CODEN:LUTEDX/(TEIE-5514)/1-68/(2024)

Industrial Electrical Engineering and Automation

Rapid Prototyping of Control Panel with Touch Display for Sliding Doors

Hampus Söderberg
Axel Nilsson

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

Rapid prototyping of control panel with touch display for sliding doors

Faculty of Engineering, Lund University

Division of Industrial Electrical Engineering and Automation

Axel Nilsson
Hampus Söderberg

September 9, 2024

Abstract

User interfaces to control machines are a critical part of any technology that requires user inputs. The user interface works as a link between human and machine, and it is vital that this link is understandable for both parts. Therefore, the design of the interface is of great importance in order to fully utilize the capabilities the technology provides. Assa Abloy is a company that offers solutions for entrance systems. The company's portfolio of operating mode selectors does not offer a touch based user interface between the entrance system and the user. This master thesis seeks to build a prototype of such an interface which is focused on satisfying the needs of the user. The prototype should implement a graphical user interface that works as a link between the user and the entrance system. To achieve this, the graphical user interface utilizes a back-end communication protocol, which is forwarding and translating user actions to the door. The graphical user interface is implemented in the python programming language and the communication protocol is implemented in the C programming language. The communication between these programs are done using an MQTT broker. The programs are implemented using a Raspberry Pi, which utilizes a touch screen through high speed DPI interface. The back-end communication protocol communicates to the entrance system using RS-485 standard. This resulted in a prototype, which in appearance and functionality resembles a finished product. The most important features of the prototype is that it can be connected to a sliding door and control the operating mode as well as receiving and displaying error information in the form of a error number, description and a solution. The results provides insights about the design of a new generation of touch based operating mode selectors.

To implement a design that focuses on the user experience, this project is done in parallel with another master thesis at Assa Abloy. This master thesis is focused on a UX-analysis of the current controller from the company and the insights gathered from the analysis is shared with our project. This offers interesting opportunities and influences the workflow of our project. Programming in industry is becoming more dependent on stakeholder feedback, and therefore workflows such as agile programming and extreme programming has been developed to account for this. To ensure that the opportunities the UX-analysis present are fully utilized, these techniques are used. In addition to develop a prototype of a new touch based user interface, this project serves as an example of how the agile programming methodology is beneficial to the results, when the initial specifications does not entirely specify the desired functionalities for a prototype.

Keywords

Prototyping, Agile programming, XP, Raspberry pi, Graphical user interface, MMI, Control panel.

Acknowledgements

We would like to thank Assa Abloy for giving us the opportunity to conduct our master thesis at the company. We both felt very welcome and were provided with a great working environment. During our stay we were given great support and encouragement. Special thanks to our two supervisors at the company, Roger Dreyer and Fredrik Brodje, for the insights and guidance. We also want to acknowledge Mathias Navne for the assist when establishing the communication to the door using the communication protocol created by the company.

We would also like to thank our supervisor at the department of Industrial Electrical Engineering and Automation at LTH, Samuel Estenlund, for the support during the project.

Finally we would like to thank Emma Liljenberg and Sofia Olsmats for the great collaboration and help during our closely linked projects at Assa Abloy.

Abbreviations

OMS : Operating Mode Selector

PCB: Printed Circuit Board

WSL: Windows Subsystem for Linux

RPi : Raspberry Pi

GUI: Graphical User Interface

MVC: Model-View-Controller

SBC: Single Board Computer

MCU: Micro Controller Unit

MMI: Man-Machine Interface

UI: User interaction

UX: User experience

MQTT: Message Queuing Telemetry Transport

CAD: Computer Aided Design

IoT: Internet of Things

HAL: Hardware Abstraction Layer

QoS: Quality of Service

XP: Extreme Programming

FPS: Frames Per Second

DPI: Display Parallel Interface

Contents

1 Introduction **1**
- 1.1 Problem description . . . 1
- 1.2 Background . . . 2
- 1.3 Previous work . . . 2

2 Theory **3**
- 2.1 Specifications . . . 3
  - 2.1.1 GUI Specifications . . . 3
  - 2.1.2 Physical Specifications . . . 3
- 2.2 Components . . . 3
  - 2.2.1 Raspberry Pi . . . 4
  - 2.2.2 Arduino uno . . . 4
  - 2.2.3 Touch display . . . 4
- 2.3 Theory about implementing a GUI . . . 4
  - 2.3.1 Program structure . . . 5
  - 2.3.2 Python libraries . . . 5
  - 2.3.3 Kivy . . . 6
  - 2.3.4 Tkinter . . . 6
  - 2.3.5 UX design . . . 7
- 2.4 Signal processing . . . 9
  - 2.4.1 RS485 . . . 9
  - 2.4.2 MQTT . . . 9
- 2.5 Programming workflow . . . 10

3 Methodology **11**
- 3.1 Selection of components . . . 11
- 3.2 Working setup . . . 12
- 3.3 Specifications for the GUI . . . 14
- 3.4 Design principles of the GUI . . . 17
- 3.5 Code structure . . . 19
  - 3.5.1 Structure of the model element . . . 21
  - 3.5.2 Structure of the view element . . . 24
  - 3.5.3 Structure of the controller element . . . 26
- 3.6 Communication . . . 34
  - 3.6.1 Assa Abloy communications protocol . . . 34
  - 3.6.2 MQTT . . . 34
  - 3.6.3 Hardware layer of the communication . . . 36
  - 3.6.4 Path of communication . . . 39
- 3.7 Casing . . . 40

4 Results **44**
- 4.1 GUI navigation and design . . . 44
- 4.2 Casing . . . 50
- 4.3 Communication . . . 50
- 4.4 Final result . . . 51

5 Discussion **53**

6 Conclusion **56**
6.1 Changing scope of project . . . 56
6.2 Acquired knowledge . . . 56
6.3 Future Work . . . 57

7 Appendix A - Gantt charts **60**

8 Appendix B - source code for project **61**

9 Appendix C **62**
9.1 Guide to use RPi and display to create a rapid prototype . . . 62

1 Introduction

This section gives a short description of the task and aim of this report. It also gives context to the reason for conducting this work by connecting it to the background and previous work done by the company in the same field.

1.1 Problem description

Assa Abloy currently offers solutions for a wide range of door systems. This project is targeted at the sliding door systems. The user interface used to control the door is a small controller that currently allows the user to change the operating mode of the door. This interface is produced in different variants. The selection of the operating mode is done by pressing buttons or inserting a key, and rotate it to the desired operating mode. The five different operating modes the door can be set to are those described in the list below.

- **Automatic** - The door opens automatically for people coming from both directions.
- **Closed** - The door is closed and does not open when the sensors detect an incoming person.
- **Auto Partial** - The door operates as in the automatic mode but the door leaves are partially opened, typically used during winter to reduce airflow.
- **Hold open** - The door is held in an open position.
- **Exit Only** - The door only opens for people exiting the building.

These controllers can also communicate when there is an error in the system. It is of interest to produce a touch based interface for the user to control the door. The current controller portfolio of Assa Abloy does not offer a touch based solution, and the development of such a product has the potential to benefit the company and broaden the selections of products of the entrance systems market.

The aim of the project is to make a rapid prototype of a touch display which lets the user interact with a sliding door. The main functionality of the product is to change the mode the sliding door is operating in. Other functionalities can also be added, for instance the user could alter parameters of the door. One example of this is to change the speed the door moves at or the time it remains in its open position before it starts closing again. Since the aim is to make a rapid prototype the main focus is at proving the concept and in a rapid manner give the company an indication of how such a product might work and interact with the sliding door. This means that the focus will not be on minimizing cost or creating the hardware ourselves, but instead on using available components to speed this process up.

The thesis work will be done in combination with another thesis being conducted in parallel at Assa Abloy. The aim of that thesis is to conduct a study of customer behavior regarding the current operating mode selectors available in the company's portfolio and what improvements could be made. Towards the end of our project these findings will be applied to our work in order to create a product that best suits the customer needs.

At the early stages of the project during discussions with the supervisors at the company the problem description was changed from the initial plan which was described in the target document. The company expressed that they would not benefit from us implementing a prototype which we would build ourselves from the bottom up, designing the printed circuit board (PCB) and deciding all the components. This was because the company would do this themselves and that it probably would differ a lot from what could be implemented in our master thesis. The early ideas of using a wireless connection was also put to the side since this no longer was a priority for the company.

1.2 Background

The current portfolio of Assa Abloy, visualized below, only consists of Operating Mode Selectors (OMS) which uses buttons or keys to change between the different operating modes. In recent years several competitors to the company have started offering an alternative OMS which has a touch display.

Figure 1: Currently available Assa Abloy OMS. On the left OMS Basic and on the right OMS Standard.

1.3 Previous work

A previous attempt of creating this product has been done within the company. This was done several years ago and was put on hold due to lack of success. The main reason it was not brought to market was the rapid change in the components available for this product. This meant challenges in ensuring the product would be up to date and would require several changes to be compatible with the components. Today this market is more mature and a larger variety of components are available meaning this is no longer seen as an obstacle.

2 Theory

This section aims to describe the theory used during the project. It includes the research done at the start of the project to know what direction and decisions to take. During the project, while faced with different challenges, more theory and information was needed which is also described in this section and later put into context under *Methology*.

2.1 Specifications

During the course of the project, discussions were taken with stakeholders at the company to create a list of specifications for the prototype. This included software specifications and functionality of the user interface as well as physical specifications of the hardware.

2.1.1 GUI Specifications

The graphical user interface (GUI) should be able to do several different things. While it is not in use it should display the current operating mode and a warning if active errors exists on the door. When the user desires to change something in the system, the screen should be tapped. When this action is done, the user enters mode-selection, which provides alternatives for five different operating modes. The user should also be able to change settings of the door, which is in a separate menu. If there is any active errors in the door, the user should be able to enter an error log that displays all current errors with descriptions and possible solutions.

Some settings will also require the user to enter a code in order to access the functionalities. An example of this is when the user wants to *closed* or accessing the settings which includes modification of the speed of which the door opens and closes. This is mandatory for sliding doors situated at emergency exits which 80-90 percent of the company's doors are [1].

2.1.2 Physical Specifications

The specifications related to the physical device were mostly related to the size. The company had an aim to be able to integrate the display together with other electrical installations like outlets and switches. As a result of this the width and height dimensions of the display should be smaller than 85 x 85 mm. Regarding the technical specifications nothing was stated other than that it should be able to run the program without problems.

2.2 Components

To begin the work different components were looked into which had to be ordered to construct the prototype. It was stated early on by the supervisors at the company that an Arduino or a Raspberry Pi should be used as the control unit. Apart from the control unit the other main component needed is a display with touch capabilities. The following sections delves deeper into the specifics of the choices of each controlling unit.

2.2.1 Raspberry Pi

A Raspberry Pi (RPi) is a single board computer (SBC) with an onboard operating system. It has most of the components used in a regular computer, but the difference is that all of the components fits on a single board. For the project, it was decided to focus the research on a Raspberry Pi 4 model b. The raspberry pi 4 has a RAM-memory ranging from 1-8 GB. The CPU is a 1.5 GHz quad core ARMv8-A. The device support USB-connection, ethernet and wifi-connections, Bluetooth and HDMI. It also supports serial communication by SPI, I2C and UART protocols [2]. The memory on the board is an SD-card that can be ordered separated from the Raspberry Pi which means that the user can decide on the capacity.

2.2.2 Arduino uno

The Arduino Uno is a microcontroller that utilizes the 16 MHz ATmega328P which is a low power 8-bit microcontroller from Atmel [3]. The Arduino Uno has a static random accessory memory (SRAM) of 2kb. It has a 32kb flash memory. The Arduino is based on open source hardware and software. This means that different developers can access the schematics of the Arduino and build custom designs based on individual needs.

2.2.3 Touch display

The display was required to be around the size of a standard electrical socket in order for the OMS to be mounted neatly in connection which other electrical installations. This meant that the display is limited to the size of 85x85 mm.

The display should be able to handle touch inputs. Both the options of using capacitive and resistive touch displays were regarded. A resisitve touch display registers input from the force applied on the surface, meanwhile a capacitive uses the change in an electrical field to detect the input. The advantage of using a resistive touch display is that it works with touch from any material since electrical properties of material are not a concern. The drawback of the solution is that it requires a force to be applied which makes it less responsive than using a capacitive display [4].

Regarding the properties of both these displays it was decided that a capacitive display would suit the situation best. It makes the application more intuitive and responsive and therefore makes the program more user friendly. The need of using the display with gloves on is limited and therefore not to be of high priority.

2.3 Theory about implementing a GUI

The following section contains the theory used to implement the GUI. Initially, research was done regarding different theories of how to implement an extensive user interface with several components and different menus. This was done in order to have a model for reference when programming the RPi and in order to achieve understandable code structure. Once this was decided, available libraries used for GUI development were investigated to find one suitable for our application. Finally UX and UI research was done in order to better understand how to structure the application in order to make it logical and relatable for the user. This research is presented later in this chapter.

2.3.1 Program structure

A graphical user interface can be structured in several different ways. One of the most simple variants is the smart UI approach [5]. This structure combines all the functionalities of the program into one class. An example of this approach would be a calculator which consists of 20-25 buttons, a textfield and some menus. The calculator could be implemented in one class, consisting of the GUI, the handling of user events an the logic for the mathimatical operations. For smaller applications it is considered the most feasible approach but as the program gets more extensive the structure of the code quickly becomes hard to manage and navigate. By not structuring the code in different classes and modules which has specific and clear purposes, both understanding and implementing the code becomes difficult. In the beginning of the project this approach was tested but it was realized that it would not suit the task of handling the GUI, user inputs and communication to the door, since the application would be more complex than this approach was suited for.

By further researching different well documented GUI application patterns, in particular the Model-View-Controller and Presentation-Abstraction-Control patterns, it was eventually decided to use the Model-View-Controller pattern (MVC) [5]. In a MVC approach the program is divided into three different components which are models, views and controllers and the components have their own area of responsibility [6]. This is displayed in Figure 2.

The model is the component which handles and stores the data. It also contains the functions used to handle the data. This data is presented to the user by the view component. The model and view component is linked together by the controller which handles the actions of the view and signals to the model what should be done with the data depending on the user input.

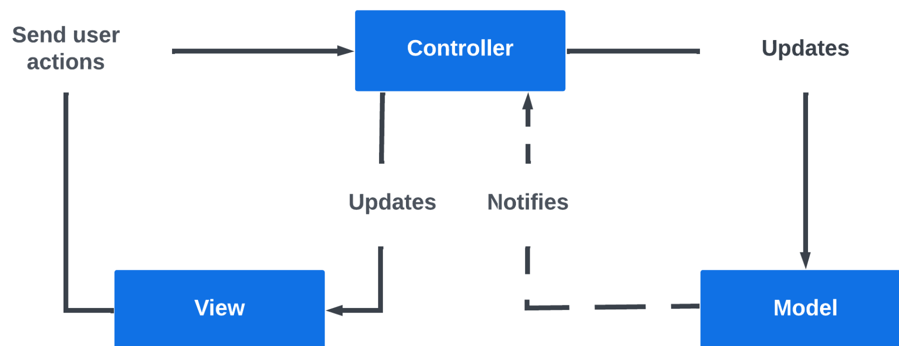


Figure 2: Relation between the different components in the MVC-pattern. [7]

2.3.2 Python libraries

Before starting to properly implement the GUI, different python libraries were looked into in order to facilitate the development. After discovering what is available it was narrowed down to the following alternatives: *Tkinter* and *Kivy*. These two options are both well used and documented while also allowing for the development of a complex and visually appealing GUI.

It was decided that a simple GUI should be made using each of these libraries. During this process, both libraries would be evaluated in order to find which library was best suited for the project and included the best resources and widgets.

2.3.3 Kivy

Kivy is an open source python framework for developing cross-platform graphical user interfaces [8]. Cross platform means that with a single code-language base a developer can develop applications for different platforms such as (but not limited to) windows, IOS or Linux. Kivy is widely used to develop touch-based GUIs which was applicable to the project and one of the factors that made it worth experimenting with. Another important factor is that the framework and the belonging objects are well documented on the developers web-page. When experimenting with Kivy, a project called *KivyMD* [9] became a subject of interest. KivyMD is a fork of the kivy repository. A fork is a repository that shares code with the original repository. KivyMD is has much of the Kivy-widgets available, but the widgets have been customized. The goal of the KivyMD project, as written on their github webpage, is to approximate google's material design guidelines. This allows for a more aesthetically pleasing GUI.

In the experiments with Kivy, the code that builds the GUI was divided into a python file and a kv-file (kivy language file). The python file creates the app, in which all the widgets are added to. It is also responsible for all the functions that runs when interacting with the interface. The kv-file defines the widgets that are displayed in the application. It also enables the developer to create widgets without declaring them in the python script, which is beneficial for structuring of the code. The application is built in the python file, but reads from the kv-file in order to display its contents in the window. This division of labour between the files enables a cleaner code in which the tasks are clearly outlined between the two files.

When experimenting with kivy an object with the name of screen manager was used to navigate between different layouts. Screen-objects with different layouts were added to a stack and the screen manager made it possible to switch between different screens. This proved successful, but the orientation and size of the widgets inside each screen were difficult to determine. In addition some of the widgets inside the KivyMD project did not work in the same way as the widgets provided by Kivy and therefore functionality problems arose. For that reason it was determined not to spend time on learning the kivy framework in depth, but rather continue to develop the GUI with the Tkinter and CustomTkinter frameworks.

2.3.4 Tkinter

Tkinter[10] is an open source python library for user interface implementation. The library consists of a number of modules that each has different functionalities. It offers a range of widgets that can build a window in the GUI such as labels, textfields, buttons and more. Tkinter also offers functions for sizing and orientation of the widgets which allows for rapid development of customized GUIs. The library offers tools for event driven programming, which makes it possible to develop an interface that is interactive. When implementing a user interface with tkinter, it was discovered that it was intuitive to use, but the widgets had a too outdated appearance for our project. Therefore it was decided to use CustomTkinter [11] which is a python UI-library that is based on Tkinter. This

library has the advantage that it offers fully customizable widgets that are more modern in appearance than the widgets that Tkinter offers. It comes with the same capabilities as Tkinter when managing widgets. Functions for sizing and orientation is available and easy to use for the developer.

One important tool is the onpress event which happens when a button is pressed. The developer is free to implement a self-developed function and then link it to the event which allows for more complex back-end programming. This was a must since it allowed for the GUI to interact with the sliding door.

2.3.5 UX design

There are several different principles and theories regarding how a user interface should be constructed to enhance the experience for the user. One of the most renowned is *10 usability heuristics for user interface design* by Jakob Nielsen [12]. Nielsen has been a pioneer in the field of human to computer interaction and usability of the internet. In his article, Nielsen describes the ten most important aspects of a user interface in order to make it understandable and easy for the user to interact with. During the GUI implementation these ten principles have been used as a guideline to make sure the prototype are inline with theory regarding usability and to make sure the program is easy to interact with. Below, the ten heuristics will be shortly described.

1. The application should always signal to the user which state it is currently in. This messaging should preferably be done at an instant when the state is changed but otherwise as soon as possible.
2. The design should use the same words, symbols etc as the user have encountered earlier in their life. This makes the interface more familiar for the user to navigate.
3. Mistakes should be easy to restore with undo buttons and it should be easy to go back to the previous page if desired.
4. The design should be consistent with the same naming and design to not confuse the user and or create uncertainty.
5. The interface should be created in a way that prevent errors from occurring. This includes both errors caused by the user as well as errors with the software itself.
6. Users should be able to navigate the interface without needing to recall the structure. Each menu and frame should contain the required information of what an action results in.
7. The design should accommodate both skilled and untrained users. Hidden shortcuts for trained users can enhance the experience and result in more effective use.
8. A minimalist design should be used by not displaying unnecessary information which might distract the user.
9. Error messages should be clearly explain followed by a recommended action to solve it.
10. Help and documentation should be provided to assist the user to understand the interface.

One of the challenges is how to implement the heuristics in an actual application. For example, one could argue that the GUI should consist of only one window that the user can interact with, in order to satisfy the need for easy navigation of the interface. However, this is not realistic since in order for all the information to fit in one frame, the widgets and texts would be too small for the user to see properly. Another challenge is to decide what information should be presented in the GUI. As engineers, we might think it is useful that system information, such as sensor-temperatures, response-time or other diagnostics are presented. But another user might only be interested in the different operating modes for the door and would find information about the diagnostics to be annoying or unnecessary if presented in the GUI. In the book *Design of everyday things* by Don Norman[13], Norman states that "Great designers produce pleasurable experiences". Don Norman is a co-founder of the Nielsen Norman Group, along with Jakob Nielsen. He has a background in electrical engineering and psychology and is respected for his knowledge in design, cognitive science and usability-engineering. In the book, Norman writes that a good experience is founded in discoverability. Discoverability means that the user discovers what the product does, how it works and what operations are possible. If the user can learn these things in an easy manner and without frustration, the experience will be positive. Norman writes that Discoverability is a result of appropriate appliance of five fundamental psychological concepts. *Affordance*, *signifiers*, *constraints*, *mappings* and *feedback*. But also a conceptual model.

1. An affordance refers to the relationship of the item and the user. This determines how the object can be used. A shelf for example affords placing objects in it and can therefore affords storing books. The GUI will for example, afford signaling to the door to open.

2. Affordances determine what actions are possible to do, but signifiers communicates to the user where those actions should take place. In the application of this project the affordance to select a specific operational mode is signified by a button with a text that describes that mode.

3. Mapping is the relationship between two different elements. In this case, the relationship between the controls and the result. An example of poor mapping would be to place a right arrow on the left side of the screen. Or another example, by turning the steering wheel to the left (anti-clockwise) and the car that you are driving would turn to the right. This is important for UI-design in which the user clearly sees the result of pressing a button in the interface.

4. Constraints limits the actions a user can take in each state. Long forms, and many different actions can be frustrating and make it difficult to navigate. However, if the actions and information are constrained to different states, and navigation between these states are logical and easy to understand, the experience becomes better. The developer divides the information and actions in to smaller groups which means that the user do not have to process a large amount of impressions at the same time.

5. Feedback is important for the user when an action has been made. This applies to all sorts of actions the user can do. For example, if the user presses a button to switch the operating mode on the door, the GUI should indicate when this has been completed. Otherwise the user will not be sure that the mode indeed has been switched until a person tries to walk through the door.

6. A conceptual model is an explanation that helps the user understand what can be done. An example is a manual that explains what can be done with the product. In the GUI, the design is extremely important to explain what can be done. Symbols, texts and labels that helps the user understand what will happen when, for example, a button is pressed.

These concepts are used in this project to communicate how to use the prototype and what actions can be taken by the user. By applying these concepts, a technology with the user in focus can be developed, which will result in a user friendly product. Norman emphasizes that, engineers often risk designing products that are easy for the developers to use, but not for the user. According to Norman, this happens when the engineers do not take into account the fact that the knowledge about the product differs from user and developer, due to the time and education gained by developing the product.

2.4 Signal processing

There are several different communication protocols available to choose between for wired communication between devices. The company, Assa Abloy, have developed their own protocol which is used for communication between devices constructed by the company. This protocol requires the RS485 interface.

2.4.1 RS485

As previously mentioned RS485 is a communications interface for communication between two electrical devices. The interface uses two cables to transfer data. One of which is the original signal while the other is the same signal but inverted. By doing this it reduces the effect from electrical noise since both signals are effected equally and the difference in voltage between them remains intact.

2.4.2 MQTT

Message Queuing telemetry transport (MQTT) is a message protocol created by OASIS primarily intended for internet of things (IoT) devices [14]. It is a lightweight protocol which allows it to run on small and resource limited devices and with limited network connection. The protocol uses a publish-subscribe architecture where one device publishes a message and the subscribers to that topic will receive the message from the server. The protocol requires a broker to mediate the communication. The broker can be run remote on a cloud or locally on a computer. All messages are sent to the broker which forwards the message to the clients subscribing to the topic. The message sent contains a topic, a payload as well as a quality of service (QoS). The topic has the primary purpose to distinguish the different messages and divide which clients should receive the message. The payload contains the information related to the topic sent. Together with the message a Qos can also be stated. This describes the importance of the message being delivered. If the QoS is set to 0 the message is sent once and no means of ensuring a proper delivery is taken. QoS 1 makes sure that the message is received at least once to the appropriate clients where as QoS 2 ensures that the clients receive the message exactly once. Overall a higher quality of service results in more security in regards to that the messages are received correctly while the cost being more time and data needed for the communication.

2.5 Programming workflow

Since the list of specifications were not fully specified from the beginning of the project, but rather updated and written during the project, the workflow used had to account for this. Agile programming is a methodology that shifts the focus from process and tools to interaction and feedback [15]. This way of working adapts a more holistic perspective and appreciates sharing information fluidly over the course of a project. The benefit of this approach is that extensive documentation is not needed for each group to produce results. In this project, this is desirable for several reasons. In order to start the project, a full documentation of how the GUI should be designed is not required. This should rather be provided by the other master thesis in the later stages of the project. Another reason that suggests that this methodology is beneficial to our project is that the development becomes closer to the stakeholders, mainly the user and Assa Abloy. When programming in this project, the approach is inspired by agile programming, mainly the extreme programming (XP) approach. This includes the stages of discussion with stakeholders, implementing code, testing and reviewing the code, and feedback from the stakeholders. By iterating over these stages the development of the project is influenced by the stakeholders and produces results that has a greater potential to be accepted by the stakeholders.

3 Methodology

At the start of the project a plan over the assigned weeks were put together. A Gantt chart had previously been made in the target document but since the problem description had changed in a couple of aspects during the initial meetings this chart and the plan had to be revised. The initial Gantt chart together with the actual Gantt chart can be found in Appendix A.

As can be seen in the final Gantt chart the first couple of weeks were spent doing research and deciding on the components to be used for the project. During this time research was also done regarding implementation of a graphical user interface. This included everything from the structure of the code to UX design, as seen under the theory section. Following this the components were ordered, assembled and configured. Once the RPi was setup correctly with the display, the foundations of the GUI was implemented. This included the structure and the outline for the design. When the framework for the GUI had been implemented, this part of the project was put on hold to await the results from the user study simultaneously being conducted. The next phase of implementing functionality for the communication between the man-machine interface (MMI) and the sliding door was started. The communication part of the application needed to handle user actions done in the GUI that regarded the door and forward them to the control unit of the sliding door. In addition, it was also of necessity to handle relevant signals from the door, such as state and errors that the prototype should display to the user. Towards the end of the project, once all needed functionality for the prototype was in place, the majority of the time was spent changing the design and adapting it to the findings of the UX-analysis. At the later stages a case made of plastic to host the RPi and the touch display was also designed.

3.1 Selection of components

As described in the *Components* section under *Theory*, the decision of the control unit on the MMI stood between the Raspberry Pi 4 model B and an Arduino Uno. It was decided to use the Raspberry Pi 4 model B in this project.

This decision is mostly based on the performance of the raspberry pi in comparison to the Arduino. The Raspberry Pi supports a lower response time since the microchip of the Raspberry Pi 4 runs at 1.5 GHz and the Arduino Uno runs at 16 Mhz.

Another important aspect in the decision is the capability to use it for serial communication. Arduino is limited to simple serial communication where as the Raspberry Pi covers a larger variety and has more available ports. Additionally the Arduino comes with it's own programming language. The raspberry pi can be programmed with python, which is beneficial since the group have more experience and knowledge with python.

These finding were backed from an employee at the company who had prior experience of implementing a GUI on both the Arduino and the Raspberry Pi [16]. The experience was that the Arduino struggled in performance with larger GUI:s and therefore the employee suggested that the RPi should be used. The issues with the arduino mainly regarded response time, and the experience the employee had was that the raspberry pi allowed for faster communication with the GUI as well as the sliding door.

Once the decision was taken to use the Raspberry Pi model 4, research of available touch screen displays which were compatible for the RPi was conducted. The most important feature was to find a screen with similar size to what the company would like to have in their future product. After finding a few devices which would work, it was finally decided to opt for the Hyperpixel 4.0 square Touch [17]. This display has a touch screen with capacitive touch with the dimensions of 72 x 72 mm. The measurements of the outer rim is 84 x 84 mm. The measurements of this display resulted in the device being slightly larger than the stated maximum size when placing it in the case which was designed in the later stages of the project. After discussing the decision to use this touch-screen with the supervisor at Assa, it was decided that it would work for the project. The screen has an Frames per second (FPS) rate of 60 and uses the Display parallel interface (DPI) which is more than sufficient for our purpose. An advantage [17] of this display is that it required very little setup to work for the Raspberry Pi. The company who has manufactured it, Pimorino, has created a one-line installer script which can be run in the linux terminal of the RPi.

Once all the major components were selected, research was done in order to determine which smaller components would need be ordered to be able to work with the device properly. The RPi needs an SD card in order to store the OS as well as files within it. It was decided that a 16 GB SD card would be enough for the project. In order to work with the RPi a power cable, USB-C to outlet, and a cable to connect to an external display, HDMI-micro to HDMI would also be required. The HDMI cable was not vital, because the Hyperpixel screen was able to display the OS. However, being able to use an external screen would make the development process less problematic, and it was decided that it was beneficial to be able to work on a larger screen. An Ethernet cable was also ordered, in order to get a stable internet connection while working on the device. After deciding upon all the components needed, an initial order of components was done. Research was done to try and find a retailer used by that company that had all of theses components in stock. Digikey had all of the components which made the ordering more convenient. The below Table 1 shows all of the ordered components together with their respective prize followed by the total at the bottom.

Table 1: List of ordered components, prices in SEK

Component	Cost
Raspberry Pi 4 model B	462,6
Pimorino Hyperpixel 4.0	786,4
SD card	89,74
Power supply for Raspberry Pi	82,24
Ethernet cable	26,5
HDMI to HDMI micro cable	51,4
Total Cost	**1516,97**

3.2 Working setup

Once the components arrived they were assembled and configured. Initially the Raspberry Pi was set up. To start the setup, an operating system had to be decided upon and installed on the SD card for the device. It was decided to use Raspbian GNU/Linux 12, since it is the most safe and reliable option. In order to install the OS the file for the system, it must initially be downloaded to the SD card using Raspberry Pi Imager on a

different computer. The SD card containing the OS was then inserted to the SD card slot on the RPi and the device was booted up using an external display connected with the HDMI cable.

After ensuring that the Raspberry Pi was working correctly it was configured for the display. As mentioned previously this was a simple process due to the one line script created by the company, Pimorino [17]. The display was then attached to the board by connecting it to the 20 GPIO pins. It is mounted securely using the four included legs that are screwed on. This can be seen in Figure 3.

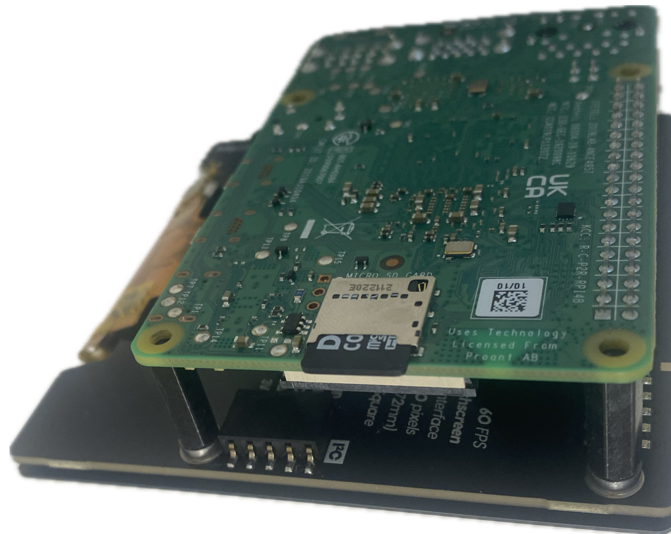


Figure 3: Mounting of display on the Raspberry Pi board.

The needed python and C packages were then installed on the Raspberry Pi. For python this included *customtkinter* and *MQTT paho* which both were installed using PIP. For the communication program the MQTT paho library for C was also needed. The needed files for the MQTT client were downloaded from Eclipse Github page [18]. The package is then build using CMake following instruction listed on their page.

In order to have a smooth work flow considerations were taken at an early stage to ensure a good working setup on the computer provided by Assa Abloy. Since it was decided to use a Raspberry Pi 4 as the micro control unit (MCU) which runs Linux as its OS, Windows Subsystem for Linux (WSL) was installed on the computers, to allow for the work to be done on the same environment as the Raspberry Pi. This was done since the code eventually will run in Linux, and to eliminate problems of something working differently when changing between the operating systems. To ease the collaboration and allowing simultaneous programming, Gitlab was used, which the company has its own license to. This also allows for quick access of the latest code version, on the raspberry Pi to quickly test the program on the prototype.

When running the customtkinter program on the virtual machine it was realised that the visuals were not working as they should and that the window with the GUI was warped. After researching the reason for this it was discovered that WSL not yet had

full support for graphical programs. When researched it appeared that this is a problem several other users have had. On Microsofts github forum for reported issues it was found that it was possible to fix by adding a few lines of code within the *.wslconfig* file [19].

3.3 Specifications for the GUI

The process of developing specifications for the GUI spanned over the whole project. Most of the inputs came from other entities within Assa Abloy and were both hard, must haves, and soft, could be implemented, functionalities. The initial inputs is compiled in the list "Functionalities for the first concept". Our initial guideline was to implement the results of another master thesis which had a task of collecting data from the customers and develop concept for a GUI that met the demands from the customers. However, this process would not yield any results until the late stages of the master thesis. Therefore the initial directions received from Assa Abloy was to develop a concept of how a GUI would look like, and then modify it when the other master thesis had reached a conclusion.

To get an idea about the design, a meeting with an employee at the company that works in Switzerland as a commercial product manager was scheduled . In this meeting, different versions of touched based control units from competitors were shown and the design of the interfaces were discussed. Strengths and weaknesses from the respective designs were discussed and formulated. The primary goal of this meeting was for the group to be introduced to commercially marketed versions of a touch based MMI. The takeaway was how different visual styles and designs resulted in a different product. By combining this experience with the theory of design from Norman and Nielsen, which is presented in the theory section, an idea was formed of the visual style of the GUI.

After the meeting with the commercial product manager, the need for a specification of the GUI arose. In order to implement the visual style, functionalities of the GUI needed to be formulated. To achieve this, a meeting with the product owner of sliding doors at Assa Abloy's location in Landskrona were set up. In this meeting, some required functionalities of the MMI were discussed. As an example, as mentioned earlier, functionalities such as locking the door must be password protected due to regulatory frameworks for fire safety. Functionalities for selection of door-modes and displaying of errors was also a requirement for the prototype. In addition, it was decided within the group that it would be desired to implement an interface for adjusting the settings of the door, such as modifying the speed of the doors and changing the password. The access to the settings would require password authentication of the user. The graphical design of an interface with the listed functionalities, see the list below, were initially done by applying the insights gathered from the meeting with the employee in Switzerland and a first concept of the GUI was formed.
Functionalities for the first concept.

- Compulsory: The GUI should allow the user to change the operating mode of the door. The modes that can be selected are "Open, Auto partial, Automatic, Exit only and Closed".
- Compulsory: The operating mode "Exit only" has to be password protected"
- Compulsory: A menu of displaying errors in the system must be included in the prototype.

- Optional: An interface for adjusting settings (such as door opening and closing speed, change of password, and the amount of time the door should stay open), should be included in the prototype.
- Optional: The access of the settings should be password protected.

The workflow methodology of developing the GUI was iterative, and anchored in the first concept. After the first concept had been developed, a meeting with the product manager of sliding doors in Landskrona was scheduled. In this meeting, the concept was discussed and was subject of constructive criticism. This was the workflow for every iteration of the interface and is a common way for software development in industry [20].

This way of working proved to be very effective for the project, since discussing an implemented concept, that can be experimented with reveals potential strengths and weaknesses very clearly. In addition to the product manager in Landskrona, the other master thesis group, which where conducting a UX-field analysis [21] for the touch based interface were of much help in this work-process. Regular discussions and workshops were conducted, in which the design of the GUI were discussed and tested. In these meetings, the focus were shifted to the perspective of the customer. An essential conclusion was to not to sacrifice simplicity for a complex product. The interface should be easy to use for all people, which means that users with a low technical understanding should be able to use the product.

The result of this perspective was to try and incorporate all functionalities in a way that makes the GUI easy to use. From our perspective, this meant that ambitions for advanced interactivity and functions had to be lowered, and instead focus on designing a prototype where simplicity is a guideline for the design. However, this also proved to be a challenge, as the number of frames should be kept to a minimum to allow for easy navigation, but the functionalities of the prototype would still have to be implemented. These meetings resulted in more functionalities for the prototype which is listed in the end of this section, and would have to be implemented.

It was decided that a screen saver would be beneficial to the product, since hiding the functionality of selecting door operating modes would yield a smaller risk of people (such as customers in the store) interacting with the controller. As an addition, the current operating mode of the door should be displayed on the screen saver to allow for quick forwarding of information to the user. In order for generating a feedback to the user when a new operating mode is selected, the pressed button should also change colour for the user to know that the action has been processed and completed. When an error is detected, the user should be notified by an alert in the GUI. This was implemented by making an error button light up if there exists any errors. By pressing the error button, the user could access an error log. In this log, error description and potential solutions would be available. The work process finally resulted in a list of specifications for the GUI which is presented below.

- The GUI should allow the user to change the operating mode of the door.
- The current operating mode should be presented to the user.
- The GUI should allow the user to be able to see active errors in the door.
- The errors should have descriptions and suggested solutions.
- The MMI should notify the user if the door has active errors.
- The product should enter screen saver mode if the user is inactive
- The screen saver should contain information about the current operating mode of the door.
- The GUI should allow the user to change settings such as sliding speed for the door, and changing password.
- The settings and shutting off the door should be password protected for legal reasons.

By formulating the specifications of the GUI, the code could be implemented accordingly.

3.4 Design principles of the GUI

As presented in the previous section, specifications of the GUI had been formulated. By following the design principles of Norman and Nielsen, the design of the application aimed to make it as user friendly as possible. In the previous section, it was mentioned that the other master thesis group [21], was of aid to the project by shifting the perspective to the customer. In addition to keep the number of frames to a minimum, the symbols and texts in the GUI are an essential part of the user experience. By following the second heuristic of Nielsen, images were researched that are commonly used for settings, errors etc.

The navigation is done by means of buttons that are coherent throughout the GUI. The settings screen, error log and password authentication frame has a button that when pressed brings the user back to the mode-selection screen. This decision was made in order to follow the third heuristic of Nielsen. Additionally, some of the menus has a back button to allow the user to navigate back to the previous frame.

During most of the development of the GUI, the home button would take the user to a screen that had three buttons. One for accessing the error log, one for accessing the settings and one for accessing the mode-selection screen. This screen can be seen in Figure 4. However, during the late stages of GUI development, the decision was made to erase this screen from the application. This decision was based on a meeting with the other master thesis group [21]. In this meeting, our prototype was presented and experimented with, and the realisation was made that a more smooth navigation of the application could be achieved as a result of deleting the frame. Instead, the error log would be accessed by pressing the warning triangle, which appears if an error is present.

The functionality of the error log is to give the user or service technician more information about active errors, and it will not have to be shown otherwise. The settings can be accessed by pressing a cog wheel button that is implemented in the mode-selection screen. The benefits of this change is that the navigation to error log and settings is shortened, and becomes more intuitional for the user. This change is depicted in Figure 5 and 6. Figure 5 shows the screen for mode selection before the change and Figure 6 shows the mode selection after the change.

Another change that was made in this process was that the grouping and the sizing of the buttons. This was made according to one of the design principles and based on the research made by the other master thesis. That research showed that the operating modes *Auto* and *Open*, were most commonly used in the MMI and therefore these buttons are larger than the others.

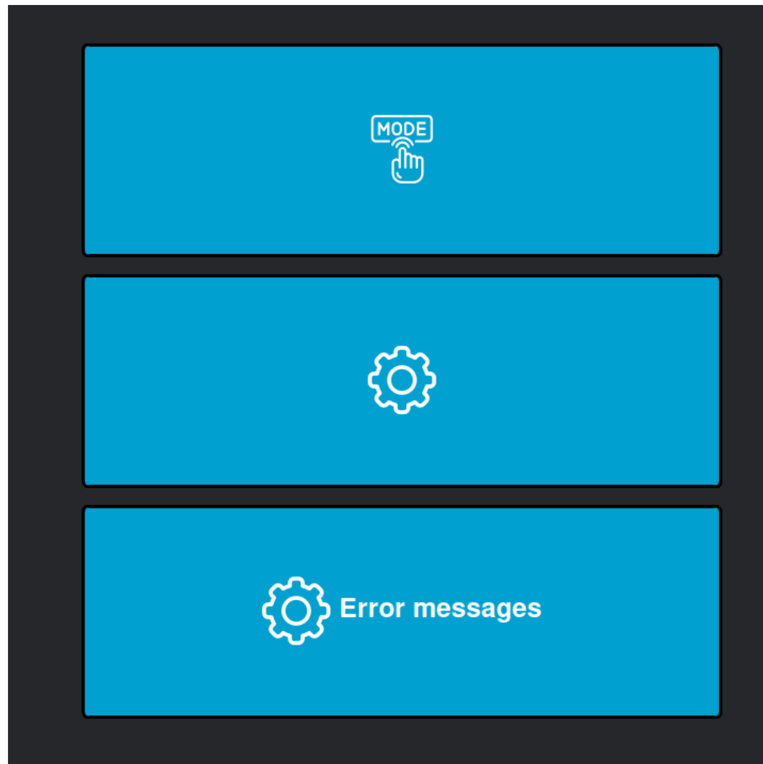


Figure 4: Display that shows an old screen, which was removed in later stages in the project.

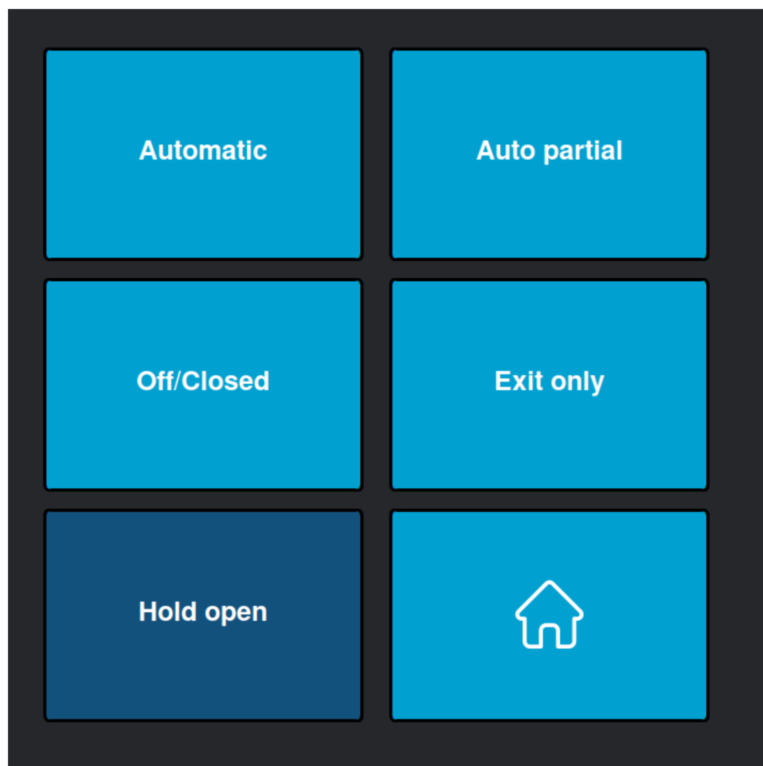


Figure 5: Display that shows the old mode selection screen.

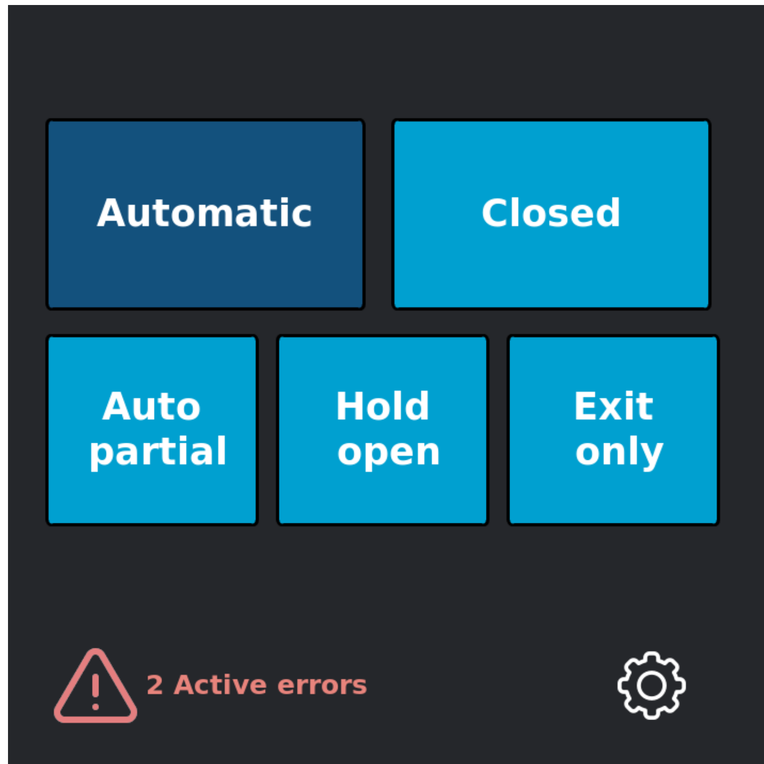


Figure 6: Display that shows the new mode selection screen.

3.5 Code structure

To begin with the code was contained within one class. Once more features and frames were added it quickly became apparent that this was not sustainable option since the code became difficult to navigate and understand. As previously mentioned under the theory section of this report the MVC-pattern [5] was chosen to structure the code. To recap, this is an architectural pattern used to separate the code into different assignments. Table 2 shows the packages and a short description for each package in the MVC-pattern.

Table 2: Packages for building the GUI

Packages	Description
Models	Contains modules that are models for the application. The models stores data and does the alternations of the data.
Views	Contains modules that are views for the application. The views contain all the graphics displayed to the user.
Controllers	Contains modules that are controllers for the application. The controllers connects the view and the model with functions connected to the actions by the user.

Model: The model is responsible for handling and storing the data in the program. Once data needs to be altered, a method which is in the model is called and does the desired alternation of the data.
View: The view element presents the data to the user in a graphical user interface. Each frame of the GUI contains widgets such as labels or buttons and the widgets are implemented and positioned within the view.
Controller: The controller connects the view and the model with functions connected to the actions by the user. It then calls the appropriate model function.
In the implementation of the GUI, it was decided to split the code up even more. The reason behind this decision is mainly to make it easier to navigate the code and more clearly divides the different menus. Figure 6 shows the structure after the decision was taken to restructure the code. Previously all the controllers, models and views were only contained within one module for each package. When researching information regarding the MVC architectural pattern an article showcasing how to use the MVC-structure in python was found [22]. Figure 7 shows a class diagram over the entire package implemented to build the GUI.

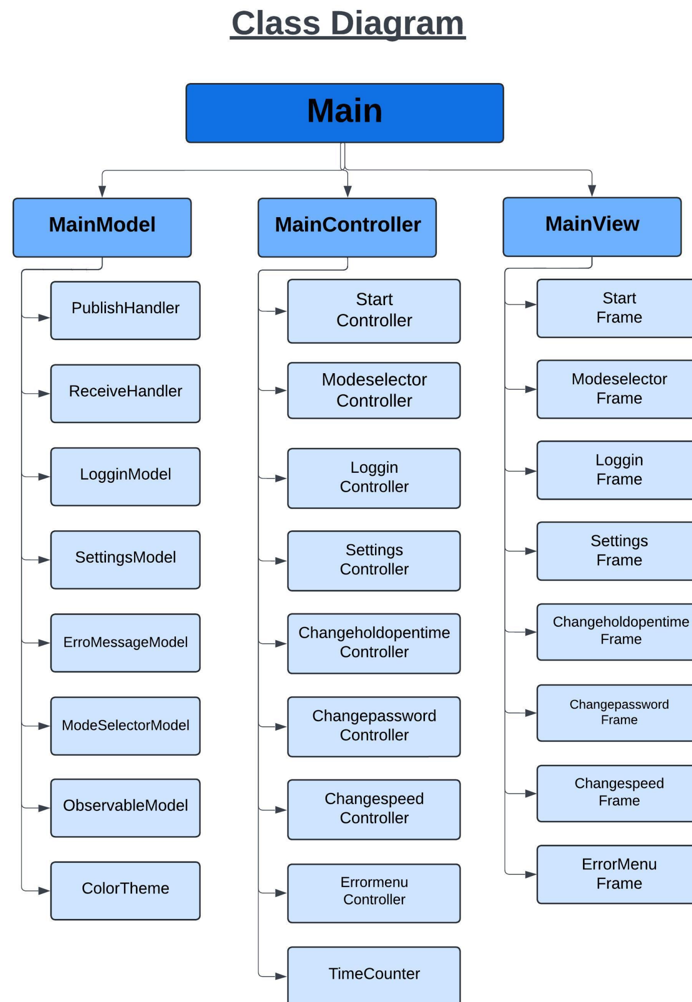


Figure 7: Class diagram of the code for the GUI

3.5.1 Structure of the model element

The model element, Table 2, is the part of the program which handles the required data within the interface. It is the model that stores the data and determines which action should be taken when data is changed. The following Tables 3 to 7 describes the purpose of each model component and gives details of the reason behind each method within the model. Some of the methods were left out from the tables. This is because some methods are generic and are described in other tables.

Table 3: Description of the most important methods contained in the main model

Method	Description
init(self):	The main model only contains an init method which purpose is to create instances of the different model modules. It also ensures that the servercom classes, *publishHandler* & *receiveHandler*, are started to run their *run* method.

Table 4 shows the most important methods implemented in loginModel. The loginmodel inherits from observableModel. LoginModel handles the data related to password authentication. The password is written in a text file which is stored in the same directory as the models. The loginModel compares the current password-attempt to the correct password in the text file, and can also write a new password in the text file when the user wants to change the password.

Table 4: Description of the methods within LoginModel

Method	Description
init(self):	The init method initiates a couple of parameters needed for password authorisation to be possible. The first is a boolean logged_in which is set to false at the start. This is used to keep track of whether the user has been authorized. The current password of the OMS is also retrieved from the password.txt file and stored as an attribute.
addNumber (self,number):	This method is used to append password digits into a list called current_password, when a digit is pressed in the log in menu. The method also checks the length of the current entered password after the digit has been entered. If the length has reached the length of a complete password (4 digits), an attempt to log in is made by calling the log in method.
addNumberChange (self,number):	Does the same as the previous mentioned method but used for the change password menu.
undo(self):	Called when the undo button in the log in menu is pressed to undo the latest entered digit in the password. This is done by using the pop method on the current password list. The method makes sure to check that the current list is not empty since using pop on an empty list results in an error.
undo2(self):	Does the same as the previous mentioned method but used for the change password menu.

Continued on next page

Table 4 – continued from previous page

Method	Description
login(self):	Compares the entered password *current_password* with the set *password*. If the password is correct an event is triggered to redirect the user to the frame they want to enter. If the password was incorrect another event is triggered which changes to information label to let the user know of this. Finally the *current_password* list is cleared to allow the user to reenter the password.
changePassword(self):	This method is called when the user has entered 4 digits in the change password menu. If this was the first iteration, meaning *currentiteration* is 0, the new password entered is stored as *firstPassword*. An event is also triggered which changes the text label on the frame to tell the user to reenter the password. Finally the iteration attribute is set to 1. The second time the user enters 4 digits *currentiteration* is 1 and another action is performed. Now the two entries are compared. If the two passwords match the password is changed into the new password. This is done by changing the password.txt file. If the passwords don't match, this is signaled to the user using an event. Regardless of if the password change was successful or not, the *currentiteration* is changed back to 0 and *firstpassword* is cleared.
log_out(self):	Sets the logged_in boolean back to false.

The settingsModel is described in Table 5. This module inherits from observableModel. This model contains data about selected settings and the current mode. It reads from a the text file containing the previous settings which the user has specified. This is done in order to preserve the current settings, even if the system is restarted.

Table 5: Description of the methods within settingsModel

Method	Description
init(self):	The init method reads from the settings.txt files and stores the settings as attributes. If connected to a door these are then overwritten with the actual parameters of the door.
change_setting (self, setting, value):	Called when a setting should be changed. The setting parameter decides which setting should be changed and the value parameter is the value the setting will be set to. This setting is also written into the corresponding .txt file to store the parameter when the program is terminated.
get_setting(self, setting):	Method used to access the different settings.
change_current_mode (self,newmode):	Method called when the current mode of the door should be changed. The mode setting is changed to *newmode*. The setting is also changed in the corresponding text file.

The errorMessageModel is described in Table 6. It contains the class ErrorMessage, which describes a frame with 4 attributes: errorNumber, errorName, errorDescription and errorSolution. The errorMessage does not contain any methods, but rather defines what should be included in an error. The errorMessageModel also contains a class that inherits from observableModel and initiates a list with all potential errors. Each element in the list is an ErrorMessage.

Table 6: Description of the methods within errorMessageModel

Method	Description
ErrorMessage.innit (self, errorNr, errorName, errorDescription, errorSolution):	This innit method is used to create an object of an errorMessage. The input parameters are all stored as attributes. The ErrorMessage object also has a boolean attribute called active which initially is False. This attribute is True when the door signals that the specified error exists in the system.
innit(self):	Creates a list and appends objects of all the available errors of the door that should be displayed for the user once they occur.
newErrorList (self,errorList):	This method is called when the latest list of active errors received from the door has changed. The method starts by setting all errors activeStatus back to False. It then iterates over the input list *errorList* and sets the activeStatus to the errors in errorMessageList with matching errorNr to True. At last it triggers the *errorsChanged* event which causes the errorMessageController to update the errors to be displayed.
getActiveErrors(self):	This method returns the amount of active errors by iterating over the errorMessageList and counting the amount of errors which have an active status.

Table 7 describes the methods in the observable model. The observable model contains a dictionary called event_listeners. In the dictionary, the key is a string which maps to a function. It also has a method for triggering events. The event_listeners builds the communication from models to controllers. When a model wants to influence the other packages, it triggers an event which then makes the controller run the corresponding function in the dictionary.

Table 7: Description of the methods within observableModel

Method	Description
add_event_listener	This method is used to create an event - function pair and stores them in a dictionary.
trigger_event(self,event):	Called from the models, who all inherits the obervableModel class, in order to access a function within one of the controller classes. Often done as a response to a change in the data within the model.

3.5.2 Structure of the view element

The view element consists of a main view, a main window as well as several frames. The main view's primary purpose is to combine the other files into a single View object which can be reached from the model and controller.

In the initiation of the class, the graphical user interface window is first created. This is an empty window with 720x720 pixels to fit the touch screen. It is also set to run in full screen mode to ensure that the application fills the entire screen. All of the used frames are added to this window using the addFrame method. This method takes a frame-class and a string, the name of the frame, as parameter. The method then creates an object of that frame class, assigns it a name and adds it to the window. This is done in order to make sure that all other modules shares the same instance of the frame. When something is changed within the frame, other modules that uses the same frame will be aware of the change, since the same instance is shared among them.

This class also contains a method called switchFrame. This method takes a string as a parameter and uses the customTkinter function called *raise* which moves the chosen frame to the top of the stack and therefore is the frame being shown. This method can be used by the controllers to change the frame which is displayed to the user.

As touched on briefly, the view element consists of several frames. These frames can be seen as individual menus that can be shown in the application. Each frame takes up the whole application and contains the required widgets for that menu. Table 8 describes the views that were created.

Table 8: Description of the view modules

View	Description
mainView	The mainView contains two classes. One is a frame that inherits from TypeDict. All other modules in the view-package is of the type frame. The mainView imports all the frames and assigns a name to them in the frame-class. The other class in mainView initiates a dictionary, which holds all the frames. The most important function of mainView is to initialize all of the frames in the application.
mainWindow	The mainWindow inherits from customtkinter.CTk and initializes the app.
startFrame	The startFrame initializes and places the widgets for the screensaver. Since the screensaver consists only of a button, one button is initialized that fills the whole screen. An error button is also created, but not placed, since it should not appear unless there is an error.
modeSelectorFrame	The modeselectorframe initializes and places the widgets that is building the interface for door-operating mode selection. This is done in the init-method of this class and creates 5 buttons (one for each operating mode). An error button is also created (but not placed), and lastly one button for accessing settings.

Continued on next page

Table 8 – continued from previous page

View	Description
errorMenuFrame	The errorMenuFrame initializes and places the widgets that are building the interface for the error-log. It inherits from customktinter.CTkFrame. In errorMenuFrame, three classes are defined, errorMenuFrame, ScrollableFrame and ErrorMessageFrame. The errorMenuFrame holds instances of the other classes. The scrollableFrame creates a scrollbar, which provides the function of scrolling the errormessages displayed in errorMenuFrame. The errorMessageFrame creates a frame that holds information about the active error. In addition to the error information, the frame holds a button that expands the message, to reveal more information about the error.
loginFrame	The loginframe initializes and places widgets that builds the interface for password authentication. This includes the number buttons that are used to formulate a password, the containers where a *-symbol appears to give feedback when a number is pressed, an erase button to erase a number in the current password attempt, a back-button to exit the loginframe and lastly a label that indicates if the password was correct.
settingsFrame	The settingsFrame initializes and places widgets that builds the interface for changing settings in the application. Five buttons are created and placed, which when pressed navigates the user to the setting that should be changed. As mentioned before, the settings that are available is changing the speed of the door, changing the password, and changing the time the time the door is open, in auto mode. three buttons are dedicated to this. One button navigates the user to the mode selection frame and another is linked to the error log, and only appears when the door has an active error.
changeHoldOpen TimeFrame	The changeHoldOpenTimeFrame initializes four widgets. A slide bar that is used to set the time the door stays open is initialized and placed in the frame. A labels is created, that holds information about the setting, which describes the function of the slide bar. The menu has a back and a home button to navigate to other menus. A button for confirmation is also created to confirm the selected time. Lastly, an error button is created, which appears when an active error is present.
changeSpeedFrame	The changeSpeedFrame builds the interface for change of door speed. This is done by the creation of two slide bars. One is controlling the opening speed of the door, and one is controlling the closing speed of the door. The slide bars is operating in a range of 200-600mm/s, with 5 steps in the range. Each step is corresponding to a change of +-100 mm/s.

Continued on next page

Table 8 – continued from previous page

View	Description
changePassword Frame	The changePasswordFrame builds the interface for the functionality of changing the password. It is identical to the loginFrame, with the goal of keeping the application coherent throughout the navigation.
colorTheme	Inside this package, a python file named colorTheme.py was implemented. This module initialized attributes and assigned colors to each attribute. This is used in all other frames and serves the purpose of keeping the color theme of the application coherent, and allows for easily changing the colors inside the frames.

3.5.3 Structure of the controller element

For each of the view class instances which creates a frame there is a corresponding controller class which configures all the actions related to the widgets of that frame. Primarily these controllers contain the action functions called when a widget is interacted with. If the action is related to any form of data it then connects the correct model to handle this.

The individual controllers for each frame are combined into a single controller object in a main controller class. This class initiates all the individual controllers as well as adds the different events which the event listener should listen to. The event listeners are required in order for the model to call functions from the controller. Tables 9 - 17 describes the controllers that were created. The changeSpeedController is excluded from this Table since it is very similar to the changeHoldOpentimeController.

Table 9 describes the methods within the mainController. The mainController is responsible for creating instances of all the other controller-classes. This controller is also responsible for handling an alarm that overflows and triggers the application to go to screensaver-mode. It also creates events that are added to the eventListener dictionary created in observableModel.

Table 9: Description of the most important methods contained in the main controller

Method	Description
alarmHandler(self):	alarmHandler function implements what happens when the alarm overflows. When our alarm overflows, the alarmHandler is called and takes the user to the screensaver by calling loginModel.logout().
clicked():	This function resets the alarm to fire in 10 seconds from the time clicked() was called. This function is used when the user touches the application and signals that the user is active.
init(self,model,view):	This function initiates all other controller modules. It also adds eventlisteners to the dictionary in observable model. It also initiates the alarm signal and binds it to the alarmHandler function.

Continued on next page

Table 9 – continued from previous page

Method	Description
settingChanged(self):	This function is used when the user changes the setting controlling the speed of the sliding door, and updates the view accordingly.
start(self):	This starts the application in the screensaver view.

Table 10 describes the methods within the modeSelectorController. The modeSelector-Controller is responsible for handling events related to the modeSelectorFrame. It is responsible for publishing messages to the MQTT-broker which is then registered and processed by sending a low-level signal to the door. Another method that is implemented in this class is to highlight the selected door operating mode for giving feedback to the user which mode is selected. It also implements a method to force the user to authenticate via password if the selected operating mode of the door is off.

Table 10: Description of the most important methods contained in the modeSelector controller

Method	Description
init(self,model,view, timer):	The init-function of the modeselector-module assigns the attributes of this module to imports of mainModel and mainView. Since these main-classes have initiated objects of all other classes, the objects are accessible in this implementation. It also highlights the current operating mode.
bindButtons(self):	This method binds the button in the modeselector interface to functions within the module. This is necessary in order to implement functionalities for the buttons.
buttonAction (self,button):	In the modeSelectorController -module, there are multiple methods that ends with *buttonAction*. These are methods that are bound to a button in the interface, and looks almost indentical, hence the reason to group them in the same category. In the methods, a publish-statement is written, which publishes to a topic to the MQTT-broker. This is then forwarded to the communication protocol implemented in C, which changes the operating mode. Addidionally, the button passed as a parameter is highlighted.
checkAccess (self, button):	This method is binded to the button that sets the operating mode to "closed". It forces the user to authentication with password when this mode is to be set, by setting the application to the login-screen.
highlight(self,button):	This method highlights the currently selected operating mode in the interface. It also alerts the application that the operating mode has changed by invoking *alertModeHasChanged* which is a method implemented in the same module. This allows for tracking the current operating mode.

Continued on next page

Table 10 – continued from previous page

Method	Description
alertModeHas Changed(self):	This method calls a method in modeSelectorModel which reads a text file that contains the current operating mode. The modeSelectorModel is keeping track of the current operating mode, and this method is responsible for reporting when to the model, when a new operating mode is selected. In addition, this method configures the text displaying the operating mode in the screensaver.
enterErrorMenu(self):	This method is bound to the error-button. When pressed, this method is called and changes the view from mode selector to the error log. It also increases the alarm from 10 to 100 seconds.

Table 11 describes the methods implemented in the startController. The startController is responsible for handling events related to the screensaver(startFrame). The motivation for naming this controller *startController* is because the application is starting in screensaver mode. The screensaver is implemented as one button that fills the whole screen, and therefore the startController has a method for switching frames to mode selection when the user touches the screen. It also implements a method for the action that happens if the error-button appears and the user presses it.

Table 11: Description of the most important methods contained in the start controller

Method	Description
init(self,model,view, timer):	The init-function of the modeselector-module assigns the attributes of this module to imports of mainModel and mainView.
bindButtons(self):	Binds the button that builds the screensaver screen to a method (enterModeSelectorMenu).
enterModeSelector Menu(self):	Takes the application to the modeSelectorMenu by calling the *switchFrame*-method in mainView.

Table 12 describes the most important methods that are implemented in the loginController. The loginController is responsible for handling events related to password-authentication. It determines actions that happens in the loginFrame such as collecting data from the buttons pressed and sending it to logincontroller for verification.

Table 12: Description of the most important methods contained in the login controller

Method	Description
init(self,model,view, timer):	The init-function of the modeselector-module assigns the attributes of this module to imports of mainModel and mainView. It also assigns a *frame*-attribute to the login-frame. In addition it calls *bindButtons* in order to assign each button in the loginframe to a method.

Continued on next page

Table 12 – continued from previous page

Method	Description
bindButtons(self):	Binds all buttons in the login-interface to a method. The number-buttons (that builds the password) is assigned to the same function (numberButtonAction) that is described later in this table.
numberButton Action(self,button):	This method is bound to the number-buttons that builds the password attempt by the user. Each time a number button is pressed, this method adds a *-symbol to the containers that gives feedback to the user that a click has been registered. It also signals to the logincontroller that a number is being added to the current password attempt by calling loginmodel.addNumber(). Additionally, it changes the colour of the last pressed number button by calling *changeColor* in this module. This is done in order to give feedback to the user that the action has been registered.
undo(self):	This method is bound to the button that erases the current password attempt. It does so by erasing the *-symbols in the containers, and signaling to the loginmodel that the current password attempt is not valid.
changeColor(self, button):	Changes the color of the button passed as a parameter.
back(self):	This method is bound to the back-button in the login-interface. This button is pressed when the user wants to exit the password-authentication. It signals to the loginModel that the password has not been provided (therefore, not allowing the user to access password-protected actions in the MMI) and takes the user back to the previous frame.

In Table 13, the methods implemented in the errorMenuController is described. The errorMenuController handles events related to the errorMenuFrame. When an error is registered, this controller is responsible for adding an *errorMessageFrame* to the error-log. It also contains methods used by the errorMessageModel and aids in the tracking of the current status of the door. If an error is solved a method in errorMenuController is called and updates the log.

Table 13: Description of the most important methods contained in the errorMenu controller

Method	Description
init(self, model, view, timer):	Works the same as the init methods described in previous tables. The difference is that instead of calling a "bind_Buttons"-method, it binds them individually inside the init-method. This is justified by the small amounts of buttons in this interface.

Continued on next page

Table 13 – continued from previous page

Method	Description
addError Message(self, error	This method is used to add error messages to the error log. An instance of an errorMessageFrame is created and added to the scrollableFrame inside the error log. These frames are described in the table of frames earlier in the paper. In addition it appends the error passed as a parameter to the list of active errors that is an attribute in the scrollableFrame-class. Lastly it configures the expand button in the scrollable frame to use the *expand*-method described later in this table, and increases the attribute *nbrErrors* in this module by one.
updateErrors(self):	This method is called when a new error status is reported to the application. It removes all current errors displayed in the log. Then, the errorMessageList in errorMessageModel is iterated and all active errors is added to the errorlog by calling *addErrorMessage*. If active errors exists, the error buttons across the application is configured to appear and display the number of current active errors.
expand(self, error-number):	When an errorMessageFrame is created in *addErrorMessage*, the expand button in the frame is bound to this method. It expands the ErrorMessageFrame and adds an errordescription and a solution to the frame.

Table 14 describes the methods contained within the settingsController. The settingsController handles the actions that are taken when the user presses a button in the settingsFrame. The settingsFrame contains four buttons, which means that four different methods are implemented in the settingsController. It also implements a method for the action that happens if the error-button appears and the user presses it.

Table 14: Description of the most important methods contained in the settings controller

Method	Description
init(self, model, view, timer:	This init-method works the same as the init methods described in the previous tables. It assigns attributes and binds the buttons to individual methods, that is used in this interface.
bindButtons(self):	This functions binds the buttons in this interface. It uses helper-methods defined in this module. The helper-methods switches the frame to the frame that the button is connected to. These are, changePasswordFrame, changeHoldOpenTimeFrame, changedSpeedFrame and errorMenuFrame.

Table 15 describes the most important methods implemented in the holdOpenTimeController. One of the settings that can be changed is the time the door spends in open position in auto-mode. This controller handles the events related to the time-bar that can be adjusted by the user. It also implements a method for the action that happens if the error-button appears and the user presses it.

Table 15: Description of the most important methods contained in the change hold open time controller

Method	Description
init(self, model, view, timer):	Apart from doing the same as the previously described init methods for the controllers this also runs the update_view method at start. This is to ensure that the correct values are read in from the setting files.
bind_buttons(self):	This functions binds the buttons in this interface. It uses helper-methods defined in this module. The helper-methods switches the frame to the frame that the button is connected to. This includes the back button, confirm change and home button.
update_view(self):	This method is called when a setting has been changed or when the back button has been used to ensure that the correct value of the setting is shown. The value of the slider is then changed of the current value.

The timeController is described in Table 16. This module is rather simple and implements an alarm-functionality. When the alarm triggers the screen is going into screensaver mode and remains in that state until the screen is pressed. When the screen is pressed, the alarm increases the increment, which allows the user more time in the interface until the alarm overflows.

Table 16: Description of the most important methods contained in the time controller

Method	Description
alarmHandler (signum, frame):	The alarmhandler function implements what happens when the alarm overflows. When our alarm overflows, the alarmhandler is called and takes the user to the screensaver by calling loginmodel.logout().
increaseAlarm(self):	This function is used when the user presses the errorButton. It increases the alarm to 100 seconds, so the user has sufficient amount of time to read the ErrorMessageFrames displayed in the log.
main():	The main function in this module starts the alarm.

Table 17 describes the most important methods in the changePasswordController. Another setting that can be applied in the settings interface is to change the password. This controller handles the events related to that action. It is responsible for registering user input and displaying the length of the current attempt in the frame. Each password needs to be precisely 4 digits and to give feedback to a user when a number is pressed, a *-symbol appears in a container in the frame. This action is done in changePasswordController. This controller also handles confirmation that the password has changed successfully, as well as giving feedback if the attempt was unsuccessful.

Table 17: Description of the most important methods contained in the change password controller

Method	Description
back(self):	This method is called when the user wants to go back to the settingsframe, without changing the password. It works similarly to the *back*-method in loginController. The difference is that it erases the attempt of the changing the password, and restores it to the previous password. This is done in order to prevent the user to accidentally change the password.
successfullChange(self):	This method is called when the user has successfully changed the password. It resets the label to "Enter new password" and switches the frame to settings. In addition it clears the attributes in loginModel that keeps track of the password the user wants to change to. This is done in order to reuse them in the next attempt to change password.
reenterPassword(self):	This function changes the label in this interface to inform the user to reenter the new password. This makes sure that the user needs to enter the new password twice, in order to prevent mistakes. In addition this method clears the container containing the star-symbols in preparation for the password to be reentered by the user.

In Figure 8 below it is visualized how the different components within the package interact with each other. This is a sequence diagram and is an extension of the class diagram in Figure 7. In addition to the classes it also shows how the classes generally interact with each other. The arrows in the Figure indicate that a class calls a method in the class the arrow is pointing to. The main program is responsible for starting the application and is located outside the environment of the other modules. The source code for the entire GUI package implemented can be found on the GitHub repository in Appendix B.

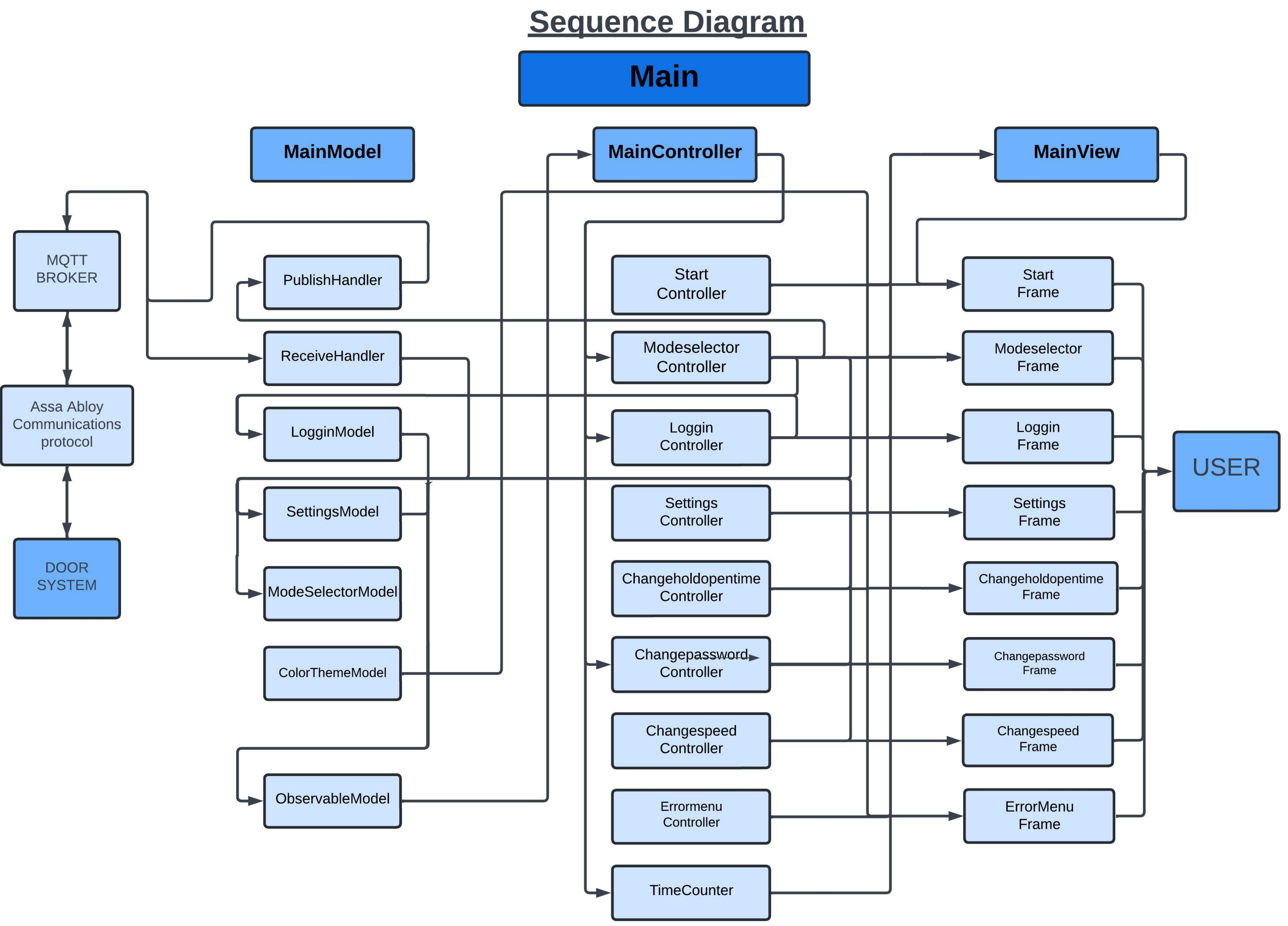


Figure 8: Sequence diagram

3.6 Communication

As mentioned in *Theory*, an important part of the prototype would be to establish communication with the door system, allowing the actions done by the user in the GUI to result in an actual event. When discussing how to implement this functionality with the supervisors at the company it was decided that a communications protocol already developed by the company could be reused and modified to make it easier and more efficient to set up the communications.

The communication protocol created by the company is written in C, which meant that it could not directly be integrated in the GUI application created in python. In order to allow these two programs to communicate with each other it was decided to use an MQTT broker which both the applications can connect to.

The door uses RS485 to communicate and has an available Ethernet port which the OMS can connect to. To enable the RPi to communicate with the door a custom cable, capable of sending RS485 from an USB port on the raspberry Pi to the Ethernet port on the door operator, needed to be created.

3.6.1 Assa Abloy communications protocol

To begin with, the communications protocol provided by the company was modified to suit the situation. The provided package was designed to serve as the communication protocol for the company's IoT devices. The primary function of the IoT device is to gather information multiple door systems and transmit this data to a remote server.

Initially, the code was streamlined by removing functionalities that were no longer needed and could potentially cause conflicts with our application. Some conflicts included for instance, internet connectivity was no longer required, since the RPi hosted the server locally instead of using the cloud, so components related to internet were excluded. Another example was the removing of communication with other door models in order to avoid conflicting communication sequences for different door models. Additionally for the program to run correctly, symbolic links (symlinks) had to be created for the compiler to find the downloaded packages stored elsewhere on the device.

Once the code only contained the essential parts needed, further modifications were made to adapt the code to the project. Previously, communication was conducted via RS485 through GPIO pins 14 and 15 on the Raspberry Pi. However, due to the display's requirement of utilizing all GPIO ports on the RPi, this setup was not possible. This was addressed by changing the port used for the communication. Previously it was set to */dev/ttyAMA0*, which points to the GPIO pins previously used. The communication should now be done using one of the USB-ports on the RPi. This is stated by using */dev/ttyUSB0*.

3.6.2 MQTT

To allow communication between the GUI application and the communications protocol an MQTT broker would be used. To set up the MQTT broker locally on the device, an MQTT broker was first installed. The Mosquitto broker packaged was installed with the following command in a terminal: *sudo apt-get install mosquitto*. After the installation,

a few configurations were necessary to ensure proper operation of the broker. These configurations were made by editing the *mosquitto.conf* file located at */etc/mosquitto/*. Two lines were added to this conf-file:

listener 1883
allow_anonymous true

The first row specifies the port, 1883, on which the broker should listen for incoming messages, which is the default port for MQTT communication. The second line indicates that clients can connect to the broker without requiring authentication with a username or password. Since the broker operates locally on the device, this functionality was deemed unnecessary. With these configurations, the setup of the Mosquitto broker was completed, and is started by running *mosquitto -v* in a terminal.

To setup communication between the two applications, both needed to connect to the MQTT broker. This was achieved using the MQTT PAHO library, which is available for both Python and C.

In Python, it was decided to create two clients, each running on its own thread: one for publishing messages and one for subscribing to and receiving messages. Utilizing available documentation [23], classes for these two clients were created. Instances of these classes are then instantiated in the mainModel class, followed by the execution of the start method to initiate the thread's operation.

The publishHandler class, responsible for publishing messages, includes a method named publish. This method publishes a message to the broker containing a topic and a message payload. Given that the publishHandler is instantiated within mainModel all classes taking model as a parameter can access and call the publish method. This means that all controllers can call the method as a response to user actions.

The communication protocol already has functionalities to connect to an MQTT broker since it previously was connected to a remote MQTT server. Modifications were made to instead connect it to the broker hosted locally on the device.

In order to make the prototype more realistic which means making it more similar to a MMI for sliding doors, it was desirable to implement a functionality on the RPi to make it only display the application and not anything else of the OS for showcasing purposes. This meant that the application had to start as part of the boot process when starting the RPi. This could be achieved in two steps. The first action that had to be taken was making a shell script that starts all the necessary processes. This was done by making a .sh-file that runs in a terminal. From this terminal, the shellscript is starting three more terminals that starts the mosquitto broker, one that runs the communication protocol and one that runs the GUI. This is all the processes needed for the application. The shellscript is displayed below.

```
#!/bin/bash
function start_application(){
    lxterminal -e "bash -c 'mosquitto -v; exec bash'" &
    sleep 2
    lxterminal -e "bash -c 'python3 /home/hampus/src/main.py; exec bash'"  &
    sleep 1
    cd /home/hampus/src/ccom-development
    #lxterminal -e "bash -c 'sudo /home/hampus/src/ccom-development./ccom; exec bash'" &
    sudo ./ccom
}

start_application
```

The second action that had to be taken was to make the shell script part of the booting process. This was done by creating a service file in */etc/systemd/system* directory from root in linux. SystemMD (system and service manager) is the first process to run on startup in linux and starts all other processes that is running on boot. By creating a service file within the system folder that points to the shellscript, it was achiveable to make the shellscript part of the booting process. The service file consists of three headers. *Unit, Service, Install.* The Unit header contains for example description of the service and when the service should be started in the boot process. The service header describes the process that should be running and points to the location of it (in our case, it points to the location of the shell script). Finally, the Install header allows the process to run by handling the dependencies of the process.

3.6.3 Hardware layer of the communication

The connection between the control unit in the door is done by RS-485. RS-485 is a standard for the physical layer of communication. As mentioned previously, typically, the connection is done by using the GPIO-pins on a Raspberry Pi. The GPIO-pins support a number of functions, including PWM, SPI, I2C and serial communication. The communication for our door is done by serial connection which is supported. However as touched on before in section "Working setup" under chapter 3, the GPIO pins needed to be designated for the touch-screen and could therefore not be used for serial communication. Instead, it was decided to designate one of the four USB-ports on the RPi for serial communication to the door.

To achieve this, an USB to serial-adapter was used called RedCom USB-COMi [24]. The adapter has two ports. One supports USB, and the other supports D-SUB9 connector. The door system communication is done using half-duplex mode. This means that both units (RPi and door) can send/achieve data, but not at the same time. In addition the data that is sent with an inverted signal, to reduce noise in the communication. Therefore only three pins are required for the serial adapter to be engaged in the communication. One sends Data+ and the other Data- and the third as ground. The RedCom adapter had to be configured to be set up for the half-duplex mode. This was done by shorting wires on the board of the adapter according to the manual provided by the data sheet[24]. The correct setup for our implementation can be seen in Figure 9 below.

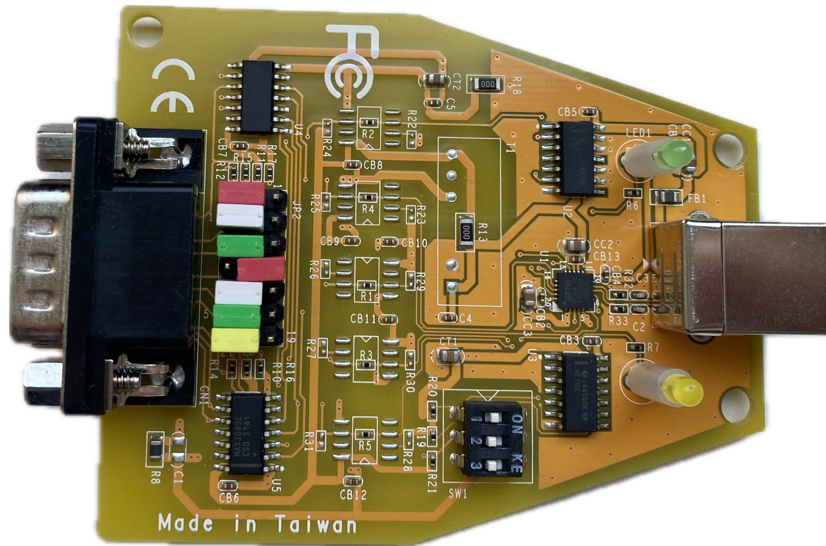
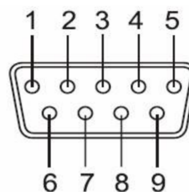


Figure 9: Inside of the redcom adapter displaying the setup for half duplex communication

The door system uses Ethernet sockets to connect peripherals and since the adapter only has a D-SUB9 outlet, a custom made cable was needed. This cable was created by cutting an Ethernet cable in half and adding a D-SUB9 female connector to the end that was cut. Since only two wires and a ground wire were needed, they were the only ones needing to be soldered on the D-SUB connector. When creating the cable the first step was to find which three wires from the door is used for Data +, Data - and ground. The documentation for this was provided by the company but is no to be shared. These three wires were now to be soldered on to the correct pins on the D-SUB connector. This information was obtained from the data sheet of the Redcom serial adapter and can be seen in Figure 10 and 11.

DB-9 Male Connector Pin Assignment



USB-COMi & USB-COMi-SI USB to RS-422/485 Adapters User's Manual

Figure 10: Pinout for the D-SUB9 (male)

RS-485 2-Wire (Half duplex) Signal Pin-outs of DB-9 Male

Pin 1	Data- (A)
Pin 2	Data+(B)
Pin 5	GND

Figure 11: The pins that are used for half-duplex communication

Once the cable was made it was tested with a multi-meter to see that a signal is sent through each wire. When connecting the setup to the door unit, an oscilloscope was used to test the signals. As can be seen in Figure 12, a number of messages is being received through the cable at an interval of 1 second. Furthermore the characteristics of the message can be seen in Figure 13. It is apparent that there is a larger difference in voltage between the ones and zeros being sent, ensuring correct transmissions of bits. As can be seen in the figure, some noise is recorded in the signal, but it is not enough to disturb the communication. The complete cable to be used between the door and the OMS prototype is displayed in Figure 14.

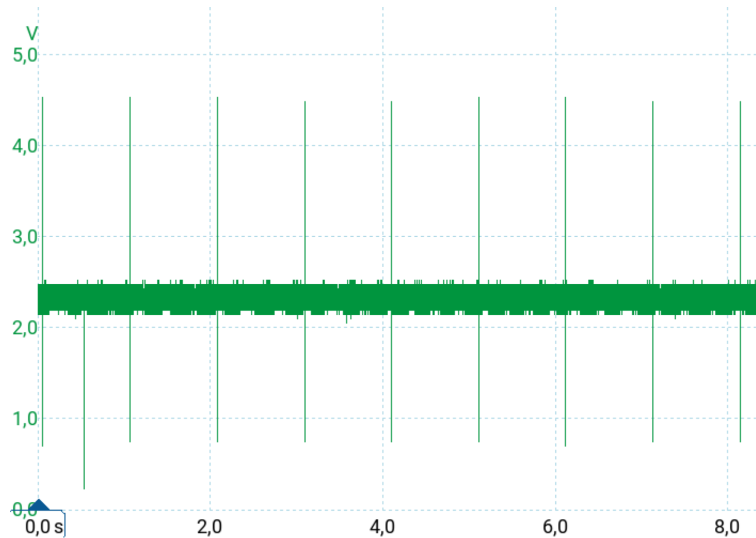


Figure 12: Picoscope measurement showing messages sent once every second.

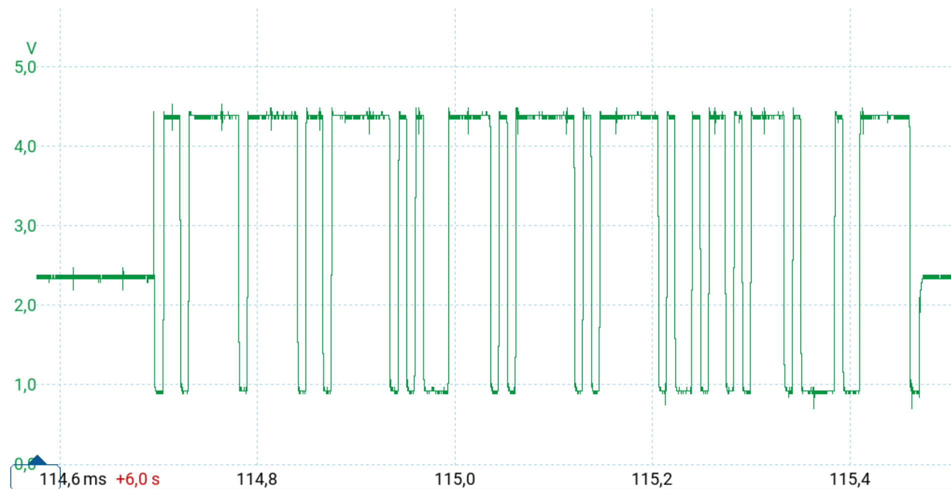


Figure 13: Picoscope measurement showing the characteristics of a message.

Figure 14: The full cable between the RPi and the door.

3.6.4 Path of communication

In order to describe the complete process and the path of the communication between the user and the door, it is believed that the best way of doing this is to look at an example and follow the message. The following example is of what happens if the user wants to change the operating mode of the door to *Automatic*. Due to confidentiality, it is not possible to go into details about the bytes being sent or the low level communication as a whole.

Once the user presses the button for Automatic in the mode selector menu a message is published to the MQTT broker from the GUI application. The topic of this message is *'c/testdevice/slidingDoor/setOpMoAuto'* while the payload is kept unused. This topic together with the rest of the available modes are subscribed to by the communication protocol at the start of the application. (In reality, the communication protocol subscribes to many more topics).

When the message has been received by the communication program, the correct handler is selected. When the handler is created, a subscription topic and a function is passed as parameters. When a message is posted on this topic, the handler connected to that topic runs the function. In our example, this function creates a message, consisting of a specific set of bits and this message is added to a queue of messages that is going to the door. Once the message reaches the front of the out-queue it is sent to the door.

In order to change parameters of the door, which includes modes, a few iterations of communication back and forth is required. This methodology is based on the TLS handshake protocol [25] and is aimed to improve the robustness of the system. It ensures that when a two party communication (in our case, the OMS and the control unit och the door) is communicating, both parties are ready for the requested change. Firstly the operator must be unlocked to allow changes to be made. This is done by sending an unlock request followed by a message payload including the correct pass code from the communication program. Then, the OMS waits for a positive response, which occurs when the pass code is correct. When this message has been received a request is sent back to the control unit on

the door to change the operating mode to Automatic. The door responds to this by stating that it is allowd to change the operating mode. By responding to this with a message to change the operating mode, the mode is changed. A final message is returned from the door indicating that the mode has been changed to the requested mode, Automatic. Once the parameter is changed to door returns to the locked mode and a new unlock command must be sent in order to make further changes on the door.

When the door sends messages (like error information), this is handled by the communication protocol. The messages are received in the form of bytes and are added to an InQueue in the program. This queue is maintained by response handlers in the communication code. These handlers works similarly to the handlers described earlier in this section. The correct handler is selected to receive the message. Then, the method attached to the handler processes the message by validating it's legitimacy and then processes it by dereferencing the message and creating a Json-object to store it. This Json object is posted as a message in the MQTT-server which then can be accessed by the GUI.

3.7 Casing

In order to achieve a finished and elegant look of the prototype a case was designed to house the raspberry Pi and the touch display. The design was constructed using the design software Solidworks which the company has a license to.

In order to be able to visualise how the components should be mounted and how the case should be designed around it, available CAD-models of the RPi and the display were used. These files were obtained from CADgrab which is a CAD community where people upload CAD models that are free to use. The used models can be found here [26] & [27]. After downloading the files they were assembled together like the components are to be mounted in reality, which can be seen in Figure 15.

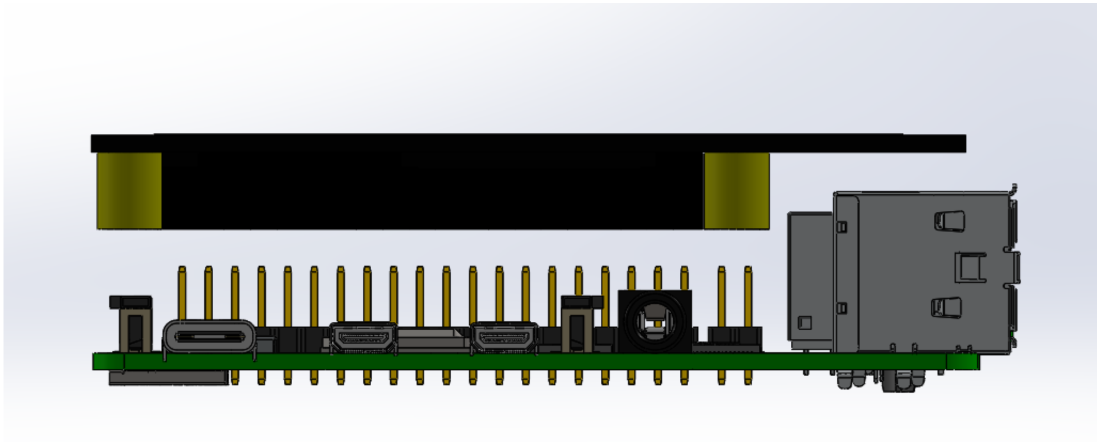


Figure 15: Used CAD models mounted together

Cutouts were made for the necessary ports of the device, as well as for the ports required for working on the Raspberry Pi, such as connecting to an external display and accessing the internet via Ethernet.

The LED display is mounted on the Raspberry Pi using the four mounting holes on the board. The display has screw holes in its four pillars, which extend from the display to the Raspberry Pi board. This mounting method was also used to secure the Raspberry Pi within the case, as depicted in Figure 16. To ensure secure mounting without damaging any components on the RPi, the board was elevated a few millimeters above the case's surface, as shown in Figure 17.

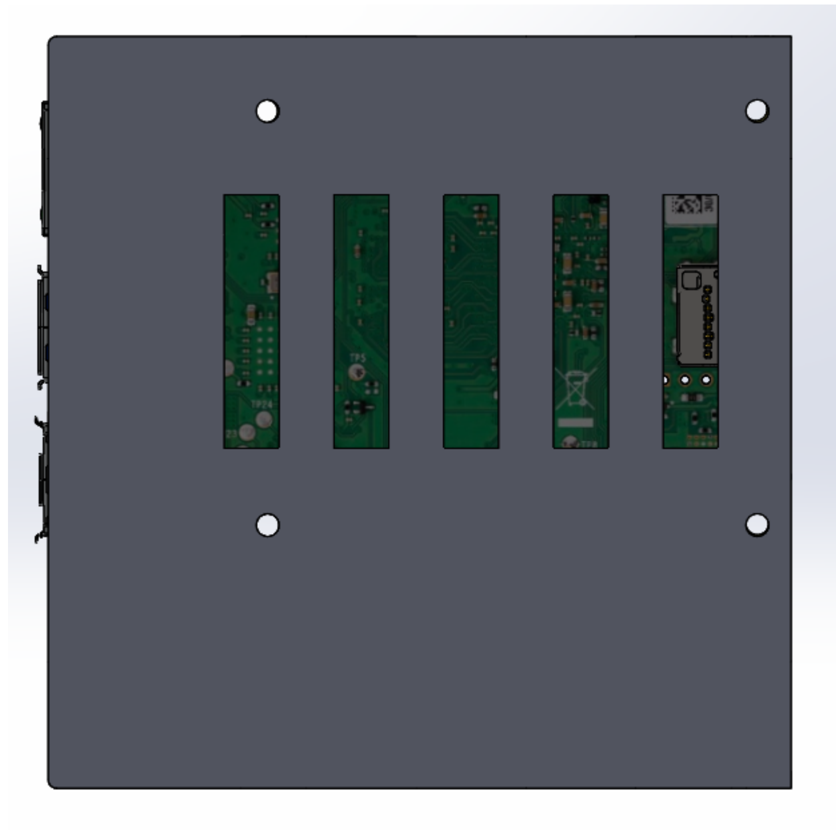


Figure 16: Bottom view with 4 mounting holes and ventilation slits in between the mounting holes.

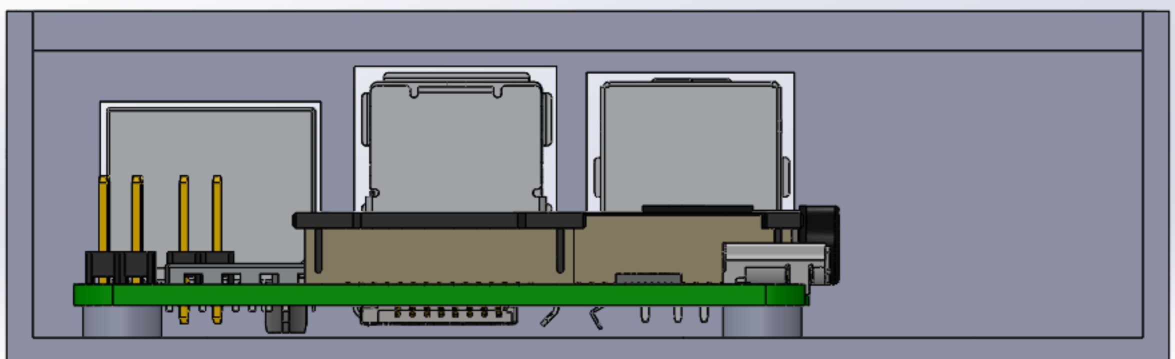


Figure 17: Side view showcasing how the Raspberry Pi is mounted.

To facilitate easy mounting of the Raspberry Pi within the case, the casing was divided into two parts. This was done by dividing the CAD model into two pieces using the split function.

After printing the initial iteration of the 3D model, it became apparent that several adjustments were necessary. The first case design overlooked the need to prevent overheating of the Raspberry Pi. Therefore, the second and final iteration featured a grid on the backside of the case to promote airflow and keep the Raspberry Pi cooler. Additionally, the mounting mechanism for the side part was modified in the second iteration. Instead of screws, the decision was made to snap it into place using holes on the main part and small pins on the side part.

It was also realized after the initial iteration that it would not be feasible to place the power cable hole on the same side as the other ports (USB and ethernet ports), as desired, due to the cable's inability to bend sharply. Consequently, it was relocated directly in front of the power cable port on the opposite side. The final design is depicted in the accompanying Figure 18 below.

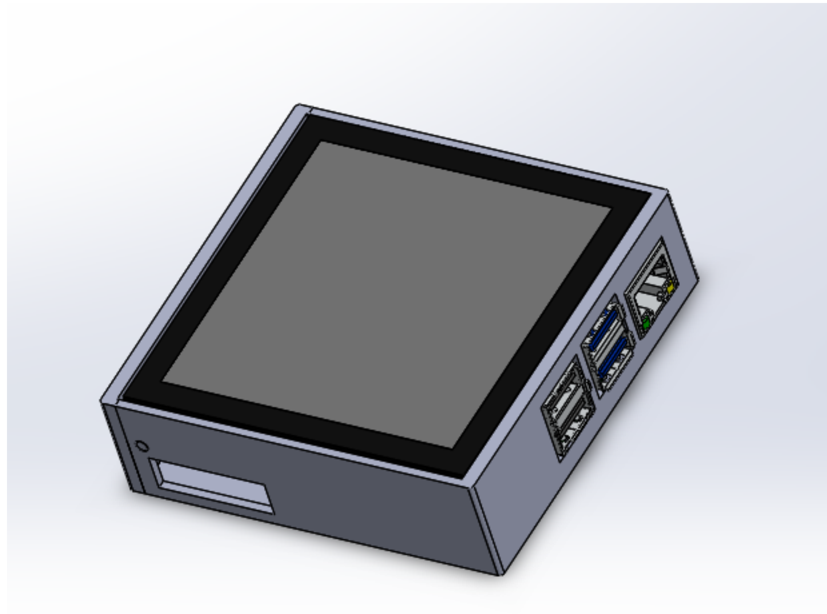


Figure 18: Final CAD model of the case with the component mounted inside.

When the desired design was fulfilled the design was exported to .3mf format to be 3D printed. The .3mf file was then modified within Prusa's software to be prepared for 3D printing. The infill setting was changed to *Gyroid* with 60 percent thickness as recommended by the company to save on material while maintaining a robust model. Once the print completed successfully the raspberry Pi board and the display was mounted to achieve the finished model as seen in Figure 19.

Figure 19: The printed and assembled case.

4 Results

The final result contains most of the features described in the specifications. The prototype is able to establish communication with the sliding doors of the company. Communication has been implemented for changing operating mode, receiving errors as well as obtaining the current mode of the operator. The graphical user interface works as intended, altough some flaws are still present and this is touched on more in the discussion. The final design of the interface includes most of the findings from the user analysis.

4.1 GUI navigation and design

When the GUI application is started the user is first presented with the screen saver display which is the menu shown when no user interacts with the OMS. This display shows the user and customers the current mode of the sliding door as well as an error status if the operator currently has an error. An example of this can be seen in Figure 20.

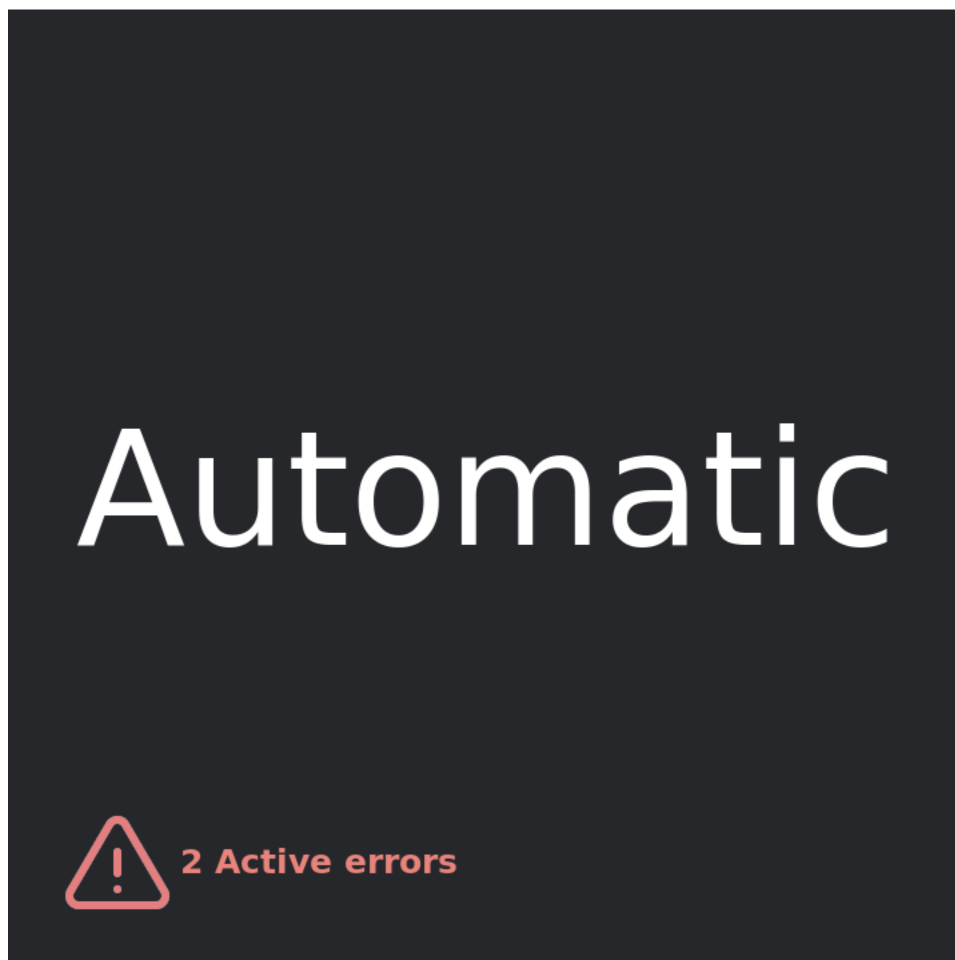


Figure 20: The screen saver menu.

If the user has been idle for more than 10 seconds, meaning that no button has been pressed for that period of time, the display returns to showing the screensaver display.

When the user interacts by pressing on the screen, except for on the error button if it is visible, the user is brought to the Mode Selector menu which also acts as the main menu of the OMS. In this display the user can switch between the available modes: Automatic, Closed, Auto partial, Hold open and Exit only. The currently selected operating mode is clearly indicated by a darker blue color compared to the other modes. The Mode Selector menu can be seen in Figure 21.

As mentioned before in this work, the master's thesis conducted by other students [21] at the company found that the *Automatic* and *Closed/Off* modes were the most frequently used, with some user exclusively using these two modes [21]. Consequently, it was decided to prioritize these modes by placing them at the top of the display, ensuring they are the first options users encounter. To enhance their visibility and accessibility, they are presented as larger buttons.

Additionally, insights from the customer study indicated that the label *Closed/Off* was unclear to some users, as *Off* might imply complete shutdown of the door system, which is not the case[21]. Therefore, the label for this mode was simplified to *Closed* to eliminate any confusion.

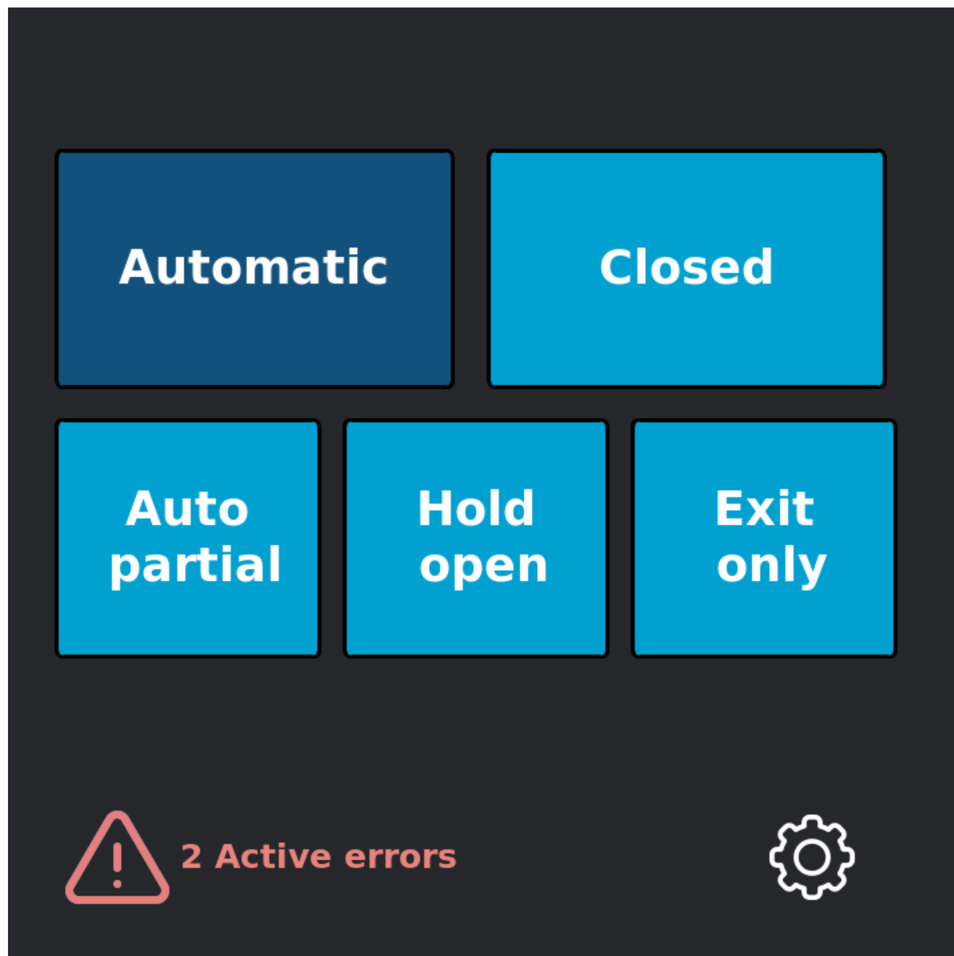


Figure 21: The Mode Selector menu.

If the user desires to close the door by changing the mode to *Closed* an authorization with password must be done since this might hinder an emergency exit. The user is then brought to the password menu and the action is not performed until the correct password is entered.

From the mode selector menu the user can also access the settings menu via the cog wheel button in the bottom right. If the user not yet has been authorised by entering the password the user is redirected to the password menu to enter the correct password before reaching the settings menu. When the user has entered the password once, authorization is no longer required. The application remembers that the user has already entered the password until the user has been idle and returned to the screensaver. Then, password is required again to enter the options that requires authentication, since it might be a new user that is using the OMS.

In the settings menu the user is presented with different settings that can be changed, see Figure 22. These include *Hold open time*, *Opening & Closing speed* and *Change password*. The user also has a button returning them to the main menu. When pressing one of the different buttons the user is brought to a menu dedicated to that setting.

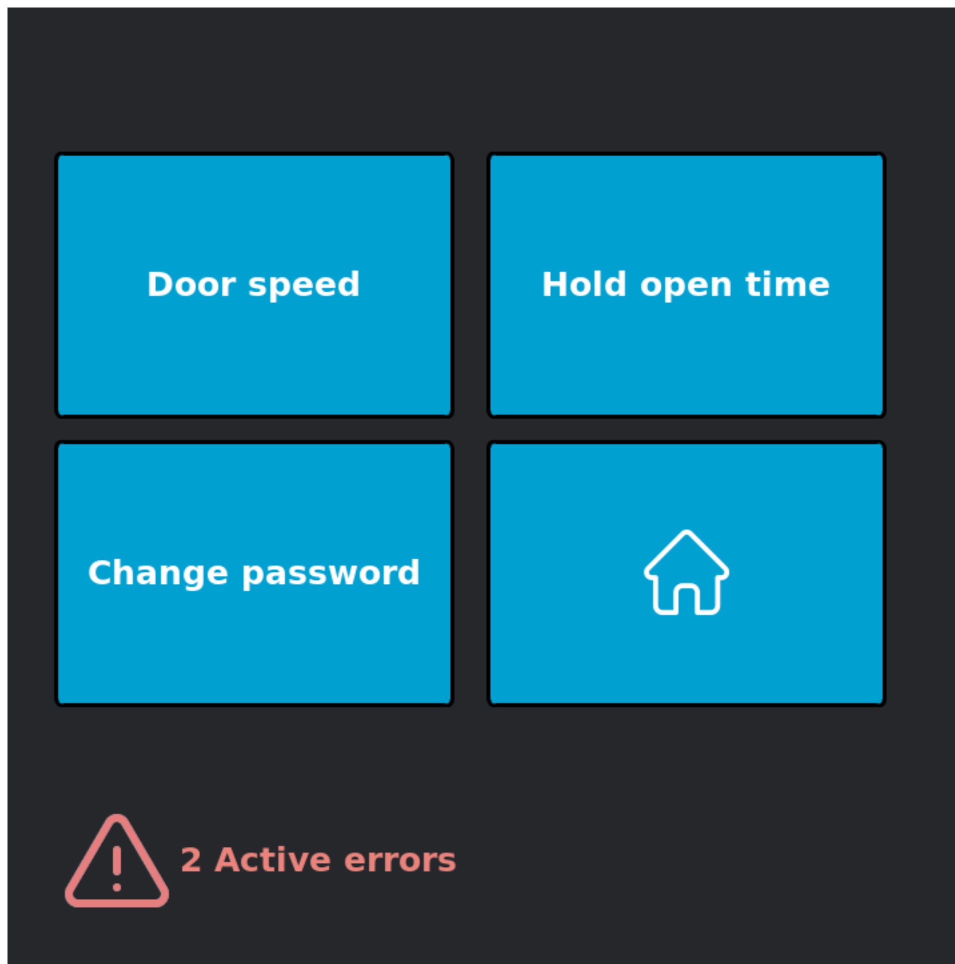


Figure 22: The Settings menu.

The menus for adjusting parameters are nearly identical, differing only in the parameter being modified. Illustrated in Figure 23 below is the menu used for adjusting the door speed. In thia menu, the user is presented with two sliders that can be dragged to achieve the desired setting. Upon finalizing the adjustment to the desired value, the user can confirm their selection by pressing the confirm button, which returns them to the settings menu. The user also has the option to go back to the settings menu without changing any of the parameters using the back arrow. The same is true for the home button as the user is redirected to the mode selector menu. The display for changing the speed of the door can be seen in Figure 23.

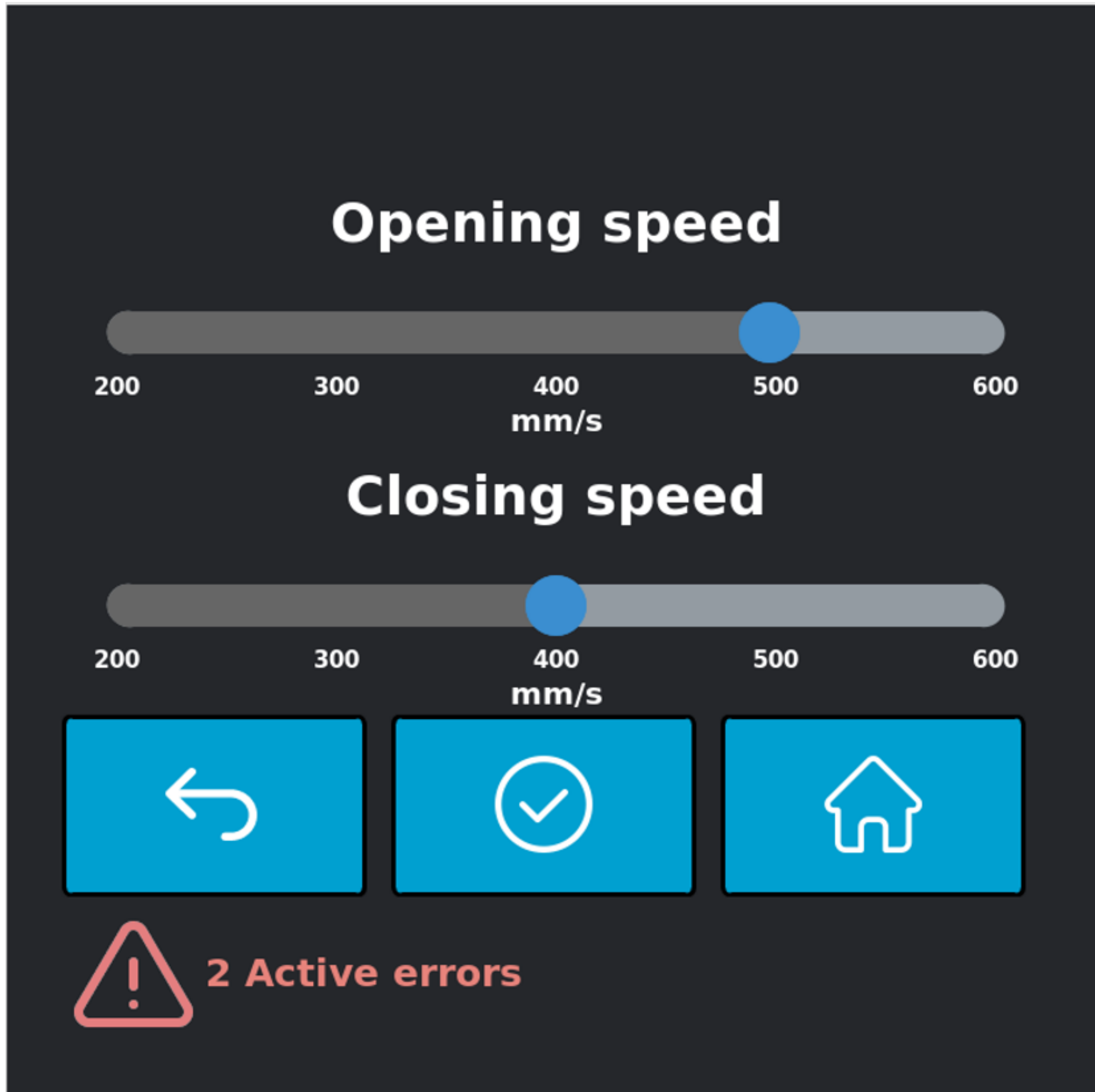


Figure 23: The door speed settings menu.

As touched on previously the user needs to enter a password to access certain functionalities, this includes setting the door mode to closed and changing parameters on the door. When this is required the user is brought to the password menu as can be seen in 24 below.

In this menu, the user can enter the password using the buttons corresponding to number 0-9. The amount of digits entered is visualised in the 4 white boxes above the number pad.

The frame also contains an undo button to remove the last entered digit. Once four digits are entered, the length of the password, a response is given to the user directly without the need of confirming that the password has been entered. If the password matches the current selected password the user is therefore redirected to the menu they are attempting to reach. Once the user has been authorized by entering the password the user has full authority to change all settings and the mode to close without having to provide the password again. The authorization is lost when the user has been idle for more than 10 seconds and the display has returned to the screen saver frame. However, if the entered password does not match the correct password the user is made aware of this by changing the text to *Wrong password, try again* with a red font to make it clear for the user that this was not correct. The user may now attempt to enter the password again. If the user regrets the decision to enter password or does not have authority it is possible to navigate back to the previous menu using the back or the home buttons at the bottom.

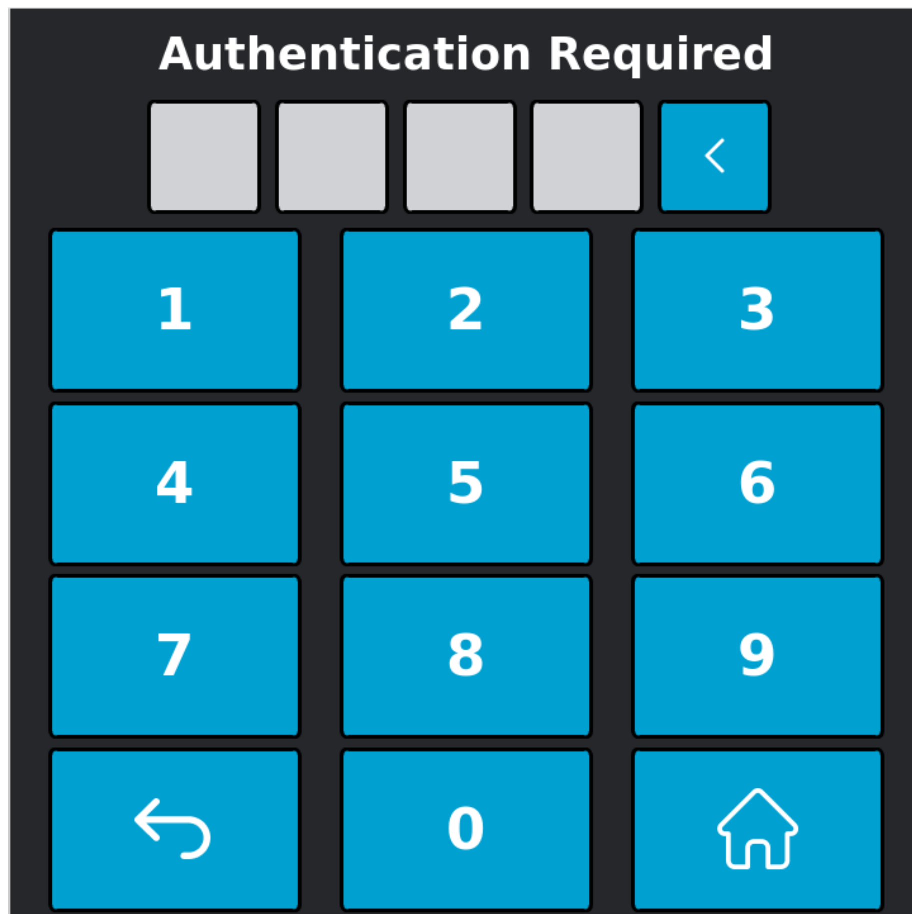


Figure 24: The Password menu.

All of the previously mentioned menus, except the menu for entering passwords, indicate present errors by displaying a red warning triangle when an error is active. Upon pressing this button, users are directed to the error menu. This interface is presented in Figure 25. This menu displays the current errors for the user in a list. Each error is presented in its own box, containing the error number followed by the error title. On the far right of each box, there is a button that users can press to access additional information regarding the

error. When this button is pressed, the respective error box expands vertically to reveal a more detailed description about the error followed by a suggested solution that the user can follow to resolve the error. The expanded error is also highlighted in a darker color to draw attention to it.

At the bottom off the error menu, three buttons are present. The left button is a back button, which returns the user to the menu from which they entered the error log. The middle button is a reset button, intended to perform a system reset, including the door system. Although the functionality of the reset button has not yet been implemented, it exits to visualize how it would operate in the real product. When reading the documentation for errors in the door, it became clear that many solutions include trying to reset the system. The idea is that the user reads the suggested solution displayed in the error box and if it suggests the user to reset the system, this can be done without problems in the error menu. On the right, a home button is present, allowing users to navigate back to the mode selector menu. The current Error Menu features a thin slider at the right of the screen which allows the user to navigate up and down in the error log to see all the active errors, when multiple are active.

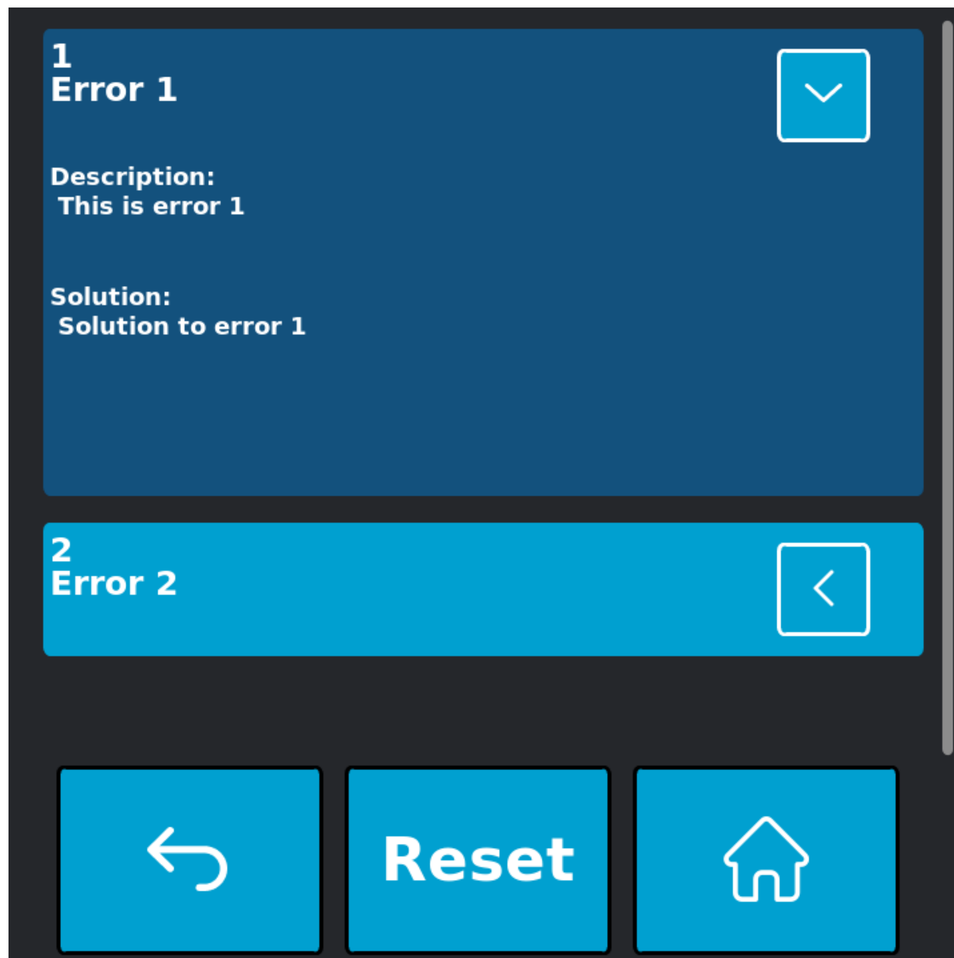


Figure 25: The Error menu.

4.2 Casing

The 3D printed case to house the components turned out well. The measurements and fit of the case together with the RPi and the display are perceived as sufficient. The robustness of the case is enough for its purpose as a prototype but would need additional support for a final product. The mounting mechanism for the raspberry Pi and the display to the case works well and ensures that the components are mounted securely. The finished printed model can be seen in Figure 26 and Figure 27

Figure 26: Main part of the casing, showing holes for ports and slits ventilation

Figure 27: The printed case, displaying how the two parts are attached together

4.3 Communication

The communication between the door and the graphical user interface works without issues. The communication implemented so far is to retrieve the current operating mode and errors from the door as well as changing the operating mode. Due to time limitations the communication for changing parameters on the door has not yet been implemented.

However, the functionality for it exists within the GUI and the user can still get the picture of how it should be done.

4.4 Final result

Combining the GUI with the printed case the result is as seen in Figure 28. The entire program is started automatically when the device is connected to power. A concept picture of how the product would look when mounted to the door is visualized in Figure 29.

Figure 28: The finished prototype displaying the mode selection menu and the operating modes that can be selected

Figure 29: Concept picture of the prototype mounted on the wall next to a door.

5 Discussion

Overall the result has been successful and we have been able to produce to the company what was desired of this master thesis. The prototype allows the company to experience how an OMS with a touch display could look like and operate. During the project the decision within the company was taken to start developing such a product resulting in our first prototype to be in the center of the development. Our work and research proved to be of relevance in taking early decisions about the product specifications. The decision to opt for the 72 x 72 mm LED display resulted in the company transitioning from the smaller, around 60 x 60 mm display to a similar size our prototype used.

The product owner also mentioned that the prototype will be used during the software development to quickly be able to test new functionalities before being able to see it in the actual product. By modifying the existing communication protocol within the company, our prototype is currently the only device within the company that can implement a touch based GUI and test it against a sliding-door. This is of value to the process of developing a touch based interface within Assa Abloy, as it will be important to show ideas and designs to other stakeholders within the company, and may aid this process.

Additionally, this project is an example of how to navigate the extreme programming workflow which in this context was applied throughout the development phase. Initially, the specifications for the project were vague and did not cover the whole GUI. Therefore we had to iteratively change the appearance of the interface, as well as add functionalities that were not described in the beginning. In the world of today, this way of programming is becoming more and more important. Since the tasks that the programmers faces becomes more complex, the initial list of specifications the developer works with might not contain the full information that the project needs, or it might change due to new demands. The experience from this project is that the results benefited from a vague formulated set of instructions. By first implementing the most important or general specifications which in this example are the initial specifications listed in chapter 3 (section 3.3), and then customize the implementations in more detail over multiple iterations, does provide a result that is in a high degree customized to the stakeholder. This is based on that the stakeholders are available for feedback in each iteration.

This way of working is mostly applicable to projects where it is important that the result is highly oriented around the stakeholders opinions, and where these opinions may differ. In our project, the stakeholders where represented by the other master thesis (users), representatives from Assa Abloy (company) and representatives from LTH (academic). By identifying that the success of the prototype were most dependent on the opinions of the users and of Assa Abloy, the workflow were mostly centralized around these entities.

In our experience, the drawback by working with a vague formulated list of demands is that the development of the prototype is in risk of taking longer time to yield an end result. This is a consequence of the iterative workflow. If the end result is clearly stated in the beginning and a full list of specifications that comprehensively describes the desired outcome of the project is provided, the iterative element of extreme programming is not needed and the amount of time the project will take is decreased. However, the risk of dissatisfaction of the stakeholders increases as they have not been part of the process in the same regard as when working in an iterative manner. In our master thesis, we are convinced that the results yielded from working in an iterative manner, were enhanced rather than diminished.

The prototype is intended to work both while connected to a door and by itself. It is primarily meant to be connected to a door since this displays the full capability of the device. However, it is not always easy to showcase it when connected to a door and therefore it is made possible to also demonstrate how it works by simulating incoming messages from the door and reading the parameters directly from the settings file. The simulation of incoming messages is done by using the GUI:s publishHandler to post the simulated door messages direct to the MQTT broker. By allowing the device to function without connection it facilitates the iterative work process by quickly being able to showcase the visuals of the GUI. During the development process the device was connected to a real door only during a few occasions. Connections to the door were almost explicitly done when testing that different aspects of the communication were working as intended with the prototype connected to the sliding door.

Establishing communication proved to be more time consuming than expected. Since this was an seen as an important part of what we wanted to present to the company it was decided that, although it would take more time than planed for, it should be of focus. It was also of value to experiment with the low level communication and understand how a robust program for low level communication is built. One problem encountered when implementing the communication protocol was the creation of symlinks in the library folder which is passed in the make-file when building an executable. Symlinks are symbolic links in the linux os that points to a file/folder or a filesystem which is in another location in the computer [28]. In our case, it was needed to create symlinks to libraries, for example, libpaho, libmqtt, libmodbus and log4c. The libmodbus is an open source library for communicating with modbus devices Modbus is a binary protocol for transmitting bytes over RS-485. It was needed in our project to establish communication with the door. Log4c is a library for C, which is used for logging throughout the communication protocol. This is primarily used when debugging the program, and logs mostly errors in the communication. In previous course project we have done during the education at LTH, we have never implemented symbolic links which is why we had some problems locating the errors produced by having improper symlinks in the library. Therefore, it took some time to locate the error, locating the location of the proper files in our system and then linking them to the symlinks in the library used in the makefile. Other problems that arose were that the communication protocol is extensively built and implemented by another person within Assa Abloy[28]. In addition, its primary use was to listen to multiple IoT-devices, which meant that it had multiple packages which were not of relevance to our project. Therefore, as mentioned in the methodology section, downsizing and modification was needed in order to make the communication work for our prototype. By having re-occuring meetings with the developer, we were able to make it work, but this would probably not have been

the case if the developer that originally had written the library, was not available to aid our development.

This resulted in that less time was available for adjusting the design of the graphical user interface from the final conclusions found from the master thesis produced by Liljenberg and Olsmats [21]. However, since a solid foundation for the GUI has been implemented it would not require substantial time to change the appearance nor the navigation within it. We feel like the amount of time spent on the final adjustments in the GUI that were based on the UX-analysis were sufficient to produce a GUI designed for the user.

Reflecting on the project, we feel that choosing python as the programming language instead of C was the correct decision. Using python allowed for the GUI to be implemented quickly and changes to be made during the agile process with ease. However, had the graphical user interface been implemented in C, integration of the communication component within the same program would have been possible. This would have eliminated the need for an MQTT broker as a mediator between the two parts, streamlining communication between the user and the door. Additionally, this would also have meant that the programming structure would more resemble the code needed in the final product meaning that the company could gain more inspiration and be able to use more of the implemented functionalities.

It was decided to create the program in python since both the participants of this master thesis are more familiar with the syntax of python compared to C and therefore felt that it would result in us being able to quicker implement the code. The decision was also taken together with the supervisors at the company who stated that they had no preference of programming language used and therefore agreed that the language within we have most knowledge should be chosen.

6 Conclusion

In conclusion, the resulting product achieves the intended purpose of this project. It allows the company to get an early hands on prototype in order to understand how such a product might feel and interact with a door. It has been created in an iterative manner with regular input from stake holders and from the customer analysis simultaneously being conducted. Due to the program being automatically run at start up it is only required to apply power to the device in order to use the prototype. This makes it possible for inexperienced user to quickly get the prototype of and running and test its functionalities.

The implemented GUI package has been structured in a way that allows for quick adaption and changes in design and navigation. This allows the company to keep working with it during the early stages of the product development to test it in a rapid manor.

The project can be seen as an example of how to rapidly develop a prototype resembling a finished product can be done with limited resources, using off the shelf equipment, a Raspberry Pi and a display. The report can be used as a guide and inspiration of how such a product can be created for other project with similar purposes. The structure of the code allows for easy modification to create a GUI which meets the requirements for other purposes as well.

6.1 Changing scope of project

As mentioned in the problem description, when we first started the master thesis, the scope of this project was extremely wide and our understanding of what was required of us from Assa Abloy was slightly misunderstood. Before the project started, we described the project in a target document. This description was changed during the course of the project. Initially, it was intended to start with an Arduino or a Raspberry Pi to design an initial prototype of a GUI and establish communication with the door. In the later stages, it was intended to design a circuit board with a micro controller, with an implemented GUI based on the results from working with the RPi. However, it quickly became apparent that this was neither a requirement from Assa Abloy nor feasible to accomplish within a single master thesis. This description should rather be spanned over several master thesis' in order to be accomplished. The scope of our project was shifted towards implementing a GUI that directly meets the criteria of the stakeholders, and in the later stages, implement communications to the door through our application.

6.2 Acquired knowledge

The project has brought us challenges within several different engineering fields. This has suited us well since we enjoy the variety as well as are interested in gaining experience in different fields. Programming the GUI, testing the communication by reading voltage signals and modifying low level communication has tested our knowledge within data, electronics and low level programming. Mainly the project has been centered around implementing the graphical user interface and establishing communication with the control unit on the door.

This meant a constant challenge of achieving the desired design with the available tools within the customtkinter library and understanding/modifying an extensive pre-built communication protocol.

We have also gathered knowledge about different communication protocols as well as how a hardware abstraction layer (HAL), is implemented. It was interesting to discover the substantial amount of different protocols available and see the advantages and uses for each.

6.3 Future Work

Since the final product of this project is just a first prototype to prove the concept of including an OMS with a touch display in the companies portfolio, the product has several areas needing improvement before reaching a version ready for market. Below we will list the main areas where we see possibilities for improving the performance of the product.

As mentioned earlier in the report all of the available frames (Table 8) of the GUI are created and displayed at the start up of the program. By doing this it is very simple to change between the frames and the visual changes from the data can be done in another frame when another is displayed. The downside of this approach is that the displays running in the background take up computing power and uses memory of the system. In our prototype this is not an issue and the program is not impacted negatively. However, when the company would create a product to market it is of great importance of keeping the cost down. This means that the performance of the MCU would be lower and therefore it is more important to keep the amount of tasks running on the processor to a minimum.

The current application has a known bug that was not addressed within the designated time frame, as it was not deemed a critical priority. This issue is the difficulty in scrolling through error messages using touch due to the thinness of the scroll bar for the scrollable frame. This bug persisted as it only becomes apparent when three or more errors occur simultaneously, a relatively rare occurrence. Unfortunately, the library used does not offer an option to adjust the width of the scroll bar, necessitating a more substantial modification for resolution.

One potential solution is the addition of navigation buttons within the scrollable frame, allowing users to move up and down. Another approach involves incorporating drag functionality within the frame, enabling users to swipe for navigation, work has been done for this but was not completed.

To enhance the prototype's completeness, additional functionality could be integrated into the product. For instance, the reset button functionality, as mentioned earlier, is not yet implemented. Moreover, the door has numerous adjustable parameters beyond those outlined in the report, presenting opportunities for further implementation.

References

[1] Christer Johansson. Product specialist, sliding doors, assa abloy entrance systems.

[2] 4 Raspberry Pi (Trading) Ltd. *Raspberry Pi 4 Model B*, 4 2024. Rev. 1.1.

[3] Arduino. *Arduino Uno*, 11 2023. Rev. 3.

[4] ADMetro. Resistive & capacitive touchscrens: Know how these two popular touch solutions differ. https://admetro.com/news/resistive-capacitive-touchscreens-know-how-these-two-popular-touch-solutions-differ/#:~:text=Applications%20with%20capacitive%20touchscreens%20allow,the%20human%20body%20for%20input., 2022.

[5] Alexandros Karagkasidis. Developing gui applications: Architectural patterns revisited. In *EuroPLoP*, 2008.

[6] Ph.D Burbeck, Steve. Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc). "https://www.researchgate.net/profile/Steve-Burbeck/publication/238719652_Applications_programming_in_smalltalk-80_how_to_use_model-view-controller_mvc/links/5575a00508ae7536375024c7/Applications-programming-in-smalltalk-80-how-to-use-model-view-controller-mvc.pdf, 2012.

[7] Mvc design pattern. "https://www.geeksforgeeks.org/mvc-design-pattern/, 2024.

[8] Kivy. Kivy. https://github.com/kivy/kivy, 2024.

[9] Andrés Rodríguez et al. Kivymd 2.0.1 dev0 documentation. https://kivymd.readthedocs.io/en/latest/, 2022.

[10] Python software foundation. tkinter — python interface to tcl/tk. https://docs.python.org/3/library/tkinter.html, 2024.

[11] Tom Schimansky. Customtkinter documentation. https://customtkinter.tomschimansky.com/documentation/, 2024.

[12] Kate Moran. The aesthetic-usability effect. https://www.nngroup.com/articles/aesthetic-usability-effect/#:~:text=Definition%3A%20The%20aesthetic%2Dusability%20effect,actually%20more%20effective%20or%20efficient, 2017.

[13] D. Norman. *The design of everyday things*. Hachette Book Group USA, 2013.

[14] Eclipse Foundation AISBL. Eclipse mosquitto™ an open source mqtt broker. https://mosquitto.org/, 2024.

[15] J. Hartmanis G. Goos and J. van Leeuwen. *Extreme Programming and Agile Methods - XP/Agile Universe 2003*. Springer, 2003.

[16] Eric Warnquist. Manager product developer shared service, assa abloy entrance systems.

[17] Pimorino. Hyperpixel 4.0 square - hi-res display for raspberry pi – touch. https://shop.pimoroni.com/products/hyperpixel-4-square?variant=30138251444307, 2024.

[18] Ian Craggs. Eclipse paho c client library for the mqtt protocol. https://github.com/eclipse/paho.mqtt.c/tree/master, 2024.

[19] nikosandreou. Glitches in gui applications. https://github.com/microsoft/wslg/issues/1148, 2023.

[20] K.Beck. Embracing change with extreme programming. *IEEE Xplore*, 0(1):70–77, 1999.

[21] S. Olsmats E. Liljenberg. Ux analysis and ui design of operating mode selector for sliding doors. 2024.

[22] Nazmul Ahsan. How to organize multi-frame tkinter application with mvc pattern. https://nazmul-ahsan.medium.com/how-to-organize-multi-frame-tkinter-application-with-mvc-pattern-79247efbb02b, 2023.

[23] Roger Light. paho-mqtt 2.1.0. https://pypi.org/project/paho-mqtt/, 2024.

[24] Meilhaus Electronic GmbH. Redcom usb-comi interface converter usb to rs422/rs485. https://www.meilhaus.de/en/usb-comi.htm?b2b=0.

[25] N. P. Smart B. Warinschi P. Morrissey. A modular security analysis of the tls handshake protocol. *LNSC,volume 5350*, 5350(1):70–end, 2008.

[26] Hasanain Shuja. Raspberry pi 4 model b. https://grabcad.com/library/raspberry-pi-4-model-b-1, 2019.

[27] Philippe Cadic. Hyperpixel 4.0 square. https://grabcad.com/library/hyperpixel-4-0-square-1, 2021.

[28] Mathias Navne. Software system design lead, assa abloy entrance systems.

7 Appendix A - Gantt charts

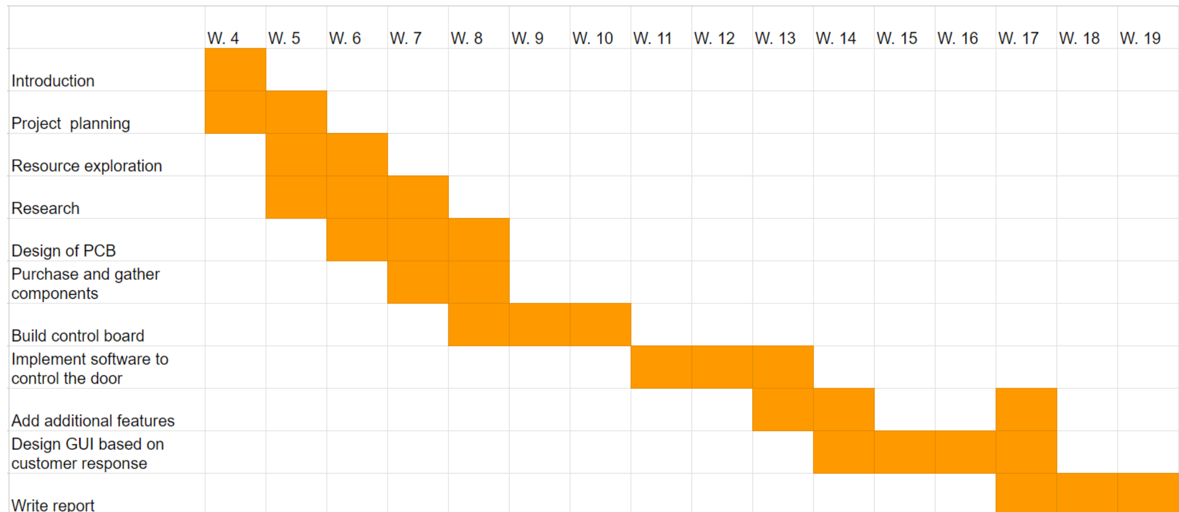


Figure 30: Initial Gantt chart

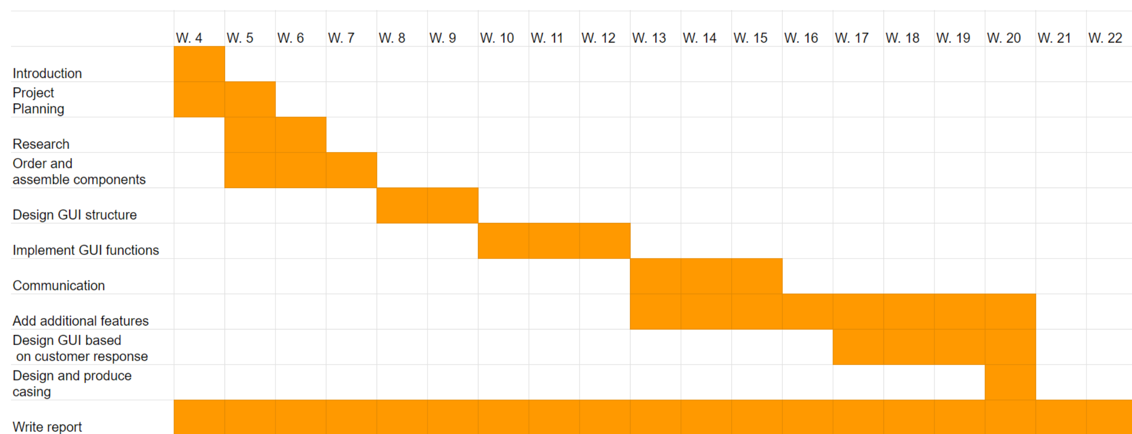


Figure 31: Revised and actual Gantt chart

8 Appendix B - source code for project

The source code for the GUI can be found in the following GUI HUB repository. The program for the communication is not included since it is the property of Assa Abloy and contains confidential information. The details about the errors are also kept undisclosed.

https://github.com/HCSoderberg/GUI-MMI-Sliding-Door

9 Appendix C

9.1 Guide to use RPi and display to create a rapid prototype

This is a brief guide on how to make similar projects using the components and GUI structure used in this report for similar projects.

Firstly the used components need to be ordered, this includes a Raspberry Pi 4 model B and a Hyperpixel 4 square touch. Additionally an SD card, a HDMI-micro to HDMI cable and a power supply for the Raspberry Pi is needed. Once these components are received the following steps can be followed to achieve the same setup as used in this project.

- Insert the SD card in a computer and install the Raspbian GNU/Linux 12 operating system using raspberry Pi Imager onto the SD card. Once this is done the SD card can be inserted into the SD card slot on the Raspberry Pi. The device can now be started by connecting the power supply.
- Before connecting the Hyperpixel display the device must be setup for it. This is done using an external display that can be connected via the HDMI micro port on the board of the RPi. Once the display is connected, open a terminal on the Raspberry Pi and run the following command: *curl https://get.pimoroni.com/hyperpixel4 — bash*. This is a one-line installer constructed by the manufacturers of the display, Pimorino, which performs the required setup for the display to work with the RPi.
- The external display can now be disconnected and the Hyperpixel display mounted on the board of the RPi. Start by screwing on the four included legs on to the back of the display and mount the GPIO pin extender. Once this is done the display can be attached to the board on the RPi on the GPIO pins. Finally the displayed is screwed in place using screws from the back of the board.
- The device is now set up to start programming the desired GUI which can be done either on the RPi itself or on your on computer and then transferring the final application to the RPi. Using the structure of the application used in this project, found under Appendix B, a GUI can easily be created by modifying our program to the desired layout and functionalities.