Transfer Documents

Project Let's Move It of the minor Adaptive Robotics

Project Members: Wouter Elfrink, Max Engels, Willem Hendriks, Joost Reintjes, Daan Vervoort & Berend Widlak
Project Tutor: Albert Aslan
Project Client: Michiel van Osch
Minor: Adaptive Robotics
Institute: Fontys Hogescholen
City: Eindhoven
Date: June 2019

Index

1 Introduction

The RAAK MKB project Let’s move it focusses on automation of manufacturing processes for SME’s, specifically in coordinating logistics of mobile robots in a manufacturing environment. The project was approved in June 2017. Already several student projects have been executed within Let’s Move it in which amongst others a trial has performed with a MIR robot that interacts with environment devices and a robot storage system has been developed that needs to deliver goods to a mobile robot.

1.1 Problem description

Mobile robots are used more and more used in a manufacturing or warehousing environment. Currently there are different suppliers of these types of robots like OMRON, Knapp, etc. However, coordination between robots (from different manufacturers), co-operation between robots and interaction of these robots with the environment (e.g. docking to machines, opening doors on the work floor, inspecting the environment for unwanted situations) is still very limited. The goal of the Let’s move it project is to work on an integrated solution for robust use of multiple mobile robots in an SME manufacturing environment.

1.2 Assignment description

Within the Let’s Move It project an automated storage system/robot is has been developed which can take boxes containing e.g. screws or 3d printed parts from a shelf and hand it to a robot. This storage system is part of the lectoraat’s smart manufacturing environment in which robots from different manufacturers can take goods and deliver them e.g. to a robot cell or a worker. Also, in a lift system has been developed to take goods from robot height to a UR5 robot that is placed on a table. However, the boxes cannot yet be transported between storage system to the UR5 robot. For this purpose a MIR robot is available which needs to be made able to take boxes from the storage system to the UR5 cell and back. A transport system needs to be developed that takes in several boxes and can deliver them at several places in the correct order.
This system needs to be designed, build and tested in the actual smart manufacturing environment within the lectoraat.

1.3 Objectives

Develop a new transport system on top of the MIR robot to take storage boxes from the robot storage system:

- Deliver storage boxes to the UR5 lift system
- Take storage boxes from the UR5 lift system
- Deliver storage boxes to the automated storage system
- Move between storage system and lift
- Should handle at least 3 boxes at once
- Would need to handle at least 5 boxes at once
- Deliver storage boxes in the right order
- Is safe
- Demonstrate the working of the transport system in lab setting

2 MiR100

The MiR100 is an AGV that we use in the project Let's Move It. The MiR drives around to deliver storage boxes from the automated storage system to the dockingstation and the other way around.
To deliver and receive the storage boxes with the add-on on top of the MiR there has to be a very precise positioning as shown on figure 2-1 & 2-2. Therefore, the MiR uses a L-marker (figure 2-3), with this L-marker the MiR can position himself between a range of 5mm which is enough for our system.

The L-marker is a static marker in the map, and it is very important that is on the same place all the time. In our project the marker is used in front of the dockingstation and in a automated storage system but isn't fixed to this two systems. As soon as the position of the L-marker is final, a bracket can be made on these two systems so that the L-marker is always at the correct position in relation to them and can be removed.

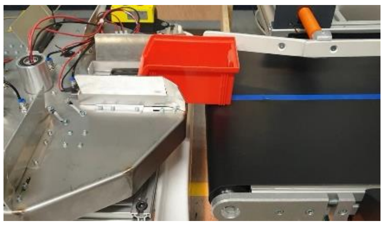


Figure 2-1: interface between MiR and dockingstation

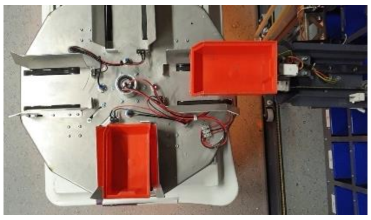


Figure 2-2: interface between MiR and automated storage system

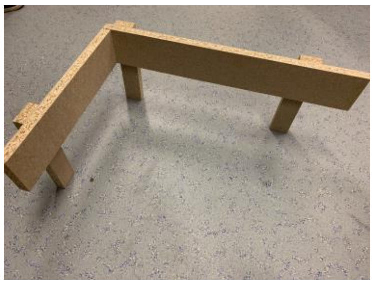
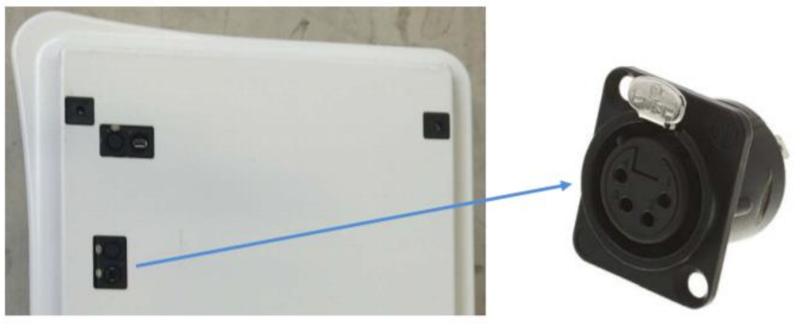


Figure 2-3: L-marker

On top of the MiR is a 4 pins connector (figure 1-4), pin 1 of this connector is always supplying the raspberry pi on the Carousel when the MiR is turned on. If the MiR is too close to an obstacle the emergency stop will switch on, pin 3 that supplies all the electrical stuff on the Carousel except the raspberry will be turned off. Same story if someone pushes the emergency button. In this way there is always a connection with the raspberry and the program always remains running but still the motors shut down by emergency system.



Pin number	Signal name	Max. current	Remarks
1	Battery voltage	3A	Always on
2	Battery voltage	3A	Starts with the robot
3	Battery voltage	10A	Stops by emergency stop
4	GND	10A	Ground

Figure 2-4: MiR connector.

3 MiR Carousel

We designed an add-on that we can place on top of the MiR100. This add-on has to deliver and receive storage container that we can move between the automated storage system and the docking station. At the beginning we had a few requirements that we had to meet with this add-on. We had the following requirements. Blue is must, green is should, orange is could.

MiR	**M2.1**	The MiR must take storage boxes from the automated storage system.
	M2.2	The MiR must deliver storage boxes to the UR-5 lift system.
	M2.3	The MiR must take storage boxes from the UR-5 lift system.
	M2.4	The MiR must deliver storage boxes to the automated storage system.
	M2.5	The MiR must be able to move between the automated storage system and the UR-5 lift system.
	M2.6	The MiR must be able to handle at least 1 box at the time.
	M2.7	The MiR must be able to deliver storage boxes in the right order.
	M2.8	The MiR must be demonstrating the working of the transport system in lab setting.
	S2.9	The MiR should be able to handle at least 3 boxes at once.
	C2.10	The MiR could be able to handle at least 5 boxes at once.

Table 1: The requirements for the add-on on top of the MiR100.

After we made these requirements we came up with a three concepts. We will briefly show each concept.

3.1 Concept 1: Storage

The first concept we came up with was a add-on on top of the MiR which has an storage. With this concept, the storage boxes are stored next to each other and above each other, resulting in a large capacity. The automated storage system takes the box from the warehouse and delivers it to the desired height and width compared to the MiR. The MiR then travels a little backwards and the automated storage system puts the container in storage. The MiR is driven forward again and the cycle for placing one storage box has ended. There is a conveyor belt under every storage box in the warehouse of the MiR. The moment the MiR is in front of the docking station, the lift must go to the desired height of the storage box. The conveyor belt of the elevator starts rotating and from then on the storage box can be delivered. In figure 3.1 there is a drawing of the concept.

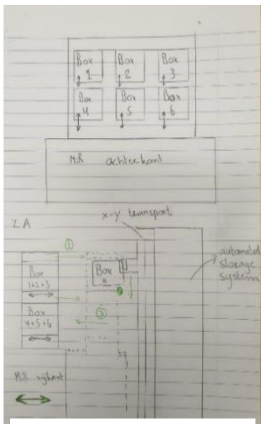


Figure 3-1: The drawing of the concept with the

3.2 Concept 2: Transfer

We use five conveyer belts for this concept. A conveyer belt serves as the input / output of storage boxes. In addition, we have three more conveyer belts where we can store boxes. So that the MiR-Robot can transport multiple boxes at the same time. To be able to move boxes to another conveyer belt, we need a transfer, a fifth conveyer belt that runs along the other conveyer belts on linear bearings. This transfer can position at the height of all conveyer belt. And then delivers or receives boxes to/from the storage. In this way we can start work with storage containers on the MiR robot based on the requirements that we have in chapter 3. In figure 3.2 you see a drawing of this concept.

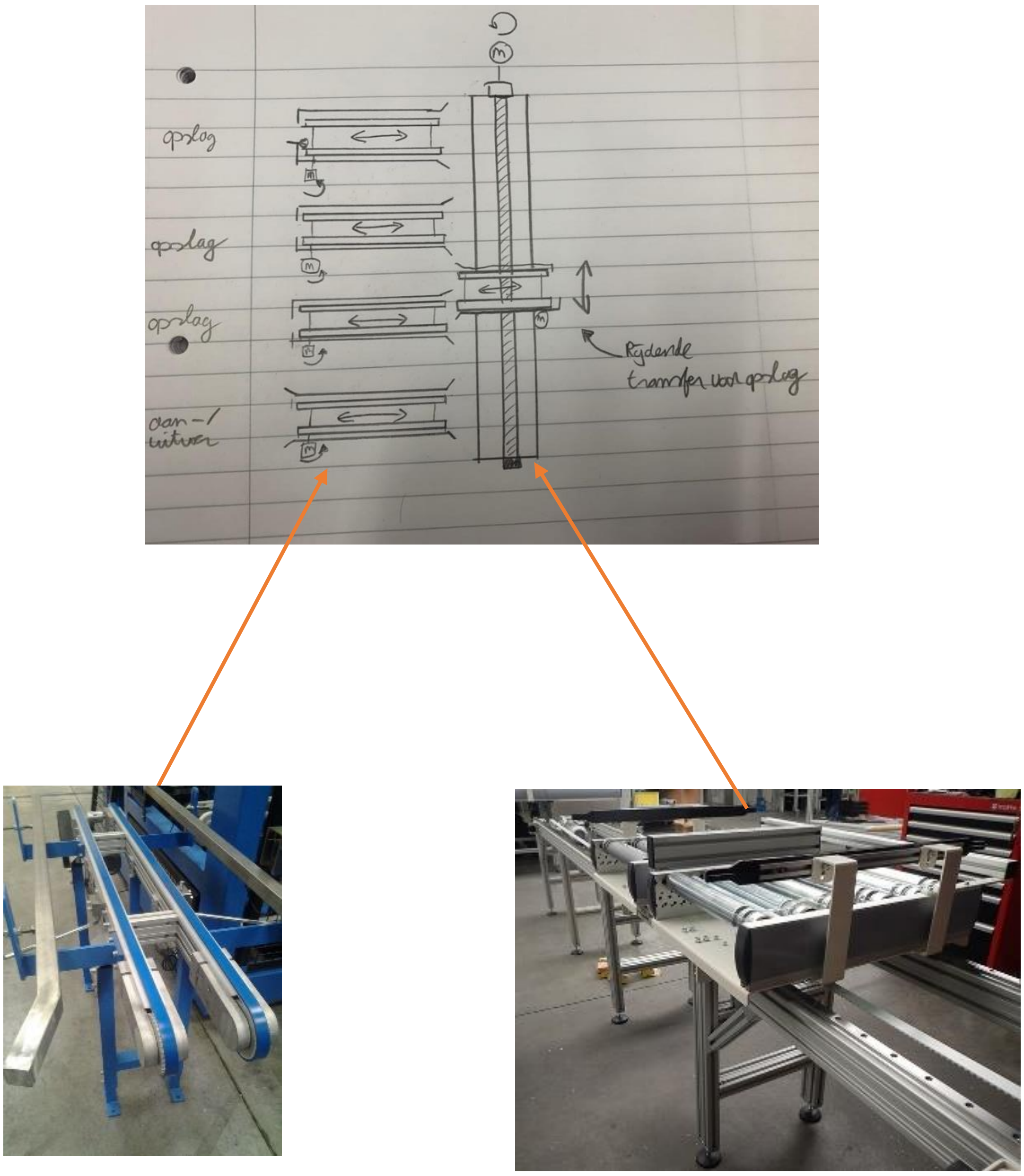


Figure 3-2: The drawing for the concept with the transfer.

3.3 Concept 3: Turntable

For this concept we use a turntable with the size of the MiR. The advantage of this is that we do not need extra input / output with this concept. Because the containers on timing belts are placed on the disk by the warehouse robot. First of all we came up with the idea to make a round disk with space for storing containers. After thinking about this concept for a while, we discovered that a round disc is not ideal. This is because there is still a gap between the end of the toothed belts and the edge of the disc (see figure 3.3). Because of this space we think we will have problems with the receiving / delivering of containers. That is why we came up with the idea of making a turntable in the shape of a hexagon. This eliminates the awkward space that we previously had with the round turntable. Another additional advantage is that we can now store a maximum of four containers instead of three. This is because we have more room for the technology of the conveyor belts to underneath the turntable thanks to the hexagon shape.
After looking at all the concepts and comparing these with the requirements we choose to go further with this concept. Because in our eyes we can get a higher capacity with this system compared to the other two concepts. Next to that, this concept is easier build and work together with the automated storage and docking station. Because with the timing belts on the turntable we can receive and deliver storage boxes. And so don't we have to make modifications to the other automated storage and docking station.

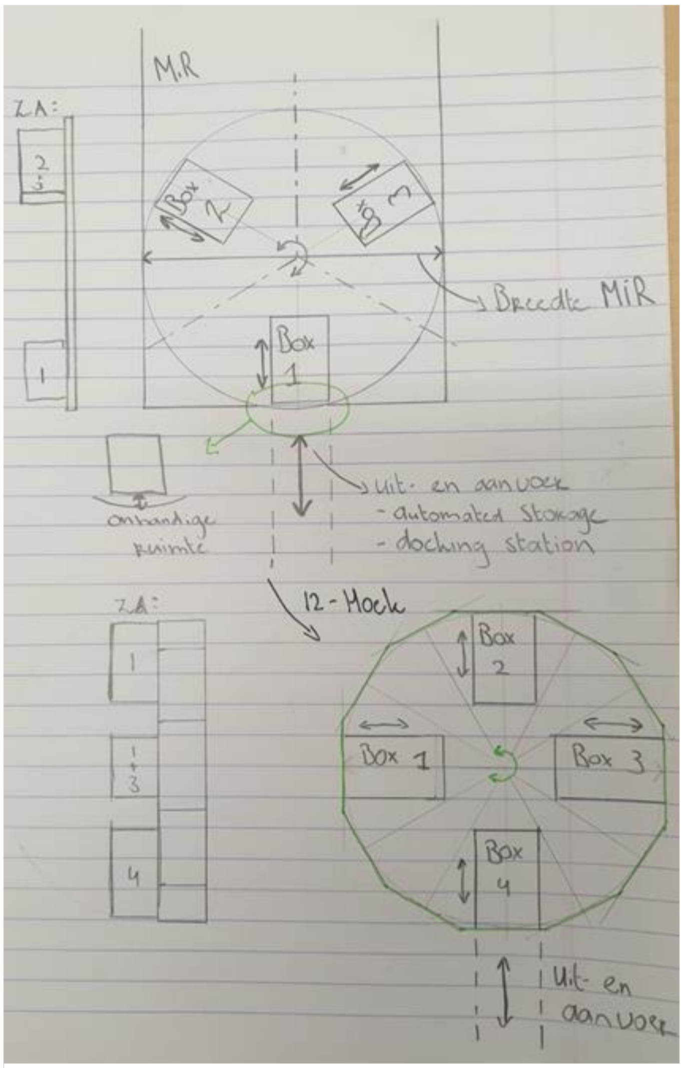


Figure 3-3: The drawing for the concept with the turntable.

3.4 Mechanical design

So we had made the choice that we would go further with concept 3. The next step was to model the whole turntable in 3D. We did this in Solidworks. First, we made the base with aluminium extrusion. After that we modeled the turntable and axis where the turntable is resting on. The next step was to implement the timing belt and motors. To deliver and receive storage containers. Then we added some electronics like limits switches and proximity sensor and stepper motor. To finish the design and make the turntable rotate we have designed gears. Now the mechanical design was done. This all went very smooth and within 1,5 week the design was ready to be manufactured. We have placed all the 2D, 3D-models and Bill of Materials on our Github page. You can find it at the following address: https://github.com/wouter291/Lets-Move-It-Turntable-Drawings
In figure 3-4 you can see the end result.

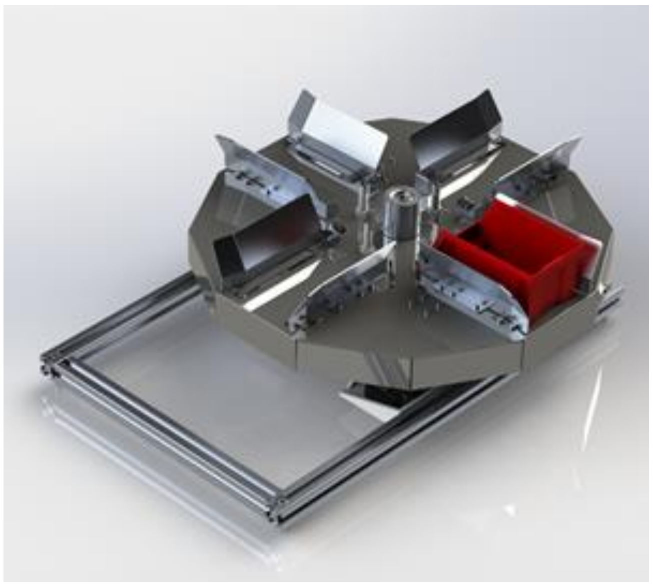


Figure 3-4: A render of the final mechanical design.

3.5 Software design

If you want a properly working system, good software must be written. The software that is written for this project will be discussed in this chapter.

3.5.1 Flowchart

Before the software code itself is explained, we first look at a flowchart that shows the structure of the code. In figure 3-5 a flowchart of the software code for the carousel is represent.

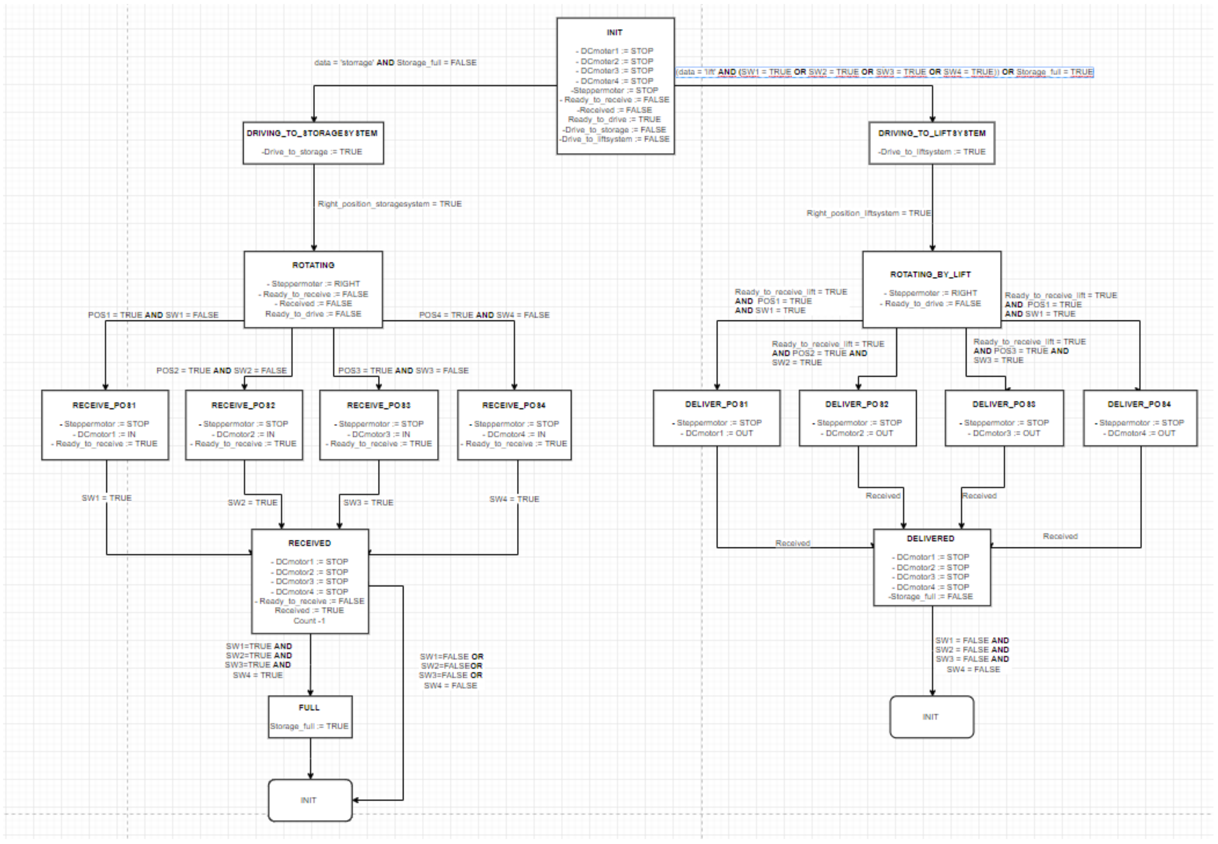

INIT
- DCmotor1 := STOP
- DCmotor2 := STOP
- DCmotor3 := STOP
- DCmotor4 := STOP
-Steppermotor := STOP
- Ready_to_receive := FALSE
-Received := FALSE
Ready_to_drive := TRUE
-Drive_to_storage := FALSE
-Drive_to_liftsystem := FALSE
data = 'storage' AND Storage_full = FALSE
(data = 'lift' AND (SW1 = TRUE OR SW2 = TRUE OR SW3 = TRUE OR SW4 = TRUE)) OR Storage_full = TRUE
DRIVING_TO_STORAGESYSTEM
-Drive_to_storage := TRUE
Right_position_storagesystem = TRUE
ROTATING
- Steppermotor := RIGHT
- Ready_to_receive := FALSE
- Received := FALSE
Ready_to_drive := FALSE
POS1 = TRUE AND SW1 = FALSE
POS2 = TRUE AND SW2 = FALSE
POS3 = TRUE AND SW3 = FALSE
POS4 = TRUE AND SW4 = FALSE
RECEIVE_POS1
- Steppermotor := STOP
- DCmotor1 := IN
- Ready_to_receive := TRUE
RECEIVE_POS2
- Steppermotor := STOP
- DCmotor2 := IN
- Ready_to_receive := TRUE
RECEIVE_POS3
- Steppermotor := STOP
- DCmotor3 := IN
- Ready_to_receive := TRUE
RECEIVE_POS4
- Steppermotor := STOP
- DCmotor4 := IN
- Ready_to_receive := TRUE
SW1 = TRUE
SW2 = TRUE
SW3 = TRUE
SW4 = TRUE
RECEIVED
- DCmotor1 := STOP
- DCmotor2 := STOP
- DCmotor3 := STOP
- DCmotor4 := STOP
- Ready_to_receive := FALSE
Received := TRUE
Count -1
SW1=TRUE AND SW2=TRUE AND SW3=TRUE AND SW4 = TRUE
SW1=FALSE OR SW2=FALSEOR SW3=FALSE OR SW4 = FALSE
FULL
Storage_full := TRUE
INIT
DRIVING_TO_LIFTSYSTEM
-Drive_to_liftsystem := TRUE
Right_position_liftsystem = TRUE
ROTATING_BY_LIFT
- Steppermotor := RIGHT
- Ready_to_drive := FALSE
Ready_to_receive_lift = TRUE AND POS1 = TRUE AND SW1 = TRUE
Ready_to_receive_lift = TRUE AND POS2 = TRUE AND SW2 = TRUE
Ready_to_receive_lift = TRUE AND POS3 = TRUE AND SW3 = TRUE
Ready_to_receive_lift = TRUE AND POS1 = TRUE AND SW1 = TRUE
DELIVER_POS1
- Steppermotor := STOP
- DCmotor1 := OUT
DELIVER_POS2
- Steppermotor := STOP
- DCmotor2 := OUT
DELIVER_POS3
- Steppermotor := STOP
- DCmotor3 := OUT
DELIVER_POS4
- Steppermotor := STOP
- DCmotor4 := OUT
Received
Received
Received
Received
DELIVERED
- DCmotor1 := STOP
- DCmotor2 := STOP
- DCmotor3 := STOP
- DCmotor4 := STOP
-Storage_full := FALSE
SW1 = FALSE AND SW2 = FALSE AND SW3 = FALSE AND SW4 = FALSE
INIT

Figure 3-5: Flowchart total system

As you can see in figure 1, the program is divided into receiving and delivering a storage box. First look to the receiving part. To get a clearer view, the receiving part is zoomed in on figure 3-6.

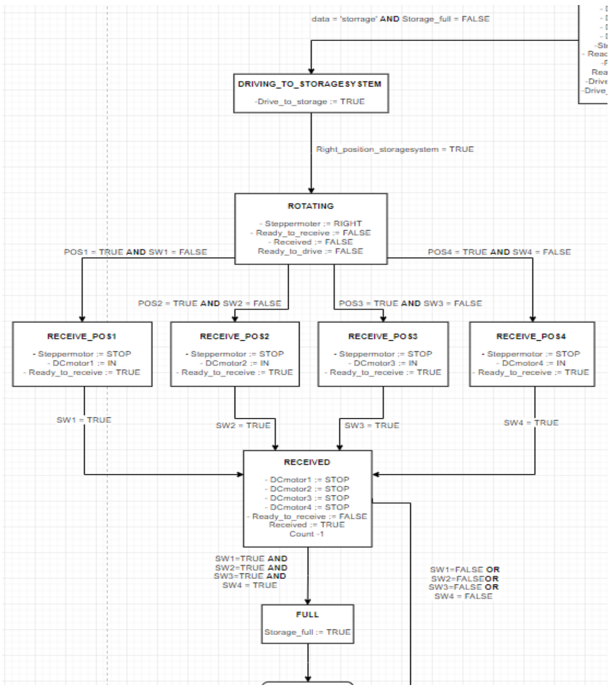


Figure 3-6: receiving part of flowchart

When the carousel gets a recieve command from the server, and the storage is not full yet. The program jumps into the state DRIVE_TO_STORAGESYSTEM. When the AGV is on the right position in front of the storage system, the carousel can start rotating. This is the state ROTATING. When the carousel is on an empty position, the carousel stops with rotating and the conveyer belt starts turning in on the right position. This happens in the states RECEIVE_POS1, RECEIVE_POS2, RECEIVE_POS3 or RECEIVE_POS4. The SMART storage can now deliver a storage box. When the box is on the carousel, it moves towards the axes of the carousel. When the storage box hits the microswitch, the conveyor belt stops turning in. This happens in state RECEIVED. Now the program gets back to the INIT state, unless all the microswitches are true. Then the program gets into state FULL, and the variable Storage_full becomes true. In this way the program can no longer jump into the receive part. This was the receiving part, now let's look to the delivering part. To get a clearer view, the delivering part is zoomed in on figure 3.7.

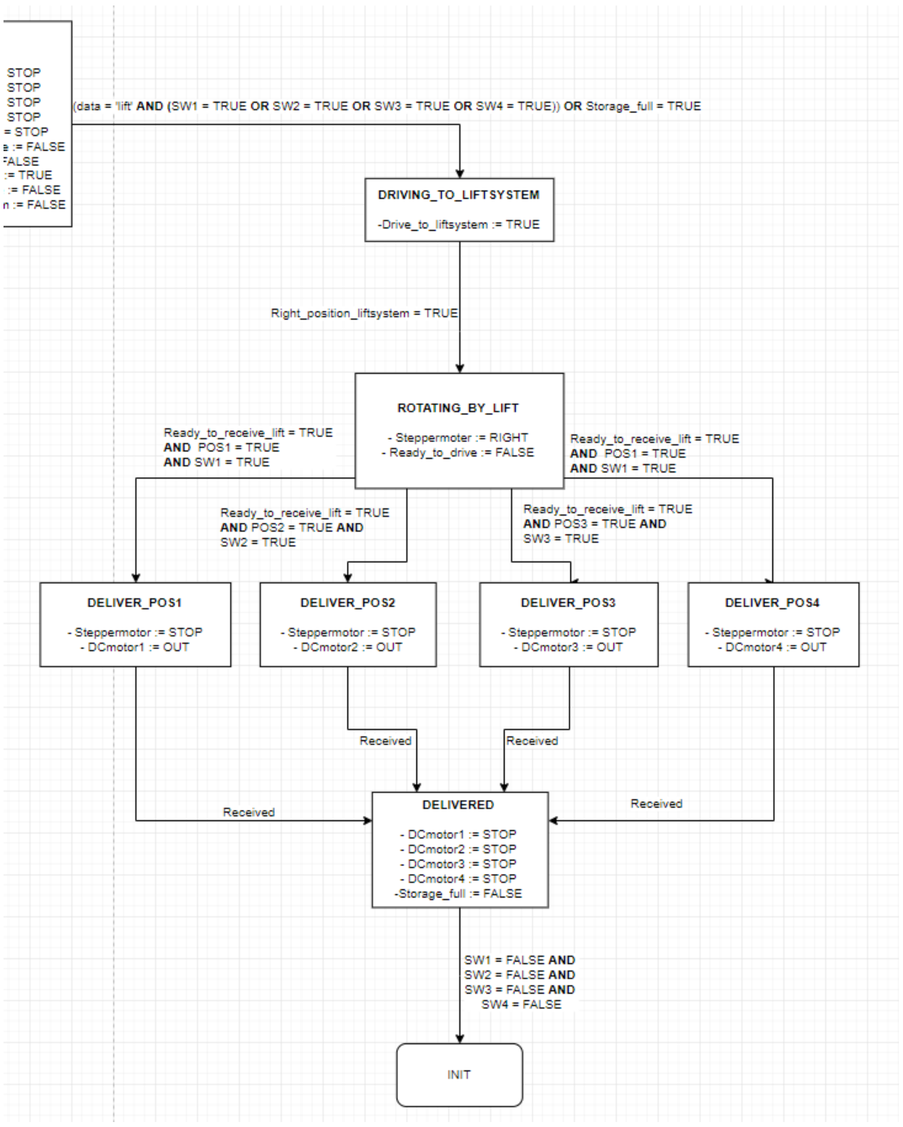
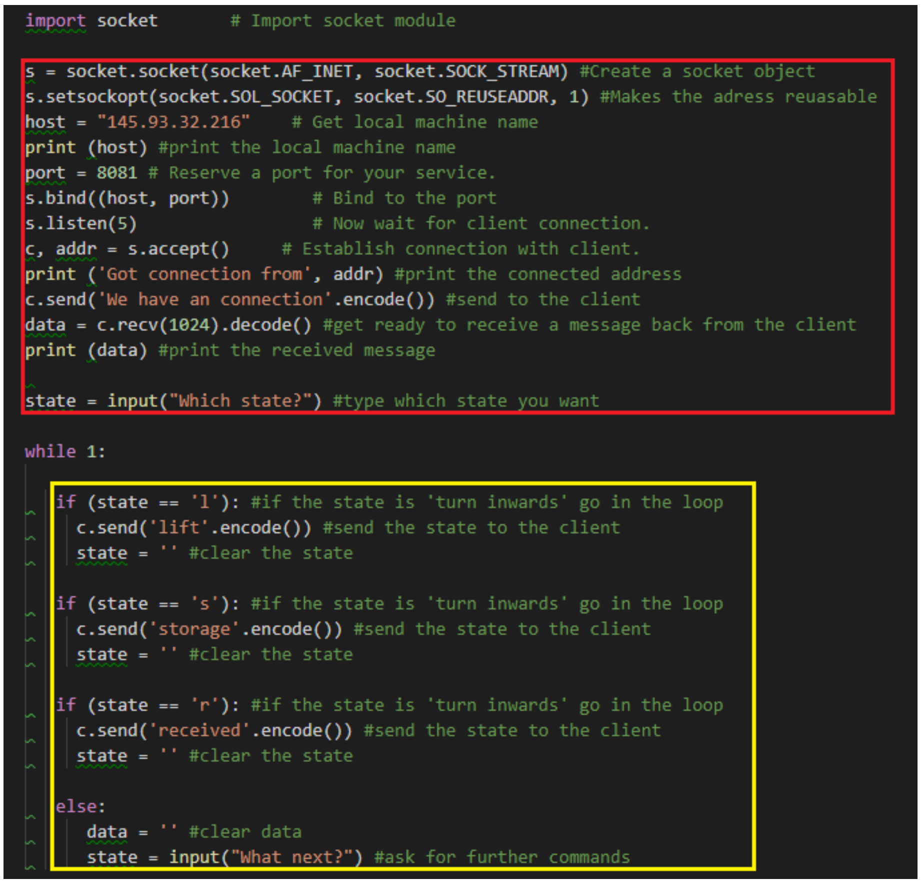


Figure 3-7: Delivering part of flowchart

When the carousel gets the command deliver, and there are some boxes on the carousel. The program jumps into the state DRIVE_TO_LIFTSYSTEM. When the AGV is on the right position in front of the lift system, the carousel can start rotating. This is the state ROTATING_BY_LIFT. When the carousel is on a full position, the carousel stops with rotating and the conveyer belt starts turning out on the right position. This happens in the states DELIVER_POS1, DELIVER_POS2, DELIVER_POS3 or DELIVER_POS4. The carousel delivers now a storage box to the lift system. When the carousel gets a received signal from the lift system, the conveyor belt stops turning out. This happens in state DELIVERED. In this state, the variable storage_full becomes false. Because the carousel is able again to receive a storage box. After this, the program gets back to the INIT state. Now the flowchart is explained, the implementation in the code is discussed.

3.5.2 Code explanation

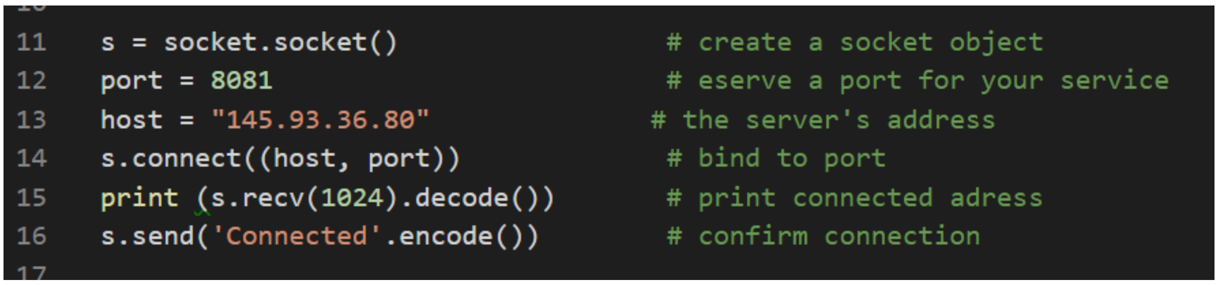
For make the system work properly, there is made a software code. This software code consists out of a main program and a server who can give commands to the main program. The main program consists of some different states, the program starts when a command is given from the server. First, let's look on the server program. (figure 3-8)



```python
import socket       # Import socket module

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #Create a socket object
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #Makes the adress reuasable
host = "145.93.32.216"    # Get local machine name
print (host) #print the local machine name
port = 8081 # Reserve a port for your service.
s.bind((host, port))        # Bind to the port
s.listen(5)                 # Now wait for client connection.
c, addr = s.accept()     # Establish connection with client.
print ('Got connection from', addr) #print the connected address
c.send('We have an connection'.encode()) #send to the client
data = c.recv(1024).decode() #get ready to receive a message back from the client
print (data) #print the received message

state = input("Which state?") #type which state you want

while 1:

    if (state == 'l'): #if the state is 'turn inwards' go in the loop
        c.send('lift'.encode()) #send the state to the client
        state = '' #clear the state

    if (state == 's'): #if the state is 'turn inwards' go in the loop
        c.send('storage'.encode()) #send the state to the client
        state = '' #clear the state

    if (state == 'r'): #if the state is 'turn inwards' go in the loop
        c.send('received'.encode()) #send the state to the client
        state = '' #clear the state

    else:
        data = '' #clear data
        state = input("What next?") #ask for further commands
```

Figure 3-8: Server program

In the red area you see the part of the code that ensures communication with the main program of the carousel. The connection can be made by using a socket IP. The host always must be the IP-address of the server. A port between 0 and 65535 can be selected. The client part is in the main

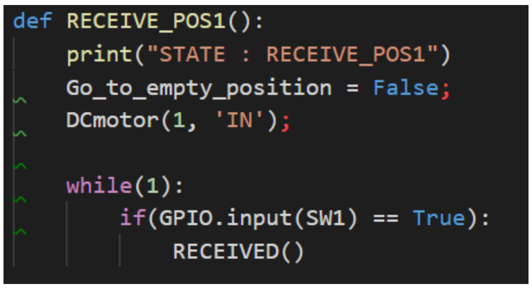


```python
s = socket.socket()                 # create a socket object
port = 8081                         # eserve a port for your service
host = "145.93.36.80"              # the server's address
s.connect((host, port))             # bind to port
print (s.recv(1024).decode())       # print connected adress
s.send('Connected'.encode())        # confirm connection
```

Figure 3-9: Communication part in main program

program. The yellow area shows the part that sends the commands to the main program. In this way, commands can be sent to the main program. Now that it is known how commands can be sent, the main program can be discussed. In figure 3-9 you see the part of the main code that ensures communication with the server program.

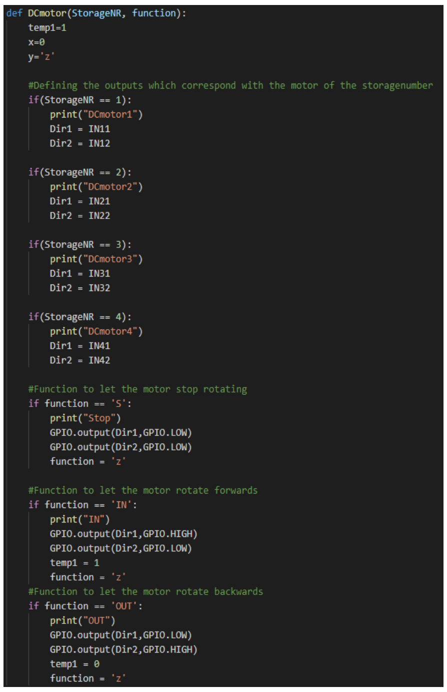
The main program works with states. It starts with the INIT state end when a certain condition is true, the program jumps into another state. In figure 3-10 you see an exaple.



```
def RECEIVE_POS1():
    print("STATE : RECEIVE_POS1")
    Go_to_empty_position = False;
    DCmotor(1, 'IN');

    while(1):
        if(GPIO.input(SW1) == True):
            RECEIVED()
```

Figure 3-10: State in the main program

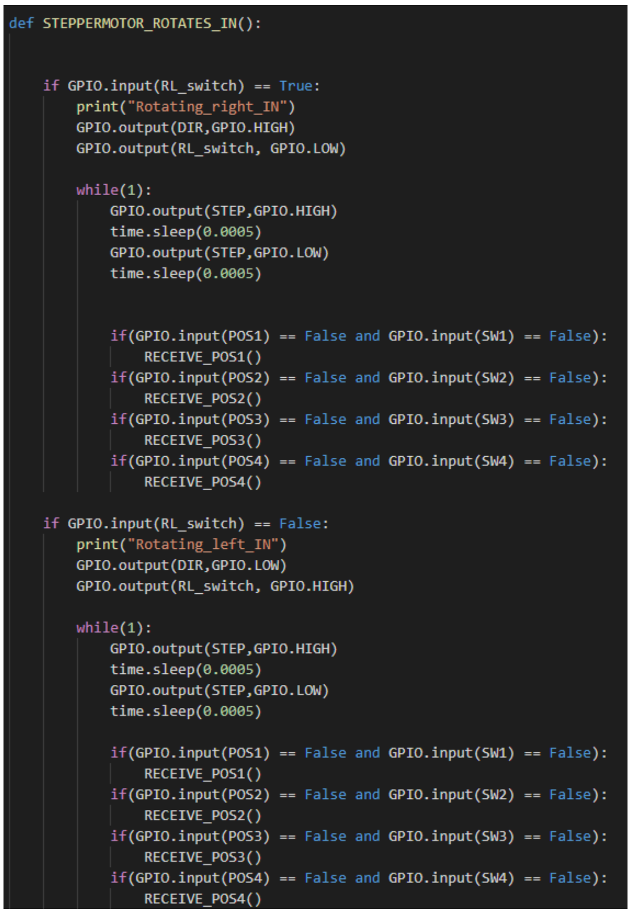
On the end of the code, there is a DC and stepper motor control. In figure 3-11 you see the DC motor controller.



```
def DCmotor(StorageNR, function):
    temp1=1
    x=0
    y='z'

    #Defining the outputs which correspond with the motor of the storagenumber
    if(StorageNR == 1):
        print("DCmotor1")
        Dir1 = IN11
        Dir2 = IN12

    if(StorageNR == 2):
        print("DCmotor2")
        Dir1 = IN21
        Dir2 = IN22

    if(StorageNR == 3):
        print("DCmotor3")
        Dir1 = IN31
        Dir2 = IN32

    if(StorageNR == 4):
        print("DCmotor4")
        Dir1 = IN41
        Dir2 = IN42

    #Function to let the motor stop rotating
    if function == 'S':
        print("Stop")
        GPIO.output(Dir1,GPIO.LOW)
        GPIO.output(Dir2,GPIO.LOW)
        function = 'z'

    #Function to let the motor rotate forwards
    if function == 'IN':
        print("IN")
        GPIO.output(Dir1,GPIO.HIGH)
        GPIO.output(Dir2,GPIO.LOW)
        temp1 = 1
        function = 'z'
    #Function to let the motor rotate backwards
    if function == 'OUT':
        print("OUT")
        GPIO.output(Dir1,GPIO.LOW)
        GPIO.output(Dir2,GPIO.HIGH)
        temp1 = 0
        function = 'z'
```

Figure 3-11: DC_motorcontrol

The StorageNR depends on the position where the conveyor belt needs to turn in or out. Further you have the options to stop, turn in or turn out the conveyor belt. At least the stepper motor control is discussed. In figure 3-12 you see the stepper motor control.



```
def STEPPERMOTOR_ROTATES_IN():


    if GPIO.input(RL_switch) == True:
        print("Rotating_right_IN")
        GPIO.output(DIR,GPIO.HIGH)
        GPIO.output(RL_switch, GPIO.LOW)

        while(1):
            GPIO.output(STEP,GPIO.HIGH)
            time.sleep(0.0005)
            GPIO.output(STEP,GPIO.LOW)
            time.sleep(0.0005)


            if(GPIO.input(POS1) == False and GPIO.input(SW1) == False):
                RECEIVE_POS1()
            if(GPIO.input(POS2) == False and GPIO.input(SW2) == False):
                RECEIVE_POS2()
            if(GPIO.input(POS3) == False and GPIO.input(SW3) == False):
                RECEIVE_POS3()
            if(GPIO.input(POS4) == False and GPIO.input(SW4) == False):
                RECEIVE_POS4()

    if GPIO.input(RL_switch) == False:
        print("Rotating_left_IN")
        GPIO.output(DIR,GPIO.LOW)
        GPIO.output(RL_switch, GPIO.HIGH)

        while(1):
            GPIO.output(STEP,GPIO.HIGH)
            time.sleep(0.0005)
            GPIO.output(STEP,GPIO.LOW)
            time.sleep(0.0005)

            if(GPIO.input(POS1) == False and GPIO.input(SW1) == False):
                RECEIVE_POS1()
            if(GPIO.input(POS2) == False and GPIO.input(SW2) == False):
                RECEIVE_POS2()
            if(GPIO.input(POS3) == False and GPIO.input(SW3) == False):
                RECEIVE_POS3()
            if(GPIO.input(POS4) == False and GPIO.input(SW4) == False):
                RECEIVE_POS4()
```

Figure 3-12: Stepper motor control

In this def you see two if statements. The reason for this is that the carousel not always can rotate in the same way, because the wires get stuck than. And then in the while loop you see the steps for the stepper motor. If the carousel is on a right position, it jumps into one of the four position states. Now the code is explained, we continue with the electrical design.

3.6 Electrical design

All the electrical parts on the carousel are connected with a Raspberry 3b+. The Raspberry runs a python scrip as shown one chapter before (software design). The Carousel has four identical storage locations, these are equipped with a limit switch, inductive sensor and DC-Drive. The limit switch let the Raspberry know if there is a storage box in the specific place. The inductive sensor let the Raspberry know in what orientation the Carousel is oriented.
A stepper motor, with stepper drive was used to turn the Carousel to different orientations and four DC-motors which are supplied by a H-Bridge are used to turn the storage boxes in- and outside.

The electrical design can be found on the following Github: https://github.com/wouter291/Lets-Move-It-Turntable-Drawings

4 Pick and Place system

For this project we are using the UR-5 robot arm for picking storage boxes from the docking station. When the lift of the docking station reached his upper position, the docking station will give the UR-5 a ready signal. With this signal the robot arm starts picking the storage box of the docking station. When the storage boxes are picked up, the robot will place the storage box at the conveyor belt on the UR-5 setup, see figure 4-1 For the UR-5 setup.

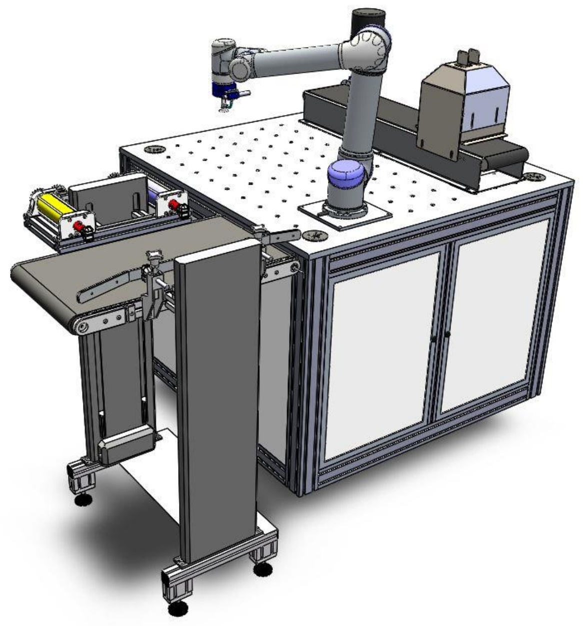


Figure 4-1: Pick and Place unit

4.1 Hardware

For picking the storage box a Shunk gripper has been used. For this gripper we've designed custom fingers for picking the storage box. See figure 4-2 for the Shunk gripper with the custom fingers. Custom grippers can be easily attached to the Shunk gripper.

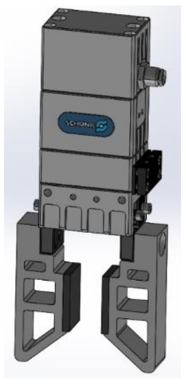


Figure 4-2: Gripper

4.2 Software

The software for the robot has been made in python with ROS. For installation of this software you will find some explanation on *Github,* see next link:

https://github.com/wouter291/LetsMoveIt-Pick-And-Place-UR5

5 Dockingstation

The dockingstation is a combination between a elevator and conveyor belt for transporting products up to a height. The task of the dockingstation in this project is to take cases from an AGV and transport them up to the table where an UR-5 is positioned where the UR-5 takes the case from the dockingstation.

5.1 Hardware

The hardware of the dockingstation consists of the original dockingstation part and the added parts by projects. This dockingstation (Figure 5-1) was custom made by Coomach in 2017 and was automatized in a project at Fontys by adding a PLC, multiple sensors and an press on guide. In this project the dockingstation was modified a bit and was integrated in this project by rewriting the whole software of the PLC and adding a raspberry pi for the external communication.

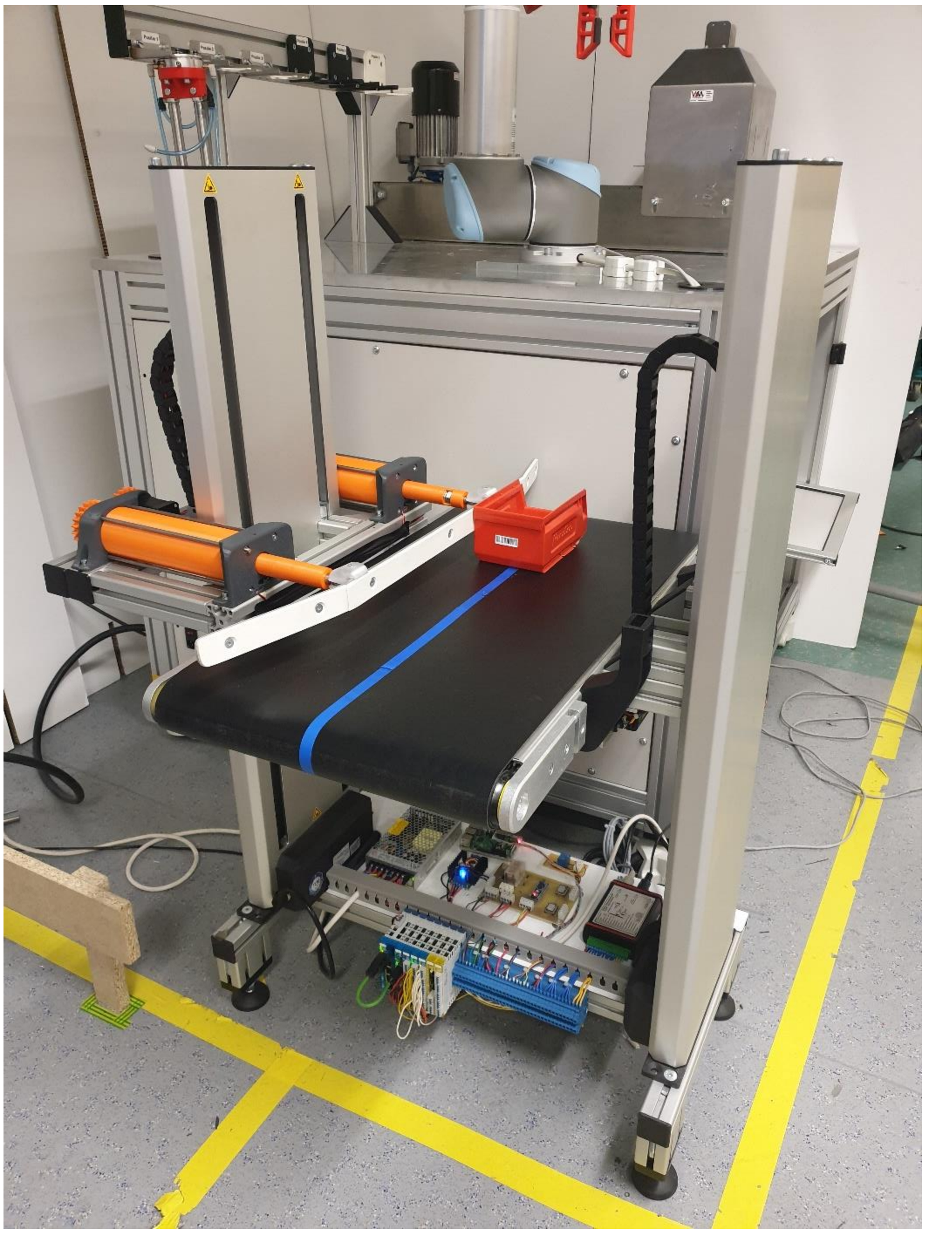


Figure 5-1: dockingstation

Most of the electrical scheme was already present when we started the project but to make it work inside our project a few modifications have been made and a raspberry pi has been added.

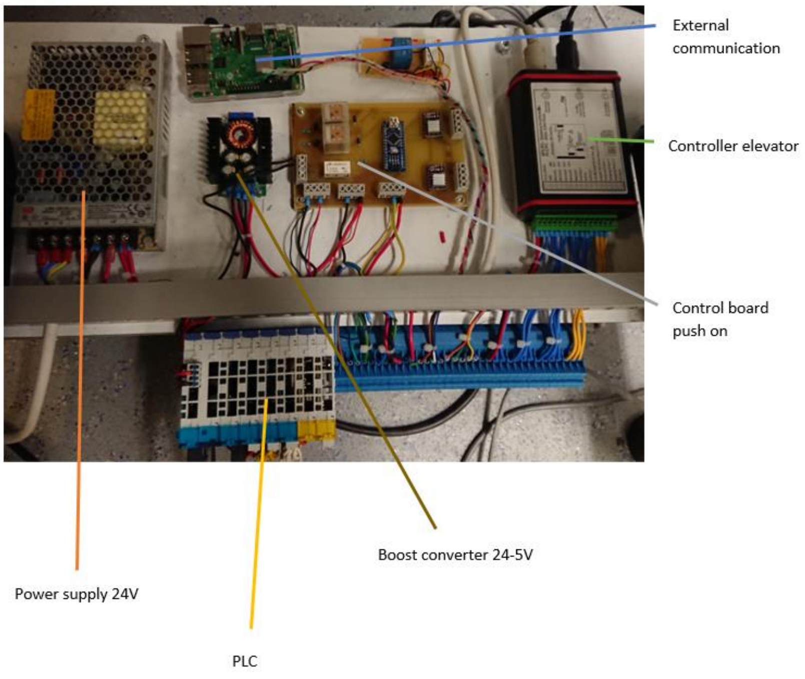


Figure 5-2: Electrical overview

The electrical scheme (Figure 5-2) consists of a power supply which generates 24 volts, a boost converter is installed on the dockingstation to also get a voltage of 5 volts.
Furthermore, there is a PLC from Sigmatek on the dockingstation which controls the elevator, the communication, the conveyor belt and the push on system on the dockingstation. The PLC consists of a ethernet module, a CPU, a power distribution, an input module, an output module, a safety CPU and safety in- and outputs (Figure 5-3).

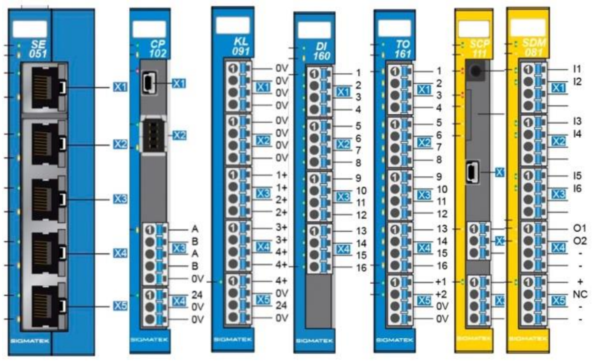


Figure 5-3: Sigmatek PLC

For the push on system a separate control board has been made because the stepper modules for the PLC were too expensive. This controller board consists of a microcontroller, stepper drivers and relays to communicate with the PLC because the PLC works on 24 volts and the controller board on 5 volts. The controller for the elevator controls the elevator part of the dockingstation. With this controller we can set pre-programmed positions in the memory. For the external communication a raspberry pi has been used with some relays. This has been chosen for wireless and easier communication.

5.2 Software

The software for the dockingstation has been made in Lasal class 2 en python. For using this software, you will find it on GitHub: https://github.com/bierent/LetsMoveIt--dockingstation
The software of the dockingstation is written in SFC and is designed with the next flowchart (Figure 5-4):

1. First the dockingstation begins in a position at about the middle of its range. It starts at that position because first the dockingstation has to take a case from the AGV. When the dockingstation is already in its intake height for the AGV the AGV can't drive onto the right position. While the AGV is positioning itself it drives a bit underneath the dockingstation thus the starting height.

2. In this position wait for a signal from the AGV which sends it to the raspberry pi via WiFi and the raspberry sends it via I/O to the PLC.

3. When it receives a message the dockingstation goes down to the height of the AGV.

4. With the dockingstation at the correct height it is ready to take in the case from the AGV. The dockingstation does this with its conveyor belt.

5. The dockingstation has a sensor installed upon the conveyor belt which detects if a case is at the end of the conveyor belt. With this sensor it can detect the case is on the right position and ready to be transported further.

6. With the sensor detecting a case it sends a message to the AGV via the raspberry pi that the case is received and the AGV can move on to its next position.

7. With the sensor detecting a case and sending a message to the AGV it goes up to the height position of the UR-5.

8. When the dockingstation is at desired height of the UR-5, the dockingstation sends a message to the UR-5 via the I/O on the PLC that it is in position.

9. When the UR-5 has taken the case from the dockingstation it sends a message back to the dockingstation that it has picked up the case. The dockingstation goes back to the home position after that.

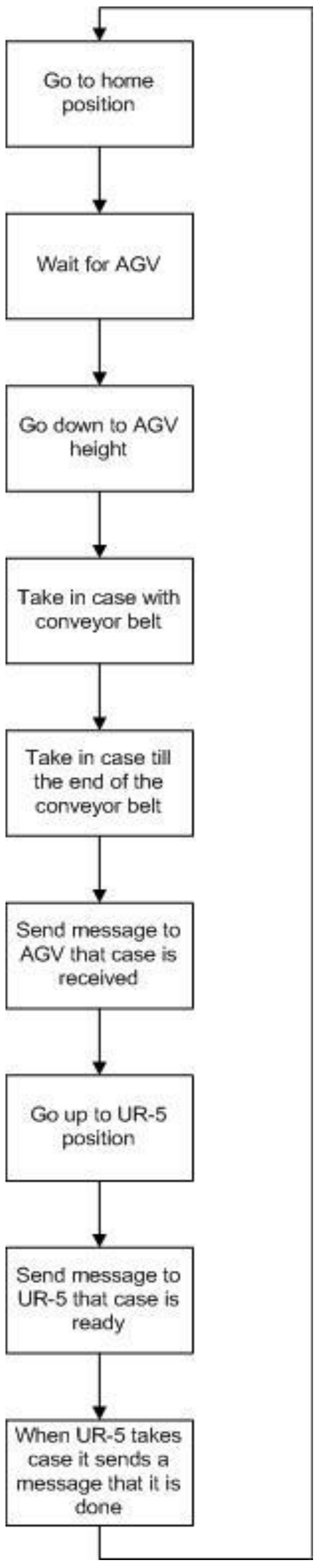
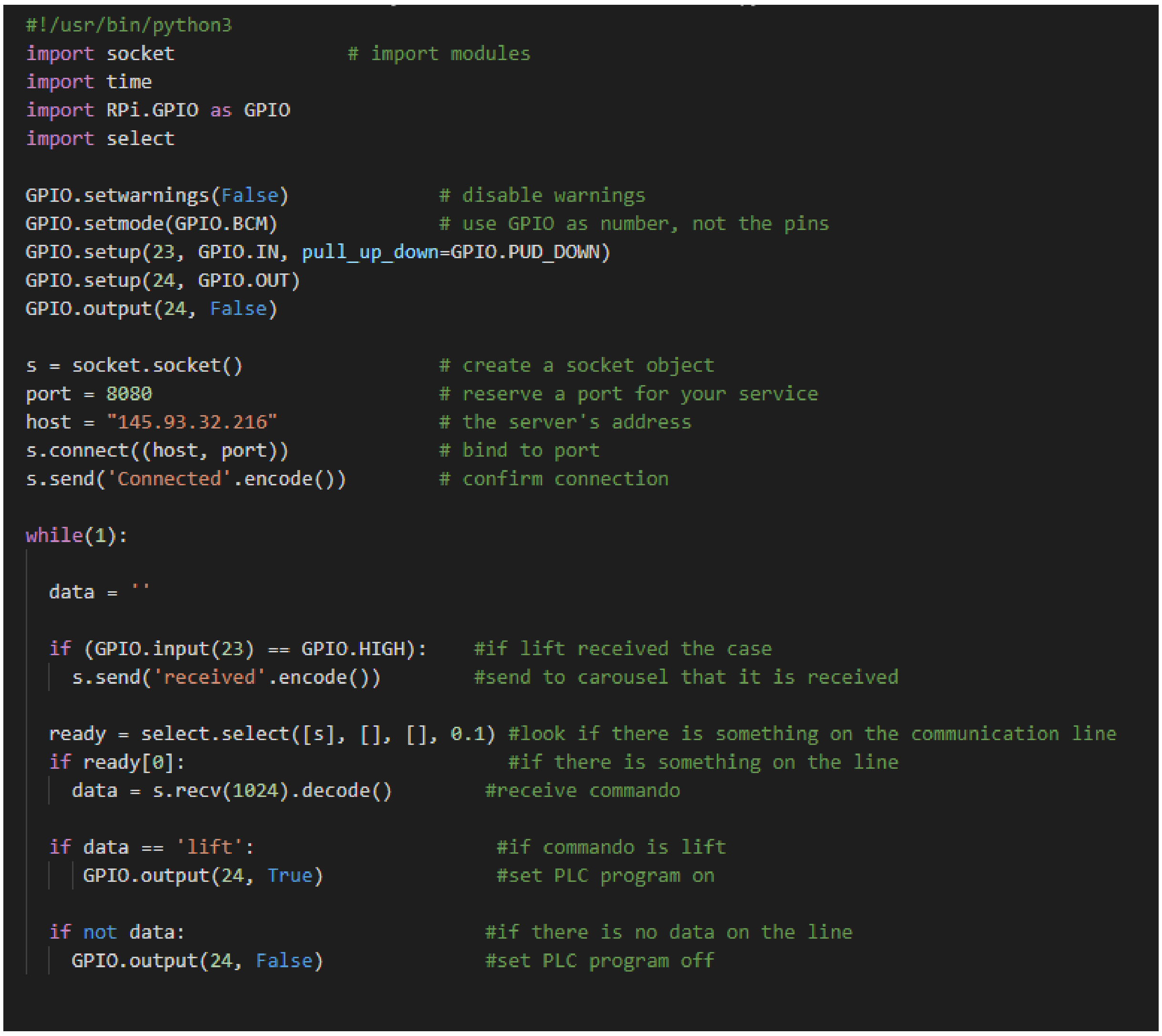

Go to home position
Wait for AGV
Go down to AGV height
Take in case with conveyor belt
Take in case till the end of the conveyor belt
Send message to AGV that case is received
Go up to UR-5 position
Send message to UR-5 that case is ready
When UR-5 takes case it sends a message that it is done

Figure 5-4: Flowchart

The software written in SFC can be found on the GitHub which was called upon earlier and is visually visible in Appendix . For letting the dockingstation and the carousel on the MiR work together a raspberry pi is added to the dockingstation. This is done for making the communication wireless and easier. The code for the raspberry pi is viewable in Figure 5-5.



```python
#!/usr/bin/python3
import socket               # import modules
import time
import RPi.GPIO as GPIO
import select

GPIO.setwarnings(False)             # disable warnings
GPIO.setmode(GPIO.BCM)              # use GPIO as number, not the pins
GPIO.setup(23, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(24, GPIO.OUT)
GPIO.output(24, False)

s = socket.socket()                 # create a socket object
port = 8080                         # reserve a port for your service
host = "145.93.32.216"              # the server's address
s.connect((host, port))             # bind to port
s.send('Connected'.encode())        # confirm connection

while(1):

  data = ''

  if (GPIO.input(23) == GPIO.HIGH):    #if lift received the case
    s.send('received'.encode())        #send to carousel that it is received

  ready = select.select([s], [], [], 0.1) #look if there is something on the communication line
  if ready[0]:                            #if there is something on the line
    data = s.recv(1024).decode()        #receive commando

  if data == 'lift':                     #if commando is lift
     GPIO.output(24, True)               #set PLC program on

  if not data:                          #if there is no data on the line
    GPIO.output(24, False)              #set PLC program off
```

Figure 5-5: Raspberry pi code

6 Recommendations follow-up project

Now we are finished with this project did learned a lot. We also earned a lot of experience during these 15 weeks. Prior to that we can come up with a lot of improvements and new ideas that can work together with the project we did. We will list recommendations below here in two categories: Improvements to our project and new project that can incorporate with our project.

Improvements to our project:

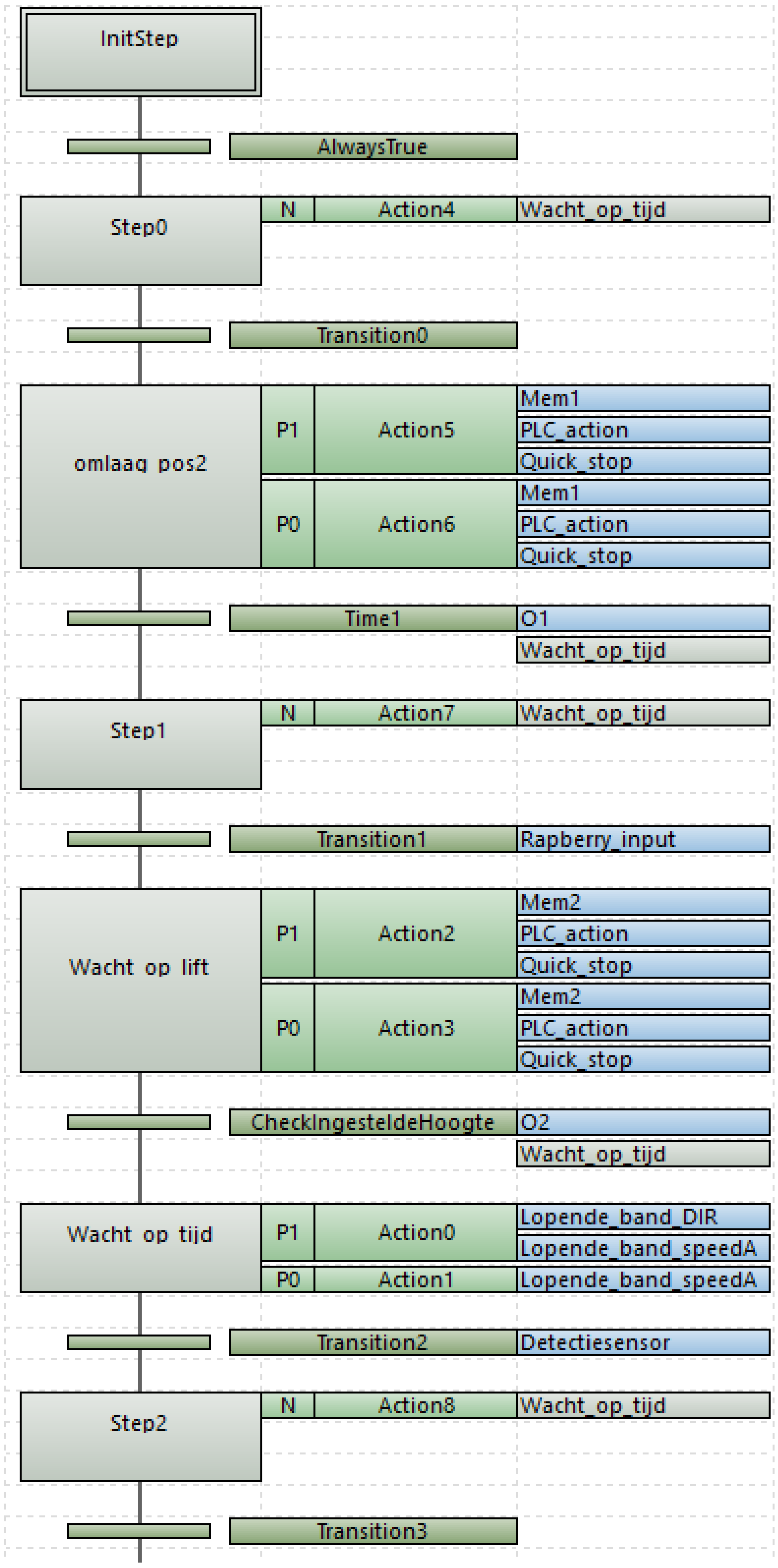
- Find a proper way to make the MiR100 communicate together with the storage and docking station. Now the signals that should come from the MiR are coming from a python script that is running on a laptop. It would be better to integrate the script from the laptop in the automated storage.
- Make a better gripper for the UR-5 robot arm so it can handle some weight. Now it isn't possible to even work with loads of 500 grammes.
- Programming the automated storage, docking station and UR-5 so you can bring back storage boxes back towards the automated storage to store them back.
- Improve the speed of the lift and Carousel
- Add vision system to the lift. So, you can position the Carousel on top of the MiR very precisely in X-position, Y-position and rotation of the Carousel.
- Add vision system to the automated storage to determine what the current stock is in the whole storage or in a single container.
- Replace the plastics gears under the Carousel with metal gears.
- Add a slipring to the Carousel for the power supply cables
- As soon as the position of the L-marker is final, a bracket can be made on the automated storage system and dockingstation. So that the L-marker is always at the correct position in relation to them and can be removed.

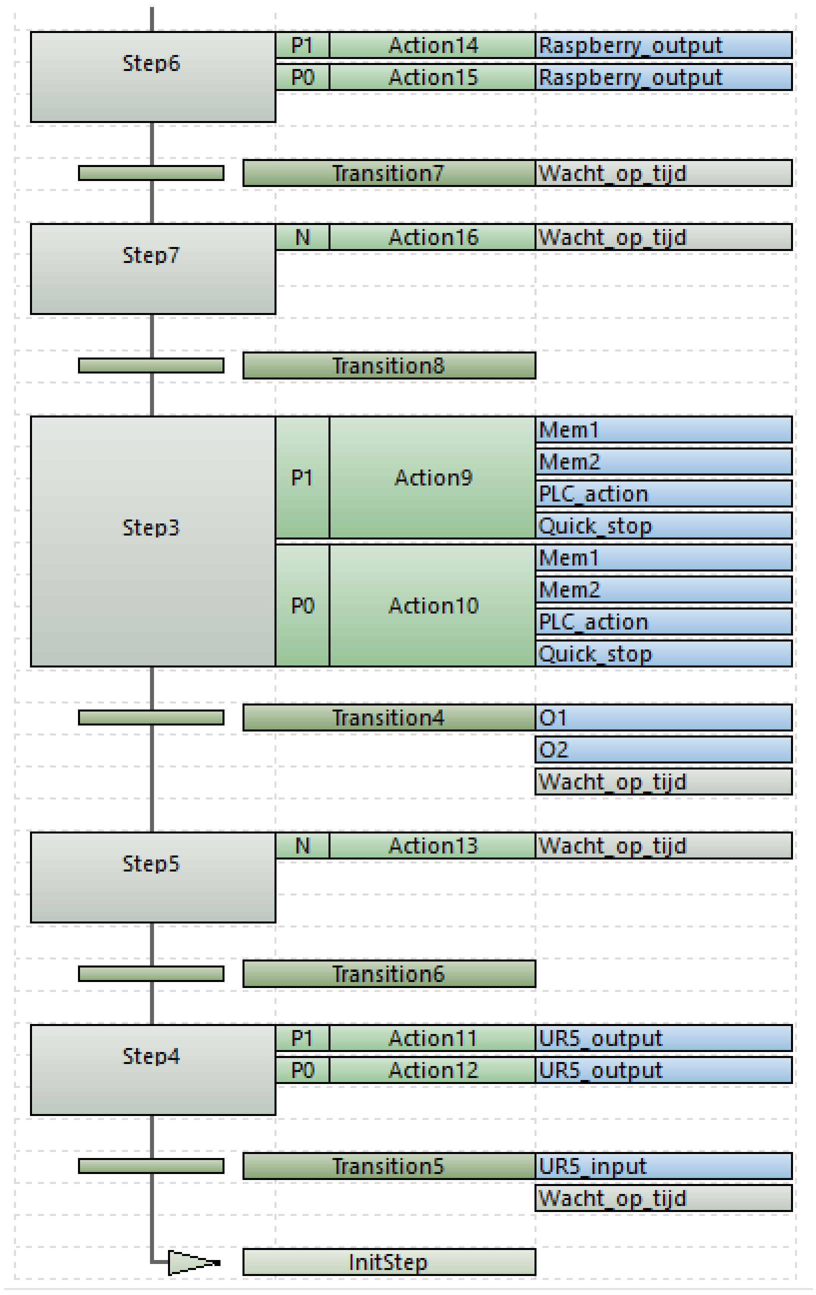
New project that can incorporate with our project:

- Add a system behind the UR-5 Robot arm that will handle product from the same containers as the automated storage. With this system you can store individual components such as a bolt in small boxes and palletize them. That the product is handled from a storage towards a package that is ready for shipment. In that way we have the same system as in a distribution center. Where you can pick your order from for example a web shop. And get packaged fully automatically and put on a small pallet or tray.
- Add an extra module where you can store parts for in the storage and storage container. From this module we can fill storage boxes with components, these will be transported to the automated storage to fill it with products.

7 Appendix

7.1 Appendix A




Step6
P1
Action14
Raspberry_output
P0
Action15
Raspberry_output
Transition7
Wacht_op_tijd
Step7
N
Action16
Wacht_op_tijd
Transition8
Step3
P1
Action9
Mem1
Mem2
PLC_action
Quick_stop
P0
Action10
Mem1
Mem2
PLC_action
Quick_stop
Transition4
O1
O2
Wacht_op_tijd
Step5
N
Action13
Wacht_op_tijd
Transition6
Step4
P1
Action11
UR5_output
P0
Action12
UR5_output
Transition5
UR5_input
Wacht_op_tijd
InitStep