CZECH
TECHNICAL
UNIVERSITY
IN
PRAGUE

FACULTY
OF ELECTRICAL ENGINEERING
Combining Electrotechnics and Informatics

Development of an integration in the open-source home automation platform Home Assistant of a motorized blind control through the microcontroller ESP32 using Bluetooth and the MQTT communication protocol

Author: Juan Carlos García Hernández

Mentor: Ing. Vladimír Janíček

Abstract

The project consists of the implementation of a motorized blind control in the open-source home automation platform Home assistant of a BlindsDrive AM43 which originally is controlled by an Android application provided by the manufacturer. For this purpose, an ESP32 microcontroller will be used, the microcontroller is going to interpret the messages via MQTT that will be sent by the Home Assistant platform in order to send the Bluetooth commands to the AM43 motor.

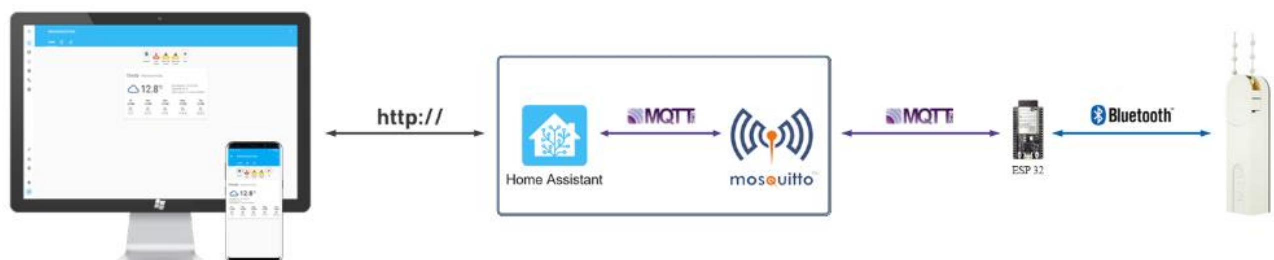
The task to be accomplished are:

- Use of reverse engineering to understand the behavior of the BlindsDrive AM43 through the interception and analysis of Bluetooth packets using Android's *Bluetooth HCI snoop log* tool and Wireshark software to read the packets.

- Programming of a *Custom Card* through Java Script for the integration in Home Assistant's user interface Lovelace UI.

- Modification of Home Assistant's *yaml* files to get the configuration that allows the communication between Home Assistant and the ESP32 microcontroller via MQTT.

- Installation of the firmware and programming the ESP32 microcontroller for being able to communicate via Bluetooth between ESP32-AM43 motor and via MQTT between ESP32-Hass.io linking MQTT messages to Bluetooth's commands.



Index

1. Introduction ... 4
2. Previous analysis ... 7
  2.1. AM43's Android App (Blind Engine) operation ... 7
  2.2. Reverse engineering ... 8
    2.2.1. Go up, go down and stop commands ... 10
    2.2.2. Percentage position movement ... 10
    2.2.3. Battery and position request ... 12
    2.2.4. Direction and velocity setting ... 13
    2.2.5. Upper and bottom limit setting ... 14
    2.2.6. Schedule setting ... 15
    2.2.7. Factory reset request ... 16
3. Implementation ... 18
  3.1. Home Assistant ... 18
    3.1.1. MQTT implementation ... 19
    3.1.2. Go up, go down and stop commands implementation ... 21
    3.1.3. Schedule setting implementation ... 24
    3.1.4. Percentage position movement implementation ... 29
    3.1.5. Battery and position request implementation ... 31
    3.1.6. Direction and velocity setting implementation ... 34
    3.1.7. Upper and bottom limit setting implementation ... 38
    3.1.8. Factory reset setting implementation ... 41
    3.1.9. Checking connection implementation ... 43
3.2. ESP-32 ... 47
    3.2.1. Wi-Fi Implementation ... 48
    3.2.2. MQTT Implementation ... 48
    3.2.3. BLE Implementation ... 50
4. Installation (HACS) ... 54
5. Conclusion ... 56
6. Bibliography ... 57
7. Appendix I: HTML5 code ... 58
  7.1. Controller mode ... 58
  7.2. Settings mode ... 62
  7.3. CSS styles ... 64
8. Appendix II: Battery life test ... 71

Figure index

Figure 1. Global Home Automation System Market evolution. Source: MRFR Analysis. 4
Figure 2. Specifications of the Am43 motor. Source: AM43 User’s manual 4
Figure 3. Detail of the ESP-32S microcontroller used in the project. Source: https://www.gme.cz 5
Figure 4. Evolution of the term MQTT (2004-2020). Source: Google Trends.................................... 6
Figure 5. Detail of the device list menu in the BlindEngine App 7
Figure 6. Scheme of the Bluetooth sniffing process 8
Figure 7. Detail of a packet in Wireshark for a Write Command (0x52) ATT method........................ 9
Figure 8. Detail of a packet in Wireshark for a Handle Value Notification (0x1b) ATT method 9
Figure 9. Detail of the “Execute basic movements” interface in the BlindEngine App 10
Figure 10. Detail of the percentage position movement in the BlindEngine App 10
Figure 11. Detail of the direction and velocity setting in the BlindEngine App.................................. 13
Figure 12. Detail of the upper and bottom limit setting in the BlindEngine App 14
Figure 13. Detail of the schedule setting in the BlindEngine App... 15
Figure 14. Detail of the factory reset setting in the BlindEngine App... 16
Figure 15. Detail of the control interface in the implemented custom card 21
Figure 16. Detail of the schedule setting’s interface in the implemented custom card......................... 24
Figure 17. input_text which has the information of the scheduele configuration................................ 24
Figure 18. Detail of the generation of the position code’s operation 30
Figure 19. Example of the MQTT sensor’s value when a notification is sent 32
Figure 20. Example of the notifications input_text’s value 32
Figure 21. Detail of the velocity setting’s interface in the implemented custom card.......................... 34
Figure 22. Example of the configuration input_text’s value.. 35
Figure 23. Detail of the upper limit’s interface in the implemented custom card................................ 38
Figure 24. Detail of the factory reset’s interface in the implemented custom card 41
Figure 25. Detail of the non conection’s interface in the implemented custom card............................ 43
Figure 26. Scheme of the ESP32 microcontroller’s operation... 47
Figure 27. Detail of the service (yellow) and characteristic (green) in the nRF Connect app. 50
Figure 28. Detail of a successful connection messages in the Arduino IDE Serial Monitor 53
Figure 29. Detail of the integration information in HACS 55

1. Introduction

Nowadays, technology has become an important part of people's lives and with the expansion of automation technology life has become easier in all aspects. In today's era, Automatic systems are being preferred over manual system. Home automation system is growing rapidly as it provides better quality of life for people.

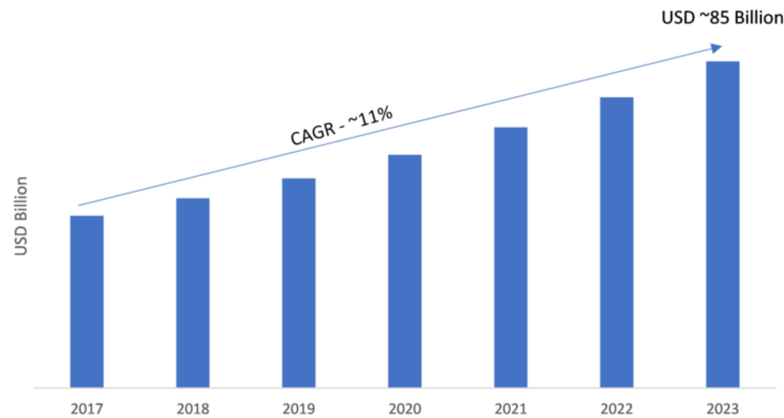
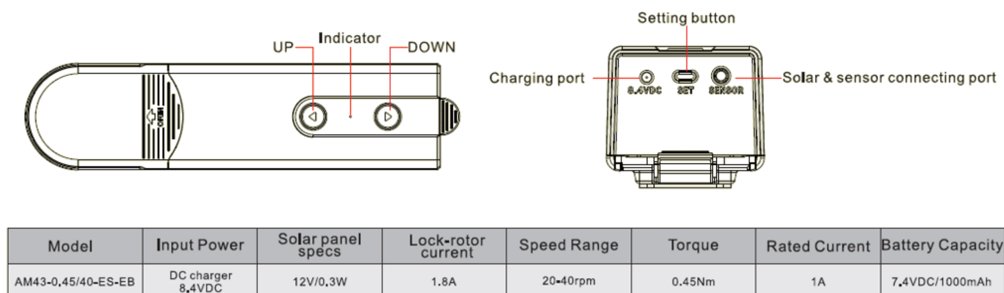


Figure 1. Global Home Automation System Market evolution. Source: MRFR Analysis.

In the trend of the *Figure 1*, it is possible to see the evolution of the Global Home Automation System Market in the forecasted period till 2023. The companies operating in the smart home automation market include Johnson Controls, ABB Ltd., Ingersoll-Rand plc, Schneider Electric, Legrand SA, Crestron Electronics, Inc., Siemens AG and Honeywell International, Inc.

In some cases, a lot of home automation applications that can be found, work independently either with their own applications or by configuring these via hardware. These cases do not fit with the actual technology philosophy which promotes that everything must be connected (e.g., IoT).

The case mentioned in the previous paragraph is reflected in the AM43 motor developed by the company A-OK motors, which specifications can be seen in the *Figure 2*.



Model	Input Power	Solar panel specs	Lock-rotor current	Speed Range	Torque	Rated Current	Battery Capacity
AM43-0.45/40-ES-EB	DC charger 8.4VDC	12V/0.3W	1.8A	20-40rpm	0.45Nm	1A	7.4VDC/1000mAh

Figure 2. Specifications of the Am43 motor. Source: AM43 User's manual

The AM43 motor uses the Android application *Blind Engine* which offers to the users the capability to control the motor. The communication between the phone and the motor is made via BLE (Bluetooth Low Energy). This feature is important because is determinant in the selection of the future solution. All these features and the communication will be explained in chapter *2. Previous Analysis*.

As mentioned before, the new tendency of the actual technologies is trying to unify all the separate devices and connect them in some way. Currently, a lot of software in the field of home automation are aiming to reach these new goals. Some of this software are: OpenHab, Home Assistant, Domoticz, etc.

In this document, the process of the functionality integration of the AM43 motor into one of the software mentioned before (**Home Assistant**) will be explained. But how does Home Assistant work?

The definition which appears in the Home Assistant's website[1] is provided below:

"Open source home automation that puts local control and privacy first. Powered by a worldwide community of tinkerers and DIY enthusiasts. Perfect to run on a Raspberry Pi or a local server"

In the definition it can be read that Home Assistant can be run on a Raspberry Pi or a local server. This is important because the new integration needs to run in any platform. Therefore, it makes no sense trying to search for a specific solution for only one kind of device (e.g., Raspberry Pi), which has BLE integrated, and could not be able to run the in a Home Assistant installed on Windows 10 system which maybe does not have BLE. To avoid this problem, it was decided to include an additional hardware to take care of the BLE communication. As a result, the integration can be installed in any sort of server. It will work even if the hardware of the server does not have BLE integrated.

The hardware that was chosen for this purpose was the microcontroller ESP-32. This microcontroller is very common in IoT applications and is one of the cheapest ones. The microcontroller is represented in the *Figure 3.* It has very interesting features (32XGPIO, 3xUART, 3xSPI, etc.), but the only features that are going to be used in this project are the **BLE (Bluetooth Low Energy)** and the **Wi-Fi: 802.11 b/g/n.**

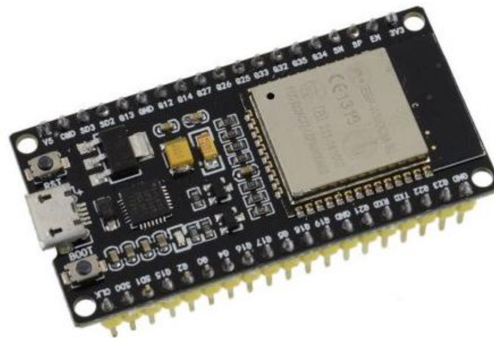


Figure 3. Detail of the ESP-32S microcontroller used in the project. Source: https://www.gme.cz

[1] https://www.home-assistant.io

Once the decision of the BLE communication is made, the next step was to decide about how Home Assistant was going to communicate with the ESP32 microcontroller. There was a protocol which was considered from the beginning because it is one of the most famous in the IoT systems. This protocol is the **MQTT** (Message Queuing Telemetry Transport).

The *Figure 4* depicts the evolution of the term MQTT in Google. So, there is no doubt that this protocol has the potential to become one of most used protocols in the IoT sectors; that was the reason about choosing it.

Figure 4. Evolution of the term MQTT (2004-2020). Source: Google Trends

Once the whole background is explained, the process followed to achieve the integration is going to be presented. This process can be divided in two main sections. In the first section, everything related to the behavior of the *Blind Engine* application and the communications will be explained. In the second one, it will be explained how the behavior studied in the first section is implemented into the elements (Home Assistant and ESP32 microcontroller) in order to get a integration which works in the same way as the AM43 motor with the *Blind Engine* application.

2. Previous analysis

As mentioned in the beginning of this project, the AM43 motor has an android application called *Blind Engine*. In this chapter, the application is going to be studied in order to understand, firstly, the basic operation of this application, and secondly, the process of obtaining the deeper operation related to the communication (BLE).

2.1. AM43's Android App (Blind Engine) operation

In the followings sections, all the functionalities of the *Blind Engine* application will be mentioned and analyzed, so, is it necessary to make a first contact with the application before and get an idea of how the app operates and what can be controlled with it.

Once the application is launched, the menu of the following image (*Figure 5*) appears.

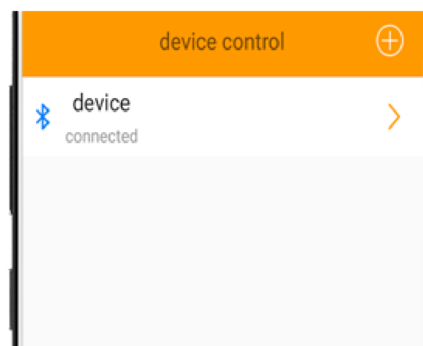


Figure 5. Detail of the device list menu in the BlindEngine App

In this menu, appears the list of devices that are in the BLE radio. When the device to be connected is clicked, a password is requested, but this security protection is only implemented in the *.apk*, so, there will not be any problems in the future integration.

Once the connection is established, the user can control the motor. In the following list appears the descriptions of the actions that can be controlled:

- **Execute basic movements**: Simple buttons to send movements requests e.g. move the blind up, move the blind down or stop the movement.
- **Set the position**: Slide button for controlling the exact position by moving it to the top and the bottom of the virtual blind which appears in the app.
- **Modify direction**: Switch button that modifies the direction of the motor's rotation.
- **Modify velocity**: Buttons that allow the user to change the motor's velocity in rpm. The range is between 20 rpm and 50 rpm.

- **Establish upper limit**: Simple buttons to move the blind, including a button in order to save the upper limit of the blind.

- **Establish bottom limit**: Simple buttons to move the blind, including a button in order to save the bottom limit of the blind.

- **Set schedules**: Combinations of buttons that allow the user to include some schedules to bring the blind to a position at a specific time, including the capability to duplicate the movements in more than one day per week.

- **Make a factory reset**: Button that activates the factory reset. This action will erase all the configuration stored in the device and will upload the original factory configuration.

In the next section, all of these functionalities will be analyzed in order to understand how the application translates these actions into commands that will be sent to the motor via BLE.

2.2. Reverse engineering

"The process of studying another company's product to see how it is made, sometimes in order to be able to copy it." Cambridge dictionary (2020).

In this case, the reverse engineering will be used to understand the communication between the Android app and the motor. Once the behavior of the communication is known, it will be possible to implement that in our integration and be able to control the motor in the same way as the android app does.

The following picture (*Figure 6)* presents a scheme of the process that was followed to realize the transmission.

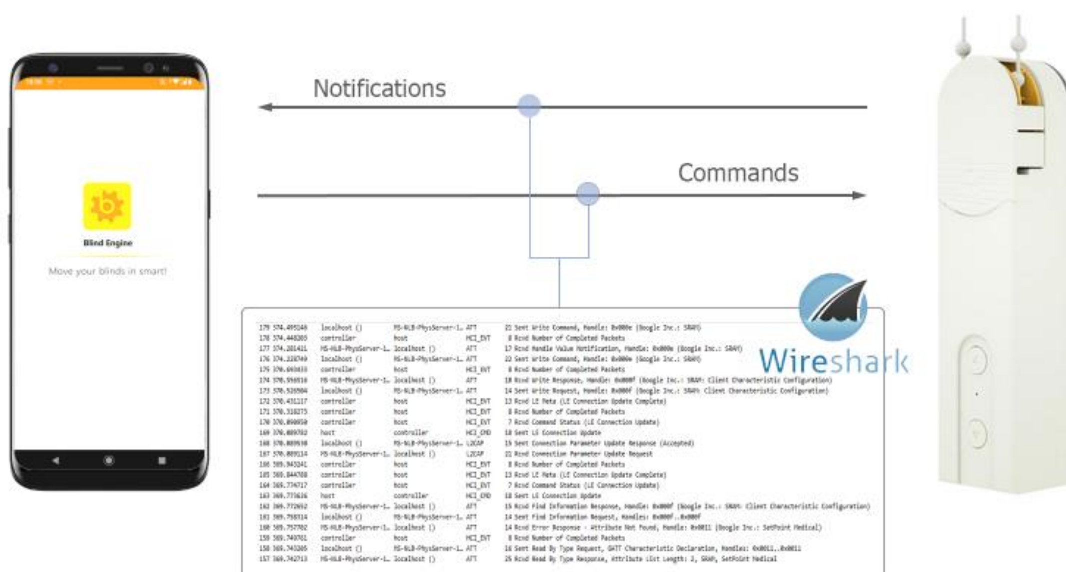


Figure 6. Scheme of the Bluetooth sniffing process

On the left side appears a generic android phone in which the *Blind Engine* application is running, if the developer option “Bluetooth HCI snoop log” is enabled, a file called *btsnoop_hci.log* is generated in the internal memory of the phone.

Once the file is generated, all the information about the packets that are being sent is stored in this file. The idea is to send all the actions to the motor in order to be able to decode the information that will be stored in the file. For this purpose, the software *Wireshark* was used.

If the *btsnoop_hci.log* is opened, it can be found two types of data which is going to be quite important in the integration. This data comes from two different methods of the ATT protocol; these methods are:

- Write Command (0x52)

```
No.   Time        Source        Destination          Protocol  Length  Info
  192 379.143304  localhost ()  MS-NLB-PhysServer-1… ATT       21      Sent Write Command, Handle: 0x000e (Google Inc.: SRAM)

> Frame 192: 21 bytes on wire (168 bits), 21 bytes captured (168 bits)
> Bluetooth
> Bluetooth HCI H4
> Bluetooth HCI ACL Packet
> Bluetooth L2CAP Protocol
v Bluetooth Attribute Protocol
  > Opcode: Write Command (0x52)
  > Handle: 0x000e (Google Inc.: SRAM)
    Value: 00ff00009a0a01dd4c
```

Figure 7. Detail of a packet in Wireshark for a Write Command (0x52) ATT method

This type of data shown in the *Figure 7*, appears when one of the actions of the *Blind Engine* app described in the previous subchapter is executed.

- Handle Value Notification (0x1b)

```
No.   Time        Source                Destination   Protocol  Length  Info
  193 379.181016  MS-NLB-PhysServer-1…  localhost ()  ATT       17      Rcvd Handle Value Notification, Handle: 0x000e (Google Inc.: SRAM)

> Frame 193: 17 bytes on wire (136 bits), 17 bytes captured (136 bits)
> Bluetooth
> Bluetooth HCI H4
> Bluetooth HCI ACL Packet
> Bluetooth L2CAP Protocol
v Bluetooth Attribute Protocol
  > Opcode: Handle Value Notification (0x1b)
  > Handle: 0x000e (Google Inc.: SRAM)
    Value: 9a0a015a31
```

Figure 8. Detail of a packet in Wireshark for a Handle Value Notification (0x1b) ATT method

This type of data shown in the *Figure 8* is received after of the Write Command (0x52) type. In fact, if the field of *Time* is analyzed, it can be seen that the Write Command appears at 379.143304 s, and the handle Value Notification appears at 379.181016 s. Usually, this data represents the feedback of the motor.

In the next sections, the process of decoding each packet will be explained in detail for each group of actions, including some variable notifications like the battery or the position request of the motor.

2.2.1. Go up, go down and stop commands

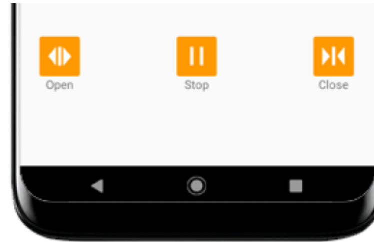


Figure 9. Detail of the "Execute basic movements" interface in the BlindEngine App

The commands that are going to be presented in this section are probably the commands which present less complexity. This is due to the fact that their content does not present any variation. The reason is that the requested actions are simple basic movements.

In order to get these codes, the buttons that are sown in the *Figure 9* were pressed in a specific order to be able to identify them in the Wireshark application as mentioned before.

In the next table appears a summary of the commands and their respective notifications. As mentioned before, these codes lack complexity, in fact, the notification is the same for all of them. This notification represents that the motor received the movement's request.

Action	Command	AM43 Notification
Go up	0x00FF00009A0A01DD4C	0x9A0A015A31
Go down	0x00FF00009A0A01EE7F	0x9A0A015A31
Stop	0x00FF00009A0A01CC5D	0x9A0A015A31

2.2.2. Percentage position movement

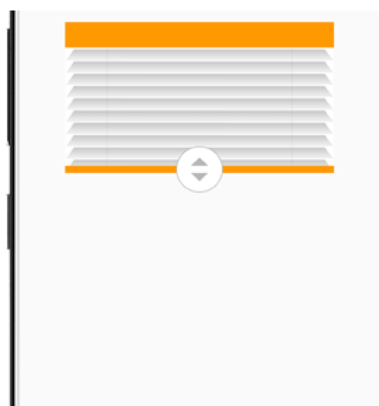


Figure 10. Detail of the percentage position movement in the BlindEngine App

The commands presented in this section are quite different compared to the ones presented in the previous section. In this case, the commands include variations depending on the configuration that is to be set.

As it happened with the previous set of commands, an organized test was made in order to be able to identify the codes with the *Wireshark* software. The way to send a new position command is by scrolling the button that has two arrows, this button can be seen in the *Figure 10.* The best procedure was to start with one limit (e.g., 0%) and increase it until it reaches the maximum limit (100%).

Once a few codes were obtained, it was possible to get the code's logic behavior. The next box shows the codes to set the blind in a range between 0% to 47%. There are 53 codes more but this amount is sufficient to understand the logic behind.

```
0x00FF00009A0D012FB9 (47%)    0x00FF00009A0D011F89 (31%)    0x00FF00009A0D010F99 (15%)
0x00FF00009A0D012EB8 (46%)    0x00FF00009A0D011E88 (30%)    0x00FF00009A0D010E98 (14%)
0x00FF00009A0D012DBB (45%)    0x00FF00009A0D011D8B (29%)    0x00FF00009A0D010D9B (13%)
0x00FF00009A0D012CBA (44%)    0x00FF00009A0D011C8A (28%)    0x00FF00009A0D010C9A (12%)
0x00FF00009A0D012BBD (43%)    0x00FF00009A0D011B8D (27%)    0x00FF00009A0D010B9D (11%)
0x00FF00009A0D012ABC (42%)    0x00FF00009A0D011A8C (26%)    0x00FF00009A0D010A9C (10%)
0x00FF00009A0D0129BF (41%)    0x00FF00009A0D01198F (25%)    0x00FF00009A0D01099F (09%)
0x00FF00009A0D0128BE (40%)    0x00FF00009A0D01188E (24%)    0x00FF00009A0D01089E (08%)
0x00FF00009A0D0127B1 (39%)    0x00FF00009A0D011781 (23%)    0x00FF00009A0D010791 (07%)
0x00FF00009A0D0126B0 (38%)    0x00FF00009A0D011680 (22%)    0x00FF00009A0D010690 (06%)
0x00FF00009A0D0125B3 (37%)    0x00FF00009A0D011583 (21%)    0x00FF00009A0D010593 (05%)
0x00FF00009A0D0124B2 (36%)    0x00FF00009A0D011482 (20%)    0x00FF00009A0D010492 (04%)
0x00FF00009A0D0123B5 (35%)    0x00FF00009A0D011385 (19%)    0x00FF00009A0D010395 (03%)
0x00FF00009A0D0122B4 (34%)    0x00FF00009A0D011284 (18%)    0x00FF00009A0D010294 (02%)
0x00FF00009A0D0121B7 (33%)    0x00FF00009A0D011187 (17%)    0x00FF00009A0D010197 (01%)
0x00FF00009A0D0120B6 (32%)    0x00FF00009A0D011086 (16%)    0x00FF00009A0D010096 (00%)
```

In the codes it can be seen how the position 00% is linked with the hexadecimal number 0x96 (blue), then the pattern followed is to add +1 and subtract -3 until all the possible combinations starting with 0x9 are fulfilled. Then, the position 16% starts with the hexadecimal number 0x86 and the pattern explained before will be applied again till all the possible combinations starting with 0x8 are fulfilled. After this, it was figured out that the next combination of codes corresponding with the position 32% started with the hexadecimal number B, so there is a pattern followed by the 16 codes group that consist of subtracting -1 and adding +3 to the first four bits of the last byte.

It is quite sure that there is more than one way to understand the pattern followed by the codes, but this way is easy to implement in the program languages that are going to be used in this project.

In the next table appears a summary of the commands and their respective notifications. It can be seen that the notifications are also simple, and they do not apport any additional information, just if the notification showed in the table is sent by the motor, that means that the command has been sent successfully.

Action	Command	AM43 Notification
Move to % position	0x00FF00009A0D01**PPCC**	0x9A0D015A31
PP: Hex byte value of the position's percentage. e.g. 22 = 34% **CC:** Hex byte code different per each position. Explained in this section.		

2.2.3. Battery and position request

In the previous sections, the commands were the codes that presented some variations. Once the variations were known, it was possible to manipulate the AM43 motor. However, the commands that are going to be presented in this section are different. In this case, the commands have a constant behavior and the variations are going to happen in the AM43 notifications.

In this project , it is only important to know the battery and the position of the AM43 motor, for knowing this information it was followed some steps that are going to be described below.

Firstly, due to the fact that these kinds of commands are usually launched randomly or after some actions, it was necessary to identify the commands in the data captured in Wireshark that were unexpected, meaning the commands that did not match with the actions explained before.

Secondly, after identifying these commands, there were two ways to find matches in the notifications. Either to know the actual value of the battery and position to find a direct match with the notification or watch if some bytes of the notification were changing and if that change had some connection with the expected behavior.

On the one hand, in the case of the battery, it was easy because the value of the battery was found in one of the notifications. When the battery changed, this notification was checked again and the byte that was supposed to be the battery had the new expected value.

On the other hand, in the case of the position, it was more difficult due to the fact that the command, in this case, generates three notifications. Finally, it was possible to identify the byte that contains the value of the position because it was moved to the limits (00-01) and (99-100), so it was only necessary to search the hexadecimal values (0x00-0x01) and (0x63-0x64) in the notifications.

There was more information in the notifications, but it was not necessary knowing it in order to achieve the objectives in this project. In addition, there is no problem of not knowing the complete notification's content, in contrast with the commands, which have to be completely correct to send an accepted request.

In the next table appears a summary of the commands and their respective notifications.

Action	Command	AM43 Notification
Battery request	0x00FF00009AA2010138	0x9AA20500000000**BBXX**
Actual position request	0x00FF00009AA701013D	0x9AA707**XXXXPP**07D00D10**XX**
BB: Hex byte value of the battery percentage left. **PP**: Hex byte value of the actual position's percentage. **XX**: Unknown Hex bytes.		

2.2.4. Direction and velocity setting

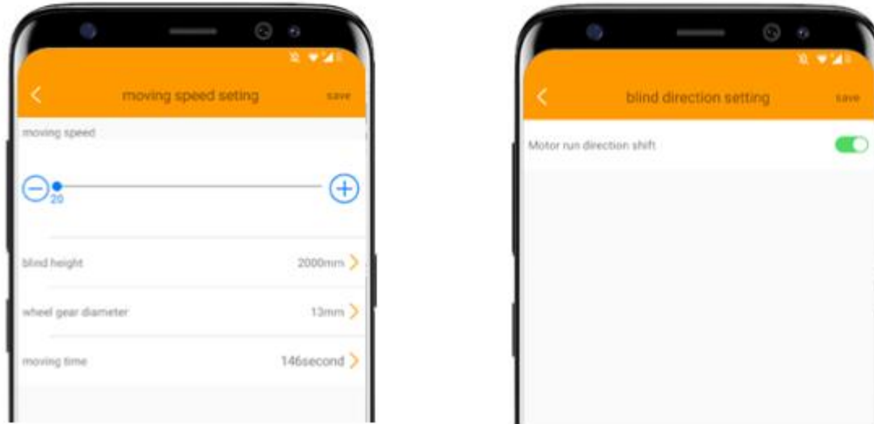


Figure 11. Detail of the direction and velocity setting in the BlindEngine App

The commands presented in this section have a similar behavior than the presented in the section *2.2.2.Percentage position movement* because they have a variable comportment. To obtain the codes, the velocity and the direction were modified using the interface that the *Blind Engine* app offers which is shown in the *Figure 11*.

At the beginning, it was thought that the configuration of the direction and the velocity had different commands, but it was a wrong conclusion. It was figured out that the codes used were quite similar and the only things that were changing were: one byte for the direction, one byte for the velocity and one byte that was changing in a different way depending on the direction.

In contrast with the percentage position codes, in which it was difficult to obtain the complete range (0-100) of codes because the interface that the apps offered (slide button), it was quite easy to obtain the full range of codes (simple buttons). All the codes are shown in the following box:

```
00FF00009A110616140007D00D55 20    00FF00009A110614140007D00D57 20
00FF00009A110616150007D00D54 21    00FF00009A110614150007D00D56 21
00FF00009A110616160007D00D57 22    00FF00009A110614160007D00D55 22
00FF00009A110616170007D00D56 23    00FF00009A110614170007D00D54 23
00FF00009A110616180007D00D59 24    00FF00009A110614180007D00D5B 24
00FF00009A110616190007D00D58 25    00FF00009A110614190007D00D5A 25
00FF00009A1106161A0007D00D5B 26    00FF00009A1106141A0007D00D59 26
00FF00009A1106161B0007D00D5A 27    00FF00009A1106141B0007D00D58 27
00FF00009A1106161C0007D00D5D 28    00FF00009A1106141C0007D00D5F 28
00FF00009A1106161D0007D00D5C 29    00FF00009A1106141D0007D00D5E 29
00FF00009A1106161E0007D00D5F 30    00FF00009A1106141E0007D00D5D 30
00FF00009A1106161F0007D00D5E 31    00FF00009A1106141F0007D00D5C 31
00FF00009A110616200007D00D61 32    00FF00009A110614200007D00D63 32
00FF00009A110616210007D00D60 33    00FF00009A110614210007D00D62 33
00FF00009A110616220007D00D63 34    00FF00009A110614220007D00D61 34
00FF00009A110616230007D00D62 35    00FF00009A110614230007D00D60 35
00FF00009A110616240007D00D65 36    00FF00009A110614240007D00D67 36
00FF00009A110616250007D00D64 37    00FF00009A110614250007D00D66 37
00FF00009A110616260007D00D67 38    00FF00009A110614260007D00D65 38
00FF00009A110616270007D00D66 39    00FF00009A110614270007D00D64 39
00FF00009A110616280007D00D69 40    00FF00009A110614280007D00D6B 40
00FF00009A110616290007D00D68 41    00FF00009A110614290007D00D6A 41
00FF00009A1106162A0007D00D6B 42    00FF00009A1106142A0007D00D69 42
00FF00009A1106162B0007D00D6A 43    00FF00009A1106142B0007D00D68 43
00FF00009A1106162C0007D00D6D 44    00FF00009A1106142C0007D00D6F 44
00FF00009A1106162D0007D00D6C 45    00FF00009A1106142D0007D00D6E 45
00FF00009A1106162E0007D00D6F 46    00FF00009A1106142E0007D00D6D 46
00FF00009A1106162F0007D00D6E 47    00FF00009A1106142F0007D00D6C 47
00FF00009A110616300007D00D71 48    00FF00009A110614300007D00D73 48
00FF00009A110616310007D00D70 49    00FF00009A110614310007D00D72 49
00FF00009A110616320007D00D73 50    00FF00009A110614320007D00D71 50
```

In the codes it can be noticed, firstly, the information about the direction (black byte) that the motor has: 0x14 for forward direction and 0x16 for reverse direction. Secondly, the velocity that is actually configured in the motor (red byte) which range is between 20 rpm (0x14) and 50 rpm (0x32). Finally, the code that is used (blue byte) follows a different pattern depending on the direction that is chosen.

On the one hand, if the forward direction is selected, the hexadecimal byte in which the code starts is 0x57, linked to the first value of the velocity (0x14). The pattern followed in this case is subtracting the value -1 to the byte for the next three velocities and adding +7 to the byte in the fourth one. This pattern is going to be followed until the last velocity (0x32) is reached.

On the other hand, if the reverse direction is selected, the hexadecimal byte in this case is 0x55 in the first value of velocity (0x14). The pattern followed this time is subtracting -1 and adding +3 in each increment of velocity until the last value (0x32) is reached.

In the following table appears a summary of the commands explained in this section and their respective notifications, it can be seen how the notifications have a constant value again.

Action	Command	AM43 Notification
Direction and velocity setting	`0x00FF00009A1106`**`DDSS`**`0007D00D`**`CC`**	`0x9A11015A31`

```
DD: Hex byte value of the direction (14: Forward, 16: Reverse).
SS: Hex byte value of the speed. (14 to 32 )
CC: Hex byte code different per each configuration. Explained in this section.
```

2.2.5. Upper and bottom limit setting

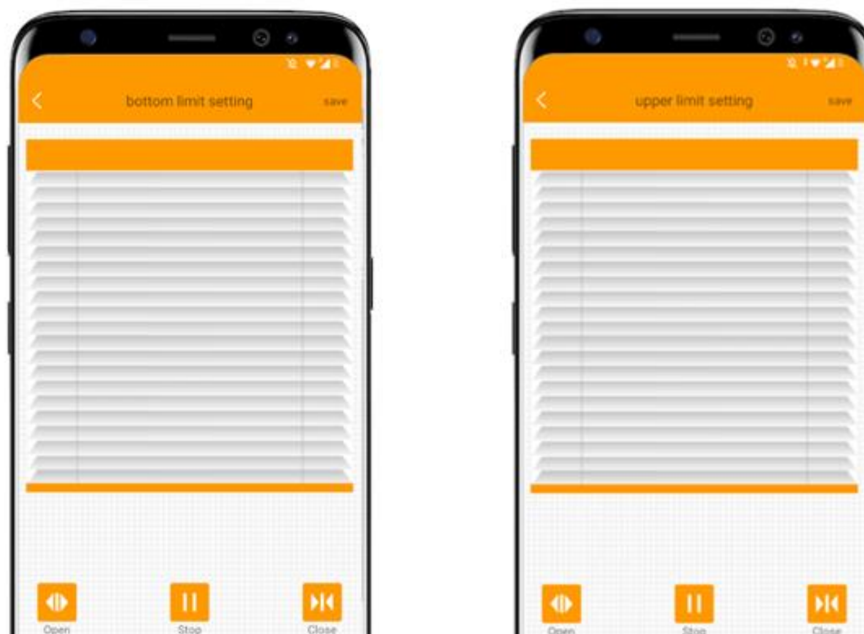


Figure 12. Detail of the upper and bottom limit setting in the BlindEngine App

The commands presented in this section have similar characteristics with the commands which were presented at the beginning of this chapter: *2.2.1. Go up, go down and stop commands*. The reason is that the commands have a constant behavior and the requests produce constant notifications too.

In the *Figure 12*, the interface responsible of sending the requests of creating a new upper or bottom is shown.

It is important to comment that the buttons of going up, going down and stop send the same commands as in the main interface.

A summary of the commands that were obtained is shown in the table below.

Action	Command	AM43 Notification
Set bottom limit	0x00FF00009A2203000200B9	0x9A22015A31
Set upper limit	0x00FF00009A2203000100BA	
Save bottom limit	0x00FF00009A220320020099	0x9A22015B31
Save upper limit	0x00FF00009A22032001009A	
Cancel limit setting	0x00FF00009A2203400100FA	0x9A22015C31

2.2.6. Schedule setting

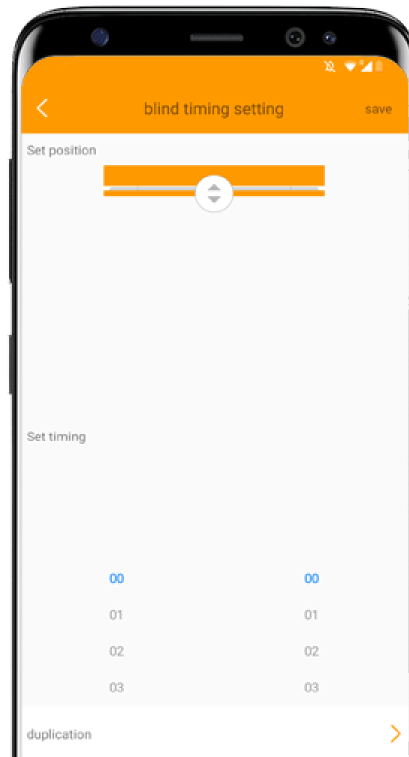


Figure 13. Detail of the schedule setting in the BlindEngine App

In this section, the commands for setting a schedule are going to be exposed, probably, this is best functionality that the motor can give to the user.

The codes were generated by the manipulation of the group of buttons and scrolls that can be seen in the *Figure 13.* It can be noticed that the difficulty of decoding these codes was more difficult than the other ones due to the complexity of the interface that the *Blind Engine* app has for this purpose.

During the manipulation in the *Blind Engine* app, a lot of unknown problems happened when the configuration was set and the codes that were obtained did not have a simple behavior. Therefore, it was necessary to invest more time in order to understand the entire behavior.

Due to these problems, the question was if it was worth it to obtain the whole behavior. The answer was definitely not, because of the fact that if the basic actions were learnt (basic movements or even specific position request) why not implement a combination of those in order to achieve the schedule behaviour. This way was followed, and it will be explained in the chapter *3.Implementation.*

In the following table appears the commands that were obtained during the tests in *Blind Engine* app. It can be seen that even the bytes that are known present more difficulty than the commands explained before.

Action	Command	AM43 Notification
Create slot 1 schedule	`0x00FF00009A15070100000002`**`HHMM`**`94`	`0x9A15015A31`
Activate slot 1 schedule	`0x00FF00009A150701000100000000088`	
Deactivate slot 1 schedule	`0x00FF00009A150701000000000000089`	
Delete slot 1 schedule	`0x00FF00009A150701010000000000088`	
HH: Hex byte value of the day’s hour + 1. **MM:** Hex byte value of the day’s minute.		

2.2.7. Factory reset request

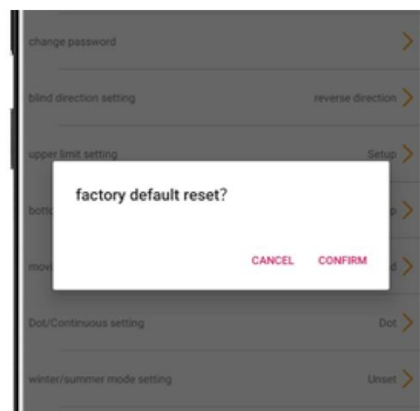


Figure 14. Detail of the factory reset setting in the BlindEngine App

In the *Figure 14* appears the popup message that appears when the factory reset option is pressed. It can be seen that this command is the easiest one and was quite easy to identify it in Wireshark.

In the implementation section it will be explained the effects that this factory reset applies into the motor. This is of high importance because it has to be known in order to represent the correct values in the graphic interface.

In the following table appears the code and the notification that is sent by the device when the factory reset is requested.

Action	Command	AM43 Notification
Factory reset	0x00FF00009A2203000001BA	0x9A22015A31

3. Implementation

Once the operation of the *Blind Engine* app is explained, the next step is to explain the implementation into the new system described in the chapter *1. Introduction.* Even though the implementation was made in parallel, the first step will be to explain the Home Assistant integration. This is going to have the user interface to interact with, and all the mechanism to be able to send the MQTT messages to the ESP32. The second step will be to explain the ESP32 implementation which is in charge of converting the MQTT messages into the BLE bytes array.

3.1. Home Assistant

Before explaining each implemented functionality in detail, it is necessary to introduce some concepts to be able to understand the whole background. These concepts are:

- *configuration.yaml:* Main configuration file that is used to define aspects such a parameter of communications protocols, define entities (sensors, lights, switch).

- *automation.yaml:* File in which the automations are defined. An automation is defined by a trigger (defined by the status change of an entities) which runs the actions of the automation in order to execute some Home Assistant services.

- *Lovelace UI:* Home Assistant's user interface, can work in mobile and desktop. The philosophy of this interface is using built-in cards which have predefined configuration for common uses such as lights, sensors, thermostat, etc.

- *Web component:* Group of different technologies that allow the user to create reusable custom elements, which have their encapsulated functionality separate from the rest of the code and use them in web applications.

- *Custom card:* Concept of cards that can be implemented in Home Assistant[2] that are based on web components. This concept gives complete freedom to the community in order to create cards that can be used for integrating new applications in Home Assistant.

- *blind-control.js:* Name of the file which contains the code used to create the web element to achieve the implementation of this project. It is programmed in JavaScript language with the base class Lit Element[3].

[2] https://developers.home-assistant.io/docs/lovelace_custom_card/
[3] https://lit-element.polymer-project.org/guide

Once every one of the important concepts are defined, it is easier to understand how the implementation in Home Assistant is made. It is obvious that it is impossible to find one built-in card that can be able to control the AM43 motor, so it was necessary to choose between two alternatives that will be discussed below.

Before discussing the options, it has to be said that both of these options are a combination of the interaction between the *configuration.yaml* file, the *automation.yaml* file and the only thing that will be different is the Lovelace UI configuration.

The first option was combining built-in cards in order to get a group of cards which had been able to get the same operation with the *Blind Engine* app. The main advantage of this choice was that it was not necessary to program in other languages, everything was done just editing the *.yaml* files. The main disadvantage of this choice was that the graphic design was limited, and the idea was to achieve a similar interface to the android app. Thus, for having specific interfaces, like options menu, it could have been very complicated to achieve using this configuration.

The second option was creating a custom card in order to achieve an implementation similar to an embedded app which could operate in a similar way with the *Blind Engine* app. Among others, one advantage of this choice was that it was so powerful because there was not limitation about creating graphical content. The main disadvantage was that there was necessary to have a deep knowledge in JavaScript and Web Components to reach a good result. Finally, the second option was decided, and the result was quite interesting. In the next sections, the specific explanation about all the important aspects of the integration implementation will be explained.

3.1.1. MQTT implementation

The first aspect that is going to be explained is the MQTT implementation in Home Assistant. The reason about explaining this in the first place is because it was necessary to know if it was possible to send MQTT messages via the Web Component and receive the ESP32 messages. If these functionalities had not been achieved, the integration with the web component would not have been possible. On the one hand, referring to the **functionality of sending messages**, in the following code, appears the MQTT configuration in Home Assistant through the *configuration.yaml* file. Three configurations had to be defined in order to succeed with the communication. First of all, it was necessary to introduce the IP address of the MQTT server[4]. The other two kinds of entities that were determinant to achieve the goal were one *input_boolean*[5] entity and one *input_text*[6] entity.

```yaml
configuration.yaml
mqtt:
  broker: XXX.XXX.XXX.XXX
input_boolean:
  mqtt:
    name: "AM3_MQTT_SendCommand"
input_text:
  mqtt:
    name: "AM43_MQTT_Commands"
```

[4] https://www.home-assistant.io/docs/mqtt/broker/
[5] https://www.home-assistant.io/integrations/input_boolean/
[6] https://www.home-assistant.io/integrations/input_text/

The reason about those kinds of entities were created, was because the functionality of sending a MQTT message in the JavaScript code could not be achieved directly, so, it was necessary to do it using another method. There were two services which could be called:

The first one was the service *set_value* in which the value of the *input_text* entity could be modified and the second one is the service *toggle* in which the *input_boolean* entity could be switched.

In the next fragment of code, it can be seen the structure followed to send a MQTT message in the Web Component.

```
blind-control.js

this.hass.callService("input_text", "set_value", {
  value: message,
  entity_id: "input_text.mqtt",
});

this.hass.callService("homeassistant", "toggle", {
  entity_id: "input_boolean.mqtt",
});
```

The idea was, call the *set_value* service to modify the *input_text* called MQTT, which was defined in the configuration *yaml*, writing the value of the message to be sent. Once this entity is modified, the *input_boolean* MQTT can be switched. But what is the purpose of this logic?

The interesting thing about the *input_boolean* entities is that this can be switched internally by program as it is shown in the code below. This is quite important if there is a service that is difficult to launch in a code programmed in Java Script but can be implemented easily in the *automation.yaml*[7] file, doing a switching on an *input_boolean*.

```
automations.yaml

- alias: 'Send MQTT Command'
  trigger:
    - platform: state
      entity_id: input_boolean.mqtt
      to: 'on'
  action:
    - service: mqtt.publish
      data:
        topic: 'am43/commands'
        payload_template: "{{ states('input_text.mqtt') }}"
    - service: input_boolean.turn_off
      data:
        entity_id: input_boolean.mqtt
```

For example, in the previous code it can be seen that the actions of the automation designed for sending a MQTT message are going to be launched because the *input_boolean* is switched. Once the state

[7] https://www.home-assistant.io/docs/automation/

changes, the service *mqtt.publish* sends a message with the value of the *input_text* that was modified in the Java Script code, and once the message is sent, the value of the *input_boolean* is reset in order to be able to attend a new request.

In the *automation.yaml* it can be seen that the topic is constant (am43/commands), this has to be mentioned because if the user wants to install more AM43 devices, all the topics have to be duplicated and renamed (e.g., am43/kitchen/commands and am43/livingroom/commands). The same happens with the entities, for example, the news *input_text* entities would be *mqttkitchen* or *mqttlivingroom.*

On the other hand, referring to the **functionality of receiving messages**, it was easier due to the built-in feature that offers Home Assistant to deal with this objective. The feature is called *MQTT sensor*. The following box shows the code written in the *configuration.yaml.* It can be seen that only writing the topic is enough to know the value of the messages that are sent by the ESP32.

```
configuration.yaml
sensor:
  - platform: mqtt
    state_topic: "am43/notifications"
    expire_after: 1
```

The way used in the JavaScript code to read the value of the sensor is: *this.hass.states["sensor.mqtt_sensor"].state.* This will be used in some of the functions of the implementation that are going to be explained in the following sections.

3.1.2. Go up, go down and stop commands implementation

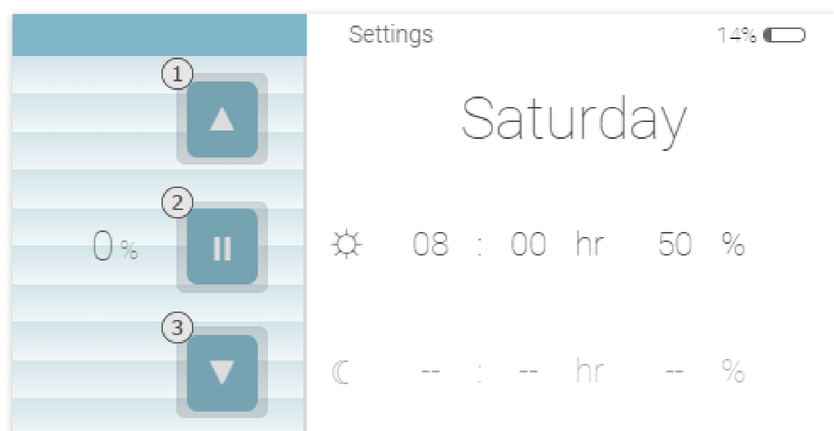


Figure 15. Detail of the control interface in the implemented custom card

At this point, the implementation of the user interface will be explained. In the *Figure 15* appears the control mode interface, the HTML5 code that generates the interface is included in the *Appendix I* of this project.

In the present section, the implementation of the execution basic movements will be explained. The *Figure 15* shows the three buttons (1: Go up, 2: Stop, 3:Go down) that the user can press to move or stop the blind. The JavaScript function that makes this possible is *_sendCommand(),* which is going to be explained below.

The methodology used to explain the JavaScript code is to explain it in parts, which will make the reader comprehend the code in a more comfortable way.

In the following code appears the conditions that have to be fulfilled in order to run the rest of the function. Some of the management of these variables will be explained at the end of this chapter, but is a good way to introduce what these variables mean.

```
blind-control.js | _sendCommand(message,type)

if (localStorage.getItem("status") == "1") {
  if (
    this.hass.states["input_text.busy"].state == "0" ||
    message == "00FF00009A0A01CC5D"
  ) { . . .
```

Basically, the meaning of the variables is: In the first instance, the local stored variable *status* indicates whether there is an established connection with the device. This is checked because it makes no sense trying to run one of these commands without connection. The program will crash if this is not checked. Secondly, the code will be run if a state of an entity called *busy*, which is an *input_text,* is on the 0 state. This is checked because the application can be run in multiple devices at the same time, so, it is necessary to establish a global variable in order to indicate to the other devices that an action is currently running, in order to not let them to initialize a new request. Lastly, the last "*OR"* checks if the message is *00FF00009A0A01CC5D,* which is the stop command's byte array and this is necessary because the code is wanted to run if the stop button is pressed, but not if it is busy and the *go up* or *go down* button are pressed.

The following code shows the process of sending a MQTT command:

```
blind-control.js | _sendCommand(message,type)

this.hass.callService("input_text", "set_value", {
   value: "1",
   entity_id: "input_text.busy",
 });

 this.hass.callService("input_text", "set_value", {
   value: message,
   entity_id: "input_text.mqtt",
 });

 this.hass.callService("homeassistant", "toggle", {
   entity_id: "input_boolean.mqtt",
 });
```

Now it is possible to understand the code because all the process of the MQTT implementation was explained in the previous section. The code only calls the automation that sends the MQTT message with the variable message written in the *input_text mqtt,* and this message variable is in the arguments of the function. In the *Appendix I,* where the HTML5 code appears, it can be seen how each button has the event listener: @click="${() => this._sendCommand("*message*", "*type*")}", consequently, each button sends the byte array (string) explained in the reverse engineering chapter to request the corresponding movement.

The other argument is the type of the command and was added because the movements are requested in the controller interface but are sent in the setting interface too in order to set the upper and the bottom limits. Due to this fact, it is necessary to include a variable in order to act in a different form. An example of this appears in the following code, which is the code that is run after sending the MQTT message:

```
blind-control.js | _sendCommand(message,type)
if (type == "c" || message == "00FF00009A0A01CC5D") {
  if (message == "00FF00009A0A01DD4C") {
    localStorage.setItem("goup", "1");
    localStorage.setItem("godown", "0");
    localStorage.setItem("stop", "0");
    setTimeout(() => {
      this._requestNotification("00FF00009AA701013D");
    }, 600);
  } else if (message == "00FF00009A0A01EE7F") {
    ...
  }
}
```

The previous code is applied when a command for moving the blind up or down is requested in the controller interface (*type: c*), basically, the information of the movement that is run have to be stored in order to use that information in other functions. Once the information of the movement is stored, the function *_requestNotification()* is launched. This action will be explained later because it is an implementation to know the virtual blind's position during the movement. However, if the stop button is pressed, the following code will be executed:

```
blind-control.js | _sendCommand(message,type)
if (message == "00FF00009A0A01CC5D") {
  localStorage.setItem("goup", "0");
  localStorage.setItem("godown", "0");
  localStorage.setItem("stop", "1");
  setTimeout(() => {
    this._requestNotification("00FF00009AA2010138");
  }, 400);
  setTimeout(() => {
    this.hass.callService("input_text", "set_value", {
      value: "0",
      entity_id: "input_text.busy",
    });
  }, 1000);
}
```

In this case, all the stored variables are set to 0 and the *stop* variable is set to 1. Once this is done, the entity busy will be set to 0 in order to attends new commands and the value of the battery will be requested through the *_requestNotification()* that will be explained in the section *Battery and position request implementation.*

3.1.3. Schedule setting implementation

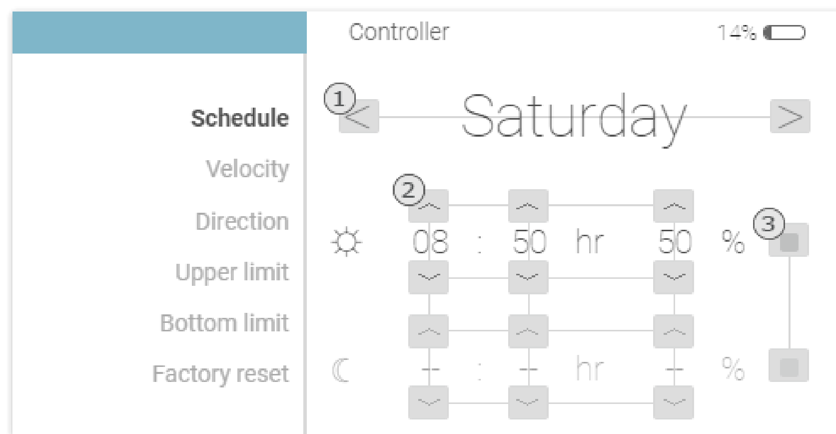


Figure 16. Detail of the schedule setting's interface in the implemented custom card

In this section, the implementation of the schedule setting is going to be explained. It was said in the previous chapter that the implementation of this feature was going to be different. Due to that fact, in this case, the commands that achieve this setting in the AM43 were not going to be followed and the alternative was making an integration with basic movement requests.

With the elements that the project has, the integration of the logic could have been done either via ESP32 or Home Assistant. The decision was doing it in Home Assistant because the server which will run Home Assistant is going to run 24/7 and it offers interesting features to reach the objective that are going to be explained below.

The first feature used is already known: *input_text.* In the explanation of the MQTT implementation this feature was presented, and it was explained how the value of this variable could be easily manipulated through the Web Component. In addition, the property that makes this *input_text* very powerful for this purpose is that the values are stored even if the server is restarted. This is the reason why this entity was assigned to store the currently schedule configuration.

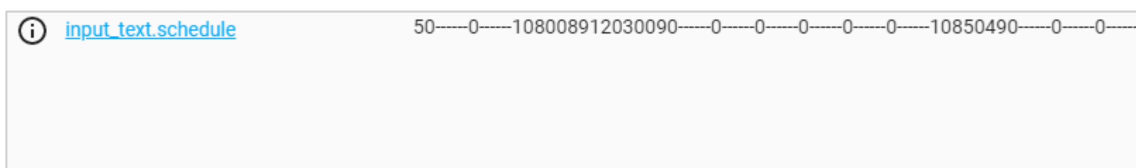


Figure 17. input_text which has the information of the scheduele configuration

The *Figure 17* shows the *input_text* used in this project. The name of the entity is *schedule*. The information that the word contains can be divided in two groups. Firstly, the first character represents the number of the day that is getting configured (0 (Monday) to 6 (Sunday)). Secondly, there are 7 groups, one per each day, that have the following information:

$$a_m hh_m mm_m pp_m a_n hh_n mm_n pp_n$$

a: Activation character
hh: Scheduled time's hour
mm: Scheduled time's minutes
pp: Scheduled position for each time *-1*
m *(sub index)*: Morning
n *(sub index)*: Night

Once the codification is known, the information that the *Figure 16* and *Figure 17* show can be understood. In the interface (*Figure 16*) appears that the morning schedule is switched on and is scheduled at 08:50 to move the blind to the 50% position and the night schedule is switched off. In the input_text (*Figure 17*) it is possible to see that in the $7 \cdot 5 + 1$ position (where 7 is the size of the group, 5 is the day which corresponds to Saturday and +1 is the offset due to the first character) appears the characters *10850490------* which represents the same that it was configured in the interface.

The second feature used, is an automation code through the *automation.yaml* file which was introduced before. As it was explained before, one automation is based in one trigger and one or some actions. In this section only the trigger is going to be explained because the action is moving the blind to a certain position and it will be explained in the next section.

The purpose of the trigger is to identify when the time that is scheduled is equal to the current time of Home Assistant. To achieve this, it was necessary to configure one entity to know the time in Home Assistant. The code to include this functionality is represented in the next box.

configuration.yaml

```
sensor:
  - platform: time_date
    display_options:
      - "time"
```

This sensor is called *time_date* and with the display options different types of sensor can be added. In this case, the display option "time" was perfect because the sensor has the actual time in the format that was configured in the interface (e.g., 08:00).

Once the sensor is described, it is time to explain the trigger of the automation code. It is important to say that the *automation.yaml* file is checked when one of the entities changes. So, the fact of adding a *time* sensor is perfect because on the one hand, the actual time of the time is going to be stored in one entity and on the other hand, this entity is going to be the reason for checking the *automation.yaml* file at least once per minute.

In the following box, the trigger's code is shown.

```
automations.yaml
- alias: 'AM43 Morning Activation'
  trigger:
    - platform: template
      value_template: >
       {% set d  = now().isoweekday()-1 %}
       {% set s  = states('input_text.schedule') %}
       {{ (s[14*d+2:14*d+4]+":"+s[14*d+4:14*d+6]) == (states('sensor.time')) }}
```

The way in which the comparison code is implemented, is using the Jinja template engine[8] which can be included in Home Assistant *yaml* files. The logic in the comparison is, firstly, load in the variable *d* the day of the week, secondly, load the value of the *input_text* in which the schedule is configured. Once these variables are loaded, the time which is configured in the *input_text* it can be compared with the *sensor.time* state and if it is equal, the automation's action will be run.

Now that the mechanism of the automation is explained, the next step is to explain the manipulation of the *input_text* through the web component. In the *Figure 16* is possible to see the button (1), which will execute the function *_toggleDay(),* the button (2) which will execute the function *_toggleTime()* and the button (3) which will execute the function *_toggleSchedule()*.

First of all, before explaining the function *_toggleDay(),* it is necessary to explain the variable *d* , whose value changes depending on the interface. In the *Appendix I,* it can be seen, specifically in the function *render()* (which is the function that the Web Component uses to return the HTML code) how this function returns the HTML either of the controller mode or the setting mode depending on the value of the local value *setting*. Apart of returning the HTML code, the value of the variable *d* is modified too. If the controller interface is selected, the value assigned will be the current day of the week using the function *_getISODay()*. The previous action is made in order to show the information of the schedule in the current day. If the setting mode is selected, the value assigned will be the first character of the *input_text* explained before.

Once the value of *d* is explained, it can be possible to understand the function *_toggleDay(),* which is shown in the following code:

```
blind-control.js | _toggleDay(i)
d = d + i;
if (d == -1) {
  d = 6;
} else if (d == 7) {
  d = 0;
}
this.hass.callService("input_text", "set_value", {
  value:
    d.toString() + this.hass.states["input_text.schedule"].state.slice(1),
  entity_id: "input_text.schedule",
});
```

[8] https://jinja.palletsprojects.com/en/master/templates/

The purpose of this function is to modify the first character of the *input_text schedule* in order to set the value *d* which is going to indicate the position of the word that will be modified.

It is important to store the value *d* in an entity because the Web Component is getting refreshed with every entity modification, so, if *d* was declared at the beginning of the code, it would be overwritten with that value in each refresh.

In the next code appears the initial code of the function *_toggleTime(),* which is called with two arguments, *i*, which is going to be the number to be added or subtracted, and *type*, which is going to indicate the option's value to be modified. These types are:

- "mh" : Morning hour.
- "mm" : Morning minute.
- "m%" : Morning percentage.
- "nh" : Night hour.
- "nm" : Night minute.
- "n%" : Night percentage.

blind-control.js | _toggleTime(*i*,*type*)

```
if (type == "mh") {
  off = 2;
  up_li = 23;
  do_li = 0;
} else if (type == "mm") {
  off = 4;
  up_li = 55;
  do_li = 0;
} else if (type == "m%") {
  off = 6;
  up_li = 99;
  do_li = -1;
} else if (type == "nh") {
  off = 9;
  up_li = 23;
  do_li = 0;
} else if (type == "nm") {
  off = 11;
  up_li = 55;
  do_li = 0;
} else if (type == "n%") {
  off = 13;
  up_li = 99;
  do_li = -1;
    }
```

In the code, it can be seen that in each type, the function defines the variables *off* (offset in the *input_text* variable), *up*_li (upper limit of the range) and *do_li* (down limit of the range). The reason about making the code in this way is because there is not sense in creating one function per each type. As it is going

to be explained, the procedure is the same, so, in this way, the fact of having 6 different functions that are going to make the code slower is avoided

Once the particular parameters are set, the function runs the following code:

blind-control.js | _toggleTime(*type*)

```
var data = parseInt(
  this.hass.states["input_text.schedule"].state.slice(
    14 * d + off,
    14 * d + off + 2
  ),
  10
);
data = data + i;
if (data == do_li - Math.abs(i)) {
  data = up_li;
} else if (data == up_li + Math.abs(i)) {
  data = do_li;
}
this._modifySchedule(data, off);
```

The code reads the value of the parameter that is going to be modified, and stores it in an internal variable called *data*, using the value of the variable *d* and the offset which was assigned at the beginning of the function for reading the value. Once the value is read, the value of *data* is modified by adding or subtracting the *i* argument and checking that the value does not reach its limits. Finally, the function *_modifySchedule()*, the most important part of which is presented as follows, is executed.

blind-control.js | _modifySchedule(*data,off*)

```
    value =
      this.hass.states["input_text.schedule"].state.slice(0, 14 * d + off) +
      data2 +
      this.hass.states["input_text.schedule"].state.slice(14 * d + off + 2);

    this.hass.callService("input_text", "set_value", {
      value: value,
      entity_id: "input_text.schedule",
    });
```

The objective of the function is to overwrite the content that was read in the *input_text* and to add the new one with a concatenation between the previous content of the word and the new data.

The last function to be explained is: *_toggleSchedule()*. The objective of this function is to activate or deactivate a scheduled time. It is called with an argument which indicates if the modification is going to be in a scheduled morning (*type* = “m”) time or a scheduled night time (*type* = “n”).

In the initial operation of this function there is a declaration of variables again which are going to depend on the type: offset and value to be written. The value depends on the local variable *s* which is the activation character of each group. There are only two possibilities: if the value of *s* is 0 (deactivated),

s will be modified to 1 (activated) and a generic configuration will be set. In contrast, if it is activated, it will be deactivated and the value "- - - - - -" will be written in the *input_text*. The interesting part of writing that value is that in the *automation.yaml* file, there is never going to be a match between the *input_text schedule* and the current time.

The function explained before is shown below:

```
blind-control.js | _toggleSchedule(type)

if (type == "m") {
      off = 1;
      time = "080049";
    } else if (type == "n") {
      off = 8;
      time = "200049";
    }
    var s = this.hass.states["input_text.schedule"].state.slice(
      14 * d + off,
      14 * d + off + 1
    );

    if (s == "1") {
      value = "0------";
    } else if (s == "0") {
      value = "1" + time;
    }

    this.hass.callService("input_text", "set_value", {
      value:
        this.hass.states["input_text.schedule"].state.slice(0, 14 * d + off) +
        value +
        this.hass.states["input_text.schedule"].state.slice(14 * d + off + 7),
      entity_id: "input_text.schedule",
    });
```

3.1.4. Percentage position movement implementation

In the previous section, it was explained how the schedule setting was implemented. The process of triggering the automation was explained, but not the actions that were supposed to occur.

The *Blind Engine* App has the capability of moving the blind in the exact position either in the main interface or in the schedule setting interface. In the implementation, it was decided not to include this functionality in an exact position in the controller interface, but obviously, include the feature in the schedule setting. To reach that objective, the action in the automation code had to include the functionality of generating the codes that were explained in the reverse engineering section.

For reaching this purpose, it was used the Jinja template as was used in the comparison between the setting time and the current time. At the beginning, the implementation with *for* loops was tried, but the problem with Jinja template is that you cannot store variables between different loops, so it was

necessary to implement the generation of codes using a different way which will be explained after the code. The code of the action is shown as follows:

```
automations.yaml
action:
  - service: mqtt.publish
    data:
      topic: 'am43/commands'
      payload_template: >
       {% set d          = now().isoweekday()-1 %}
       {% set s          = states('input_text.schedule') %}
       {% set pd         = s[14*d+6:14*d+8] | int +1 %}
       {% set ph         = '%0x' % pd %}
       {% set offsets    = [150,134,182,166,214,198,246] %}
       {% set operations = [0,+1,-2,-1,-4,-3,-6,-5,+8,+9,+6,+7,+4,+5,+2,+3] %}

       {% set pcd = offsets[(pd//16)]+operations[16-(16*(pd//16 +1) - pd)] %}
       {% set pch          = '%0x' % pcd %}

       {% if pd < 16 %}
          {{ "00ff00009a0d010"+ph+pch }}
       {% elif pd > 15 %}
          {{ "00ff00009a0d01"+ph+pch }}
       {% endif %}
  - service: input_text.set_value
    data_template:
      entity_id: input_text.notifications
      value: "{{ states('input_text.notifications')[0:2] + states('input_text.schedule')[14*(now()
     .isoweekday()-1)+6:14*(now().isoweekday()-1)+8] }}"
```

The pattern followed by the codes was explained in the chapter *2.2.2. Percentage position movement*. The way in which the generation of the codes was implemented was storing, in one vector called *offsets*, the values of the numbers that are multiples of 16 which appear in the beginning of the "groups". Another vector called *operations*, the operation that has to be calculated is stored too, taking as the reference the number of the *offsets* vector.

In the following picture appears an example of operation if the user was selected a position of 30%.

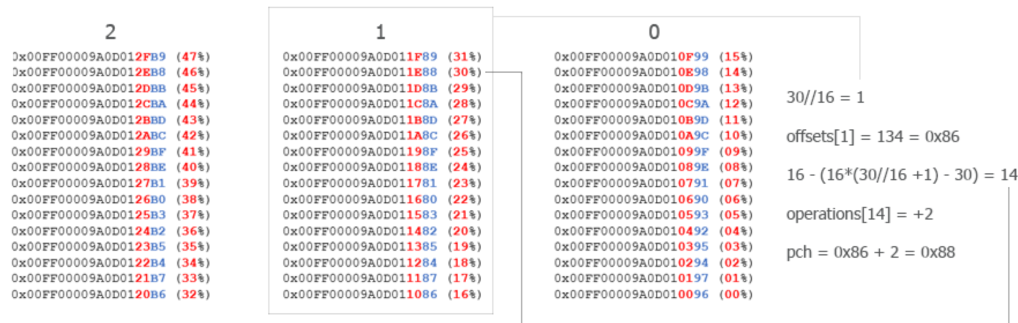


Figure 18. Detail of the generation of the position code's operation

3.1.5. Battery and position request implementation

In the section *3.1.1. MQTT implementation,* the way in which the incoming MQTT messages were read by the Web Component was explained.

In this section, what is going to be explained is the logic that was used in the implementation to identify between the different types of identifications and some actions that had to be included in order to avoid problems during the execution.

The functions *_requestNotification()* and *_getNotification()* will be explained. The first one, is called when it is necessary to request a notification. The second one, it will be called automatically in order to decode the incoming notification.

In the next code the *_requestNotification()* function is shown.

blind-control.js | _requestNotification(*message*)

```
if (localStorage.getItem("busy") == "0") {
  localStorage.setItem("busy", "1");
  this.hass.callService("input_text", "set_value", {
    value: message,
    entity_id: "input_text.mqtt",
  });

  this.hass.callService("homeassistant", "toggle", {
    entity_id: "input_boolean.mqtt",
  });

  const sleep = (milliseconds) => {
    return new Promise((resolve) => setTimeout(resolve, milliseconds));
  };

  sleep(900).then(() => {
    localStorage.setItem("busy", "0");
    this._getNotifications();
  });
}
```

In the code it can be seen that the function will be run only if the **local storage** variable *busy* is 0, and once executed, this is set to 1. In other chapters, the entity *busy* was explained but in this case, it was necessary to use the local storage variable due to the fact that the entity needs certain time to be modified and the function would be executed more than the necessary. However, the local storage variable is modified immediately so the function will be executed only when this variable will be 0 again.

Once the process of setting the variable busy to 1 is done, a MQTT message is sent and the function *_getNotification()* is launched. It can be seen that the call to the function is not immediately done, but it is call after a *sleep* function. This action is done because a certain time is needed in order to modify the entity, send the MQTT message and finally, receive the notification. If this action did not exist, the program would miss the notification because the MQTT sensor would be in *unknown* state.

In the following picture appears the content value of the MQTT sensor when a notification is sent by the AM43 motor.

Figure 19. Example of the MQTT sensor's value when a notification is sent

The notifications that will be analyzed are the battery percentage and the position percentage. In previous chapters, it was explained that the only way to watch the information in any device was creating an entity in the Home Assistant server. In this case, another *input_text* was created called notifications. This *input_text* contains the value of the battery percentage in its two first characters, and the value of the position percentage in the last two characters. In *Figure 20* it is shown an example of this entity. In that case, the value of the battery is 13% and the value of the position in that moment is 9%.

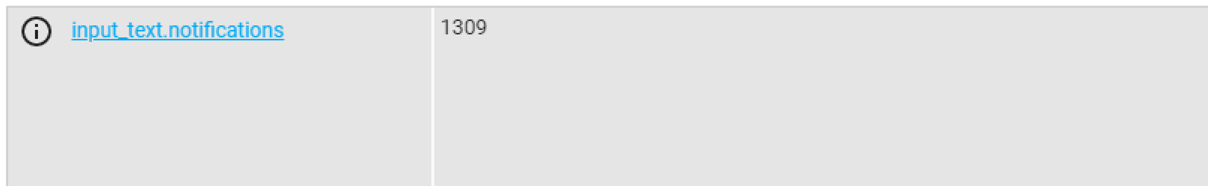


Figure 20. Example of the notifications input_text's value

Now that the way in which the notification is stored is shown, it is possible to explain the function *_getNotifications()*. The purpose of this function will be the modification of the *input_text notification*s with the values that the AM43 motor is sending. Apart of the previous purpose, this function implements the logic in which the user is going to be able to watch the updated position of the motor during the movement.

Before explaining the function, it is necessary to introduce its local variable *id.* In the section *2.2.3. Battery and position request,* the notifications received because of the requests of battery and position were explained. The notifications were similar in some bytes and with a different behavior in other bytes. Watching the notifications, it can be seen that, for example, the second byte is constant and is different between each notification. This is the reason about at the beginning of the function, the second byte of the notification is assigned to the variable *id*, and this can be compared either with A2, if it is battery notification, or with A7 if it is position notification. For example:

```
blind-control.js | _getNotifications()
if (id == "A2") {
  nbattery = parseInt("0x" + sensor.slice(14, 16)).toString();
  if (nbattery == "100") {
    nbattery = "99";
  }
  nposition = position;
}
```

In the previous function, the code which is executed when the notification is about battery percentage, *id = A2*, is shown. The operation of the code inside each conditional statement *if* is to store in a local variable called *nbattery* or *nposition* the new value. In this case, it is a battery notification so the value *nbattery* is modified with the value of the byte which contains the information transformed to decimal.

The value of the other variable will be assigned to the value which had the *input_text notification*. At the end of this section it will be understood why this action is made.

In the following code, appears the code which will be run if the notification is a position percentage, *id = A7*.

blind-control.js | _getNotifications()

```
else if (id == "A7") {
      nposition = parseInt("0x" + sensor.slice(10, 12)).toString();

      if (localStorage.getItem("goup") == "1") {
        if (this._fixPosition(nposition) == 100) {
          setTimeout(() => {
            this._sendCommand("00FF00009A0A01CC5D", "c");
          }, 1000);
        } else {
          this._requestNotification("00FF00009AA701013D");
        }
      } else if (localStorage.getItem("godown") == "1") {
        if (this._fixPosition(nposition) == 0) {
          setTimeout(() => {
            this._sendCommand("00FF00009A0A01CC5D", "c");
          }, 1000);
        } else {
          this._requestNotification("00FF00009AA701013D", "c");
        }
      }
      nbattery = battery;
```

The operation is similar to battery notification's code, but in this case, a new functionality is added. In the *3.1.2. Go up, go down and stop commands implementation,* it was shown the modification of local variables in which the state of the blind could be stored (*goup, godown* and *stop*). That information is used in this function in order to manage some aspects of the operation.

Firstly, the function will compare if the blind reaches the limits, either 100, if it is going up, or 0 if it is going down. If the comparison is TRUE, a stop command will be sent, if not, the function will send another notification request of position. This will make the program enter in a loop and the position will be updated in the whole movement of the blind until it reaches the limit.

If the user decided to stop the blind, the local variable *stop* would be set to 1 and the local variables *goup* and *godown* to 0. This would make the actual function stop and not sending more notification requests because the code inside of the conditional statements will not be launched.

The final action executed in the function is shown as follows:

```
blind-control.js | _getNotifications()
value = nbattery + nposition;

    this.hass.callService("input_text", "set_value", {
      value: value,
      entity_id: "input_text.notifications",
    });
```

This code basically modifies the *input_text notification* writing the new value which is composed by the combination between the local variable *nbattery* and *nposition.* As explained before, this is going to make the battery and the position readable by all the devices connected to Home Assistant, and not only by the device which executes the action.

3.1.6. Direction and velocity setting implementation

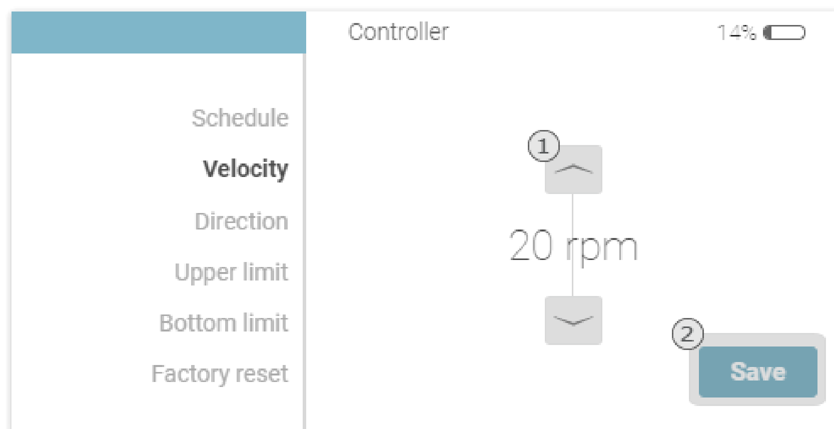


Figure 21. Detail of the velocity setting's interface in the implemented custom card

In this section, the functionality of changing the velocity and the direction will be explained. In previous sections was shown that the codes that configure the velocity and the direction are linked in some way. The interface is designed in the same way for both functionalities and can be seen in the *Figure 21*. In the interface appears the button group 1 which runs the function *_toggleConfig()* and the button 2 which runs either the function *_setVelocity()* or *_setDirection()* depending on the option selected in the menu.

Remember that in the *Appendix I* it can be seen all the HTML code and understand better how the design was made. In this case, it can be noticed than the buttons have event listeners which are going to call the functions with their respective arguments.

As it happens when it is wanted to have values in all the devices connected, it was necessary to create an already known entity, *input_text*, in order to store the value of the configuration that is already configured in the motor.

This *input_text* is called *configuration* and stores the velocity in the two first characters and the direction in the last one. In the next picture appears an example of this *input_text*.

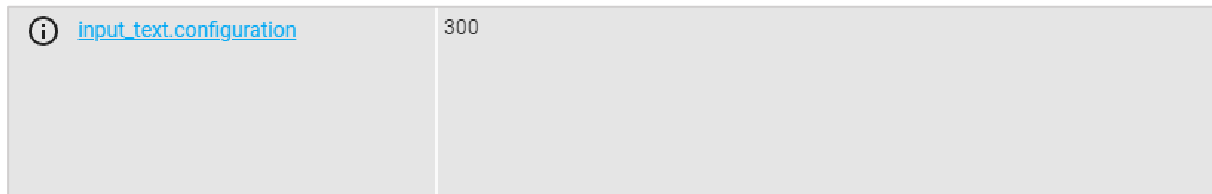


Figure 22. Example of the configuration input_text's value

The function *_toggleConfig()* has a similar behavior than the *_toggleTime()*. The only difference is that in the schedule modification, the *input_text* was modified directly because it is rare that Home Assistant does not achieve the call that modifies an *input_text*. In this case, is different because the *input_text* must not be modified unless it would know that the AM43 motor has the new configuration. For this purpose, the local variables *direction0, velocity0*, *direction1 and velocity1* were created.

The solution implemented was to initialize all the variables with the value stored in the *input_text configuration*. Once the variables have the value, in the interface, only the variables *direction1* and *velocity1* are going to be represented. The function is only going to modify those variables and the values that the motor has are going to be still stored in the variables, *direction0* and *velocity0*.

In the next code appears the function *_toggleConfig():*

blind-control.js | _toggleConfig(*i, type*)

```
if (type == "v") {
  up_li = 50;
  do_li = 20;
  id = "velocity1";
} else if (type == "d") {
  up_li = 1;
  do_li = 0;
  id = "direction1";
}

var data = parseInt(localStorage.getItem(id), 10);

data = data + i;
if (data == do_li - Math.abs(i)) {
  data = up_li;
} else if (data == up_li + Math.abs(i)) {
  data = do_li;
}

localStorage.setItem(id, data.toString());

this.requestUpdate();
```

As was mentioned before, it can be seen that the function has the same logic than the similar buttons that modify the schedule configuration, but, in this case the modification is made on the local variables with the command `localStorage.setItem(id, data.toString),` where the *id* is assigned in the beginning on the function depending on the variable that it wanted to be modified (velocity or direction).

Once the value is selected, the user will press the save button (2) and the function *_setVelocity()* or *_setDirection()* will be launched. In the beginning of the function, the generation of the variable part of the command is implemented. As it was seen in the section *2.2.4.Direction and velocity setting,* the codes act differently if the direction is reverse or forward.

In the following code appears the integration when the forward direction is set:

blind-control.js | _setVelocity()

```
if (localStorage.getItem("direction0") == "1") {
  var c = 0;
  direction = "14";
  command = 87;
  for (
    i = 0;
    i <= parseInt(localStorage.getItem("velocity1"), 10) - 20;
    i++
  ) {
    if (i > 0) {
      command = command - 1;
      c++;
      if (c == 4) {
        command = command + 8;
        c = 0;
      }
    }
  }
  command = command.toString(16);
}
```

Basically, the pattern of operations that affects to the variable byte is automated, so, the code is valid to all the values of velocity.

It is important to add that there is a little difference between the functions *_setVelocity()* and *_setDirection()*. In the code that was exposed before, included in the function *_setVelocity()*, it can be seen how the first logical statement is with the local storage variable *direction0* and for the generation of the pattern the value used is the *velocity1*. This is due to velocity being modified, so the value of direction has to be the value *direction0* because this variable contains the stored value in the AM43 motor.

If the function *_setDirection()* had been explained, it would have appeared the first comparation with the local storage variable *direction1*, and the code will be generated using the value of *velocity0*.

In the next code appears the integration when the reverse direction is set:

```
blind-control.js | _setVelocity()
else if (localStorage.getItem("direction0") == "0") {
  direction = "16";
  command = 85;
  for (
    i = 0;
    i <= parseInt(localStorage.getItem("velocity1"), 10) - 20;
    i++
  ) {
    if (i > 0 && i % 2 === 0) {
      command = command + 3;
    } else if (i > 0 && i % 2 !== 0) {
      command = command - 1;
    }
  }
  command = command.toString(16);
}
```

Once the variable byte is generated, the next step in the function is to generate the entire command that is going to be sent to the motor. For reaching this purpose, is enough to write the value in the *input_text mqtt* and toggle the *input_boolean mqtt* using the call services explained in previous sections. The MQTT message will be sent after doing these operations.

In the following code appears the actions mentioned before. Only using "+" symbols it is easy to concatenate the strings. The strings are concatenated following the structure that was explained in the section *2.2.4. Direction and velocity setting*.

It is important to mention that these codes are going to be different between the functions *_setVelocity()* and *_setDirection()* again. In this case, the velocity is being modified so it is necessary to use *velocity1* transformed to a hexadecimal number in the concatenation. In case of the *_setDirection()* function the value of *velocity0* would have been used.

```
blind-control.js | _setVelocity()
this.hass.callService("input_text", "set_value", {
  value:
    "00FF00009A1106" +
    direction +
    parseInt(localStorage.getItem("velocity1"), 10).toString(16) +
    "0007D00D" +
    command,
  entity_id: "input_text.mqtt",
});

this.hass.callService("homeassistant", "toggle", {
  entity_id: "input_boolean.mqtt",
});
```

Once the message is sent, the AM43 motor will send a notification, in the section *2.2. Reverse engineering* it was shown the notifications that the motor sends when the message is received correctly.

These notifications are perfect to know if the values are set correctly. The code implemented for this purpose appears at the end of this section. The actions made are, firstly, launch the code 400 ms later, this delay will allow the Web Component to read the value of the MQTT sensor. Secondly, it will be checked if the value of that MQTT sensor fits with the correct notification. On the first hand, if the comparation is TRUE, the value of the *velocity0* or *direction0* will be modified with the value of *velocity1* or *direction1*. On the other hand, if the comparation is FALSE, the variables *velocity1* and *direction1* will be modified with the values stored in the variables *velocity0* or *direction0*. This operation plus a pop-up message will indicate the user that the configuration is not made.

blind-control.js | _setVelocity()

```
setTimeout(() => {
  if (this.hass.states["sensor.mqtt_sensor"].state == "9A11015A31") {
    this.hass.callService("input_text", "set_value", {
      value:
        localStorage.getItem("velocity1") +
        this.hass.states["input_text.configuration"].state.slice(2, 3),
      entity_id: "input_text.configuration",
    });
    localStorage.setItem("velocity0", localStorage.getItem("velocity1"));
    this._message("Velocity was set correctly");
  } else {
    localStorage.setItem("velocity1", localStorage.getItem("velocity0"));
    this._message("Velocity was not set correctly");
  }
  this.requestUpdate();
}, 400);
```

3.1.7. Upper and bottom limit setting implementation

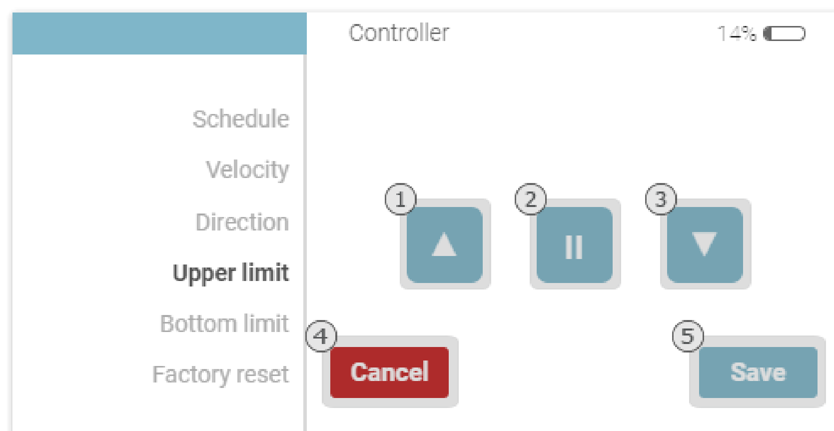


Figure 23. Detail of the upper limit's interface in the implemented custom card

In this section, the upper and bottom limit setting will be explained. It was shown that the codes of these operations are not complicated. The difficulty of reaching this implementation was to implement a logic for avoiding errors in the application that could appear, e.g., sending another kind of configuration without cancelling the actual limit operation setting.

The *Figure 23* shows the interface that will be returned by the Web Component only if there is connection with the AM43 motor. This is made following the operation of the *Blind Engine* App, in which when the button of the option's menu is pressed, the motor receives the command to attend a new limit. In the implementation, the function that is executed when the option of the menu is pressed is the function *_requestUpperLimit()* or *_requestBottomLimit()*. The first one is shown as follows:

blind-control.js | _requestUpperLimit()

```
if (localStorage.getItem("busyoption") != 3) {
  this.hass.callService("input_text", "set_value", {
    value: "00FF00009A2203000200B9",
    entity_id: "input_text.mqtt",
  });

  this.hass.callService("homeassistant", "toggle", {
    entity_id: "input_boolean.mqtt",
  });
  localStorage.setItem("busyoption", 3);

  setTimeout(() => {
    this.requestUpdate();
  }, 500);

  setTimeout(() => {
    if (this.hass.states["sensor.mqtt_sensor"].state == "9A22015A31") {
      localStorage.setItem("upper", "1");
      localStorage.setItem("busylimit", "1");
    } else {
      localStorage.setItem("upper", "0");
      this._message("Failed to establish the connection");
    }
  }, 400);
}
```

In the code, it can be seen that the function starts with a logic statement comparing the value of *busyoption*. In the last section of this chapter it will be explained the option menu operation, but it is necessary to introduce the variable now. This variable contains the number of the option that is actually selected in the menu. So basically, the code will be executed if we are not in the option 3, which is the option of the upper limit. This is done in order to avoid the user to click the option again and produce an error in the app.

Once the logic statement is checked, the code sends the MQTT message to request the setting of a limit. A code with a delay will be executed in order to check the notification, and if the notification has a correct value, the local variables *upper* and *busylimit* will be set to one. These variables are used to show the interface (*upper*) shown in the *Figure 23* and to not allow the user to change into another option unless they cancel the operation (*busylimit*).

The previous function is the same for the bottom limit, the only thing that changes is the number of the busy option, and the name of the local variable upper, which would be bottom.

The other functions are linked to the buttons as follows:

Firstly, the buttons 1, 2 and 3 will execute the function *_sendCommand()* which was explained in the section *3.1.2. Go up, go down and stop commands implementation.* In this case, the type which is send by the buttons is “s”, so, the function will only send the command to move the motor and not start the process of getting the position’s evolution.

Secondly, the button 4 will execute the function *_cancelLimit().* This function will cancel the process of setting a new limit sending the MQTT message linked to that option. The notification will be checked and if it is correct the local variable *busylimit* it will be 0, so the user will be able to change the option. If the notification is not correct a message will appear indicating that it could not be possible to cancel the operation and the user will have to do it again.

The *_cancelLimit()* function is shown as follows:

blind-control.js | _cancelLimit()

```
this.hass.callService("input_text", "set_value", {
  value: "00FF00009A2203400100FA",
  entity_id: "input_text.mqtt",
});

this.hass.callService("homeassistant", "toggle", {
  entity_id: "input_boolean.mqtt",
});

setTimeout(() => {
  if (this.hass.states["sensor.mqtt_sensor"].state == "9A22015C31") {
    this._message("Limit configuration was cancelled");
    localStorage.setItem("busylimit", "0");
  } else {
    this._message("Failed to cancell");
  }
  this.requestUpdate();
}, 400);
```

Finally, the button 5 will execute the function *_setUpperLimit(),* which has the same logic than *_setBottomLimit().* The logic of this function is very similar to the other kind of function explained in previous sections. The MQTT message corresponding with the setting of the upper limit or the setting of the bottom limit will be sent. Then, a notification check will be made, and the user will be informed about the result of this comparation. In addition, if the comparation is TRUE, the value of the *input_text notification* will be modified either with 100, if the upper limit is configured, or with 0, if the bottom limit is configured. This is made to have a real representation of the blind when the controller mode will be turned on again.

The *_setUpperLimit()* function is shown as follows:

blind-control.js | _setUpperLimit()

```
this.hass.callService("input_text", "set_value", {
  value: "00FF00009A220320020099",
  entity_id: "input_text.mqtt",
});

this.hass.callService("homeassistant", "toggle", {
  entity_id: "input_boolean.mqtt",
});

setTimeout(() => {
  if (this.hass.states["sensor.mqtt_sensor"].state == "9A22015B31") {
    this._message("Limit was set correctly");
    localStorage.setItem("busylimit", "0");
    this.hass.callService("input_text", "set_value", {
      value:
        this.hass.states["input_text.notifications"].state.slice(0, 2) +
        100,
      entity_id: "input_text.notifications",
    });
  } else {
    this._message("Limit was not set correctly");
  }
  this.requestUpdate();
}, 400);
```

3.1.8. Factory reset setting implementation

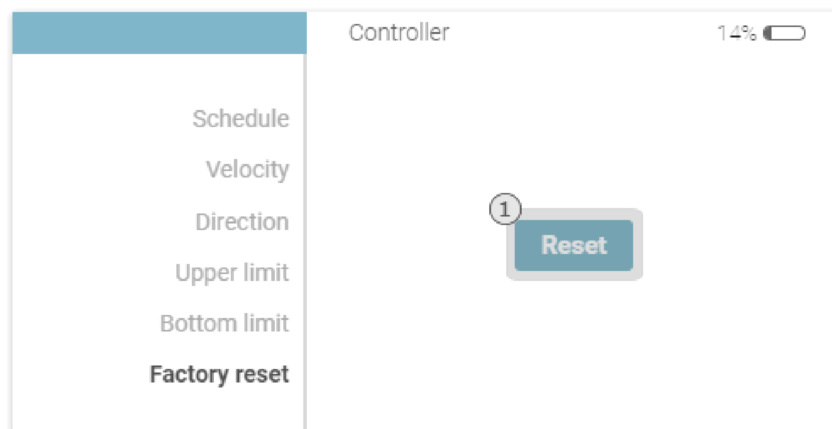


Figure 24. Detail of the factory reset's interface in the implemented custom card

In this section the implementation of the factory reset option is going to be explained. Actually, it is the easiest implementation, but there are some details that have to be explained.

In the section *2.2.7. Factory reset request* the only code that was used in order to generate the original state in the motor was shown. It is obvious that the code of sending a MQTT message is implemented in the function *_factoryReset()*. Actually, the function can be executed in two different scenarios. In one case, the function can be executed in the settings mode, which appears in the *Figure 24*, pressing the button. In the other case, the function can be executed in the function *_checkConnection()* which will be explained in the next section.

```
blind-control.js | _factoryReset()

this.hass.callService("input_text", "set_value", {
  value: "00FF00009A2203000001BA",
  entity_id: "input_text.mqtt",
});

this.hass.callService("homeassistant", "toggle", {
  entity_id: "input_boolean.mqtt",
});

if (localStorage.getItem("settings") == "1") {
  this._message("Reseting...");
}
```

In addition to sending the message, it was necessary to update all the *input_text* that are affected with this factory reset. In the *Blind Engine* app, the changes made when a factory reset is executed it could be seen. These modifications are shown as follows:

- Velocity: 50 rpm.
- Direction: Reverse.
- Schedules: Deleted.

In the next code appears the implementation of those changes in the system:

```
blind-control.js | _factoryReset()

setTimeout(() => {
  this.hass.callService("input_text", "set_value", {
    value: "500",
    entity_id: "input_text.configuration",
  });

  this.hass.callService("input_text", "set_value", {
    value:
      "00------0------0------0------0------0------0------0------0------0------0------0------0-----
       0------",
    entity_id: "input_text.schedule",
  });
```

```
  this.hass.callService("input_text", "set_value", {
    value: "00000",
    entity_id: "input_text.notifications",
  });
  window.location.reload();
}, 2000);
```

It can be seen that it is only necessary to call the service of modifying the *input_text's* value. Firstly, the *input_text configuration* is modified to have the 50 rpm velocity and the reverse direction. Secondly, the *input_text schedule* is modified in order to deactivate all the configured schedules. Finally, the *input_text notifications* is set to 00000. The reason about modifying this last *input_text* will be explained in the next section with the explanation of the function *_checkConnection()*.

3.1.9. Checking connection implementation

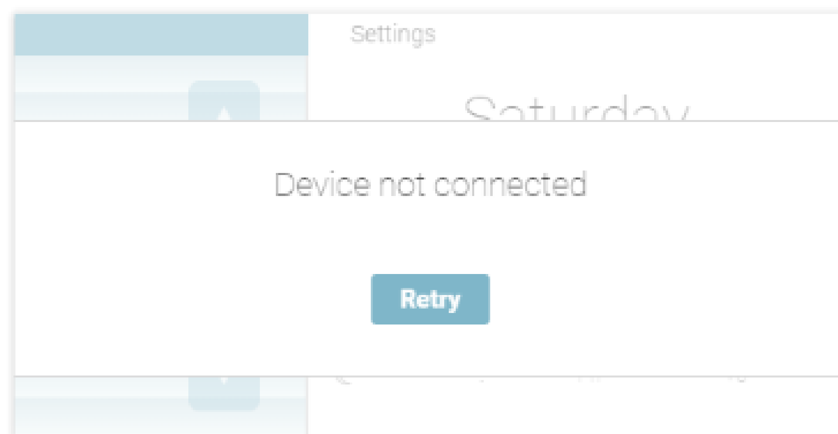


Figure 25. Detail of the non conection's interface in the implemented custom card

This section is the last one explained in this chapter. The implementation of checking the connection will be explained. This is one of the most important features that the new implementation offers because it will give the user the feedback about the connection with the AM43 motor.

In this section the built-in function that the LitElement offers that was very useful will be presented. This function is *firstUpdated()*.

In the *Figure 25*, the interface that is going to be shown if there is no connection between the ESP32 and the AM43 motor appears. The way in which this interface is shown, is through the function *_retryConnectionGraphics()* that is included in the HTML file, *Appendix I*.

The mentioned function is shown as follows:

```
blind-control.js | _retryConnectionGraphics()
if (localStorage.getItem("status") == "0") {
  return html`
    <div id="retry">
      <div class="division33"><br />Device not connected</div>
      <div class="division20"></div>
      <div class="division33">
        <div id="setting_set" @click="${() => window.location.reload()}">
          Retry
        </div>
      </div>
      <div class="division33"></div>
    </div>
  `;
} else if (localStorage.getItem("status") == "-1") {
  return html` <div id="retry">
    <div class="division33"></div>
    <div class="division33">Loading...</div>
    <div class="division33"></div>
  </div>`;
} else if (localStorage.getItem("status") == "1") {
  return html``;
}
```

The function will return three different HTML codes depending on an *if* statement. The comparison is made between the value of a localstorage variable called *status*. The first value of the variable is -1, so a loading screen will be shown until the program changes the variable into 1 (connected) or 0 (unconnected). The HTML layer will be on the top and will not let the user operate. This is the reason why when the status is connected, the HTML is empty, and the user will be able to interact with the controller interface.

The function that is going to change the value of the status variable is *_checkConnection*() and is executed through the *firstUpdated()* function which is shown as follows:

```
blind-control.js | firstUpdated()
this._checkConnection();
setInterval(() => {
  this._checkConnection();
}, 600000);

setTimeout(() => {
  this._requestNotification("00FF00009AA2010138");
  setTimeout(() => {
    this._requestNotification("00FF00009AA701013D");
  }, 1000);
}, 1000);
```

The *firstUpdated()* function is a built-in function of LitElement that will be executed when the website is shown for the first time. In this section it is important because it is going to launch the *_checkConnection()* function, but this function is used to initialize all the localstorage variables too. In addition, there are other function called *updated(),* which is included in LitElement too, but this function is used to modify the graphics of the interface (button's transparency, update values, etc.). This function can be called using the function *requestUpdate().*

In the *firstUpdated()* function shown before, it can be seen how the function *_checkConnection()* is launched when the website is shown, and then it is configured to be executed each 10 minutes (600000 ms) with the function *setInterval().* Therefore, the WebComponent will check the connection each 10 minutes. After launching *_checkConnection()* the values of battery and position will be requested from the AM43 motor. This will be executed even if the motor is not connected, if this is the case, the *getNotification()* function will return nothing.

blind-control.js | _checkConnection()

```
if (localStorage.getItem("busy") == "0") {
  this.hass.callService("input_text", "set_value", {
    value: "00FF00009A170222B815",
    entity_id: "input_text.mqtt",
  });

  this.hass.callService("homeassistant", "toggle", {
    entity_id: "input_boolean.mqtt",
  });
```

In the previous code appears the beginning of the *_checkConnection()* function. For running the function, the local storage variable *busy* has to be free (0). The reason is that the function is set to be launched each 10 minutes, so, if the user is controlling the blind, there is the possibility of launch two functions at the same time.

Once the statement is done, a command is sent to the AM43 motor. The command shown in the code, appeared in all the WireShark packets, and was a command that was launched periodically by the *Blind Engine* app. A simple notification is received when this command is sent, this is the reason about this command was used for notice if there is a current connection with the motor.

Once the code is sent, a code included in a *setTimeout*() is launched again to read the value of the notification:

blind-control.js | _checkConnection()

```
setTimeout(() => {
        if (this.hass.states["sensor.mqtt_sensor"].state == "9A17015A31") {
          if (
            this.hass.states["input_text.configuration"].state == "unknown" ||
            this.hass.states["input_text.notifications"].state == "unknown" ||
            this.hass.states["input_text.schedule"].state == "unknown"
          ) {
            this._factoryReset();
          } else {
            setTimeout(() => {
              localStorage.setItem("status", "1");
```

```
            setTimeout(() => {
              this.hass.callService("input_text", "set_value", {
                value: "0",
                entity_id: "input_text.busy",
              });
            }, 1500);
            this.requestUpdate();
          }, 3500);
        }
      } else {
        localStorage.setItem("status", "0");
        this.hass.callService("input_text", "set_value", {
          value: "1",
          entity_id: "input_text.busy",
        });
        this.requestUpdate();
      }
    }, 500);
```

In the code, the two scenarios that can occur appear.

In one case, if the notification has the expected value, firstly, the *input_text* entities will be read. In case of being *unknown* (first execution of the custom-card) the *_factoryReset()* function will be called. Remember that the *_factoryReset()* function wrote the initial values of the parameters into the *input_text*. After the *_factoryReset()* execution, the *_checkConnection*() function will be launched again, so the variables will not be *unknown* and the *status* will be set to 1, the *busy* entity will be set to 0 and the interface will be refresh in order to show the interface corresponding to *status* = 1. This will happen in every execution except for the first one.

In the other case, if the notification does not match with the expected value, the status will be set to 0 and the *busy* entity will be set to 0. An update will be requested in order to show the interface corresponding to *status* = 0 (*Figure 25*).

In the interface it can be seen that appears a Retry button. In the HTML code of the *Appendix I* it can be seen that the button has an event listener: *@click="${() => window.location.reload()*. The call will refresh the website so the *_checkConnection()* function will be launched again because of being included in the *firstUpdated()* function.

3.2. ESP-32

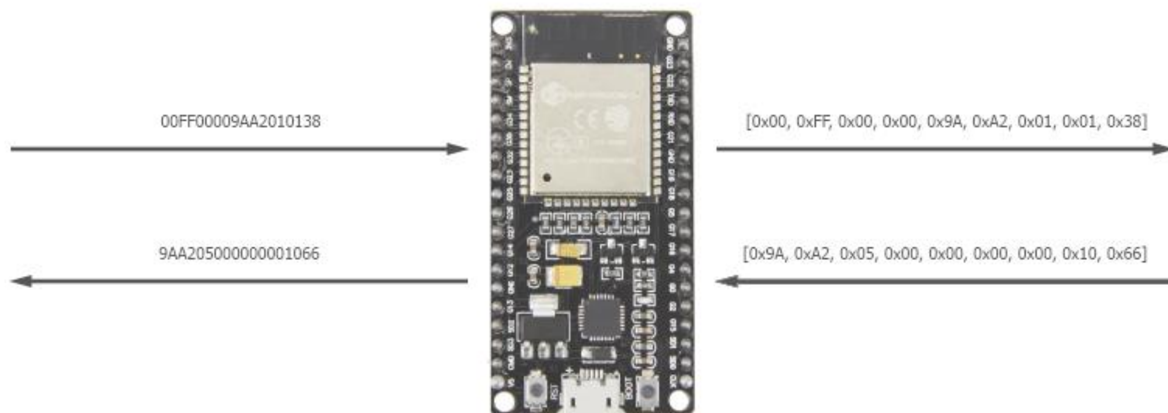


Figure 26. Scheme of the ESP32 microcontroller's operation

In the previous section, the implementation of the basic operations of MQTT in Home Assistant was explained. It was mentioned that it was possible to send MQTT messages through the topic *am43/commands* and to receive MQTT messages through the topic *am43/notifications*.

As was presented in the introduction of the project, an ESP32 was the selected device in order to receive MQTT messages and convert these messages into BLE commands.

In this section, the implementation of the MQTT-BLE converter will be explained. A graphical scheme of this operation is shown in the *Figure 26.* In the figure it can be seen how the ESP32 is going to receive the MQTT messages in the format that was shown in the Home Assistant implementation, and the device will be able to send a byte array. The reverse operation is made by the ESP microcontroller too, the microcontroller will be able to receive the bytes array and will transform it into MQTT messages.

To program the ESP32, the firmware *micropython* was chosen at the beginning, however, there were a lot of problems with the libraries, specifically with the BLE's libraries. This is due to the fact that the *micropython* firmware is actually in development, so there are a lot of functionalities that are not implemented yet.

Due of this inconvenient, a research was made an it was found that there was a way to install the Arduino core in a ESP32 device, with this action, some of the libraries that Arduino uses could be used in the ESP32.[9]

In conclusion, the **Arduino IDE** could be used in order to program the integration, so there was a whole community to get the necessary information.

The libraries that were implemented and have to be explained in this section are: Wi-Fi, MQTT and BLE.

[9] https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions

3.2.1. Wi-Fi Implementation

The Wi-Fi implementation in the ESP32 is necessary in order to connect the device to the network. The use of Wi-Fi seemed very easy and it definitely was. In the next code appears the initialization of the library and the constants that will be used later in the methods of the WiFi class.[10]

```
#include <WiFi.h>

const char* ssid        = "ssid";
const char* password    = "password";
```

Once the *ssid* and the *password* are defined, the following code will be executed in the *setup()* function:

```
WiFi.begin(ssid, password);


while (WiFi.status() != WL_CONNECTED) {
  delay(500);
}
Serial.println("WiFi Connected");
```

The device will be connected to the network and it will be able to use the MQTT protocol, which without being connected to the network would have been impossible.

3.2.2. MQTT Implementation

Once the Wi-Fi connection is explained, it can be explained the implementation of the MQTT operation in the ESP32. The difficulty of this implementation is higher than that on the Wi-Fi one. In the next codes appears the initialization of the library, the initialization of client and the code included in the *setup()* function in order to connect the client and subscribe to the topic using the MQTT methods[11]:

```
#include <PubSubClient.h>

const char* mqtt_server = "xxx.xxx.xxx.xxx";


WiFiClient espClient;
PubSubClient client(espClient);
```

```
client.setServer(mqtt_server, 1883);
client.setCallback(callback);
if (client.connect("espClient") && client.subscribe("am43/commands")) {
  Serial.println("MQTT Connected");
}
```

[10] https://www.arduino.cc/en/Reference/WiFi
[11] https://pubsubclient.knolleary.net/api.html

It was seen in the Home Assistant's implementation that the MQTT messages are going to be sent through the topic *am43/commands*. That is the reason about it is necessary to subscribe the device in this topic, only in this way the device will be able to receive the MQTT messages. In the *setup()* code appears the declaration of the function *callback* with the method *setCallback*. This function will be executed when a new message will be received in a subscribed topic. The function is shown as follows:

```
void callback(char* topic, byte* payload, unsigned int length) {

  String   message;

  for (int i = 0; i < length; i++) {
    message += (char)payload[i];
  }

  String id = message.substring(2, 4);

  char     cmessage[3];
  byte     command[message.length() / 2];

   for (int i = 0; i < sizeof(command); i++) {

      message.substring(2 * i, 2 * i + 2).toCharArray(cmessage, 3);
      command[i] = strtol(cmessage, NULL, 16);
    }

    pRemoteCharacteristic->writeValue(command, sizeof(command));
}
```

The function's operation is, firstly, storing in a local string variable called *message* the content of the *payload* value, which is the content of the MQTT message received. This action is made by a for loop which is going to convert the values of the byte into *char* variables which are concatenated into the *message* variable. Once the variable message has the string content, a bytes array vector is generated using the method *.toCharArray* and transforming the string into hexadecimal numbers. The last step is sending the BLE command: *pRemoteCharacteristic->writeValue(command, sizeof(command)).*

There is an additional function called reconnect which is going to be called when the connection of the MQTT devices was finished.

```
void reconnect() {

  while (!client.connected()) {
    client.connect("espClient");
    client.subscribe("am43/commands");
    delay(1000);
  }

}
```

3.2.3. BLE Implementation

In this section the Bluetooth Low Energy's implementation will be explained. This implementation was supposed to be the most difficult. At the end, it was relatively easy because the Arduino IDE contains built-in examples that can be used in the projects.

In the following codes appears the initialization of the necessary variables which are going to be used with the BLE class' methods[12]:

```
#include "BLEDevice.h"

static BLEUUID serviceUUID("0000fe50-0000-1000-8000-00805f9b34fb");
static BLEUUID    charUUID("0000fe51-0000-1000-8000-00805f9b34fb");

static boolean doConnect = false;
static boolean connected = false;
static boolean doScan = false;
static BLERemoteCharacteristic* pRemoteCharacteristic;
static BLEAdvertisedDevice* myDevice;
```

It can be seen that it was necessary to introduce the service and the characteristic of the BLE server. The way in which these parameters were obtained was using the Android application **nRF Connect** available in Google play store.

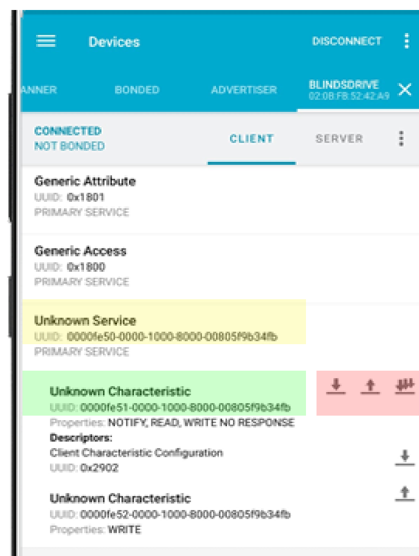


Figure 27. Detail of the service (yellow) and characteristic (green) in the nRF Connect app.

The operation with the nRF application was quite simple, it was only necessary to connect the android phone to the device, then, the interface of the *Figure 27* appeared. What appears in this interface is, firstly, the BLE service in yellow and the BLE characteristic in green. Furthermore, the application offers the capability of sending commands and monitoring the notifications. These last functionalities

[12] https://www.arduino.cc/en/Reference/ArduinoBLE

can be used pressing the buttons that appear in red. They were very useful because it was possible to check all the commands before implementing them finally in the Web Component.

Once the value of the service and the characteristics were known, it was possible to establish a connection with the device. Th code that is used to establish the connection appears in the *setup()* function. It is shown as follows:

```
BLEDevice::init("");
BLEScan* pBLEScan = BLEDevice::getScan();
pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
pBLEScan->setInterval(1349);
pBLEScan->setWindow(449);
pBLEScan->setActiveScan(true);
pBLEScan->start(5, false);
```

The code basically assigns the class *MyAdvertisedDeviceCallbacks()* to each new device that is scanned. The class contain a function that it can be seen as follows:

```
class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks {

    void onResult(BLEAdvertisedDevice advertisedDevice) {

      if (advertisedDevice.haveServiceUUID() &&
advertisedDevice.isAdvertisingService(serviceUUID)) {

        BLEDevice::getScan()->stop();
        myDevice = new BLEAdvertisedDevice(advertisedDevice);
        doConnect = true;
        doScan = true;

      }
    }
};
```

In the function it can be seen that if the service and the characteristic of the scanned device fits with the service and the characteristic declared at the beginning on the code, the scan will stop and the variable *myDevice* will be assigned with the information of the device.

Once the information of the device is stored, the connection is made via the *connectToServer()* function:

```
bool connectToServer() {
  BLEClient*  pClient  = BLEDevice::createClient();
  pClient->setClientCallbacks(new MyClientCallback());
  pClient->connect(myDevice);

  BLERemoteService* pRemoteService = pClient->getService(serviceUUID);
  pRemoteCharacteristic = pRemoteService->getCharacteristic(charUUID);
  pRemoteCharacteristic->registerForNotify(notifyCallback);

  connected = true;
  return true;
}
```

This function is called in the *loop()*. In the next code shows the *loop()* function. If some connections were lost during the operation, the system either will call direct reconnect functions or it will switch Boolean variables in order to call indirect reconnect functions .

```
void loop() {

  if (doConnect == true) {
    if (connectToServer()) {
      Serial.println("We are now connected to the BLE Server.");
    } else {
      Serial.println("We have failed to connect to the server; there is nothin more we will
do.");
    }
    doConnect = false;
  }

  if (connected) {
    if (!client.connected()) {
      reconnect();
    }
    client.loop();
  } else if (doScan) {
    BLEDevice::getScan()->start(0);
  }

}
```

One of the actions of the *connectToServer()* function, is to assign the function which is going to be executed when a notification is received, in this case, the function which was assigned is *notifyCallback():*

```
static void notifyCallback(
  BLERemoteCharacteristic* pBLERemoteCharacteristic,
  uint8_t* pData,
  size_t length,
  bool isNotify) {
  String message = "";
  char   cmessage[length * 2 + 1];

  Serial.print("Notify callback for characteristic ");
  Serial.print(pBLERemoteCharacteristic->getUUID().toString().c_str());
  Serial.print(" of data length ");
  Serial.println(length);

  for (int i = 0; i < length; i++) {
    pData = pData + i; //Pointer + (2*i)bytes
    if (String(*pData, HEX).length() == 1) {
      message += "0" + String(*pData, HEX);
    }
    else {
      message += String(*pData, HEX);
    }
    pData = pData - i;
  }

  Serial.println(message);
  message.toUpperCase();
  message.toCharArray(cmessage, length * 2 + 1);

  //Notification filter and publish
  String id = message.substring(2, 4);
  if (id == "A2" || id == "A7" || id == "11" || id == "22" || id == "17") {
    client.publish("am43/notifications", cmessage);
  }

}
```

The *notifyCallback()* function is one of the most important functions in the implementation. The function will transform the byte array sent by the motor into a MQTT message which will be received by Home Assistant.

The operation of the function is to get the bytes with the pointer *pData*, which is going to be incremented and decremented in each cycle. It has to be decremented in order to point to the first value of the memory because the *i* value is relative to that value.

Each byte will be transformed to a string and stored in a variable called message using the built-in function *String(X,HEX).* Once is converted to string, it is transformed to uppercase and converted to a char array; this conversion is needed due to the *client.publish* function which is going to send the MQTT message request this kind of variable.

It has to be mentioned that there is a filter which is going to select the notification that is needed. This has to be done because sometimes there are 3 notifications that are sent responding to the same command. Only one of those notification has to be sent to Home Assistant in order to make the program works correctly.

In the *Figure 28* the aspect of the Arduino IDE Serial Monitor when there is a successful connection can be seen. The first connection is the Wi-Fi (*WiFi Connected*), the second one is the MQTT (*MQTT Connected*) and the last connection is the BLE connection (*We are now connected to the BLE Server*). Finally, some incoming notifications sent by the AM43 motor appear.

```
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1216
ho 0 tail 12 room 4
load:0x40078000,len:9720
ho 0 tail 12 room 4
load:0x40080400,len:6352
entry 0x400806b8
WiFi Connected
MQTT Connected
We are now connected to the BLE Server.
Notify callback for characteristic 0000fe51-0000-1000-8000-00805f9b34fb of data length 5
9a17015a31
Notify callback for characteristic 0000fe51-0000-1000-8000-00805f9b34fb of data length 9
9aa205000000000e33
Notify callback for characteristic 0000fe51-0000-1000-8000-00805f9b34fb of data length 11
9aa7070f146307d00d1088
Notify callback for characteristic 0000fe51-0000-1000-8000-00805f9b34fb of data length 4
9aa80032
Notify callback for characteristic 0000fe51-0000-1000-8000-00805f9b34fb of data length 20
9aa9100000000110000000001000011000000000022
```

Figure 28. Detail of a successful connection messages in the Arduino IDE Serial Monitor

After explaining the connection, it can be seen that when the ESP32 microcontroller establish a connection with the AM42, this connection is going to be active until the AM42 runs out of battery. A battery life test was made in order to know how much time the system could be working. This test is presented in the *Appendix II.* In this appendix, a fragment of the logs that were created in order to monitor the notifications is shown. The AM43 motor could be working for 16 days. It is recommended to use the solar panel that the company offers in order to let the user forget about the fact of charging the device.

4. Installation (HACS)

In the present chapter, it will be explained how the integration is installed.

One of the main advantages of using open source software is the capability to share content with the community. In the case of Home Assistant, there is an UI that allows the user to handle downloads of custom elements that other users develop for Home Assistant. The UI is called HACS (Home Assistant Community Store)[13].

This was the selected way to share the project. The publishing process was very intuitive thanks to the support that HACS gives in its page[14]. In this page appears the conditions that have to be followed in order to make the integration appear in the HACS UI.

For being able to publish the project via HACS, it was necessary to create a GitHub repository in which all the files and an explanation about how to install them (README.md file) were included.

The repository created for this project is:

https://github.com/juagarh5/blind-control-card

Once the repository was created, a json file called *hacs.json* with the information that appears as follows:

```
{
  "name": "blind-control-card",
  "content_in_root": true,
  "render_readme": true,
  "filename": "blind-control-card.js"
}
```

Basically, with that information, it is being configured, firstly, the name in which the integration is going to appear in HACS (*name*). Secondly, whether the file which is going to be copied in the user's disk is either in the root of the repository or in another path (*content in root*). Thirdly, if it is wanted to be shown the readme file which was created in the repository in HACS as the main information for the user (*render_readme*). Finally, the name of the file that is going to be moved.

Once all the files are included in the repository, the integration will be appeared in HACS, and the user is going to be able to install it. Once installed, the *.js* file will be copied into the www folder. This folder contains media content that is used by Home Assistant, so, the user will only have to follow the instructions written in the README file.

[13] https://hacs.xyz
[14] https://hacs.xyz/docs/publish/start

The *figure 29* shows the preview that HACS gives to the user. The option render_readme was set true in the *hacs.json* file so this preview is a README.md render.

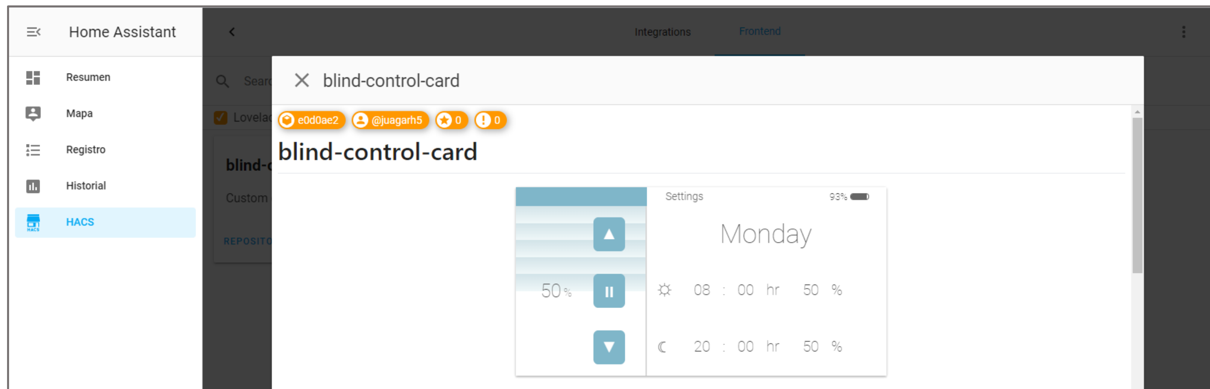


Figure 29. Detail of the integration information in HACS

Finally, it was necessary to publish the content into a community forum (Discord, Home Assistant Web Site, etc.) to let the community know that they have a new integration.

5. Conclusion

Once the project is finished, it can be confirmed that all the established objectives are accomplished.

An implementation of the AM43 motor was introduce in Home Assistant to be run in every kind of devices. For reaching this purpose, communication protocols such as BLE or MQTT and program languages such as Java Script, C++, yaml, etc. have been used. In addition, a first contact with Web Components was done. Multiples conclusions can be extracted at the end of this thesis:

Firstly, the knowledge of the features used independently is important, however, a combination of them, will give the developer the capability to make an integration of similar devices that work in a similar way with the AM43 motor (BLE). If the methodology of this thesis is followed, the integration will be able to run in any Home Assistant server, not only in Raspberry Pi based servers.

Secondly, it can be noted that because of the fact that the code is uploaded in an open-source community, the code can be modified by users. Each user could add new functionalities into the application in order to make it more useful. In addition, personally, as the author, I will be able to modify the code in order to keep the community updated, introducing new features such as: capability of connecting more motors in the same ESP32 or better battery life changing the connection method.

Finally, the main objective was accomplished, since the integration into a new interconnection platform following the Iot philosophy and leaving behind the previous independent application was achieved. The user's experience with this new operation will be increased thanks to using the application in Home Assistant and not in the *Blind Engine* app.

6. Bibliography

Cambridge dictionary. *Definition of reverse engineering*. Retrieved from https://dictionary.cambridge.org/es-LA/dictionary/english/reverse-engineering

Arduino. (n.d.). Retrieved from https://www.arduino.cc

Arduino Client for MQTT. (n.d.). Retrieved from https://pubsubclient.knolleary.net/api.html

Home Assistant. (n.d.). Retrieved from https://www.home-assistant.io.

Home Assistant Community Store. (n.d.). Retrieved from https://hacs.xyz/

Jinja template. (n.d.). Retrieved from https://jinja.palletsprojects.com

Lit Element. (n.d.). Retrieved from https://lit-element.polymer-project.org

nkolban's Repository. (n.d.). Retrieved from https://github.com/nkolban/ESP32_BLE_Arduino

Random Nerd Tutorials. (n.d.). Retrieved from https://randomnerdtutorials.com

Random Nerd Tutorials. (n.d.). Retrieved from https://randomnerdtutorials.com

7. Appendix I: HTML5 code

7.1. Controller mode

```
<head>
  <style>
    div#blind_int {
      position: absolute;
      top: 10%;
      width: 35%;
      height: 90%;
      height: ${this._setBlind()[0]};
      background: repeating-linear-gradient(
        #d3e4e9,
        #f0f8f9 ${this._setBlind()[1]}
      );
    }
    div#pile_int {
      height: 100%;
      width: ${battery + "%"};
      border-radius: 15px 0px 0px 15px;
      float: left;
      background-color: #666666;
    }
    div#schedule {
      position: absolute;
      top: 10%;
      left: 35%;
      width: 65%;
      height: 90%;
      display: inline-block;
      color: #4d4d4d;
      opacity: ${s + 0.4};
      font-size: 1rem;
    }
  </style>
</head>
<div id="card">
  ${this._retryConnectionGraphics()}
  <div id="interface">
    <div id="top">
      <div id="top_blind"></div>
      <div id="settings" @click="${() => this._openSettings()}">
        <settings_logo>Settings</settings_logo>
      </div>
```

```
      <div id="battery">
        ${battery}%
        <div id="pile_ext"><div id="pile_int"></div></div>
      </div>
    </div>
    <div id="blind_int"></div>
    <div id="blind_ext">
      <div id="position">
        <div class="division33">
          <arrow>${this._setArrows()}</arrow>
        </div>
        <div class="division33">
          <div id="number">
            ${this._fixPosition(this._fixPosition(position))}
          </div>
          <div id="percentage">%</div>
        </div>
        <div class="division33">
          <arrow>${this._setArrows()}</arrow>
        </div>
      </div>
      <div id="buttons">
        <div class="division33">
          <button
            class="buttonact"
            id="goup_button"
            @click="${() =>
              this._sendCommand("00FF00009A0A01DD4C", "c")}"
          >
            ▲
          </button>
        </div>
        <div class="division33">
          <button
            class="buttonact"
            @click="${() =>
              this._sendCommand("00FF00009A0A01CC5D", "c")}"
          >
            II
          </button>
        </div>
        <div class="division33">
          <button
            class="buttonact"
            id="godown_button"
            @click="${() =>
              this._sendCommand("00FF00009A0A01EE7F", "c")}"
          >
```

```
            ▼
          </button>
        </div>
      </div>
    </div>
    <div id="schedule">
      <div class="schedule1">
        <div id="day_selector"></div>
        <div id="day">${week[d]}</div>
        <div id="day_selector"></div>
      </div>
      <div class="schedule${this._checkSchedule("m")}">
        <div class="logo">☼</div>
        <div class="time">
          <div class="selector"></div>
          <div class="schinfo" id="mh">
            ${this.hass.states["input_text.schedule"].state.slice(
              14 * d + 2,
              14 * d + 4
            )}
          </div>
          <div class="selector"></div>
        </div>
        <div id="separator">:</div>
        <div class="time">
          <div class="selector"></div>
          <div class="schinfo" id="mm">
            ${this.hass.states["input_text.schedule"].state.slice(
              14 * d + 4,
              14 * d + 6
            )}
          </div>
          <div class="selector"></div>
        </div>
        <div id="separator">hr   </div>
        <div class="time">
          <div class="selector"></div>
          <div class="schinfo" id="mm">
            ${this._fixPercentage(
              this.hass.states["input_text.schedule"].state.slice(
                14 * d + 6,
                14 * d + 8
              )
            )}
          </div>
          <div class="selector"></div>
        </div>
        <div id="separator">%</div>
```

```
      </div>
      <div class="schedule${this._checkSchedule("n")}">
        <div class="logo">☾</div>
        <div class="time">
          <div class="selector"></div>
          <div class="schinfo" id="nh">
            ${this.hass.states["input_text.schedule"].state.slice(
              14 * d + 9,
              14 * d + 11
            )}
          </div>
          <div class="selector"></div>
        </div>
        <div id="separator">:</div>
        <div class="time">
          <div class="selector"></div>
          <div class="schinfo" id="nm">
            ${this.hass.states["input_text.schedule"].state.slice(
              14 * d + 11,
              14 * d + 13
            )}
          </div>
          <div class="selector"></div>
        </div>
        <div id="separator">hr   </div>
        <div class="time">
          <div class="selector"></div>
          <div class="schinfo" id="n%">
            ${this._fixPercentage(
              this.hass.states["input_text.schedule"].state.slice(
                14 * d + 13,
                14 * d + 15
              )
            )}
          </div>
          <div class="selector"></div>
        </div>
        <div id="separator">%</div>
      </div>
    </div>
    <div id="message"></div>
  </div>
</div>
```

7.2. Settings mode

```
<head>
  <style>
    div#pile_int {
      height: 100%;
      width: ${battery + "%"};
      border-radius: 15px 0px 0px 15px;
      float: left;
      background-color: #666666;
    }
    div#schedule {
      position: absolute;
      top: 10%;
      left: 35%;
      width: 65%;
      height: 90%;
      display: inline-block;
      color: #4d4d4d;
      opacity: ${s + 0.4};
      font-size: 1rem;
    }
    div#schtoggle {
      height: 11px;
      width: 11px;
      border: 1px solid white;
      border-radius: 3px;
      background-color: #dadada;
    }
  </style>
</head>

<div id="card">
  ${this._retryConnectionGraphics()}
  <div id="interface">
    <div id="top">
      <div id="top_blind"></div>
      <div id="settings" @click="${() => this._openSettings()}">
        <settings_logo>Controller</settings_logo>
      </div>
      <div id="battery">
        ${battery}%
        <div id="pile_ext"><div id="pile_int"></div></div>
      </div>
    </div>
    <div id="setting_menu">
      <div id="setting_option0"></div>
```

```
      <div
        id="setting_option${localStorage.getItem(options[0])}"
        @click="${() => this._manageSettingsOptions(0)}"
      >
        Schedule
      </div>
      <div
        id="setting_option${localStorage.getItem(options[1])}"
        @click="${() => this._manageSettingsOptions(1)}"
      >
        Velocity
      </div>
      <div
        id="setting_option${localStorage.getItem(options[2])}"
        @click="${() => this._manageSettingsOptions(2)}"
      >
        Direction
      </div>
      <div
        id="setting_option${localStorage.getItem(options[3])}"
        @click="${() => this._manageSettingsOptions(3)}"
      >
        Upper limit
      </div>
      <div
        id="setting_option${localStorage.getItem(options[4])}"
        @click="${() => this._manageSettingsOptions(4)}"
      >
        Bottom limit
      </div>
      <div
        id="setting_option${localStorage.getItem(options[5])}"
        @click="${() => this._manageSettingsOptions(5)}"
      >
        Factory reset
      </div>
    </div>
    ${this._manageSettingsGraphics()}
    <div id="message"></div>
  </div>
</div>
```

7.3. CSS styles

```
button:focus {
  outline: none;
}
div#card {
  font-family: Roboto, sans-serif;
  font-weight: 100;
  position: relative;
  height: 250px;
  background-color: white;
  box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.14),
    0 1px 5px 0 rgba(0, 0, 0, 0.12), 0 3px 1px -2px rgba(0, 0, 0, 0.05);
  display: block;
  user-select: none;
}
div#interface {
  position: absolute;
  height: 100%;
  width: 100%;
  display: block;
}
div#retry {
  position: absolute;
  top: 25%;
  width: 100%;
  height: 60%;
  z-index: 100;
  background-color: white;
  border-top: 1px solid #d8d8d8;
  border-bottom: 1px solid #d8d8d8;
  backdrop-filter: blur(10px);
}
div#top {
  position: absolute;
  height: 10%;
  width: 100%;
}
div#top_blind {
  height: 100%;
  width: 35%;
  float: left;
  display: flex;
  justify-content: center;
  align-items: center;
  background-color: #7fb6c9;
  border-right: 2px solid #7fb6c9;
}
```

```
div#settings {
  height: 100%;
  width: 20%;
  float: left;
  display: flex;
  justify-content: left;
  align-items: center;
  margin-left: 5%;
}
settings_logo {
  font-weight: 300;
  color: #666666;
  display: flex;
  text-align: left;
  justify-content: center;
  align-items: center;
  text-shadow: 0px 0px 3px #ffffff;
}
div#battery {
  font-size: 13px;
  height: 100%;
  width: 20%;
  float: right;
  display: flex;
  justify-content: center;
  align-items: center;
}
div#pile_ext {
  height: 28%;
  width: 23%;
  border: 1px solid rgba(0, 0, 0, 0.6);
  border-radius: 15px;
  margin-left: 4px;
}
div#blind_ext {
  position: absolute;
  top: 10%;
  width: 35%;
  height: 90%;
  border-right: 2px solid #d8d8d8;
}
div#position {
  position: absolute;
  height: 100%;
  width: 60%;
  display: inline-block;
  font-size: 1rem;
}
```

```
div#number {
  float: left;
  width: 60%;
  font-size: 25px;
  text-align: right;
}
div#percentage {
  float: left;
  width: 40%;
  font-size: 15px;
  text-align: left;
  margin-top: 5px;
  margin-left: 3px;
  opacity: 0.7;
}
arrow {
  display: flex;
  justify-content: center;
  align-items: center;
  width: 100%;
  height: 100%;
  font-size: 20px;
  text-align: center;
  vertical-align: bottom;
  opacity: 0.4;
}
div#buttons {
  position: absolute;
  left: 60%;
  width: 40%;
  height: 100%;
  display: inline-block;
  font-size: 1rem;
}
.division33 {
  height: 33.333333%;
  display: flex;
  align-items: center;
  justify-content: center;
  font-size: 20px;
  text-align: center;
  color: #4d4d4d;
}
.division20 {
  height: 20%;
  display: flex;
  align-items: center;
  justify-content: center;
```

```
  font-size: 25px;
  text-align: center;
  color: #4d4d4d;
}
.buttonact {
  font-size: 20px;
  text-align: center;
  width: 45px;
  height: 45px;
  border: 0px solid;
  border-radius: 15%;
  color: white;
  background-color: #7fb6c9;
  margin-right: 40%;
  transition: opacity 1s;
}
.schedule1 {
  height: 33%;
}
.schedule0 {
  height: 33%;
  opacity: 0.5;
}
div#day_selector {
  font-size: 35px;
  height: 100%;
  width: 20%;
  display: flex;
  justify-content: center;
  align-items: center;
  float: left;
  user-select: none;
  opacity: 0.7;
}
div#day {
  height: 100%;
  font-size: 35px;
  width: 60%;
  display: flex;
  justify-content: center;
  align-items: center;
  float: left;
  user-select: none;
}
.logo {
  width: 10%;
  height: 100%;
  font-size: 20px;
```

```
  float: left;
  display: flex;
  justify-content: left;
  align-items: center;
  user-select: none;
  margin-left: 5%;
}
.time {
  height: 100%;
  width: 17%;
  float: left;
}
.selector {
  height: 35%;
  width: 100%;
  display: flex;
  justify-content: center;
  align-items: center;
  opacity: 0.7;
}
.schinfo {
  font-size: 20px;
  height: 30%;
  width: 100%;
  display: flex;
  justify-content: center;
  align-items: center;
}
div#separator {
  font-size: 20px;
  height: 100%;
  width: auto;
  float: left;
  display: flex;
  justify-content: center;
  align-items: center;
}
div#setting_menu {
  position: absolute;
  top: 10%;
  height: 90%;
  width: 35%;
  border-right: 2px solid #d8d8d8;
}
div#setting_option0 {
  height: 10%;
  width: 95%;
  display: block;
```

```
  font-weight: 400;
  text-align: right;
  color: #aaaaaa;
  margin-bottom: 4.5%;
}
div#setting_option1 {
  height: 10%;
  width: 95%;
  display: block;
  font-weight: 700;
  text-align: right;
  color: #4d4d4d;
  margin-bottom: 4.5%;
}
div#setting_config {
  position: absolute;
  top: 10%;
  left: 35%;
  height: 90%;
  width: 65%;
}
div#setting_save {
  position: absolute;
  right: 5%;
  bottom: 10%;
  font-weight: bold;
  display: flex;
  justify-content: center;
  align-items: center;
  font-size: 15px;
  text-align: center;
  width: 70px;
  height: 30px;
  border: 0px solid;
  border-radius: 3px;
  color: white;
  background-color: #7fb6c9;
}
div#setting_cancel {
  position: absolute;
  left: 5%;
  bottom: 10%;
  font-weight: bold;
  display: flex;
  justify-content: center;
  align-items: center;
  font-size: 15px;
  text-align: center;
```

```
  width: 70px;
  height: 30px;
  border: 0px solid;
  border-radius: 3px;
  color: white;
  background-color: #c52020;
}
div#setting_set {
  font-weight: bold;
  display: flex;
  justify-content: center;
  align-items: center;
  font-size: 15px;
  text-align: center;
  width: 70px;
  height: 30px;
  border: 0px solid;
  border-radius: 3px;
  color: white;
  background-color: #7fb6c9;
}
.buttonact_limit {
  font-size: 20px;
  text-align: center;
  width: 45px;
  height: 45px;
  border: 0px solid;
  border-radius: 15%;
  color: white;
  background-color: #7fb6c9;
  margin-right: 5%;
  margin-left: 5%;
  transition: opacity 1s;
}
div#message {
  position: absolute;
  top: 20%;
  left: 40%;
  width: 55%;
  height: auto;
  text-align: center;
  font-size: 15px;
  font-weight: 400;
  transition: opacity 1s;
  background-color: #949494;
  border-radius: 5px;
  color: white;
}
```

8. Appendix II: Battery life test

```
blind-control-card.js:797 Connected 19:33
.
.
blind-control-card.js:797 Connected 20:13
blind-control-card.js:896 Position notification 9AA7070F143807D00D10D3
blind-control-card.js:924 Input_text notification: 9156
blind-control-card.js:876 Battery notification 9AA205000000005B66
blind-control-card.js:924 Input_text notification: 9156
blind-control-card.js:797 Connected 20:43
.
.
blind-control-card.js:797 Connected 00:53
blind-control-card.js:896 Position notification 9AA7070F141707D00D10FC
blind-control-card.js:924 Input_text notification: 9123
blind-control-card.js:876 Battery notification 9AA205000000005B66
blind-control-card.js:924 Input_text notification: 9141
blind-control-card.js:797 Connected 01:03
.
.
blind-control-card.js:797 Connected 02:23
blind-control-card.js:896 Position notification 9AA7070F142C07D00D10C7
blind-control-card.js:924 Input_text notification: 9144
blind-control-card.js:876 Battery notification 9AA205000000005A67
blind-control-card.js:924 Input_text notification: 9044
blind-control-card.js:797 Connected 02:43
.
.
blind-control-card.js:797 Connected 11:23
blind-control-card.js:896 Position notification 9AA7070F144607D00D10AD
blind-control-card.js:924 Input_text notification: 8870
blind-control-card.js:876 Battery notification 9AA205000000005865
blind-control-card.js:797 Connected 16:25
.
.
blind-control-card.js:797 Connected 21:45
blind-control-card.js:876 Battery notification 9AA20500000000566B
blind-control-card.js:924 Input_text notification: 8677
```