Daniel Xavier Silva Vieira

Sistema automático de controlo de transferência de líquidos

Universidade de Aveiro

2021

Daniel Xavier Silva Vieira

Sistema de Controlo de transferência de líquidos automático

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica de António José Ribeiro Neves, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e da Doutora Lúcia Bilro, administradora e diretora de produção e desenvolvimento na Watgrid.

Dedico este projeto à minha família e amigos por todo o apoio que me deram ao longo do meu percurso académico.

o júri / the jury

president / president	Prof. Doutor Telmo Reis Cunha Professor Associado da Universidade de Aveiro
vogais / examiners committee	Prof. Doutor Paulo José Cerqueira Gomes da Costa Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto
	Prof. Doutor António José Ribeiro Neves Professor Auxiliar da Universidade de Aveiro

agradecimentos / acknowledgements

Em primeiro lugar, gostaria de agradecer aos meus orientadores, Professor Doutor António Neves e Doutora Lúcia Bilro, administradora e diretora de produção e desenvolvimento na Watgrid, pela orientação e dedicação que proporcionaram. Obrigado por toda a disponibilidade que mostraram ao longo de todo o projeto, pela tolerância que tiveram aos meus erros e pela forma que sempre me trataram. Foi um percurso com altos e baixos no qual gostei de vos ter ao meu lado. Desejo a ambos a maior das felicidades.

Um enorme obrigado aos engenheiros Fábio Gonçalves, Licínio Malheiro e Pedro Costa da Watgrid pelo apoio, esclarecimento e tempo que me disponibilizaram em todos os desafios deste projeto.
Também quero muito agradecer ao designer Ricardo Pereira pela dedicação que demonstrou e pela paciência de me ensinar as bases da modelação 3D.
Agradeço ainda a toda a equipa da Watgrid, que me acolheu muito agradavelmente, pela simpatia e dedicação demonstrada em qualquer uma das minhas questões.
Obrigado a todos por tudo o que me proporcionaram.

Obrigado a todos os meus amigos por terem estado ao meu lado nos bons e maus momentos, e que me ajudaram muito a superar esta etapa. Um especial "obrigado" ao Diogo Gouveia que me deixou com o braço ao dependuro.

Por último, mas nunca com menor importância, quero agradecer à minha família por todo o apoio, carinho e esforço que me proporcionaram ao longo de todos estes anos, sem eles nada disto seria possível.

palavras-chave Automação, sistema de trasfega, Programmable Logic Controller (PLC)

resumo

O projeto descrito neste documento surge da necessidade da automatização de sistemas de transfega que a empresa Watgrid encontrou na indústria vinícola, e ainda no seu processo de produção. O documento descreve a implementação do controlo de um sistema composto por electroválvulas, bombas de transfega e sensores de nível de líquido, através de um PLC que comunica via WiFi.

O projeto consistiu no desenvolvimento de uma maquete representativa de uma estrutura com quatro recipientes interligados por tubos com as devidas unidades de atuação propriamente instaladas. Para tal foi realizado o levantamento de pré-requisitos funcionais e das soluções comerciais existentes para os diversos componentes. Após o desenvolvimento conceptual da estrutura física da maquete, foram implementadas as soluções encontradas para os módulos que simulam a operação das unidades de atuação, até à estrutura que integra esses módulos. Também foi desenvolvida a unidade de software do PLC para controlo nas diversas etapas da implementação do sistema. Ainda é referida a implementação de uma interface de utilizador alocada num servidor web criado no módulo de WiFi do PLC.

O intuito da maquete reside no pressuposto de replicar uma aplicação direta num sistema real. Desta forma, foi possível implementar, progressivamente, um algoritmo de controlo para o PLC que reproduz o processo pretendido, e realizar testes funcionais ao sistema. Finalmente, foram registados e analisados os resultados de testes ao funcionamento normal do sistema e em situações de falhas imprevistas que pudessem comprometer a integridade do processo de trasfega.

keywords Automation, racking system, Programable Logic Controller (PLC)

abstract

The project described in this document arises from the need to automate racking systems that the Watgrid company has found in the wine industry, and in its production process. The document describes the implementation of the control system composed by electrovalves, transfer pumps and liquid level sensors, through a PLC which communicates through WiFi.

The project consists in the development of a mock-up representing a structure with four containers connected by tubes, with the proper actuation units properly installed. To this end, a survey of functional prerequisites and existing commercial solutions for the various components was carried out. After the conceptual development of the physical structure of the mock-up, the solutions found for the modules that simulate the operation of the actuating units were implemented, up to the structure that integrates these modules. A PLC software unit was also developed to control in the various stages of the system's implementation. It is also mentioned the implementation of an user interface allocated on a web server created in the PLC WiFi module.

The purpose of the mock-up lies in the assumption of replicating a direct application in a real system. Thus, it was possible to progressively implement a control algorithm for the PLC that reproduces the intended process and perform functional tests on the system. Finally, test results were recorded and analysed for the normal operation of the system and in situations of unforeseen failures that could compromise the integrity of the racking process.

Index

Figure list

Table list

1. Introduction

Nowadays, most of the processes responsible for liquid transfers between containers, such as wine racking, tank truck filling, inventory, allotment, among others, are not totally automated and require the supervision of a human, who is able to monitor and optimize these processes. Thus, the performance and precision achieved in liquid transfer systems is limited and considerably prone to human errors. That can be critical for a system that requires some level of accuracy. Any flaw can compromise the integrity and quality of a certain liquid, causing the company to incur on a financial loss equal to the product's market value.

1.1 *Objectives*

This project aims to develop a liquid transfer system capable of real time monitoring with several sensors that can indicate transfer state variables such as container capacity, liquid-filled volume of the container, and transfer's flow. All information collected by the sensors should be transmitted to a control unit, where it will be examined. Afterwards, the actuation units, responsible for the mechanical transfer operation, are activated in a specific order depending on the application. The system parameters can be specified by the user through an interface that will have to be developed.

1.2 *Company presentation*

This project was developed in the company *Watgrid*, a company based on the development of solutions for monitoring indicative parameters that state the quality and integrity of different types of liquids.

Nowadays, the main focus of *Watgrid* remains in the wine production market, through the brand *Winegrid*, with solutions able to monitor the wine quality in the different winemaking processes, namely fermentation and aging. Their product becomes particularly useful in industrial cellars, where thousands of litters of wine are produced and stored in wine tanks. The solution monitors the wine/must directly inside the vats and send the data wirelessly to a webserver. This data can be accessed through a web-based platform making it easier to check a required parameter instead of having sparse manual readings. In Figure 1 is shown a diagram that represents this solution.

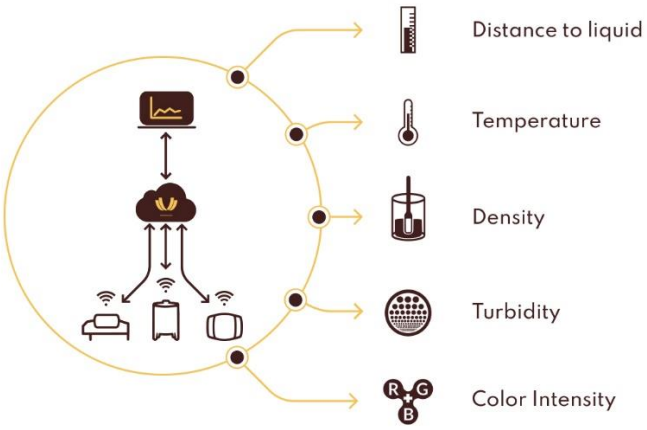


Figure 1: Diagram of Watgrid's solution [1]

Besides the development, production, and setup of the products, Watgrid also binds in giving its costumers top quality assistance.

1.3 *Project framework with Watgrid*

As the company works around the winery industry, developing a system capable of automatically transfer liquids would be an interesting opportunity to introduce this system into the market as a new product line. A fully developed racking system would be an optimized way to fill the need felt by some wineries when it comes to exchange the wine between containers, providing a chance to optimize human resources and to low down the possibility of human error.

On the other hand, the sensors produced by Watgrid must be calibrated. For that purpose, Watgrid possesses a system composed of five storage tanks containing different liquids and seven vats, all connected in series through a pipe structure. The vats are filled with the different liquids straight from the mimicking different wine fermentation states. Therefore, a structure like this one works perfectly as a test bed when developing the system meant to be implemented in this project.

Furthermore, the preparation of this calibration liquids and the complex liquid transfer circuit is operated manually and prone to errors leading to a waist of materials and a financial loss to the company. Therefore, developing a system capable of performing this process automatically would lead to fewer losses of product and, consequently, reduced financial loss, as well as an increased productivity, since the system could be operated remotely.

To develop this project in Watgrid allowed the development of new skills, the introduction to the industrial strand and also to dynamics of the daily work in a company.

1.4 *Project outline*

The developed project is a mock-up of a system comprised of a monitoring unit, a control unit, an actuating unit, and a user interface. The monitoring unit includes sensors that measure the liquid level in the containers and the valves' feedback signals that will then be transmitted to the control unit.

The control unit should be able to read the sensor data, process it and then generate a signal that will activate the actuating unit's operation in a certain way. Since there is a need to follow some industrial requirements, the implementation of this unit was based in programmable logic controller (PLC).

As for the actuating unit, racking pumps and electrovalves were used as actuators, thus responding directly to the PLC to pump the liquid and set the flow path between containers, respectively.

Finally, the complete system can be accessed and manipulated through an interface allocated on the company's server. A draft of the project architecture is presented in Figure 2.

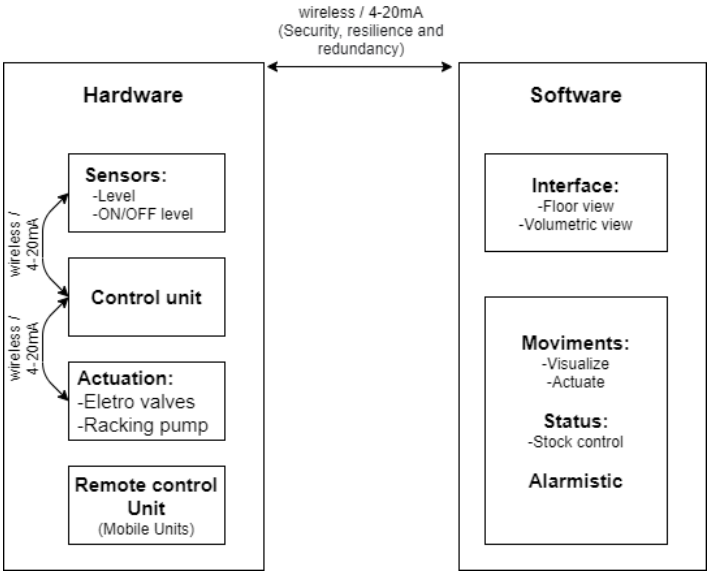


Figure 2: Transfer system scheme.

1.5 *Document outline*

The document is organized as follows:

- **Chapter 2**: It begins by explaining the project's imagined structure, followed by an investigation concerning all specifications of the main components used and their benchmarking. It ends by addressing the solution that will be developed.
- **Chapter 3**: Describes the implementation process of the hardware modules.
- **Chapter 4**: Describes the implementation process of the software modules.
- **Chapter 5**: Consists in the demonstration and verification of the obtained system's functionality through tests in different scenarios.
- **Chapter 6**: States the conclusions obtained at the end of the project, including future considerations.

2. Project Requisites

As mentioned before, the company has built a system with five tanks and seven vats connected through a pipe structure. It is an ideal test bench for the purpose of this project. This chapter will describe how the system is arranged and specify its working process.

The tanks are placed on the room's ground level, each one with the capacity of a thousand litters. Above the tanks, a platform was built at halfway the room's height, in order to support seven vats (six of them with a capacity of three hundred litters and one of them with a thirteen hundred litter capacity), which are used to blend the compounds that originate the wine-simulating solution and where the calibrations will take place.

The goal of this system is to transfer one tank's liquid into one vat at a time, or the other way around. To do that, the first need is to define a path for the liquid to flow from/to the desired containers without deviation. That was accomplished with the application of valves in the pipe system. These valves prevent the blending of liquids from different containers and isolate the transfer between two containers. These valves can be set into two different positions to let the liquid into two different paths:

- The resting position (when the valve is blocking the way to the assigned container and letting the way through the main pipe opened for the other containers);
- The flowing position (when a container is selected to receive or send liquid, the valve is activated to set the path into that container and blocking the way to the consecutive recipients).

Figure 3 illustrates this behavior in a system with two vats and two tanks, where the direction imposed to the liquid is represented by the valves' symbols.

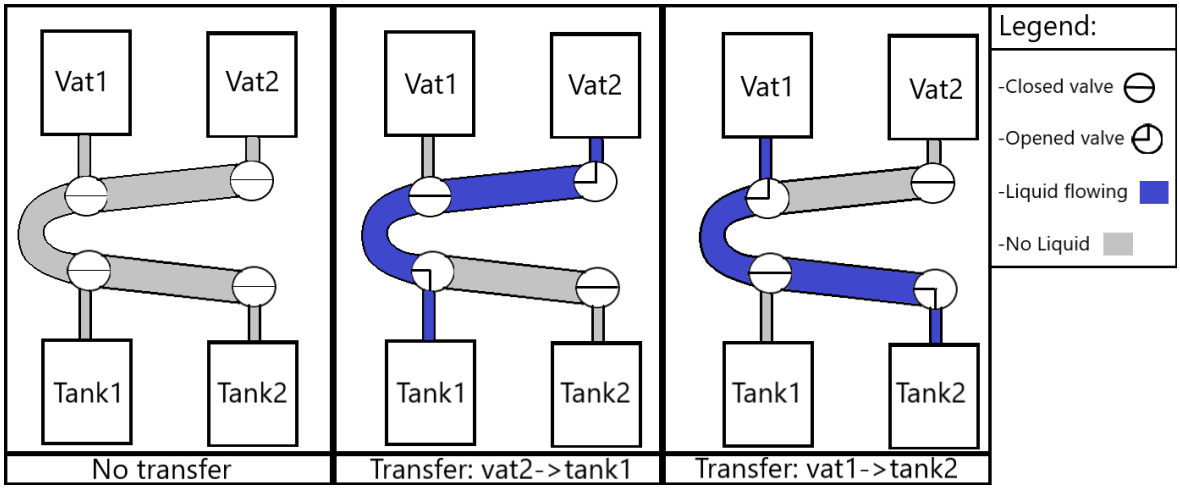


Figure 3: Liquid path according to valve positioning.

Since the vats are placed at a higher level, whenever there is a need to drain them out, the liquid will flow down the pipes just by gravity. On the other hand, when the reverse process must be fulfilled, the liquid must be

pumped from the tanks through the pipe system, all the way up to the vats. The racking pumps are responsible for this action and there is a pump installed for each tank.

After every liquid transfer, the liquid residues that still exists at the pipes must be drained before the next transfer happens to avoid unwanted contamination between solutions. Therefore, a purging method was implemented. This method consists in having a valve applied at the end of the pipe system (and connected to open air). When a transfer ends, the purging valve opens, releasing the pumping-resulting pressure and making the liquid drain out from the pipes into the utilized tank. In addition, a non-returning valve is used after every pump. In its resting position, this valve is capable of both letting liquid pass through in one direction and blocking the reverse direction. Furthermore, the valves will only open only when there is a need to let liquid flow into a tank, i.e. transfers from vats to tanks and purging.

Figure 4 shows part of the system mounted on the company. In the left, some of the tanks connected to the pipes and respective electro valves and pumps are shown, as well as the control panel. On the right side, is a picture of some vats (with sensors attached) on the platform, above the tanks, that are also connected to the pipes.

Figure 4: System on the company.

As for the approach used by the company, an ESP module (WiFi module) was integrated to receive the orders from a webserver and the signal to the actuators and to control the electrovalves and the racking pumps.

The ESP module lacks in terms of peripherals, and so, there are fewer than those required to control all the actuators. To solve this issue, several GPIO expanders (a module that increases the number of available peripherals) were connected in series to the ESP through I2C. Furthermore, the actuator's activation signals require a greater amount of power supply than the ESP outputs provide, so, instead of directly signaling the actuators, the ESP control relays are connected to a power supply.

2.1 ***Operation***

To assure that the liquid transfer occurs without flaws, there must be an order for the position of each intervening valve. For this purpose, a flowchart was created containing all steps that must be taken for every transfer – they are displayed in Figure 89 depicted in Appendix 5.

The flowchart represents two main paths: one indicating the process needed for a transfer from a tank to a vat and the other for the opposite transfer.

When the user wants to send a solution into a vat, firstly he must open the valves assigned to the chosen vat tank, freeing the path through the main pipe. Considering that only the required valves are opened, the transfer can begin with the activation of the pump.

After the transfer is concluded and the pump is turned off, the liquid must be purged into the tank, so the pipes are clean for the next transfer. In order to undertake the purging, a new path right towards the tank must be set, and so, the vat valve can be closed and the non-returning valve is opened as well as the purging valve, which initializes the purge. After a defined interval of time, all the opened valves are closed, thus terminating the transfer.

As for the transfer from a vat to a tank, the opening the pump and tank valves at the beginning is required, so the liquid can flow into the tank. Then, the vat valve is opened, and the transfer begins with the pump's activation in the right direction.

Once the transfer ends, the vat valve is closed and, as the pump valve is already opened, only the purging valve needs to be opened to drain the liquid into the tank. Next, after a specified delay, all the opened valves are closed. These tasks can be achieved by positioning the actuators one at a time through an interface hosted on the company's webserver.

2.2 *Programable Logic Controller*

A PLC, or programmable logic controller, is a digital computer using easy programming to control systems that require high processing viability and process flaw diagnoses. These systems can be used in manufacturing processes, assembly lines, robotic devices or any other industrial activity that requires a consistent control unit [2]. It is a computer based on a microcontroller with a real time operational system that is built to operate in extreme environmental conditions such as, high temperature, pressure, humidity, among other conditions that an ordinary microcontroller could not handle [2].

PLCs are usually defined by their processing capacity and number of peripherals. They are also ranked as compact or modular, depending on the possibility of attaching extra port modules to control systems with different peripheral requirements. It can read, as well as write, signals of either digital or analogue nature. In Figure 5, an example of a PLC architecture is shown.

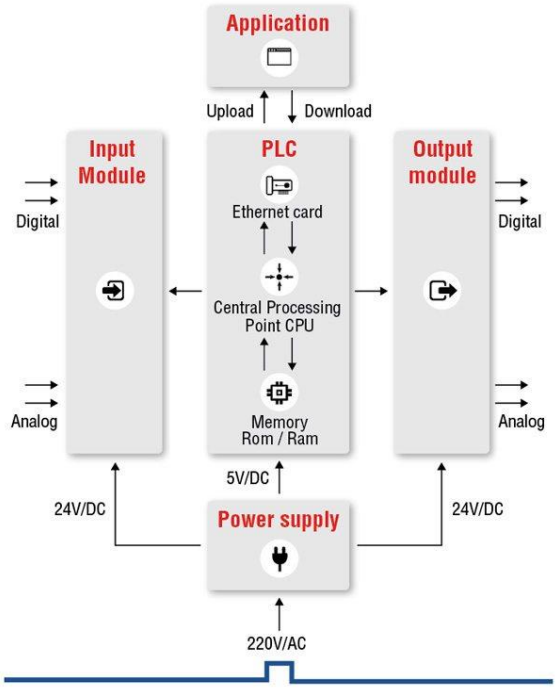


Figure 5: PLC architecture [2].

In short, the benefits mostly considered when acquiring a PLC are [3]:

- Best balance between reliability and cost reduction *per* number of relays and maintenance.
- The immunity to electromagnetic noise because of its isolation structure.
- The programming easiness.
- The flexibility to be used in different scenarios, due to the possibility of changing peripheral specifications through the attachment of I/O modules.
- Real-time and multitasking processing resources, allowing a greater accuracy on task execution.
- The online monitoring system that allows various controllers to keep communicating while an ongoing real time monitoring of the processes takes place.

One of the main specifications of a PLC is the scan cycle. It represents the time interval that a series of operations are carried out by the PLC . It corresponds to the time it takes to execute 1024 Boolean instructions (such duration is generally provided by the producer).

When a PLC is initializing, there is a set of pre-programmed operations awaiting execution before running the main program. Firstly, a status check is performed to the processor unit, the memories, and the auxiliary circuits. Moreover, another check is performed to know if there is an ongoing user program the state of the entire operation. Then, the results obtained from the input peripherals during processing are firstly compared with user-defined instructions and then stored in the memory. The output peripherals are updated with the relevant value thus initiating a new scan cycle.

2.2.1 Market analysis

When it comes to choose a PLC, first it is necessary to understand the several options available in the market.

For this purpose, looking into the market share is a good way to judge the popularity of PLCs. In 2017, Interact Analysis conducted a study on this matter and the top ten brands (i.e. holding more market shares) are listed on Figure 6 [4].

Market Share Ranking	PLC Manufacturers	PLC Brand Name/s
1	Siemens	Simatic
2	Rockwell Automation	Allen Bradley
3	Mitsubishi Electric	Melsec
4	Schneider Electric	Modicon
5	Omron	Sysmac
6	Emerson Electric (GE)	RX3i & VersaMax (GE Fanuc)
7	Keyence	KV & V-8000
8	ABB (B&R Automation)	AC500 X20 & X90
9	Bosch	Rexroth ICL
10	Hitachi	EH & H

Figure 6: List of the 10 most popular PLCs and their manufacturing company, according to market share [4].

From the previous ranking, four manufacturers were chosen to undergo investigation. The first two, *Siemens* and *Rockwell Automation*, and other two, *Omron* and *Schneider Electric*. These brands produce PLCs for many-scaled applications. In this case, the system meant for development was found to fit in medium/small scale ranges. Therefore, the products available are more accessible in terms of costs and, even so, can meet the stated requirements.

Still, these brands belong to the top shareholders of the automation market. The product's architecture is not revealed and the software must be bought for programming the PLC. Thus, an open-source solution was also investigated.

The most reasonable solutions found on the internet were *Industrial Shields*', *Controllino* and *Unipi.*

Industrial Shields have a wide range of products. Their solutions offer the possibility to choose the following: the processor (*Arduino, Raspberry pi* or even *ESP32*), functionalities such as Wi-Fi (through an ESP32-embedded module or an Ethernet port), and GSM/GPRS connectivity. Options for IoT applications are also provided [5].Yet, *Controllino* has a range of products equipped with nothing but Arduino processors of different models (and adapted to the wide application range) [6]. As for *Unipi*, the PLCs are based on Raspberry pi modules, and, like the other two, their products change based on the application range.

Table 1 allows the comparison of the above mentioned PLCs regarding: the type of architecture, the number of available relays and the price. Information is also displayed on each brand's specific peripheral expansion modules.

Table 1: PLC price list

Architecture	Brand	Product name	Peripherals (inputs; outputs; relay outputs)	Price	Link	Expansion module	Peripherals (inputs;outputs;relay outputs)	Price	Link
Open Source	Controllino	CONTROLLINO MEGA	21 ; 24 ; 16	279.00 €	[9]	Another PLC through I2C	Depends on necessity	.	
		CONTROLLINO MAXI Automation	18 ; 18 ; 10	209.00 €	[10]				
	Industrial Shields	M-DUINO PLC Arduino 57AAR	32 ; 25 ; 7	372.75 €	[11]	Another PLC through I2C	Depends on necessity	.	
		M-DUINO PLC Arduino Ethernet 58 PLUS	36 ; 22 ; 0	336.00 €	[12]				
		M-DUINO PLC Arduino Ethernet 53ARR	25 ; 28 ; 15	402.15 €	[13]				
	Unipi	Unipi Neuron L203	36 ; 28 ; 28	469.00 €	[14]	Unipi Extension xS11	12 ; 13 ; 13	158.99 €	[21]
Closed Source	Siemens	SIEMENS SIMATIC S7-1200	14 ; 10 ; 0	367.57 €	[7]	SIEMENS 6ES7223-1BL32-0XB0	16 ; 16 ; 0	267.00 €	[22]
		SIEMENS SIMATIC S7-1200F	16 ; 12 ; 10	822.85 €	[8]				
	Omron	OMRON CP1E-N40S1DR-A	24 ; 16 ; 1	374.78 €	[16]	OMRON CP1W-40EDR	24 ; 16 ; 0	337.20 €	[23]
		OMRON CP1E-N14DR-A	8 ; 6 ; 1	220.59 €	[17]				
	Schneider	Schneider Electric Modicon M221 PLC	24 - 16 - 16	366.62 €	[15]	TM3DM24R	16 ; 8 ; 8	138.37 €	[24]
	Allen Bradley	Micro850	28 ; 20 ; 20	768.83 €	[18]	Allen Bradley Micro 800 PLC I/O Module	16 ; 0 ; 0	158.36 €	[25]
						Allen Bradley Micro 800 PLC I/O Module	0 ; 16 ; 16	193.93 €	[26]
	Automation Direct	C0-00DR-D	8 ; 6 ; 6	75.53 €	[19]	C0-16CDD1	8 ; 8 ; 0	54.84 €	[27]
		C0-11ARE-D	8 ; 6 ; 6	149.24 €	[20]	C0-16CDD1	8 ; 8 ; 0	54.84 €	

The selection of the PLC was done considering the most feasible products regarding price and peripherals for this project. Mainstream brands are generally tagged with a higher cost than the open-source options, and, as the number of peripherals increases, the item becomes less affordable, as expected. The same goes for the expansion modules.

2.3 *Electrovalve*

Electrovalves are valves attached to a mechanical structure that changes their position according to a provided electrical signal and have two main parts. The magnetic head, which mainly includes a coil, a pipe, a breech, an offset ring, a spring, and the body, which includes connection holes, seat, obturator, membrane, piston, among others, according to the type of technology employed [28].

When voltage is applied to the coil, a magnetic field is generated which causes the movable core to be displaced and, consequently, the position of the valve shifts to open or close position, depending on the valve specifications [28]. Figure 7 displays a cross section of an electrovalve example.

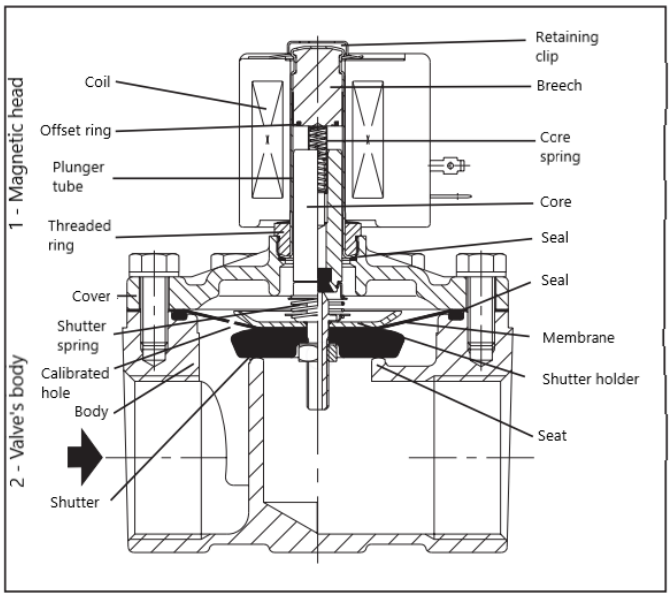


Figure 7: Cross section of an electrovalve example [28].

Some considerations about Figure 7 are the following [28]:

- Offset ring: It is placed on the bottom part of the breech, above the movable core. It provides alternating current to limit vibrations.
- Coil: It is intended to generate a magnetic field.
- Seat: It is used to close the main hole.
- Breech: A metallic mass located on the far end of the pipe, allowing an enhancement of the magnetic field during operation.
- Sealing chamber: It assures ingress of fluid on the valve's body.
- Core: A cylinder displaced by magnetic force.
- Calibrated hole: It ensures the valve is closed through inlet pressure from the membrane or the piston.
- Core Spring: On absence of supply, it imposes the position of the core.

In addition, there are various types of electrovalves, distinguished by their amount of ports (commonly referred as "ways"), their different positions and their operation method.

Electrovalves 2/2 or 2/3 types (number of ways/number of positions) can be driven through direct or indirect command. As for the ones with direct command, the core opens or closes the hole according to the voltage applied to the coil. On the other hand, the operation of the indirect command is defined by the differential pressure between entry and upstream. There are still electrovalves with greater amounts of ways and positions and with distinguished operation from the previous; however, it is not necessary to mention them within the scope of this project. [28]

2.4 *Racking Pump*

Considering reduced liquid volumes, the transfer process can be carried out with the help of gravity (through a method called siphoning) [29]. This method consists in positioning a container, from which the liquid will be excluded, at a higher height level comparing with the container that will receive the liquid. This way, the liquid will be drained only due to the gravitational force. It is a method with a very low implementation cost but also inefficient and time-consuming [29]. That said, when the amount of liquid about to be transferred is considerable, the use of a racking pump is recommended, which makes the process much more efficient [29].

When selecting a pump, one should mind the pumped material's physical structure (liquid, solid or liquid with solid sediments), as there are different types of pumps for different jobs. In the case of wine fermentation, the pump must allow the transfer of liquids with solid particles, but this specification becomes irrelevant when the process is bottling. Another important point to consider is the turbulence imposed to the wine during pumping procedure [29].

In addition, a poor pump assembly can cause an early termination of the transfer due to a gas bubble generated in the pump, which is called cavitation [30]. To prevent cavitation, there are pumps, called self-priming pumps,

with the ability to vacate air on the suction side before pumping starts. This process forces the liquid into the suction line while the air is extracted due to the pressure, until a pressure balance around the liquid is met [30].

In terms of choosing a suitable racking pump, the main technical characteristics to be considered are the following:

- **Indication:** target application: e.g. wine, alcoholic beverages, etc.
- **Capacity / flow:** flow rate (L/h).
- **Filter:** Type of filter used.
- **Rotation type:** Rotation speed (in this case low) to prevent deterioration of the wine.
- **Rotation inversion:** It indicates whether it pumps in both directions.
- **Inlet and outlet diameter:** It specifies the flow width.
- **Self-priming and direction of the vacuum:** Information of the air-evacuation functionality.

2.5 *Conceptual Solution*

Although the system implemented in Watgrid is operational, it is not automated. The valves must be positioned one at a time by the user for further activation of the pumps. This will initiate the transfer and later make the purge happen. At the end, the valves are repositioned into the initial state, ending the transfer process. Another issue to be considered arises from the use of peripheral expander modules. The system has a considerable power consumption and electromagnetic interferences may happen and, consequently occasional signal errors may cause a failure of the system.

2.5.1 Component Selection

Beginning with the PLCs, to shorten the pre-selected ones, we started by choosing only those with an open-source architecture. This is a plus for the company because it allows the study of the hardware and an adaptation to the application requirements. In addition, they have a lower price compared to PLCs with a protected architecture and the programming software is free.

Hereupon, based on the PLC benchmarking, only the *Controllino*, *Industrial Shields* and *Unipi* brands remain. These brands offer products with at least one Ethernet port, which guarantees connectivity. However, of all these three brands, *Controllino* is the only one that does not offer wireless communication methods, such as Wi-Fi or Bluetooth, and was discarded for this reason [31].

The remaining two brands are controlled by different processors (Arduino and Raspberry pi). Therefore, the programming software also differs. *Unipi* PLCs can be configured using various software solutions: *Mervis* (an official supported software platform using SCADA language to programme the *Unipi* controllers [32]), *nodeRed* (an open-source web-based programming tool [33]), among others [34]. On the other hand, *Industrial Shields*' PLCs use the Arduino editor based on C and C ++ languages. Considering these, it was concluded that

the acquisition of an MDUINO would be more workable, because it is more convenient to program in a more familiar programming language.

Now with the brand selected, it is necessary to find out which PLC model is more suitable for the project. For this purpose, one must achieve a correspondence between the controller's price, the number of peripherals and the presence of a WiFi module.

Considering all the aspects mentioned previously, the chosen PLC was the MDUINO 53ARR PLUS with WiFi, shown in Figure 8. This controller has fifteen relays and twenty-five possible digital inputs. Relays will be essential to activate the actuation units and digital inputs will be required to read actuation unit states. The ESP32 module allows wireless operation. The cost was 432.64€ [13].

Figure 8: MDUINO 53 ARR PLUS [13].

The acquired PLC has the following main characteristics [35]:

- 25 inputs: 14 analog (10 bits, 0-10Vdc) / digital (7-24Vdc) configurable by software; 5 digital inputs (7-24Vdc); interrupt inputs (7-24Vdc) that can function as digital inputs.
- 28 outputs: 15 relays (220Vac-5A); 8 analog (10 bits, 0-10Vcd) / digital (5-24Vdc) outputs; 5 opto-isolated digital outputs (maximum 24Vdc).
- Communication: WiFi and Bluetooth with ESP32; Ethernet, USB, I2C, RS-2323, RS-485, SPI ports; TCP IP / Modbus TCP / Modbus RTU.

Bearing in mind that a functional system is already implemented in the company, the remaining components (the electrovalves, the racking pumps and the level sensors) are already available, and, therefore, it is possible to use them without having to acquire new ones.

The pumps are the Rover BE-M 20 from Figure 9. These are self-priming pumps, which work in both directions and are suitable for wines (among other liquids). They have a small solid-pumping percentage [36].

Figure 9: Racking pump and specifications [36].

To define the fluid path, Tonhe A100-T valves are used. These offer three routes, with two possible positions, and provide the user some model flow plans. The ones in the company follow the flow plan B shown in Figure 10.

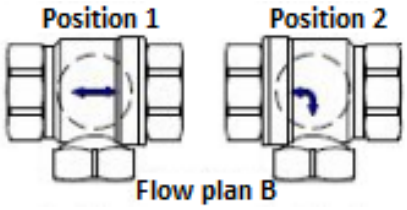


Figure 10: Valve's possible positions with flow plan B [37].

The user will decide how to control and monitor these valves. Several connection circuits are shown in the datasheet, thus providing different ways of interaction. In this application, the CR5 02 diagram, shown in Figure 11, was used, as it can be controlled with an on/off signal and provides a feedback signal for each valve's position.

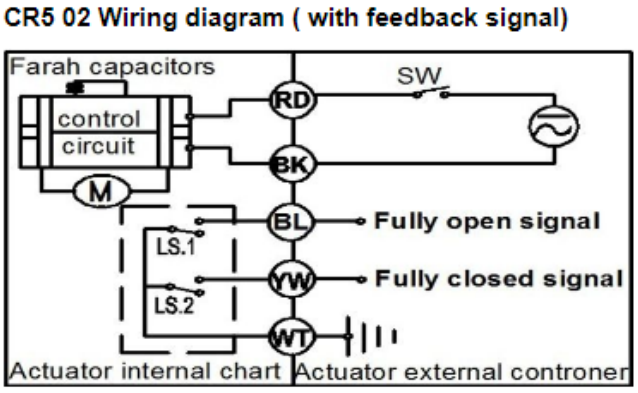


Figure 11: Electrovalve CR5 02 wirig diagram [37].

When assembling the valves with the connection diagram of Figure 11, it is possible to change their position by applying, or not, a voltage on the RD terminal. It is also possible to obtain the status of the valve through two other terminals: one indicating that the valve is open, "Fully open signal" (BL), and the other that it is

closed, “Fully closed signal” (YW). In addition, with this diagram, the operating voltage can be 110-230 V AC or 12-24 V DC [37].

3. Hardware Implementation

3.1 *PLC configurations*

First, it is essential to understand the peripheral dimensions for monitoring all the components of the actuation unit:

- The racking pumps have two activation signals, each one responsible to start the operation in a specific direction, without any feedback signals. Thus, for each pump, two relays would be required.
- The electrovalves responsible for fluid path definition (V 3/2) alternate their position according to the voltage applied to a terminal and indicate their status through two other terminals. For this reason, these valves require one relay and two digital inputs each.
- The electrovalves that control the passage of liquid to each tank (V2/2) have two terminals monitoring their position and, as is the case with the valves mentioned above, return to their state through two terminals. Therefore, they need two control relays and two digital inputs, each.
- The ON / OFF level sensors (Level S.) are activated when they detect liquid and only have a terminal indicating their state: on or off. Therefore, they only need 1 digital input to check the status.

Now, considering all the components used in the system, the number of necessary input and output ports is calculated, and for this purpose, Table 2 was developed.

Table 2: Peripheral estimation

Peripheral estimation							
	Application		Peripherals				
	Tank	Vat	R	DI	Quantity	Total R	Total DI
V 3/2	x	x	2	2	12	24	24
V 2/2	x		1	2	5	5	10
Pump	x		2	0	5	10	0
Level S.		x	0	1	7	0	7
V 3/2 (Purge)	n.a.	n.a.	2	2	1	2	2
					Total	41	43

In short, a total of 41 relays and 43 digital inputs is required. The acquired PLC does not have the required number of peripherals to control the system in the company. This means that later it will be crucial to acquire an input and output expansion module. In the case of Industrial Shields, the only way to increase the number of peripherals is to acquire another PLC and connect it to the main PLC via I2C - one being the slave and the other the master, respectively.

Since both pumps and electrovalves are powered by a 230Vac signal, the PLC cannot act as power supply. An external power source will be needed to supply the components through the relays.

3.2 *Mock-up planning*

The main objective of the project is to implement the PLC on the company's testbed. However, it was concluded that it would be prudent to develop before a test mock-up for safety reasons.

Since the PLC lacks in peripherals, initially, it was necessary to adapt the system. For this purpose, when developing the mock-up, the simulation-targeted system was minimized to two vats and two tanks, thus reducing the number of components in the actuating unit. This way, only six electrovalves, two pumps and two level sensors were considered. Figure 12 represents the schematics of the control system.

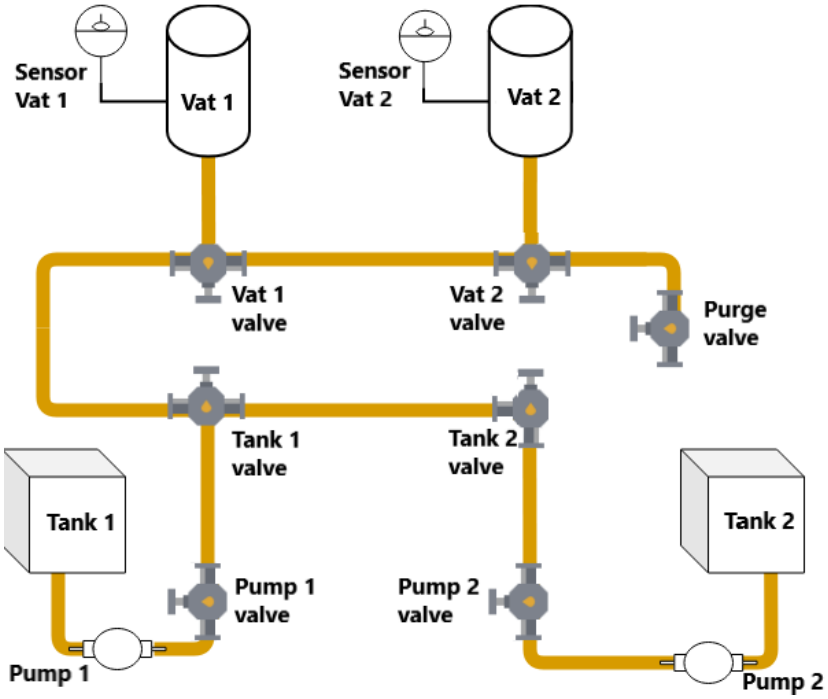


Figure 12: Scheme of the system to be implemented in the mock-up.

To develop this mock-up, it was also necessary to adapt all components of the real system, all elements of the actuation unit, as well as some aspects of their operation. Then, the following sections will describe the carried approach in order to simulate the system.

3.2.1 Modules structure and operation

To simulate the components, representative modules of the electrovalves, pumps and the level sensor/vat set were developed. These modules are a set of elements working together to reproduce an electrical behaviour like that of the system's mechanical components.

Starting with the electrovalves, as already mentioned, two types are being used, but it was considered that, despite having a different operation and goals, both their activation and state reading were implemented in the module similarly (that is, one module, with two control signals and two feedback signals, represents both valves). Therefore, the required peripherals are mentioned Table 3 (now there are two required relays for valves 2/2).

Table 3: Peripheral estimation for adapted system

Peripheral estimation							
	Application		Peripherals				
	Tank	Vat	R	DI	Quantity	Total R	Total DI
V 3/2	x	x	2	2	4	8	8
V 2/2	x		2	2	2	4	4
Pump	x		2	0	2	4	0
Level S.		x	0	1	2	0	2
VP 3/2 (Purge)	n.a.	n.a.	2	2	1	2	2
					Total	18	16

In the Figure 13 it is possible to check both the structure of an electrovalve through the block diagram and its operation with the state diagram. As for the state diagram, it can also be added that the transition between states is not immediate. The mechanical process of the electrovalve takes few seconds and, during this period, the outputs are zero.

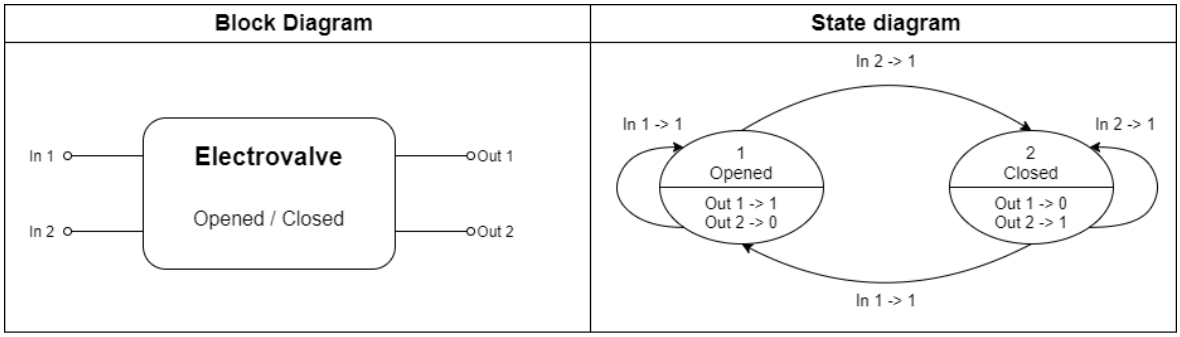


Figure 13: Block and state diagram of the electrovalves implemented in the modules.

Thus, to simulate the operation of a valve, a DC motor (which represents the valve transition) and two limit sensors (which, when pressed, indicate the position of the valve) were considered. An LED was also connected to each sensor in order to allow visual inspection of the position.

The components were assembled in such a way that, at a certain point, the motor rotation activates one of the limit sensors. This arrangement can be achieved with the development of a 3D support structure, which can be produced at the company on two available 3D printers (*Creativity CR-10*).

As the level sensors are installed in the vats, we considered it appropriate to represent the vat with the level sensors attached in a single module, instead of representing just a separate sensor. Therefore, it is possible to use the same module to simulate the variation of the liquid in the vats. In this case, the position of the motor represents the liquid's level (and the limit sensors the level sensors). Thus, the motor would have to be rotating and, when reaching one of the limit sensors, the state of the tank would be indicated: full or empty. The structure would also be the same.

Finally, since transfer pumps do not produce any feedback signal and as there is a shortage of relays, the conclusion was reached that the pumps would be represented by two LEDs, each illustrating the pumping direction. They would be activated by two digital PLC outputs.

3.2.2 Required Components

Since most of the components can be simulated by the same module mentioned above, the development of nine of these modules was considered: two representing the vats, two the pump valves, four the tanks and vat valves, and one the purge. Nine DC motors, eighteen limit sensors and eighteen LEDs, and four more LEDs were used to represent the two remaining pumps.

Starting with the DC motors, these were chosen considering the price and the speed of rotation. As they represent the valves, the rotation speed must be comparable to of the speed of the valve transition. Therefore, they will have to present a relatively low speed: about 180º in 1 to 3 seconds. A benchmarking of DC motors is shown in Table 4.

Table 4: DC motors' specifications and prices.

Name	Shop	Price (€)	Rot/min	sec/90º	delivery fee (€)	Link
FIT0495-A DFROBOT	tme	12.64	15	1	7.9	[38]
82869014 CROUZET	tme	82.98	13	1.15	7.9	[39]
499:1 METAL GEARMOTOR 25DX58L MM LP 6V POLOLU	tme	16.25	11	1.36	7.9	[40]
ROB-12472 SPARKFUN ELECTRONICS INC	tme	33.7	6	2.5	7.9	[41]
Motorreductor DC con escobillas RS PRO, 7 rpm	rs	16.58	7	2.14	free	[42]
Motorreductor DC con escobillas RS PR6 rpm	rs	17.72	6	2.5	free	[43]
JGY370 Reversible High Torque Worm Geared 12V 6rpm	ptrobotics	14.88	6	2.5	4.67	[44]
1000:1 Micro Metal Gearmotor	ptrobotics	25.95	14	1.07	4.67	[45]
Cytron 12V 17RPM 194.4oz-in Spur Gearmotor	robotshop	11.93	16.7	0.90	10.9	[46]

From the previous table, the motor *FIT0495-A DFROBOT* from TME with a speed of 15 r.p.m. was selected. This was the chosen motor because the company places several orders at TME, which mitigates the shipping cost. It is also the one with the better price-to-speed ratio. It can be powered at between 2 to 7.5 Vdc, has a working current of 50 mA and turns 180º in 2 seconds.

When selecting sensors and LEDs, only TME was the brand of choice. 10 mm green and red LEDs were chosen to illustrate the valves' two states. The green LED works on a voltage of 1.7 V to 2.8 V and a current of 20 mA [47], and the red one between 1.6V and 2.6V at the same current [48]. The sensors, on the other hand, allow a maximum voltage of 250Vac and a maximum current of 1 A [49]. Figure 14 contains pictures of the acquired components.

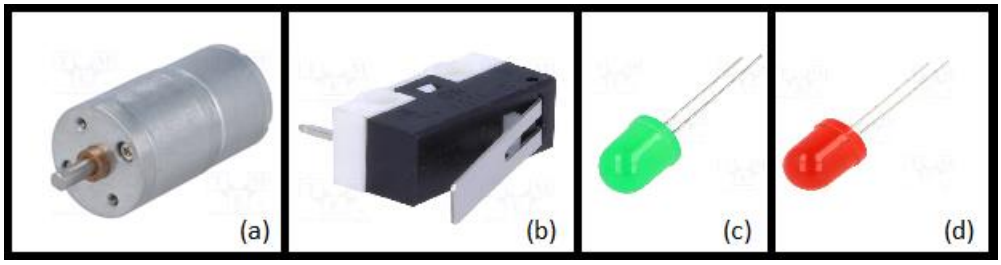


Figure 14: Acquired components: (a) - DC Motor [38]; (b) - Limit switch [49]; (c) - Green LED [47]; (d) - Red LED [48].

DC motors have two terminals and the rotation direction is dependent on the supply bias. Therefore, it is easy to conclude that the terminals of the electrovalves do not match to those of the motor. One way to solve this problem is to use an H bridge, which allows the current to flow in both directions. For this purpose, the ZXMHC10A07N8 H-bridge integrated circuit from *Diodes Incorporated* was selected. It is shown in Figure 15.

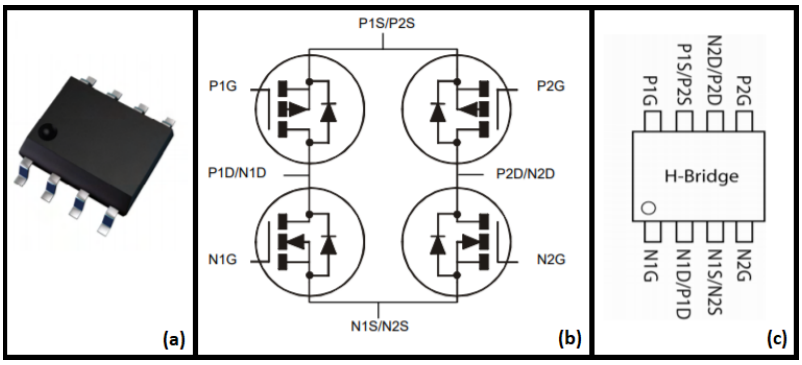


Figure 15: H-bridge integrated circuit: (a) - component; (b) - circuit; (c) - pinout. [50].

Finally, in Table 5 of necessary components is shown below. Note that there are resistances available at the company and the ones in the table are a similar component from *TME* that were selected in order to estimate the total price of the components.

Table 5: Components' price table.

	TME Components List				
Component	Name	Reference	Price (€)	Quantity	Total (€)
DC motor	FIT0495-A DFROBOT	[38]	12.64	9	113.76
H-bridge	ZXMHC10A07N8	[51]	0.66	9	5.94
Limit Sensor	MSW-22 NINIGI	[49]	0.38	18	6.84
Red LED	LL-1003VD2D-V1-1A LUCKYLIGHT	[48]	0.11	11	1.21
Green LED	LL-1003GD2D-1A LUCKYLIGHT	[47]	0.11	11	1.21
Resistance SMD	AR03BTCX1001 VIKING	[52]	0.046*	36	1.66
					130.62

*Price per unit when ordering 100+ items.

3.3 *Module Implementation*

Starting with the DC motor's power supply, Figure 16 shows a configuration of the H-bridge circuit connected to the DC motor.

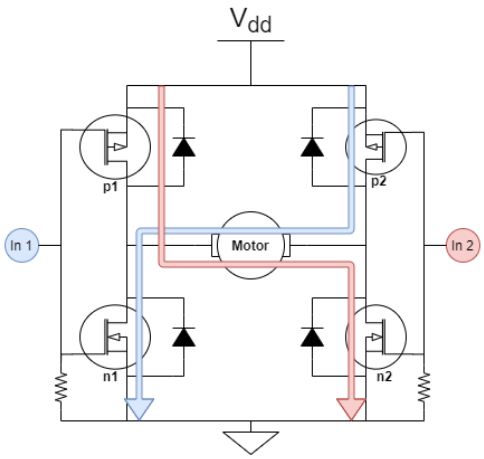


Figure 16: H-bridge circuit with DC motor power supply

Analysing the previous circuit, it is possible to observe that the gates of the nmos and pmos pairs (from Figure 15: (c), P1G-N1G and P2G-N2G) are connected. This allows the current to flow in one of the two represented directions when feeding one of the terminals: In 1 or In 2. When supplying voltage to the In 1 terminal, the transistor n1 will reach the driving zone and p1 the cutting zone, while p2 and n2 are driving and cutting, respectively, thus causing the current to flow through the motor in the direction represented in blue. Feeding the In 2 input, the opposite will happen and the current flow will be the one represented in red. The 1kΩ resistors were placed to protect the transistors from the input current.

Further than the valve operation and control, it is also necessary to implement the limit sensors as feedback signals. Here, the goal was simply to design a circuit that connects a LED and sets a terminal on whenever the sensor is pressed. The circuit is shown in Figure 17.

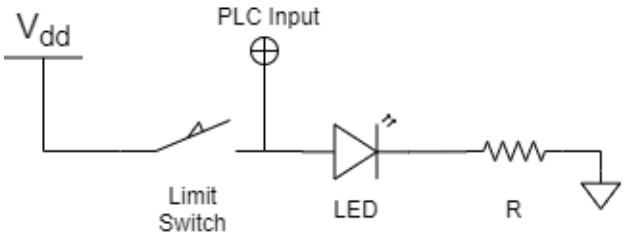


Figure 17: Limit switch a LED circuit.

To dimension the previous circuit, a 24V Vdd and a 20mA current flowing through the LED with a voltage drop equivalent to 2V was considered, thus obtaining $R \approx \frac{24-2}{0,02} = 1000\Omega$. Two circuits like this are required for each electrovalve module.

Now, to develop the structure that will merge the components of the modules, a study of their dimensions was conducted (shown right below in Figures 18, 19 and 20).

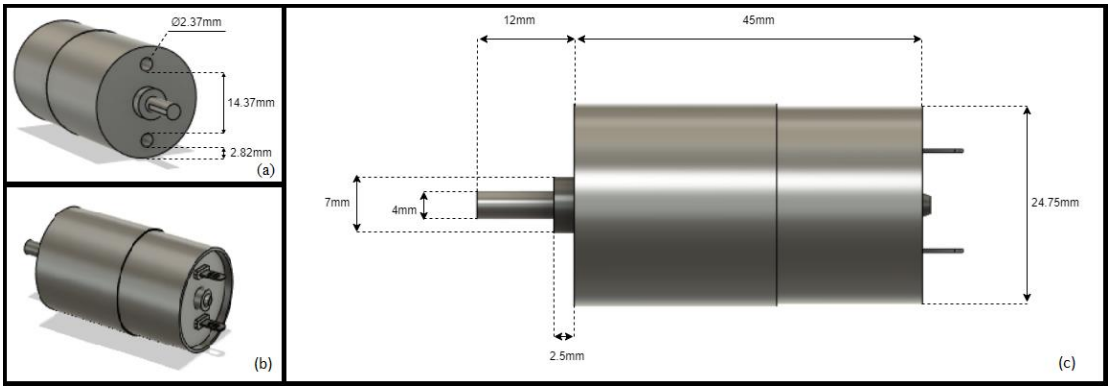


Figure 18: 3D model of the dc motor (with dimensions) [53].

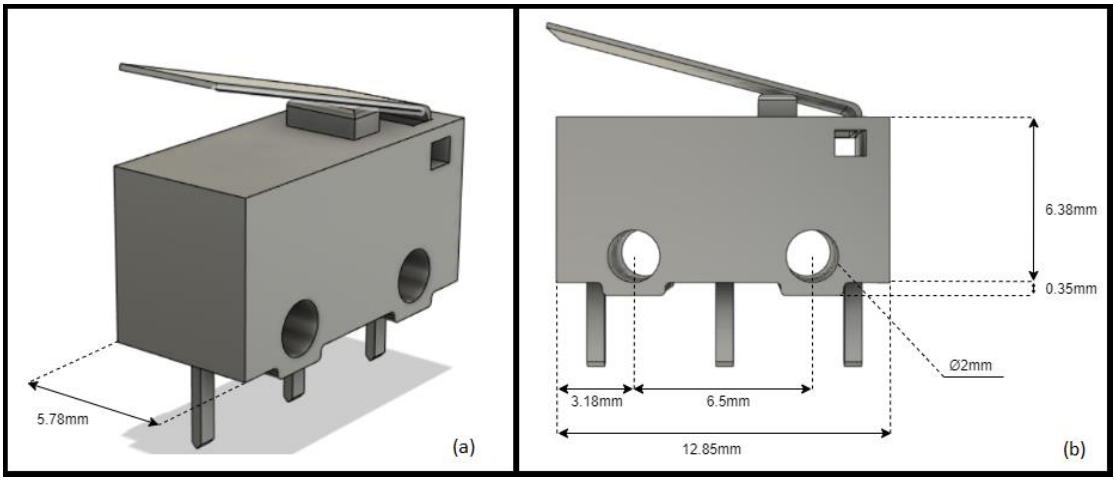


Figure 19: Limit switch 3D model (with dimensions) [54].

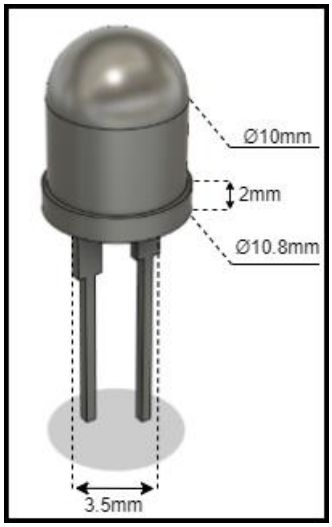


Figure 20: LED 3D model (with dimensions) [55].

The 3D parts of the previous figures were obtained from the web and changed (using the modelling software Autodesk Fusion 3D) to match to the exact dimensions of the components.

Now, knowing the dimensions of all components needed for this module, the support and integration structure was designed in Autodesk Fusion 3D. The designed 3D model of the support is shown below in Figure 21.

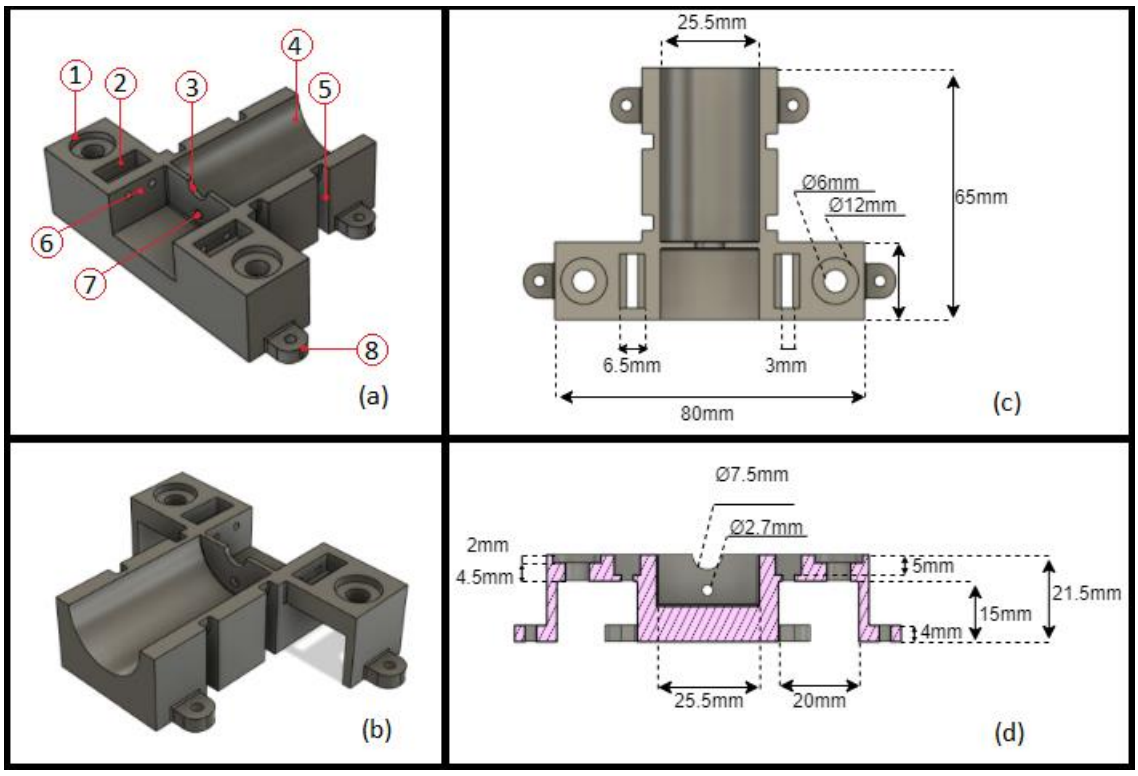


Figure 21: 3D model of the module support: (a) – perspective from front with elements marked; (b) - perspective from back; (c) – top view with dimensions; (d) – vertical cut from front view with dimensions.

Figures (a) and (b) show the structure's 3D drawings. The numbered items are described below:

- 1 - LEDsholder. (2)
- 2 - Holder for the limit switch. (2)
- 3 - Base of the motor with space for the rotation axis. (1)
- 4 - Cylindrical base to support the DC motor. (1)
- 5 - Slit open under the motor base, where a clamp was placed to hold the motor (2).
- 6 - Screw holes for the limit sensors (2x2).
- 7 - Hole for the motor screw (1).
- 8 - Screw holder to fix the module (2x2).

Figure 21(c) represents the top view of the model, with all the relevant measures and (d) represents the view of a vertical-cut section in the middle of the limit sensors and LEDs sockets. The dimensioning of this last piece was carried out considering a 1mm margin of (added to the actual measurements). A piece to fit the motor shaft was also designed follow the motor's rotation and press the limit sensors (Figure 22).

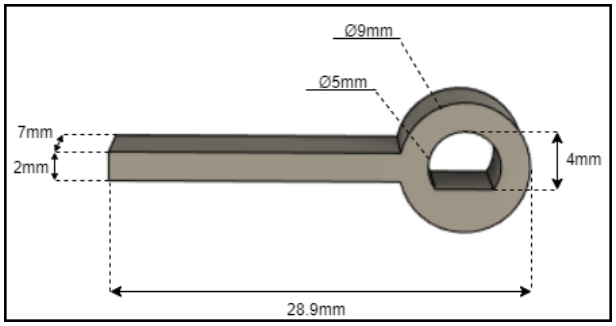


Figure 22: Reed 3D model with dimensions.

Finally, the 3D model of the module was obtained, with all the components assembled. It meets the expected result and is shown in Figure 23.

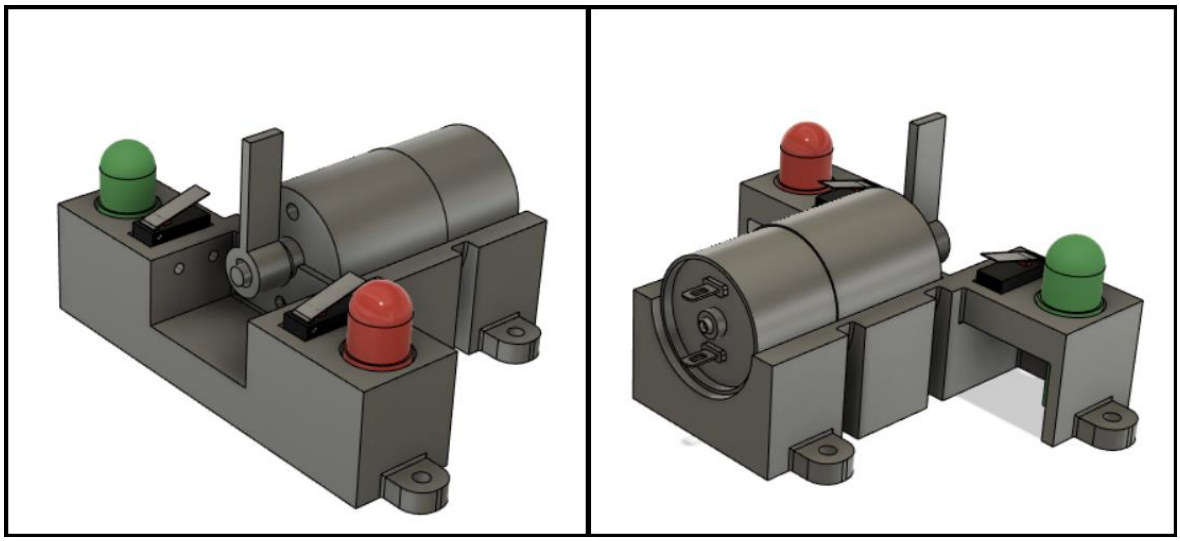


Figure 23: 3D model of the expected module (with components) - front and back views, respectively

Due to the error margin of the printers, the first printed parts were discarded as they the LEDs and limit sensors did not fit. After the drawing's correct margin of error was setup, the parts were printed and the module assembled. The assembly consisted in welding wires to the terminals of all components, attach them to the corresponding supports. The full module is shown in Figure 24.

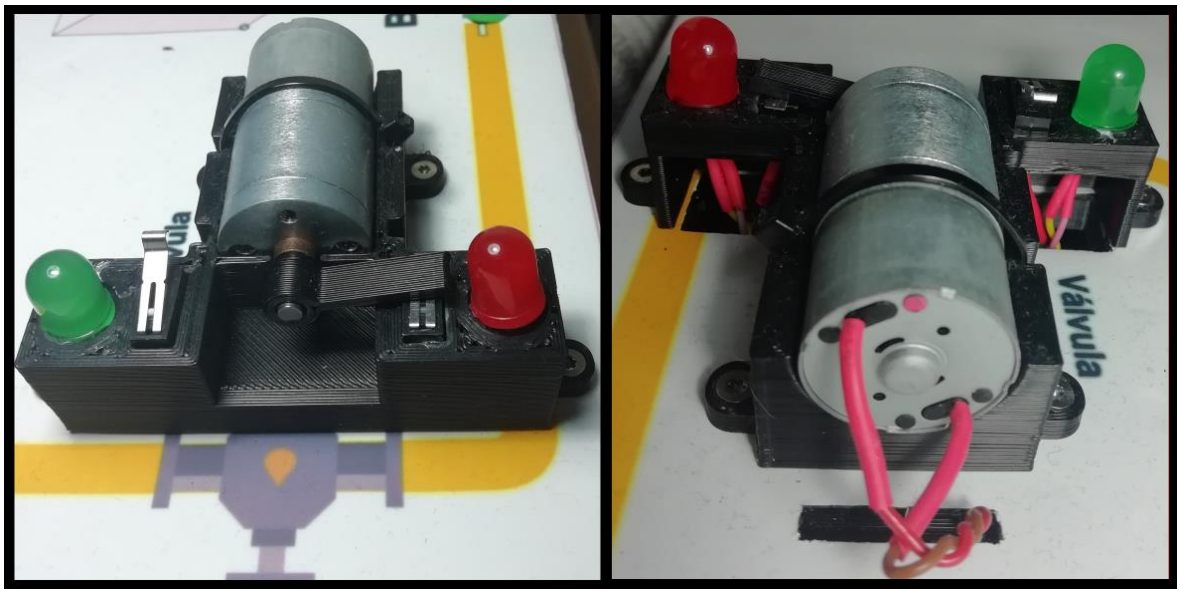


Figure 24: Assembled valve module (front and back, respectively).

3.4 *Initial approach*

The purpose of this initial approach was to test the operation of the modules and their components, as well as to include a PLC control. The circuits necessary for this early implementation were assembled on a bread board.

It started by using just one module to understand if everything was working as intended. Several points were tested: the rotation of the motors, the activation of the limit sensors when pressed and, consequently, the LEDs connection. Then, next steps would be the control of a set of modules to simulate a vat and a tank, in order to confirm how modules behaved when working together. It is noteworthy that, as these systems were being implemented, the H bridges used to drive the motors proved to be an unfriendly structure when used on bread board mounting. For this reason, instead of these bridges, alternative ways of feeding the motors were used, described later in this chapter.

3.4.1 One module implementation

Two circuits (like the one shown in Figure 17, designed for the limit sensors) and LEDs were assembled on a breadboard. These circuits have direct supply from the PLC (24 V) and the motor was connected to a digital PLC output. A diagram of this assembly can be seen in Figure 25.

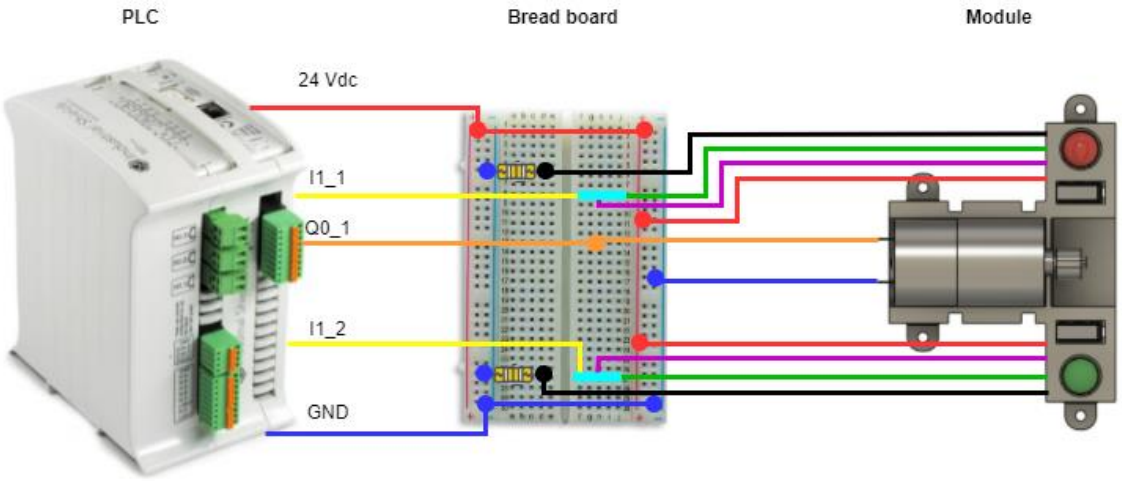


Figure 25: PLC to module connection scheme.

In the previous diagram, LEDs connection wires may be seen in green and black. The wires of the limit sensors are in purple and red (power signal). PLC's digital inputs (for sensor state reading) are shown in yellow. Also note that the motor connection, orange and blue (GND signal), is made directly with the PLC's digital output Q0_1 (also orange) and has a specific polarization, which implies the operation of the motor in just one direction. To change the motor's rotation direction, the changing of this polarization proved to be sufficient.

A simple code was developed for the PLC. Activation only occurs by the digital output Q0_1, which initiates the motor operation. Switching it off occurs as soon as any of the digital inputs (I1_1 and I1_2) read the value 1. In other words, as soon as one of the limit sensors is pressed by the reed attached to the motor it stops the operation.

3.4.2 One vat and one tank system implementation

To implement a vat and tank system, five modules are required, four to represent the electrovalves of the tank, the vat, the purge and the pump, and another one to represent the vat.

As an alternative to the H bridges in the motor supply, Finder 40.52 relays available on the company were used. These relays have two configurable contacts, so it is possible to assemble a motor drive circuit in both possible directions, as the one shown in Figure 26.

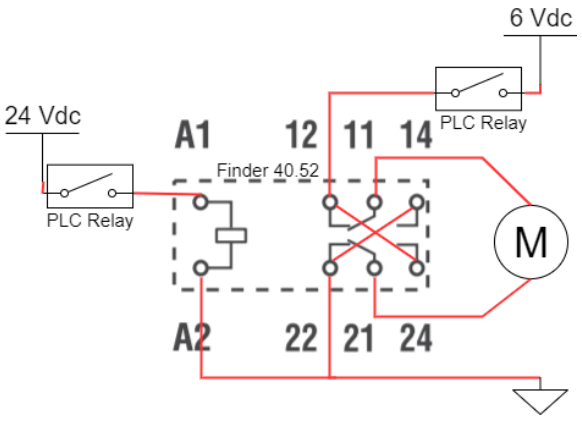


Figure 26: Relay configuration for DC motors power supply.

The previous circuit consists of two PLC relays, one that supplies 24 V of PLC power to the control relay drive coil, and another one that supplies 6 V to the motor. Connection occurs in terminals 12/24 and 22/14, respectively. Therefore, when the 6 V relay is activated the motor will be polarized from top to bottom, according to the image, and when the 24 V relay is activated, the position of the control relay will switch, and, consequently, change the motor polarization, leading to a rotation in the opposite direction. It should be noted that the 6 V comes from a voltage regulator (available at Watgrid) and is sufficient to power the motor.

Using the LED connections and limit sensors from the previous implementation, together with this relay configuration to drive the motors, the module is then fully operational. Therefore, the circuits of the five modules were assembled, and the PLC peripherals were used as indicated in Table 6. Figure 27 shows how the five modules and the PLC were placed together.

Table 6: PLC peripherals' distribution

PLC peripherals	Bomb valve	Tank valve	Vat valve	Vat	Purge valve
24V relay	R1_4	R1_5	R1_6	R1_7	R1_8
6V relay	R1_1	R1_2	R1_3	R2_1	R2_2
Opened DI	I1_2	I1_4	I2_2	I2_4	I0_7
Closed DI	I1_3	I1_5	I2_3	I2_5	I0_8

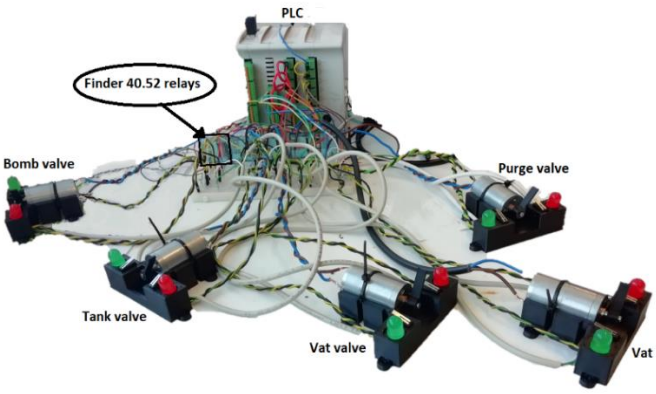


Figure 27: One vat and one tank system mount

3.5 *System integration*

3.5.1 PCB development

For the final integration of the system, a PCB that addressed all the circuits necessary for a proper module operation was developed. The connection points between the PLC and the modules were also considered. The PCB started to be outlined in the electronic design software *EAGLE* from *AUTODESK*®. Figures 28 and 29 show the schematics and the board.

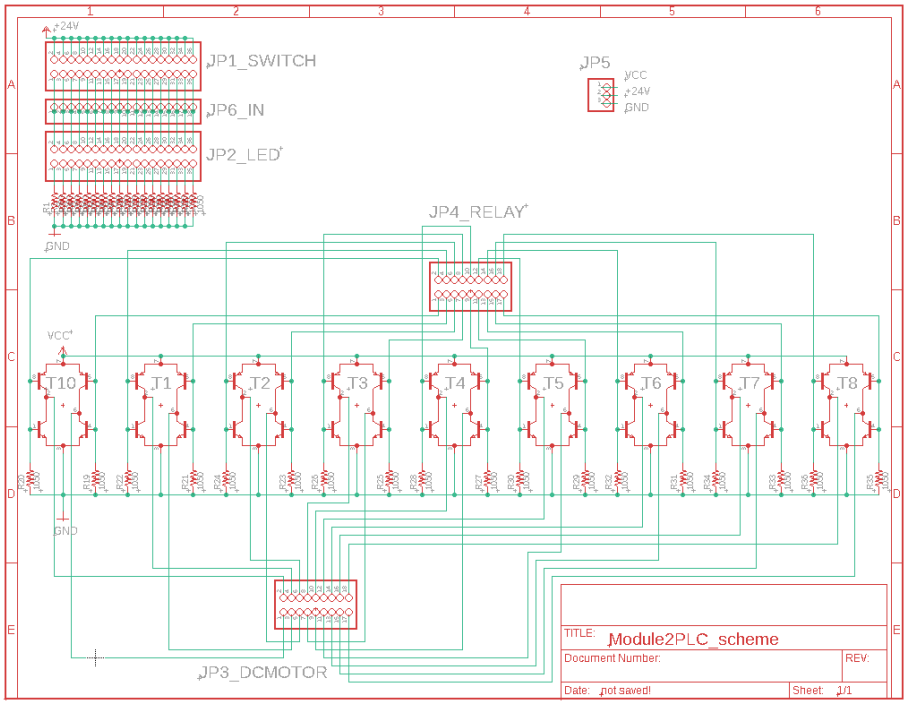


Figure 28: Pcb scheme of the circuit that includes all modules.

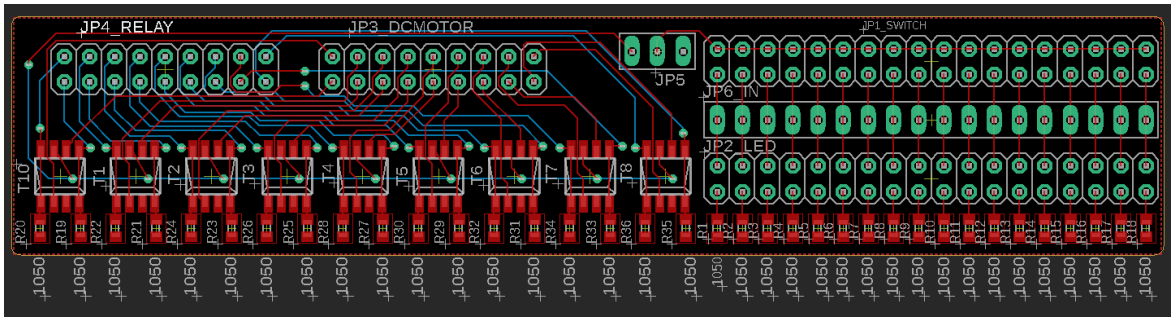


Figure 29: Board of the circuit that includes all modules .

In the schematics of Figure 28, it is possible to check the nine H-bridges, T1 to T10, connected to the JP3_DCMOTOR connector (where the motors are powered) and to the JP4_RELAY connector (where the PLC relays are connected), which stipulates the motor rotation's direction. In the upper left corner, there are connectors for the limit sensors, the PLC inputs and the LEDs (JP1, JP6 and JP2, respectively), which represent eighteen circuits, just like the one shown in Figure 17. 1kΩ resistors with SMD package were used, as can be seen on the board in Figure 29. One of the goals of this PCB was size reduction to lower down the production cost.

Figure 30 show a 3D version of the expected result of the PCB. It served its purpose of checking whether everything would be within the supposed dimensions of the components.

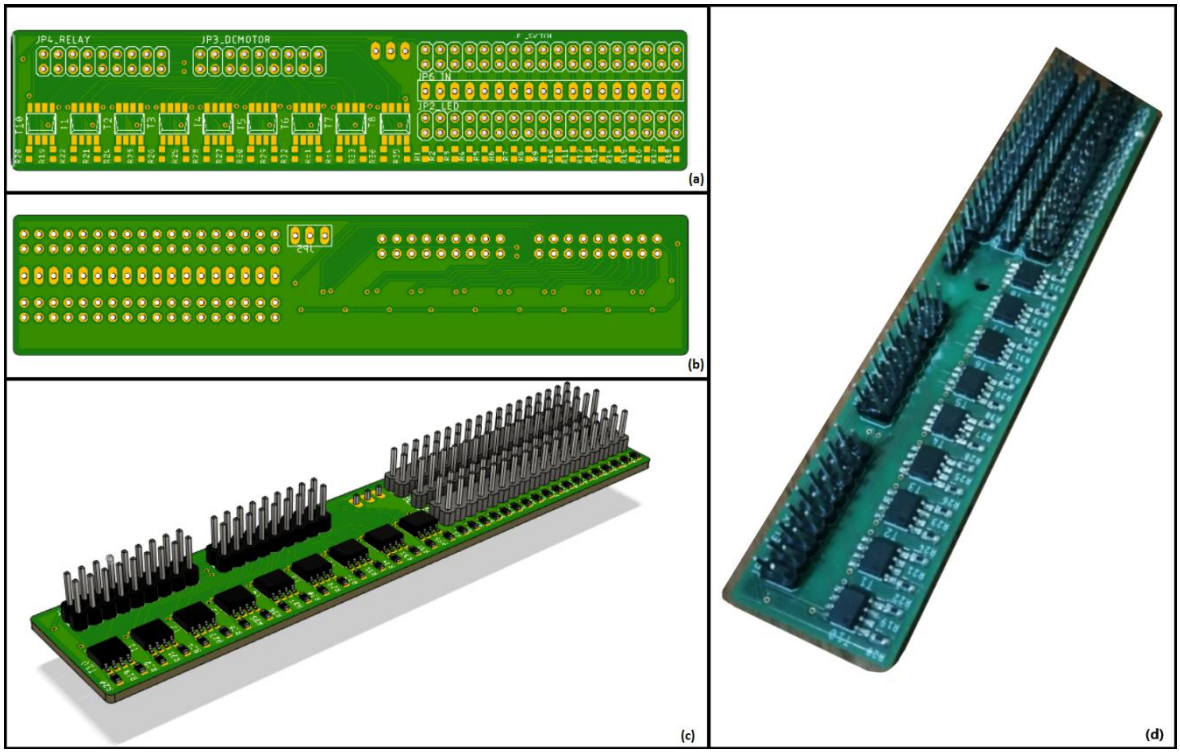


Figure 30: 3D model of the pcb: (a) – top view; (b) – bottom view; (c) – 3D view with all components mounted; and assembled PCB (d).

In Figure 30, it is possible to observe the expected result of the printed PCB without (a-b) and with (c) components, and the assembled PCB (d). the PCB was produced at *Eurocircuits* with a cost of 38.12 € [56].

3.5.2 **Platform**

After all the necessary modules assembled and fully operational, the next step was focused developing a support platform for all modules. This platform allowed the modules to be arranged with a specific system layout. Since the company has 3D printers, the conclusion was reached that the most economical and workable option was to print the platform in 3D. For this, a 3D model showed in Figure 31 was designed.

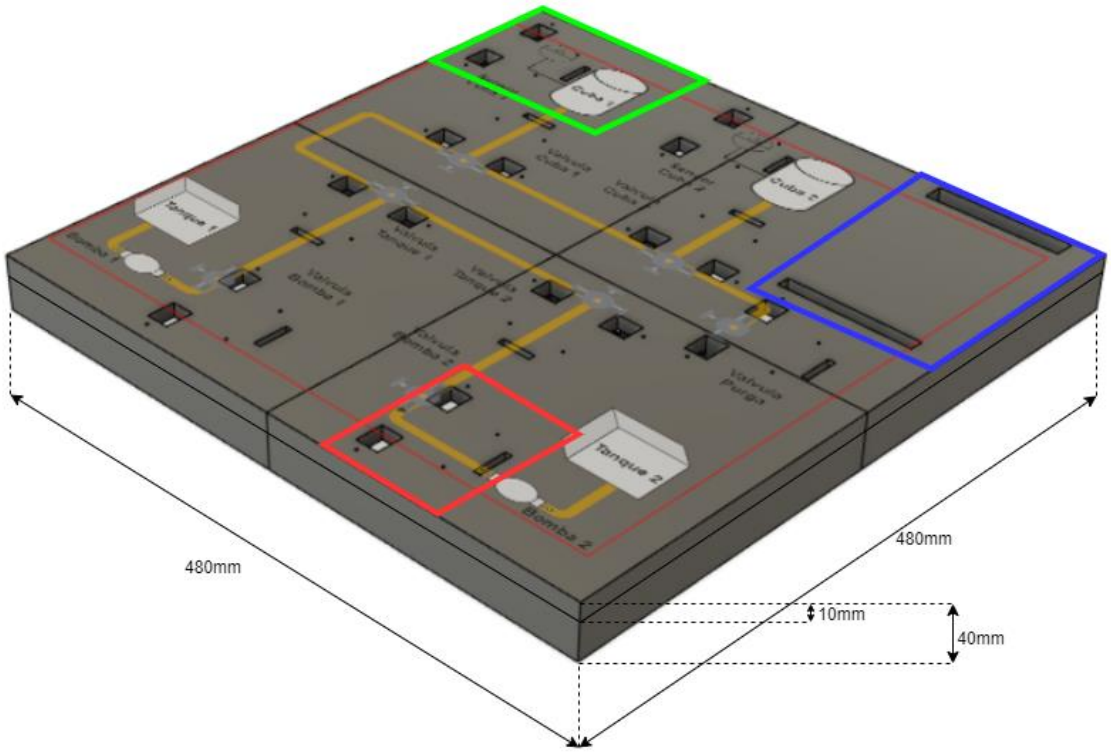


Figure 31: Platform 3D model.

The platform was dimensioned according to the arrangement of the modules in the image of the system, which is represented in Figure 12 of Section 3.2, and their respective occupied space, which resulted in the above structure.

The space dedicated to an electrovalve module can be seen in red. In this space, there are four holes where supporting paws will be screwed, two identical holes through which the LEDs and limit sensors are placed (on each side of the module), and another hole for the motor wires. It should be noted that the front of the module faces the two largest holes (and that its rear is faced towards the tightest). For each position where a valve should be represented, this hole configuration was considered. The green colour is displayed where the vat module will be positioned. As these modules are physically the same as those of the solenoid valves, the size of the holes is also the same. Finally, the space enclosed in blue may be observed. This is where the PLC will be placed. On each side, two holes were made for the passage of the wires connecting to the PLC.

Since the printers have a limit printable area, the platform has been divided into four parts.

When looking at Figure 32, it is possible to identify the four parts resulting from the division of the platform: A, B, C and D. In these parts, some elements have been inserted to reinforce the structure. All parts have two inner walls (1), thinner than the outer ones, with three holes each. The holes represented by 3 (one per inner wall) serve as a tunnel for the wires to pass between parts, and the two holes represented by 4 allow to fix them. These walls only touch at the far ends, base and centre (2), thus avoiding irregularities when assembling the platform. The element 5 represents holes, which are in a semicircle and do not penetrate the base of the platform, thus making it possible to clamp the wires connecting the modules and the PLC to the pcb shown in space 6. Figure 33 shows a 3D model of what is expected to result from the platform, with all modules assembled.

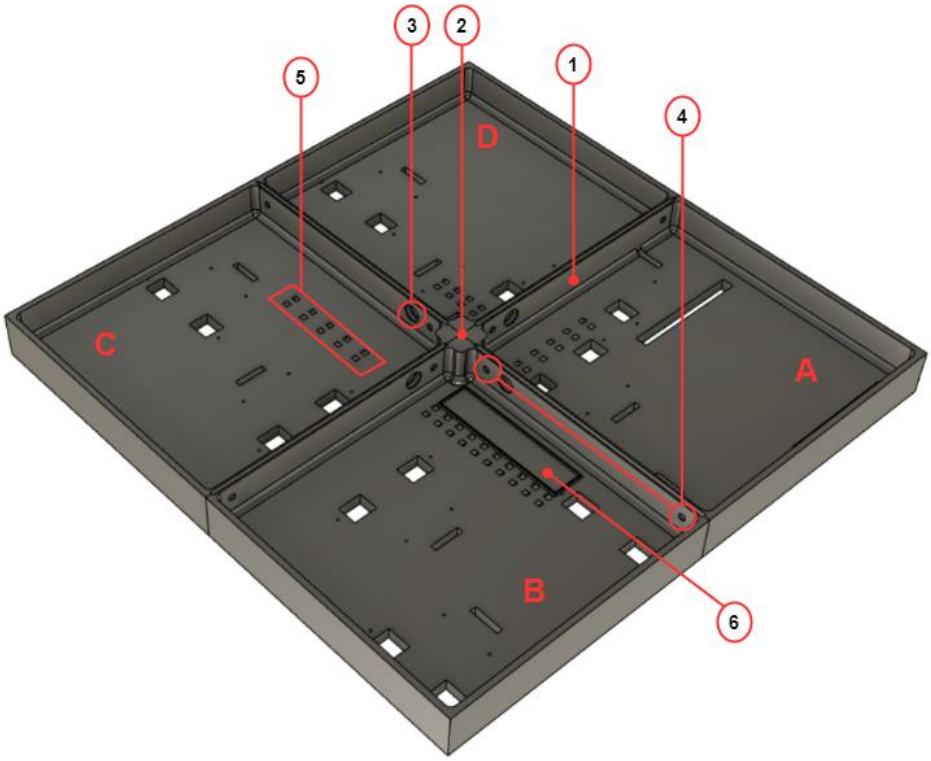


Figure 32:Bottom side of the platform.

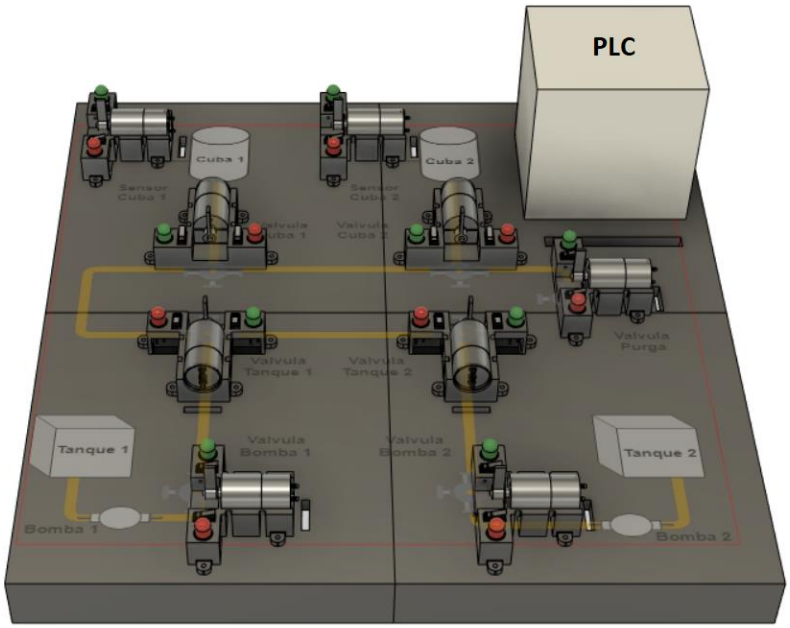


Figure 33: 3D model of the platform with modules.

The four pieces of the platform structure were printed. Although these were dimensioned according to the limits of the printers, they continued to be relatively large pieces for what was recommended, and the prints came out with some defects, namely on the surfaces of the bases, which were not completely flat and displayed some relief. To correct these flaws, they were filled with resin, which ended up to minimize these defects. After the parts had been attached, an image of the system with the appropriate dimensions for the platform was printed on vinyl, and then glued to its surface. The outcome is that the platform is ready to support the modules and the PLC.

3.5.3 Final assembling

With all the mock-up components ready, the integration proceeded. It started with the screwing of all the modules in the concerning spot, the same happening with the PCB. In addition, it was also necessary to attach the PLC to the platform. Therefore, the 3D printing of a din rail was considered since the PLC has its own fitting for it. The 3D model of a din rail was obtained from the website [57]. It was later resized to match the PLC's appropriate size. The 3D model obtained is shown in Figure 34.

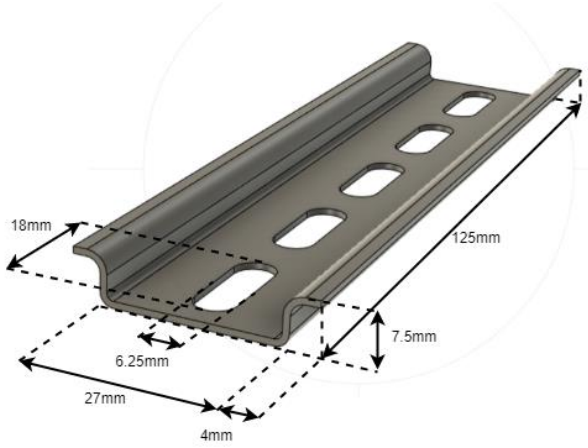


Figure 34: 3D model of the din rail with dimensions.

The din rail was screwed to the platform in the PLC area, and then the same occurred with the LEDs aimed to demonstrate the transfer pumps: the red LED on the left, indicating liquid flow into the tank, and the green one on the right, indicating the liquid outgoing from the tank.

After the integration of the components on the platform, the modules and PLC were connected to the PCB. Table 7 indicates the PLC peripherals used by the modules.

Table 7: PLC peripheral distribution

		Tank valve 1	Pump valve 1	Vat valve 1	Vat 1	Purge valve	Tank valve 2	Pump valve 2	Vat valve 2	Vat 2
DC Motor	open	R2_4	R2_6	R2_7	R2_8	R2_1	R2_2	R1_3	Q0_0	Q0_2
	close	R1_4	R1_6	R1_7	R1_8	R1_1	R1_2	R2_3	Q0_1	Q0_3
Limit Sensor	opened (Green)	I0_8	I0_9	I0_12	I2_3	I1_4	I1_2	I2_5	I0_3	I0_1
	closed (Red)	I0_7	I0_10	I0_11	I2_2	I1_5	I1_3	I2_4	I0_2	I0_0

	Bomb 1	Bomb 2
Out (Green)	Q1_0	Q0_6
In (Red)	Q1_1	Q0_7

Note that, in Table 7, in vat valve 2 and vat 2, digital outputs are being used instead of relays. It turns out that three more relays are required to cover the needs of all components of this system, so it was decided to use the digital outputs that, in this case, only activate the H bridges. This is possible because the PLC's digital outputs provide enough current for the H bridge transistors to transit through zones. In a real system, the corresponding valves would not work due to the lack of power from the digital outputs. In addition, other adaptations have been considered and are illustrated through the scheme presented in Figure 35.

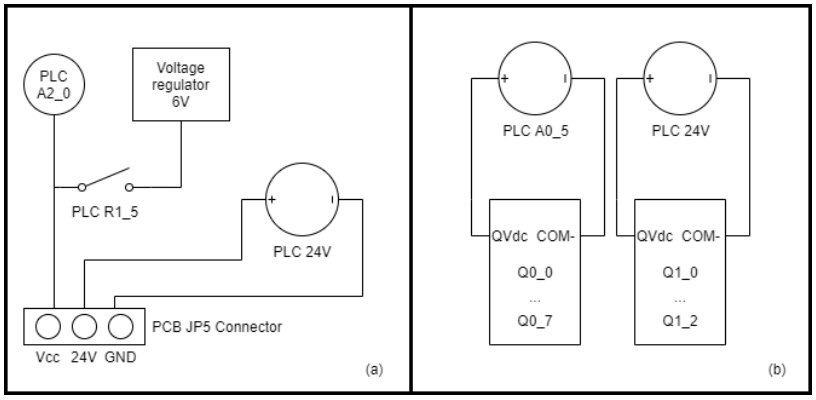


Figure 35: PLC peripheral adaptions: (a) - dc motor supply; (b) - Digital outputs supply.

In section (a) of the diagram, there is the JP5 PCB connector with the 24 V and GND pins, for supply of the LEDs connected to the PLC power supply. The Vcc pin, on the other hand, from where the H bridges are supplied, and which will later supply the motors, has a relay connected (when active, it provides 6 V) and an analogue output from the PLC. This analogue output connection was implemented because the need to decrease the rotation speed of the vat-representative modules motors. By software, it was possible to turn ON the analogue peripheral and leave the relay open only when these modules work, so that the current supplied to the motor is reduced. Then, after turning OFF the analogue output and closing the relay, the power to the Vcc pin is changed again, which will now supply the 6 V to the electrovalve modules.

Figure 35(b) shows how the sets of the PLC digital outputs are powered ON, the explanation being that the logical values '1' and '0' they reproduce are dependent on 'QVdc' and 'COM-', respectively. Just observe the supply of the second block of digital outputs that corresponds to the 24 V PLC supply. This occurs because it is the necessary voltage to supply pumps' green LEDs.

Regarding to the LEDs that represent the transfer pumps no connectors were added. They were wired connected to the PLC: the anodes connected to the digital outputs, and the cathodes to a resistance of 1 kΩ and then connected to the PLC ground.

Finally, with all the model's components integrated, the mock-up was obtained and is shown in Figure 36.

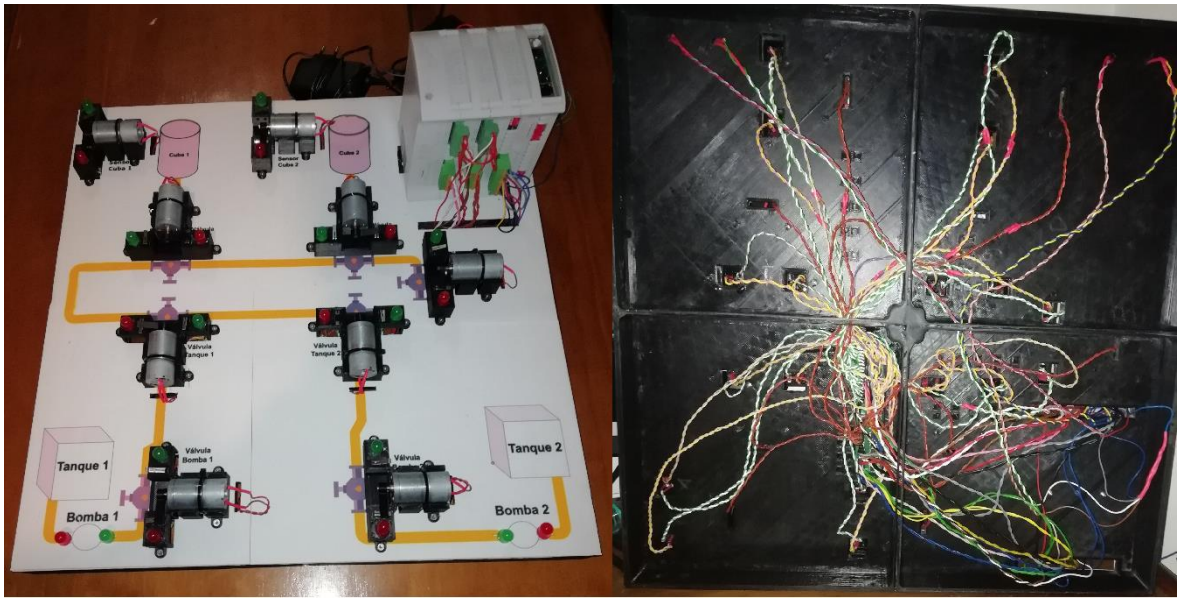


Figure 36: Assembled mock-up (top and bottom, respectively).

4. Software

4.1 *Introduction to PLC programming*

Before starting to write the modules' control code for the PLC, it was necessary to understand how to interact with the peripherals and other functionalities.

As already mentioned, to program the PLC, it is necessary to use the *Arduino* open-source software, *Arduino IDE*, since an *Arduino Mega is used*. For this purpose, the *MDUINO 53 ARR +* datasheet contains instructions about the setup of the editor and has the required resources to make the programming more accessible.

The first step was to add the specific PLC board library to the *Arduino IDE* and provided by *Industrial Shields*. Once installed, all boards corresponding to *Industrial Shields* products can be programmed using this software by just selecting the board and model of the PLC in question. After completion of the previous steps, the software is ready to compile functional code for the PLC.

The installed library assigns specific names to the PLC peripherals and configures their operation mode, which can be seen in Figure 37.

Base front (common unit) — A Zone

M-Duino Connector	Arduino Pin	Function
SCL	21	I2C/SS
SDA	20	I2C/SS
RX0	1	RX0/SS
TX0	0	TX0/SS
NC	-	Not Connected
NC	-	Not Connected
TX	16	RX2(serial 2)
RX	17	TX2(serial 2)
Z-	-	RS485
Y+	-	RS485
B-	-	RS485
A+	-	RS485
PIN3	3	Arduino Pin
50 SO	50	SPI
51 SI	51	SPI
52 SCK	52	SPI
Reset	Reset	SPI
Vin5	Vin5	5V
PIN2	2	Arduino Pin
GND	-	Gnd
GND	-	Gnd
24Vdc		Power Supply

Base back (common unit)

AREF	AREF	Arduino PIN
IOREF2	IOREF2	Arduino PIN
IOREF1	IOREF1	Arduino PIN
7Vdc	7Vdc	-
GND	GND	GND
3.3Vdc	3.3Vdc	Arduino PIN
GND	GND	GND
5Vdc	5Vdc	-
GND	GND	GND

B Zone front

M-Duino Connector	Arduino Pin	Function
I0.12	59	Analog/ Digital In
I0.11	58	Analog/ Digital In
I0.10	57	Analog/ Digital In
I0.9	56	Analog/ Digital In
I0.8	55	Analog/ Digital In
I0.7	54	Analog/ Digital In
(-)I0.6/INT1	-	GND I0.6
I0.6/INT1[1]	3	Interrupt 1 In
(-)I0.5/INT0	-	GND I0.5
I0.5/INT0[1]	2	Interrupt 0 In
(-)I0.4	-	GND I0.4
I0.4	26	Digital Input
(-)I0.3	-	GND I0.3
I0.3	25	Digital Input
(-)I0.2	-	GND I0.2
I0.2	24	Digital Input
(-)I0.1	-	GND I0.1
I0.1	23	Digital Input
(-)I0.0	-	GND I0.0
I0.0	22	Digital Input

B Zone back

GND	GND	GND
A0.7[1]	6	Analog Out
A0.6[1]	5	Analog Out
A0.5[1]	4	Analog Out
Q/Vdc	-	External Isolated Out Vdc
COM(-)	-	External Isolated Out GND
Q0.7[1]	6	Digital/PWM Out
Q0.6[1]	5	Digital/PWM Out
Q0.5[1]	4	Digital/PWM Out
Q0.4	40	Digital Out
Q0.3	39	Digital Out
Q0.2	38	Digital Out
Q0.1	37	Digital Out
Q0.0	36	Digital Out

C Zone front

M-Duino Connector	Arduino Pin	Function
R1.8	42	Relay Out
R1.7	43	Relay Out
R1.6	44	Relay Out
R1.5	45	Relay Out
R1.4	29	Relay Out
A1.2[1]	11	Analog Out
A1.1[1]	10	Analog Out
A1.0[1]	8	Analog Out
GND	GND	GND
Q1.2[1]	11	Digital/PWM Out
Q1.1[1]	10	Digital/PWM Out
Q1.0[1]	8	Digital/PWM Out
GNDCOM	-	External Isolated Out GND
24VCOM	-	External Isolated Out Vdc

C Zone back

R0.3	30	Relay Out
R0.2	27	Relay Out
R0.1	28	Relay Out
GND	GND	GND
I0.5	63	Analog/Digital Input
I0.4	62	Analog/Digital Input
I0.3	61	Analog/Digital Input
I0.2	60	Analog/Digital Input
I0.1[1]	19	Interrupt 1 In
(-)I0.1	-	GND I0.1
I0.0[1]	18	Interrupt 0 In
(-)I0.0	-	GND I0.0

D Zone front

M-Duino Connector	Arduino Pin	Function
R2.8	47	Relay Out
R2.7	48	Relay Out
R2.6	49	Relay Out
R2.5	NC	NC
R2.4	34	Relay Out
A2.2	NC	NC
A2.1[1]	13	Analog Out
A2.0[1]	12	Analog Out
GND	GND	GND
Q2.2	NC	NC
Q2.1[1]	13	Digital/PWM Out
Q2.0[1]	12	Digital/PWM Out
GNDCOM	-	External Isolated Out GND
24VCOM	-	External Isolated Out Vdc

D Zone back

R2.3	35	Relay Out
R2.2	32	Relay Out
R2.1	33	Relay Out
GND	GND	GND
I2.5	69	Analog/Digital Input
I2.4	68	Analog/Digital Input
I2.3	67	Analog/Digital Input
I2.2	66	Analog/Digital Input
I2.1[1]	21	Interrupt 1 In
(-)I2.1	-	GND I2.1
I2.0[1]	20	Interrupt 0 In
(-)I2.0	-	GND I2.0

Figure 37:Mduino pinout table [35].

To interact with the PLC peripherals in the *Arduino IDE*, it is necessary to run the already defined writing and reading functions. These functions are described below:

- digitalWrite(pin, value):

    - Description: It adds a “High” or “Low” value to a digital pin. It is used for driving relays and digital outputs.
    - Pin: *Arduino* pin number.
    - Value: Value that has to be assigned to the pin, “HIGH” or “LOW”.
- analogWrite(pin, value):
    - Description: It adds a square PWM wave to a digital pin. It is used for analogue outputs.
    - Pin: *Arduino* pin number.
    - Value: The duty cycle between 0 (always off) and 255 (always on).
- digitalRead(pin):
    - Pin: Arduino pin number.
- analogRead(pin):
    - Description: It reads the “High” or “Low” value of a specific digital pin. It is used for reading digital inputs.
    - Pin: Arduino pin number.

In addition, the Arduino has a defined programming structure, which is followed by most users. This structure, which can be seen in Figure 38, is divided into:

- The *setup()* function, that is called only at the beginning of a program and it is used to initialize variables, to define the pin mode, among other possible configurations that need to be performed before the program starts [58]
- The *loop()* function is where the control code of the Arduino board should be and it is executed consecutively in a cycle, allowing changes and program response [59].

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Figure 38: Arduino IDE coding structure.

4.2 *Control of one module*

This section describes the software developed for monitoring only one module(chapter 3.1.3.2). As already mentioned, the goal was to write a code for the PLC that would allow to activate the module's motor and detect when one of the limit sensors is pressed, which will then stop the motor.

Starting with the peripherals from Figure 25, it is possible to check that the digital output Q0_1 is responsible for the motor control, and digital inputs I1_1 and I1_2 are responsible for reading the sensors status. The code developed for this operation is shown in Figure 39.

```
void setup() {
  Serial.begin(115200);                                          //Init Serial port
  analogWrite(A0_5, 255);                                        //Writing on an Analog Output (supply Q0_1)
}

void loop() {
  int input=0;
  if (Serial.available() > 0) {                                  //When user input available
    // read the incoming byte:
    input = Serial.parseInt();                                   //Read input

    switch(input){
      case 1:
        digitalWrite(Q0_1, HIGH);                                //Writing a HIGH on a Digital Output (start dc motor)
        break;
      default:
        digitalWrite(Q0_1, LOW);                                 //Writing a LOW on a Digital Output (stop dc motor)
        break;
    }
  }
  if (digitalRead(I1_1) == HIGH or digitalRead(I1_2) == HIGH){
    digitalWrite(Q0_1, LOW);                                     //Stop dc motor when sensor pressed
  }
}
```

Figure 39: One-module control code

On the *setup()* function of the, the detection of the initialization of the PLC's serial communication and the assignment of a wave with a 100% duty cycle to the analogue output A0_5 happens. Serial communication was used to make it possible to control the module's operation through the terminal, in order to prevent some problematic hardware operation (for example: the motor not stopping when the limit sensor is active). The analogue output A0_5 is set to its maximum, as it was used for powering the digital output Q0_1.

Regarding module monitoring (the instructions in the *loop()* function), if there is no user input initially, nothing happens. When the user enters the value “1” in the terminal, the digital output Q0_1 is set high and the motor rotation starts according to the previously polarised direction. If the user does not send anything else, the motor will run until one of the limit sensors is activated, and then the digital output is set to zero, thus ending motor operation. After changing the motor terminals, it is possible to repeat the process, and the motor will run in the opposite direction. In the case where a sensor is connected and the user enters option “1” without changing the motor polarization, the digital output is turned off almost at the same time it is turned on, and the module will remain in the same state.

4.3 *Control of one vat and one tank*

After assembling the vat and tank system described in chapter 3.1.2.3, the code that controls the electrovalves and vat modules was developed. Figure 40, shows a diagram that describes the software-run process.

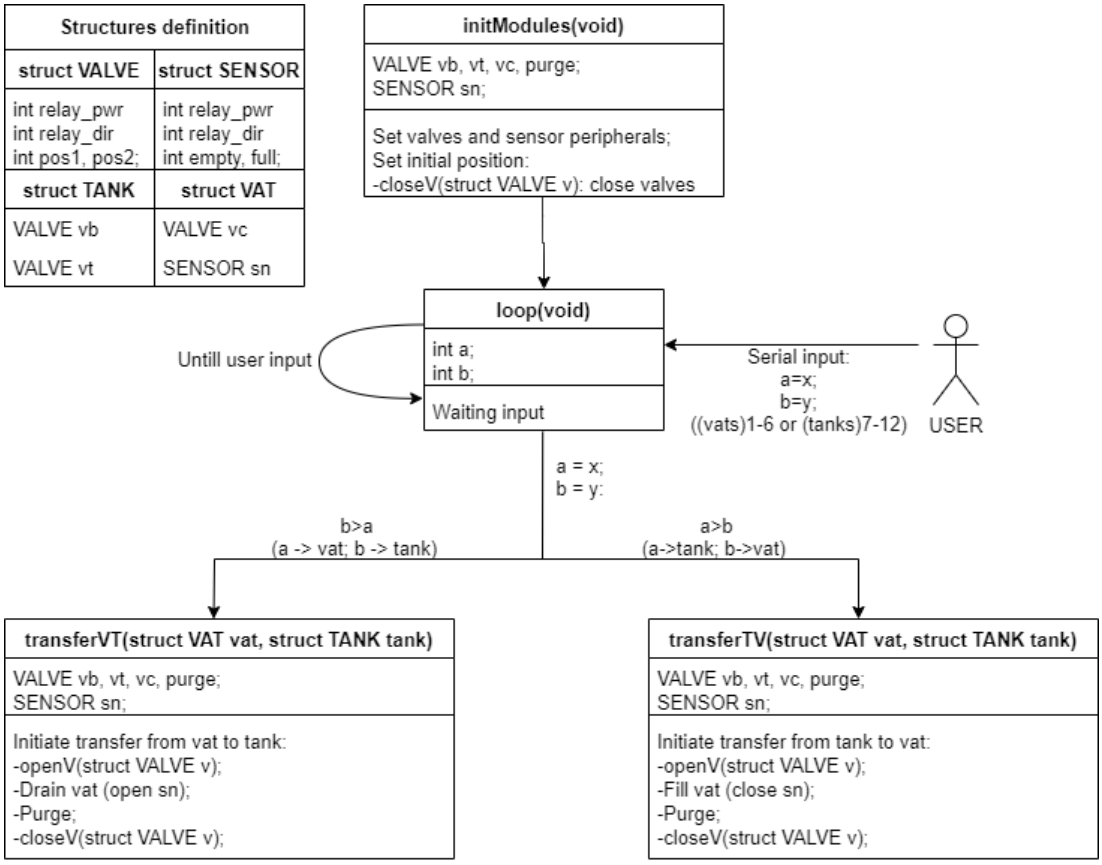


Figure 40: One-vat and one-tank system's software diagram.

Looking at the diagram in Figure 40, four structures were defined to differentiate each module's functionality and store the peripherals assigned to them. Thus, two lower-level structures were considered, which establishes the physical element that a module represents and the definition of peripheral assignments (VALVE and SENSOR). Two other higher-level structures are established that gather the modules according to their function in the system (VAT and TANK). The structures were implemented with the code in Figure 41.

```
 9 struct VALVE{              // Electro valve structure
10   int relay_pwr;          // Motor power supply relay (6V)
11   int relay_dir;          // Finder 40.52 coil relay (24V)
12   int pos1;               // Limit sensor input (closed)
13   int pos2;               // Limit sensor input (opened)
14 }purge;                   // Purge valve initialized

16 struct SENSOR{            // Level sensor structure
17   int relay_pwr;          // Motor power supply relay (6V)
18   int relay_dir;          // Finder 40.52 coil relay (24V)
19   int empty;              // Limit sensor input (empty)
20   int full;               // Limit sensor input (full)
21 };

23 struct TANK{               // Tank structure
24   VALVE vb;               // Pump valve initialized
25   VALVE vt;               // tank valve initialized
26 }tank[2];                  // 2 tanks initialized (only one configured)

28 struct VAT{                // Vat structure
29   VALVE vc;               // Vat valve initialized
30   SENSOR sn;              // Sensor (vat module) initialized
31 }vat[2];                   // 2 vats initialized (only one configured)
```

Figure 41: Structure definition.

The VALVE structure is composed of four fields:

- the "relay_pwr" and "relay_dir", where the PLC relays are assigned to supply the motor and make the Finder 40.52 relay commute (by changing the motor direction), respectively,
- the "pos1" and "pos2", which store the digital inputs of the PLC connected to the limit sensors that represents the position of the valve.

The SENSOR structure fields are identical to the previous ones. In this case, "pos1" and "pos2" are "empty" and "full", respectively, and now represent the tank state, empty or full, and the rest remains.

In the TANK and VAT structures, the following are initialized: the modules assigned to the relevant container, the pump valve (vb), the tank valve (vt), the vat valve (vc) and the vat (sn). Here, although only one tank and one vat are needed, two of each were defined because the algorithm was developed to be scalable.

In order to put the electrovalve modules into operation, two functions were created, *openV(struct VALVE v)* and *closeV(struct VALVE v)*, which activate the relays of a module to perform the valve opening or closing operation, respectively, shown in Figure 42.

```
140 void openV(struct VALVE v){          // Open a specific valve
141   digitalWrite(v.relay_dir, HIGH);   // Set direction
142   digitalWrite(v.relay_pwr, HIGH);   // Start operation
143   while(digitalRead(v.pos2) == 0);   // Wait untill opened
144   digitalWrite(v.relay_pwr, LOW);    // Stop operation
145 }
```

```
147 void closeV(struct VALVE v){          // Close a specific valve
148   digitalWrite(v.relay_dir, LOW);    // Set direction
149   digitalWrite(v.relay_pwr, HIGH);   // Start operation
150   while(digitalRead(v.pos1) == 0);   // Wait untill closed
151   digitalWrite(v.relay_pwr, LOW);    // Stop operation
152 }
```

Figure 42: openV() and closeV() code.

In the functions of Figure 42, the state of the relay that dictates the direction of the motor rotation must run in (relay_dir). It is initially set for “high”, when the opening operation is pretended, or “low” when it wants to close. Then the relay that supplies the motor will be turned on (relay_pwr) and start the operation. This order prevents the engine from starting to run in a direction opposite to the desired one. After activation of the relays, the program waits for the sensor until the desired position is attained and then continues the operation by turning off the power relay.

The program starts by executing the initModules() function described in Figure 43, according to which, as seen in Table 7. The PLC peripherals are assigned to the previously initialized modules. Also, the system is placed in the initial state, which consists in closing the electrovalve modules using the closeV() function.

```
69 void initModules(void){                    // Assign pins to modules
70                                           //*-------------------------------------------------------*\\
71   tank[0].vb = {R1_4, R1_1, I1_2, I1_3};  //
72   tank[0].vt = {R1_5, R1_2, I1_4, I1_5};  //
73                                           //
74   vat[0].vc = {R1_6, R1_3, I2_2, I2_3};   //
75   vat[0].sn = {R1_7, R2_1, I2_4, I2_5};   //
76                                           //
77   purge = {R1_8, R2_2, I0_7, I0_8};       // Assigned pins only to the modules of 1 vat and 1 tank
78                                           //*-------------------------------------------------------*\\
79   closeV(vat[0].vc);                      //                          //
80                                           //
81   closeV(tank[0].vb);                     // Close all valves
82   closeV(tank[0].vt);                     //(set everything to initial position)
83                                           //
84   closeV(purge);                          //
85 }                                         //*-------------------------------------------------------*\\
```

Figure 43: Function initModules().

Next, the system control cycle begins. At this stage, the PLC remains stable, ready for receiving a user's input via serial. When an entry is received, the transfer proceeds.

From the code in Figure 44, it is possible to identify two blocks. Block 1 is responsible for reading and processing a user input with the format (int)a # (int)b if no operation is occurring; it does so by assigning the variables “a” and “b” to the source and destination containers, respectively. It considers the values from 1 to 6 for the vats and from 7 to 12 for the tanks (in this case, only vat 1 and tank 7 were used). The beginning of the transfer is thus indicated. In block 2, after the operation is started, variables “a” and “b” are compared between

them and the direction of the transfer will be defined, as the variable with the lowest value will represent the vat (the tank being assigned the other variable). The process is performed by means of the transferTV() and transferVT() functions, which are examined below.

```
41 void loop() {
42
43   // Vats 1..6 Tanks 7..12
44  if(opRun == 0){                          // Check if no operation running
45    if (Serial.available() > 0) {          // Check for user input
46      a = Serial.parseInt();               // Get container origin
47      Serial.print("a = ");
48      Serial.println(a);
49
50      while(Serial.read()!='#');           // Wait for input with next container
51      b = Serial.parseInt();               // Get destination container
52      Serial.print("b = ");
53      Serial.println(b);
54
55      opRun = 1;                           // Ready to start operation
56    }                                                                    1
57  }
58
59  if(opRun == 1){                          // Check if is ready
60    if(a>b){                               // Check direction (tank to vat)
61      transferTV(vat[b-1],tank[a-7]);      // Run tranfer (tank to vat)
62    }else if(b>a){                         // Check direction (vat to tank)
63      transferVT(vat[a-1],tank[b-7]);      // Run transfer (vat to tank)
64    }
65    opRun = 0;                             // Operation ended
66  }                                                                      2
67 }
```

Figure 44: One-vat and one-tank system's loop() function.

Figure 45 shows the code for the transfer from a tank to a vat on the right and from a vat to a tank on the left. It is possible to check that the logical grounds to both functions include a vat and a tank. This is because, as already mentioned above, it adds scalability to the program, although - in this case - only one vat and one tank are used. The transfer process follows the flow diagram in Figure 89 in Appendix 5.

```
87 void transferTV(struct VAT vat, struct TANK tank){// Run transfer from a vat to a tank
88   openV(tank.vt);                              //
89   openV(vat.vc);                               // Open tank and vat valves
90                                                //*-----------------------------------*\\
91                                                // Turn on bomb
92   digitalWrite(vat.sn.relay_dir, LOW);         //
93   digitalWrite(vat.sn.relay_pwr, HIGH);        // Fill vat (turn on module)
94                                                //
95   while(digitalRead(vat.sn.full) == 0);        // Wait "vat full" signal
96                                                //
97   digitalWrite(vat.sn.relay_pwr, LOW);         // Turn off bomb
98                                                //*-----------------------------------*\\
99   closeV(vat.vc);                              // Close vat valve
100                                               //*-----------------------------------*\\
101                                               // Start Purging
102  openV(tank.vb);                              //
103  openV(purge);                                // Open pump and purge valves
104                                               //
105  delay(3000);                                 // Wait untill purge is done
106                                               //
107  closeV(purge);                               // Close purge valve
108                                               //*-----------------------------------*\\
109  closeV(tank.vt);                             //
110  closeV(tank.vb);                             // Close opened valves (tank and pump)
111 }                                             //*-----------------------------------*\\
```

```
113 void transferVT(struct VAT vat, struct TANK tank){// Run transfer from a tank to a vat
114   openV(tank.vb);                              //
115   openV(tank.vt);                              // Open pump and tank valves
116                                                //*-----------------------------------*\\
117   openV(vat.vc);                               // Open vat valve
118                                                //*-----------------------------------*\\
119                                                // Turn on bomb
120   digitalWrite(vat.sn.relay_dir, HIGH);        //
121   digitalWrite(vat.sn.relay_pwr, HIGH);        // Empty vat (turn on module)
122                                                //
123   while(digitalRead(vat.sn.empty) == 0);       // Wait "vat empty" signal
124                                                //
125   digitalWrite(vat.sn.relay_pwr, LOW);         // Turn off bomb
126                                                //*-----------------------------------*\\
127   closeV(vat.vc);                              // Open vat valve
128                                                //*-----------------------------------*\\
129                                                // Start purging
130   openV(purge);                                // Open purge valve
131                                                //
132   delay(3000);                                 // Wait untill purge is done
133                                                //
134   closeV(purge);                               // Close purge valve
135                                                //*-----------------------------------*\\
136   closeV(tank.vt);                             //
137   closeV(tank.vb);                             // Close opened valves (tank and pump)
138 }                                              //*-----------------------------------*\\
```

Figure 45: Code of the transfers.

In the case of the transfer from a tank to a vat, it starts with the opening of the tank and valves, proceeding with the transfer, until the vat is full. Then, the tank valve is closed, and the pump valve is opened to perform a three-second purge, which ends with the closing of the tank and pump valves.

When a transfer takes place from the vat to the tank, the pump and tank valves open, followed by the vat valve, and the pump is turned on, starting the transfer. After the vat empties and the pump is turned off, the tank valve is closed and the purge is started; then, the opened valves are closed and the process is finished.

4.4 *Control of the system mock-up*

With the system of two vats and two tanks fully assembled, the development of its monitoring algorithm begins. The control was carried out in a similar way to the one described in chapter 3.2.4, with some changes in the structures and the module's interaction functions.

In the previous section, four structures had been developed: one for the valve modules, another one for the vat modules, and two more for the tanks and vats. This time, only two structures were considered (for valves and pumps), as shown in Figure 46.

```
 8  struct VALV{
 9
10    const char* name;           //
11    unsigned int id;            // Valve identification
12
13    int relay_open;             //
14    int relay_close;            // Pins for the relays to activate for open/close valve
15
16    int led_green;              //
17    int led_red;                // Pins for inputs opened/closed valve state
18
19    int state;                  // current valve state
20                                // (0->Not initiated; 1->Opened; 2->Closed; 3->Not responding)
21  }vt[2],vb[2],vc[2],sn[2],vp;  // Valve intialization
23  struct BOMB{
24    unsigned int id;            // Bomb identification
25
26    int green;                  // Pump out of tank
27    int red;                    // Pump into the tank
28
29    int state;                  // 0->Off; 1->Out; 2->In; 3->Malfunction
30  }bb[2];                       // Valve intialization
```

Figure 46: Structures for valves and pumps.

In Figure 46, the top structure, VALV, was defined to integrate the electrovalve modules. In addition to the fields already targeted at registering the peripherals, it has two identification fields ("name" and "id"), and one to store the current position of the module, ("state"). As H-bridges are now used for powering the motors, instead of the Finder 40.52 relays, the trigger of one of the two PLC relays will activate the bridge's transistors, and the motor will rotate in a specific direction. This way, the "relay_open" and "relay_close" fields are made available to choose which mode the motor will operate: open or close, respectively. The digital inputs connected to the limit sensors (that indicate whether the valve is open or closed) are represented by the fields "led_green" and "led_red", respectively. In this case, arrays of the electrovalve modules are initialized with the same size as the amount of tank/vat sets (two in this system). It should be noted that the vat modules (sn) also use this structure because they have the same operation as the electrovalve modules.

The bottom structure, BOMB, was designed to represent the pump modules (only two LEDs each module). In addition to the identification ("id") and state ("state") fields, "green" and "red" fields were created to store the digital outputs associated with each of the module's LEDs, that when active indicated the operation of pumping out or into the tank, respectively.

The functions for opening and closing valves, shown in Figure 47, also had to be adapted to the new structure. Now a pointer of an already-defined valve is given as argument, and, at the beginning, the relay respective to the operation in progress, “relay_close” or “relay_open”, is turned ON, thus starting the motor. After pressing the limit sensor (dependent on the valve’s final-stage position), the previously connected relay is turned off and the status is updated. These functions return a value of 1 (at the end) that is not used for now.

```
83  int openValve(struct VALV *v){              // Open a Valve
84    digitalWrite(v->relay_open, HIGH);        // Turn on Open relay
85    while(digitalRead(v->led_green) == 0);    // Wait opened feedback
86    digitalWrite(v->relay_open, LOW);         // Turn off open relay
87    v->state=1;                               // Change state valve open
88    return 1;                                 // Return Valve opened
89  }
101 int closeValve(struct VALV *v){            // Close valve
102   digitalWrite(v->relay_close, HIGH);     // Turn on close relay
103   while(digitalRead(v->led_red) == 0);    // Wait closed feedback
104   digitalWrite(v->relay_close, LOW);      // Writing a HIGH on a Relay
105   v->state=2;                             // Change state valve close
106   return 1;                               // Return Valve closed
107 }
```

Figure 47: Functions to open and close a single valve.

The previous functions can open or close one valve at a time. For convenience, two functions were developed, both capable of carrying out this process on several valves at once. This makes system operation more realistic, since, during the transfer process, several valves may be positioned at the same time. Then, the functions *closeValves()* and *openValves()* were developed and are represented below in Figure 48.

```
int closeValves(struct VALV *v[], size_t size){                 // Close selected valves
    long unsigned int i = 0,end = 0, cnt = 0;                   //
    for(i=0; i<size;i++){                                       // Iterate through selected valves
      if (digitalRead(v[i]->led_red)!=1 && v[i]->state!=3){     // Only valves not closed and responding
        digitalWrite(v[i]->relay_close, HIGH);                  // Turn on close relay
        v[i]->state=0;                                          // Set valve state to "not set"
        cnt++;                                                  // Keep track of the valves closing
      }
    }
    while(end<cnt){                                             // Iterate until all valves close,
                                                                // excluding the already closed ones
        for(i=0; i<size;i++){                                   // Iterate through selected valves
            if(digitalRead(v[i]->led_red)==1 && v[i]->state==0){ // If valve closed and state "not set"
                digitalWrite(v[i]->relay_close, LOW);           // Turn off close relay
                v[i]->state = 2;                                // Set Valve state to "closed"
                end+=1;                                         // Increment closed valves count
            }
        }
    }
    return 1;                                                   // All valves closed successfully
}
int openValves(struct VALV *v[],size_t size){                   // Open selected Valves
    long unsigned int i = 0,end = 0, cnt = 0;
    for(i=0; i<size;i++){                                       // Iterate through selected valves
      if (digitalRead(v[i]->led_green)!=1){                     // Only valves not opened and responding
        digitalWrite(v[i]->relay_open, HIGH);                   // Turn on open relay
      }                                                         //
      v[i]->state=0;                                            // Set Valve state to "not set"
      cnt++;                                                    // Increment closed valves count
    }
    while(end<cnt){                                             // Iterate until all valves open,
                                                                // excluding the already opened ones
      for(i=0; i<size;i++){                                     // Iterate through all selected valves
        if(digitalRead(v[i]->led_green)==1 && v[i]->state==0){   // If valve opened and state "not set"
          digitalWrite(v[i]->relay_open, LOW);                  // Turn off open relay
          v[i]->state = 1;                                      // Set Valve state to "opened"
          end+=1;                                               // Increment opened valves count
        }
      }
    }
    return 1;                                                   // All valves opened successfully
}
```

Figure 48: Functions to open and close multiple valves.

These functions are logically grounded on an array of pointers corresponding to the electrovalves (that are meant to be repositioned) and the concerning length. One by one, the current status of the valve is checked. If the operation in progress did not reached the final stage, the relevant relay is switched on, the status is updated to “not set” and the number of activated valves is incremented (variable cnt). The system waits for the valves to reach the desired position. When a valve in the “not set” state is detected signalling that it is in the expected position, the respective relay is switched off. Its status is changed to “opened” or “closed”, and the number of valves reaching the operation’s conclusion stage (end variable) is incremented. The process continues until the number of valves reaching the conclusion stage is equal to the number of valves to be transitioned (end == cnt).

Based on the functions in Figure 48, two more functions were developed to open or close all valves to define an array with the system’s seven valves and to run one of the previous functions, *openValves()* or *closeValves()*, as can be seen in Figure 49.

```
void closeAllValves(void){
  struct VALV *v[]={&vt[0],&vb[0],&vc[0],
                    &vp,&vt[1],&vb[1],&vc[1]};
  closeValves(v,7);
}
void openAllValves(void){
  struct VALV *v[]={&vt[0],&vb[0],&vc[0],
                    &vp,&vt[1],&vb[1],&vc[1]};
  openValves(v,7);
}
```

Figure 49: Functions to open and close all valves.

Still regarding module operation, two specific functions were developed for vat module application, in order to represent module filling and emptying. Next, the process underlying these functions is described.

```
204 void fillCuba(struct VALV *sn,struct BOMB *bb){          // Fill Vat (=) Close Valve
205   digitalWrite(sn->relay_close, HIGH);                  // Turn on close relay          1_a
206   analogWrite(A2_0,0);                                  //
207   digitalWrite(R1_5,HIGH);                              //
208   delay(50);                                            // Initial current boost for 50 ms
209   digitalWrite(R1_5,LOW);                               //
210   analogWrite(A2_0,125);                                // Power the motor with PWM 125/256*100 % duty cycle
211
212   int count = 0;                                        // Variable initialization - count cycles
213
214   if(digitalRead(sn->led_red) == 0){                    // Only not filled valves       2
215     bb->state=2;                                        // Change bomb state to filling
216     while(digitalRead(sn->led_red) == 0) {              // Iterate untill is full
217       if(digitalRead(sn->led_red) == 1){                // When it is full
218         sn->state=2;                                    // Change vat state to full (closed valve)
219         break;                                          // Vat full, end
220       }
221
222       if(count%7000==0){                                // Every 7000 loops             3
223         digitalWrite(bb->green,!digitalRead(bb->green)); // Bomb working (Led blinking)
224       }
225       count++;                                          // Increment cycle count
226     }
227   }
228   digitalWrite(sn->relay_close, LOW);                   // Turn off close relay         4
229   digitalWrite(bb->green,LOW);                          // Turn off Bomb output
230   bb->state=0;                                          // Change bomb state to stoped
231   sn->state=2;                                          // Change vat state to full
232
233   analogWrite(A2_0,0);                                  //                              1_b
234   digitalWrite(R1_5,HIGH);                              // Change motor supply
235 }
```

Figure 50: Function to fill vat.

Figure 50 contains the code written so that a vat module might perform the representative operation of the vat filling (through the *fillCuba()* function). As the pumps are activated so that the vat fills or empties, the logical grounds to this function include both the vat to be filled (sn) and the pump that performs the process (bb). The blocks shown in the figure are explained below:

- 1a: As it was intended to lower the rotation speed of the DC motors in this process, an analogue output from the PLC (A2_0) with a duty cycle of 50% was used to supply them. It turns out that this analogue output does not provide enough current to start the motor, but it can maintain its operation. This means that, with the connection displayed in Figure 35 (a), it is possible to start it through relay R1_5 and (almost immediately) change the supply to the analogue output A2_0. First, the relay associated to the vat filling (relay_close) is started. Then, the analogue output A2_0 is switched off and relay R1_5 is switched on. The motor is started and, after fifty milliseconds, relay R1_5 is switched off and the analogue output A2_0 is switched on at 50%, which keeps the motor rotation at a slower speed.
- 2: Initially, it must be confirmed that the vat is not full. In that case, pump status is changed to “filling”, and cyclical checking occurs to determine when the vat is full, by reading the vat’s digital input “led_red”, which is connected to one of the limit sensors. When the sensor is pressed, the program exits the cycle to block 4 to end the operation.

- 3: To simulate the pump's operation, the variable "count" was used, which increases with each cycle while checking the state of the vat. Every 7000 cycles, the pumping direction's LED (in this case, the green LED) switches, causing a flashing effect, which signals the pump operation. After some tests with arbitrated values, the value 7000 was selected because it makes the flashing visually possible.
- 4: When the tank is full, the filling relay is switched off and, consequently, both the motor and the green LED on the pump module signal the conclusion of the pump operation. Pump and vat states are updated to "stopped" and "full", respectively.
- 1b: The process ends with the repositioning of the motor's power supply, that is, the A2_0 analogue output is switched off and the relay R1_5 is switched on.

The process of emptying the vat is identical to that shown in Figure 50. As opposed to the filling process, only the peripherals used by the vat and pump modules and their final state are changed. The function responsible for it, *emptyCuba()*, is displayed in the chunk of code in Figure 51.

```
247 void emptyCuba(struct VALV *sn, struct BOMB *bb){                // Fill Vat (=) Close Valve
248   digitalWrite(sn->relay_open, HIGH);                            // Turn on open/drain relay
249   analogWrite(A2_0,0);                                           //
250   digitalWrite(R1_5,HIGH);                                       //
251   delay(50);                                                     // Initial current boost for 50 ms
252   digitalWrite(R1_5,LOW);                                        //
253   analogWrite(A2_0,125);                                         // Power the motor with PWM 125/256*100 % duty cycle
254
255   int count = 0;                                                 // Variable initialization - count cycles
256
257   if(digitalRead(sn->led_green) == 0){                           // Only not empty valves
258     bb->state=1;                                                 // Change bomb state to draining
259     while(digitalRead(sn->led_green) == 0){                      // Iterate untill vat is empty
260       if(digitalRead(sn->led_green) == 1){                       // When it is empty
261         sn->state=1;                                             // Change vat state to empty (opened valve)
262         break;                                                   // Vat empty, end
263       }
264       if(count%7000==0){                                         // Every 7000 lopps
265         digitalWrite(bb->red,!digitalRead(bb->red));             // Bomb working (Led blinking)
266       }
267       count++;                                                   // Increment cycle count
268     }
269   }
270   bb->state=0;                                                   // Change bomb state to stoped
271   sn->state=1;                                                   // Change vat state to empty
272   digitalWrite(sn->relay_open, LOW);                             // Turn off open relay
273   digitalWrite(bb->red,LOW);                                     // Turn off bomb output
274
275   analogWrite(A2_0,0);                                           //
276   digitalWrite(R1_5,HIGH);                                       // Switch motor supply
277 }
```

Figure 51: Function to empty vat.

Next, the program is started with the Arduino *setup()* function shown in Figure 52. In this section, the serial communication is initialized. Then, the PLC peripherals responsible for feeding the motor-related relays are setup. Relay R1_5 is switched on and analogue output A2_0 is switched off (operation previously referred to in 1_a and 1_b of Figure 50), the digital output assigned to supply the H-bridge control relays (digital output Q0_4) is switched on, and likewise for the PLC digital outputs supply (analogue output A0_5 at 100%). Then, the *setValves()* function is executed, where peripherals are assigned to the modules, and then the *initRelays() function*, which reverts the relays to their initial state (off) and sets initial states for the modules, stopping all

resetting of the electrovalve modules in the closed position, by using the *closeAllValves()* function. These two functions are described below.

```
25  void setup() {
26    IS_initDefaultIOPins();    // MDUINO pins initialization for platformIO
27
28    Serial.begin(115200);      // Init Serial monitor for debug
29
30    analogWrite(A2_0,0);       // Supply for the vats
31    digitalWrite(R1_5,HIGH);   // Activate relay for valves supply
32
33    digitalWrite(Q0_4,HIGH);   // Power for relays (H-bridge Gates)
34    analogWrite(A0_5,255);     // Power for outputs (QVdc) Q0_0-Q0_7
35
36    setValves();               // Configure Valves pins
37    initRelays();              // Set initial relays state
38    closeAllValves();          // Put all modules on initial sate
39  }
```

Figure 52: setup() function.

```
63  void setValves(void){                  // Assign Relays
64    vt[0] = { "Válvula tanque 1", 0,       // Input ports to Valves
65              R2_4, R1_4, I0_8, I0_7, 0};  // Tank valve 1
66    vb[0] = { "Válvula bomba 1", 1,        //
67              R2_6, R1_6, I0_9, I0_10, 0}; // Pump valve 1
68    vc[0] = { "Válvula cuba 1", 2,         //
69              R2_7, R1_7, I0_12, I0_11, 0}; // Vat valve 1
70    sn[0] = { "Sensor cuba 1", 3,          //
71              R2_8, R1_8, I2_3, I2_2, 0};  // Vat 1
72
73    vp = {    "Válvula purga", 4,          //
74              R2_1, R1_1, I1_4, I1_5, 0};  // Purge valve
75
76    vt[1] = { "Válvula tanque 2", 5,       //
77              R2_2, R1_2, I1_2, I1_3, 0};  // Tank valve 2
78    vb[1] = { "Válvula bomba 2", 6,        //
79              R1_3, R2_3, I2_5, I2_4, 0};  // Pump valve 2
80    vc[1] = { "Válvula cuba 2", 7,         //
81              Q0_0, Q0_1, I0_3, I0_2, 0};  // Vat valve 2
82    sn[1] = { "Sensor cuba 2", 8,          //
83              Q0_2, Q0_3, I0_1, I0_0, 0};  // Vat 2
84
85    bb[0] = {vb[0].id,Q1_0,Q0_6,0};        // Pump 1
86    bb[1] = {vb[1].id,Q1_1,Q0_7,0};        // Pump 2
87  }
88
```

```
32  void initRelays(void){                          // Set off all relays
33    int c;                                        // (initial state)
34    for(c=0;c<2;c++){                             // Go through all valves
35      digitalWrite(vt[c].relay_open,LOW);         // *Tank valve
36      digitalWrite(vt[c].relay_close,LOW);        // Turn of Valves relay's
37      vt[c].state = digitalRead(vt[c].led_green)==HIGH?1:  // Set valve state
38                    digitalRead(vt[c].led_red)==HIGH?2:0;  // according to position
39                                                  // *
40      digitalWrite(vb[c].relay_open,LOW);         // *Pump valve
41      digitalWrite(vb[c].relay_close,LOW);        //
42      vb[c].state = digitalRead(vb[c].led_green)==HIGH?1:  //
43                    digitalRead(vb[c].led_red)==HIGH?2:0;  // *
44      digitalWrite(vc[c].relay_open,LOW);         // *Vat valve
45      digitalWrite(vc[c].relay_close,LOW);        //
46      vc[c].state = digitalRead(vc[c].led_green)==HIGH?1:  //
47                    digitalRead(vc[c].led_red)==HIGH?2:0;  // *
48      digitalWrite(sn[c].relay_open,LOW);         // *Vat
49      digitalWrite(sn[c].relay_close,LOW);        //
50      sn[c].state = digitalRead(sn[c].led_green)==HIGH?1:  //
51                    digitalRead(sn[c].led_red)==HIGH?2:0;  // *
52    }
53    digitalWrite(vp.relay_open,LOW);              // *Purge vavle
54    digitalWrite(vp.relay_close,LOW);             // (just one, no need
55    vp.state =  digitalRead(vp.led_green)==HIGH?1:  // to go through the cycle)
56                digitalRead(vp.led_red)==HIGH?2:0;  // *
57  }
```

Figure 53: Functions to set initial configurations.

On the left side of Figure 53, the code for the *setValves()* function is displayed, and on the right, the *initRelays()* function. It can be seen that, in the *setValves()* function, the following are assigned to all electrovalve modules:

- Two identification fields (“name” and “id”),
- Two fields that identify the relays that trigger the operation (“relay_open” and “relay_close”),
- Two fields for the digital inputs connected to the limit sensors (“led_green” and “led_red”),
- The state field (“state”) with the value zero (“not set”).

The pumps are also initialized here with the identification (“id”), the relevant digital outputs connected to the LEDs (“green” and “red”) and the status (“not working”).

After configuring all modules, the *initRelays()* function is executed. All relays connected to the modules are closed to interrupt any possible operation that may be taking place. Then, a status is assigned to the module depending on its position (0 - “not initiated” if no sensor is pressed; 1 - “Opened” / ”Empty” if the sensor connected to the green LED is pressed; 2 - “Closed”/ “Full” if the sensor connected to the red LED is pressed).

This process is performed on all valves and vats through a cycle that travels the arrays, and, finally, on the purge valve.

The program proceeds to the *loop()* cycle, where the system control algorithm is located. The code used in this section is the same as the one described in the previous chapter (Figure 44). Only the functions responsible for liquid transfer from a tank to a vat and vice versa have been changed. Figure 54 shows the code for the *loop()* function.

```
72  void loop() {
73                                          // Vats 1..6 Tanks 7..12
74      if(opRun == 0){                     // Check if no operation running
75          if (Serial.available() > 0) {   // Check for user input
76              a = Serial.parseInt();      // Get container origin
77              while(Serial.read()!='#');  // Wait for input with next container
78              b = Serial.parseInt();      // Get destination container
79              opRun = 1;                  // Ready to start operation
80          }
81      }
82      if(opRun == 1){                     // Check if is ready
83          if(a>b){                        // Check direction (tank to vat)
84              activateTC(b-1,a-7);        // Execute tranfers from specified break point (0->initial)
85          }else if(b>a){                  // Check direction (vat to tank)
86              activateCT(a-1,b-7);        // Execute tranfers from specified break point (0->initial)
87          }
88      opRun = 0;                          // Operation ended
89      }
90  }
```

Figure 54: Function running on the PLC.

As can be seen, the program is waiting for an input, via serial, with the format *(int)a # (int)b*; these two integers are compared and one of the transfer functions is performed. The difference is that now, the transfer functions' only logical foundation is the indicative value of the origin and destination containers (*int cuba* and *int tank*), unlike those in the previous chapter that received the containers (objects from the TANK and VAT structures). Figure 55 shows the new transfer functions.

```
334 int activateTC(int cuba, int tanque){           // *Initialize transfer from tank to vat
335     if(digitalRead(sn[cuba].led_red)==1)        // *When vat already full
336         return 1;                               //  End transaction, successfull
337                                                 //
338     struct VALV *v1[]={&vt[tanque],&vc[cuba]};  // *Open vat and tank valves
339     if(openValves(v1,2)==0) return 0;           //  if something wrong, end operation
340                                                 // *Turn on pump
341     fillCuba(&sn[cuba],&bb[tanque]);            //  Fill the vat
342                                                 //  Turn off pump
343     if(closeValve(&vc[cuba])==0) return 0;      // *Close vat valve
344                                                 //  if something wrong, end operation
345     struct VALV *v2[]={&vb[tanque],&vp};        // *Open pump and purge Valve
346     if(openValves(v2,2)==0) return 0;           //  if something wrong, end operation
347     delay(3000);                                //  Wait untill purge done
348
349     struct VALV *vpp[]={&vp};                   // *Close purge valve
350     if(closeValves(vpp,1)==0) return 0;         //  if something wrong, end operation
351
352     struct VALV *v3[]={&vt[tanque],&vb[tanque]};// *Close tank and pump valves
353     if(closeValves(v3,2)==0) return 0;          //  if something wrong, end operation
354
355     return 1;                                   // *Return operation successfull
356 }
357
358
```

```
436 int activateCT(int cuba, int tanque){          // *Init vat to tank transfer
437     if(digitalRead(sn[cuba].led_green)==1){     // *When vat already empty
438         return 1;                               //  End transaction, successfull
439     }
440
441     struct VALV *v1[]={&vt[tanque],&vb[tanque]}; // *Open tank and pump valves
442     if(openValves(v1,2)==0) return 0;           //  if something wrong, end operation
443
444     if(openValve(&vc[cuba])==0) return 0;       // *Open vat valve
445                                                 //  if something wrong, end operation
446                                                 // *Turn on pump
447     emptyCuba(&sn[cuba],&bb[tanque]);           //  Empty Vat
448                                                 //  Turn off pump
449     if(closeValve(&vc[cuba])==0) return 0;      // *Close vat valve
450                                                 //  if something wrong, end operation
451     if(openValve(&vp)==0) return 0;             // *Open purge valve
452                                                 //  if something wrong, end operation
453     delay(3000);                                //  Wait purge
454     if(closeValve(&vp)==0) return 0;            //  Close purge valve
455                                                 //  if something wrong, end operation
456     struct VALV *v2[]={&vt[tanque],&vb[tanque]}; // *Close tank and pump valves
457     if(closeValves(v2,2)==0) return 0;          //  if something wrong, end operation
458
459     return 1;                                   // *Return operation successfull
460 }
```

Figure 55: Transfer functions.

These functions, like those of the previous system, follow the flow diagram in Figure 89 in Appendix 5. They start by checking if the tank is already full, when transferring from a tank to a vat, or empty, otherwise. When neither status is confirmed, the operation continues. At times, when it is possible to open or close more than one valve, an array is created with the valves that will run and be provided to one of the opening or closing functions (*openValves()* or *closeValves ()*) together with the number of valves in the array. It should be noted that now, if one of the valve-operating functions returns zero (i.e. it was not completed successfully), the

transfer will be interrupted by also returning the value zero; otherwise, at the end of the transfer the value 1 is returned. This last feature was used on later implementations.

4.4.1 Implemented functionalities

To make the system more realistic, it was convenient to add some features to the algorithm that could prevent possible problems from a real system implementation such as equipment malfunction, as well as others that allow a broader interaction.

System status diagnosis

Before starting a racking, it is necessary to check that all valves are in the rest position. This is because if any of the valves, intervening or not in the process, is in the wrong position, it may condition the liquid's path, and when starting the transfer, it is possible that it flows into the wrong container, or that it mixes with another liquid that is not supposed to be there, which could cause significant damage. The considered way to solve this problem was to check and, if necessary, reposition the valves. For this purpose, a function was created, *checkValvesReady()*, which, through an array of valves, checks whether all of them are closed. If any of them are, the function returns zero. At the beginning of the two transfer functions, the previous function is run and, if any valve is open, the *closeAllValves()* function is executed, closing all valves and proceeding with the process. Figure 56 shows the *checkValvesReady()* function, in section (a), and the way it is applied at the beginning of the *activateTC()* and *activateCT()* functions, in section (b).

```
618  int checkValvesReady(void){                                  // Check valves position to see if they'r ready
619    uint8_t i;                                                 //
620    struct VALV v[]={vt[0],vb[0],vc[0],vp,vt[1],vb[1],vc[1]}; // Valves array
621
622    for(i=0;i<sizeof(v)/sizeof(*v);i++){                       // Iterate through the valves array
623      if(v[i].state!=2){                                       // if not all valves are closed
624        return 0;                                              // Return not Ready
625      }
626    }
627    return 1;                                                  // Return valves ready (when all are closed)     (a)
335    if(!checkValvesReady()) closeAllValves();                  // Check if all valves are in the correct position,
336                                                               // if not, close them                            (b)
```

Figure 56: (a) – Function to check valves' initial state; (b) – Application.

Malfunctioning valves

In a real system, it is possible for the valves to be damaged for some reason, and/or to stop responding. Therefore, the system is required to have the capability of identifying the failure of one or more valves, and to apply preventive solutions. The implemented solution consists of defining a time interval (that each valve must have) to transit from one position to another; exceeding this interval finishes the operation in process. In the functions *closeValves()* and *openValves()*, code was added that performs the described process (such code is identified in red in Figure 57). In the first block, the recording of the time taken during the instant immediately after the activation of the valves occurs ; then, valves must take their time to complete the operation. In the wait cycle, the second code block was added, which checks the time elapsed since the activation. If it exceeds six seconds (value defined in VALVE_TIMEOUT), the valves that remain in the "not set" state are checked,

as they are the ones that did not transit, and are then updated to the “Not responding” state, and finally a value of zero is returned indicating that the operation has failed, causing the current transition to end.

```
119  int closeValves(struct VALV *v[], size_t size){                  // Close selected valves
120      long unsigned int i = 0,end = 0, cnt = 0, time = 0;          //
121      for(i=0; i<size;i++){                                        // Iterate through selected valves
122        if (digitalRead(v[i]->led_red)!=1 && v[i]->state!=3){      // Only valves not closed and responding
123          digitalWrite(v[i]->relay_close, HIGH);                   // Turn on close relay
124          v[i]->state=0;                                           // Set valve state to "not set"
125          cnt++;                                                   // Keep track of the valves closing
126        }
127      }
128      time = millis();                                             // Save instant time (ms)
129      while(end<cnt){                                              // Iterate until all valves close,
130                                                                   //  excluding the already closed ones
131          for(i=0; i<size;i++){                                    // Iterate through selected valves
132              if(digitalRead(v[i]->led_red)==1 && v[i]->state==0){  // If valve closed and state "not set"
133                  digitalWrite(v[i]->relay_close, LOW);            // Turn off close relay
134                  v[i]->state = 2;                                 // Set Valve state to "closed"
135                  end+=1;                                          // Increment closed valves count
136              }
137          }
138          if((millis()-time)>=VALVE_TIMEOUT){                      // If it takes to long something is wrong
139            for(i=0; i<size;i++){                                  // Iterate through all selected valves
140              if(v[i]->state==0){                                  // if Valve in not set
141                  digitalWrite(v[i]->relay_close, LOW);            // Turn off close relay
142                  v[i]->state=3;                                   // Change state to not responding
143              }
144            }
145            return 0;                                              // One or more valves are not responding
146          }                                                        // End operation
147      }
148      return 1;                                                    // All valves closed successfully
149  }
```

Figure 57: Function to close valves with valves timeout.

Electrovalves status change

Subsequently to the previous point, it was necessary to develop a method that would allow the user to carry out the proper damaged valve maintenance, and then to set the system functional again. To perform this procedure, the *setValveState()* function was developed (shown in Figure 58). This function changes the user-provided state of a specific valve, and, for this, it is founded on a logic resulting from the valve’s “id” field and the desired state field (“state”). An array of all valves is defined, and the status of the valve in the “id” position is updated to “state”. The user can define the status of the valve by entering the command *&(int)id*. Since the only functionality of this function is to update the status of valves that have been fixed, it is only used to define the status as “not set”. In other words, when the user sends the previous command, the function is executed with the “id” supplied and the zero state, *setValveState(id,0)*, changing the state from “not responding” to “not set”, which turns the system functional again. The command interface will be discussed later in this chapter.

```
603  void setValveState(int id, int state){                           // Set Valve State
604    struct VALV *v[]={&vt[0],&vb[0],&vc[0],&sn[0],                 //
605                      &vp,&vt[1],&vb[1],&vc[1],&sn[1]};            // Array with all valves
606    v[id]->state=state;                                            // Set valve (id) with state (state)
607  }
```

Figure 58: Function to set valve state.

Still within the scope of changing the status of the valves, a function was developed that would allow the user to change the valve's position. As can be seen in Figure 59, the *changeValveState()* function's logical input is "id", which identifies the valve chosen for transition, and, if it is open or "not set", the valve is closed. If it is already closed, it proceeds with its opening. The function can be performed using the command "*".

```
610  void changeValveState(int id){                          // Change the state of the valve(id)
611    struct VALV *v[]={&vt[0],&vb[0],&vc[0],&sn[0],&vp,    // depending on the current state
612                      &vt[1],&vb[1],&vc[1],&sn[1]};        // Valves array
613    if(v[id]->state==1 || v[id]->state==0){                // When valve is not set or opened
614      closeValve(v[id]);                                  // Close valve (ID)
615    }else if(v[id]->state==2){                            // When valve is closed
616      openValve(v[id]);                                   // Open Valve (ID)
617    }else Serial.println("Valve not responding");         // When valve is not responding do nothing
618  }
```

Figure 59: Function to change a valve position.

Stop in the middle of an operation

Since there may be some power failure and the system will then go down, which causes it to start from the initial state, a method was developed so that the operation status could be recovered after the restart. Next, the best way to recover the staging point after a system reboot would be using the Arduino flash memory. Through the "EEPROM" library [60]provided by the Arduino developers, it is possible to read and write information into the flash memory that will not be lost even after the processor is turned off. From this library, only the functions of writing and reading in memory (*EEPROM.read(address)* [61] and *EEPROM.write(address)* [62], respectively) were required, which implies that each memory address represents 1 byte of information.

First, it was defined that it would be necessary to keep only four fields: the identification of the tank and the vat in operation, the direction of the transfer and the stopping point. All these fields can be represented numerically, and none of them exceeds the value 15, which allows the use of only four bits per field, that is, only two bytes of the flash memory need to be used to store this information. Therefore, the following organization was used:

Table 8: Flash memory coding

Address	Bits 7-4	Bits 3-0
0	Vat id	Tank id
1	Direction (op)	Break Point (bp)

Afterwards, two functions were developed to interact with the memory: one for writing and the other for reading, both represented in Figure 60. The *saveState()* function's logical inputs are the four mentioned fields - in case the operation is transferring from a vat to a tank (op = 2) or from a tank to a vat (op = 1) - which are written on the flash, as referred to in Table 8. In the case where the system is not in operation (op = 0), the two bytes are filled with ones, which indicates that the system is at rest.

The write function, *getState()*, is run with two pointers that represent the origin and destination containers (* a and * b, respectively). Here, addresses zero and one from memory are read and stored in two variables ("containers" and "states"). Then, depending on the four most significant bits of address one, the transfer's

direction is identified, and pointers a and b are updated with the number of the relevant container, coming from the zero address of the memory. At the end, the stop point stored in the memory's least significant four bits of address one is returned. In case the system is not in operation, a zero value is returned, indicating that the system is at rest.

```
557 void saveState(int op, int bp, int cuba, int tanque){// Save state into EEPROM
558   switch(op){                                       // Depending on the operation
559     case 1:                                         // Case operation is transfer from tank to vat
560       EEPROM.write(0,cuba<<4|tanque);               // Write address 0 vat(4 MSB) tank (4 LSB)
561       EEPROM.write(1,op<<4|bp);                     // Write address 1 operation (4 MSB)
562       break;                                        // breakpoint (4 LSB)
563     case 2:                                         // Case operation is tranfer form Vat to Tank
564       EEPROM.write(0,cuba<<4|tanque);               // Write address 0 vat(4 MSB) tank (4 LSB)
565       EEPROM.write(1,op<<4|bp);                     // Write address 1 operation (4 MSB) and the
566       break;                                        // breakpoint (4 LSB)
567     case 0:                                         // Case not in operation
568       EEPROM.write(0,255);                          //
569       EEPROM.write(1,255);                          //
570       break;
571     default:
572       break;
573   }
574 }
576 int getState(int* a, int* b){                       // Read operation state from EEPROM
577   int containers = EEPROM.read(0);                  // Get the receipients from address 0
578   int states = EEPROM.read(1);                      // Get operation and breakpoint from address 1
579
580   switch(states>>4){                                // Depending on operation
581     case 1:   //T->C                                // Case operation was tranfers from tank to vat
582       *a = (containers&15) + 7;   //tanque         // Get tank (4 LSB + 6 vats + 1 ignore 0 )
583       *b = (containers>>4) + 1;   //cuba           // Get vat (4 MSB + 1 ignore 0)
584       return states&15;                             // Return breakpoint (4 LSB)
585     case 2:   //C->T                               // Case operation was tranfers from vat to tank
586       *a = (containers>>4) + 1;                     // Get vat (4 MSB + 1 ignore 0)
587       *b = (containers&15) + 7;                     // Get tank (4 LSB + 6 vats + 1 ignore 0 )
588       return states&15;                             // Return breakpoint (4 LSB)
589     case 15:                                        // Case no operation
590       return 0;                                     // Return initial breakpoint (0)
591     default:
592       return 0;                                     // Return initial breakpoint (0)
593   }
594 }
```

Figure 60: Function to enable EEPROM reading and writing.

In order to make it possible to recover the operating state, the *saveState()* function is run within the transfer (*activateTC()* and *activateCT()* functions), each time one or more valves finish opening or closing, as shown in green in Figure 61. A stopping point was always considered before a valve or vat transition. Therefore, in a tank-to-vat transfer function, six stopping points are identified, and in a vat-to-tank one, seven. Note that, at the end of the transfer, the stopping point returns to zero.

In addition, each of the transfer functions had to be provided with two logical inputs: the breakpoint (bp) and the indication of whether the function should run normally or not (run). If the run logical input is 1, the transfer will occur normally; if it is zero, the transfer will occur only from the last break point, bp, that is, from the last successful valve transition, because the condition, in red in Figure 61, is applied to all transitions.

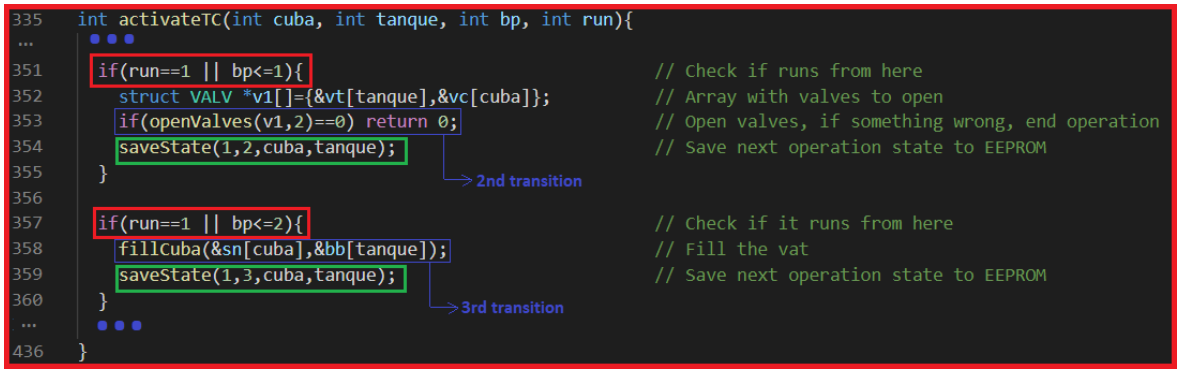


Figure 61: Chunk of code from activateTC() with break points and state saving.

With these features added, it remains to add the code responsible for identifying the last stopping point and telling the system how to proceed. This was achieved with the addition of the code chunk in Figure 62 at the end of the PLC *setup()* function. Initially, checking occurs as to whether the system stopped in the middle of a transfer or not. This is carried out through the *getState()* function, which returns the stopping point stored in the variable bp and updates the pointers a and b with the containers of the last transition. If bp is zero, the system proceeds normally as if nothing has been interrupted; if not, the program waits for a user input starting with the character "$". Then, if the user enters "continue", the system proceeds from the point where the transfer it was stopped; if he enters anything different, the system is reverted to its initial state, the flash addresses, the stop point and the container variables are reset, and normal operation continues.

```
bp=getState(&a,&b);                    // Get last state of operation
 if(bp!=0){                            // Operation interrupted in the middle of a proccess;
   while(Serial.read()!='$');          // Wait for User feedback;
   if(Serial.readString()=="continue"){// User chose to continue interrupted operation;
    flag=1;                            //
    run=0;                             // Set flags to start from break point;
   }else{                              // User chose end last operation and set the system to the initial state
     closeAllValves();                 // Close all valves
     saveState(0,0,0,0);               // Save initial state in EEPROM
     bp=getState(&a,&b);               // Set breakpoint ot zero (initial state)
   }
 }else{                                // No operation interrupted
   run =1;                             // Set flags to normal procceedure
 }
```

Figure 62: Code to restart from break point

A method was also developed that enables users to stop the system in the middle of a transfer, on a voluntary basis, with the possibility of later continuing from the same point (or not). This functionality was achieved by adding the red code line of Figure 63 to the transfer functions, before any operation with a valve or vat. Thus, if the user sends the character "_" in the middle of a transfer, zero ends up being returned. Then, the system is waiting for a command that can be "+ continue", which will make the program resume the stopped operation, or another command associated with the options mentioned above.

```
335  int activateTC(int cuba, int tanque, int bp, int run){
...    • • •
365    if(Serial1.available()>0) if(Serial1.read()=='_') return 0;    // Check for stop signal
366    if(run==1 || bp<=2){                                           // Check if it runs from here
367      fillCuba(&sn[cuba],&bb[tanque]);                             // Fill the vat
368      saveState(1,3,cuba,tanque);                                  // Save next operation state to EEPROM
369    }
370    if(Serial1.available()>0) if(Serial1.read()=='_') return 0;    // Check for stop signal
371    if(run==1 || bp<=3){                                           // Check if it runs from here
372      if(closeValve(&vc[cuba])==0) return 0;                       // Close valve, if something wrong
373      saveState(1,4,cuba,tanque);                                  // Save next operation state in EEPROM
374    }
...    • • •
436  }
```

Figure 63: Chunk of code from activateTC() (with stop command).

Interface Serial

After all the features reported in the previous points, the implementation to execute them will now be described. The interface was inserted in the *loop()* control cycle, using the C language switch / case method. When the user enters a command at a time when the program is at rest, the first character is stored in variable “c”, then the case to which “c” corresponds is executed. Note that, if more information is required from the user, it is possible to send the character of the desired command followed by other characters that will be associated to the second field of the initial command. All commands that can be executed by the user are listed in Table 9.

Table 9: Serial commands to interact with the system.

1st cmd	2nd cmd	Function	When it can be used	Observations
#	(int)a#(int)b	Sets "a" as origin recipient and "b" as the destination	System waiting for input	vats 1-6; tanks 7-12:
@	cal	Initiates calibration	System waiting for input	
&	valve id	Sets valve with specified id to state "not set"	System waiting for input	
*	valve id	Changes the states of valve with specified id	System waiting for input	("not set", "opened")->"closed"; “closed”->"opened"
+	continue	Continues transfer from last break point	System waiting for input	Used after transfer interrupted, otherwise does nothing
_	not defined	Stops operations	During transfer	Operation can be resumed with "+continue"
$	continue	Continues transfer from last break point	System setup	Used after system reboot in mid operation

4.5 *Interface with the WiFi module*

As already mentioned, the PLC has a WiFi module integrated, ESP32. Therefore, it was decided to implement the system interface on a webserver allocated in the module. For this, it is necessary to send the information about the PLC-read status of the system elements to the ESP32 module, so that it can represent the system in process on the webserver; it is also necessary to transmit the user's options back to the PLC.

The Arduino and ESP32 are internally connected by UART, which indicates that it is possible to send and receive, in both directions, frames of information in the same way that until now has been utilized by the user,

that is, via serial. This communication can be carried out through UART 1 of Arduino (Serial1) and UART 2 of ESP32 (Serial2), with the proper initialization of the serial ports in the *setup()* function.

Starting with the transmission from the PLC to the WiFi module, the status of all system modules will have to be forwarded whenever any change occurs. To do so, the json data format was considered, because it is relatively easy to generate and analyse by machines and humans and has a relatively small size,.

Through the "ArduinoJson" library, a function was developed to send an information grid with the identification of the electrovalve modules and tanks and their states. This function can be seen in Figure 64.

```
314  void sendAll(void){                                    // Send all valves states to interface
315
316    long unsigned int i = 0;
317    const size_t capacity = 2*JSON_ARRAY_SIZE(9) +       //
318                            JSON_OBJECT_SIZE(2);         // Define size of json file
319    DynamicJsonDocument doc(capacity);                   // Initialize Json document
320
321    struct VALV *v[]={&vt[0],&vb[0],&vc[0],&sn[0],&vp,   //
322                      &vt[1],&vb[1],&vc[1],&sn[1]};      // Array with all valves
323    JsonArray valve = doc.createNestedArray("valve");    //
324    JsonArray state = doc.createNestedArray("state");    // Give json document id and state parameters
325
326    for(i = 0;i<sizeof(v)/sizeof(*v);i++){               // Iterate for all valves
327      valve.add(v[i]->id);                               // Store ID
328      state.add(v[i]->state);                            // Store state
329    }
330    serializeJson(doc, Serial1);//S1                     // Send document to ESP
331    Serial1.flush();//S1                                 // Clear Tx buffer
332  }
```

Figure 64: Function responsible to send modules states to ESP32.

The *sendAll()* function starts by generating a json document whose size is dependent on the number of fields. It includes nine modules, the objects associated with those fields, the state of the modules, and parameters that are then created in the document, "valve" and "state" respectively. Subsequently, through an array with all modules, the module's "id" and its respective state ("state") are added to the document. Finally, the document is sent to Serial1, which, as already mentioned, is associated with ESP32. This function is run at all times when there is a change in the status of one or more modules (i.e. after using the *openValves()* function or at the end of the *initRelays()* function).

In addition to the status of the modules, it is also necessary to convey the state which the system's operation is in. In this case, the approach taken was simply to send to Serial1 the instruction *%(int)state*, where "state" represents an indicative number of the state, as indicated in Table 10.

Table 10: Indicative commands of the operation state.

Command	State	Command	State
%0	Init transaction	%9	Cuba full
%1	Open VT and VC	%10	Open VT and VB
%2	Filling Vat	%11	Open VC
%3	Close VC	%12	Draining Vat
%4	Open VB and VP	%13	Open VP
%5	Purging	%14	Cuba empty
%6	Close VP	%15	Calibrating
%7	Close VT and VB	%16	Operation interrupted
%8	End transaction		

Once the data sent from Arduino to ESP32 is defined, it remains to be seen how ESP32 receives and treats system information. As the Arduino code editor has a programming interface dedicated to the ESP32 board, it is possible to program this module in a similar way to Arduino. To read and decode the json files received from the PLC, the library “ArduinoJson” was used, resulting in the *readJson()* function described in Figure 65. This function’s logical grounds result from a document in the json format, which will be saved in another json document with the same capacity. Subsequently, decoding is carried out, and in the variable “Valve”, the “id” of the modules is inserted, and, in the variable “State”, the relevant states are entered. Then, the states are stored in an array ("states") with the position indicated by the corresponding "id".

```
113 void readJson(String json){                          // Parse json received with valves states
114   int i, index, state;
115   const size_t capacity =   2*JSON_ARRAY_SIZE(9)
116                           + JSON_OBJECT_SIZE(2)+20;    //
117
118   DynamicJsonDocument doc(capacity);                   //
119
120   deserializeJson(doc, json);                          // Parse Json
121   JsonArray Valve = doc["valve"];                      // Store valves ids
122   JsonArray State = doc["state"];                      // Store valves states
123     for(i = 0; i<Valve.size(); i++){                   // For all valves
124         index = Valve[i];                              //
125         state = State[i];                              //
126
127         states[index]=  (state == 0) ?"Notset":        //
128                         (state == 1) ?"Opened":        //
129                         (state == 2) ?"Closed":        //
130                                       "Not responding";// Change state to corresponding
131     }
132 }
```

Figure 65: Function to parse json document.

The reading of the data received on the PLC's Arduino is performed in the *loop()* control cycle, as shown in Figure 66. Here, the first character of the received frame is read. If it is “%” the system has entered in a new state of operation; therefore, the global “status” variable responsible for storing this information is updated according to the “AllStatus” array, which contains all the states referred to in table 10. In case the first character

received happens to be "{", a json file is received; it is subsequently read in its entirety and incorporated into the *readJson()* function.

```
88  void loop(){
89    if(Serial2.available()>0){                // If Cmd received from PLC
90      char c = Serial2.read();                // Store Cmd
91      if(c=='%'){                             // New status Cmd
92        int ind = Serial2.parseInt();         // Get new status
93        status = AllStatus[ind];              // Change status
94      }else if(c=='{'){                       // Received valves states
95        String json;
96        json = Serial2.readStringUntil('}');  // Read json doc from PLC
97        readJson("{"+json+"}");               // Parse Json
98      }
99    }
100 }
```

Figure 66: ESP32 loop() function where data from PLC is read.

4.5.1 Webserver

Next, the interface is designed where: i) the data indicating the system's state will be exposed and ii) it will be possible to interact with it through a web page. To do so, it is initially necessary to connect the ESP to a wireless network, that is possible with the use of the "WiFi" library. In the code shown in Figure 67, it is observed that the network data is provided, and in the *setup()* function the connection is initialized and followed by its verification until it is stable.

```
12  const char* ssid = "Network-ID";            // ID
13  const char* password = "password";          // PASS
      ...
72  setup(){
      ...
77    WiFi.begin(ssid, password);               // Connect to Wi-Fi
78    while (WiFi.status() != WL_CONNECTED) {   // Repeat until connected
79      delay(1000);                            // 1 sec
80    }
      ...
86  }
```

Figure 67: ESP32 WiFi connection.

Then, with the WiFi module connected to a network, it is necessary to generate a webserver to allocate the interface page. The "ESPAsyncWebServer" library allows ESP32 to create an asynchronous server on the ip assigned by the network. Figure 68 shows the necessary structure for ESP to run the server. In block 1, included is a library where the code of the webserver-represented html page is stored and, in block 2, the *processor()* function, which is executed by the webserver to replace specific fields in the html code. Block 3 contains the function responsible for managing the server and all requests that will be made. For now, a single request is

observed: the one requiring to run the html page created on the server with the necessary replacements undertaken by the *processor()* function. A reference must also be made to block 4, where the instruction that the server will start is present.

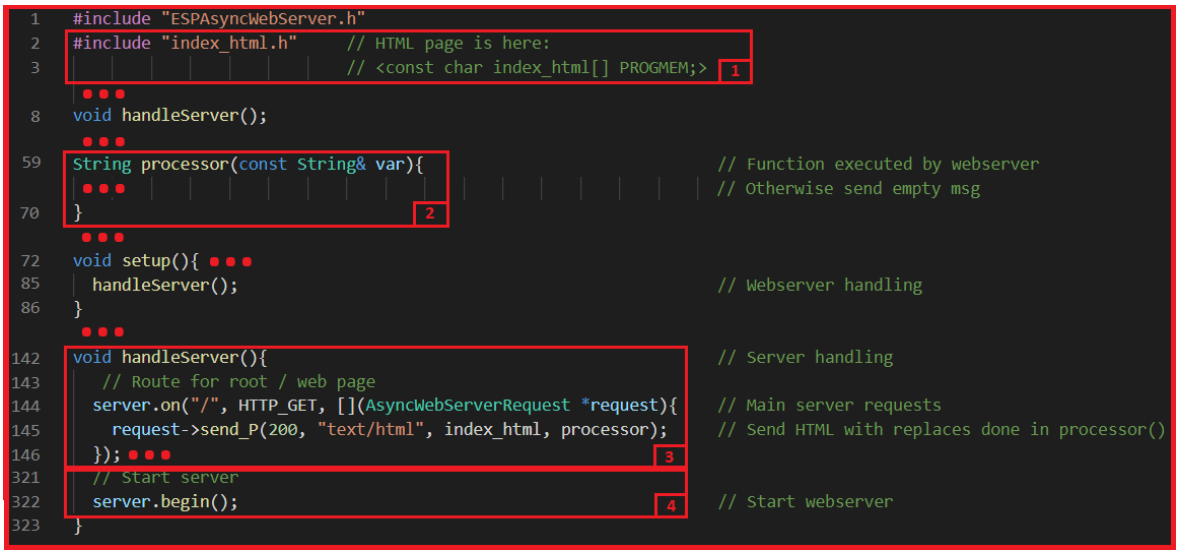


Figure 68: Structure for webserver handling.

Then, interface implementation proceeded. To explain all its elements, Figure 69 shows the final model obtained from the interface where the different elements were identified by numbers. Next, all the code developed to implement each element will be described (both the one applied directly to the ESP and the relevant html code).

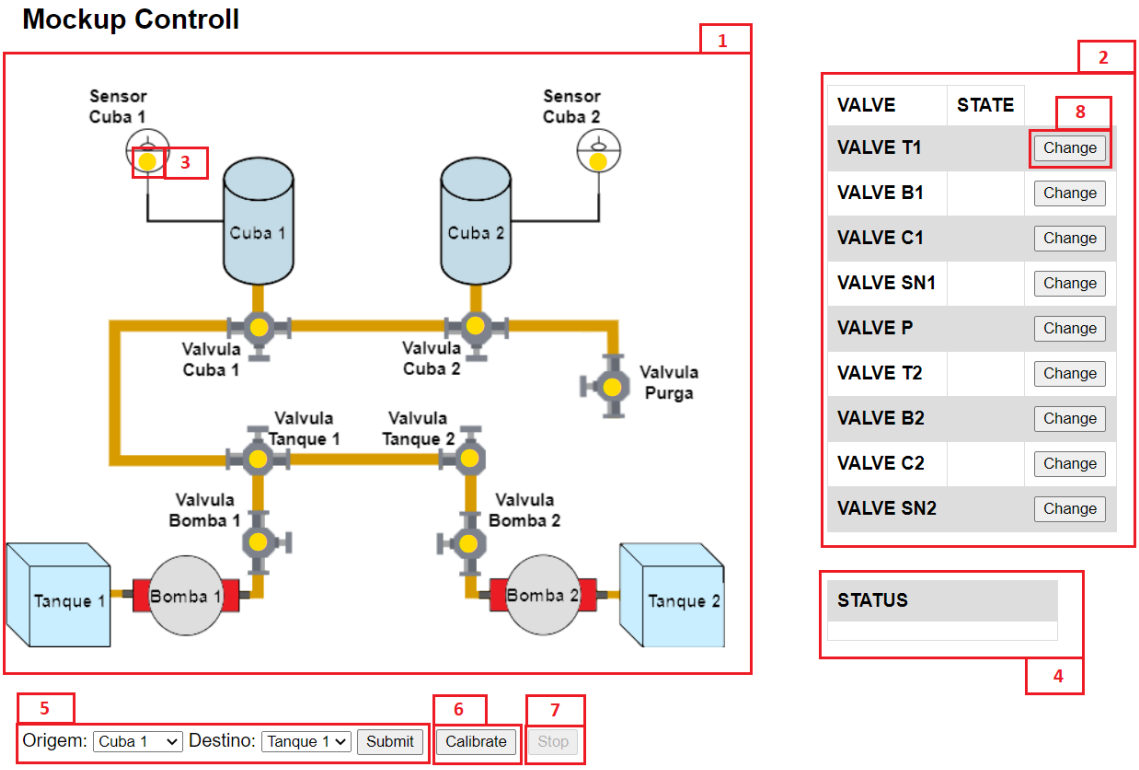


Figure 69: Developed interface.

Block 1

In the block marked with 1, the image representing the system is present. The image was uploaded to an image hosting server, imgur, and it is incorporated into the html code (in the style defined by the upper block code in Figure 70) and through the instruction contained in the lower block with a size of 200 by 240.

```
img {
    position: absolute;
    left: 0px;
    top: 100px;
    z-index: -1;
    height: auto;
    width: auto;
    background: rgb(255, 255,255);
}
```

```
<img src="https://i.imgur.com/L9e8dM8.png" width="200" height="240">
```

Figure 70: Interface image html code.

Block 2

Block 2 represents a table that contains all the valves and their status. Through the html code of the table represented in Figure 71 (a), it is possible to check the markers where the status of each module will be present (second column). The *processor()* function, shown in Figure 71 (e), checks whether the marker corresponds to

the desired module, comparing it with the “valves” array elements, in (d), and replacing it with the appropriate state indicated in the “states” array.

As the modules change their status, the table must follow the changes. That is why the *updateField()* function was developed, which performs a “GET” request to the server of the specified field, and is run by the *setInterval()* function, which makes requests every second, as can be seen in Figure 71 (b) and (c). On the ESP side, requests are answered according to their address (“uriv”) and through the code snippet displayed in (f), which returns to the server the status of the request module present in the “states” array.

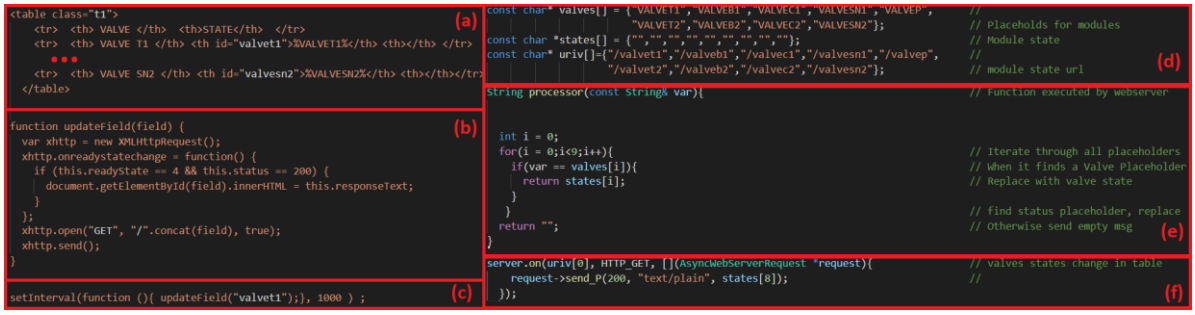


Figure 71: State table implementation.

Block 3

In this block, there is another element indicating the status of the modules. These circles are positioned in the image above each represented module, and can come in three different colours (green, red or yellow) indicating open valve or empty vat, closed valve or full vat, and module in indefinite state, respectively. In order for the colour of the circles to follow that of the valve’s status, the code chunk in Figure 72 in the red block was added to the *readJson()* function, which updates the colour (as stored in the “color” array) according to the state, as indicated above.

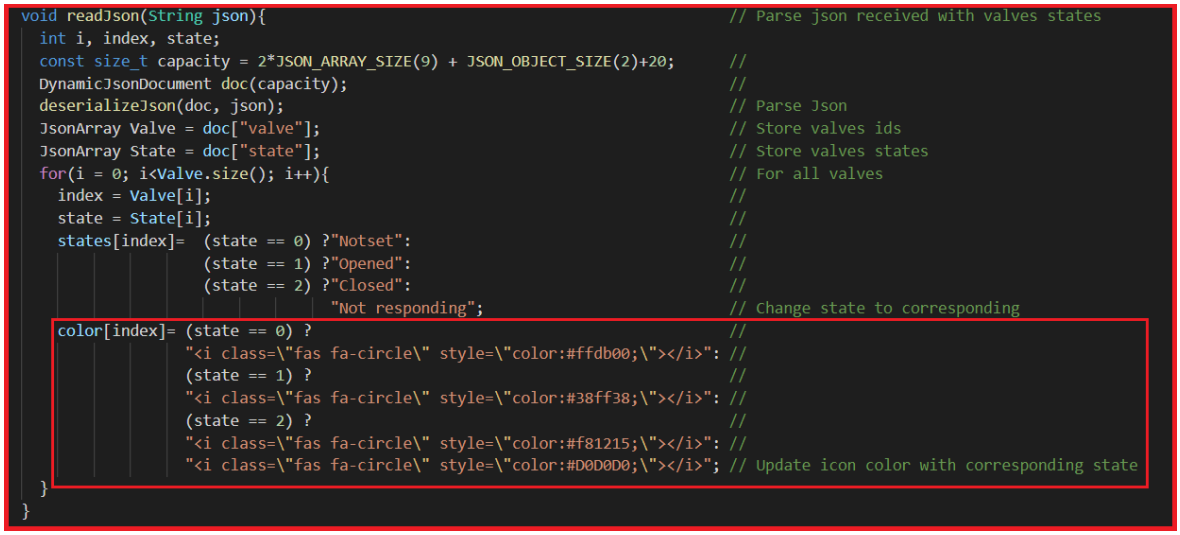


Figure 72: Function to parse json files with icon colour update.

The rest of the implementation is identical to the previous one. In this case, code was added to the *processor()* function to replace the markers with circles with the appropriate colours. Regular update requests are also

performed with the *setInterval()* function, which calls the *updateField()* function with the respective circle address. The requests are fulfilled by ESP in the same way, now with a different array of addresses.

Block 4

The small table represented in this block is where the status of the system's operation is indicated. As already mentioned, within the ESP, this information is stored in the "status" variable, which is why another line was added to the *processor()* function to replace the marker with the transfer status, as shown in blue in Figure 73(c). The page code builds a table with two lines – see Figure 73(a) - and the second is where the status of the operation is displayed, which makes regular requests to update itself, in the same way as it was done previously: through of the *setInterval()* and *updateField()* functions, as shown in section (b) of Figure 73. Again, the request is answered through the code snippet in (d).

```
<div class="status">
    <table class="t2">
      <tr></tr>
      <tr><th>STATUS</th></tr>
      <tr><th><div id="status">%STATUS%</div></th></tr>
    </table>
  </div>
```
(a)

```
setInterval(function (){ updateField("status");}, 1000 ) ;
```
(b)

```
  String processor(const String& var){                          // Function executed by webserver
  int i = 0;
  for(i = 0;i<9;i++){                                           // Iterate through all placeholders
    if(var == valves[i]){                                       // When it finds a Valve Placeholder
      return states[i];                                         // Replace with valve sate
    }else if(var == ctoggles[i]){                               // When it find an icon placeholder
      return color[i];                                          // Replace with valve state corresponding color
    }
  }
  if(var == "STATUS") return status;                            // When it find status placeholder, replace with status msg
  return "";                                                    // Otherwise send empty msg
}
```
(c)

```
  server.on("/status", HTTP_GET, [](AsyncWebServerRequest *request){    // Status update
    request->send_P(200, "text/plain", status);                         // Send request with the current status
  });
```
(d)

Figure 73: Operation status implementation.

Block 5

This block is where the user can choose the transfer to be executed. The user must pick the desired source and destination container, which is present in the form, and then make the submission. This form was built using the code shown in Figure 74 on the left, and when submitting, a request will be made to the server with the address /get, which contains the indication of the chosen containers. Note that the choice of the second option is limited by the first, since transfers can only occur from a vat to a tank and vice versa, which is performed through the *show_select()* function shown in Figure 74 on the right.

```
<form id="command" action="/get">

   Origem: <select name="O" id='main_select' onchange='show_select()'>
   <option value="1">Cuba 1</option>
   <option value="2">Cuba 2</option>
   <option value="7">Tanque 1</option>
   <option value="8">Tanque 2</option>
   </select>

   Destino:
   <select name="D" id='select_1' >
   <option id='op1' value="1" style='display:none'>Cuba 1</option>
   <option id='op2' value="2" style='display:none'>Cuba 2</option>
   <option id='op3' value="7" style='display:' selected>Tanque 1</option>
   <option id='op4' value="8" style='display:'>Tanque 2</option>
   </select>
   <input type="submit" value="Submit" id="submbutton">
</form>
```

```
function show_select()
{
  var main_select = document.getElementById("main_select");
  var op1 = document.getElementById("op1");
  var op2 = document.getElementById("op2");
  var op3 = document.getElementById("op3");
  var op4 = document.getElementById("op4");
  var desired_box = main_select.options[main_select.selectedIndex].value;
  if(desired_box == 7 || desired_box == 8 ) {
    op1.selected="selected";
    op1.style.display = '';
    op2.style.display = '';
    op3.style.display = 'none';
    op4.style.display = 'none';
  } else {
    op3.style.display = '';
    op4.style.display = '';
    op3.selected ="selected";
    op1.style.display = 'none';
    op2.style.display = 'none';
  }
}
```

Figure 74: Form for user input.

The request made in the submission is answered by the server from the chunk of code on the bottom block of Figure 75. Here, if the parameters exist, they are assigned to the “Origin” and “Destination” variables, followed by the operation status change to “Init transaction”. The “s2e” variable is also set to 1, which will cause the *#Origin#Destination* instruction to be sent to the PLC in the loop cycle, in order to proceed with the transfer.

```
void loop(){
 if(s2e){                                                    // When Operation is submited
    Serial2.print("#"+Origem + "#" + Destino);               // Send operation request to PLC
    s2e = 0;                                                 // Set sumbtion signal to zero
  } ● ● ●
}
```

```
server.on("/get", HTTP_GET, [] (AsyncWebServerRequest *request) {          //
    if (request->hasParam(PARAM_INPUT_1) && request->hasParam(PARAM_INPUT_2)) {   // If Containers were chosen
      Origem = request->getParam(PARAM_INPUT_1)->value();                  // Get origin container
      Destino = request->getParam(PARAM_INPUT_2)->value();                 // Get destination container
      status = AllStatus[0];                                               // Change status to Init transaaction
      s2e = 1;                                                             // Set submited operation signal
    }
    else {                                                                 //  else
      Origem = "0";                                                        //
      Destino = "0";                                                       // Change params to be ignored
    }
    request->redirect("/");
  });
```

Figure 75: Form submission handling.

Block 6

The user has the option to reposition all modules to return the system to its initial state, through the calibration operation. Block 1 in Figure 76 contains the code used to create a button that, when pressed, performs the function of block 2 that sends a request to the webserver “/calibration” address. When the ESP receives the calibration request, it sends the command “@cal” to the PLC, as described in block 3 of Figure 76, and, if the system is at rest, the calibration starts.

```
181     <input type="button" id="calibration" onclick="calibration()" value="Calibrate">                    1

337 function calibration() {
338   var xhttp = new XMLHttpRequest();
339   xhttp.onreadystatechange = function() {
340     if (this.readyState == 4 && this.status == 200) {
341       document.getElementById("calibration").innerHTML = this.responseText;
342     }
343   };
344   xhttp.open("GET", "/calibration", true);
345   xhttp.send();
346 }                                                                                                       2

316   server.on("/calibration", HTTP_GET, [](AsyncWebServerRequest *request){  // User chooses to calibrate
317     Serial2.print("@cal");                                                 // Send calibration cmd to PLC
318     request->send(200);
319   });                                                                                                   3
```

Figure 76: Calibration implementation.

Block 7

Within this block is the “stop” button that allows the user to finish a transfer in progress. Figure 77 shows, in block 1, the construction of the button, which when pressed will execute the *stop()* function of block 2. In this function, the operation and calibration buttons are disabled, and two new buttons, “Continue” and “End”, are available, allowing the user to continue with the operation or to finish and return to the initial state, respectively (the implementation of these two buttons will be described later). Then, a GET request is made with the address “/stop” to the Webserver, and when it is answered, ESP32 sends the command "_" to the PLC (block 3 of Figure 77), which stops the operation.

```
183   <div class="stop"><input type="button" id="stop" onclick="stop()" value="Stop" disabled></div>        1

246 function stop(){
247   var xhttp = new XMLHttpRequest();
248   xhttp.onreadystatechange = function() {
249     if (this.readyState == 4 && this.status == 200) {
250       document.getElementById("submbutton").disabled=true;
251       document.getElementById("calibration").disabled=true;
252       document.getElementById("blockbreak").style.display='';
253       document.getElementById("blockcontinue").style.display='';
254       document.getElementById("stop").disabled=true;
255     }
256   }
257   xhttp.open("GET", "/stop", true);
258   xhttp.send();
259 }                                                                                                       2

204   server.on("/stop", HTTP_GET, [](AsyncWebServerRequest *request){  // User chooses to stop operation
205     Serial2.print("_");                                             // Send cmd to PLC to stop operation
206     status = AllStatus[16];                                         // Change status to operation interrupted
207     request->send(200);
208   });                                                                                                   3
```

Figure 77: Stop operation button.

In Figure 77, the “stop” button is disabled, because there is no operation in progress. When a transfer starts, the button becomes available until the system finishes the operation. Figure 78 shows that the availability of the button is dependent on the state of the system, which is regularly checked with a 1 second period. This request is handled on the “/status” address by the ESP32 as previously mentioned in block 4.

```
setInterval(function (){
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("status").innerHTML = this.responseText;
      if(this.responseText=="Init transaction" || this.responseText=="Open VT and VC" || this.responseText=="Open VT and VB" ){
        document.getElementById("stop").disabled=false;
      }else if(this.responseText=="End transaction"){
        document.getElementById("stop").disabled=true;
      }
    }
  };
  xhttp.open("GET", "/status", true);
  xhttp.send();
}, 1000 ) ;
```

Figure 78: State check to set stop button availability.

Block 8

In the case that a valve is in the wrong position, the "change" button on block 8 has been added, which allows the user to change its position. A column was added to the status table containing a "change" button for each module, as can be seen in the upper block of Figure 79. Below in this figure, the *changePos()* function is represented, which makes a request to the server with the identification of the respective module.

```
<table class="t1">
  <tr>  <th> VALVE </th>  <th>STATE</th>  </tr>
  <tr>  <th> VALVE T1 </th> <th id="valvet1">%VALVET1%</th>      <th> <input type="button"id="VALVET1" onclick="changePos(this.id)" value="Change"> </th> </tr>
  <tr>  <th> VALVE B1 </th> <th id="valveb1">%VALVEB1%</th>      <th> <input type="button"id="VALVEB1" onclick="changePos(this.id)" value="Change"> </tr>
  <tr>  <th> VALVE C1 </th> <th id="valvec1">%VALVEC1%</th>      <th> <input type="button"id="VALVEC1" onclick="changePos(this.id)" value="Change"> </tr>
  <tr>  <th> VALVE SN1 </th> <th id="valvesn1">%VALVESN1%</th>   <th> <input type="button"id="VALVESN1" onclick="changePos(this.id)" value="Change"> </tr>
  <tr>  <th> VALVE P </th> <th id="valvep">%VALVEP%</th>         <th> <input type="button"id="VALVEP" onclick="changePos(this.id)" value="Change"> </tr>
  <tr>  <th> VALVE T2 </th> <th id="valvet2">%VALVET2%</th>      <th> <input type="button"id="VALVET2" onclick="changePos(this.id)" value="Change"> </tr>
  <tr>  <th> VALVE B2 </th> <th id="valveb2">%VALVEB2%</th>      <th> <input type="button"id="VALVEB2" onclick="changePos(this.id)" value="Change"> </tr>
  <tr>  <th> VALVE C2 </th> <th id="valvec2">%VALVEC2%</th>      <th> <input type="button"id="VALVEC2" onclick="changePos(this.id)" value="Change"> </tr>
  <tr>  <th> VALVE SN2 </th> <th id="valvesn2">%VALVESN2%</th>   <th> <input type="button"id="VALVESN2" onclick="changePos(this.id)" value="Change"> </tr>
</table>
```

```
function changePos(id){
  var xhttp = new XMLHttpRequest();
  xhttp.open("GET", "/changePos?id="+id, true);
  xhttp.send();
}
```

Figure 79: Change button html code.

In ESP32, the placed order is fulfilled by executing the code in Figure 80. The ID associated with the order is compared with the valves array ("valves []"), and when the valve is identified, the command *"*(int)id"* is sent to the PLC, that will alternate its position.

```
server.on("/changePos", HTTP_GET, [](AsyncWebServerRequest *request){ // Handles the change of a single valve
  String id;
  int i = 0;

  if(request->hasParam("id")){                                        // If it was requested to change a valve position
    id = request->getParam("id")->value();                            // Get valve ID
    for(i = 0;i<sizeof(urit)/ sizeof( const char * );i++){            // Iterate through all valves
      if(id==valves[i]){                                              // If it is the correspondent valve
        Serial2.print("*"+String(i));                                 // Send command to PLC to change the valve
        break;
      }
    }
  }
  request->send_P(200, "text/plain", status);                         // Send request to update status
});
```

Figure 80: Handle change position request.

In addition to the interface features mentioned above, methods to inform the user in the event of a system failure, namely power failures and non-operational valves, were also implemented, which allow the user to choose the procedure after the fault is resolved.

Starting with the malfunction of a valve, when, in the middle of a racking, a valve does not respond, the system suspends the operation. In this situation, the interaction with the user on the interface is shown in Figure 81, which consists of three phases, the user notification, the representation of non-functional valves and the user options after maintenance, which will be described below.

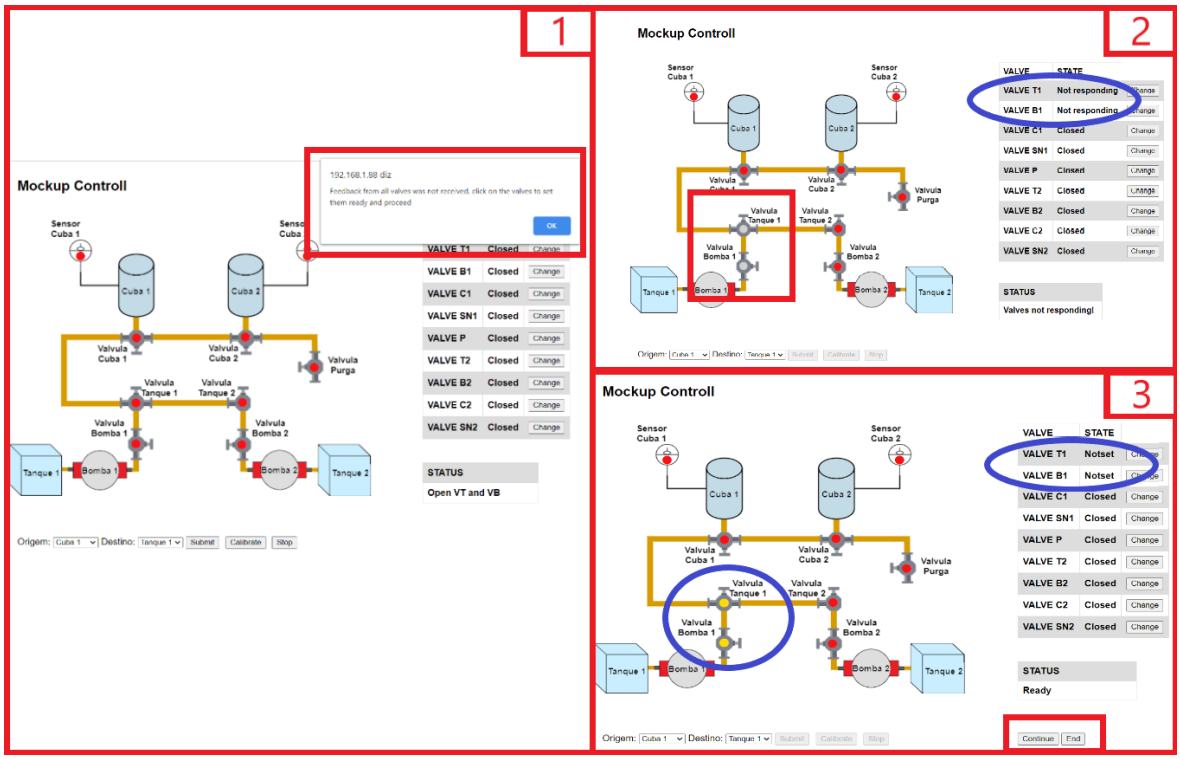


Figure 81: Interface on valve malfunction.

Phase 1

In this phase, it is highlighted the notification that one or more valves have failed, with the message “Feedback from all valves was not received, click on the valves to set them ready and proceed”. This happens when a valve state is updated to “not responding” on the PLC, which is then sent to ESP32. At the address “/block”, represented in the upper block of Figure 82, the status of all valves are analysed, and if any of them does not respond, the message “block” is sent to the server. If there are no failing valves, the message sent is “/ unblock”, and the operation is updated to the respective state. Upon receiving the message indicating a failure, the server blocks the user's options and displays the alert message mentioned above, using the code of the second block in Figure 82.

```
server.on("/block", HTTP_GET, [](AsyncWebServerRequest *request){                // Handle operation interruptions
  int i = 0;
  const char* msg="";
  String block="Not responding";
  for(i = 0;i<sizeof(states)/ sizeof( const char * );i++){                       // Iterate through valves states
    if(block==states[i]) {                                                      // If a valve is not responding
      msg="block";                                                              // The msg will indicate interruption
      status = "Valves not responding!";                                        // Change the status to not responding
      break;
    }else if(strcmp("Notset",states[i])==0){                                    // If the status is not set
      status = "Ready";                                                         // Changes status to ready
      msg="unblock";                                                            // Msg will indicate to continue
    }
  }
  request->send_P(200, "text/plain", msg);                                      // Send request with Msg
});
```

```
setInterval(function ( ) {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      if(this.responseText == "block" && setBlock==0){
        setBlock=1;
        document.getElementById("submbutton").disabled=true;
        document.getElementById("calibration").disabled=true;
        document.getElementById("stop").disabled=true;
        alert("Feedback from all valves was not received, click on the valves to set them ready and proceed");
      }else if(this.responseText == "unblock" && setBlock==1){
        document.getElementById("blockbreak").style.display='';
        document.getElementById("blockcontinue").style.display='';
      }
    }
  };
  xhttp.open("GET", "/block", true);
  xhttp.send();
}, 1000 ) ;
```

Figure 82: Alert notification on valve malfunction code.

Phase 2

After the user selects the “ok” option of the alert, the interface page will represent the faulty valves in grey, highlighted in red, and update their status in the table, highlighted in blue. This is where the user should fix the non-functioning valves, and then press the grey ball of the respective valve. In Figure 83, block 1, the html code used to represent each of the valve icons is present, which when pressed will execute the *changeState()* function, represented in block 2, which makes a request to the server at the address “/changestate” with the ID of the selected valve. The order is then fulfilled using the code in block 3 of Figure 83, where the repaired valve is identified and the PLC is signalled through the *&(int)id* command, which will put the valve in the “Not set” state.

```
1
<img src="https://i.imgur.com/L9e8dM8.png" width="200" height="240">
<div onclick="changeState(this.id)" class="VT1" id="ctoggle_t1"> %CTOGGLE_T1% </div>
<div onclick="changeState(this.id)" class="VT2" id="ctoggle_t2"> %CTOGGLE_T2% </div>
<div onclick="changeState(this.id)" class="VB1" id="ctoggle_b1"> %CTOGGLE_B1% </div>
<div onclick="changeState(this.id)" class="VB2" id="ctoggle_b2"> %CTOGGLE_B2% </div>
<div onclick="changeState(this.id)" class="VC1"id="ctoggle_c1"> %CTOGGLE_C1% </div>
<div onclick="changeState(this.id)" class="VC2"id="ctoggle_c2"> %CTOGGLE_C2% </div>
<div onclick="changeState(this.id)" class="SN1" id="ctoggle_sn1"> %CTOGGLE_SN1% </div>
<div onclick="changeState(this.id)" class="SN2" id="ctoggle_sn2"> %CTOGGLE_SN2% </div>
<div onclick="changeState(this.id)" class="VP" id="ctoggle_P"> %CTOGGLE_P% </div>
</img>

2
function changeState(id){
  if(setBlock==1){
    var xhttp = new XMLHttpRequest();
    xhttp.open("GET", "/changestate?id="+id, true);
    xhttp.send();
  }
}

3
server.on("/changestate", HTTP_GET, [](AsyncWebServerRequest *request){ // Change the state of non responding valve
  String id;
  int i = 0;
  if(request->hasParam("id")){                                          // If a valve state was changed
    id = "/"+request->getParam("id")->value();                          // Get vavle ID
    for(i = 0;i<sizeof(urit)/ sizeof( const char * );i++){              // Iterate through valves ids
      if(id==urit[i] && strcmp(states[i],"Not responding")==0){         // When it finds the correspondent id and it is not responding
        Serial2.print("&"+String(i));                                   // Send command to PLC to set valve state
        break;
      }
    }
  }
  request->send_P(200, "text/plain", status);                           // Update status
});
```

Figure 83:Code for valve state indetifications icons.

Phase 3

When all the faulty valves are in the “Not set” state, in blue, the “Continue” and “End” buttons, highlighted in red, made with the code of block 1 in Figure 84, are set available in the interface. Pressing the button “End”, the *blockBreak()* function (Figure 84 block 2_a) is executed, which enables the interaction buttons again, and in ESP32 the command “+break” is sent to the PLC (Figure 84 block 2_b) that will make the system return to initial state. The “Continue” button runs the *blockCont()* function (Figure 84 block 3_a), and, in ESP32, the command “+continue” (Figure 84 block 3_b) is sent, indicating the PLC that should continue the suspended operation.

```
1
<div class="blockButton" >
  <button id="blockcontinue" onclick="blockCont()" style='display:none'> Continue </button>
  <button id="blockbreak" onclick="blockBreak()" style='display:none'> End </button>
</div>

2_a
function blockBreak(){
  var xhttp = new XMLHttpRequest();
  document.getElementById("submbutton").disabled=false;
  document.getElementById("calibration").disabled=false;
  document.getElementById("blockbreak").style.display='none';
  document.getElementById("blockcontinue").style.display='none';
  document.getElementById("stop").disabled=true;
  setBlock=0;
  xhttp.open("GET", "/blockBreak", true);
  xhttp.send();
}

2_b
server.on("/blockBreak", HTTP_GET, [](AsyncWebServerRequest *request){
  Serial2.print("+break");
  status = AllStatus[16];
  request->send(200);
});

3_a
function blockCont(){
  var xhttp = new XMLHttpRequest();
  document.getElementById("submbutton").disabled=false;
  document.getElementById("calibration").disabled=false;
  document.getElementById("blockbreak").style.display='none';
  document.getElementById("blockcontinue").style.display='none';
  document.getElementById("stop").disabled=false;
  setBlock=0;
  xhttp.open("GET", "/blockCont", true);
  xhttp.send();
}

3_b
server.on("/blockCont", HTTP_GET, [](AsyncWebServerRequest *request){
  Serial2.print("+continue");
  request->send(200);
});
```

Figure 84:Code for buttons when system operation breaks.

That said, remains the unexpected system shutdown situation, which is the case in the event of a power failure. When this happens, the PLC notifies ESP32 and the message “Operation interrupted. Do you want to continue?”, pops up, with the selection options “ok” and “Cancel”, as seen in the red block of Figure 85,

limiting the user to choose whether to proceed from the point where the system stopped, or to reposition the system in the initial state, respectively.

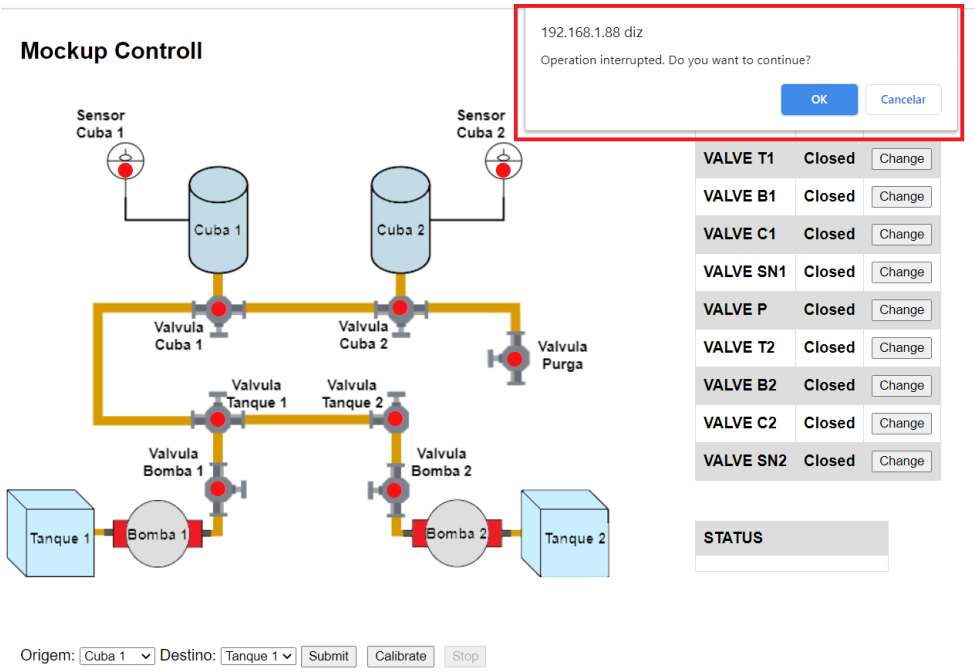
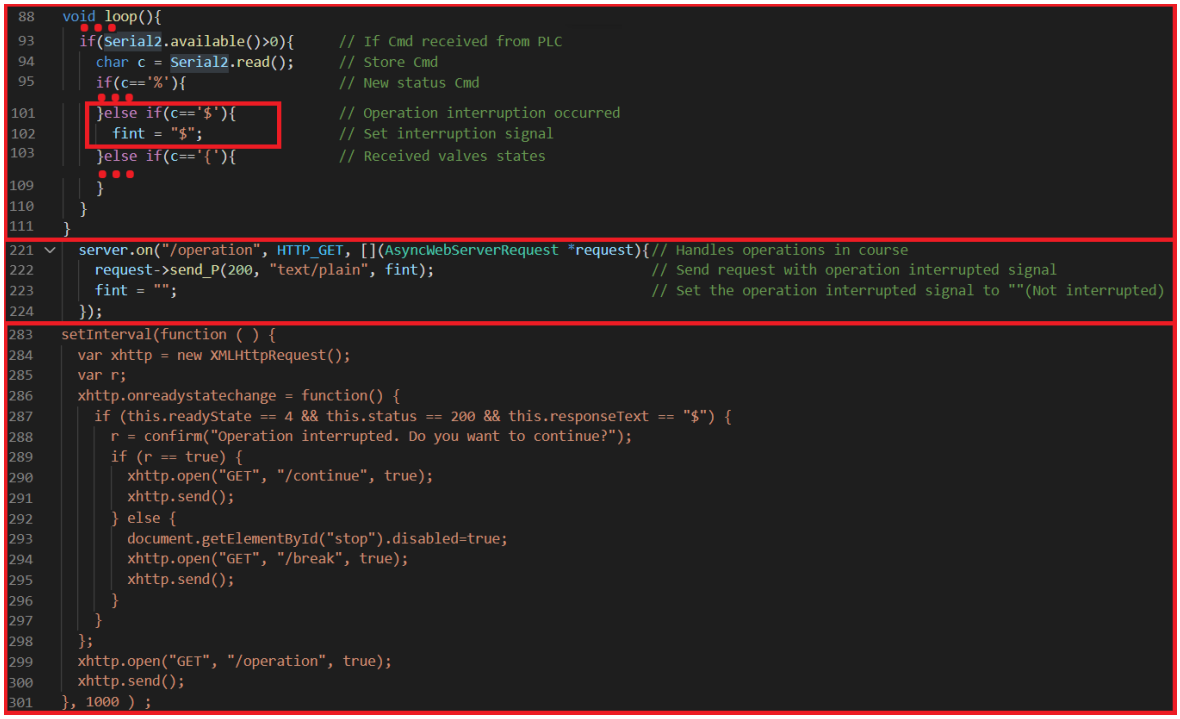


Figure 85: Interface when system stops unexpectedly.

Figure 86 shows the implementation of this latter functionality on the server. As the PLC indicates that the system stopped unexpectedly, by transmitting the character “$” via serial communication to the ESP32, *loop()* function was added to the code indicated in the first block of the figure. When this character is read, the variable “fint” is updated with it, which will later be sent to the server as shown in the second block of Figure 86. In the third block of Figure 86, when the “fint” contains the character “$”, the message in Figure 85 appears, and, in the case of “ok” option being selected, a request will be sent to the address “/continue” continuing with the operation from the stop point. Otherwise, the request will be sent to the address "/break" which returns the system to its initial state. Addresses “/continue” and “/break” are the same used in the buttons “Continue” and “End” mentioned earlier.



```
void loop(){
  ...
  if(Serial2.available()>0){      // If Cmd received from PLC
    char c = Serial2.read();      // Store Cmd
    if(c=='%'){                   // New status Cmd
    ...
    }else if(c=='$'){             // Operation interruption occurred
      fint = "$";                 // Set interruption signal
    }else if(c=='{'){             // Received valves states
    ...
    }
  }
}
server.on("/operation", HTTP_GET, [](AsyncWebServerRequest *request){// Handles operations in course
  request->send_P(200, "text/plain", fint);                          // Send request with operation interrupted signal
  fint = "";                                                         // Set the operation interrupted signal to ""(Not interrupted)
});
setInterval(function ( ) {
  var xhttp = new XMLHttpRequest();
  var r;
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200 && this.responseText == "$") {
      r = confirm("Operation interrupted. Do you want to continue?");
      if (r == true) {
        xhttp.open("GET", "/continue", true);
        xhttp.send();
      } else {
        document.getElementById("stop").disabled=true;
        xhttp.open("GET", "/break", true);
        xhttp.send();
      }
    }
  };
  xhttp.open("GET", "/operation", true);
  xhttp.send();
}, 1000 ) ;
```

Figure 86: Code for system reset in middle of operation.

5. Experiments and results

5.1 *Individual module test*

After all modules built and before they were assembled on the platform, tests were carried out on how each module was running, in order to identify possible problems in their operation. These tests were performed at the breadboard assembly shown in Section 3.4.1, where all modules were, one by one, connected to the PLC, as in Figure 25, and programmed with the algorithm shown in Figure 39 of Section 4.2.

The procedure consisted in connecting the limit sensors to the appropriate digital inputs and the motor to the digital output, and, after PLC initialization, sending the command "1" via serial to the PLC, which would start the motor rotation in one direction. As soon as one of the limit sensors was reached, the motor should stop, confirming half of the process, and then the motor wires are changed. Repeating the process would provide the engine a chance to run in the opposite direction, to confirm the rest of the operation.

The test's obtained results are shown in Figure 87. Here, only one module is observed in the final two positions, but they all passed the test. One of the modules did not work well the first time due to contact issues at the motor terminals which were solved allowing the module to work correctly.

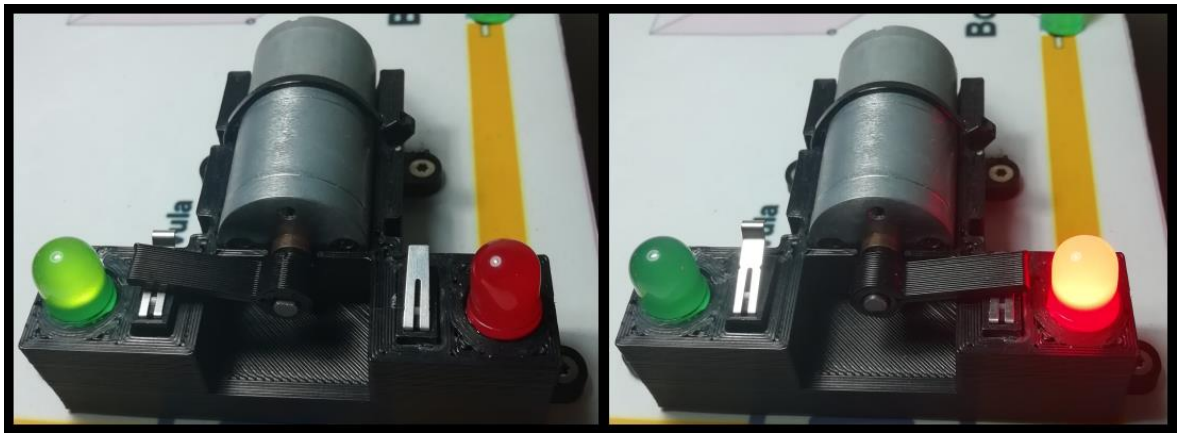


Figure 87: Individual module test result (opened valve and closed valve).

5.2 *Basic operation*

After the system was completely assembled, it was necessary to verify that all the operations carried out were completed successfully, as in the ideal case in which the system is not subject to failures. To this end, a test routine consisting of transfers between containers was stipulated below:

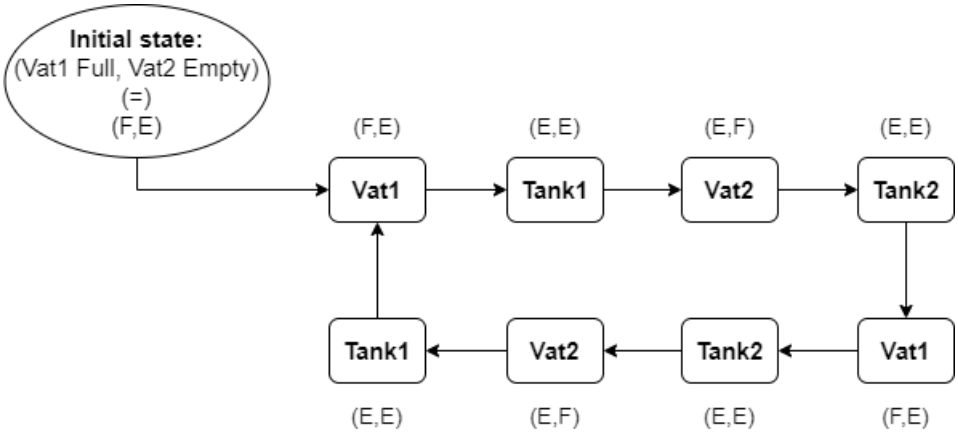


Figure 88:Test routine.

Figure 88 shows the various containers connected by arrows, which corresponds to an arrow-direction transfer. Next to the containers, the state of the two vats is observed after the transfer of the container that precedes it.

This routine was applied ten times, and in all executions, the status of each module was checked at each stage of each transfer, whether the transfer ended without errors or not, and if the transfer did not end. These data were recorded in the table in appendix 2. The system's response was confirmed as follows: a case in which it is supposed to fill a vat when it is already full, or, on the contrary, to empty the vat when it is already empty.

All operations went smoothly with no anomaly, which resulted in a 100% success rate, indicating that, in an ideal scenario, the system should work without any problems. The system operating normally can be seen at https://drive.google.com/file/d/1KqGsjJvpAy0B_YWfah6TMo4BVub5G41C/view?usp=sharing.

5.3 *Power failure scenario*

The experiments performed here served to test the system's response to a power cut but can also be related to the voluntary shutdown of the system in the middle of a transfer. When this happens, the system, after restarting, should ask the user if he wants to continue at the point where he left off or if he intends to return to the initial stage. Tests were then carried out that made it possible to analyse these two options. Transfers from a tank to a vat (unspecified) and vice versa were considered, and at each point of the transfer, the power was cut. After the restart, the system's response to each of the options provided was recorded. The expected response when selecting the option to continue in each transfer status can be seen in Table 11.

Table 11: Expected results after continuing operation on all transfer states

Transfer	1	2	3	4	5	6	7
Vat->Tank	O(VB,VT)	O(VC)	E(SN)	C(VC)	O(VP)	C(VP)	C(VT,VB)
Tank->Vat	O(VT,VC)	F(SN)	C(VC)	O(VP,VB)	C(VP)	C(VT,VB)	x

Legend:
-O(x,…)-> Open valve x,…
-C(x,…)-> Close valve x,…
-E(x)-> Empty vat x
-F(x)-> Fill vat x

Afterwards, the tests were performed ten times for each hypothesis and the results obtained were recorded in the table in Appendix 3. It appears a failure occurred in one case alone. When transferring from a vat to a tank, it stopped at the beginning of the purge process and the purge valve did not open after the system reset. A new restart solved the situation, which indicates that the status was saved correctly; so, it is likely that there was a communication failure between ESP32 and Arduino, when it comes to sending the user's choice. All other tests were successful. A video with the response of the system to a power cut simulation is presented in https://drive.google.com/file/d/1TlCPMNFwCkYr5Ygy02tG04eRMVRm80y2/view?usp=sharing.

5.4 *Valve malfunction*

The last set of tests was conducted to understand the system's response to the malfunction of a valve. If a valve is damaged and, therefore, does not react to the trigger signal or transitions to the wrong position, then, after a specified time (six seconds in this case), the transfer in process must stop, and, at the interface, the user can choose if he wants to proceed or move the system back to the initial stage.

To test this functionality, the transfer state's scheme shown in Table 11 was used. In this case, in each of the states, the tab that presses the limit sensors of the active modules was removed. Thus, the modules in question do not give a feedback to the PLC, and, after the time limit is reached, the system should stop. The results were recorded after the user's choice, in the event that the tab is replaced and the transfer is continued or stopped for all possible combinations of containers. The table in Appendix 4 contains the confirmation of each hypothesis' expected results. An example of this process can be seen in https://drive.google.com/file/d/1c-bj_wO0OH4kbfuQs8mJetkicEuENocb/view?usp=sharing.

The system responded to all tests as expected. It should be noted that in the case of the transition from a representative module of a vat, the tests were not carried out, as the functionality was not implemented because it cannot be transposed to a real case.

6. Conclusions

The automation of liquid transfer systems is a slightly explored facet in the area of automation and industrial control. As Watgrid is a company mostly focused on the wine market, it was able to acknowledge this and realize that it would be an opportunity to expand its range of applications and provide its customers with a product capable of optimizing wine production. Developing such a project becomes even more advantageous for the company, as it brings the possibility of automating its own racking system, allowing a more cost effective products' calibration and maintenance.

The company's structure already contains some level of control over the actuating elements but does not have the necessary power to launch the product into the market. The use of a PLC will provide the system with the ability to mitigate these flaws, resulting in a stable system and one prepared for multi-scaled applications. In addition, the fact that the system is completely structured and ready to be automated makes it ideal for developing and testing such a project.

Even though the company's system is suited for the project, it is a delicate structure regarding the proper interaction with its components. One failure is enough to jeopardise the whole process, thus incurring in high costs. Developing a model with modules that (in the eyes of the programmer) work in the same way as the components of the real system allows a risk reduction when implementing in the real system.

The developed modules do not represent, to the maximum resemblance, the real components, but present an identical operation. That way, with some software changes, the PLC can control them, and thus, all the necessary tests can be performed with no substantial damage. The control algorithm implemented also can control several sets of components associated with tanks and vats, as it was designed to be scalable.

The developed model behaved well when subjected to functional tests. In an ideal scenario, the PLC should be able to control the real system without failure. Regarding situations in which failures occur, the mitigation strategies implemented in the software provided risks mitigation measures for damage prevention. This mitigation measures were tested with satisfactory results. They evidenced only one issue in the power cut scenario, which was solved with a system restart. This failure may have been a communication error between the Arduino and the PLC, which indicates that there is a lack of robustness in this connection. That said, it is concluded that, with the necessary adjustments, it should be possible to adapt the developed work to the real system.

Throughout the development of this project with Watgrid, several work areas were explored, starting from automation with the PLC and the various mechanical components, all the way to the design of support parts (also through simple back and front ends software programming in ESP32). Not to mention the thorough work of assembling the electrical circuits of the PCB.

6.1 *Future work*

In addition to the main goal, future work to implement the project in a real system may include:

- Connecting the PLC to the company network via Ethernet cable, which provides more robustness than wireless, and using the company interface allocated on its server, with access via a Websocket.
- Develop software to handle possible failure scenarios other than the ones approached in this project.

References

[1] Watgrid, “Winegrid,” [Online]. Available: https://www.winegrid.com/wp-content/uploads/2020/07/solution_graphic_whitebg2.jpg.

[2] Unitronics, “What is the definition of "PLC"?,” [Online]. Available: https://www.unitronicsplc.com/what-is-plc-programmable-logic-controller/.

[3] D. Eller, “Velki - PLCs: O que são e quais as suas vantagens?,” [Online]. Available: https://velki.com.br/pt/blog/aprenda-com-a-velki/plcs--o-que-sao-e-quais-as-suas-vantagens.

[4] Ladder Logic World, “PLC Manufacturers: The Latest PLC Brands, Rankings & Revenues,” [Online]. Available: https://ladderlogicworld.com/plc-manufacturers/.

[5] Industrial Shields, “Industrial Shields,” [Online]. Available: https://www.industrialshields.com/.

[6] Controllino, “controllino,” [Online]. Available: https://www.controllino.com/.

[7] TME, “SIEMENS SIMATIC S7-1200,” Siemens, [Online]. Available: https://www.tme.eu/pt/details/6es7214-1ag40-0xb0/controladores-plc/siemens/.

[8] TME, “SIEMENS SIMATIC S7-1200,” Siemens, [Online]. Available: https://www.tme.eu/pt/details/6es7214-1ag40-0xb0/controladores-plc/siemens/.

[9] Controllino, “CONTROLLINO MEGA,” [Online]. Available: https://www.controllino.com/product/controllino-mega/.

[10] Controllino, “CONTROLLINO MAXI Automation,” [Online]. Available: https://www.controllino.com/product/controllino-maxi-automation/.

[11] Industrial Shields, “ M-DUINO PLC Arduino 57AAR I/Os Analog / Digital / Relay PLUS,” [Online]. Available: https://www.industrialshields.com/shop/product/is-mduino-57aar-m-duino-plc-arduino-57aar-i-os-analog-digital-relay-plus-436?category=1.

[12] Industrial Shields, “M-DUINO PLC Arduino Ethernet 58 I/Os Analog/Digital PLUS,” [Online]. Available: https://www.industrialshields.com/shop/product/is-mduino-58-m-duino-plc-arduino-ethernet-58-i-os-analog-digital-plus-176?category=1.

[13] Industrial Shields, “MDUINO PLC ARDUINO ETHERNET & WiFi & BLUETOOTH LE 53ARR I/Os ANALOG/DIGITAL/RELAY PLUS,” [Online]. Available:

https://www.industrialshields.com/shop/product/007001001000-mduino-plc-arduino-ethernet-wifi-bluetooth-le-53arr-i-os-analog-digital-relay-plus-1243?search=53+ARR.

[14] Unipi, “Unipi Neuron L203,” [Online]. Available: https://www.unipi.technology/unipi-neuron-l203-p100.

[15] RS, “Schneider Electric Modicon M221 PLC CPU, Mini USB Interface,” Schneider Electric, [Online]. Available: https://uk.rs-online.com/web/p/plcs-programmable-logic-controllers/8066815/.

[16] TME, “OMRON CP1E-N40S1DR-A,” OMRON, [Online]. Available: https://www.tme.eu/pt/details/cp1e-n40s1dr-a/controladores-plc/omron/.

[17] TME, “OMRON CP1E-N14DR-A,” OMRON, [Online]. Available: https://www.tme.eu/pt/details/cp1e-n14dr-a/controladores-plc/omron/.

[18] RS, “Allen Bradley Micro850 PLC CPU - 28 Inputs, 20 Outputs, Ethernet, USB Networking,” Allen Bradley, [Online]. Available: https://uk.rs-online.com/web/p/plcs-programmable-logic-controllers/7867213/.

[19] Automation Direct, “C0-00DR-D,” [Online]. Available: https://www.automationdirect.com/adc/shopping/catalog/programmable_controllers/click_series_plcs_(stackable_micro_brick)/plc_units/c0-00dr-d.

[20] Automation Direct, “C0-11ARE-D,” [Online]. Available: https://www.automationdirect.com/adc/shopping/catalog/programmable_controllers/click_series_plcs_(stackable_micro_brick)/plc_units/c0-11are-d.

[21] Unipi, “Unipi Extension xS11,” [Online]. Available: https://www.unipi.technology/unipi-extension-xs11-p336?categoryId=40.

[22] TME, “SIEMENS 6ES7223-1BL32-0XB0,” Siemens, [Online]. Available: https://www.tme.eu/pt/details/6es7223-1bl32-0xb0/controladores-plc/siemens/.

[23] TME, “OMRON CP1W-40EDR,” OMRON, [Online]. Available: https://www.tme.eu/pt/details/cp1w-40edr/controladores-plc/omron/.

[24] RS, “Módulo de E/S PLC Schneider Electric, Serie TM3, 24 x Entrada/Salida,” Schneider Eletric, [Online]. Available: https://pt.rs-online.com/web/p/accesorios-para-automatas-programables/8066988?cm_mmc=PT-PPC-DS3A-_-google-_-3_PT_PT_M_A_and_C_Exact-_-Schneider_Electric%7CM%C3%B3dulos_E%2FS_para_Aut%C3%B3matas_Programables-_-TM3DM24R&gclid=CjwKCAiA-vLyBRBWEiwAzOkGVND0MU_.

[25] RS, “Allen Bradley PLC I/O Module for use with Micro850 Series,” Allen Bradley, [Online]. Available: https://uk.rs-online.com/web/p/plc-accessories/7867229/.

[26] RS, “Allen Bradley PLC I/O Module for use with Micro850 Series,” Allen Bradley, [Online]. Available: https://uk.rs-online.com/web/p/plc-accessories/7867248/.

[27] Automation Direct, “C0-16CDD1,” [Online]. Available: https://www.automationdirect.com/adc/shopping/catalog/programmable_controllers/click_series_plcs_(stackable_micro_brick)/dc_i-z-o/c0-16cdd1.

[28] Asco Joucomatic, “TECNOLOGIA ELECTROVÁLVULAS E VÁLVULAS - Funcionamento, terminologia,” [Online]. Available: http://www.asconumatics.eu/images/site/upload/_pt/pdf1/00005pt.pdf.

[29] More Wine, “Racking & Transferring Wine,” [Online]. Available: https://morewinemaking.com/category/racking-transferring-wine.html.

[30] Michael Smith Engineers, “Useful information on Self-Priming Pumps,” [Online]. Available: https://www.michael-smith-engineers.co.uk/resources/useful-info/self-priming-pumps#.

[31] Controllino, “Instruction Manual,” [Online]. Available: https://www.controllino.com/wp-content/uploads/2019/02/CONTROLLINO-Instruction-Manual-V1.5-2018-12-14.pdf.

[32] Unipi Technology, “Mervis,” [Online]. Available: https://kb.unipi.technology/en:sw:01-mervis.

[33] Unipi Technology, “Node-RED,” [Online]. Available: https://kb.unipi.technology/en:sw:03-3rd-party:nodered.

[34] Unipi Technology, “Choosing the right software,” [Online]. Available: https://kb.unipi.technology/en:sw:00-start.

[35] Industrial Shields, “MDUINO 53ARR PLUS - datasheet,” [Online]. Available: https://www.industrialshields.com/web/content?model=ir.attachment&field=datas&id=94002&.

[36] Rover Pompe, “Pumps and Electropumps Self-Primers - Dual Rotation,” [Online]. Available: https://www.hidraulicart.pt/wp-content/uploads/manual-rover-pt.pdf.

[37] TONHE, “A100-T Series Electric Stainless steel304 Shut off Valve,” [Online]. Available: http://www.motorized-valve.com/Upload/Image/pdf/Tonhe-A100-Motorized-valve.pdf.

[38] TME, “FIT0495-A DFROBOT,” [Online]. Available: https://www.tme.eu/pt/details/df-fit0495-a/motores-de-cc/dfrobot/fit0495-a/.

[39] TME, “82869014 CROUZET,” [Online]. Available: https://www.tme.eu/pt/details/crouzet82869014/motores-eletricos/crouzet/82869014/.

[40] TME, “499:1 METAL GEARMOTOR 25DX58L MM LP 6V POLOLU,” [Online]. Available: https://www.tme.eu/pt/details/pololu-1591/motores-de-cc/pololu/499-1-metal-gearmotor-25dx58l-mm-lp-6v/.

[41] TME, “ROB-12472 SPARKFUN ELECTRONICS INC,” [Online]. Available: https://www.tme.eu/pt/details/sf-rob-12472/motores-de-cc/sparkfun-electronics-inc/rob-12472/.

[42] RS, “Motorreductor DC con escobillas RS PRO, 1500 gcm, 7 rpm, 3 V dc, 1,71 W,” [Online]. Available: https://pt.rs-online.com/web/p/motorreductores-dc/0420681/.

[43] RS, “Motorreductor DC con escobillas RS PRO, 15 Ncm, 6 rpm, 12 V dc, 1.31 W,” [Online]. Available: https://pt.rs-online.com/web/p/motorreductores-dc/0420647/.

[44] PTRobotics, “JGY370 Reversible High Torque Worm Geared 12V 6rpm,” [Online]. Available: https://www.ptrobotics.com/dc-com-engrenagens/8054-jgy370-reversible-high-torque-worm-geared-12v-6rpm.html.

[45] PTRobotics, “1000:1 Micro Metal Gearmotor,” [Online]. Available: https://www.ptrobotics.com/dc-com-engrenagens/2381-1000-1-micro-metal-gearmotor.html.

[46] Robot Shop, “Cytron 12V 17RPM 194.4oz-in Spur Gearmotor,” [Online]. Available: https://www.robotshop.com/eu/en/cytron-12v-17rpm-1944oz-in-spur-gearmotor.html.

[47] TME, “LL-1003GD2D-1A LUCKYLIGHT,” [Online]. Available: https://www.tme.eu/pt/details/ll-1003gd2d-1a/leds-tht-10mm/luckylight/.

[48] TME, “LL-1003VD2D-V1-1A LUCKYLIGHT,” [Online]. Available: https://www.tme.eu/pt/details/ll-1003vd2d-v1-1a/leds-tht-10mm/luckylight/.

[49] TME, “MSW-22 NINIGI,” [Online]. Available: https://www.tme.eu/pt/details/msm-22/microcomutadores-snap-action/ninigi/msw-22/.

[50] Diodes Incorporated, “ZXMHC10A07N8,” [Online]. Available: https://pt.mouser.com/datasheet/2/115/ZXMHC10A07N8-78491.pdf.

[51] TME, “ZXMHC10A07N8TC DIODES INCORPORATED,” [Online]. Available: https://www.tme.eu/pt/details/zxmhc10a07n8tc/transistores-multi-canal/diodes-incorporated/.

[52] TME, “AR03BTCX1001 VIKING,” [Online]. Available: https://www.tme.eu/pt/details/ar0603-1k-0.1%25/resistencias-de-precisao-smd-0603/viking/ar03btcx1001/.

[53] Grabcad, “motor robot fit0495,” [Online]. Available: https://grabcad.com/library/motor-robot-fit0495-b-20rpm-6v-itytarg-1.

[54] Grabcad, “micro limit switch,” [Online]. Available: https://grabcad.com/library/micro-limit-switch-1a-1.

[55] Gabcad, “All leds,” [Online]. Available: https://grabcad.com/library/all-leds-1.

[56] Euro Circuits, “Euro Circuits,” [Online]. Available: https://www.eurocircuits.com/.

[57] Duncan Hine, “TS-35 Din Rail Top Hat, Slotted, 35mm x 7.5mm,” 20 5 2019. [Online]. Available: https://grabcad.com/library/ts-35-din-rail-top-hat-slotted-35mm-x-7-5mm-1.

[58] Arduino, “setup(),” [Online]. Available: https://www.arduino.cc/reference/en/language/structure/sketch/setup/.

[59] Arduino, “loop(),” [Online]. Available: https://www.arduino.cc/reference/en/language/structure/sketch/loop/.

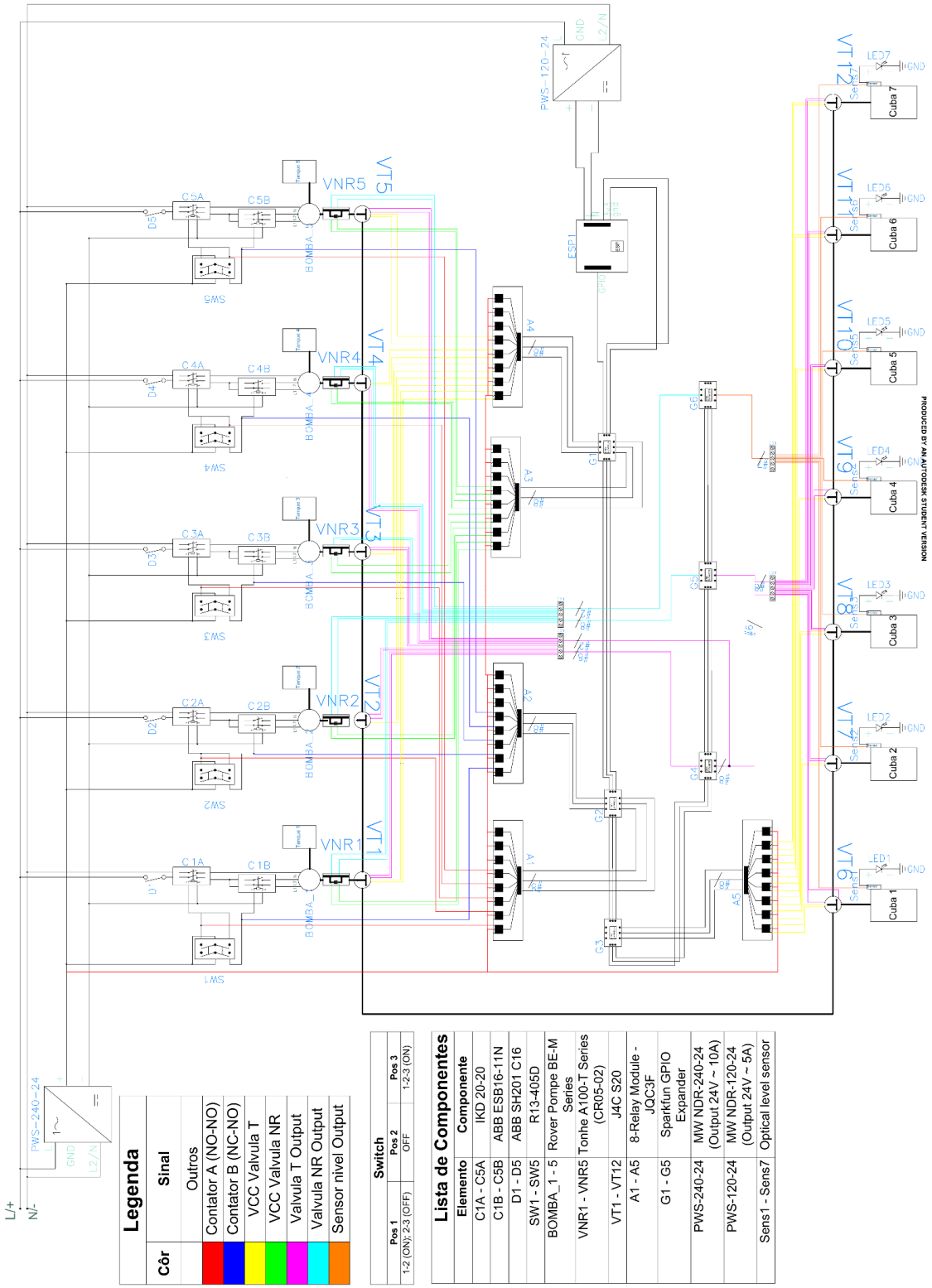
[60] Arduino, “EEPROM library,” [Online]. Available: https://www.arduino.cc/en/Reference/EEPROM.

[61] Arduino, “EEPROM.read(),” [Online]. Available: https://www.arduino.cc/en/Tutorial/LibraryExamples/EEPROMRead.

[62] Arduino, “EEPROM.write(),” [Online]. Available: https://www.arduino.cc/en/Tutorial/LibraryExamples/EEPROMWrite.

Appendixes

Appendix 1: Company's System Scheme



Legenda

Cór	Sinal
	Outros
	Contator A (NO-NO)
	Contator B (NC-NO)
	VCC Valvula T
	VCC Valvula NR
	Valvula T Output
	Valvula NR Output
	Sensor nivel Output

Switch

Pos 1	Pos 2	Pos 3
1-2 (ON); 2-3 (OFF)	OFF	1-2-3 (ON)

Lista de Componentes

Elemento	Componente
C1A - C5A	IKD 20-20
C1B - C5B	ABB ESB16-11N
D1 - D5	ABB SH201 C16
SW1 - SW5	R13-405D
BOMBA_1 - 5	Rover Pompe BE-M Series
VNR1 - VNR5	Tonhe A100-T Series (CR05-02)
VT1 - VT12	J4C S20
A1 - A5	8-Relay Module - JQC3F
G1 - G5	Sparkfun GPIO Expander
PWS-240-24	MW NDR-240-24 (Output 24V ~ 10A)
PWS-120-24	MW NDR-120-24 (Output 24V ~ 5A)
Sens1 - Sens7	Optical level sensor

Appendix 2:

Table 12: Basic operation test results

Initial state	Origin	Destin	Expected operation	Espected state	# tests	Not finished	Finished with errors	Successfully finished	Errors	Success rate	Error rate
(V1 F, V2 E) Test rotine	V1	T1	O(VB1,VT1)>O(VC1)>E(SN1)>C(VC1)>O(VP)>C(VP)>C(VT1,VB1)	C1->E	10	0	0	10		100%	0%
	T1	V2	O(VT1,VC2)>F(SN2)>C(VC2)>O(VP,VB1)>C(VP)>C(VT1,VB1)	C2->F	10	0	0	10		100%	0%
	V2	T2	O(VB2,VT2)>O(VC2)>E(SN2)>C(VC2)>O(VP)>C(VT2,VB2)	C2->E	10	0	0	10		100%	0%
	T2	V1	O(VT2,VC1)>F(SN1)>C(VC1)>O(VP,VB2)>C(VP)>C(VT2,VB2)	C1->F	10	0	0	10		100%	0%
	V1	T2	O(VB2,VT2)>O(VC1)>E(SN1)>C(VC1)>O(VP)>C(VP)>C(VT2,VB2)	C1->E	10	0	0	10		100%	0%
	T2	V2	O(VT2,VC2)>F(SN2)>C(VC2)>O(VP,VB2)>C(VP)>C(VT2,VB2)	C2->F	10	0	0	10		100%	0%
	V2	T1	O(VB1,VT1)>O(VC2)>E(SN2)>C(VC2)>O(VP)>C(VP)>C(VT1,VB1)	C2->E	10	0	0	10		100%	0%
	T1	V1	O(VT1,VC1)>F(SN1)>C(VC1)>O(VP,VB1)>C(VP)>C(VT1,VB1)	C1->F	10	0	0	10		100%	0%
(V1 E, V2 E)	V1	T1	No Execution	C1->E	10	0	0	10		100%	0%
	V1	T2	No Execution	C1->E	10	0	0	10		100%	0%
	V2	T1	No Execution	C2->E	10	0	0	10		100%	0%
	V2	T2	No Execution	C2->E	10	0	0	10		100%	0%
(V1 F, V2 F)	T1	V1	No Execution	C1->F	10	0	0	10		100%	0%
	T2	V1	No Execution	C1->F	10	0	0	10		100%	0%
	T1	V2	No Execution	C2->F	10	0	0	10		100%	0%
	T2	V2	No Execution	C2->F	10	0	0	10		100%	0%

Legend:

-O(x,…)-> Open

-C(x,…)-> Close

-E(x)-> Empty va

-F(x)-> Fill vat x

Appendix 3:

Table 13: Power failure obtained results

Operation	State	Option	Expected	# tests	Failed	Finished with errors	Successfully finished	Errors	Success rate	Error rate
	1		O(VB,VT)	10	0	0	10	0	100%	0%
	2		O(VC)	10	0	0	10	0	100%	0%
	3		E(SN)	10	0	0	10	0	100%	0%
	4		C(VC)	10	0	0	10	0	100%	0%
	5		O(VP)>d	9	0	1	9	VP did not open - reset worked	100%	11%
	6		C(VP)	10	0	0	10	0	100%	0%
	7	Continue	C(VT,VB)	10	0	0	10	0	100%	0%
	1		C(VB,VT)	10	0	0	10	0	100%	0%
	2		C(VT,VB,VC)	10	0	0	10	0	100%	0%
	3		C(VT,VB,VC)	10	0	0	10	0	100%	0%
	4		C(VT,VB,VC)	10	0	0	10	0	100%	0%
	5		C(VT,VB,VP)	10	0	0	10	0	100%	0%
	6		C(VT,VB,VP)	10	0	0	10	0	100%	0%
V->T	7	End	C(VT,VB)	10	0	0	10	0	100%	0%
	1		O(VT,VC)	10	0	0	10	0	100%	0%
	2		F(SN)	10	0	0	10	0	100%	0%
	3		C(VC)	10	0	0	10	0	100%	0%
	4		O(VP,VB)	10	0	0	10	0	100%	0%
	5		C(VP)	10	0	0	10	0	100%	0%
	6	Continue	C(VT,VB)	10	0	0	10	0	100%	0%
	1		C(VT,VC)	10	0	0	10	0	100%	0%
	2		C(VT,VC)	10	0	0	10	0	100%	0%
	3		C(VT,VC)	10	0	0	10	0	100%	0%
	4		C(VT,VB,VP)	10	0	0	10	0	100%	0%
	5		C(VT,VB,VP)	10	0	0	10	0	100%	0%
T->V	6	End	C(VT,VB)	10	0	0	10	0	100%	0%

Appendix 4:

Table 14: Valve failure obtained test results

Operation	peration stat	not respo	Option	Expected	Obtained
V1->T1	1	VT1	Cont	O2->SO1->c->O1->...	v
			End	O2->SO1->e->C1C2	v
		VB1	Cont	O1->SO2->c->O2->...	v
			End	O1->SO2->e->C1C2	v
	2	VC1	Cont	SO3->c->O3->...	v
			End	SO3->e->C1C2C3	v
	3	SN1	x	x	x
	4	VC1	Cont	SC3->c->C3->...	v
			End	SC3->e->C1C2C3	v
	5	VP	Cont	SO5->c->O5->d->...	v
			End	SO5->e->C1C2C5	v
	6	VP	Cont	SC5->c->C5->...	v
			End	SC5->e->C1C2C5	v
	7	VT1	Cont	C2->SC1->c->C1	v
			End	C2->SC1->e->C1	v
		VB1	Cont	C1->SC2->c->C2	v
			End	C1->SC2->e->C2	v
V1->T2	1	VT2	Cont	O7->SO6->c->O6->...	v
			End	O7->SO6->e->C6C7	v
		VB2	Cont	O6->SO7->c->O7->...	v
			End	O6->SO7->e->C6C7	v
	2	VC1	Cont	SO3->c->O3->...	v
			End	SO3->e->C6C7C3	v
	3	x	x	x	x
	4	VC1	Cont	SC3->c->C3->...	v
			End	SC3->e->C6C7C3	v
	5	VP	Cont	SO5->c->O5->d->...	v
			End	SO5->e->C6C7C5	v
	6	VP	Cont	SC5->c->C5->...	v
			End	SC5->e->C6C7C5	v
	7	VT2	Cont	C7->SC6->c->C6	v
			End	C7->SC6->e->C6	v
		VB2	Cont	C6->SC7->c->C7	v
			End	C6->SC7->e->C7	v
V2->T1	1	VT1	Cont	O2->SO1->c->O1->...	v
			End	O2->SO1->e->C1C2	v
		VB1	Cont	O1->SO2->c->O2->...	v
			End	O1->SO2->e->C1C2	v
	2	VC2	Cont	SO8->c->O8->...	v
			End	SO8->e->C1C2C8	v
	3	x	x	x	x
	4	VC2	Cont	SC8->c->C8->...	v
			End	SC8->e->C1C2C8	v
	5	VP	Cont	SO5->c->O5->d->...	v
			End	SO5->e->C1C2C5	v
	6	VP	Cont	SC5->c->C5->...	v
			End	SC5->e->C1C2C5	v
	7	VT1	Cont	C2->SC1->c->C1	v
			End	C2->SC1->e->C1	v
		VB1	Cont	C1->SC2->c->C2	v
			End	C1->SC2->e->C2	v
V2->T2	1	VT2	Cont	O7->SO6->c->O7->...	v
			End	O7->SO6->e->C6C7	v
		VB2	Cont	O6->SO7->c->O7->...	v
			End	O6->SO7->e->C6C7	v
	2	VC2	Cont	SO8->c->O8->...	v
			End	SO8->e->C6C7C8	v
	3	x	x	x	x
	4	VC2	Cont	SC8->c->C8->...	v
			End	SC8->e->C6C7C8	v
	5	VP	Cont	SO5->c->O5->d->...	v
			End	SO5->e->C6C7C8	v
	6	VP	Cont	SC5->c->C5->...	v
			End	SC5->e->C6C7C5	v
	7	VT2	Cont	C7->SC6->c->C6	v
			End	C7->SC6->e->C6	v
		VB2	Cont	C6->SC7->c->C7	v
			End	C6->SC7->e->C7	v

Operation	peration stat	not respo	Option	Expected	Obtained
T1->V1	1	VT1	Cont	O3->SO1->c->O1->...	v
			End	O3->SO1->e->C1C3	v
		VC1	Cont	O1->SO3->c->O3->...	v
			End	O1->SO3->e->C1C3	v
	2	x	x	x	x
	3	VC1	Cont	SC3->c->C3->...	v
			End	SC3->e->C1C3	v
	4	VB1	Cont	O5->SO2->c->O2->d->...	v
			End	O5->SO2->e->C1C2C5	v
		VP	Cont	O2->SO5->c->O5->d->...	v
			End	O2->SO5->e->C1C2C5	v
	5	VP	Cont	SC5->c->C5->...	v
			End	SC5->e->C1C2C5	v
	7	VT1	Cont	C2->SC1->c->C1	v
			End	C2->SC1->e->C1	v
		VB1	Cont	C1->SC2->c->C2	v
			End	C1->SC2->e->C2	v
T1->V2	1	VT1	Cont	O8->SO1->c->O1->...	v
			End	O8->SO1->e->C1C8	v
		VC2	Cont	O1->SO8->c->O8->...	v
			End	O1->SO8->e->C1C8	v
	2	x	x	x	x
	3	VC2	Cont	SC8->c->C8->...	v
			End	SC8->e->C1C8	v
	4	VB1	Cont	O5->SO2->c->O2->d->...	v
			End	O5->SO2->e->C1C2C5	v
		VP	Cont	O2->SO5->c->O5->d->...	v
			End	O2->SO5->e->C1C2C5	v
	5	VP	Cont	SC5->c->C5->...	v
			End	SC5->e->C1C2C5	v
	7	VT1	Cont	C2->SC1->c->C1	v
			End	C2->SC1->e->C1	v
		VB1	Cont	C1->SC2->c->C2	v
			End	C1->SC2->e->C2	v
T2->V1	1	VT2	Cont	O3->SO6->c->O6->...	v
			End	O3->SO6->e->C6C3	v
		VC1	Cont	O6->SO3->c->O3->...	v
			End	O6->SO3->e->C6C3	v
	2	x	x	x	x
	3	VC1	Cont	SC3->c->C3->...	v
			End	SC3->e->C6C3	v
	4	VB2	Cont	O5->SO7->c->O7->d->...	v
			End	O5->SO7->e->C6C7C5	v
		VP	Cont	O7->SO5->c->O5->d->...	v
			End	O7->SO5->e->C6C7C5	v
	5	VP	Cont	SC5->c->C5->...	v
			End	SC5->e->C6C7C5	v
	7	VT2	Cont	C7->SC6->c->C6	v
			End	C7->SC6->e->C6	v
		VB2	Cont	C6->SC7->c->C7	v
			End	C6->SC7->e->C7	v
T2->V2	1	VT2	Cont	O8->SO6->c->O6->...	v
			End	O8->SO6->e->C6C8	v
		VC2	Cont	O6->SO8->c->O8->...	v
			End	O6->SO8->e->C6C8	v
	2	x	x	x	x
	3	VC2	Cont	SC8->c->C8->...	v
			End	SC8->e->C6C8	v
	4	VB2	Cont	O5->SO7->c->O7->d->...	v
			End	O5->SO7->e->C6C7C5	v
		VP	Cont	O7->SO5->c->O5->d->...	v
			End	O7->SO5->e->C6C7C5	v
	5	VP	Cont	SC5->c->C5->...	v
			End	SC5->e->C6C7C5	v
	6	VT2	Cont	C7->SC6->c->C6	v
			End	C7->SC6->e->C6	v
		VB2	Cont	C6->SC7->c->C7	v
			End	C6->SC7->e->C7	v

Name	ID
VT1	1
VB1	2
VC1	3
SN1	4
VP	5
VT2	6
VB2	7
VC2	8
SN2	9

Legend	Description
Ox	Open Vx
Cx	Close Vx
SOx	Timout opening Vx
SCx	Timout closing Vx
c	Continue pressed
e	End pressed
d	delay

Appendix 5:

Figure 89: Flow chart