin downwards
CabLight	set *TRUE* to activate the light inside the cabin
CabLed0	set *TRUE* to define the presence of the elevator at floor 0
CabLed1	set *TRUE* to define the presence of the elevator at floor 1
CabLed2	set *TRUE* to define the presence of the elevator at floor 2
CabLed3	set *TRUE* to define the presence of the elevator at floor 3
Floor1LedDw	set *TRUE* to define that the button to go down at floor 1 has been pressed
Floor2LedDw	set *TRUE* to define that the button to go down at floor 2 has been pressed
Floor3LedDw	set *TRUE* to define that the button to go down at floor 3 has been pressed
Floor0LedUp	set *TRUE* to define that the button to go up at floor 0 has been pressed
Floor1LedUp	set *TRUE* to define that the button to go up at floor 1 has been pressed
Floor2LedUp	set *TRUE* to define that the button to go up at floor 2 has been pressed
Floor0DoorClose	set *TRUE* to activate the motor that closes the door at floor 0
Floor1DoorClose	set *TRUE* to activate the motor that closes the door at floor 1
Floor2DoorClose	set *TRUE* to activate the motor that closes the door at floor 2
Floor3DoorClose	set *TRUE* to activate the motor that closes the door at floor 3
Floor0DoorOpen	set *TRUE* to activate the motor that opens the door at floor 0
Floor1DoorOpen	set *TRUE* to activate the motor that opens the door at floor 1
Floor2DoorOpen	set *TRUE* to activate the motor that opens the door at floor 2
Floor3DoorOpen	set *TRUE* to activate the motor that opens the door at floor 3

Table 2.1: Output table

INPUTS	
Name	**Description**
CabPresence0	*TRUE* if the cabin is present at floor 0
CabPresence1	*TRUE* if the cabin is present at floor 1
CabPresence2	*TRUE* if the cabin is present at floor 2
CabPresence3	*TRUE* if the cabin is present at floor 3
CabCallP0	*TRUE* if the button 0 inside the cabin is pressed
CabCallP1	*TRUE* if the button 1 inside the cabin is pressed
CabCallP2	*TRUE* if the button 2 inside the cabin is pressed
CabCallP3	*TRUE* if the button 3 inside the cabin is pressed
CabObstacle	*TRUE* if the obstacle lever is switched on
CabStop	*TRUE* if the stop button inside the cabin is pressed
Floor0DoorClosed	*TRUE* if the door at floor 0 is closed
Floor1DoorClosed	*TRUE* if the door at floor 1 is closed
Floor2DoorClosed	*TRUE* if the door at floor 2 is closed
Floor3DoorClosed	*TRUE* if the door at floor 3 is closed
Floor0DoorOpened	*TRUE* if the door at floor 0 is opened
Floor1DoorOpened	*TRUE* if the door at floor 1 is opened
Floor2DoorOpened	*TRUE* if the door at floor 2 is opened
Floor3DoorOpened	*TRUE* if the door at floor 3 is opened
Floor0UpBt	*TRUE* if the up button at the floor 0 is pressed
Floor1UpBt	*TRUE* if the up button at the floor 1 is pressed
Floor2UpBt	*TRUE* if the up button at the floor 2 is pressed
Floor1DwBt	*TRUE* if the down button at the floor 1 is pressed
Floor2DwBt	*TRUE* if the down button at the floor 2 is pressed
Floor3DwBt	*TRUE* if the down button at the floor 3 is pressed

Table 2.2: Input table

Safety features

In order to avoid any uncontrolled situations, a safety control is implemented in the plant.

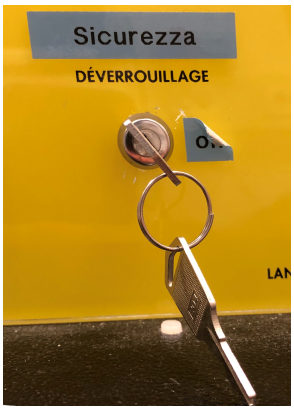


Figure 2.4: Safety key

By rotating the key on the bottom of the didactic elevator, the system introduces the usage of some added safety sensors.
These are mechanical sensors and are placed at each door (one for the closure and one for the opening), above the 3rd floor and below the ground floor.

If a safety sensor is activated, the entire system is switched off. For instance if the door is completely opened/closed, but the system is still trying to open/close it, the current is cut off. The same happens if the cabin is moving down beyond the ground floor or up above the last one.

The aim of this functionality is to avoid any damage to the system when the program is not handling properly the situation. For this reason, this type of sensor are not reachable through the program.

2.2 Programmable Logic Controller module (PLC)

Figure 2.5: ABB PLC PM571

The core of the project was the configuration and implementation of the program on the PLC module. A Programmable Logic Controller is an industrial digital computer which has been adapted for the control of manufacturing processes, such as assembly lines, robotic devices or any activity that requires high reliability control, ease of programming and process fault detection.

PLC programs are typically written in a special application on a personal computer, then downloaded by a direct-connection cable, like in our case, or over a network to the PLC. The program is stored in the PLC either in battery-backed-up RAM or some other non-volatile flash memory.
There are two modules (DC532) directly connected to the PLC, composed by 16 inputs and 16 outpus, useful to manage the I/O signals of the elevator.

Figure 2.6: ABB module DC532

The control system also has a little interface (HMI) that allows the user to act directly on the system after programming it through its specific software (CP400 Soft).

2.3 Programming environment

The PLC was coded through a software provided by ABB: Automation Builder 2.0.
ABB Automation Builder is an integrated software suite for machine builders and system integrator, that wants to automatize the machine and the system in a productive way. It has an intuitive interface and it is mainly used to program, debug and maintain automation projects.

The development of the project was done through the software CoDeSys (version 2.3.9.49).

CoDeSys stands for Controller Development System and it lets the user create an application to be uploaded on the PLC module.
It allows the usage of different languages inside the same program, such as:

- **LD** : Ladder Diagram
- **ST** : Structured Text
- **SFC** : Sequential Function Chart
- **FBD** : Function Block Diagram
- **CFC** : Continuous Function Chart

Structured Text was mainly used in our project. This choice is accounted for the belief to have more control over the development of the program. ST is one of the five languages supported by the IEC 61131-3 standard and it is designed for programmable logic controllers. Moreover, it is a high level language, therefore complex statements and nested instructions are supported.

Before starting to program with the software, it is necessary to set in the software options the exact model of the module. Additionally, the connection port between the PC and the PLC must be checked in order to let the communication work in the correct way.

It is also possible to work without programming directly on the module thanks to the simulation mode, since it is also possible to check the value of each variable inside the program and their behavior over time graphically (as will be discussed in chapter 5).

General overview of CoDeSys

This software has four main sections:

1. **POUs** : Program Organization Unit.
   It is the part of the software where the main program and its functions are localized.

2. **Data types** : In this section it is possible to create new type of variables.
   This is useful to have clearer names of the variables and in our case it was used to define two variables of *struct* type:

   CabinDef : It contains all the I/O of the elevator cabin.

   FloorDef : It contains all the I/O for one single floor. Thus, it was defined an array of 4 elements of this type.

```
TYPE CabinDef :
STRUCT

    Call0: BOOL;
    Call1: BOOL;
    Call2: BOOL;
    Call3: BOOL;
    GoUp: BOOL;
    GoDown: BOOL;
    Light: BOOL;
    Obstacle: BOOL;
    Stop: BOOL;
    ErrorCabin: BOOL;

END_STRUCT
END_TYPE
```

Figure 2.7: CabinDef

```
TYPE FloorDef :
STRUCT

    DoorClose: BOOL;
    DoorOpen: BOOL;
    DoorOpened: BOOL;
    DoorClosed: BOOL;
    CabPresence: BOOL;
    CallUp: BOOL;
    CallDw: BOOL;
    CabLed: BOOL;
    LedUp: BOOL;
    LedDw: BOOL;

END_STRUCT
END_TYPE
```

Figure 2.8: FloorDef

3. **Visualization** : Graphic interface that lets the user simulate the program independently from the hardware.

4. **Resources** : Section in which is possible to configure the PLC parameters of the system, define global variables and add libraries.

Chapter 3

Tasks

3.1 Global view

As previously described, different logic were implemented in order to control the plant according to various management choices.
They can be summarized as:

1. One single call management;

2. Multiple calls management;

3. Optimization of multiple calls management.

Before starting each program, an initialization phase is required in order to define the initial state of the elevator. This phase can be regarded as the final setting that the plant technician imposes to the elevator after operating on it.

The initialization consists in checking that the door of each floor is closed. If this is not verified for all the floors, the corresponding door motor is activated until the door itself results to be closed. Then, the cabin must be at floor 0. If the floor sensor does not detect its presence, the elevator is moved downwards until it reaches the ground floor.
In this section of the program, we also take care of the initialization of different variables that are used inside the code to understand the behavior of the plant and must be reset to avoid wrong decisions.

Here below it is represented a flow chart describing all the steps performed by the algorithm. These are the same for all the three different programs that we have developed, since their main difference is contained in the desired floor definition function.

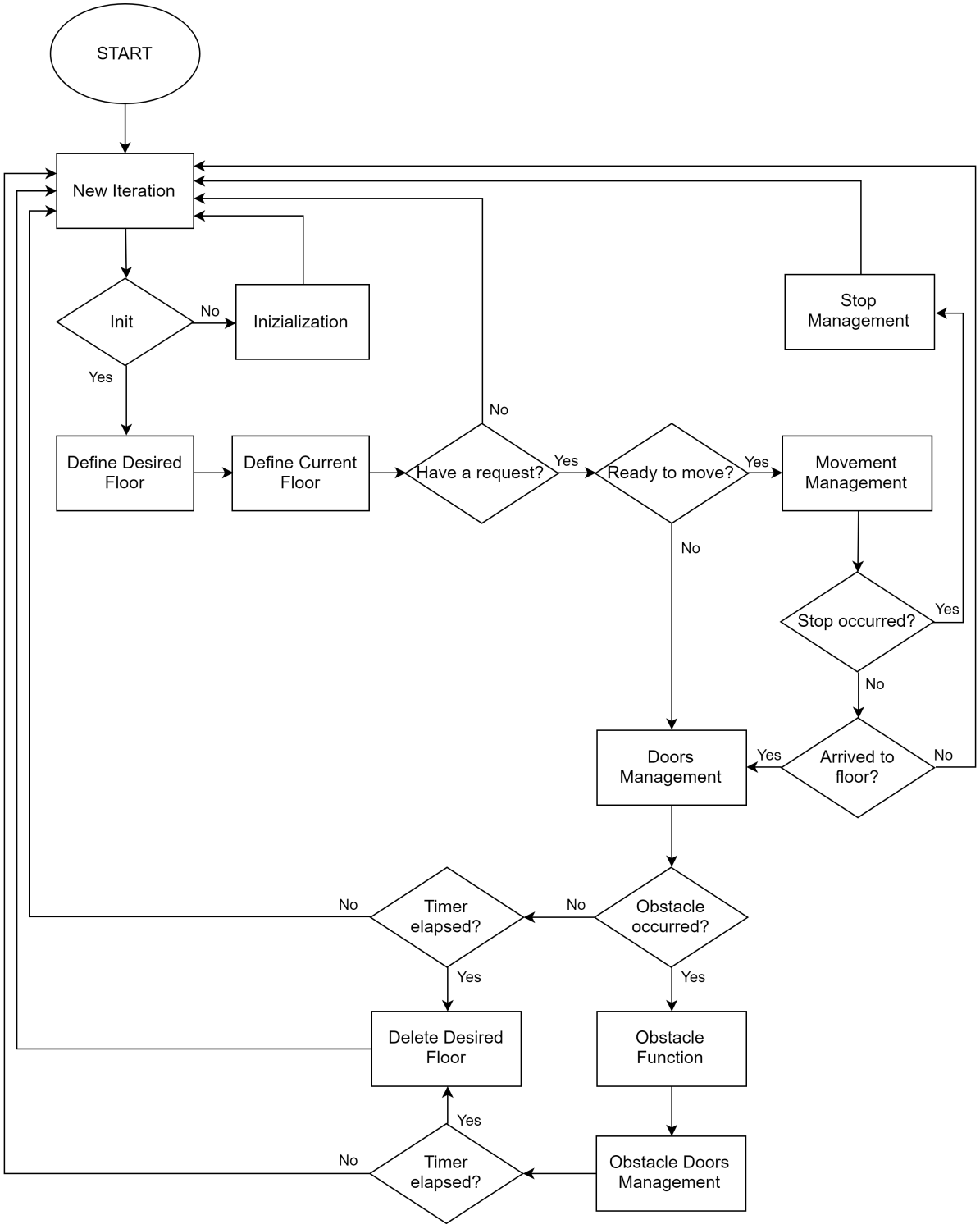


Figure 3.1: Program flow chart

In the next sections we will explain in details the different logic developed, whereas a more precise description of all the implemented functions will be given in Appendix B.

3.2 One single call management

3.2.1 Description

The first program that we have exploited regards the simpler way in which an elevator could work.

In this case the plant does not have a memory, therefore each call is fully completed before having the possibility of taking care of another request.

The user presses a call button, inside or outside the cabin, to define the desired floor. Consequently, the elevator must reach the selected floor, open the door and then close it. At the end of this process, it has to wait for another call.

3.2.2 Requirements

Since in this phase of the project we cannot store multiple calls, we had to ignore simultaneous requests and to discard every call that is selected during the managing of another call, rehabilitating them only when all the movement phases related to the previous destination are concluded.

The call is performed by pressing on one of the buttons linked to the desired floor. In this program we considered all the calls related to the same floor as equal, since we did not have the need to differentiate whether the user wants to go to an upper or lower floor.

Once the call is saved as the desired position to be reached, the elevator must move upwards or downwards, depending on its current position, until the sensor detecting the cabin presence at the desired floor becomes active.

The plant can only move when all the doors are closed, therefore the doors cannot change their state until the elevator stops at their corresponding floor.

When the cabin reaches its destination, there is the management of the doors. They need to be fully opened, stay open for a certain time interval, to allow the entrance or exit of the users, and then be closed again.

During the door management phase, we also need to check for possible obstacles between the doors, that must cause the interruption of the closure of the doors and their reopening.

The presence of an obstacle in a real plant is detected by infrared sensors positioned in correspondence with the doors, whereas in our system, it is defined by the position of the obstacle switch of the didactic elevator. As long as the obstacle is present, the doors must be kept open and they can start the closure phase only after a fixed amount of time from the removal of the obstacle.

After the doors are completely closed and the desired floor has been reset, the system stays still waiting for the next call.

Moreover, during the motion of the cabin, there is also the possibility of the activation of the stop command. This causes the elevator to stand still in the position at which the button was pressed and the lights indicating the cabin presence to blink intermittently, until another floor is selected.

This command is taken into account only when the elevator is not already present at a floor, but it is moving between two of them.
Lastly, there is the lights management that concerns different kind of lamps.
One of them is the light inside the cabin, that has to be turned on for all the time in which the elevator is operating.

Then we have the LEDs representing the presence of the cabin at the corresponding floor, which are used to indicate, both to who is inside and outside the elevator, at which floor the cabin is located.

They are triggered by the presence sensors positioned at the various floors and the light is on only for the time that the related sensor is active.

The last kind of lights is the one related to the requested call. When a call button is pressed, its light is switched on until the corresponding floor is reached, so in this case only one of these LEDs can be lit up at the same time.

3.2.3 Modeling

To build logically and in an understandable way the program, a modeling part is necessary. This part is really important because it allows the programmer to discover bugs in the code logic.

Since the project is dealing with a discrete time system, finite state machines are used to characterize the evolution of the states of the different elements of the plant.

Here will be reported the modeling phase of the single call program, which will be used also for multiple call storage and optimized programs with some differences.

The elevator can be defined by seven states: three defining the presence of the cabin at a precise floor and four indicating that the elevator is between two floors.

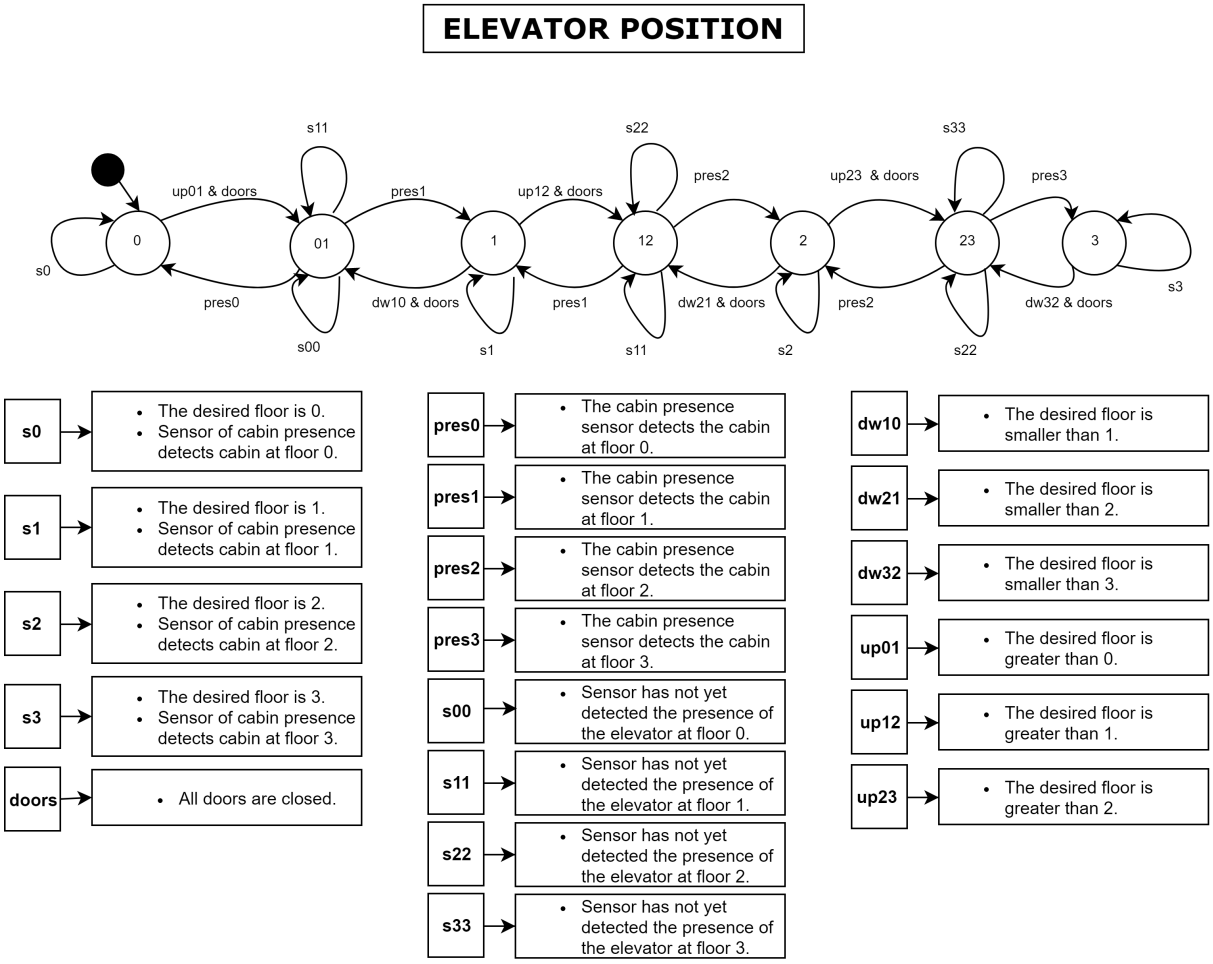


Figure 3.2: Elevator model

Conceptually the model of the door of each floor is the same as the one of the position of the elevator.
It can be modeled by four states: two steady states, when it is completely closed or open, and two dynamic states, when it is closing or opening.

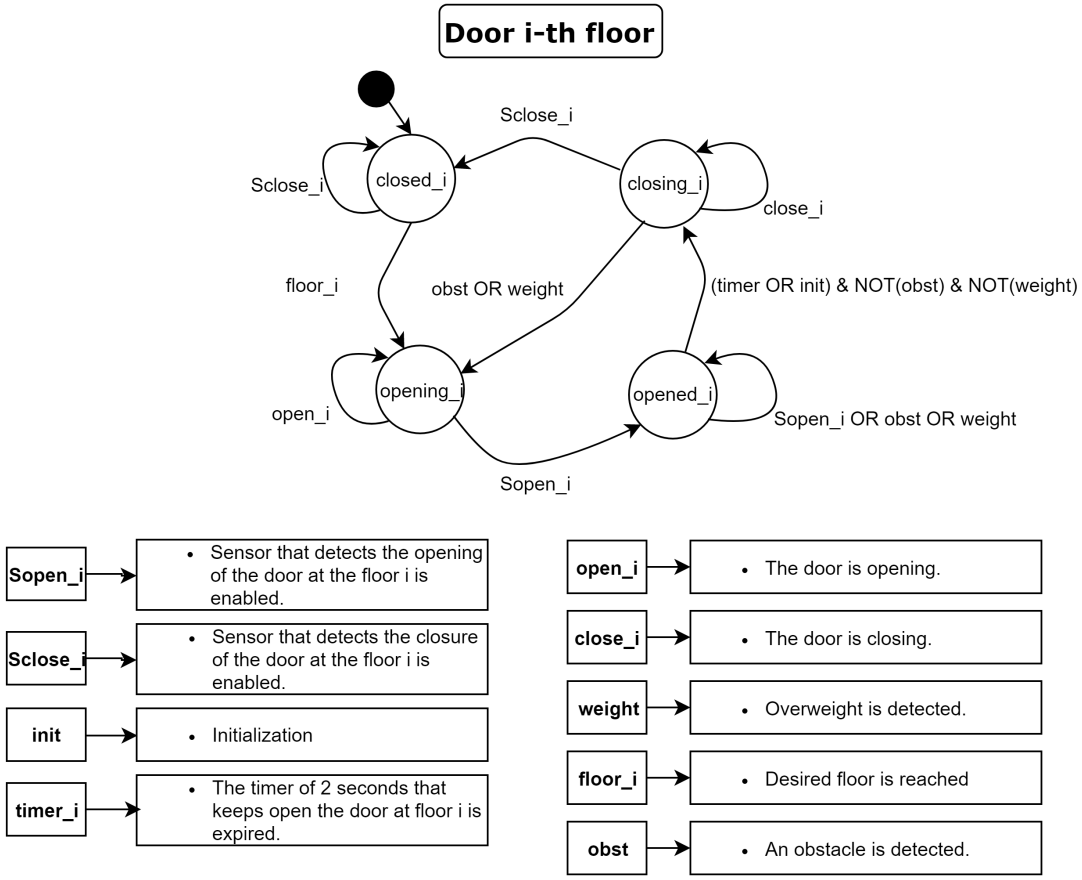


Figure 3.3: Door model

Also the behavior of the LEDs and cabin light have to be modeled. All lights have two states (ON and OFF), the main difference consists on the activation and deactivation conditions.

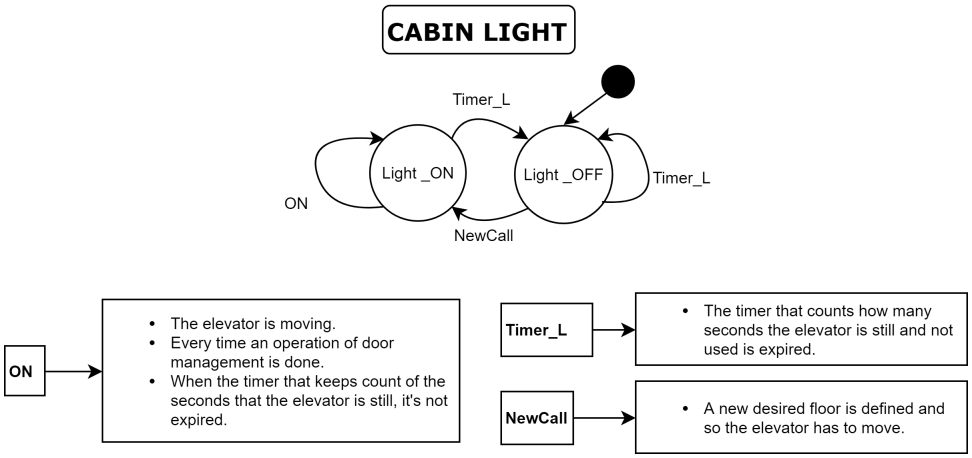


Figure 3.4: Cabin light model

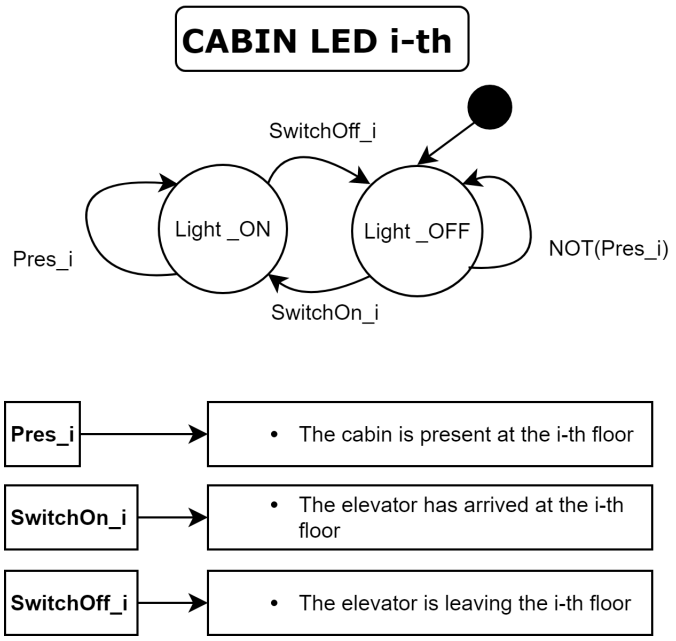


Figure 3.5: Cabin LEDs model

The floor LEDs that indicates the presence of the elevator at that floor behave exactly as the ones inside the cabin.

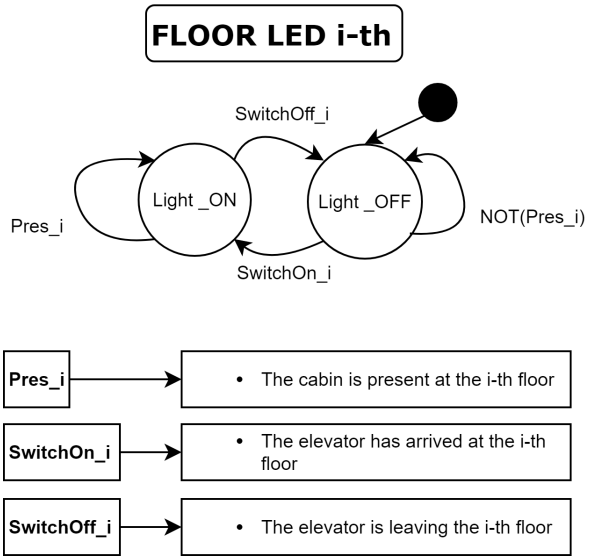


Figure 3.6: Floor LEDs model

The timer that keeps the door open for 2 seconds has its own model. A wrong definition of the timer will lead to an incorrect behavior of the doors.

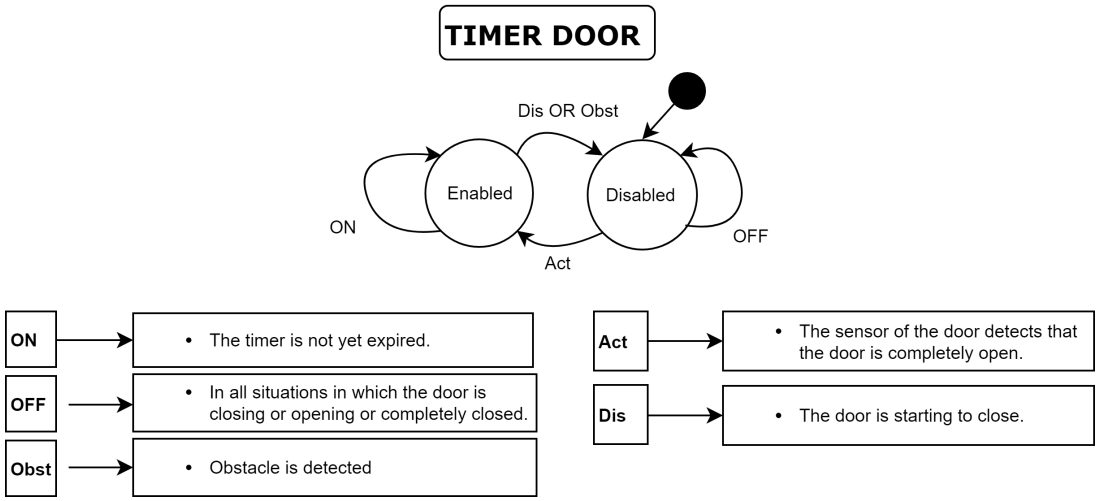


Figure 3.7: Door timer model

3.2.4 Implementation

From the modeling phase we have the basis for the software implementation, which was entirely developed using structured text.

The code is composed of a main program in which initially there is the assignment of the system inputs through a specific program, then there are all the functions defining the actual software and at the end there is another program that allocates all the system outputs. In this way we have the reading phase of the inputs at the beginning of each iteration of the code and the setting of the outputs at the end of the elaboration of all the system variables inside the entire program.

Figure 3.8: Phases of the implementation

In the central part of the main program, the first step focuses on the initialization of the plant. When this phase has been concluded, the software enters the part of the script that actually defines the system behavior during its execution and that is described below.

Initially we have the *Case_Division* function that defines if some of the cases, that cause the necessity of a peculiar management of the elevator, are active or not.
These cases are the presence of an obstacle or the presence of a stop request.
Then, we have the definition of the current floor at which the cabin is located and

the selection of the desired floor based on the call buttons through the function *Define_ Desired_ Floor.*

At this point, if none of the two particular cases are active, the program checks if the system is ready to move and calls the function *Movement_ Management*, that regulates the displacement of the cabin through the *Move_ To* function.
This last function computes the difference of floors between the current position and the desired destination, deciding consequently if the elevator has to move upwards or downwards, until the difference results to be null and the cabin is stopped.

In case there was a stop request during the motion, the *Movement_ Management* function will activate the blinking of the floor presence LEDs and when a new floor is selected it will call the *Stop_ Management* function.
In this part of the program, if the desired floor differs from the current one, the script will request the *Move_ To* function, otherwise it must check if the cabin is exactly at the desired floor or it is between that floor and the next one, moving the elevator accordingly.

At the end of the movement, when the cabin is still at the destination, there is the activation of the *Doors_ Management_ No_ Obstacle* function in which the doors are opened and closed as it will be described in the function description section.

After the first opening of the doors, we can have the enabling of the obstacle case that causes the stoppage of the door and the usage of the function *Doors_ Management_ Obstacle.*
This function behaves as the previous one, with the difference that the possibility of starting the closure of the doors can be triggered only after the removal of the obstacle.

After the door has been closed and a certain amount of time is elapsed, the desired floor is reset to the value that indicates the absence of requests and the plant waits for the next call.

3.3 Multiple calls storage management

3.3.1 Description

The next step made to increase the complexity of the program, regards the introduction of a calls storage memory in order to keep track of the requests done from inside and outside the cabin.

The rule that characterizes the system for what concerns the management of the calls is a FIFO logic: the first call received will be the first request satisfied.
The basic behavior of the system is the same as the one of the previous logic, to which only the calls storage has been added.

3.3.2 Requirements

The introduction of the memory requires that the program must consider every single call requested, even while the elevator is operating.
Thus, the system must be aware of every pressure of the floors buttons either inside or outside the cabin.

The memorization can be done through the implementation of an array in the program that stores the calls in the order in which they were made.
Of course, some precautions must be added: if the same floor is requested two or more times, the request must be added just once in the queue of calls.

Moreover, within two seconds after the closure of the door at a floor, the pressure of one of the buttons of the same floor causes the door be opened again. In this way, we have overcome the absence in our physical system of a button to require the opening of the doors.

3.3.3 Modeling

Only the part related to the call storage of this logic will be explained here, since the other elements have the same functioning as in the previous case.

The model of the queue is characterized by three states: an «idle» state in which the array is not modified, an «add» state in which the request is added to the queue and a «delete» state in which the reached destination is removed from the list.

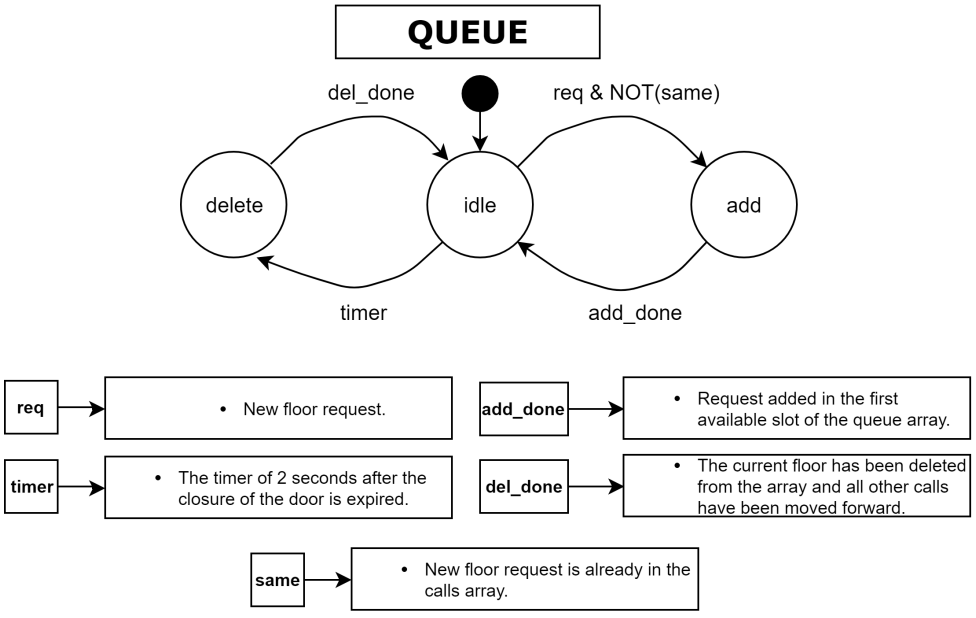


Figure 3.9: Queue management model

3.3.4 Implementation

A new variable, which consists in an array that has a length equal to the number of floors, must be introduced. Its task is to store the incoming requests arriving from any button. After a user presses a call button, it must be checked if the corresponding floor is already present in the array of the calls. If not present, the function *Add_ To_ Queue()* will put the floor number in the first available space of the queue.

When the queue is not empty, the system always defines the floor declared in the first element of the array as the desired one and consequently the elevator heads towards it. Once the movement is completed, similarly to the single call case, the door management logic fulfills its job.

After the door ends its motion phase a timer of two seconds is enabled before the removal of the request from the list and the possibility for the elevator to move. During this time interval, if a request for the current floor is made, the door will be opened again without adding the call in the queue.

3.4 Optimization of multiple calls storage management

3.4.1 Description

The FIFO approach previously described does not embed any kind of "smart logic" but, as the name suggests, simply directs the elevator on the first floor requested until there are no more calls.
This approach is not optimal for many aspects (i.e. power consumption, travel time, user waiting time, user average travel time). Consider, for instance, the overall travel time: the FIFO approach results inefficient if there is a sequence of calls of distant floors.

In order to limit all those problems, we have decided to design an additional logic that, given all the floor requests, is able to decide in which floor go first.

Furthermore, a new feature was added: the elevator is able to manage situations where the weight that the plant is able to bear is exceeded.

3.4.2 Requirements

The optimization of the previous logic is mainly related on the management of the calls got.
In order to minimize the distance traveled by the cabin, an algorithm is implemented to consider the calls received inside and outside the cabin.

Of course, for this purpose, differently from the FIFO case, we have to make a distinction over the various type of calls.

If a person presses the UP button, it is meant that, once the elevator arrives on his floor, the considered user will request a floor that is above the actual one. The same consideration applies for all the DOWN buttons.
Moreover, the program has to define whether the elevator is moving upwards or downwards to be able to avoid the cabin to stop when it is not optimal for the traveled distance.

For what concerns the overcome of the weight bearable by the elevator, the system does not allow the closure of the doors until the weight decreases enough to be under the limitations.

3.4.3 Modeling

The main difference of this logic with the two previous one relies on the management of the three lists of calls from which we have to choose the desired floor.
A model of the rules behind this choice is given below:

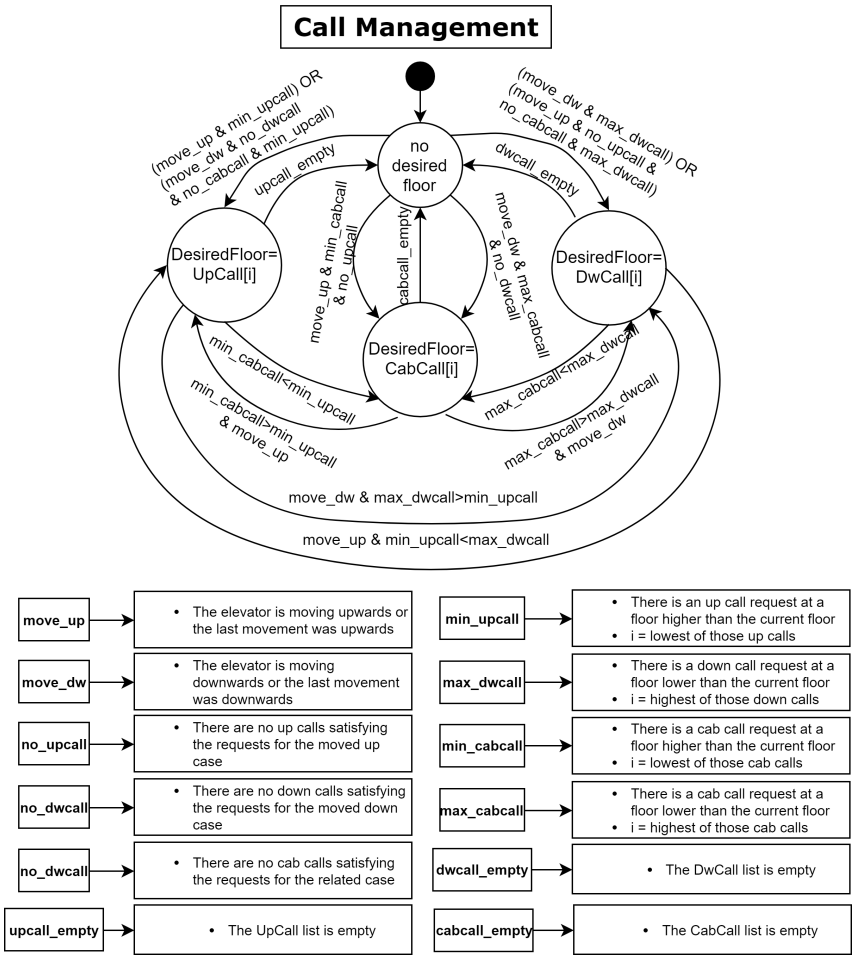


Figure 3.10: DesiredFloor choice model

Due to the difficulty of comprehension of this model, we will later explain the behaviour of the plant regarding the choice of its destination with an example.

3.4.4 Implementation

The management of the requests is done by the introduction of a new function that decides which is the desired floor between all the performed calls.
This means that there is no need to modify the arrays that collect the calls got from the system.

First of all, the direction of the displacement of the cabin must be recognized and the state of the elevator position has to be known. Depending on those two parameters, the algorithm will look for a precise desired floor in the arrays of the collection of the calls.
If we are moving up, the program will look for the requests stored inside the array that contains the calls done inside the cabin and the one that contains the calls to go upwards.
A dual situation happens when the elevator is moving down: instead of the up calls array, it will be used the array that collects the requests to go down.

At each iteration of the program, the desired floor will be updated. Therefore, even if a call is introduced during the reaching of another floor and it is a preferable destination, it will be considered as the new desired floor.

3.4.5 Example

To prove the functioning of the optimized multiple call logic, we provide an example here below.

Consider for instance, the following case:

- A person enters the elevator at the ground floor, and calls the third floor from inside the cabin.
- Another user requests an up call from the first floor, before the elevator overcomes the first floor, to go to the third floor.
- Meanwhile, a person located on the second floor makes a down call request.

The logics stops the elevator on the first floor, then it ignores the call to the second floor and directs to the third floor.
When the first two users exit, the elevator moves to the second floor.

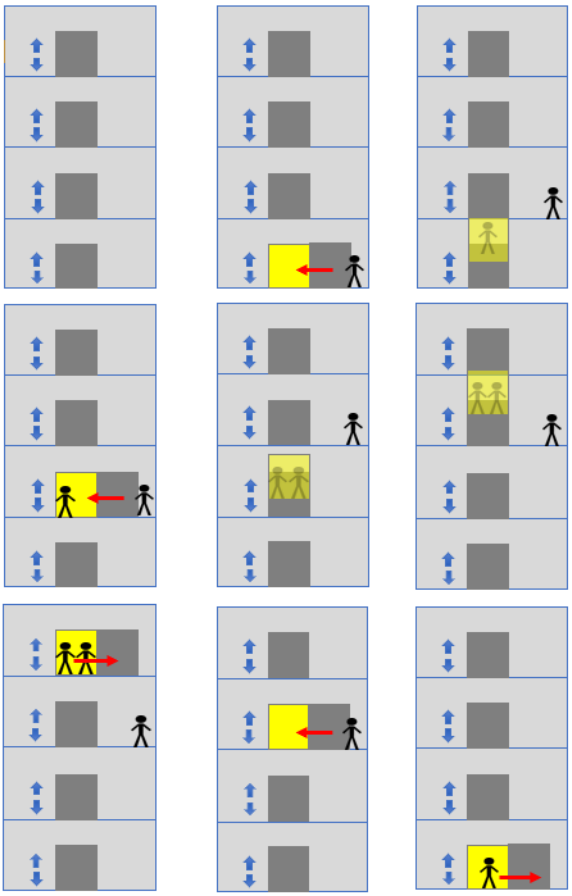


Figure 3.11: Example optimized logic behavior

3.5 Faults detection

3.5.1 Description

The final step in the implementation of the software is related to the fault detection of the system. In our case, the problem is that we do not have any sensor or signal telling us if some elevator component has a malfunction.

If the problem is confined to a led, a periodical maintenance is enough, since this issue does not affect the functionality of the elevator itself. The situation turns over when the piece affected by the malfunction is an actuator or a sensor entrusted to detect the completion of the movement.

In order to avoid those critical issues, the only way is to adopt a time-based fault detector. The general concept beyond is the following: if we know the actuator speed and the time at which is activated, we know exactly how much time it should take to complete a generic task.

Besides the fault detector, it could be interesting the implementation of a logic for a sort of predictive maintenance. Just by taking into account the number of time that the elevator is used, it is possible to decide if a maintenance check up is needed in order to prevent some major problems.

3.5.2 Requirements

The critical issues we have to deal with are essentially two: the movement of the elevator between different floors and the opening/closing action of the door in correspondence of a floor.

We consider the average travel time between two consecutive floors. In order to recognize a fault of the cabin movement we can set a timer that is a Δt longer than the actual time of movement between two floors. Hence, if the timer expires during the movement, the system recognizes that something is not working properly.

Now consider the task of opening and closing a door. In a parallel way to the previous case, we can set a timer that, in case it expires, warns us that something is wrong in the door actuator or sensor. In both cases we are not able to detect if the source of the problem is the actuator, that has stopped working, or the sensor, unable to perceive the cabin/door presence.

After the occurrence of the malfunction, the elevator is no more able to guarantee a safe working condition and for this reason the presence of an expert is needed to restore the normal functionality. In our case, we simulated the occurrence of a problem by physically disconnecting the target signal pin on the board.

3.5.3 Modeling

The introduction of the timer for each door and for the movement of the cabin has to be modeled accordingly to the requirements.
Finite state machines is still the solution that was used to model them.

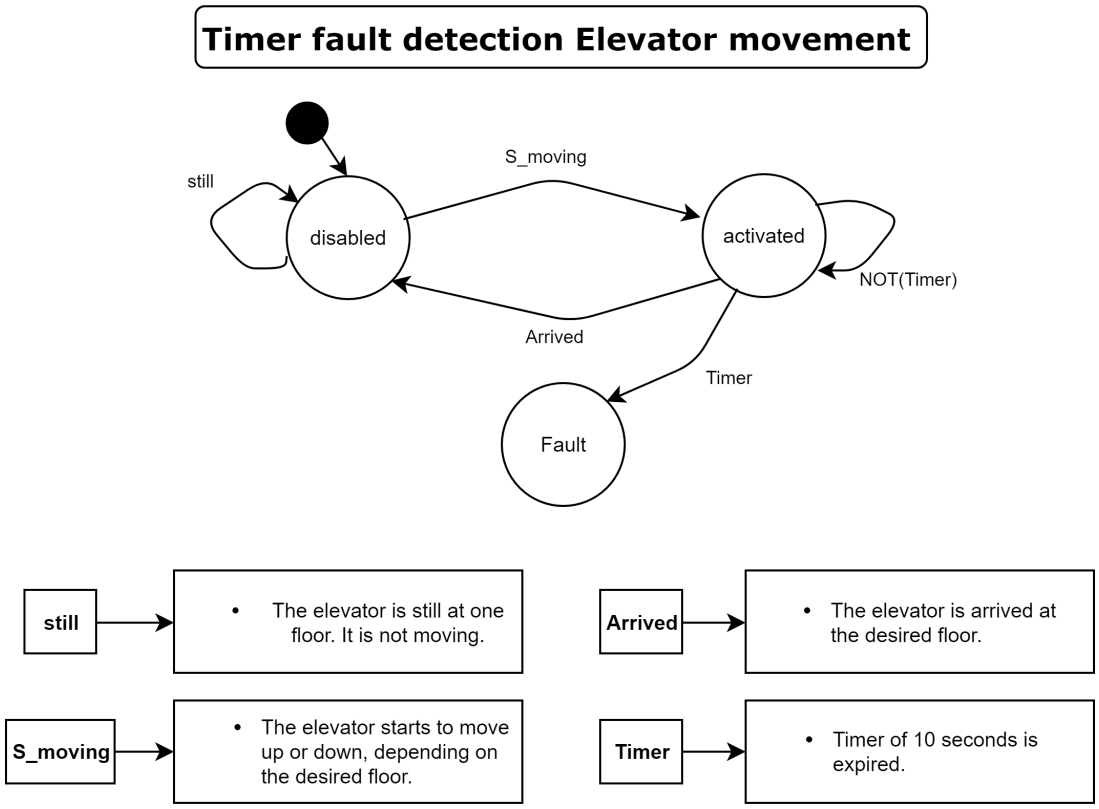


Figure 3.12: Fault timer elevator

For the doors, it has been defined a timer both for the opening and for the closing phases.

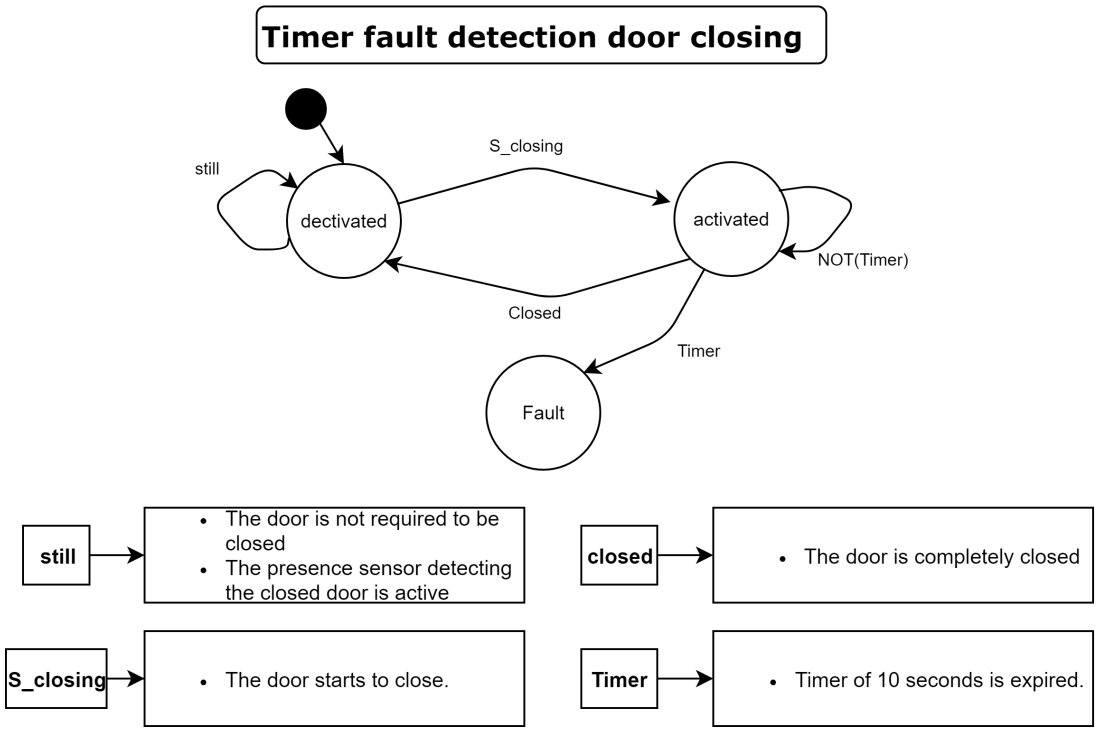


Figure 3.13: Fault timer closing door

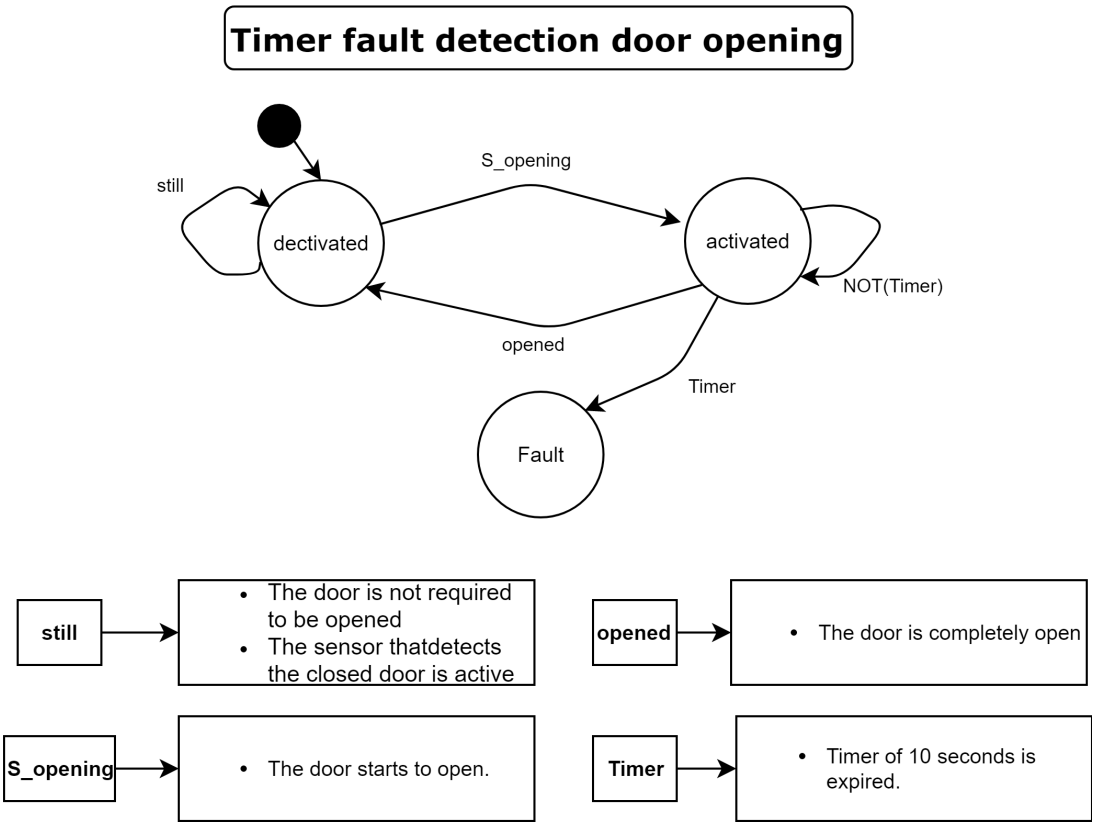


Figure 3.14: Fault timer opening door

3.5.4 Implementation

The cabin timer starts every time the elevator is moving between two floors and it is set to zero when a cabin presence sensor of one of the floors is activated. When the fault is detected, the motor that moves up and down the cabin is turned off.

In a similar way, the opening/closing door timer starts when the elevator begins to open/close the door and it is reset in the exact moment in which the system senses that the door is fully opened/closed. When the fault is detected, the door motor is immediately switched off.

In case one of the two malfunctions occurs, the elevator is no more active and only the action of a professional technician can restore the behavior of the system.

Chapter 4

N-floor implementation

A fundamental specification to be satisfied by the program consists in its scalability.
The advantage of the implemented software is its flexibility to different system configurations.

On this purpose all the variables, that appears to be repeated in each different floor, are merged into arrays of dimension equal to the number of floors. Moreover, all the functions are meant to work with an arbitrary number of floors.

In fact, the idea behind the scalability relies on the fact that all the elevators have one bottom floor, one top floor and a variable number of middle floors which can be treated in the same way.

Hence, as long as the physical inputs and outputs are manually assigned to the corresponding variables, the program can be used to build up an elevator of four floor, ten floors or fifty floors without any difference.

Unfortunately, we were not able to physically test the software on a real elevator due to the limits of the available material, but we were able to create a simulation example of a six floor elevator that worked as expected.

Chapter 5

Simulation and independence from the hardware

In order to verify and validate our programs we initially studied the behavior of the system on virtual simulators so that we did not risk damaging the physical structure at our disposal.
Then, when the programs resulted to be quite reliable, we started the actual testing on the didactic elevator to identify further bugs in the programs.

5.1 Static simulation

For what concerns the simulation of the system behavior, we initially used a static visualization implemented on CoDeSys.

This simulator was thought just to perform the identification of the very basics movements and conditions for the functioning of an elevator system. For this reason, it was kept as simple as possible and we did not focus on the graphical aspect.
In this simulator we have the buttons representing the main input and output variables, like the calls, the activation of the motors and the sensors of the floors and doors.

The main problem with this visualizer was that we had to impose the value of the input sensors through predefined buttons, therefore the behavior of the systems depended entirely on the inputs inserted by the user, instead of depending on the inputs triggered by physical events.

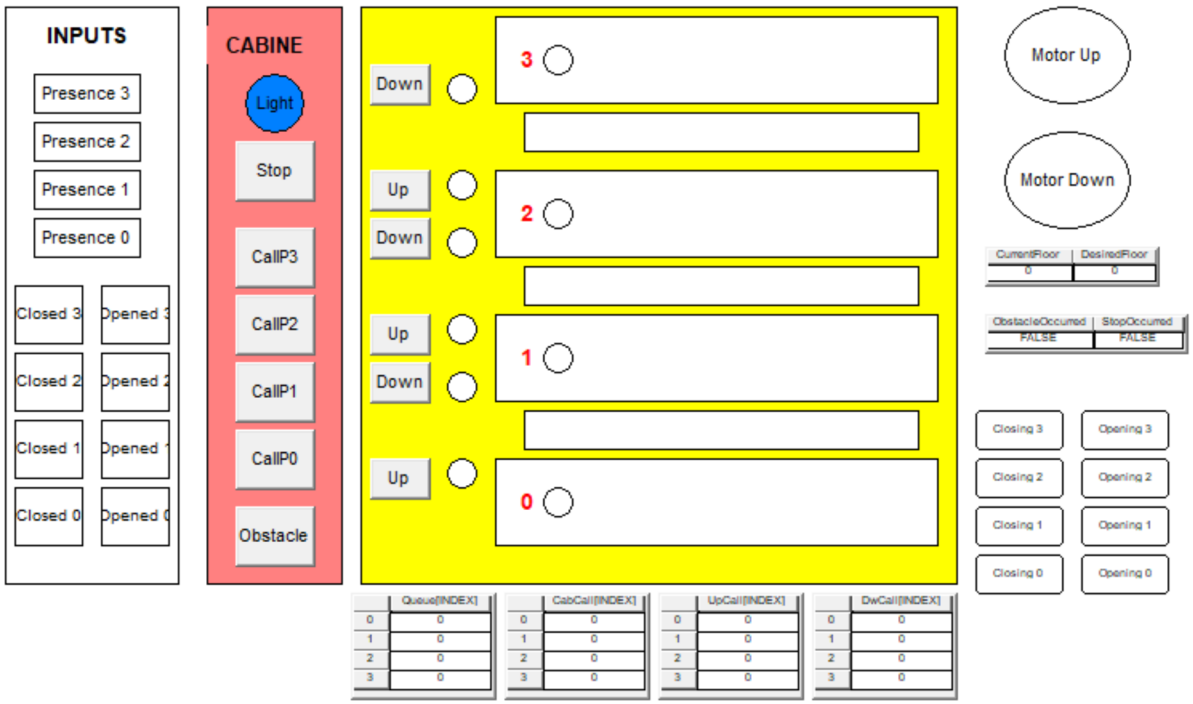


Figure 5.1: Static visualization

On the bottom there is the visualization of the calls stored into the arrays of the program. The Queue array is used in the multiple call logic, whilst the other three lists are used in the optimized multiple call logic.

	Queue[INDEX]
0	0
1	0
2	0
3	0

	CabCall[INDEX]
0	0
1	0
2	0
3	0

	UpCall[INDEX]
0	0
1	0
2	0
3	0

	DwCall[INDEX]
0	0
1	0
2	0
3	0

Figure 5.2: Static visualization: Queue

5.2 Dynamic simulation

To be able to work on this project also in remote relying on the simulator results, we decided to implement a dynamic visualizer in which we could see the actual behavior of the elevator subject to our control programs.

For the graphical simulation, we created a static background symbolizing the building, in which there are five moving blocks, four of which represent the sliding door at each floor and the latter acting as the elevator itself.
For the displacement of these elements we introduced some variables that increment or

decrement their values depending on the commands received from the used program.

Then we added some input buttons related to the tap variables associated to the different floor calls. For each floor we have a call button inside the elevator cabin and two call buttons on the corresponding floor, one to go upwards and one to go downwards (except for the first and last floor that only have the up-call button and down-call button respectively).

On the panel inside the cabin we also have a stop button that causes immediately the stoppage of the system, until a new floor is called, and that triggers the utilization of a specific part of our programs.
Moreover, we added a toggle button to simulate the presence of an obstacle between the doors, that forbids the closure of the doors until it is removed by pressing it again.

At the end, we considered the different lights of the system. The light inside the cabin is represented by a change of color of the rectangle representing the elevator, triggered by the variable linked to the cabin light.
The lights denoting the elevator presence at a precise floor are displayed as the change of color of the buttons inside the cabin and of the presence indicators at the floors, which are activated by the detection of the elevator from the corresponding presence sensor.
For what concerns the lights indicating the calls that have been carried out, they are depicted as the change of colors of the call buttons themselves.

Using this simulator with all of our programs we were able to check that all the codes satisfied the specific requirements and behaved as expected. After this preliminary analysis, we uploaded the programs on the PLC module connected to the real system and we performed further tests on the physical system to be certain that all codes work properly.

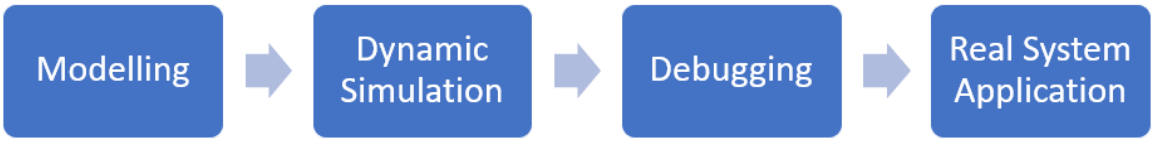


Figure 5.3: Schematic approach

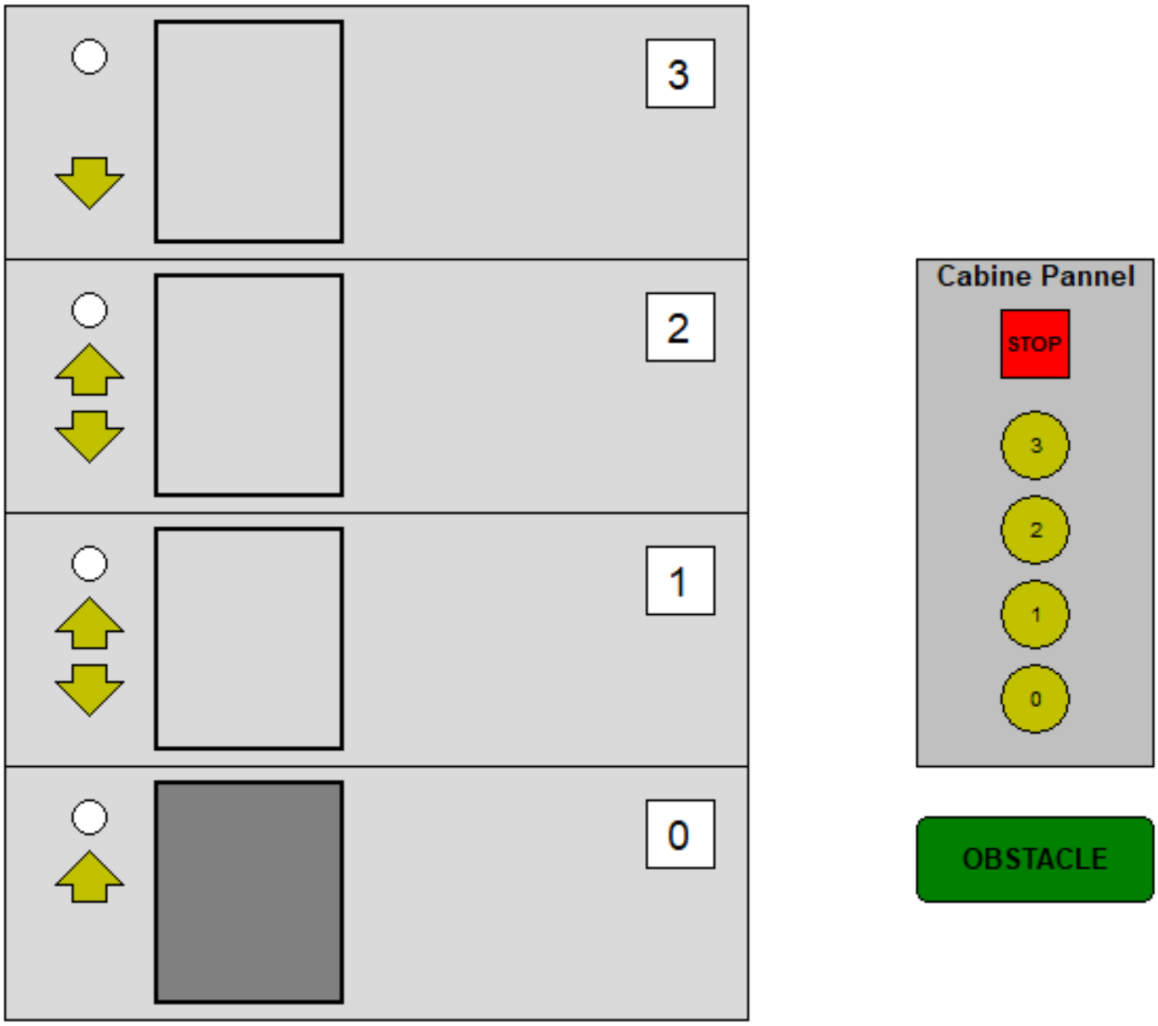


Figure 5.4: Dynamic simulation on CoDeSys

Chapter 6

HMI

6.1 Introduction

The HMI (Human-Machine Interface) consists of hardware and software that allow user inputs to be translated as signals for machine that, in turn, provides the required result to the user. The HMI grants pre-determined modifying functions on the system so its aim is not to program the controller, but lets the operator monitor the status of the system and trim some parameters.

Typically, information is displayed in a graphic format (Graphical User Interface or GUI) and, in our case, this can be managed directly from the touch screen of the LCD.
A CP400 series panel from ABB was used, precisely a CP415 HMI module. It is formed by an active monochromatic control panel of 4 inches that it is programmed through a single RS-232 serial interface.

Figure 6.1: HMI CP415

The environment used to develop the interface program implemented in the HMI is different from the one used to code the PLC module.
The new design suite is still provided from ABB and it is called CP400 Soft. It provides

a really intuitive drag-and-drop style interface, that allows to create your own interface easily.

To carry out this task, firstly, the assignment of the variables must be done and then the layout interface can be created.

6.2 Assignment of the variables

Every variable in the program must be allocated in a well-defined space of the RAM memory of the PLC module and this is done in the variable declaration of the program in CoDeSys.
The assignment of the address to a variable is possible through the use of special character sequences. These sequences are a concatenation of the percent sign %, a range prefix, a prefix for the size and one or more natural numbers separated by blank spaces.

The following range prefixes are supported:

- **I** : Input
- **Q** : Output
- **M** : Memory location

The following size prefixes are supported:

- **X** : Single bit
- **None** : Single bit
- **B** : Byte (8bits)
- **W** : Word (16 bits)
- **D** : Double Word (32 bits)

Several type of variables exist and what differ from each other are the size to be allocated and the way to be accessed.
The main data types used for our purposes are:

- %MX0.xx.y for boolean values;
- %MB0.xx for strings;
- %MW0.xx for int or uint values;
- %MD0.xx for real values.

The prefix 'xx' stands for the byte that is required to be allocated, while 'y' stands for the bit.

The assignment of the variables needs to be done carefully. CoDeSys does not notify if a variable has the same address of another one and this could lead to a completely wrong operation of the program.
The variables that were not statically assigned are automatically placed in free memory areas.

Every time an address of a variable changes or the HMI application is modified, the panel must be disconnected from the controller and restart, in order to upload the new program version. There is no possibility to make online changes.

6.3 HMI program

The HMI program is developed in order to be used by a technician for a deeper control of the plant but even from a user to check some basic information about the elevator.

Through the main interface it is possible to choose four different types of layouts. This means that the first layout is composed by four buttons and each one leads to a different environment.
These environments are:

- Technical
- Logic
- User interface
- Faults detector

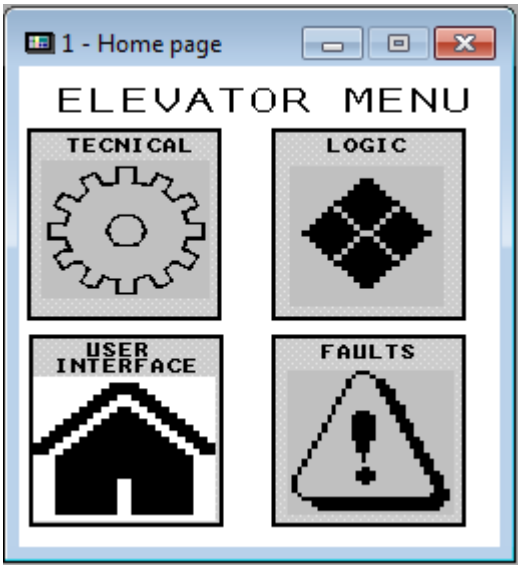


Figure 6.2: Home page

Technical

Entering this layout, the program interface will be brought to the manual functioning. In this section is possible to directly control the output of the plant.

First of all, the manual mode must be activated. After that the system will be independent from any logic, therefore the technician can act freely on the plant.
From the graphic layout we can move up and down the elevator or, pressing the «door» button, it is possible to switch in the relative screen in order to open and close any door of each floor.
This is useful to check that every actuator is working properly.

Once the manual usage of the plant is concluded, the «*INIT*» button must be pressed to let the elevator behave accordingly to the chosen logic.

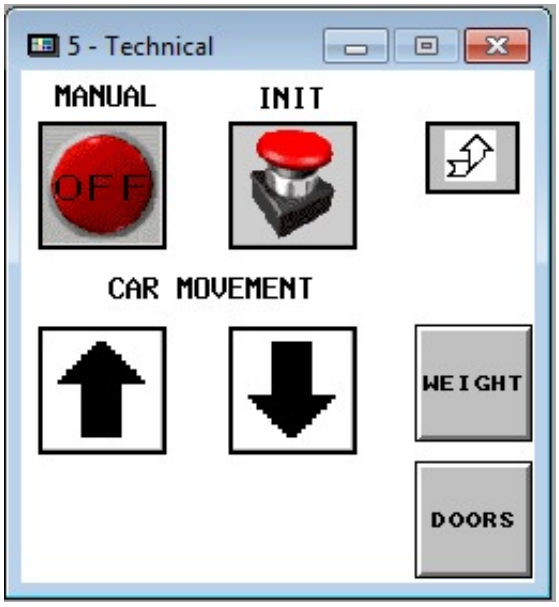


Figure 6.3: Manual interface

The doors interface is defined here below:

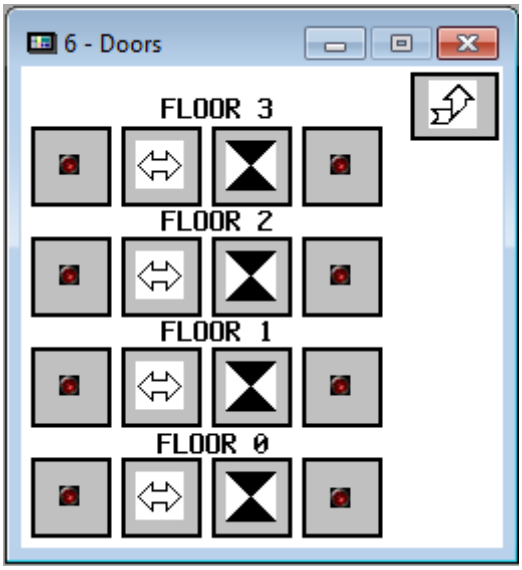


Figure 6.4: Doors interface

The LEDs at the side of the opening or closing button define respectively that the door is completely open or closed.
It is shown also a button on the right top of the interface. This is used to go back to the previous interface.
Coming back to the Technical layout, the current value of the weight carried by the elevator can be checked.

Logic

In this section is possible to choose the logic we want to control the whole plant. Before starting the program a logic must be chosen.

The introduction of a number between 1 and 3 will define the selected logic.

- 1 : Single call logic
- 2 : Multiple call logic
- 3 : Optimized logic

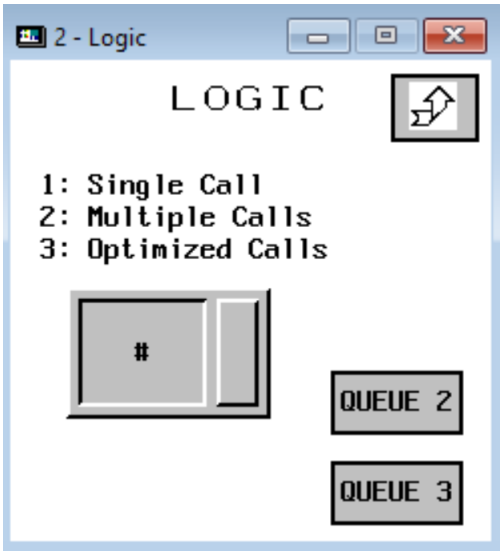


Figure 6.5: Logic selection interface

After the logic is defined, it is possible to check the queue of the calls by clicking the related button in the same interface.
In order to visualize the right queue, a constraint is imposed:
if we choose the logic 2, nothing happens when the «QUEUE 3» button is pressed.
The same happens if we choose the logic 3 and someone presses the «QUEUE 2» button.
Obviously, the one single call logic does not have any queue to check.

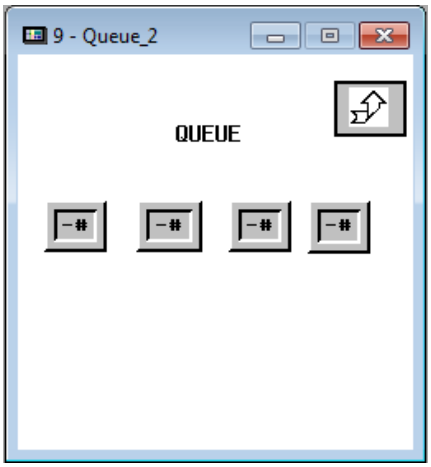


Figure 6.6: Queue interface logic 2

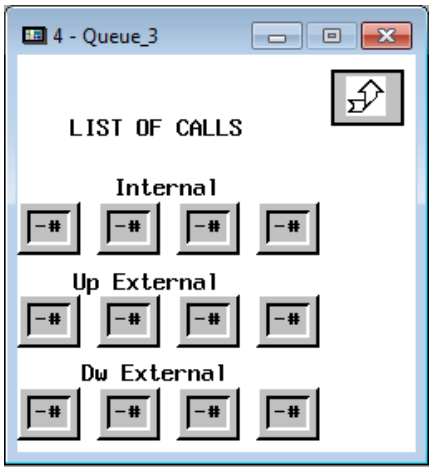


Figure 6.7: Queue interface logic 3

User interface

This screen is supposed to be place inside the cabin and been the only one available from the user. Through it, is possible to check different status information:

- Current position of the car
- Floor that is going to be reached
- Maximum weight allowed

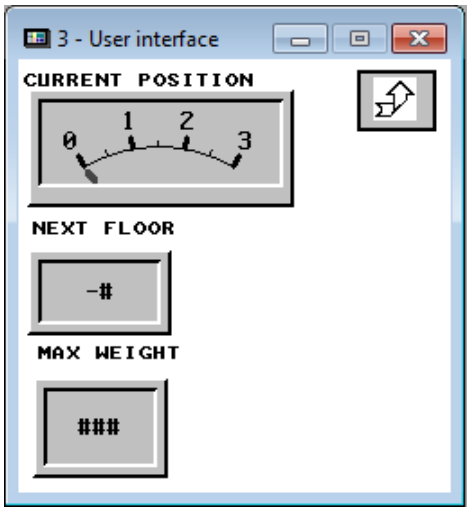


Figure 6.8: Cabin status interface

Fault detection

This layout is an important tool to check if the system is behaving correctly. In this interface it is possible to check if any fault of the system has been detected.

Accordingly to the main program, a fault is a particular situation in which a possible malfunction has affected the system.
In addition to stopping the functioning of the program, the algorithm tags the detected malfunctions with a corresponding label. Based on the tag, the displayed message describes the error.

Having a direct feedback from the system, the technician is able to save time and solve the problem as soon as possible.

Figure 6.9: Fault detection interface

Chapter 7

Conclusions

This experience introduced the team to the world of the PLC.
The main challenge we faced was the fulfillment of the requirements of every logic and the search for the best solution we could find.

7.1 Problems

The main problems that we dealt with are related to:

- **software** : we were not used to this logic and software development environment, so we had a slow beginning learning curve.
  After the development of the easiest task, we had steep progresses in the development of the program.

- **hardware** : some of the buttons of the didactic elevator do not work properly.
  For example, sometimes it happens that one of the button of the 1^{st} floor get stuck and remains set to *TRUE*.
  This problem causes unexpected behaviours that can be solved only if we realize that the button is blocked.

7.2 Results

All the requests are accomplished and our final project is characterized by the following features:

- **Modularity** : the program is build with modules.
  Hence, every change or improvements can be accomplish easily by modifying just the related part of the code.
  Moreover, the division on modules of the program leads to a better comprehension of the program itself.

- **Fault detector** : the introduction of the fault analysis of the system leads to an increasing of safety of the physical environment, allowing us to block the system in case something is not working properly.

- **Scalability** : the logic developed can be used for a higher (or lower) number of floors of a building.

- **Simulation** : the dynamic simulation of the program through the visualization environment of CoDeSys let us be independent from the hardware.
  This option is really useful when we want to try new logic or we want to change something in the code without having the physical system available.

- **Optimization** : the optimization is based on the minimization of the path that the elevator has to cover in order to satisfy all the calls.
  The call optimization logic has proven to bring better results than the FIFO logic.

7.3 Further implementations and goals

The most critical problems related to the development of a software that involves humans are the ones referring to the safety conditions.

The behavior of the plant also depends on the reliability of the sensors used. This means that a failure of a sensor can lead to a really dangerous situation. For example, if the cabin presence sensor of a floor is not working, the motor of the elevator will continue to let the cabin go up.

A possible solution that was exploited is based on timers; we are conscious that sometimes this is not enough to avoid anomalies.
A better solution can be the use of multiple sensor to measure the same input. The comparison of two measures of the same quantity tells us if one of the two sensor is not working properly.

Possible future goals that can be implemented in the program are:

- **Multi-elevators systems**: if a building has more than one elevator, it could be interesting to develop a logic able to optimize the calls.

- **Fault detection based on Sequential Probability Ratio Test (SPRT)** : According to a research from the Department of Electrical and Information Engineering, Jinan University, Zhuhai, China, through data pre-processing, representative value extraction and sequential probability test, the model of the plant can obtain the fault diagnosis results for a typical elevator.

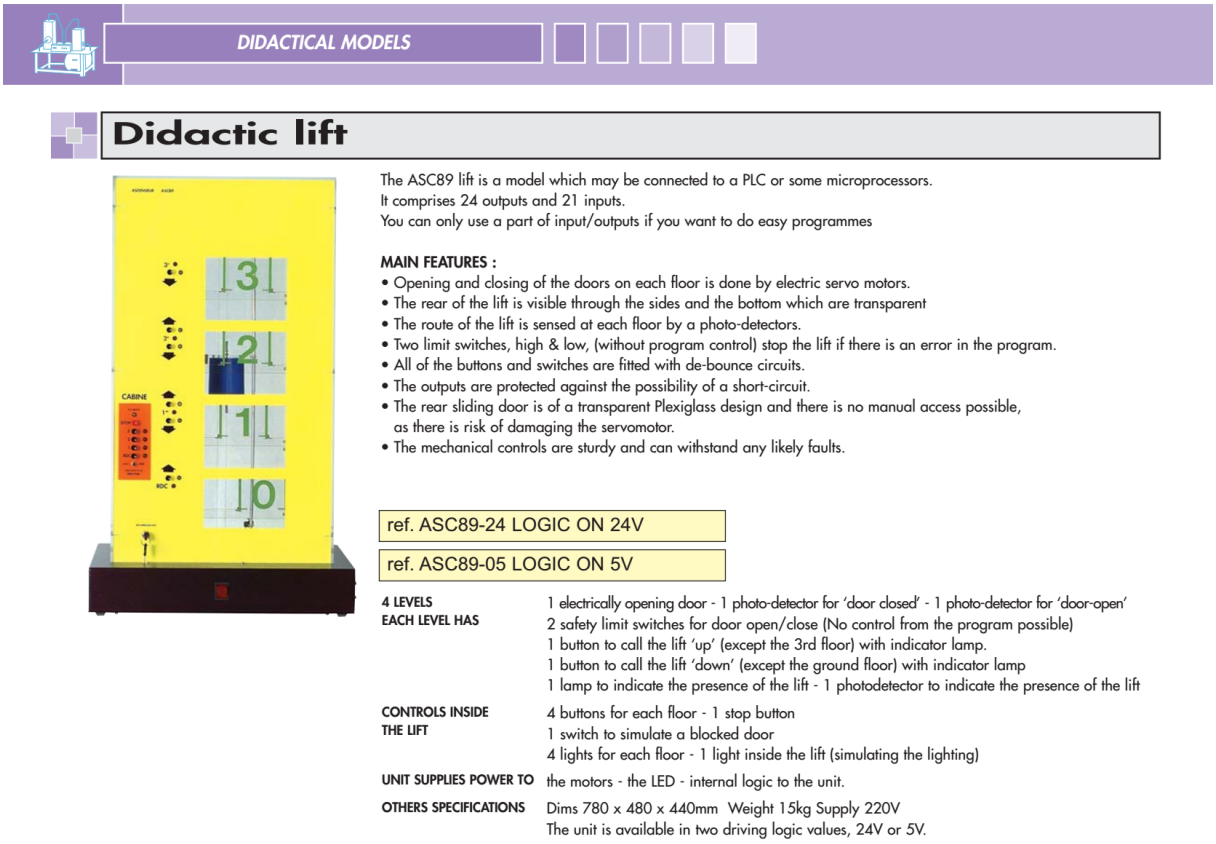
To use this method, accelerometers need to be included in the system where we want to detect the possible faults. For example, if the fault is related to the motor of the door, the accelerometers must be applied on the motor stator and rotor.

- **Probabilistic analysis**: Build a probabilistic model that takes into account the most common requests during different moments of the day.
  This can be used to move the elevator at the most frequented floors when it's not busy in order to reduce the average waiting time for users.

Appendices

Appendix A

Didactic Elevator data-sheet



DIDACTICAL MODELS

Didactic lift

The ASC89 lift is a model which may be connected to a PLC or some microprocessors.
It comprises 24 outputs and 21 inputs.
You can only use a part of input/outputs if you want to do easy programmes

MAIN FEATURES :

- Opening and closing of the doors on each floor is done by electric servo motors.
- The rear of the lift is visible through the sides and the bottom which are transparent
- The route of the lift is sensed at each floor by a photo-detectors.
- Two limit switches, high & low, (without program control) stop the lift if there is an error in the program.
- All of the buttons and switches are fitted with de-bounce circuits.
- The outputs are protected against the possibility of a short-circuit.
- The rear sliding door is of a transparent Plexiglass design and there is no manual access possible, as there is risk of damaging the servomotor.
- The mechanical controls are sturdy and can withstand any likely faults.

ref. ASC89-24 LOGIC ON 24V

ref. ASC89-05 LOGIC ON 5V

4 LEVELS EACH LEVEL HAS	1 electrically opening door - 1 photo-detector for 'door closed' - 1 photo-detector for 'door-open' 2 safety limit switches for door open/close (No control from the program possible) 1 button to call the lift 'up' (except the 3rd floor) with indicator lamp. 1 button to call the lift 'down' (except the ground floor) with indicator lamp 1 lamp to indicate the presence of the lift - 1 photodetector to indicate the presence of the lift
CONTROLS INSIDE THE LIFT	4 buttons for each floor - 1 stop button 1 switch to simulate a blocked door 4 lights for each floor - 1 light inside the lift (simulating the lighting)
UNIT SUPPLIES POWER TO	the motors - the LED - internal logic to the unit.
OTHERS SPECIFICATIONS	Dims 780 x 480 x 440mm Weight 15kg Supply 220V The unit is available in two driving logic values, 24V or 5V.

Figure A.1: Didactic elevator data-sheet

Appendix B

Function definition

In order to keep the program intuitively, understandable and easily readable, several functions were defined. Using various functions, we were able to obtain a modular program that can be adjusted to different scenarios, reusing some of our functions and implementing new ones to satisfy new specifications.
A description of all the functions that were implemented in our programs is given below.

Initialize() : it resets all the lights and queues saved during the previous iterations of the program. Then, it closes the door at each floor until the corresponding sensor is active. At this point, the cabin is moved downwards until it reaches the ground floor. At the end there is the resetting of all the timers used for the errors detection and of all the other variables used inside the main program to make decisions on the behavior of the system.

Logic_Choice(Logic) : it chooses which kind of logic will be used inside our program, based on the value of the input parameter *Logic*.

- *Logic* = 1 is used to select the single call logic;
- *Logic* = 2 is used to select the multiple calls logic;
- *Logic* = 3 is used to select the optimized multiple calls logic.

When the logic changes from the previous iteration, this function sets to false the variable related to the initialization, causing the reactivation of the Initialize function.

Case_Division() : it defines if the stop button has been pressed or an obstacle was detected by a sensor during the closure of a door. Its outputs (*StopOccurred* and *ObstacleCase*) are used to select different functions inside the main program.

Define_Current_Floor() : it defines where the cabin is located, depending on the values of the input sensors related to the detected presence at the floor, and it turns on the presence LEDs accordingly.
It specifies also if the cabin is between two floors, instead of being in a precise correspondence with a floor sensor.

Define_Desired_Floor() : it checks which button was pressed (inside or outside the cabin) and defines the selected floor as desired destination. In case of an external call, the function has the task to activate the Led corresponding to the used call button. This function is used only in the single call management program.

Add_To_Queue() : it checks if the called floor is already present in the queue and, if it is not, adds the requested floor at the end of the queue with the *Add_Last_To_Queue* function. In case the called floor is the same floor at which the cabin is and the time before being able to change floor is not elapsed yet, the request is added at the beginning of the queue with the *Add_First_To_Queue* function.
This function takes care also of turning on the LEDs corresponding to the carried out calls and it is used only with the multiple call management logic. In this case the *DesiredFloor* is set as the first element of the queue.

Add_To_Calls() : it manages three different lists: one referring to the calls performed inside the cabin and two referring to the external calls, one to go upwards and the other to go downwards. Even in this case, it is checked that the selected call is not already present in the related list and, if this is satisfied, the floor is added at the end of the list with the *Add_Last_To_Calls* function.
The *Add_To_Calls* function is used only with the optimized multiple calls logic and the desired floor is selected through the *Manage_Calls* function.

Manage_Calls() : it decides which is the destination of the cabin in the optimized multiple call logic. The choice is made depending on the current floor at which the elevator is and on the direction of is motion.
If the cabin is moving upwards, it checks on the cabin calls list and on the up calls list which is the lowest floor required and selects it as the desired floor.
If there are no requests made from inside the cabin or to go upwards, the function selects the higher floor with a request to go downwards.
If even in this case we do not have a destination, it ultimately looks for a request to go upwards from a floor lower than the actual one.
Vice versa if the cabin is moving downwards.

Movement_Management() : it controls the entire motion of the cabin in normal operating conditions using the function *Move_To*. If the stop button is pressed, the *Stop_Management* function is launched, the lights indicating the cabin presence at the floors starts to blink intermittently and all the previous requests are eliminated through the *Delete_All_Queue* function.

Move_To(CurrentFloor, DesiredFloor) : it computes the difference between the desired floor and the current floor.Therefore, it defines how many floors the elevator has to move from its position to reach the desired floor.

- If the difference is positive, the elevator has to move upwards.
- If the difference is negative, the elevator has to move downwards.
- If the difference is zero, the desired floor is already the current floor and the elevator has to stand still.

Stop_Management() : it defines how the elevator has to behave after a stop request is performed. If a new floor is selected, it causes the stoppage of the blinking LEDs and checks if the desired floor differs from the floor at which the cabin is. If this condition is verified, it calls the *Move_To* function. If the selected floor is the same as the current floor, but the presence sensor is not active, the cabin is moved upwards or downwards, depending on the variables *IsBetween*, until it activates the desired sensor.

Doors_Management_No_Obstacle() : it manages the opening and closure of the door when the obstacle has not been detected. It starts with the opening of the doors through the *Open_Door* function. Once this operation is completed, it waits for 5 seconds before enabling the call of the *Close_Door* function. After the complete closure of the doors, there is another timer of 3 seconds before having the possibility of considering the movement fully terminated.
Both during the closure of the doors and before the expiration of the timer for the conclusion of the movement, by making a request on the floor at which the cabin is present, the door will reopen. This was implemented in order to overcome the absence of a button to request the reopening of the doors when the elevator is steady at one floor.
At the end of the motion of the doors and after the timer is elapsed, there is a call to a delete function that is used to delete the floor request. In case we are using the multiple call logic, the function is *Delete_To_Queue*, whereas if we are using the optimized logic, the function is *Delete_To_Calls*.

Obstacle_Function() : it manages the movement of the doors when an obstacle is detected.
The variable that defines that an obstacle was detected is kept to *TRUE* until the obstacle is removed.
At the expiration of the timer at the end of the motion of the doors, with the doors closed, the obstacle case is deleted and the elevator goes back to its normal operating conditions.

Doors_Management_Obstacle() : it manages the opening and closure of the door when the obstacle is detected.
The door is opened completely if an obstacle is detected. At this point, a timer is switched on and after 2 seconds the door starts its closure phase.
Every time the obstacle reappear between the doors, they are reopened and the timer is reset.
Even in this case, if during the closure of the door or before the expiration of the timer for the end of the entire motion a call for the actual floor is made, the door will be reopened.
At the end of all the operations related to this function, the request for the current floor is deleted from the lists with the same two functions as in the case without obstacles.

Open_Door() : it opens the door until the corresponding sensor detecting the complete opening of the door is activated.
Once the door is completely open, the timer of 2 seconds is activated and the variable that defines the end of the opening phase is set to *TRUE*.

Close_Door() : it closes the door until the corresponding sensor detecting the complete closure of the door is activated.
When the door is completely closed, the timer of 2 seconds is deactivated and the variable that defines the end of the closure phase is set to *TRUE*.

Delete_All_Queue() : it sets to -1 all the cells of all the storage lists of our program, deleting all the previously carried out requests.

Delete_To_Queue() : it turns off all the LEDs associated to a call made for the current floor. Then it proceeds with the removal of the first element of the queue and with the advancement of position of all the other calls.

Delete_To_Calls() : it turns off all the LEDs associated to a call made for the current floor. It then checks if the current floor is present in the three lists (cabin calls, up calls and down calls).
To remove the floor from the lists, its value is substituted with one used to identify the cell that has to be occupied by the subsequent call. At this point it is performed a swap, that eliminates all the meaningless cells.

ASS_INPUT : to have a simpler notation, we created two different data types: *FloorDef* and *CabinDef*. The first includes all the variable related to one single floor and the latter contains all the variables linked to the cabin. The task of this function is the assignment of the input variables of the plant to the variables contained inside the two struct variables.

ASS_OUTPUT : as explained in *ASS_INPUT*, the assignment must be done also between the output variables defined by the data types and the ones related to the physical plant.

SIMULATION_INPUT : it is a program that is called instead of *ASS_INPUT* in case we want to use the dynamic simulator. It assigns the value of the input variables of the plant depending on certain conditions imposed to some fictitious variables representing the position of the elevator and of the doors.

HMI : it is the program that was implemented for the usage of the human-machine interface. It permits the correlation between the HMI and the variables defined in CoDeSys.

MANUAL : it is the program needed to activate the manual mode. In this configuration the technician, through the HMI, can act on the physical actuators of the plants freely.