TALLINN UNIVERSITY OF TECHNOLOGY

SCHOOL OF ENGINEERING

Department of Electrical Power Engineering and Mechatronics

EMBEDDED SOFTWARE FOR SMART HYDROGEN FUEL CELL GENERATORS

ARUKA VESINIK-KÜTUSEELEMENDI MANUSTARKVARA

MASTER THESIS

Student:	Mohammad Mokhalled
Student code:	195454MAHM
Supervisor:	Professor Mart Tamre
Co-supervisor:	Ivar Kruusenberg, PhD

Tallinn 2021

AUTHOR'S DECLARATION

Hereby I declare, that I have written this thesis independently.

No academic degree has been applied for based on this material. All works, major viewpoints and data of the other authors used in this thesis have been referenced.

Date: *10/06/2021*

Author: Mohammad Mokhalled

/signature /

Thesis is in accordance with terms and requirements

".....” ..... 20….

Supervisor: ….....

/signature/

Accepted for defence

".....”.....20… .

Chairman of theses defence commission: .....

/name and signature/

Non-exclusive Licence for Publication and Reproduction of GraduationTthesis[1]

I, Mohammad Mokhalled (date of birth: 16/09/1995) hereby

1. grant Tallinn University of Technology (TalTech) a non-exclusive license for my thesis EMBEDDED SOFTWARE FOR SMART HYDROGEN FUEL CELL GENERATORS (in Estonian ARUKA VESINIK-KÜTUSEELEMENDI MANUSTARKVARA)

supervised by Professor Mart Tamre and Dr. Ivar Kruusenberg,

1.1 reproduced for the purposes of preservation and electronic publication, incl. to be entered in the digital collection of TalTech library until expiry of the term of copyright;

1.2 published via the web of TalTech, incl. to be entered in the digital collection of TalTech library until expiry of the term of copyright.

1.3 I am aware that the author also retains the rights specified in clause 1 of this license.

2. I confirm that granting the non-exclusive license does not infringe third persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

---

[1] *Non-exclusive Licence for Publication and Reproduction of Graduation Thesis is not valid during the validity period of restriction on access, except the university`s right to reproduce the thesis only for preservation purposes.*

_____ (*signature*)

10/06/2021

Department of Electrical Power Engineering and Mechatronics

THESIS TASK

Student: Mohammad Mokhalled, 195454MAHM

Study programme: Mechatronics (MAHM)

Supervisor: Professor Mart Tamre, +3726203202

Co-supervisor: Dr. Ivar Kruusenberg, +3725036963

Thesis topic:

(in English) *EMBEDDED SOFTWARE FOR SMART HYDROGEN FUEL CELL GENERATORS*

(in Estonian) ARUKA VESINIK-KÜTUSEELEMENDI MANUSTARKVARA

Thesis main objectives:

1. Efficient and reliable software to control a fuel cell stack and its output
2. Increasing the fuel cell stack's life by keeping the conditions as ideally as possible using the embedded software
3. Keeping the source code reusable to use in different generators with different hardware

Thesis tasks and time schedule:

No	Task description	Deadline
1.	Designing the software architecture	31/01/2021
2.	Impelementing the program	15/04/2021
3.	Testing all software modules on the hardware	31/04/2021
4.	Writing the thesis	08/06/2021

Language: English **Deadline for submission of thesis:** 10/06/2021

Student: Mohammad Mokhalled *10/06/2021*

/signature/

Supervisor: Professor Mart Tamre …….. "....."....20….a

/signature/

Co-supervisor: Dr. Ivar Kruusenberg ….....…….. "....."....20….a

/signature/

Head of study programme: Mart Tamre ..... "....."....20….a

/signature/

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ADC	Analog-to-Digital Converter
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
DMA	Direct Memory Access
FMC	Flexible Memory Controller
GCC	GNU Compiler Collection
GPIO	General Purpose Input/Output
GPS	Global Positioning System
GSM	*Global System for Mobile* communications
HAL	Hardware Abstraction Layer
HRTIM	High-Resolution Timer
I2C	Inter-Integrated Circuit
LCD	Liquid-Crystal Display
LTDC	LCD-TFT display controller
MCU	Microcontroller Unit
MISO	Master-In Slave-Out
MOSFET	Metal–Oxide–Semiconductor Field-Effect Transistor
OLED	*Organic Light-Emitting Diode*
PC	Personal Computer
PEKE	polümeerelektrolüüt-kütuseelement
PEM	Proton Exchange Membrane
PID	Proportional–Integral–Derivative
PWM	*Pulse Width Modulation*

QSPI	Quad SPI
RAM	Random Access Memory
RGB	Red, Green, Blue
RPM	Revolutions per minute
RTOS	Real-Time Operating System
SMBUS	System Management Bus
SMS	Short Message Service
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
TFT	Thin Film Transistor
UART	Universal Asynchronous Receiver Transmitter

1 INTRODUCTION

Increasing the demand for energy leads to an increase in using fossil fuels which causes global warming and climate change. Because of the effects of global warming on the earth, the energy industry is ready to use clean, renewable, and sustainable energies. Since Hydrogen is a sustainable and zero-emission fuel, it is getting more and more popular as a fuel. Between the different functionalities that a Hydrogen fuel cell generator can have, the portable fuel cells have the least share of the market that shows a great opportunity to work in this field. Moreover, there is modest research and product in the field of portable generators, which can be worthwhile for all the situations that there is no access to the electricity grid. For this purpose, a type of fuel cell should be chosen that is lightweight and does not need a very high temperature for starting up.

The thesis discusses two different portable fuel cell generators with different specifications. Also, as the software for embedded systems of a generator can be so critical in safety and there is no research about it, this thesis covers the software implementation of these products. Since the embedded software is completely dependent on hardware, the hardware is described and shown in the thesis. The project has several challenges. The main challenge for the software is to have software that can be used as the main platform for different generators with different components by the least possible changes. Moreover, the challenge of working with a dual-core CPU and synchronising all the different parts is another challenge.

In general, this document aims to show an efficient, reusable, and modular implementation of firmware for a hydrogen fuel cell generator. On the other hand, the safety of the consumer devices and users is one of the most critical parts that must be achieved by this software. The software should recognise the possible errors and critical situation as soon as possible which makes which means no part of the program must not block the CPU processes or the MCU peripherals.

2 BACKGROUND RESEARCH

2.1 Hydrogen as fuel

Global warming is one of the worldwide concerns that lead to climate change. There are many suggestions to control global warming; one of these approaches is using hydrogen fuel cells instead of fossil fuels [1]. Fossil fuels produce greenhouse gasses emissions that mainly include carbon dioxide (CO_2), methane (CH_4), and nitrogen oxide (NO_2), which cause global warming; hence, mitigating this problem by applying hydrogen fuel cell can be an excellent option since they do not have emissions compare with fossil fuels [2].

Although the structure of hydrogen is the simplest among other molecules, it has the highest amount of energy content that can be employed as a fuel application [3].

Hydrogen fuel cells can be used for various applications such as transportation, material handling, backup power application, etc.

Although hydrogen as a fuel has its own drawbacks, still the advantages outweigh the disadvantages. For example, compared to any other chemical fuel, hydrogen has the most energy per unit mass [4]. It is environmentally friendly and has low emissions[5]; most carbon-based emissions, nitrogen oxides and other greenhouse gas emissions are eliminated by using hydrogen as fuel. For more explanation, nitrogen oxides can lead to other environmental issues like acidification, formation of smog, and ozone layer depletion [4]; therefore, there would be no nitrogen oxides in the emissions when there is no nitrogen oxides in the emissions other environmental problems. However, it should be mentioned that the amount of nitrogen oxides produced during this procedure is not significant and can be ignored. Adopting hydrogen in a jet can stabilize the fuel price and because it can be obtained from different sources, it causes a reduction of reliance on fossil fuel in some region of the world [6]. Another benefit of a hydrogen fuel cell is it has slight vibration, and it is silent [4]. Other advantages of this fuel include long time usage, no visual pollution, reduction in carbon footprint [7].

On the other hand there are also some disadvantages of using hydrogen as a fuel. For instance, it is expensive because it is difficult to separate this element from others. One difficulty that arises from this fuel is storing and transporting of that, since moving around this element is so hard. In addition flammability risk of this fuel is high, because

it burns in the air very easily [7][8]. It has lower power density and lower power response in comparison with conventional fuel [9].

2.2 Types of hydrogen fuel cells

In general, fuel cells are devices that generate electricity through a chemical reaction. Each fuel cell has two electrodes called a cathode and an anode. The reaction that produces electricity occurs at these electrodes. Each fuel cell has an electrolyte that carries electrically charged particles from one end of the electrode to the other. Each electrode also has a catalyst that speeds up the reactions that occur at these electrodes. By this procedure, Hydrogen fuel cells can produce electricity constantly[10][11]. While fuel cells generally operate under similar principles, there are differences between different types of fuel cells. The different types of fuel cells are explaining in this section.

2.2.1 Alkaline fuel cell

This kind of fuel cell operates on compressed hydrogen and oxygen. They typically use a solution of potassium hydroxide in water as their electrolyte. Efficiency is about 70%, with operating temperature ranging from 150 to 200$^{\circ}c$ [12][13]. Cell output ranges from 300 watts to 5 kilowatts. Alkali cells were used in the Apollo spacecraft to provide electricity and drinking water to the crew [14]. However, they require pure hydrogen, and their platinum electrode catalysts are expensive. Since they are a container filled with liquid, they also run the risk of leaking. Figure 2.1 shows the mechanism of an alkaline fuel cell membrane.

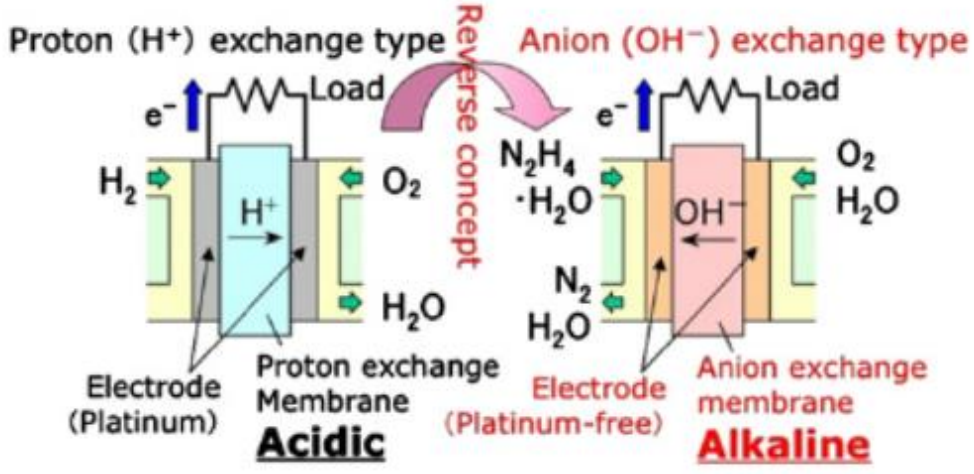

Figure 2.1 The mechanism of an alkaline fuel cell[15].

2.2.2 Molten carbonate fuel cell

Molten carbonate fuel cells use high-temperature compounds of salt carbonate as their electrolyte. Efficiency ranges from 60 to 80 percent [16]. An operating temperature is about 650 $^{\circ}C$ [17]. Units have been constructed up to two megawatts, and designs for up to a hundred megawatts exist[18] [19]. The high-temperature limits damage from carbon monoxide poisoning of the cell, and waste heat can be recycled to make additional electricity. Nickel electrode catalysts are inexpensive compared with Platinum used in other cells, but the high temperature limits the materials and safe uses of this cell type as they would probably be too hot for home use. In addition, carbonate ions from the electrolyte are used up in the reactions, making it necessary to inject carbon dioxide to compensate [17]. Figure 2.2 illustrate the mechanism of molten carbonate fuel cell.

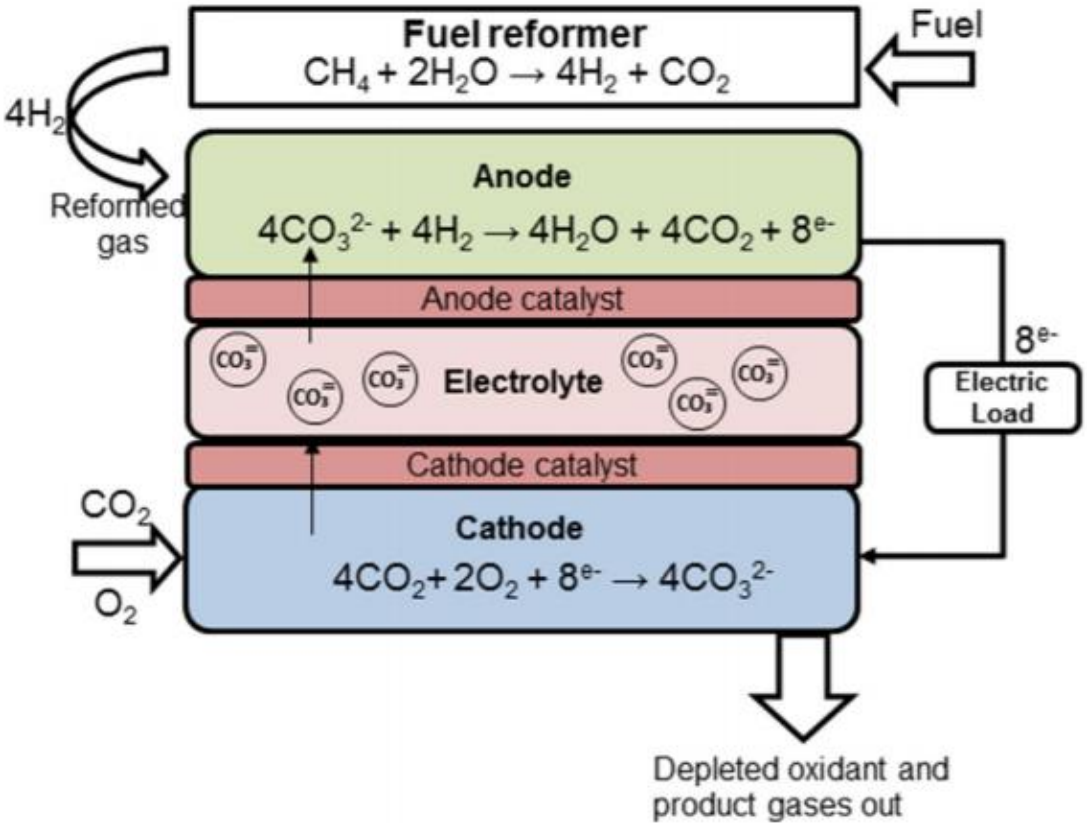

Figure 2.2 Scheme of a molten carbonate fuel cell with chemical reaction [20]

2.2.3 Phosphoric acid fuel cell

It uses phosphoric acid as its electrolyte. Efficiency ranges from 40 to 80 percent [21], and operating temperature is between 170 and 210$^{\circ}C$ [22]. Acid fuel cells of up to 200 kilowatts exist and units up to 11 megawatts have been tested [21]. They are able to tolerate carbon monoxide concentration of about 2 percent [23]. Broadening the choice of usable fuel cell sources though if gasoline is used, the sulphur must be removed prior to use. Platinum electrode catalysts are needed, and internal parts must be able to withstand the corrosive acid [24]. Figure 2.3 represents how a phosphoric acid fuel cell works.

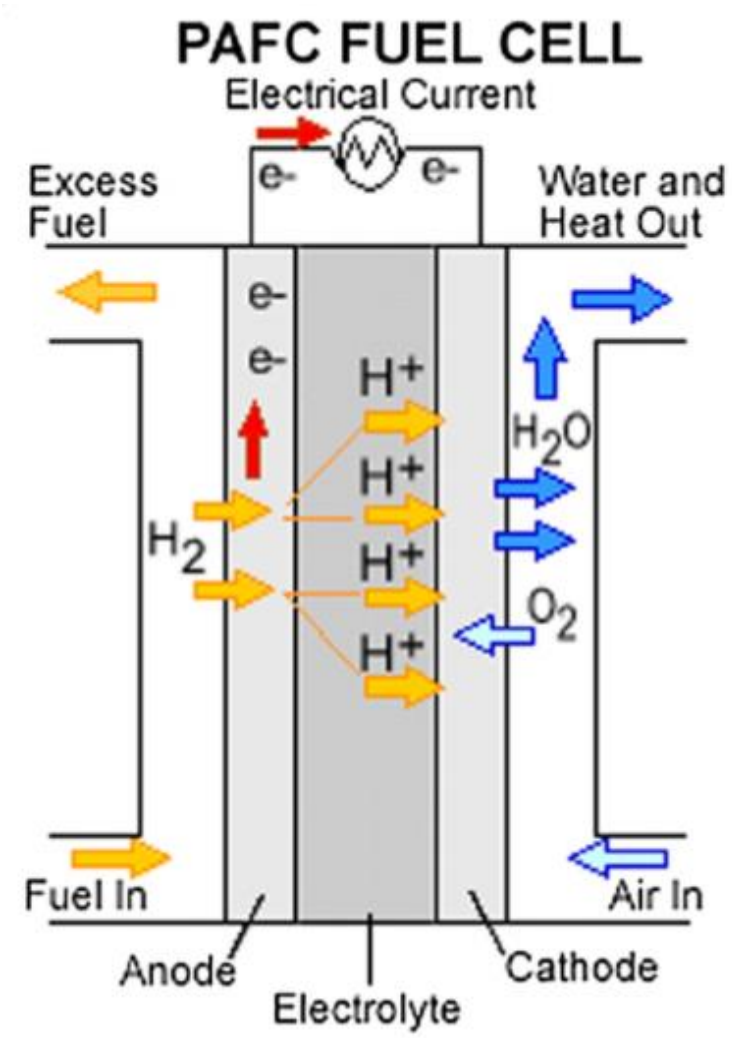

Figure 2.3 Schematic figure of a phosphoric acid fuel cell [25]

2.2.4 Proton exchange membrane fuel cell

Proton exchange membrane fuel cells work with polymer electrolyte in the form of a permeable sheet. Efficiency is about 40% to 50% [13], and the operating temperature is around 60 to 80$°C$ [26]. Cells output available range is from 50 to 250 kilowatts [27]. The solid and flexible electrolyte will not leak or crack. They operate at a low enough temperature to make them viable for homes and vehicles. However, the fuel cells must be purified, and platinum catalyst is required on both sides of the membrane, raising costs [28]. The scheme of one proton exchange membrane fuel cell is shown in Figure 2.4.

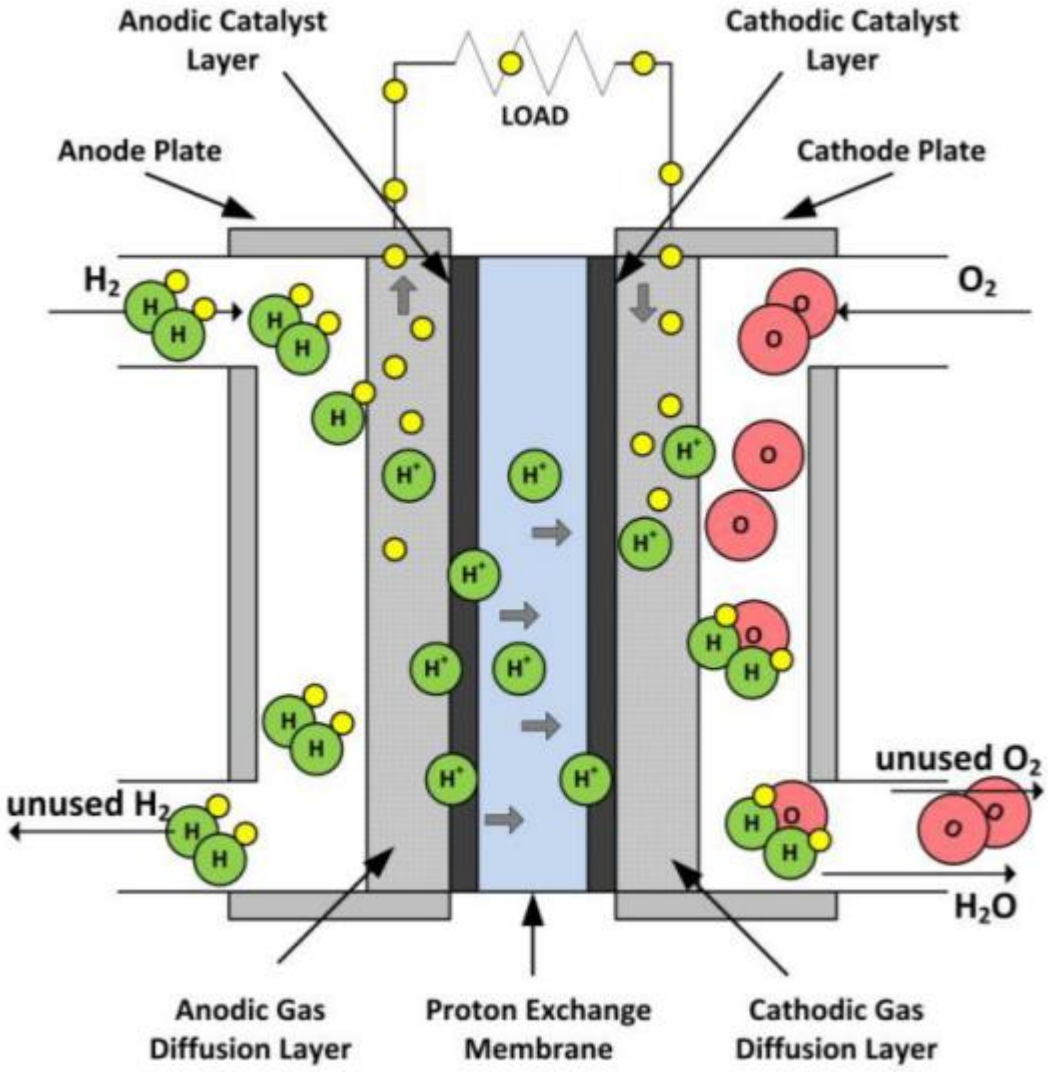

Figure 2.4 Schematic representation of Proton Exchange Membrane fuel cell [29]

2.2.5 Solid oxide fuel cell

Solid oxide fuel cells use a hard ceramic compound of metal oxides such as calcium or zirconium as an electrolyte. Efficiency is around 60% [30]. The operating temperature is around 700 to 1000°C [31]. This type of fuel cell can output 100 kilowatts [32]. At the high temperatures produced by this type of fuel cell, a reformer is not required to extract hydrogen from the fuel, and waste heat can be recycled to make additional electricity [33]. The high temperature also limits the potential applications of this fuel cell type [34]. They tend to be fairly large in size. While their solid construction means that they cannot leak, they are capable of potential cracking [35]. Figure 2.5 Indicates the process of a solid oxide fuel cell.

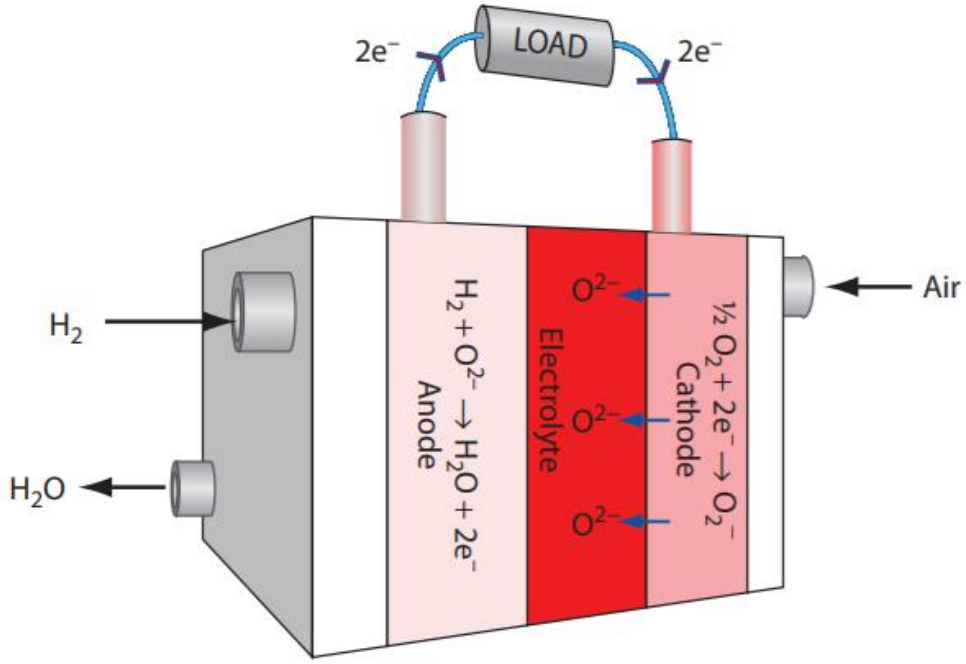

Figure 2.5 Schematic representation of a solid oxide fuel cell [36]

2.3 Hydrogen fuel cell applications

The Hydrogen fuel cells can be separated into three major groups based on their application that are transportation, stationary, and portable (Mobile). Each type of fuel cell that is described can be more suitable for one of these segments. This section discusses the different applications that fuel cells have.

2.3.1 Transport

Transportation is the most significant application of Hydrogen Fuel Cells, which consumes the most generated power in the world [37]. Many types of vehicles use Hydrogen as fuel which needs fuel cell to generate power from it. These vehicles can be automobiles (passenger cars and buses), motorcycles and industrial trucks, marine vehicles, aeroplanes, and spacecraft [38] [4]. As the research shows, the demand for this application is increasing rapidly 300 MW per year in 2016 to around 1000 MW per year in 2020 [37]. Large companies like Toyota investing in this area which makes a brighter future for it [39].

2.3.2 Stationary

Stationary hydrogen fuel cell application is usually known as Hydrogen power plants. These power plants use one or multiple fuel cell stacks in a grid to generate energy for an area, a building or a specific purpose with or without using it in combination with another energy resource [40]. This is the second most used application in recent years. Usually, these fuel cell generators have more efficiency, but they are much heavier than the other fuel cell applications [37].

2.3.3 Portable

Portable fuel cells are lightweight fuel cells that can move easily. They are usually used to power outdoor personal uses like battery charging or consumer electronic devices such as laptops and smartphones. Actually, they produce electricity in places that there is no access to the electricity grids. Moreover, they can have military applications [40]. This type of generator can be used to supply the power of caravans, yacht, or camping areas. Also, this the least used application comparing to the other areas of application

in recent years [37]. It means there are many opportunities for small companies to work on it. This is the application group of hydrogen fuel cell that is described in this thesis.

2.4 Similar products

There are different competitors for the Hydrogen generators described in the thesis that are shown here and discussed their strengths and weaknesses. The UP200 and UP1K products' details which this thesis focused on them are described in chapter 4.

2.4.1 H2SYS BOXHY 1

BOXHY 1 product is supposed to generate 1 kW energy. In the datasheet, it produces maximum 1100 W power at boost mode and 650 W at eco mode. Its weight is 25 kg, and the working temperature range is from +5°C to 45°C. Moreover, it has two 230 VAC outlets. It has a PEM fuel cell for generating power [41].

Figure 2.6 H2SYS Generator (5 kW) [41]

This generator can be compared with UP1K. The strength of it is the integrated AC output. However, it does not have any DC output or battery charger, which can be a weakness. Also, it cannot work in very cold and very hot weather. The other weakness

is that it cannot give the nominal output, and it can produce the max power for 20-30 minutes. Moreover, it is heavier compared to UP1K.

2.4.2 GreenBox 2 200

The rated power of this generator is 160 W with both AC and DC outputs. It has 13 kg weight [42]. It has a PEM fuel cell inside.

Figure 2.7 GreenBox 2 200 generator [42]

This generator can be compared with UP200. Its strength is different outputs that it supports like AC, DC, and battery charger. About the weaknesses, it does not generate 200 W, and its maximum output power is 160. Although it generates less power than UP200, its weight is more than it.

2.4.3 300W Portable Fuel Cell HyMo

HyMo generator's rated power is 300W. It has 220 VAC and 5 VDC outputs. The generator can work in the temperature from -5°C to 40°C. It is 4 kg (only the fuel cell). Also, its different parts are separated, and they are not placed in a closed enclosure. It uses PEM fuel cell.

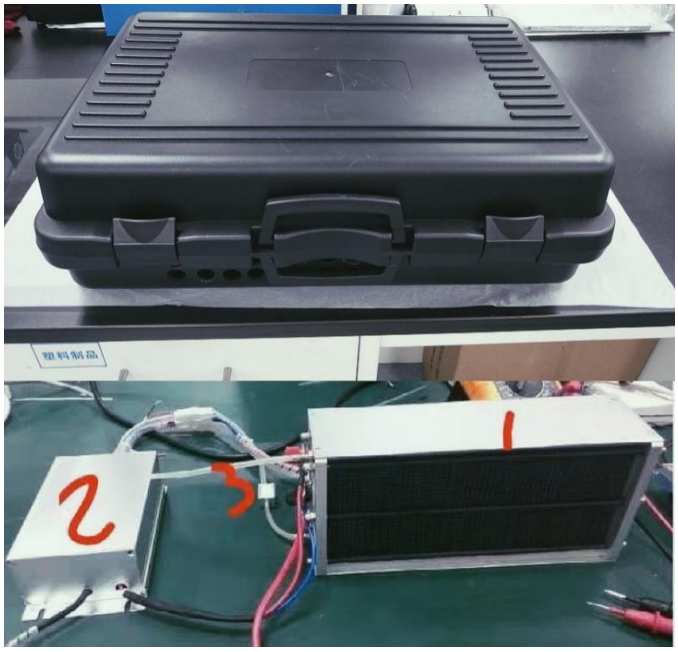

Figure 2.8 300W Portable Fuel Cell HyMo [43]

This generator can be compared with UP200. Although it has a smaller size, it does not have a solid structure, and each part should be connected to each other, which makes it difficult to use. Also, it does not have any capability for charging batteries.

2.5 Conclusion

In the fuel cell market, portable generators have the least share of the market compared to other fuel cell generators for other application types. It means mostly fossil fuels are used for portable generators, and it makes a neat opportunity to change the future of portable generators. Also, according to the specifications of different fuel cell types, PEM fuel cell is the best option for a hydrogen fuel cell portable generator. This fuel cell type has the least operation temperature and weight compared to the other types. The low temperature helps to start the generator at a temperature near room temperature safely and manageable. Also, the weight makes it carriable, which is the most important requirement as a portable device.

Generally, most hydrogen fuel cell portable generators in the market are heavy to carry and do not support smart solutions like Bluetooth, SMS, and GPS capabilities. PowerUP generators solve this problem and use a solid and reliable box.

On the other hand, UP200 and UP1K generators have just DC output which might be a disadvantage for this product. But, a DC to AC converter can be used to generate a 220 or 110 VAC for the general electrical consumers.

3 PEM HYDROGEN FUEL CELL STACK

The hydrogen fuel cell stack is the heart of generators that produce electricity by the chemical process explained in the previous chapter. The stack of the focused products consists of multiple cells that each cell generates less than 50 W power and a maximum voltage of 1 V. As the maximum current of each cell is 78 A, which is enough to supply consumers, the cells place in parallel in the stack.

This chapter explains how the fuel cell stacks should be controlled and what conditions should be considered in both hardware and software. Since the thesis discusses two different products, the number of cells in a product is assumed as a variable in this chapter to cover the calculation and values for both products.

3.1 Hydrogen fuel cell stack setup

The Hydrogen fuel cell stack needs Hydrogen as fuel to generate power and airflow. The Hydrogen input is controlled by a supply valve which is an electrical valve controlled by the MCU and electronics board. Before the valve, a pressure regulator must be placed to prevent damage to both the valve and fuel cell stack. On the other hand, to accelerate the oxidate reaction, a purge valve must be implemented to release the hydrogen in the airflow. For producing airflow, there are two fans on the two sides of the fuel cell stack that leads air from one side to another. Furthermore, the fans help to cool the stack when its temperature is high. The whole flow is shown in Figure 3.1.

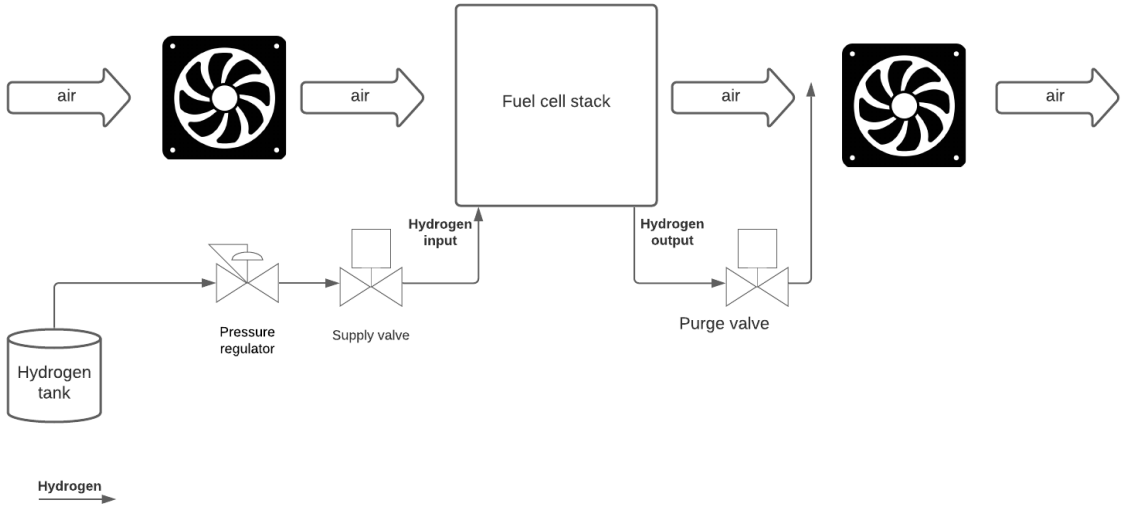

Figure 3.1 The fuel cell stack setup

3.2 Hydrogen fuel cell stack operations

Hydrogen fuel cell stacks need some operation for getting ready to work and work safely. Controlling and monitoring this operation is the main goal of this thesis. This chapter discusses the conditions that must be met for getting the fuel cell stack safely. Chapter 3 and 4 explain how the hardware and software were designed to make this applicable.

The hydrogen fuel cell stack has five primary operation states, which are described here briefly.

Standby state is the state that the fuel cell stack does not produce or deliver any power

Start-Up state changes the stack's state from Standby to a state that stack can give out current

Warmup state changes the state from Start-Up to the state that stack deliver rated power safely

Run is the state that the stack delivers rated power

Shutdown is the state that changes the stack's state from Run state to Standby state

The following figure shows changing the states in order.

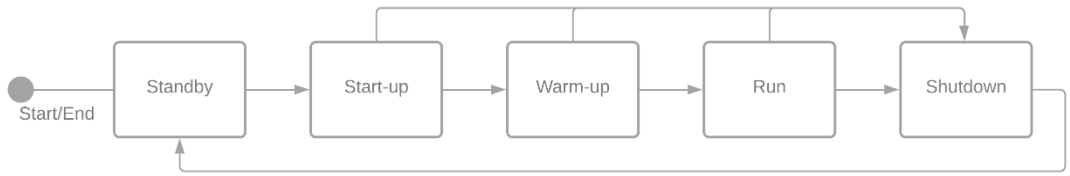

Figure 3.2 stack operation states

The following sections describe the states in detail.

3.2.1 Standby

In this mode, the stack does not consume any hydrogen or produce any power. It is the first state after pressing the power button and the last state before disconnecting the power and turning the stack off completely. The actuators in this mode are in their safe state.

The minimum ambient temperature that the stack should be in to be able to go to the run state is -10 °C. Otherwise, the stack does not work and cannot produce power. In the described products, because of the mechanical structure of the generator, the whole generator works in the temperature down to -20 °C, which means it keeps the generator isolated from the outside temperature. Also, the ambient rated humidity should be more than 2% for going to start-up mode. Otherwise, it needs almost 2.5 times longer time to reach the run mode and deliver the rated power.

Table 3.1 Stack's standby actuators status [44]

Actuator	Command
Fan	0%
Hydrogen Valve	Closed
Purge Valve	Closed
Stack Contactor	Open
Stack Current	0 A

3.2.2 Start-up

The start-up state is the first step to enable the output current of the stack. It has three steps inside.

Step 1 purpose is to spool up the fan. It also gives enough airflow for diluting after opening the purge valve. This step lasts until the fan reaches the desired speed.

Step 2 gives fresh Hydrogen to stack. In this step, the voltage of the stack reverses because of the exchanging air in the anode with hydrogen gas. The purge valve must be open at the same time as the hydrogen valve to reduce the performance loss in the reversing voltage phase. Also, a load applies to the output of the stack to reduce the start-up corrosion.

Step 3 still applies the load to pull down the average cell voltage to reduce the corrosion.

Table 3.2 Stack's standby actuators status and its different steps [44]

Step	Actuator	Command
1	**Fan**	70%
	Hydrogen Valve	Closed
	Purge Valve	Closed
	Output of Generator	Closed
	Stack Current	0 A
After the fan reaches the desired speed (Fan spool up time)		
2	**Fan**	70%
	Hydrogen Valve	Open
	Purge Valve	Open
	Output of Generator	Closed
	Stack Current	Start-Up Current
After 0.2 seconds (Start-up purge duration)		
3	**Fan**	10% (Fan minimum)
	Hydrogen Valve	Open
	Purge Valve	Closed
	Output of Generator	Closed
	Stack Current	0 A

3.2.3 Warmup

The warmup state increases the temperature and hydrates enough the membrane to enable the stack to deliver rated power. In just a few cases, this step happens fastly, but usually, this step takes more than 1 minute to be finished. The ambient temperature, humidity, and stack hydration level increase the duration of this step. It leads to taking more than 5 minutes in the situation, as the first power on after a long while.

Table 3.3 Stack's warmup actuators status [44]

Actuator	Command
Fan	10% (Fan minimum)
Hydrogen Valve	Open
Purge Valve	Opens each 30 seconds for 400 ms
Output of Generator	Closed
Stack Current	Described below

Current, in warmup state, must be controlled by an internal load. The goal is to keep the voltage on minimum and the current high which makes the the stack warm faster. For this purpose the voltage relation to each current value is calculated frome the following chart.

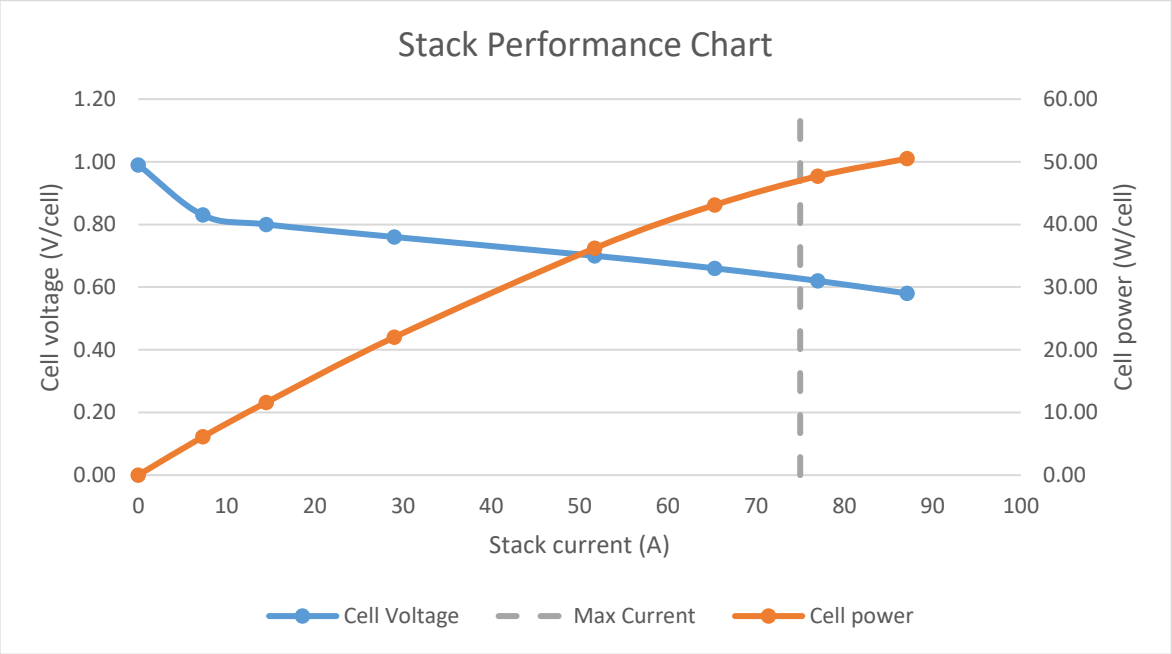

Figure 3.3 Power and Voltage of a cell according to the stack's current chart [44]

According to Figure 3.3, there are the following equations for the relation between stack current and voltage.

$$if\ v_{Stack} \leq 7{,}0\ A:$$
$$v_{Stack} = Cells \times (-0.0178 \times I_{stack} + 0.93\ V) \quad \text{[44] (3.1)}$$

$$else:$$
$$v_{Stack} = Cells \times (-0.0031 \times I_{stack} + 0.823\ V) \quad \text{[44] (3.2)}$$

Where V_{stack} - output voltage of stack, V,

Cells - number of cells in the stack,

I_{stack} – output current of stack

In this stage by increasing the load, the system waits for the stack to ptovide the calculated voltage. Then the system increases the load and waits for the voltage until the stack can provide 75% of the rated power. After this, the stack is ready to deliver rated output power safely, and it can change the state to the run state.

3.2.4 Run

This state is the normal working state of the stack that delivers power to the consumer. Most of the time that stack is on, is spent in this state.

Table 3.4 Stack's actuators status in run state [44]

Actuator	Command
Fan	Described below
Hydrogen Valve	Open
Purge Valve	Described below
Output of Generator	Open
Stack Current	0 to 75 A

Fan, in this state, works to keep the stack temperature (T_{stack}) near the optimal temperature (T_{opt}) as well as keeping enough flow to provide oxidant for the fuel cell reaction. For this purpose, a linear PID control can be used. The following graph shows how the control should be.

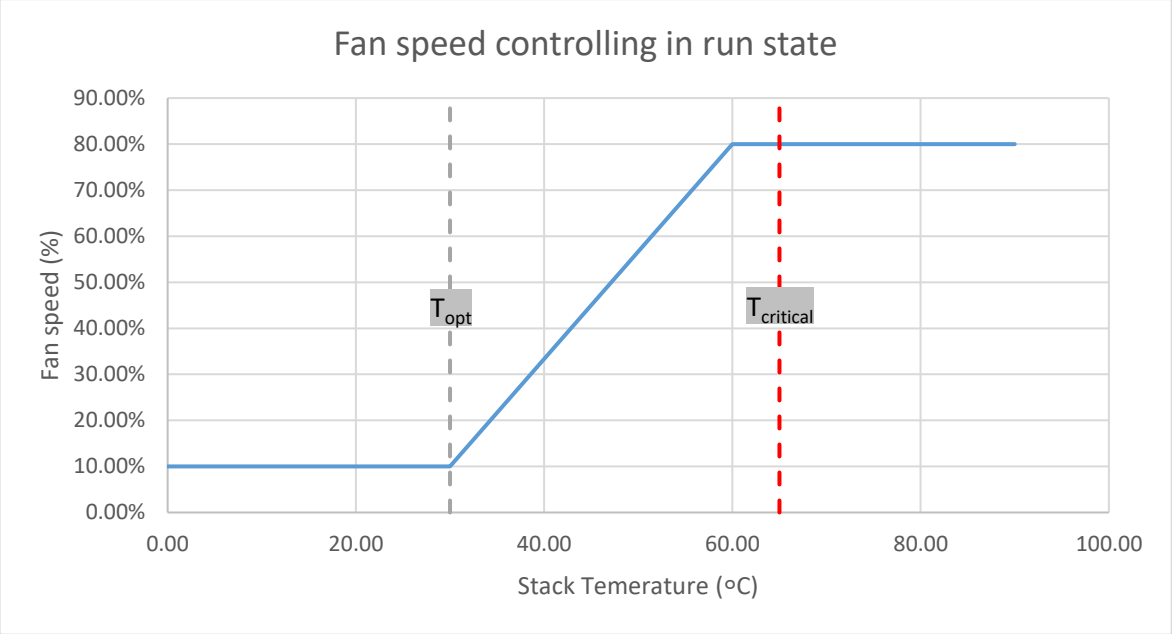

Figure 3.4 fan speed according to the stack temperature in run state [44]

Purge valve should be open for less than 500 ms every 2300 A.s. The following equation shows the condition that it should happen. It means when the consumption is higher, the purge valve will be opened more frequent.

$$\sum_{previous\ purge}^{now} I_s \times t_s > 2300\ A.s \qquad [44]\ (3.3)$$

Where I_s - last measurement of stack current, A,

t_s – current measurement sampling interval, s

For example, when the current value is 65.3 A, almost every 35 s, the purge valve opens. Moreover, when the current value is 7.3 A, almost every 315 s, the purge valve is opened.

3.2.5 Shutdown

It changes the stack from any state (usually run) to the standby state, where the connections to the stack can be released safely. The goal of this state is to cool down and dehydrate the stack. For this purpose, the fans must be on their maximum speed until the stack temperature reaches 20 °C or the ambient temperature if it is higher than 20 °C.

Table 3.5 Stack's actuators status in shutdown state [44]

Actuator	Command
Fan	Maximum
Hydrogen Valve	Closed
Purge Valve	Closed
Output of Generator	Closed
Stack Current	0 A

3.3 Stack alarms and error states

There are several alarm and error states that should be considered in the system. These errors take the stack into a critical state that might be dangerous for the user and also the stack itself. This section describes these situations and conditions in brief.

3.3.1 Stack temperature

However, the stack highest temperature is 75 °C; for more safety, if the temperature goes higher than 65 °C, the output should be disconnected because consumption of power makes the stack warmer. Moreover, a cool-down process should start with fans. The fans must work at the maximum speed until the temperature comes back to the

normal value. As mentioned, the fans start working with the maximum speed as soon as the temperature reaches 60 °C. The linear control usually does not let the stack reach the highest temperature except in the very high ambient temperature.

3.3.2 Stack current

If the stack current exceeds 78 A, it must disable the output. This reaction must happen in less than 2 s to prevent damaging the stack. There can be a refreshing mechanism to enable the output once a while for some milliseconds and check if the load is decreased to reenable the output or not.

3.3.3 Stack Voltage

If the stack voltage is more than 1.05 V/cell or $number\ of\ cells \times 1.05\,V$, then again, the output must be disabled and go to the normal mode. It rarely happens for the stack.

4 HARDWARE LAYOUT

This thesis discusses two different hydrogen fuel cell generators with different output powers. As the products' name defines, UP1K produces 1 KW output energy, and UP200 produces 200 W output. This difference caused many changes in the Hardware and Electronics design. This chapter reviews the hardware for each product.

It is important to mention that the thesis writer was not involved in designing and producing any of the hardware, except for checking the possibility of integration with software and participating in the main concepts before designing and just for planning. All the hardware are outsourced or designed and produced by PowerUP Energy Technologies Company.

4.1 UP1K

The following table shows the product specifications.

Table 4.1. UP1K generator specifications [45]

Output voltage	24 V DC | (48, 72 V DC)* * by using a converter
Max charging current at 24 V	42 A
Max continuous power output	1000 W
Max charging power per day	1008 Ah
Electricity source	PEM fuel cell
Fuel	Compressed Hydrogen
Fuel consumption (at 1000 W)	13 l/min
Size (L × W × H)	600 × 170 × 421 mm
Weight	20 kg
Operating temperature	−20 °C to +50 °C
User interface	On unit via Bluetooth to mobile or PC
Outlet and inlet ports	
Main outlet port	Max 21 A
External outlet port options	USB, high current port
Inlet port for external charging	Max 10 A

As this product's hardware should handle more than 42A current, which is a pretty high current, this product is more difficult to control. Because of that, more advanced microcontrollers are used to control the different parts of this generator. Also, since the product is in the development and test stages, its hardware is designed modular. As it is discussed later, it helped us to change some approaches separately and with fewer expenses.

On the other hand, whereas this generator can handle more power consumption for internal hardware, a TFT LCD is used as a user interface to configure and monitor the generator's status.

4.1.1 General Structure

In the development process, different iterations of hardware are tested. This section shows the different versions that were developed and explains the reasons for changing and using them. These changes affect the software structure and functions a lot.

Version 0.1 is developed to have a Front Panel Board for the interfaces (LCD, Buttons, GSM, Bluetooth, and upgrading process) and a Controlling MCU board to control the Fuel Cell Stack processes, Fans, and DC-DC Buck-Boost Converter.

After software testing, it shows that the DC-DC converter needs much more monitoring and real-time feedback than was expected. This problem was the reason for moving to the version 0,2 approach.

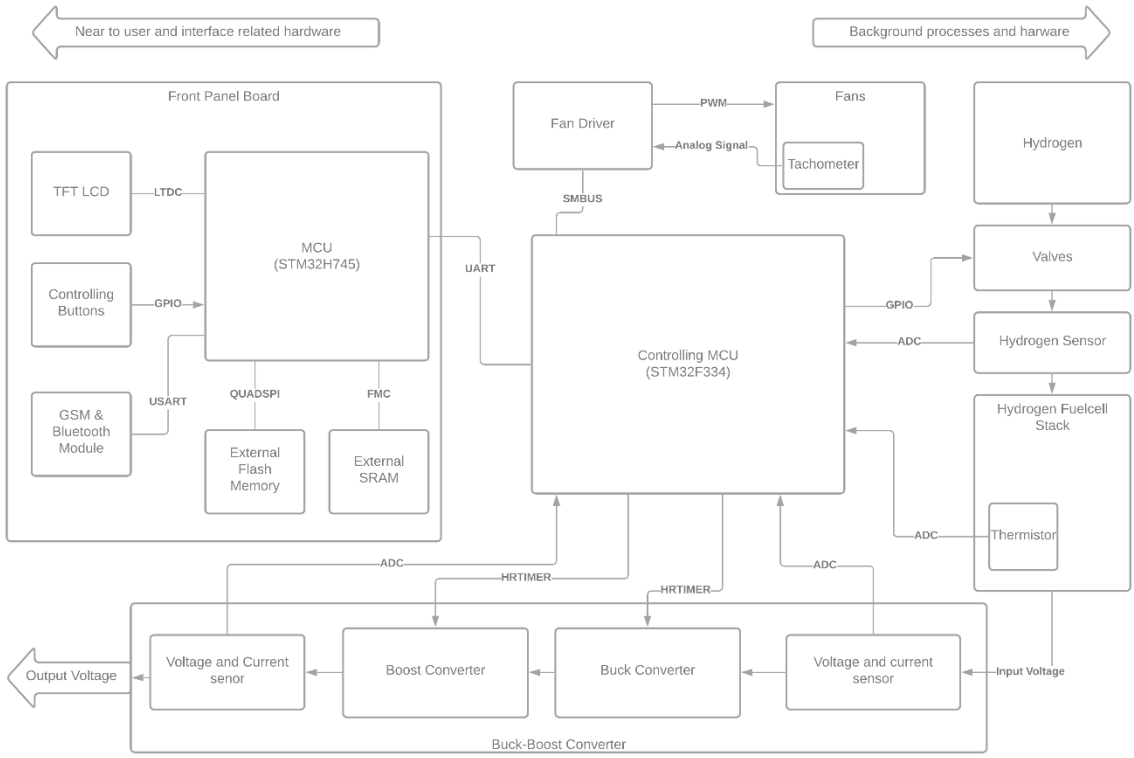

Figure 4.1. UP1K general structure version 0.1 diagram

Version 0,2 uses the same hardware modules as the previous version, but some connections and the software tasks for each MCU board are different.

As the front panel board's MCU has two cores, it is possible to split the processes on the cores. For this purpose, the Cortex M7 core handles the graphics, LCD, and other interfaces. The Cortex M4 core controls the fuel cell stack process and communications with controlling MCU. It could increase the performance of the whole system and especially the DC-DC converter.

However, the performance and efficiency increased, there was still some delay in controlling the DC-DC converter. It happened because of the UART communication between the front panel and controlling MCU boards.

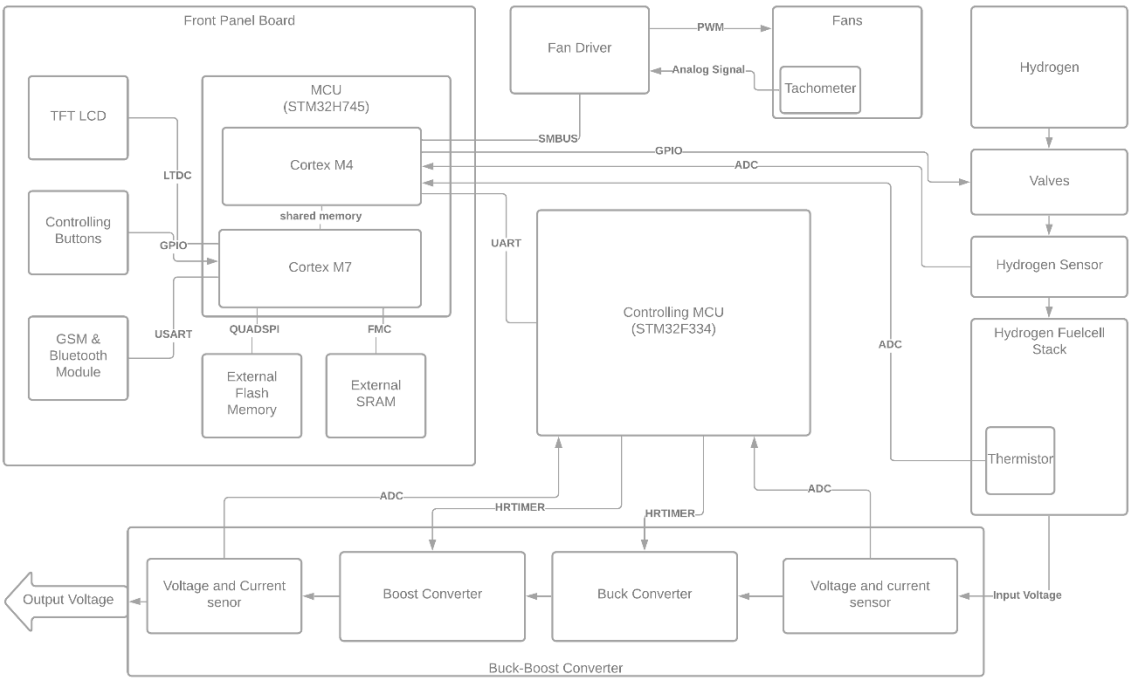

Figure 4.2. UP1K general structure version 0,2 diagram

Version 0,3 removes the controlling MCU and uses an external DAC to control the DC-DC converter. In this version, there is no need to control the DC-DC converter using PWM signals, and it should be controlled by a DAC connected to the MCU via SPI. In this case, the performance and safety become maximum because the interface processes are done on the separate processor's core, and in the case of a problem on the other core, it cannot stop the stack controlling process. The following figure shows the final layout of the hardware till writing the thesis.

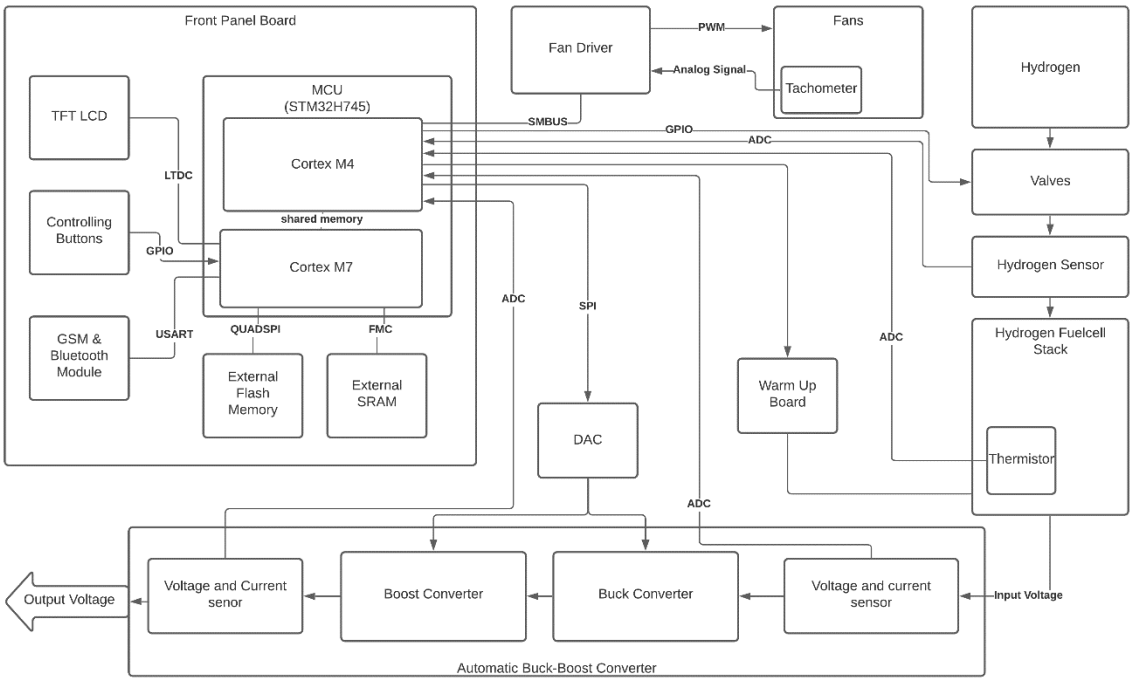

Figure 4.3. UP1K general structure version 0,3 diagram

4.1.2 Front Panel Board

The following table shows the components which are included in the front panel board.

Table 4.2. The UP1K Front Panel components

	Part Number	Features
Microcontroller	STM32H745XIH6	Arm® Cortex® core-based microcontroller with 2 Mbytes (STM32H745XIH6) or 128 Kbytes (STM32H750XBH6) of Flash memory and 1 Mbyte of RAM, in TFBGA240+25 package [46]
GSM and Bluetooth Module	Quectel MC60	Quad-band GSM/GPRS/GNSS module Bluetooth 3.0 [47]
External Flash Memory	MT25QL512	2 x 512 Gbit (128MB) Quad-SPI NOR Flash memory [48]
External SRAM	MT48LC4M32B2B5	128 Mbit SDRAM (32 x 4 Banks of 1 Mbit) [49]
TFT LCD	SC7283	720x544 System-On-Chip Driver for 480RGBx272 TFT LCD [50]

This board is designed by the inspiration of the STM32H745I-DISCO [51] development board. All the software test and development is done on the mentioned board.

Microcontroller of this board is a dual-core Arm Cortex family. It helps to separate the more challenging to be run on Cortex M7, like graphics processing and driving LCD, to make a higher performance for controlling the fuel cell stack on the same chip. In general, the Cortex M7 core is responsible for the interface like GSM, Bluetooth, and driving LCD, while the Cortex M4 Core is responsible for monitoring and controlling the internal hardware like fuel cell stack, DC-DC Converter, fan drivers. Another task that the Cortex M7 core is supposed to do is keep the bootloader and update the firmware using UART and Bluetooth.

GSM and Bluetooth are used in this board to communicate and alert the user in critical situations. Also, Bluetooth is used for updating the firmware. The other feature of this module is GNSS which can help to locate the product which is portable. This module is connected to Microcontroller by UART with Rx, Tx, RTS and CTS pins. The handshake pins (RTS and CTS) are used for more reliability in the transmission of data.

TFT LCD is used for a better user experience. As with the other products, just a one-colour OLED is used to display the data; the feedbacks showed it is better to use a colourful display as the user interface. This device is connected to the MCU using an LTDC port which can manage a much higher datarate than SPI. The LCD resolution is 480 × 270 which can show enough details in images.

Figure 4.4. A demo picture on the U1K LCD

External memories (SRAM and NOR FLASH) are used in this project as a resource to save and render images and graphics. The first software tests showed that the internal memories of the MCU are not enough to load two pages and the firmware of Cortex M7 (not the bootloader) moved to the external Flash. For the connection of Flash memory, Quad SPI protocol, and for the SRAM connection, an FMC port is used.

4.1.3 Controlling Board

It is a board with an STM32F334 Microcontroller and customisation of output pins related to our application. As mentioned in item 4.2.1, in the first version, it was connected to the fuel cell stack, fan driver, and DC-DC converter. In the second try, it was controlling just the DC-DC converter. In the final version, this board is not included in UP1K. The software and its details are described in the next chapter.

This board has five pairs of HRTIM output to control Buck-Boost Converters in the sync topology. Moreover, it has RS-485 and UART outputs for communicating with the Front Panel MCU. As it is a demo board, a programming socket is also placed on the board, supporting Serial Wire Debug (SWD). It also has an ADC input for reading the temperature of the stack. The rest of the pins are ground, 12, 5, and 3.3 V.

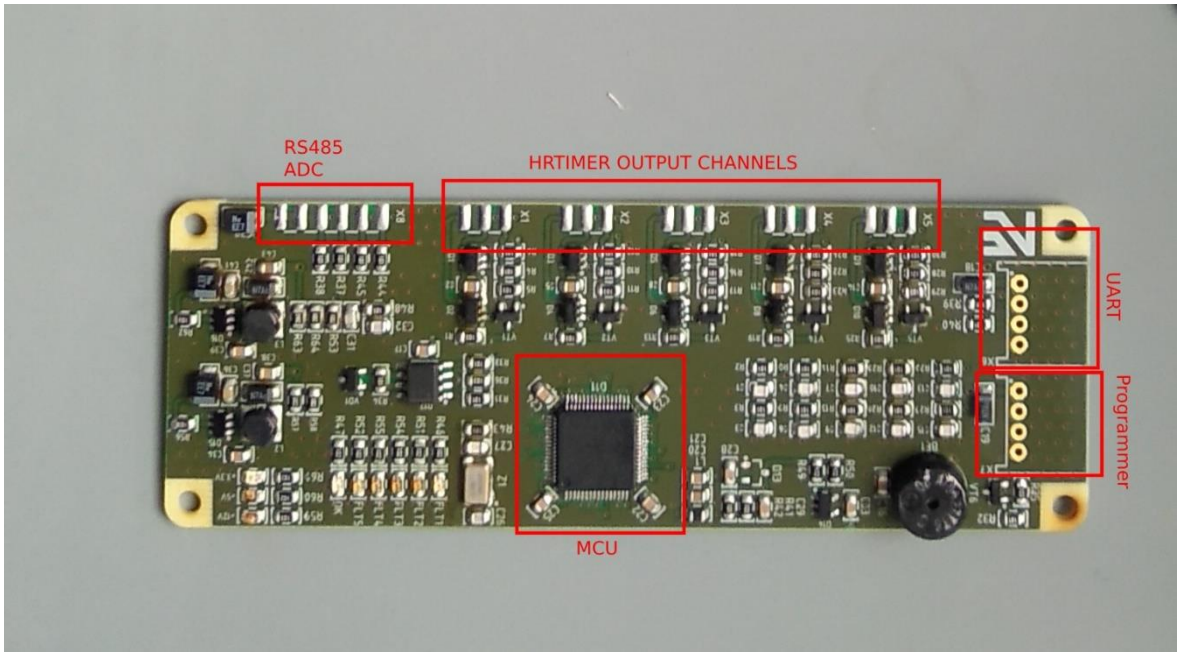

Figure 4.5 UP1K Controlling board picture and pinout

4.1.4 DC-DC Conveter (Buck-Boost Converter)

Since the desired output voltage for this product is 24 V, and the fuel cell stack's output voltage fluctuates in the range of 18 to 36, we need a converter that can step down (buck) and step up (boost) the voltage. Managing both functionalities for 1000 W is one of the most challenging parts of this project. Any unpredictable change of output voltage might damage the device that uses this generator as a power supply. This section discusses different solutions that are used to manage this problem.

Version 0.1 of Buck-Boost Converter must be controlled by an MCU to set a specific voltage regarding the input (fuel cell stack) voltage. The MCU could read the input and output voltage by a voltage divider that changes the whole voltage range with 0 to 3.3V to be readable by MCU. This version includes two separate boards for Buck and Boost. The following diagram shows how these board should have been connected to the controller MCU.

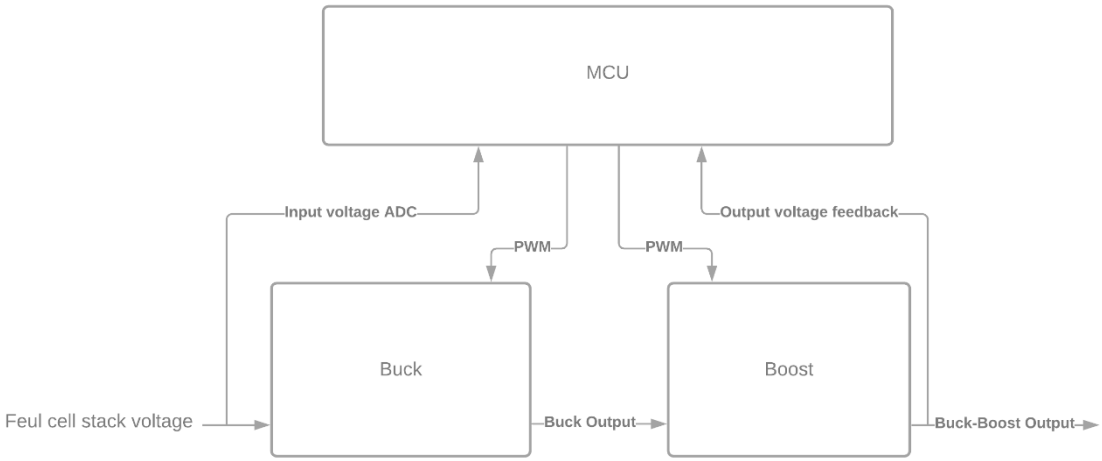

Figure 4.6. UP1K Buck-Boost Converter diagram version 0.1

These converters use sync topology, which means two 100 kHz PWMs with the same duty cycles and the reverse polarity are needed to control the value output voltage. Also, the secondary PWM should have dead-time for turning high to prevent any conflicts in turning on the transistors on the board and burning them. This kind of signal is shown in the following figure.

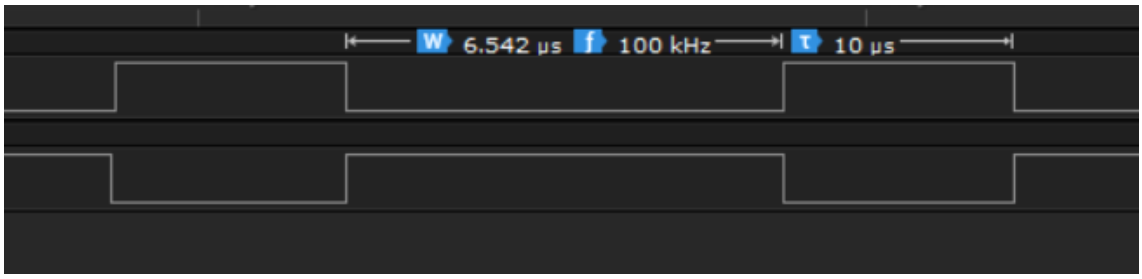

Figure 4.7. Sync PWM signal sample for controlling Buck-Boost Board

The buck and boost boards are quite similar in this project because they used the same topology and structure. The following picture shows these devices.

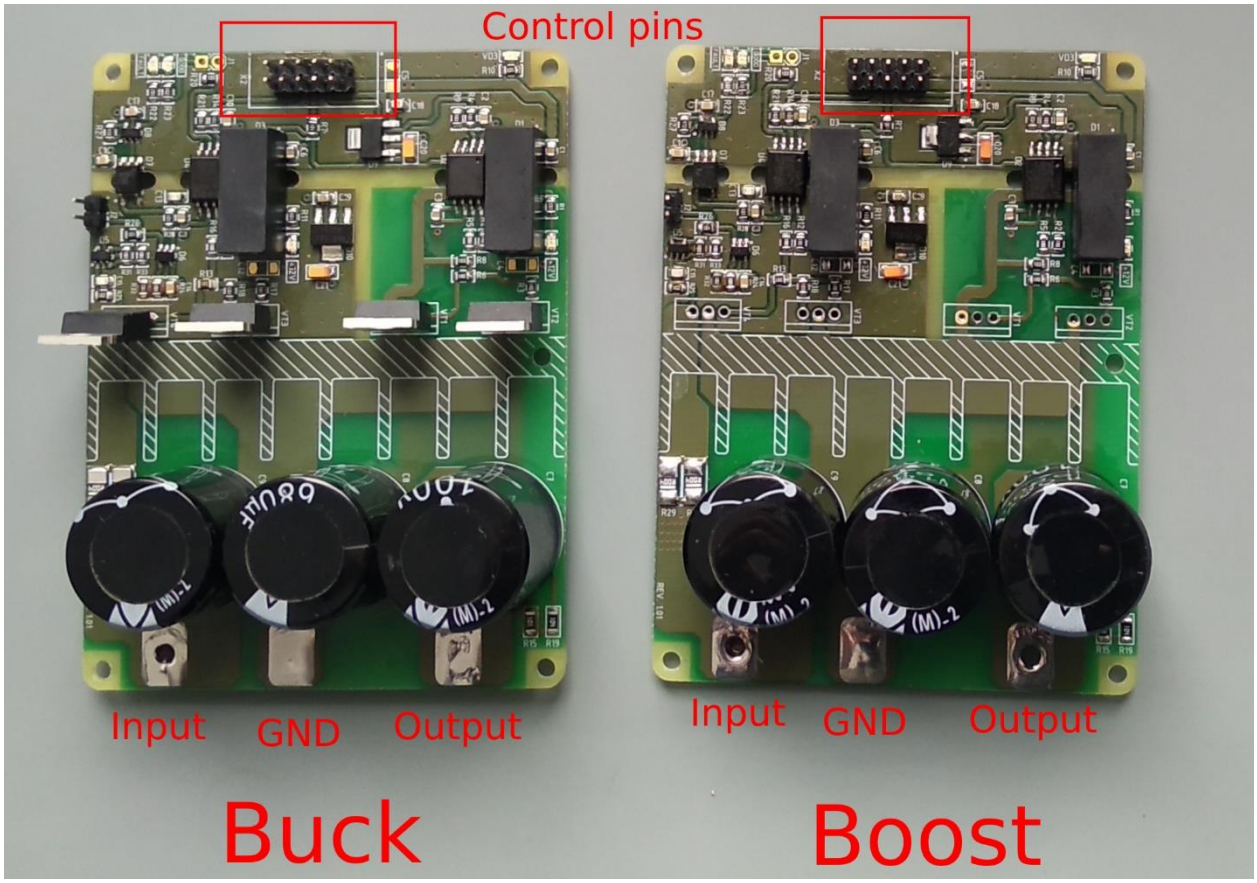

Figure 4.8. U1K Buck-Boost Boards version 0.1

4.1.5 Fan and valve driver board

There are two types of fan in this project. A group of fans is used to cool the hydrogen fuel cell stack, and the other group of fans is used to cool the Electronics board. The flow rate and speed of these fans are different, which cause to use of a fan driver that can drive different fans with different specifications. The fan driver used in this project is EMC2305 [52], which can drive up to 5 fans, and each fan's configuration can be different from the others. The communication protocol for commanding this board from a microcontroller is SMBUS. Due to the fact that SMBUS protocol operates based on I2C [53], the default configuration of this board is set on I2C, but for more reliability at the first stage of the program, it sets the communication protocol on SMBUS to use the alert pin and timeout features. As 6 fans are used in this product (3 fans for cooling stack and 3 fans for cooling the electronics boards), two of these boards are used. These boards are connected on the same communication bus with different slave addresses. Furthermore, because the alert pin is active low, it is connected to the alert pin by an *AND* gate. The And gate makes the alert pin of MCU active if any of the drivers alert pins become active.

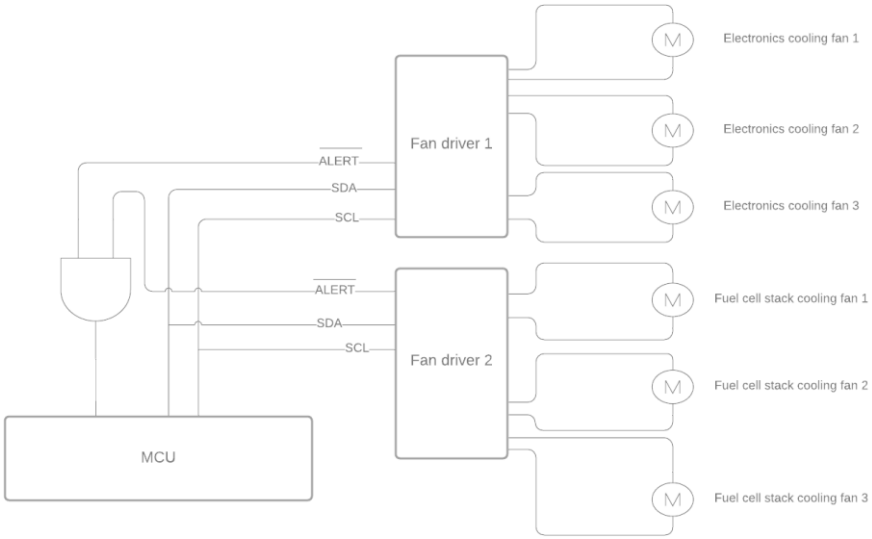

Figure 4.9. UP1K fan drivers connection diagram

The following table is the truth table of this connection. It shows that the MCU recognises the alert even if one of the devices enables the alert pin.

Table 4.3. Truth table for activation of SMBUS alert of fan driver

Alert 1	Alert 2	MCU Alert	Status
0	0	0	Error Alert
0	1	0	Error Alert
1	0	0	Error Alert
1	1	1	Normal

This board can also control the valve using DRV8876RGTR [54] chip. Because of the high current needed, the valves cannot be controlled by MCU GPIOs directly. The mentioned chip is just a bridge between the MCU GPIO with a small current and the valves that need more current. It also has feedback to check if the valve is open regarding the controlling signal.

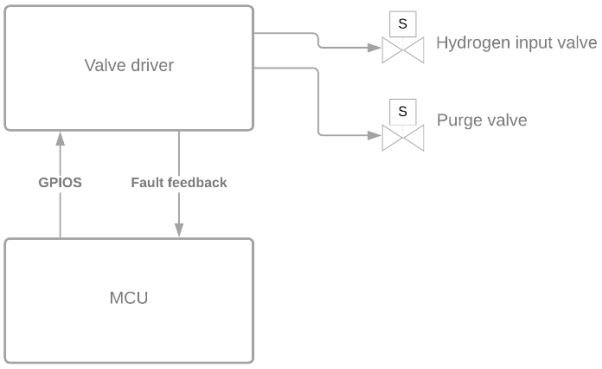

Figure 4.10. UP1K valve driver diagram

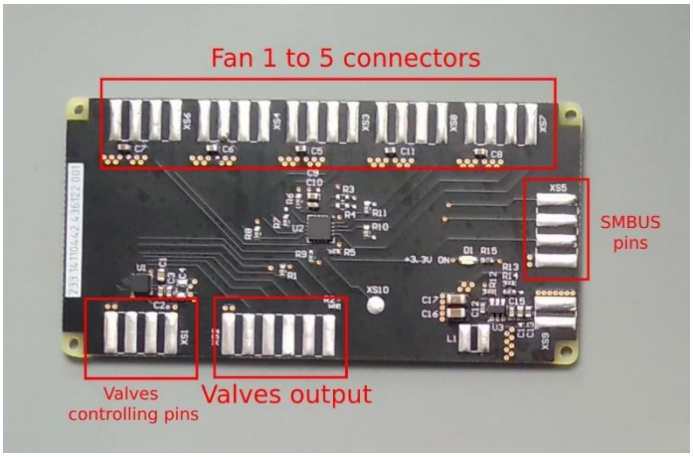

Figure 4.11. UP1K fan and valve driver board

4.2 UP200

The following table shows the specification of UP200 fuel cell generator.

Table 4.4. UP200 specifications [55]

Output voltage	12 V DC
Max continuous charging current at 12 V	17 A
Max continuous power output	200 W
Max charging power per day	400 A h
Electricity source	PEM fuel cell
Fuel	Compressed Hydrogen
Fuel consumption (at 200 W)	2.8 l/min
Batteries	13.2 V | 2.5 A h | LiFePO4
Size (L × W × H)	525 × 153 × 256 mm
Weight	8 kg
Operating temperature	−20 °C to +50 °C
User interface	On unit via Bluetooth to mobile or PC
Outlet and inlet ports	
Main outlet port	Max 17 A
External outlet port options	USB, high current port

The UP200 version, which is described in this thesis, is the second version of this product. Due to fixing errors and bugs on the previous version, the electronics of this version is designed the electronics boards as a single board. The electronics of this board is redesigned to fix the previous errors and improving the performance. Furthermore, in this version, the Bluetooth and GSM interfaces are added, and an RS-485 communication port is added for communicating and connecting multiple generators.

This section describes the different hardware parts and components of this product. It also shows the connections between different parts of the circuit.

4.2.1 General Structure

UP200 board consists of an ARM Cortex M4 MCU which controls Bluetooth and Display interfaces, fan driver and stack processes, and Power output and controlling the source of power. The mentioned parts are the most general classification of functionalities that the MCU must handle. This section explains the different hardware parts of UP200 thoroughly.

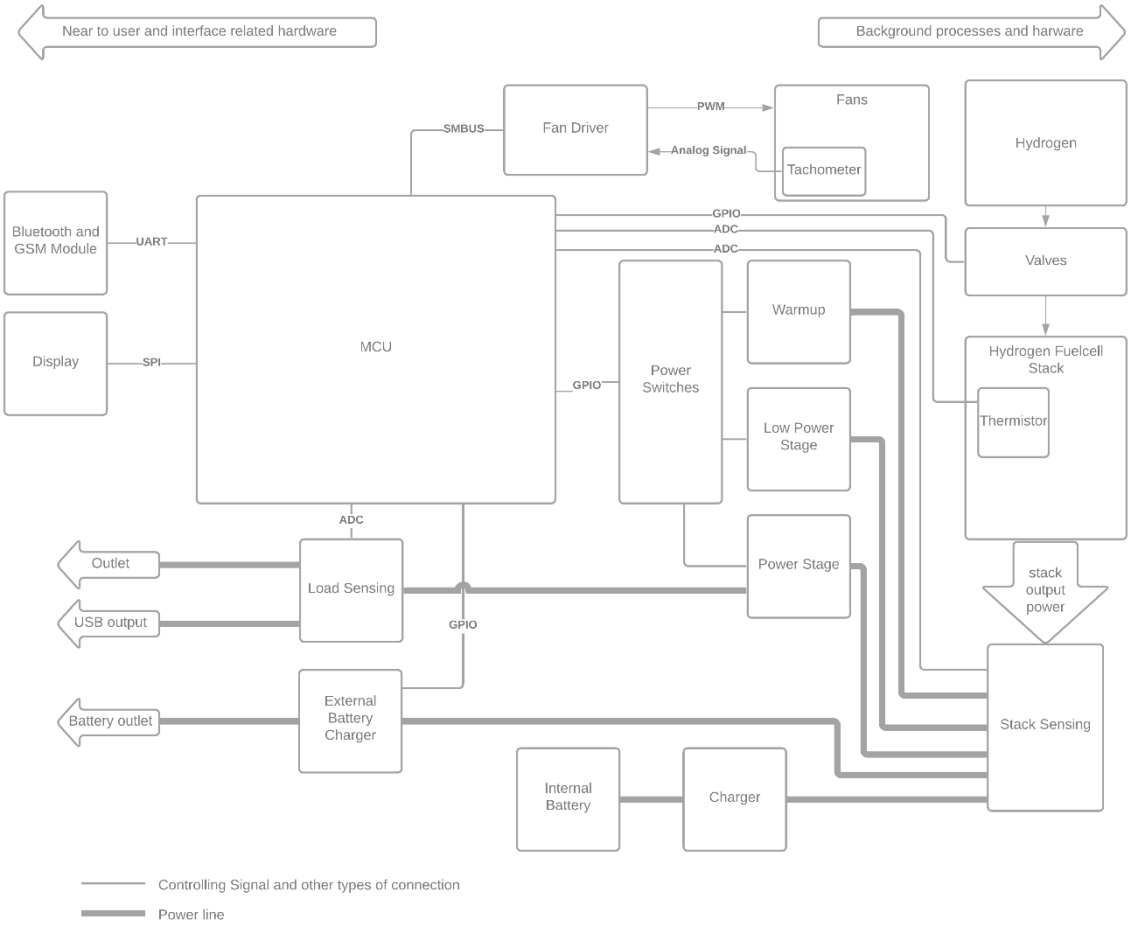

Figure 4.12. UP200 general structure

4.2.2 MCU

An STM32F427VIT6 [56] is used as the processor of this board. This microcontroller is an ARM microcontroller with a Cortex M4 core. It has 100 pins, 2 MB of flash memory and 260 KB of RAM. This microcontroller supports has 180 MHz of clock [56].

Although it does not have high processing and resource, it was manageable to implement all features by high-performance software. Also, as the pins are so limited in

this microcontroller, we tried to use all of them in the best way. The next chapter discusses the software and the management of outputs and inputs in detail.

4.2.3 Display

The display for this project is a one-colour OLED. It has 128×64 pixels resolution and also uses SSD1322 [57] IC as the driver. Due to the low power consumption of OLED screens Comparing to TFT LCD and Color OLEDs, it is a decent choice to place a one-colour OLED on a 200 W generator, which produces limited power and cannot handle high power for internal structure and hardware. It is connected to MCU using an SPI line as a slave. Also, as the driver does not transmit any data to the master, the slave to master or Master-IN Slave-Out (MISO) communication pin is disabled to have more place for the other functions of MCU.

The display shows the needed voltage, current and temperature values. It also should show errors in case of error happening. In chapter 6 explains what and how it shows the data on the screen.

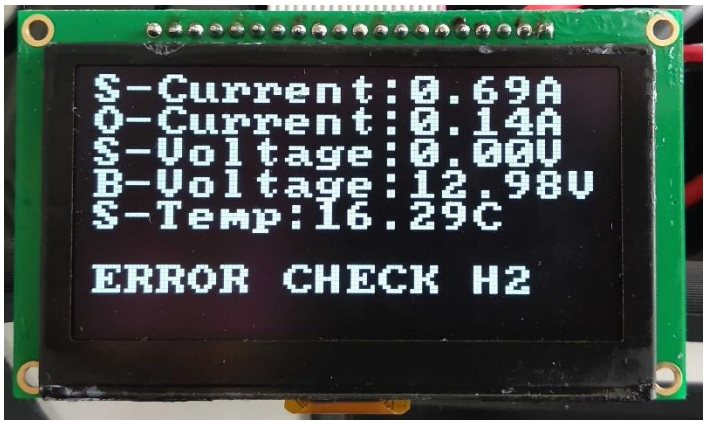

Figure 4.13. UP200 OLED screen

4.2.4 Bluetooth and GSM Module

The same as UP1K, it uses MC60 as the module for Bluetooth and GSM communications. This module is connected to MCU using UART. It also uses CTS and RTS pins for more reliability of UART communication. It also uses Ring pin to determine when GSM is ringing. This ringing can be an option for diagnosing and restarting the generator in some cases.

4.2.5 Power Switches

It changes the power source of electronics boards and outlet from the internal battery to the stack's output. This part includes an LTC1473 with multiple MOSFETs that connect the battery and DC input to the output. It manages the input source like the following diagram.

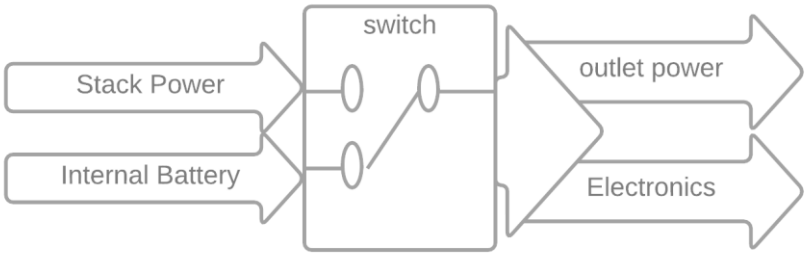

Figure 4.14. UP200 Power Switch diagram

The outputs are controlled by GPIOs. They also have a feedback pin to check the functionality of the switch. The power switch, by default, is connected to the internal battery. After stabilising the stack's output, the program changes the source of the electronics board and enables the output of the generator.

4.2.6 Warmup

The warmup circuit includes two lines of load that are resistors. The MCU controls the output of the stack to be connected to each of them or both of them. As described in chapter 3, the stack uses these loads to warm up and normalise the process and output. The resistance of the branches is 0.94 Ω and 1.47 Ω. The following diagram illustrates an overview of this circuit.

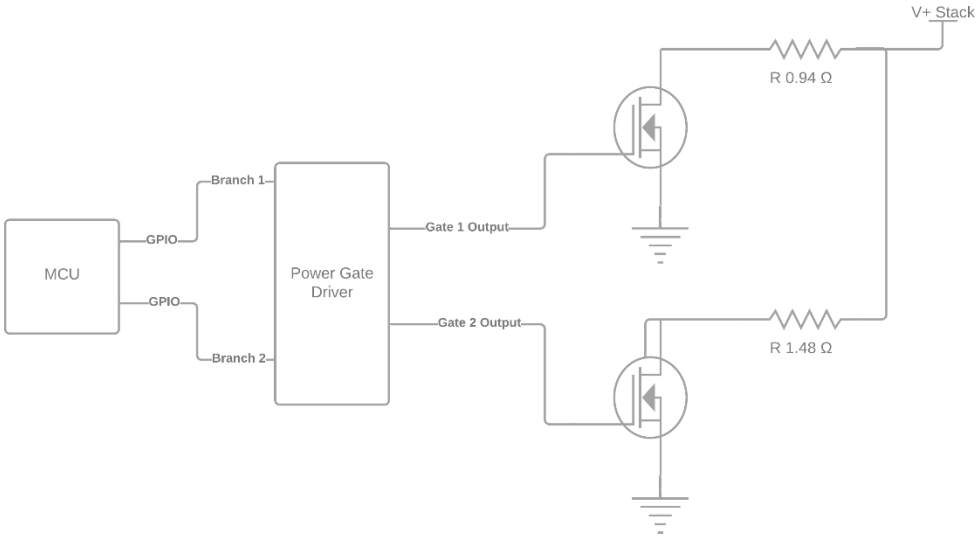

Figure 4.15. UP200 Warmup circuit diagram

For the warmup process, firstly, the larger resistor becomes enabled as a load and then the smaller branch, and after all, both of them becomes active, which means the load and current increase. All of the steps lasts until the voltage reaches more than 13, which means the stack power is stable respectively to that amount of load. The following chart shows the amount of load's resistance regarding time. The time of 1, 2, 3 are the number of times the stack voltage reaches 13 V, which leads to a change in the value of load by a control signal of MCU. The detailed process and experiments are described in the next chapter.

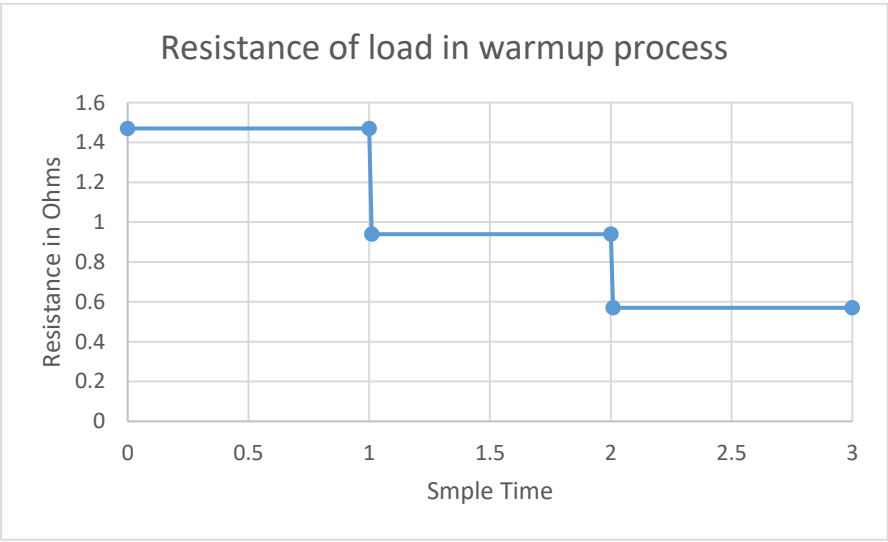

Figure 4.16. UP200 Warmup load chart

4.2.7 Low Power Stage

This part is a Buck-Boost DC-DC converter that regulates the voltage and supplies the power of internal hardware. In both cases of getting the power from the battery and or the stack, it sets the voltage to 12 V. The maximum output current of this DC-DC converter is 12 A, which is enough for managing the electronics structure and fans. This converter supports an input voltage range of 4 V to 36 V that covers the stack output voltage range. The following diagram shows which parts of the board are supplied by the low power stage circuit. Note that the lines in the diagram show the supply source of the electronic components and ICs, and it is not about output power to the outlet.

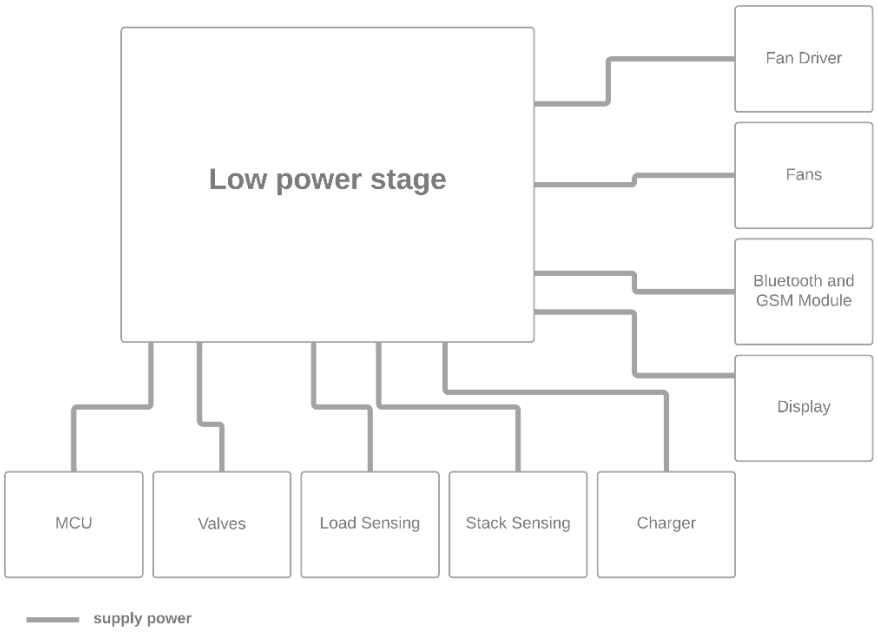

Figure 4.17. UP200 Low power stage diagram

4.2.8 Power Stage

This circuit is a DC-DC converter with 12 V and a maximum 19.36 A output. As the desired maximum output power of this product is 200 W, the maximum needed current is 16.7, which is less than the maximum current of this converter, and it can support it. It is used to regulate the voltage of the power outlet, which is the primary output port of the whole generator. It can support an input voltage range of 9 to 36 V, covering the

whole range of the stack voltage. For turning it on and off, the MCU uses a digital output pin. The MCU also has a digital input for checking the fault that happens on this circuit.

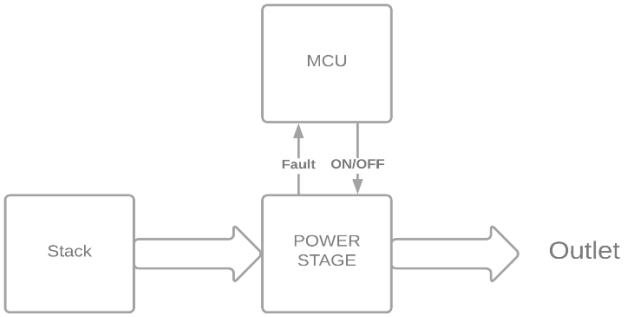

Figure 4.18. UP200 power stage diagram

4.2.9 Stack and Load Sensing

For monitoring the input and output power, there are two circuits to measure the voltage and current of each. These circuits use ACS770LCB, a Hall-effect-based current sensor [58]. It gives the output as an analogue voltage which measurable by ADC of MCU. Furthermore, both of them use a voltage divider by two resistors to reduce the voltage to be measurable by the MCU. The following sample circuit shows how this circuit works. The value of resistors is selected in relation to the range of input voltage and the voltage that MCU can measure.

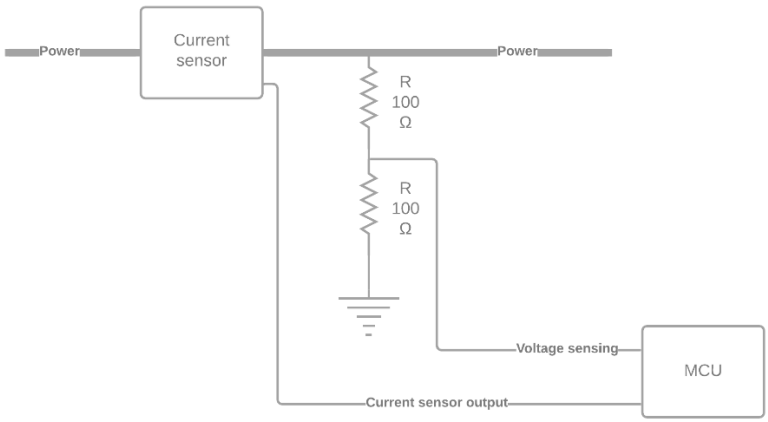

Figure 4.19. UP200 current and voltage sensing circuit diagram

4.2.10 Internal Battery Charger

The Hardware of UP200 needs power for booting up and starting the process of generating electricity from Hydrogen. As this battery must always be full-charged in order to start the operation of the stack, we need a charger that charges the battery while the fuel cell stack is in normal and generating mode. The charger is also designed to get input from an external charger in the cases that the battery is completely empty.

This part uses an LTC4020 [59] battery charger IC. It is controlled by a digital output pin to enable and disable the charging process. It has two direct digital output pins, which are enabled, in order, when the battery is charged more than 10% and is charged utterly. Also, another current sensor is used in connection with this charger to get more detailed value about the charging process and battery status. This IC is LTC2499 [60] that communicates with the MCU via I2C. It can measure the voltage, current, and temperature in one place. By accumulated charge value, it can also show the percentage of battery charge.

4.2.11 Fan Driver

The fan driver in this project is the same as the fan driver in UP1K product. It uses EMC2305 to drive the fans, with a difference that each of the two fan drivers drives four fans. It means there are four fans for cooling the stack and four fans for cooling the electronics. Both fan drivers are connected to the MCU using the same SMBUS but with different addresses. Item Fan and valve driver board in the previous section and Figure 4.9. UP1K fan drivers connection diagram show this driver in more detail.

5 Software Modules

For different components and parts used in the UP200 and UP1K projects, a new software module is created which can be used on different projects. This program design helps increasing portability, bug fixing and refactoring. This section explains all the modules that are used in the UP200 and UP1K projects.

5.1 Errors

Errors module is a file that defines an enumeration as a type for the result of other functions in the program. It leads to getting the same error and status types from all functions, which increase the readability of the code. All the statuses in this enumeration are started with *POWERUP_*, and all the letters are capital. The following part of code shows the existing type of errors. Each naming shows the application of the error or status value. For example, *POWERUP_OK* is always used when all the tasks in functions is done correctly and it returns this value to show it. As another instance, *POWERUP_ERROR_CONNECTION* occurs when the connection between MCU and another external component is lost (e.g. I2C connections).

```
enum powerup_error_e
{
  POWERUP_OK = 0,
  POWERUP_ERROR_UNKNOWN      = 1,
  POWERUP_ERROR_VALUE        = 2, // may be redundant
  POWERUP_ERROR_RANGE        = 3,
  POWERUP_ERROR_SEND         = 4,
  POWERUP_ERROR_READ         = 5,
  POWERUP_ERROR_BUSY         = 6,
  POWERUP_ERROR_CONNECTION   = 7,

  POWERUP_ERROR_PACKETLEN    = 8,
  POWERUP_ERROR_CHECKSUM     = 9,
  POWERUP_ERROR_PACKET       = 10,
  POWERUP_ERROR_PACKETTYPE   = 11,
  POWERUP_ERROR_PACKETHEADER = 12,

  POWERUP_ERROR_CONFIG       = 13,
  POWERUP_ERROR_TIMEOUT      = 14
};

typedef uint8_t powerup_error_t;
```

5.2 Config

The config module contains a header file with many definitions for different projects. These are the values needed in the program and depend mostly on hardware or the configurations needed to be done before compile. As a styling rule, again, all the letters of these constants are capital. For example, the following line of code shows the value of the resistor used on the board as the balance resistor of the fuel cell stack thermistor.

```
#define RESISTOR_THERMISTOR_BALANCE          100000.0f
```

As this module can be used in any other modules or programs, it must not include any other header file from the project.

5.3 PID

PID module includes a program that works as a single input single output PID controller system. PID is a closed-loop controller that calculates the input that must be set, which leads output to reach the desired value. It works based on proportional, integral, and derivative terms. It shows that it has three controllers included.

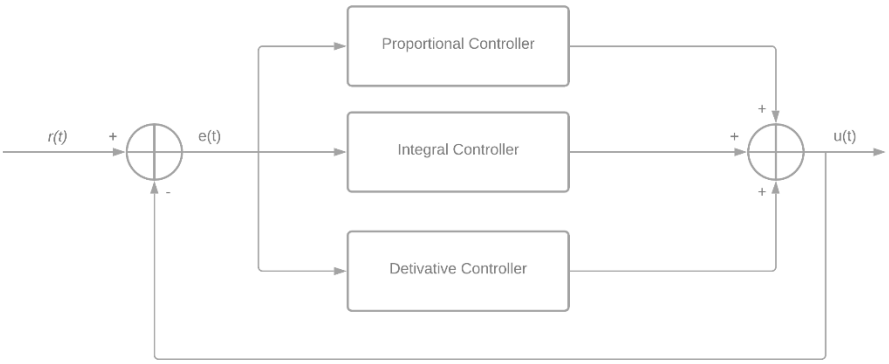

Figure 5.1 PID Controller diagram

In general, the following equation shows PID controller works.

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de}{dt} \quad 5.1)$$

Where u(t) is PID control variable

K_p is proportional gain

e(t)	is	error value
K_i	is	integral gain
K_d	is	derivative gain
de	is	change in error value
dt	is	change in time

For implementing this controller in the code, a struct is defined as a handle that includes the PID controller data. It helps to simulate object-oriented programming in C, which leads to the use of the same functions for different controllers. It means by defining different pid_handles; there will be different individual PID controllers with different characteristics. This struct contains *kp*, *ki*, and *kd,* which are gains of proportional, integral, and derivative terms. The variables *output_min* and *output_max that shows the output range of the controller. The output range is set 0 to 1 by default. The last_input, last_output and integral values are used to calculate the output for integral and derivative values. Moreover, in the end, iteration_time and last time values are used to fix the time between each calculation. The interval time between calculations must be a fixed value because the value of the derivative term is calculated based on the previous input value in a specific value of time.*

```
typedef struct {
    float kp;
    float ki;
    float kd;

    float output_min;
    float output_max;

    float last_output;
    float last_input;
    float integral;

    int iteration_time;
    int last_time;
} pid_handle;
```

This module has an initialise function that sets all variables in the *pid_handle*. It just gets a pointer to pid_handle and the gain constants, first input, and at the end, the interval time in millisecond and put them in the pid_handle.

After all, the pid_calculate function should be called in a loop. The loop must repeat the function in a frequency more than the frequency of interval_time for PID. This function has a float output. As input, it gets the pid_handle, input value, and a target (set-point) value. In the beginning, this function checks if it is the time to calculate the output or the interval time is not passed yet. If the time is not expired yet, then it returns the previous output value. Then it calculates the error value in proportional value. Then integral value, and the differential value. After all, it generates the final output by adding the product of proportional and its gain and integral value subtracted by differential value. Then it sets the last_time, last_input, and last_output values. In the end, it returns the output.

```
float pid_calculate(pid_handle *pid, float input, float target) {
    if (HAL_GetTick() < pid->last_time + pid->iteration_time)
        return pid->last_output;

    float proportional = target - input;
    pid->integral += proportional * pid->ki;
    if (pid->integral > pid->output_max) pid->integral = pid->output_max;
    if (pid->integral < pid->output_min) pid->integral = pid->output_min;

    float differential = (input-pid->last_input) * pid->kd;

    float output = pid->kp * proportional + pid->integral - differential;
    if (output > pid->output_max) output = pid->output_max;
    if (output < pid->output_min) output = pid->output_min;

    pid->last_input = input;
    pid->last_output = output;
    pid->last_time = HAL_GetTick();
    return output;
}
```

5.4 PTimer

As in many cases, the program needs a delay or needs to measure the time, and the general delay functions block the program, a module like *ptimer* is required. This program uses SysTick of the STM32 microcontroller, which is an interrupt and can be configured to happen once after a specific time. Then in this interrupt, a uint32_t can be increased by 1. In this case, the SysTick timer is set on 1 ms, which leads to having the number of ticks variable as the time the generator started working in milliseconds. This module stores the time when the timer is started and the amount of time that it

should wait. Then this module, by comparing the difference of current time and start time with the timeout value, shows if timeout happened.

This module has a handler to store the starting time and the timeout value. It is called *ptimer_t*. The *ptimer_set* starts the timer. It gets a pointer to the handler of ptimer, and the duration of that timer is valid in milliseconds.

The other function is *ptimer_timeout* that checks if the timer is expired and if the time that was set is passed. The only challenge in this part is that the tick counter variable is a 32-bit unsigned int, and it overflows after almost 50 days or 4294967295 ms. If a timer starts some milliseconds before the overflow, the now time value will be less than the start time value, and the difference will be a negative value. To prevent any malfunctioning, it checks if the current time is larger than the start time value or not. In each case, it uses the corresponding equation to calculate it.

```
/* checks if the overflow of the system tick */

if (current_time >= timer->start_time)
  {
    /* checks timeout */

    if (current_time - timer->start_time >= timer->duration)
      {
        return 1;
      }
  }
else
  {
    /* checks timeout */

    if (((UINT32_MAX - current_time) + timer->start_time)
        >= timer->duration)
      {
        return 1;
      }
  }
 return 0;
```

5.5 ADC Conversion

In the projects, DMA is used for ADC conversion. It converts analogue to digital as a non-blocking program that does not block the CPU's main task for this purpose. This module helps to start DMA for this purpose, convert values to voltage, and oversample

the values. This module also has a struct that is defined as a type for that handle of ADC. This is an implementation near to object-orient, which allows using this module for different instances of ADC and various channels. To use this module, firstly, the ADC channels must be activated on one of the device ADCs and add each of them to a rank in STM32Cube.

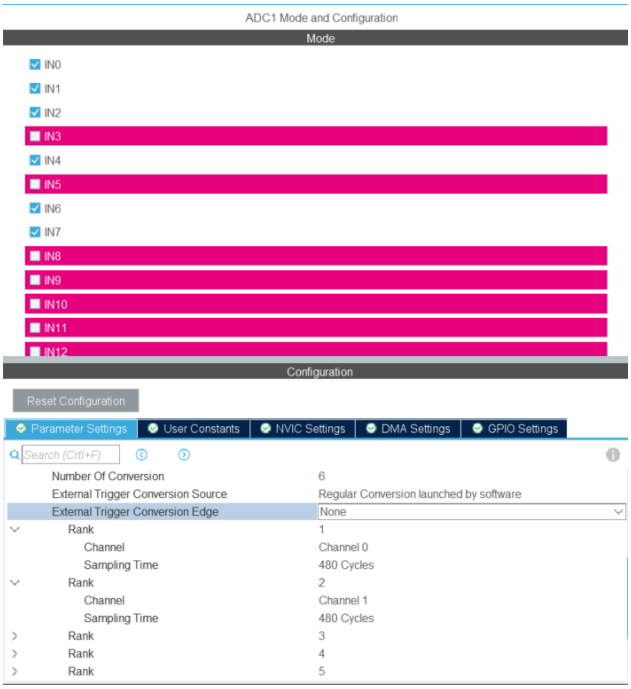

Figure 5.2 ADC activation

Moreover, the configuration for DMA and continuous conversion must be set in STM32Cube.

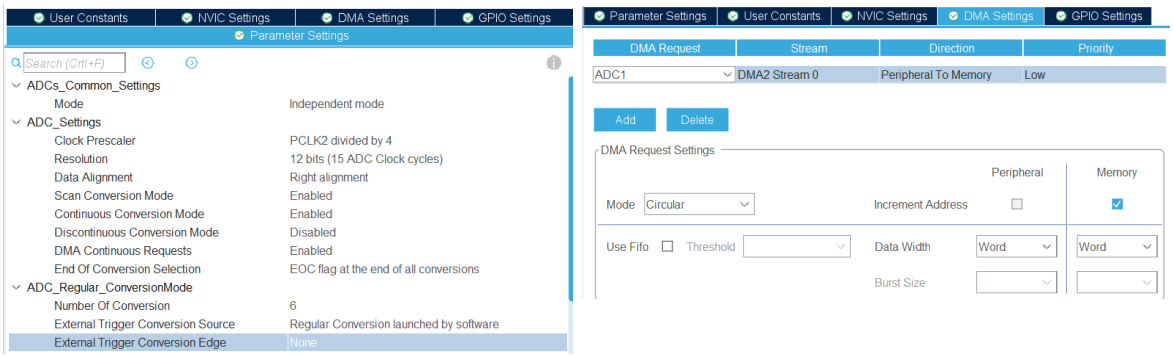

Figure 5.3 ADC DMA Configurations

Because we want to oversample the sampled data, multiple values of each channel are needed to calculate the average of them to decrease the noise values. For this purpose, the type of DMA is set on circular. It buffers each sample circularly in a part of the memory. It means if the buffer size is a product of the sample's number, active channel

number and size of each sample, each index plus the number of active channels will be the next sample of the same channel. The following Figure shows which index of the buffer is related to the value of which channel.

0	1	2	3	4	5	6	7	8	9	10	11
CH 0	CH 1	CH2	CH 0	CH 1	CH2	CH 0	CH 1	CH2	CH 0	CH 1	CH2

Figure 5.4 Memory layout for ADC buffer of 3 ADC channels and 4 oversamples

Also, as it is a circular buffer, it always can calculate a moving average of the values. The following figure shows how the circular buffer will be full. Each cell is a memory place, and the numbers in it show the order of income values to the buffer.

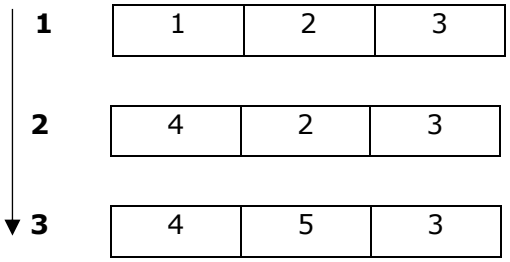

Figure 5.5 Sequence of values in a circular buffer

In the first place, the HAL ADC handler, number of active channels, number of oversamples, the maximum possible value of ADC should be set in the struct type of the ADC defined in this module. Then *ADC_Init* function allocates enough memory for the sample values and starts the DMA for storing values in the allocated memory.

```
adc_var->adc_values = malloc(adc_var->number_of_channels * adc_var->
                              oversamples * sizeof(uint16_t))

if (HAL_ADC_Start_DMA(adc_var->adc_instance, adc_var-> adc_values, adc_var-
>number_of_channels * adc_var->         oversamples) == HAL_OK)
    {
        return 1;
    }
```

ADC_GetValue function returns an average of all oversamples.

```
uint32_t ADC_GetValue(adc_t *adc_var, uint8_t channel)
```

```
{
    uint32_t value_sum = 0;
    for (int i = 0; i < adc_var->oversamples; i++)
    {
        value_sum += adc_var->adc_values[channel + (i * adc_var-
>number_of_channels)];
    }
 return value_sum / adc_var->oversamples;
}
```

Also, another function is ADC_GetVoltageDiv that gives the value of a resistor voltage divider. It is used for reading the voltage and current in different parts of the projects.

```
    float adc_val   = ADC_GetValue(adc_var, channel);
    float result    = (((adc_val * 3.3f) / adc_var->
                          max_value) * (r1 + r2) / r2);
```

5.6 EMC2305

This module drives EMC2305 fan driver that is described in chapter 4. It supports two different types of communication, I2C and SMBUS. It can be selected by defining *EMC2305_USING_SMBUS* or *EMC2305_USING_I2C*. To have both options, preprocessing macros such as *#ifdef*, *#ifndef*, and *#if* are used. It leads to having just the related on the MCU instead of compiling both options.

It uses a struct to store the data of each instance. It includes a pointer to handle of the communication protocol, device address, and errors of the device.

The program uses interrupt for communication with the device. It helps to communicate with a non-blocking method. For sending, *emc2305_writebyte* function is used in which the program does not check if the message is sent entirely or not. It leads to the necessity to check the availability of the communication peripheral before the subsequent use of that peripheral. This function writes a byte value in an 8-bit register in the EMC2305 device. For this purpose, it gets a pointer to the emc2305_t handle, the register that must be filled with the new values and also the value that must be written in the register. Moreover, emc230x_readbyte function gets reads a value from the This is a static function used just in this module and cannot be used outside of the emc2305.c file.

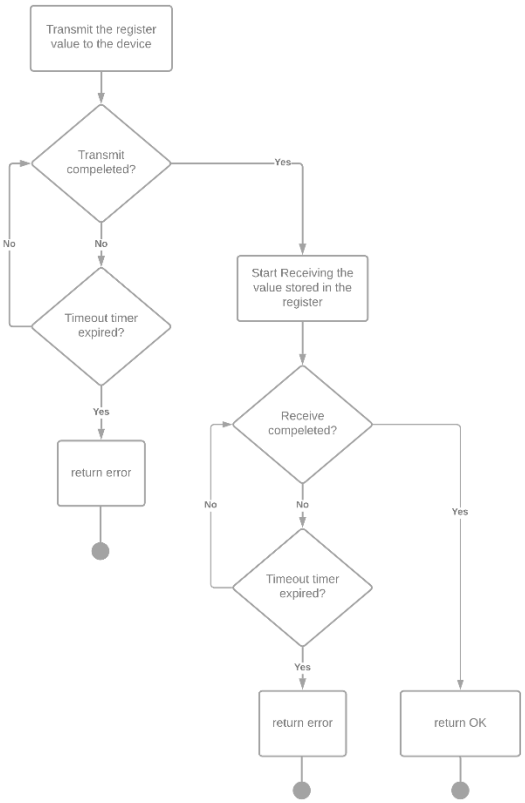

Figure 5.6 EMC2305 read byte function flowchart

There are several functions that just use transmitting. Most of them are used to configure the device and set the fans' specifications.

The *emc230x_enablesmbus* function enables the SMBUS protocol instead of I2C by setting a bit in CONF register.

The *emc230x_enablewatchdog* function enables the internal watchdog in the fan driver device. This again by enabling a bit in CONF register.

The *emc230x_config1* function configures the CONFIG1 register of EMC2305 for each fan. It contains updating interval time, the number of edges for the fan tachometer, the controlling algorithm, which can be automatic with internal PID or setting the tachometer value, and minimum tachometer range. In addition to a pointer to the emc230x_t value, this function gets the index number of the fan.

The *emc230x_setminspeed* function sets the minimum speed that driver sets. It can be from 0 to 255, which can be mapped from 0% to 100%. This function also sets this value for one fan as 0. It means the function gets the pointer to the handle and the fan index number.

The *emc230x_setvalidtach* sets the maximum valid value of the tachometer for each fan. It is the maximum value of the tachometer that can be read at full speed. This value can be calculated using the following equation. To calculate the value, the specifications of the fan must be known.

$$RPM_{max} = \frac{(n-1)}{poles} \times \frac{1}{COUNT_{max} \times \frac{1}{m}} \times f_{TACH} \times 60 \qquad [52](5.2)$$

Where *poles* is number of poles of the fan (typically 2)

f_{TACH} is The frequency of the clock (default = 32.768kHz)

i is number of edges measured (typically 5 for a 2 pole fan)

m is the multiplier defined by the RANGE bits (It is set in CONF1 register)

$COUNT_{max}$ is TACH maximum value (that is unknown here)

RPM_{max} is the maximum fan speed

The *emc230x_setspeed* sets the speed of a fan. It just sets the tachometer target high byte because we always need a minimum speed to be run and provide enough air for the stack. Also, in many cases, the driver ignores the low byte value. This function gets a pointer to EMC2305 handle, fan index, and speed rate from 0 to 1 float value as input. Moreover, this function should be called whenever the speed needs to be changed or for more reliability, frequently in the main loop with a delay in between.

Some functions use the receive function to get the status of the driver and fans. The most important function that does this is *emc230x_getstatus* that gets the current status of the driver. It shows if there is an error in driving the fans. It shows watchdog error and stall and drive error. If the error is related to drive or stall, this function calls *emc230x_getstallerror* or *emc230x_getdriveerror* to recognise which fan has the mentioned error exactly.

5.7 Fan Controller

This module is a higher level module to control the fans. It uses EMC2305 module to control the fans. This module uses a state machine pattern, and it has different states and sub-states for different situations. The following figure shows how it is located in the software layers.

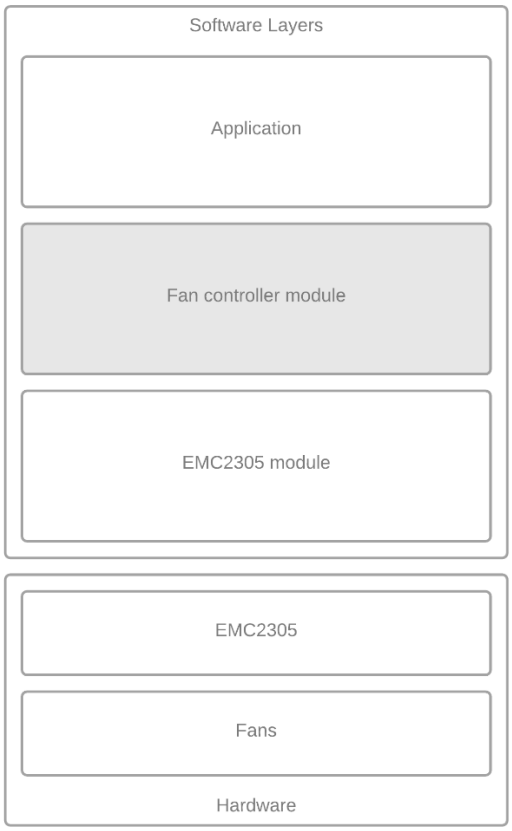

Figure 5.7 Fan controller module position in the software layers

This module has different states and sub-states, showing what the program should do and what task should be done to control the fan. It helps to use this program as a non-blocking software with using all features of the EMC2305. The following figure shows the states and sub-states of the module.

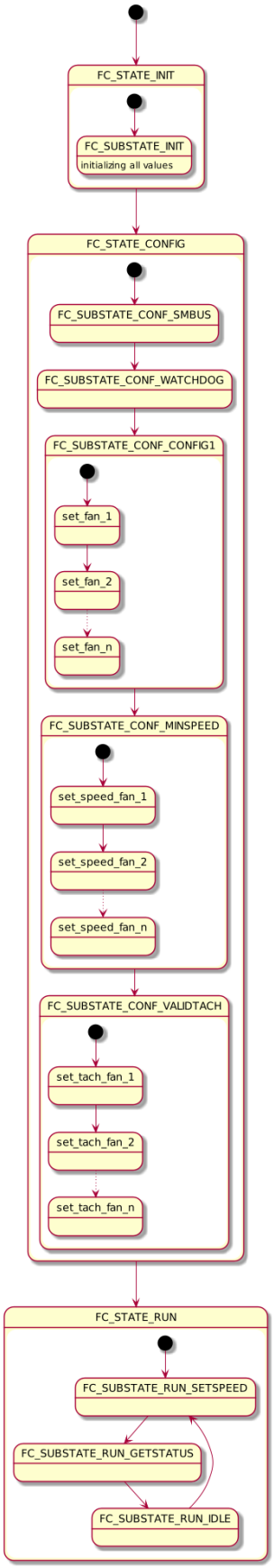

Figure 5.8 fan controller state chart

It has a struct as a handle variable with the name of the *fc_handle_t* that contains the pointer to pointers of communication peripherals, and EMC2305 handles, a timer for idle times, number of devices that are connected, the device addresses, application type of each fan, state, sub-state, current device index, and current fan index. The application of each fan can be not connected, stack, or electronics. It shows the fan is connected to which part of the generator.

To use this module, firstly, *fc_create* should be called to initialise a new *fc_handle_t.* It gets the number of devices that should be controlled then allocates enough memory for inner variables in the struct. After that, each communication protocol handle, device address, and then each fan application map should be set using the *fc_sethandle, fc_setaddress* and *fc_setmap*. After that, it goes to init state.

In the loop of the program, *fc_checkstate* function should be called. It checks the program is in which state and then calls its related function. The states and the related functions are described here.

FC_STATE_INIT initialises the handle of fan control. It sets the initial values of the variables in the handle struct. It has just one sub-state, *FC_SUBSTATE_INIT*, which means it can pass the *fc_init* function by the first call. In the end, it sets the state and sub-state of the handle on FC_STATE_CONFIG and FC_SUBSTATE_CONF_SMBUS to change the state and continue the procedure.

FC_STATE_CONFIG has 5 sub-states. It passes one sub-state by one time call of the *fc_config* function if it is possible. First of all, it checks if the communication peripheral is available and free to use, then it goes to check the sub-states; otherwise, it returns *POWERUP_ERROR_BUSY* value. The first sub-state, *FC_SUBSTATE_CONF_SMBUS*, changes the communication protocol of the first EMC2305 device from I2C to SMBUS, then it changes the current device value in the *fc_handle_t to the next device*. It checks if the current device index is 0 again, which means all the devices are configured for this purpose, and it can go to the next sub-state. Each device is configured at a time of calling the function. Then it happens for FC_SUBSTATE_CONF_WATCHDOG sub-state and *emc230x_enablewatchdog* function*,* which enables the watchdog time of all devices one by one. The next configuration and sub-state is *FC_SUBSTATE_CONF_CONFIG1,* which uses *emc230x_config1* function but for each fan. It means it goes to the next fan index, and if the fan numbers are finished in a device, it goes to the next device. After all, it comes back to fan 0 and device 0. By checking both current fan and device indexes, it is possible to determine that the configuration CONFIG1 register is finished for all fans, and then it can go to the next sub-state. The sub-states *FC_SUBSTATE_CONF_VALIDTACH* and *FC_SUBSTATE_CONF_MINSPEED* call

emc230x_setvalidtach and *emc230x_setminspeed* functions to for each fan separately. Then it sets the current state and sub-state values on *FC_STATE_RUN* and *FC_SUBSTATE_RUN_SETSPEED* to change the functionality in the next call of *fc_checkstate.*

FC_STATE_RUN state has 3 substates. The *fc_checkstate* function calls *fc_run* when the program is in this state. This state should happen in the main loop of the program. The first sub-state of this state is *FC_SUBSTATE_RUN_SETSPEED* which set one of the fans' speed by *emc230x_setspeed* each time and then changes the current_fan index value to the next fan. It happens until all the fan speeds are set. In the next step, it goes to the next sub-state, which is *FC_SUBSTATE_RUN_GETSTATUS*. In this sub-state, the status and error values. It happens for one device at a time, and then the next device is chosen. After that, the devices send back the values; it sets the idle timer of the devices with idle timeout value and the sub-state on *FC_SUBSTATE_RUN_IDLE.* In this sub-state, the function checks if the timer is expired or not. If the timer is not expired, it is not time to change the state, and the function returns POWERUP_OK. On the other hand, if the timer expires, it changes the sub-state to FC_SUBSTATE_RUN_SETSPEED again. It leads to create a loop of changing the sub-states in a non-blocking manner.

5.8 SSD1322

As mentioned in chapter 4, SSD1322 is the driver for OLED used as a display for UP200. This module is a software driver for controlling this hardware. It handles an SPI peripheral to write data on the screen. Also, it has to control a digital output pin to determine that the transmitted bytes are data or command. Data is the values that are set on the pixels of the OLED or the settings values. Command is the value that shows where the data goes and which setting or pixel is changed by the data. It works in blocking mode, but as the MCU works as a master, it does not stop the MCU for a long time and sends the data anyways. Having no feedback from the display leads to the issue of recognising a problem in the communication with the display, or the display is not initialised and needs a reset. To reduce these problems effects, the reset pin is also connected to the MCU, and it resets the driver every 1 minute to make sure that the display is initialised correctly.

This module includes one more header file in addition to its core files. This file stores all ASCII characters as 8x8 pixels area values. It means each character on display needs 8x8 pixels space. These values, later on, are used in the functions for writing a text on

the OLED display. As each pixel is a binary value, a bit can be shown as a pixel, and a byte can show a row of pixels value. Because of this, each character is saved as an array of 8 bytes. For example, the following figure shows the "A" character pixels. The code for storing this value is shown here.

```
{ 0x0C, 0x1E, 0x33, 0x33, 0x3F, 0x33, 0x33, 0x00}
```

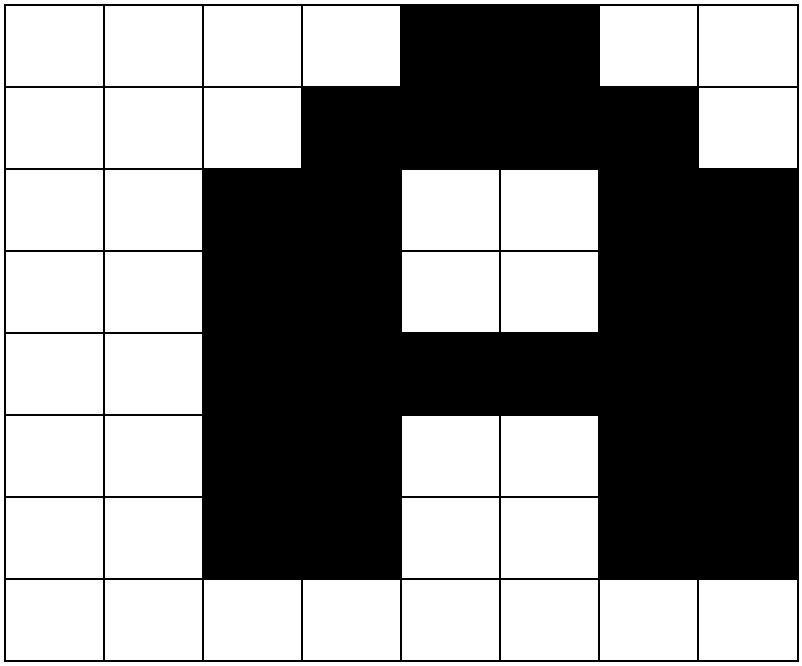

Figure 5.9 "A" character pixels in SSD1322 module

The module is created to work in a separate task of RTOS. However, the internal functions can be used in a program without RTOS. Also, the program uses DMA for SPI communication to make sure that it does not block the CPU process for sending the values to the driver. This module creates an array of bytes which saves the data of all frame pixels or, in other words, a frame to display. Every time this frame data changes in the memory, then it goes to the driver to be shown on the OLED.

All the writing happens by *ssd1322_puts* function that gets the start position of the text and the text itself as a string. Then it gets the values of pixels for that character of the string according to the mentioned table and updates the frame in the memory.

To use this function, firstly *ssd1322_control_start* function must be called as the RTOS task function. In the beginning, it sets two timers of *ssd1322_launch_timer* and *ssd1322_frame_update_timer*. The first one is to make sure that there is enough time without resetting the device, and the second one is for the interval between updating

the frames on the screen. Then it calls *ssd1322_init* function to initialise the module variables. It gets the pointer to the stack current, output current, stack voltage, output voltage, stack temperature, state, sub-state, special state values and set them in the current module variables and show them on the OLED. It also configures the registers of the device. After initialising, the control function checks if the pointers are valid. Otherwise, it shows that the display is offline on display. If it passes this step, it goes to a forever loop that decides what function should be done in the coming up situations. If initialising timer expired, it resets and initialises the display again. If the stack goes to a special state (like the overload for a long time), a message regarding that will be shown on the screen. If the timer of frame update is expired, it writes all the values and refreshes to display the texts on the screen and resets the update frame timer.

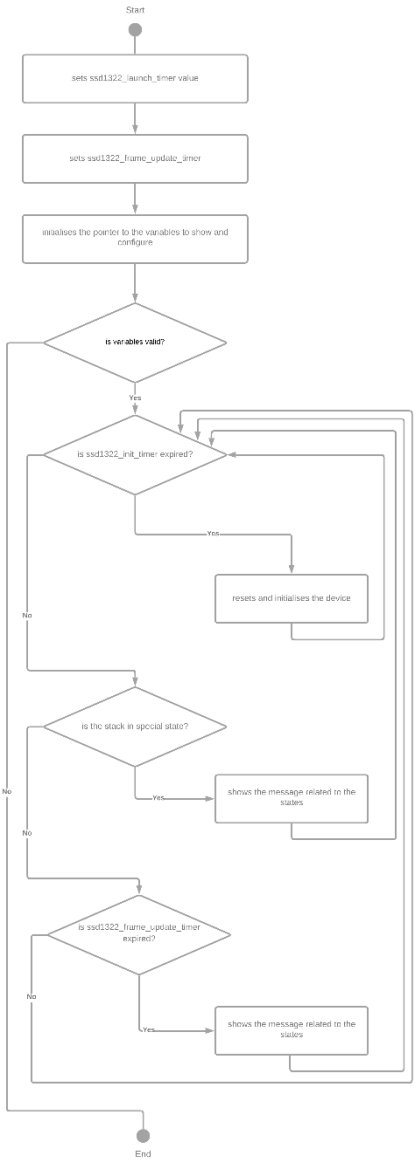

Figure 5.10 SSD1322 module flowchart

5.9 LTC2944

As is mentioned in section 4.2, LTC2844 is a battery gas gauge that measures voltage, current and temperature. It needs to monitor the battery charging circuit in UP200. This module uses the state machine pattern for controlling the device without blocking the CPU for a long time.

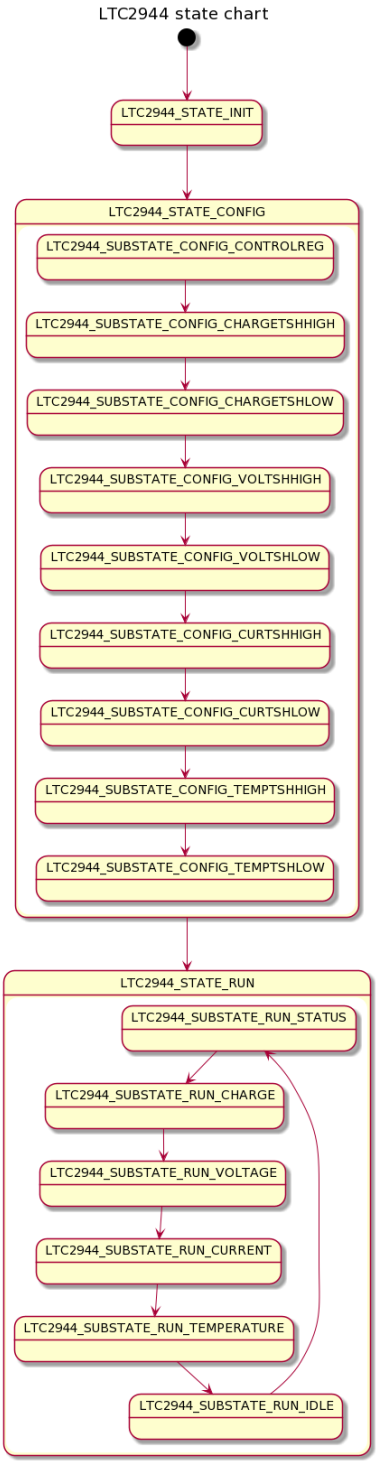

Figure 5.11 LTC2944 module state chart

The program also has another state for reading the values. It is tried to keep the program as non-blocking as possible. For this purpose, it uses DMA for I2C to communicate with the device. In this case, it uses a state machine for reading the data from the device. It has 4 states.

```
enum ltc2944_receiving_state_e
{
  LTC2944_RSTATE_IDLE,
  LTC2944_RSTATE_TRANSMIT,
  LTC2944_RSTATE_RECEIVE,
  LTC2944_RSTATE_FINISHED
};
```

Using this method, by calling *ltc2944_readbyte* function, it checks the current state of reading. If it is not in *LTC2944_RSTATE_IDLE*, it returns *POWERUP_OK*. Otherwise, it sends the register value using DMA and changes the state to *LTC2944_RSTATE_TRANSMIT.* Every time this function is called and the program does not complete the transmit, it returns POWERUP_ERROR_BUSY. After that, it starts receiving the data using DMA and changes the state to LTC2944_RSTATE_RECEIVE. Again returning POWERUP_ERROR_BUSY happens while it is in this state and the receiving is not completed. After all, when the receive buffer is filled with the received data, it goes to LTC2944_RSTATE_FINISHED. In this case, the reading is finished, and the higher-level program should use the received value and reset the reading state to *LTC2944_RSTATE_IDLE* to be available for the later receivings. All of the methods are described to show that the reading function should be called repeatedly in the loop of the program, and by one time calling this function, the functionality might not be completed.

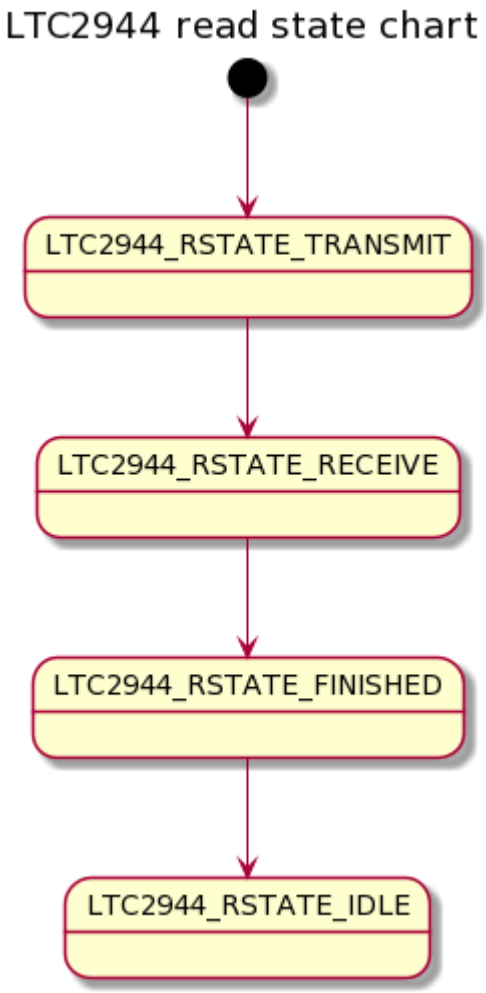

Figure 5.12 LTC2944 module read state chart

In general, the module has defined a struct, *ltc2944_t*, as a type to save all data of each instance of this type of device. This struct includes a pointer to the I2C handle for communication, the device address, a buffer for the received values, state, sub-state and receiving state values. Moreover, it stores some other values to give them to the other parts of the program. The values are status, charge value, voltage, current, and temperature values. To use this module, it is required to create a new *ltc2944_t* and set the pointer to I2C and the device address of it correctly. Afterwards, calling *ltc2944_update* function in a loop will organise everything like Figure 5.11. if the program is in the state of *LTC2944_STATE_INIT*, this function calls ltc2944_init, which initialises the values inside the *ltc2944_t* and sets the state and sub-state on *LTC2944_STATE_CONFIG* and *LTC2944_SUBSTATE_CONFIG_CONTROLREG*. The config function sets all the required registers and the limits for critical situations in the device. After finishing configuration, it sets the state value on *LTC2944_STATE_RUN* and sub-state on *LTC2944_SUBSTATE_RUN_STATUS*. In the *LTC2944_STATE_RUN,* just sub-states changes, and it works like a loop. It gets the status, charge value, voltage, current

and temperature data one by one. Then it goes to the idle sub-state (LTC2944_SUBSTATE_RUN_IDLE), which keeps the module inactive while it is time for the next reading of data. This helps not to use and block resources like DMA for a long time.

5.10 Shared Memory

As it is shown in section 4.1, the microcontroller for the UP1K product has two cores. These cores need to communicate with each other. As they use the same RAM, the best way is to share a part of the memory in a managed way. In this case, a part of the memory is removed to be used as RAM for both cores. Then a struct with all needed variables is defined in this module.

```
struct sharedmemory_s
{

  /* The data from Cortex-M4 to Cortex-M7 */

  float stack_voltage;
  float output_voltage;
  float stack_current;
  float output_current;
  float stack_temprature;
  float battery_voltage;
  uint8_t state;
  uint8_t substate;
  uint8_t special_state;
  uint8_t supply_valve;
  uint8_t purge_valve;
  float fan;
  uint8_t converter_Mode;

  /* The data from Cortex-M4 to Cortex-M7 */

};
```

In the next step, the address of reserved memory is set to an instance of this struct that is available globally. It helps to use this header file anywhere and access the same part of memory.

```
static volatile struct sharedmemory_s *const sharedmemory_instance =
    (struct sharedmemory_s*) SHAREDMEMORY_ADDRESS;
```

The important point in using this module is to update each variable of the struct in only one part of the program. Otherwise, the access to the data by different parts of the program might cause a crash in the program.

5.11 COMMUNICATION PACKETS

This module generates packets of data efficiently and reliably to transmit. It can also parse the received data and retrieve the values inside. All the variable, function, and type names in this module start with cp_, an abbreviation for Communication Packet. Each packet in this program is an array of *uint8_t*. The following table shows the structure of packets in this module and the values inside the array.

Table 5.1 Structure of a communication packet with N+1 bytes length

Type	Header	Packet length	Content type	data	Checksum
Value	0xAA	N	*cp_contentType*	Values	XOR of all bytes including Header
Byte number	0	1	2	3 … (N-1)	N

Header is a value that shows the beginning of a packet. The value of it is AA in hexadecimal space. This value in binary is 10101010, which is the most difficult challenge for different communication protocols as it changes the level from 1 to 0 and reverses in each bit.

Packet length is just one byte in this module. It means the maximum size of the packet can be 256 bytes.

Content type byte shows the type of data that is in the packet. The different content types are defined in the enumeration that is named *cp_contentType*. It contains the types such as stack temperature, DC-DC, setting voltage, etc. A demo of it is shown below, but the types can be more than this.

```
enum cp_contentTypes
{
    cp_data_stackTemp       = 0x10,
    cp_data_dcdc            = 0x11,
    cp_data_setVoltage      = 0x12,
```

```
    cp_data_setLoad         = 0x13,
    cp_data_setDcdcStatus   = 0x14
};
```

Data in a packet is being located directly, byte by byte. For example, a float type data in STM32 and GCC compiler has 4 bytes length. Also, a float variable can be accessed byte by byte if the address of the float value be defined as *uint8_t* pointer. The following code shows how this conversion between an array of bytes and float happens.

```
float   floatVlue = 10.0f;
uint8_t *byteValue = malloc(4);

// first way of float to byte array conversion
byteValue[0] = ((uint8_t*)(&floatVlue))[0];
byteValue[1] = ((uint8_t*)(&floatVlue))[1];
byteValue[2] = ((uint8_t*)(&floatVlue))[2];
byteValue[3] = ((uint8_t*)(&floatVlue))[3];

// second way of float to byte array conversion
byteValue = (uint8_t*)(&floatVlue);

// conversion of byte arrat to float value
floatValue = *((float*)(byteValue))
```

The above method is applicable for all the value types that have more than one byte size. The one-byte values can be placed directly in one byte of the packet. For the packets that contain more than one value (e.g. two different floats), the values are set one after another. Also, as the STM32 microcontrollers' CPU works as little-endian, all the values stores as little-endian, and this program does not work on big-endian devices.

Checksum is a byte that shows if the combination of data in the packet is correct. For this purpose, a simple error checking is used, which uses XOR between all the bytes of the packet. It helps to check the packet's content with this value in a received packet and check if there is any data loss or communication problem. This must be keep in mind that it is the simplest error checking method, and it cannot consider the problem in disorder locations of bytes. It just checks the value of the bytes, not their positions.

After all, the module has several functions to create different packets. It has also *powerup_error_t cp_process(uint8_t *packet)* function which checks if the checksum is is correct and then assign each value in the packet to its respective variable. It happens by reviewing the content type byte in a switch-case statement and extracting the values according to each type.

5.12 UART COMMUNICATION

UART Communication module is a layer that controls the UART in a non-blocking method reliably. It uses both DMA and interrupts to control the transmission and to receive without missing values or announcement to the CPU. In an OSI model [61],it places in the third layer (Network), which collect and sends the data bytes. This module also solved the problem of receiving the values with unknown size. The following figure illustrates the sequence diagram of this module.

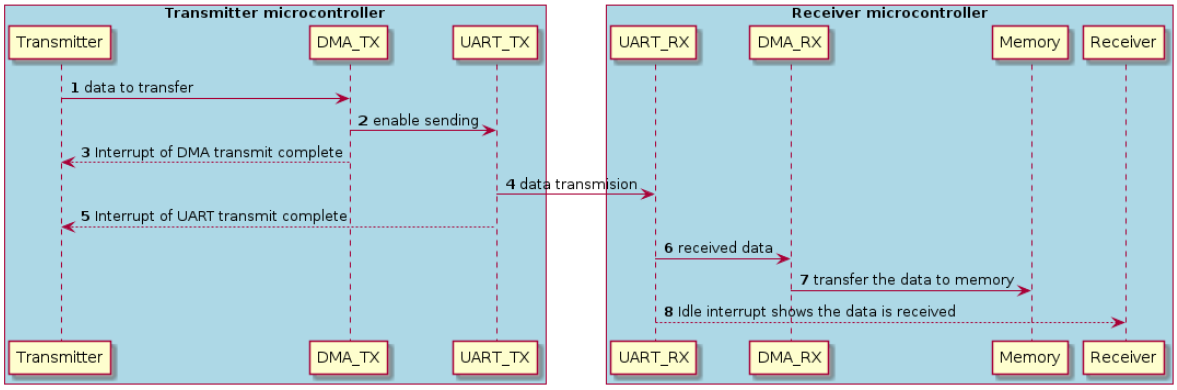

Figure 5.13 UART Communication module sequence diagram

As Figure 5.13 shows, two different microcontrollers are Transmitter, and another one is Receiver. The dotted lines in the diagram show the events that happen in the MCU automatically. These have two different flowcharts that are shown, and the functionality is described below.

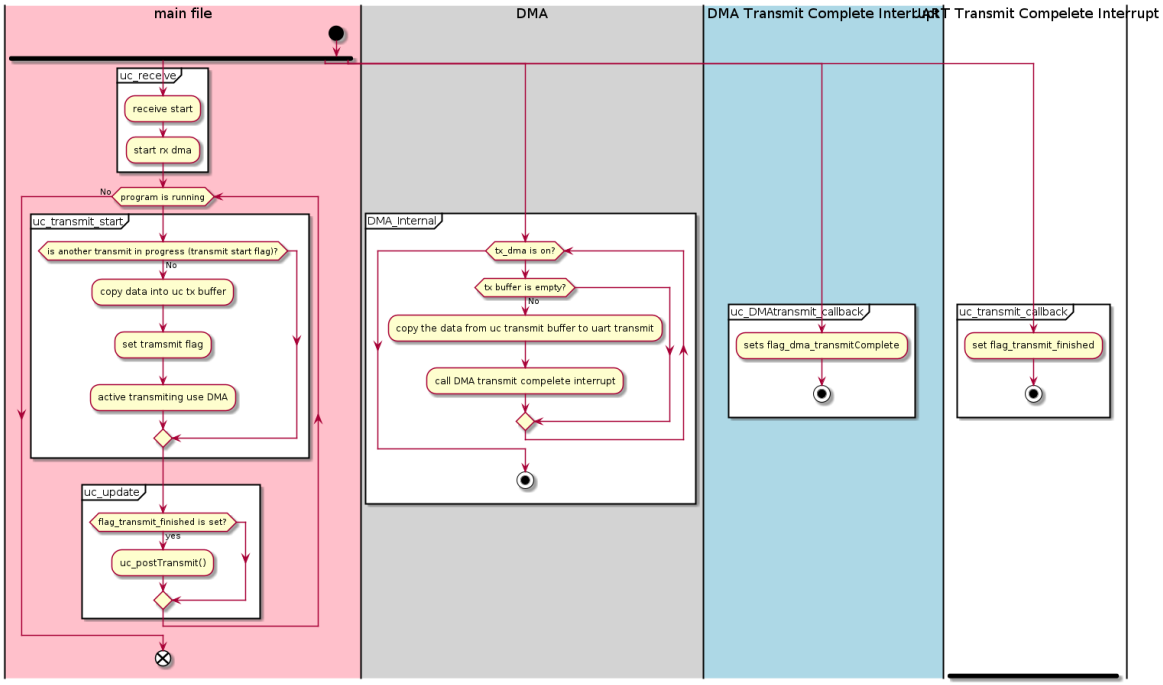

Figure 5.14 UART Communication transmission flowchart

By default, the program enables receiving, which leads to being ready for receiving data all the time. If *uc_transmit_start* function is called, it checks if another transmission is not happening. Then it copies all the data to be sent in a buffer, sets the transmission flag, and calls DMA transmit function to start transmission from buffer to UART by the DMA. At the same time, when the DMA enables transmission complete, interrupt when the transmission is finished. The callback function sets *flag_dma_transmitCompelete* variable to show the transmission from the memory buffer to the UART is completed. It does not indicate that the transmission by UART is also finished. For that purpose, a UART transmits complete is needed, which in the callback function of this interrupt flag_transmit_finished is set. After all of these events and changes, *uc_update* function, which is called repeatedly in the main loop of the program, recognises that the MCU sent data successfully and resets all the flags by calling uc_postTransmit function. In Figure 5.14, the DMA and interrupt callback functions are shown concurrently because the other parts of the microcontroller check them and do not block the CPU process. Moreover, two interrupts are used to check the completion of the task for more reliability and preventing malfunctionality, especially in communication with UART to RS485 converter.

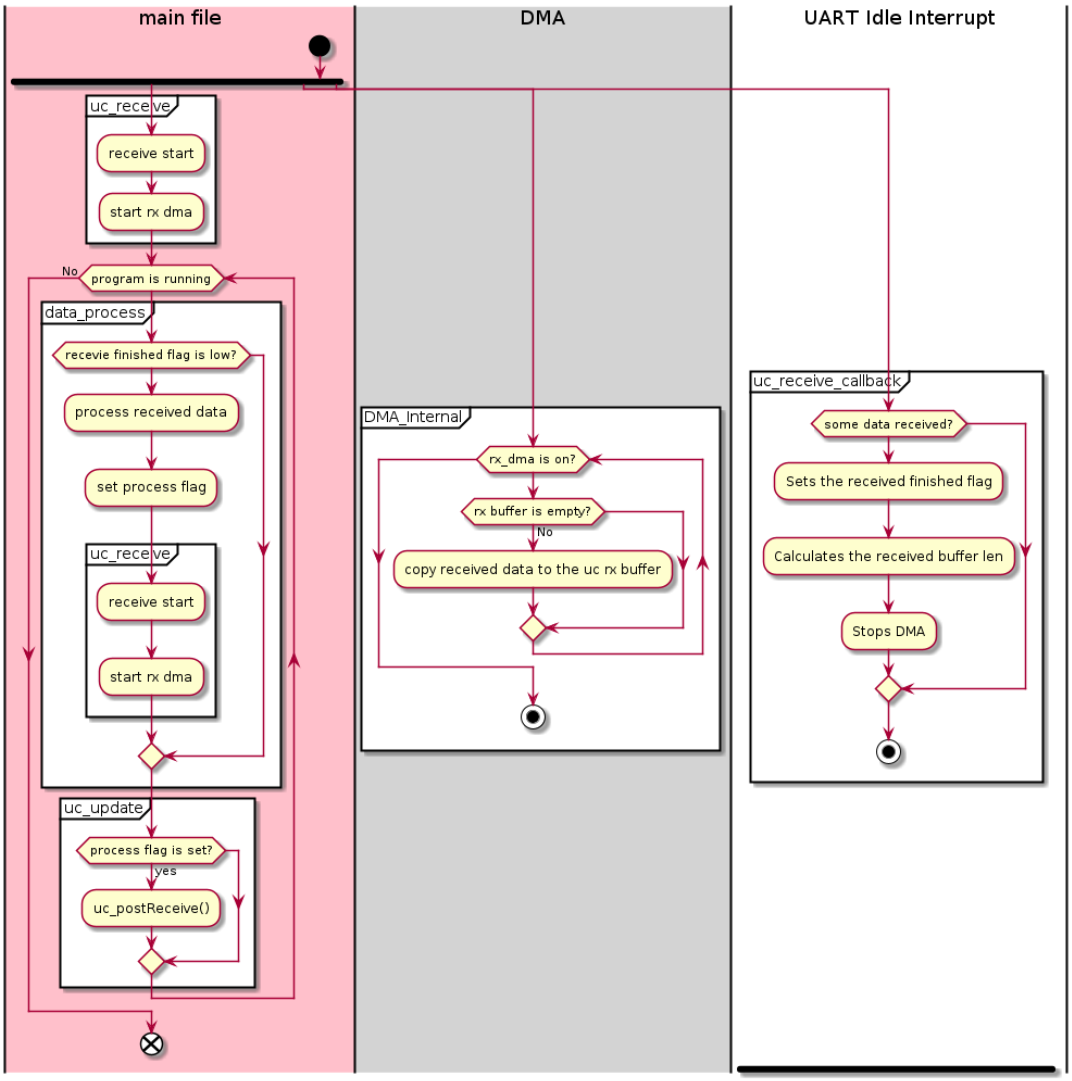

Figure 5.15 UART Communication receiving flowchart

As it is mentioned in the previous part, receiving starts before the main loop by the data_process function. In the process, it first checks if the flag is set, and if it is correct, then it goes to process the received data and set the process flag. The data_process function also calls *uc_receive* function to start receiving again at the end of the part. Like the previous process, *uc_update* function must be called in the main loop and checked frequently. The uc_update function checks if the received data is processed, and if it is true, it calls uc_postReceive function, which reset all the flags related to receiving data and resumes the data DMA for UART. The DMA process shown in this flow chart is an internal procedure that checks if there is any data from UART to copy them to the buffer in the memory. In receiving, the DMA transmit complete interrupt is not used because it works reliable enough with just one interrupt. The interrupt used in this procedure is the UART interrupt. For this purpose, a callback function called uc_receive_callback must be called in the interrupt handler.

In general, this module is used in all the cases that UART is needed. For example, the MC60 module uses this function to send and receive the data.

5.13 BUCK-BOOST

The buck-boost converter is one of the most critical parts of the program in all generators. Any fault in this module might damage the consumer device. This module controls the output voltage by the feedback of output voltage and setting voltage using a PID controller. The following diagram shows the system model of this module.

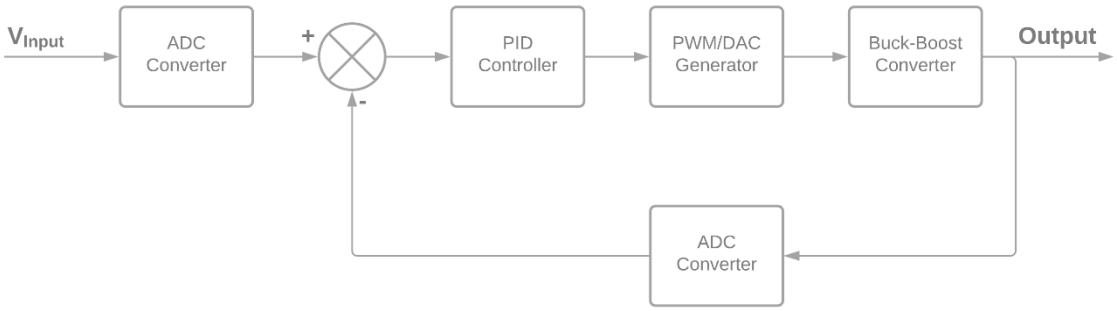

Figure 5.16 Buck-Boost converter control system diagram

In this system, the input voltage is used as input of the system, and the output voltage is the output and feedback of the system. Both of them are decoded by ADC to be readable by the MCU. The ADC conversion and PID modules are used in the module.

According to 4, this module is needed just in UP1K because it has adjustable voltage output. This part's hardware had two versions until the day of writing the thesis. The principal difference between the versions is the method of controlling the output voltage. In the first version, the output could be controlled by two pairs of PWM channels, and in the second version, it can be controlled by DAC and analogue voltage. Since the control algorithm is almost the same for both controlling signals, both functionality is supported in this module.

To create a PWM signal, it uses High-Resolution Timers (HRTIM). As these timers have advanced configurations like dead-time and event config and the synchronisation between different channels, it generates highly accurate signals that are synchronised in time and reverse in voltage level. Also, the dead-time configuration prevents the conflict of having two high signals at the same time. Different events can trigger HRTIM outputs. For this purpose, the "Set Sources" and "Reset Sources" of an output must be

defined. Each HRTIM channel has two outputs. As a pair of sync PWMs is needed in this project, both channel's outputs are used. The following pictures show the configuration of both outputs for a channel that are used in this module. In this case, the timer period sets the output1, and the compare value resets it. The dead-time value sets the output2, which means it is set in a little while after setting output1, it will be set. And again, the timer compare value resets it, and it happens as much as dead time value sooner. In this program, the timer's *compare 1* value is the value that should be changed to change the duty cycle of the signal.

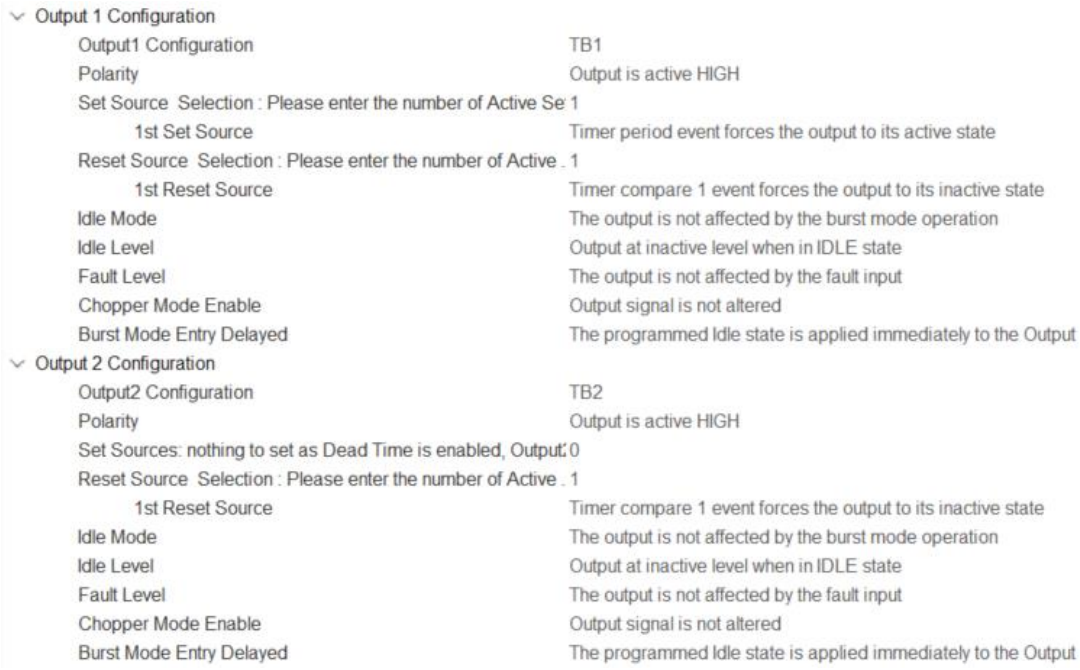

Figure 5.17 HRTIM outputs configuration in STM32Cube

Furthermore, the dead-time setting is shown below, which is possible to be set for rising and/or falling edges. In this program, both of them are set to have a centre-aligned signal.

Figure 5.18 HRTIM dead-time configurations in STM32Cube

The fundamental and frequency of all of these signals are configured in the general configuration of HRTIM. For example, in this case, the frequency is set to be 100 kHz. In this case, the period will be 2000, which means the duty cycle can be controlled by the precision of $\frac{1}{2000}$ of $\frac{1}{200\,0000}$ second.

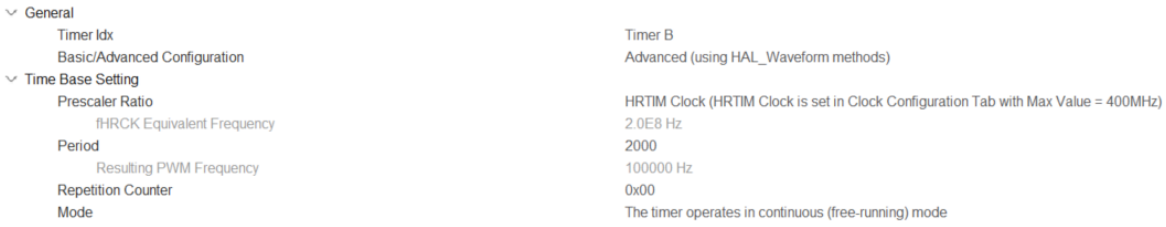

Figure 5.19 HRTIM General configuration in STM32Cube

The final signal output is shown in Figure 4.7.

According to the description of this item, in the new version, a digital to analogue converter is used to control the output voltage. For this purpose, an external DAC is used that is controlled by SPI. For changing the value of the analogue signal for controlling this converter, an SPI packet is sent in a non-blocking mode.

As the standard modules in the projects, it has a defined struct as a handle of buck-boost converter that stores the internal needed values like the input and output voltage values and a PID module handler. In general, the flowchart of this program is shown below. In the beginning, the *Converter_Init* function must be called. This function initialises all the variables and also the peripherals that are needed to be controlled. Then in the main loop of the program, the *Converter_Update* function must be called. In this case, firstly, the values are read and updated. In the next step, it checks if the output is on calculates and sets the value of DAC or HRTIM in the different versions of the hardware. It happens as fast as possible to correct the output voltage as soon as possible, which helps to prevent damaging the consumer value.

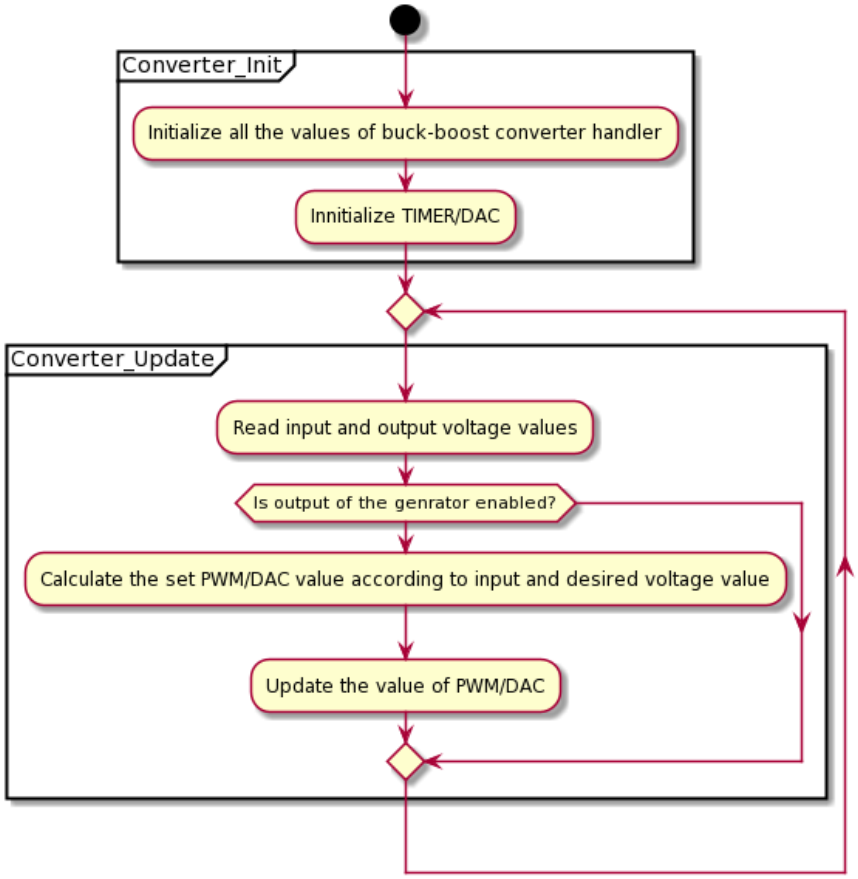

Figure 5.20 Buck-Boost module flowchart

5.14 MC60

The MC60 module communicates with the MC60 hardware that provides Bluetooth and GSM communication. The module supports SMS and Bluetooth data communication. It helps alert the critical situations to the user. For communication with the hardware, UART protocol with CTS and RTS control pins is used. To send and receive commands, a set of rules is generated by the manufacturer called AT Commands. All the communications use ASCII characters, and all the commands and responses have "AT+" characters at the beginning.

This module runs the UART Communication module for receiving the data. It sets the Bluetooth with a unique name to be visible for the devices. Whenever a device is connected, the data like a serial data packet is sent to the device. Sending data packets happens while the generator is working as a debugging method. Moreover, the connected device can send a command to the generator and configure it. All of the data transmissions are under the Communication Packet rules and methods.

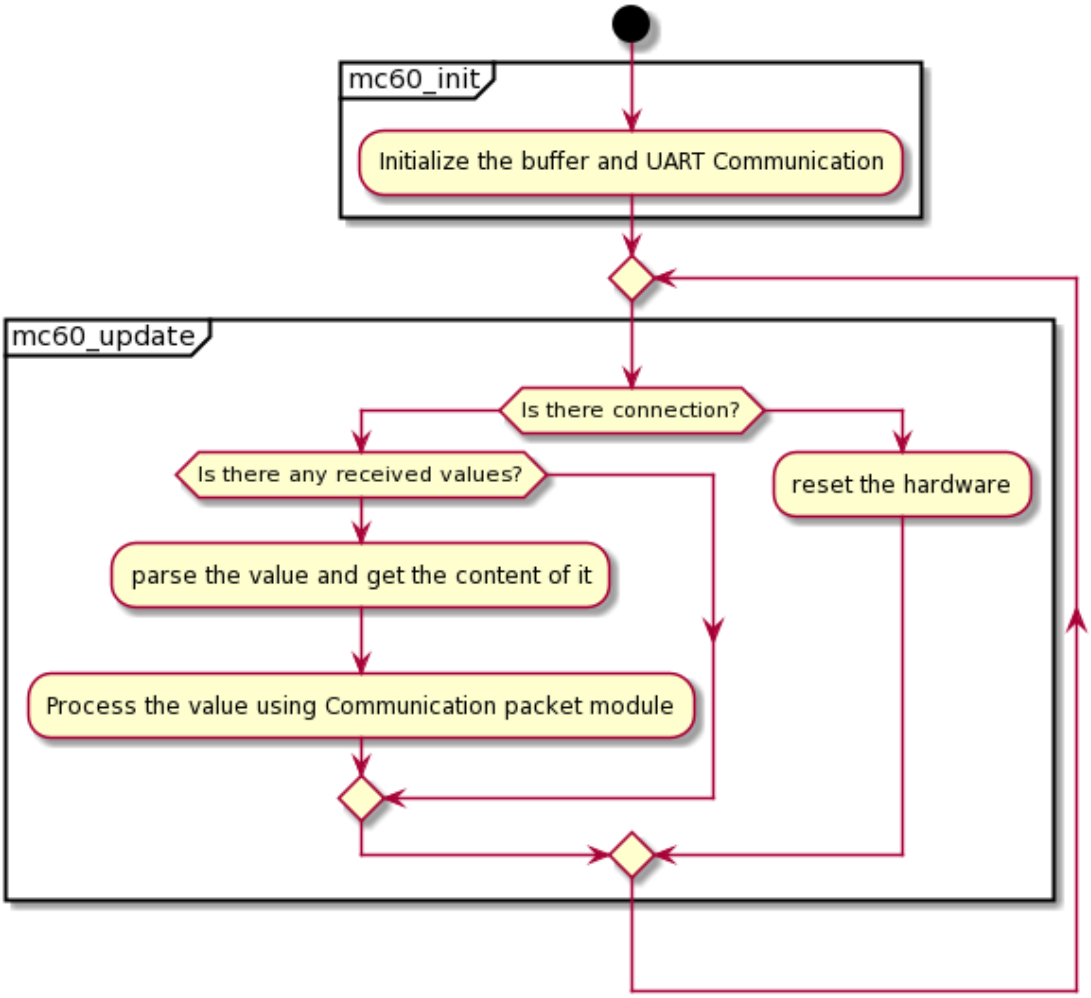

Figure 5.21 MC60 module flowchart

This module, first of all, initialises the buffer and empty it for receiving. Then in a loop, it checks if there is no connection, resets the hardware by a GPIO pin. Otherwise, it checks if any data is received by the UART. In that case, it removes the extra parts and gets just content in it. Then it processes the command and data by the Communication Packet module.

Furthermore, for sending the data, in any part of the program, *mc60_bt_send* function can be used, and it generates the packet and command, then transmit the data using UART Communication module.

5.15 STACK-CONTROL

Stack control module most of the other modules and controls the input, output and the fans. It monitors all the functionality of the stack and generator and executes commands according to the situation. This module uses a state machine pattern to handle different modes and conditions, and also each state has multiple sub-states. The name of states comes from the states that the hydrogen fuel cell stack has. These states are described in chapter 3 in details.

Standby state is the state that the generators start and end in. In this state, the generator does not have any production or consumption. The task in this state is to manage to close the valves and disable outputs.

Startup state initialises all the components and starts the fan with the maximum speed. Then it opens the purge valve on time for 400 ms and then goes to warmup state.

Warmup state increases the load that is connected to the stack slightly and waits for the corresponding voltage. Every 53 seconds, also it opens the purge valve. It is the state that ensures the stack is ready to give enough output current. It changes the state to Running state.

Running state enables the output and gives the output and at the same time measures the voltage and current and monitors them. In this state, the fan speed and purge values are set according to the input values like temperature, current, voltage. This state is like a loop, and while the generator does not have any problem or the user does not want to turn the generator off, it stays in this state.

Shutdown state is the state that the user commands to happen, and it can turns of the generator. It disables the output and waits to ensure that the stack's temperature is less than 30°C, which is the safe value for the stack. Then it disconnects all the power paths and keep the generator off.

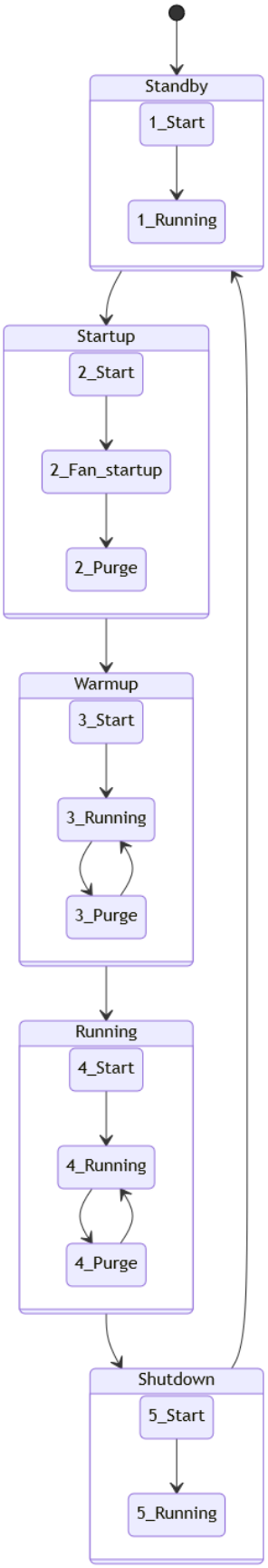

Figure 5.22 stack-control module state diagram

This module first initialises all the state variables, PID, timers, and needed modules to control the stack and outputs. Then it disables the outputs. In the loop, it checks if the conditions suppose to change the state of the program. In that case, it changes the program state of the program. After checking the need to change the state and setting the configuration if it is needed, it sets the output value using the PID calculation. This should happen all the time that the generator is working.

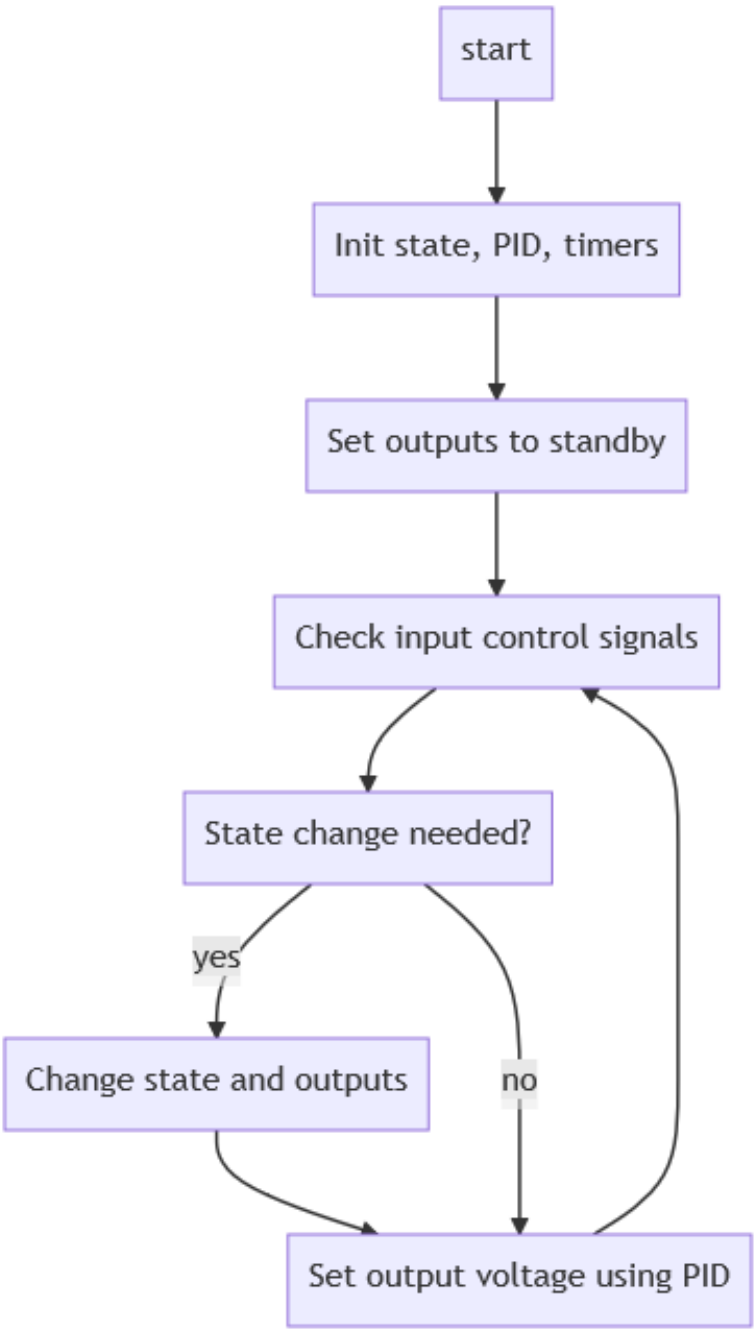

Figure 5.23 stack control module flowchart

This module also considers several special states that define the critical or error situation. These states are so essential to work on time for the safety of users and generators. The following table describes these situations and their conditions.

Table 5.2 Special states of the stack control module and the responses to them

State	Condition	Response
High temperature	The temperature of the stack be more than 65°C	1. Disabling output 2. Closing hydrogen valve 3. Set fan speed to maximum 4. Wait until the temperature drops to 50°C 5. Coming back to normal state
High Load	The consumer load needs more than 200 (for UP200) or 1k (for UP1K) watts of power	1. Disabling output 2. Enabling and checking if it happens after a 10s 3. Repeating step 2 for three times and if the problem still persists, keep the generator suspended until a restart
No Hydrogen	When there is no hydrogen in the tank connected to the stack and the voltage of the stack drops below a minimum value	1. Disable output 2. Wait for Hydrogen 3. Set the state of the stack control module on warmup to warmup the generator again
Low Battery Charge	When the battery charge value reaches less than 10%	1. Disabling output 2. Waiting for the battery to be charged up to 40% using the stack generated power 3. Come back to the normal state

6 CONTROL SOFTWARE

The main purpose of this thesis is to show how to implement the software, which is described in this chapter.

6.1 Development Toolchain

All the software is written in C/C++ language as most of the embedded systems software. The code is compiled using GCC for ARM targets or, in another word GNU ARM Embedded Toolchain. As the MCUs used in these projects are different types of STM32, STM32CubeIDE is used as the free and recommended IDE by the manufacturer. STM32CubeIDE [62].is built based on Eclipse IDE [63].

Also, as the first reversion, the low-level drivers are generated by STM32Cube and HAL drivers.

HAL driver is an abstraction layer that has APIs to control the peripherals of STM32 microcontrollers. It is free and open-source software that ST provides [64]. It helps to increase the development speed as there is no need to write code and test the low-level software and drivers. The disadvantage of this library is its low performance. As STM32 HAL driver is built to be portable and work with a wide range of microcontrollers and features, it does not work as efficiently as possible, but it is a good and fast option for the first iteration.

STM32Cube is a software tool that provides a graphical user interface for configuring and adding necessary HAL drivers, middlewares, and structure to the C/C++ project for STM32. It helps to configure and change the configuration easily and without conflict.

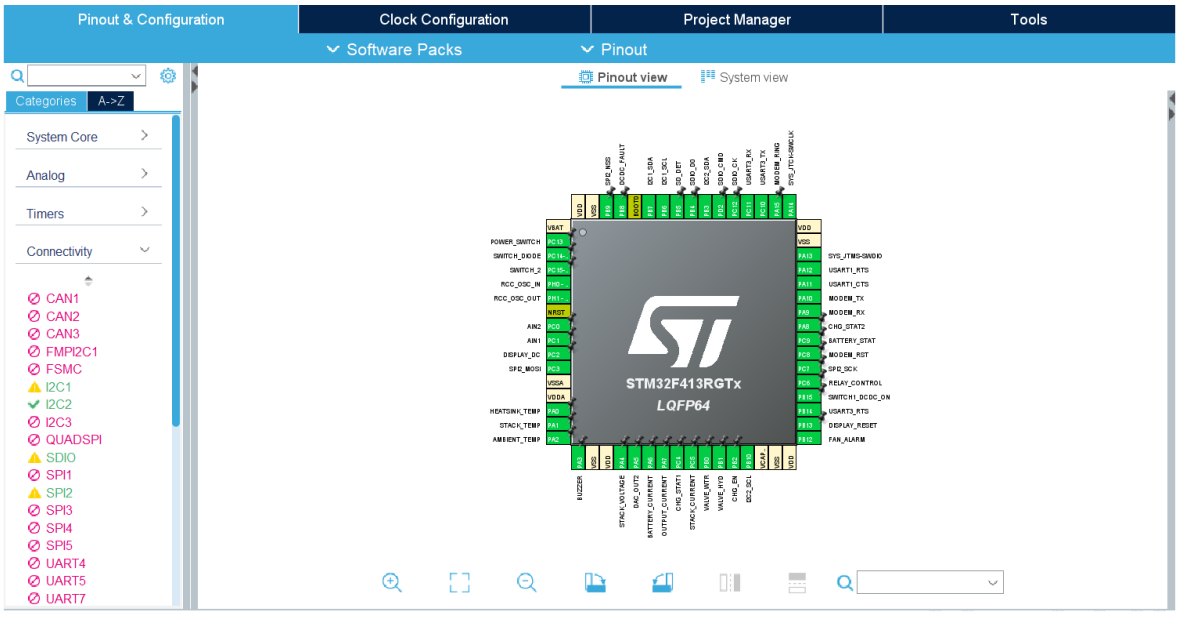

Figure 6.1 STM32Cube sample screenshot, a view of UP200's microcontroller

Middlewares that are added to the projects by STM32CubeIDE includes FreeRTOS, FATFS. The FreeRTOS that is used in the projects is described in the following sections. FATFS is software to support FAT16/32 file systems. It is also described in detail in the following sections.

6.2 Modules

The modules are the different components of the software that are created by the author to control different parts of the program or to work as a driver for a device. It is described in the previous chapter in detail.

6.3 Middleware

In this chapter, the middleware is used as software that cannot be categorised as modules because they are not written, especially for this project, and they are located in the middle layers of software. The middle layer here means the layers of software that are not directly communicating and control hardware and not in the application layer. In these projects, three middlewares are used, which are described in detail in this section.

6.3.1 FreeRTOS

Although it is tried to implement the software modules as non-block as possible, a Real-Time Operating System (RTOS) is used to ensure that all functionalities can work simultaneously. For this purpose, FreeRTOS is used. FreeRTOS is an open-source, free, and also efficient choice RTOS.

This RTOS is used for two different cases. The first case is using TouchGFX as the graphics library for UP1K, which is explained in details in this chapter. The other application of this operating system is to manage the stack controlling software. The stack controlling software for UP200 is implemented on its single Cortex M4 core. There are different tasks for different functionalities of the controlling stack. It is tried to use as few tasks as possible to increase the efficiency of the program. Also, some of the other libraries that are managed on the Cortex M7 core of UP1K use different tasks besides the TouchGFX library tasks.

UP200 tasks are wrapped up in three. The main task is the *stack-control* task. It includes the stack control, SSD1322, and fan controller modules. It is basically all the functions that directly affect the stack and output. This task's priority might increase in critical situations to ensure it finishes all the functionalities. The next task is called *MicroSD-Logging*. It ensures that it logs all the changes and concerns on the MicroSD memory card. It uses FATFS middleware that is defined later. *Debugging-UART* is the next task that sends the generator's status once in a while through UART and RS-485 converter and communicates with the user. It also helps in calibration and monitoring the situation. Moreover, this task handles the *MC60* module to communicate through Bluetooth and SMS. It repeatedly checks to receive and transmit buffers and for all kind of UART that there is in the program and process them as soon as possible.

UP1K tasks are three again, and it is implemented just on the Cortex M7. Moreover, ss it has a dual-core CPU, stack controlling related modules and libraries like stack control, buck-boost, fan control modules work on the Cortex M4 core, which can be considered a software thread without an RTOS. The most frequent task on Cortex M7 core is GUI task which handles the graphics processing and loading the frames on the LCD, which happens in the TouchGFX library. The other task runs MC60 and debugging UART port programs. The last task controls the synchronisation between the different generators. In that case, with one UART debug port or a Bluetooth connection, all the connected generators can be monitored and controlled.

The configuration of the FreeRTOS library is tried to be as efficient as possible. For this purpose, the type of tasks is static because there is no need to remove and create tasks dynamically. Furthermore, the tick rate is set at 1000 Hz, an efficient value for the scheduler. Finally, some of the features, like Semaphores, Mutexes, and queues, are enabled to be used for communication between different tasks without blocking the other tasks. For example, to show the values on the screen, the values are sent by queues to the graphics task. All the tasks are created initially, and no new task will be created while running the program.

6.3.2 FATFS

FATFS is an open-source and free library that supports FAT/exFAT File Systems. In other words, it can create and edit files in a way that regular operating systems such as Windows 10 can read the file. Therefore, it is very useful for logging data of UP200 on an SD Card. In UP200, all the data are written in a text file in a FAT32 volume.

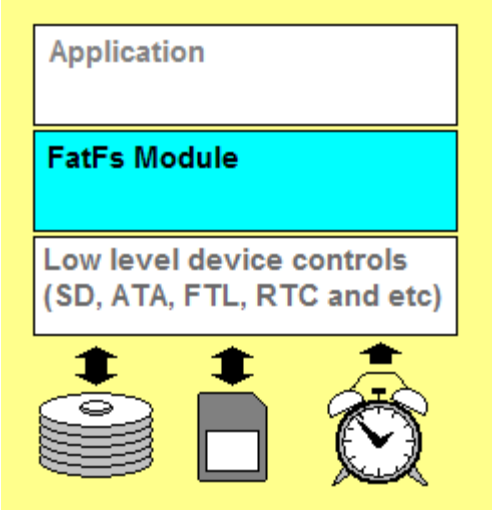

Figure 6.2 FATFS Software layer diagram [65]

In this case, the peripheral that manages the communication with the SD card is SDIO. It is set on 1-bit bandwidth. Also, it communicates by DMA, which decreases the load of the CPU significantly. Like previous parts of the program, it is tried to keep the program as less size and efficient as possible. The encoding of this middleware is set on UTF-8 as all the characters that are used are supported by it, and it has the least size.

6.3.3 TouchGFX

TouchGFX includes an optimised and hardware-accelerated graphics library for embedded systems called TouchGFX Engine, as well as a designer and a project generator PC software. TouchGFX Engine is created in C++, and it is the only part of the program that is in C++. It is used in UP1K and Cortex M7 core program to drive the LCD and generates the UI.

The colour format used in this project is RGB565 which is 16-bit depth for non-opaque images, and ARGB8888, with 32-bit depth for opaque images. It helps to reduce the size of frames by reducing the size of the background and many other non-transparent images.

The Frame buffer is so large compared to the other parts of the program. Moreover, using multiple frame buffers to achieve a higher frame rate and smooth animation makes the required buffer size multiple times larger[66]. For this purpose, an external RAM is used to provide enough memory. In this case, although access to the external memory is slower than the internal memory, there is much more place to put the file. Also, using an FMC peripheral as a communication method with external RAM and enabling CPU cache increase the speed of communication as much as possible. The FMC

peripheral and diagram of its connection to UP1K is described in chapter 4. In the Cortex M7 microcontrollers, two different caches are available, DCache and ICache. For this purpose, DCache, which fetch data from RAM, helps to increase the frame rate and speed of updating the frames.

The screens that are created for this project is shown here. The splash screen is a screen that shows the logo of the PowerUp technologies company and changes automatically to the main screen after 3 seconds.

Figure 6.3 Splash screen created by TouchGFX

The main screen shows the status of the generator in brief. It shows the time since the generator is started, the generator's state, its sub-state, special state, stack voltage, and temperature. These values are updated using different queues from a task to the GUI task.

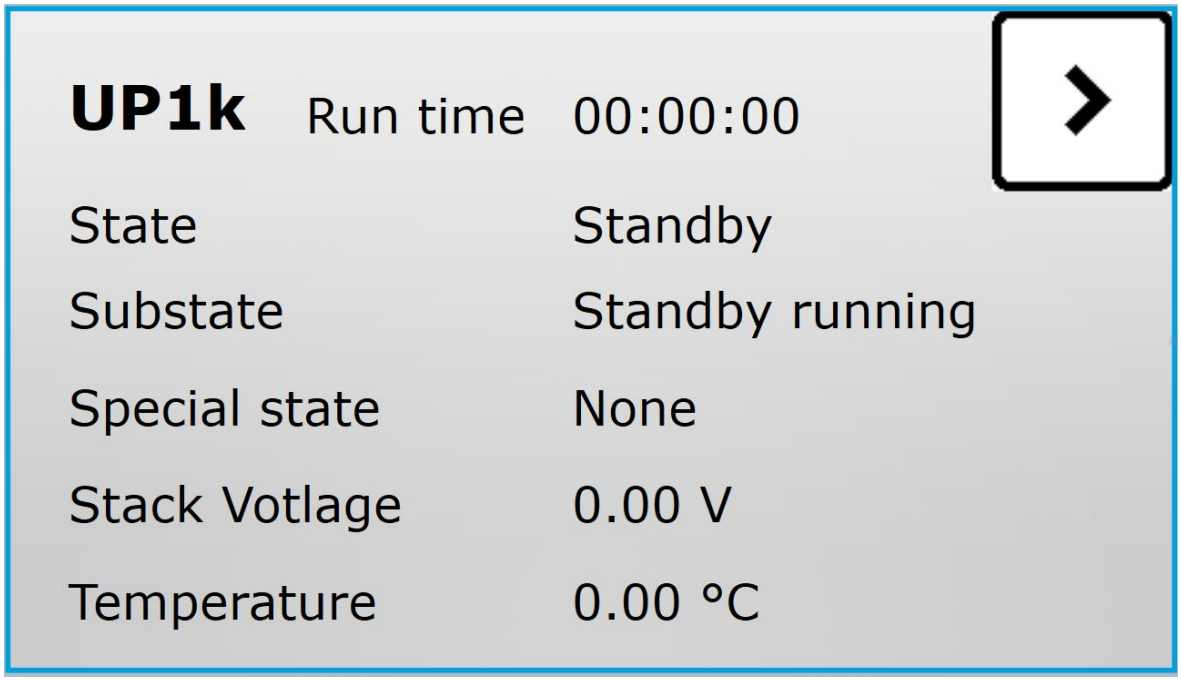

Figure 6.4 Main screen created by TouchGFX

The next two screens are created to show more information in detail. Changing the previous screens to these screens happens by pressing the physical buttons placed on the right side of the display. The info screen 1 shows the battery voltage, temperature, currents of input and output, the voltage of input and output, and converter mode. Furthermore, the info screen 2 shows the speed of fans and valves status.

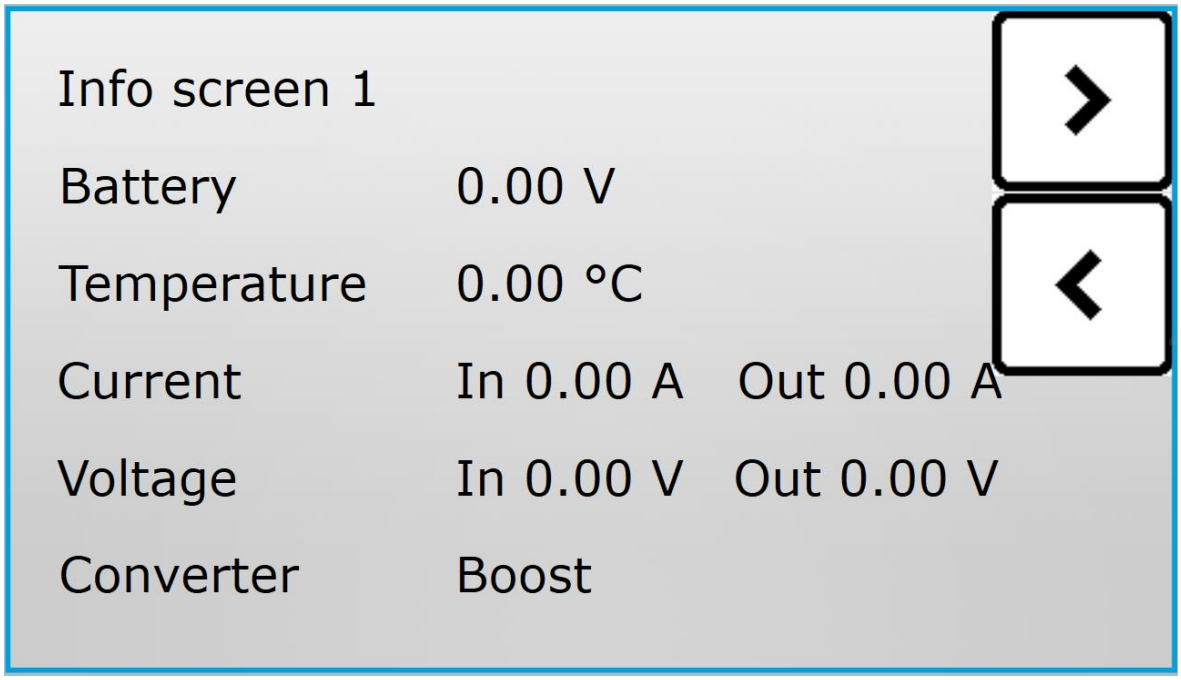

Figure 6.5 Info screen 1 created by TouchGFX

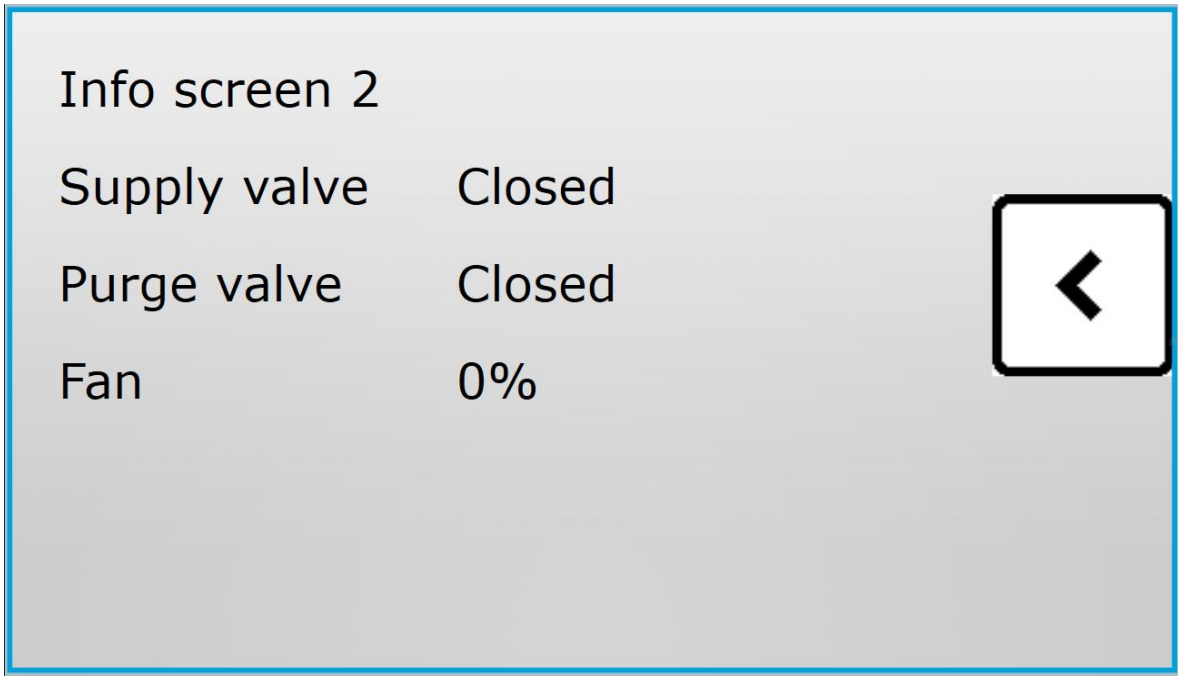

Figure 6.6 Info screen 2 created by TouchGFX

6.4 UP1K Software

UP1K contains two different projects for different cores. Each project should be uploaded separately to the board. STM32CubeIDE supports multiple nested projects. It means both of them are created and configured by one STM32Cube (.ioc) file.

6.4.1 Directory Structure

In the main directory of UP1K project, there are the .project and .ioc files and some sub-directories. In the Common directory, there is an initialisation file for both Cortex M4 and Cortex M7 projects which is generated by STM32Cube. Drivers directory contains the CMSIS and HAL driver directories. Then there is the Modules directory which contains the modules that are described before. The modules are added to the project as a git submodule. In this case, it is possible to update the module independently. Then there are the nested projects. The first nested project is located in CM4 directory which includes the Cortex M4 project. It contains the Core directory, which includes the project C files like *main.c*, *main.h*, *uart.h*, etc. Moreover, CM4 directory consists of a Debug directory that stores the binary files of compiled files. As well as this, the linker file for the Cortex M4 project is placed in this directory. As well as CM4 directory, there is CM7 directory which includes the same files as CM4 and some additional directory and files. Another Drivers directory contains the additional libraries that are mostly needed for the graphics library and driving the LCD. An external memory bootloader program is also

located in this project with a program inside that loads the external Flash memory and RAM. Then jumps to the program stored there, which is the main program for Cortex M7 and contains the program for the graphics library. Middleware directory contains TouchGFX and FreeRTOS files. In the end, there is a TouchGFX folder that consists of the generated files by the TouchGFX designer. The directory map is shown in the following figure.

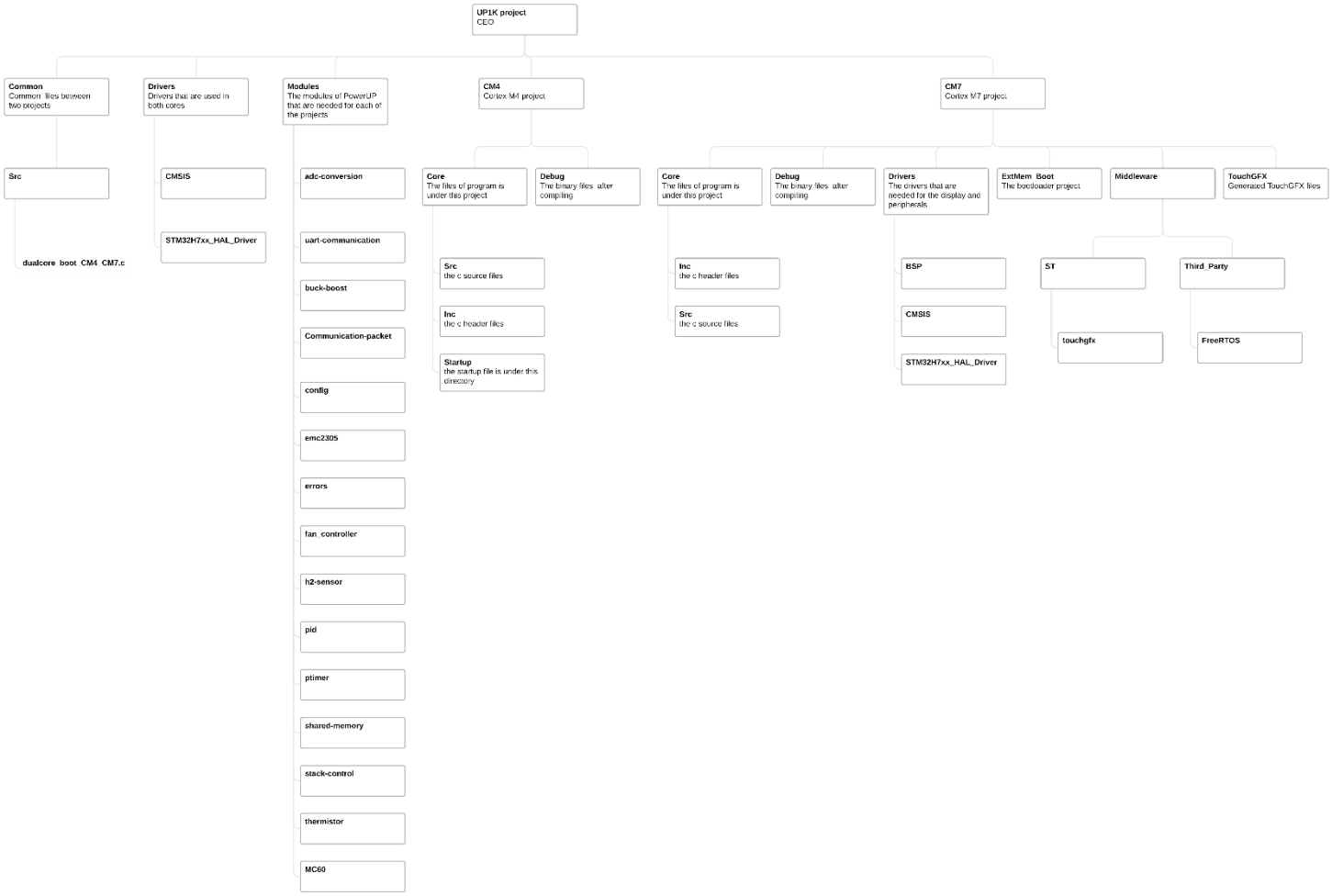

Figure 6.7 UP1K project directory structure map

6.4.2 Modules and dependencies

The modules that are used in UP1K projects are connected as a dependency. The following diagrams show these dependencies. As it is shown, the Cortex M7 project uses fewer modules comparing the Cortex M4 project that handles the stack and output. Moreover, many modules depend on the *config* and *ptimer* modules. The most important point about this diagram is that since all modules depend on the *errors* module, this module is not shown in the diagram to increase readability. If we do not consider *config* and *ptimer* modules, in the Cortex M7 project, *mc60* depends on the *communication-packet* module, and it depends on the *uart-communication* module. Also, the shared-memory module does not have any dependency as it has just some defined variables in a specific place. In Cortex M4 project, the *fan-controller* module depends on the *emc2305* module while the *stack-control* module depends on *buck-boost* and *thermistor* modules, and they both rely on the *adc-conversion* module. Also, *buck-boost* needs the *pid* module for its calculations. Like the Coretx M7 project, shared-memory works independently.

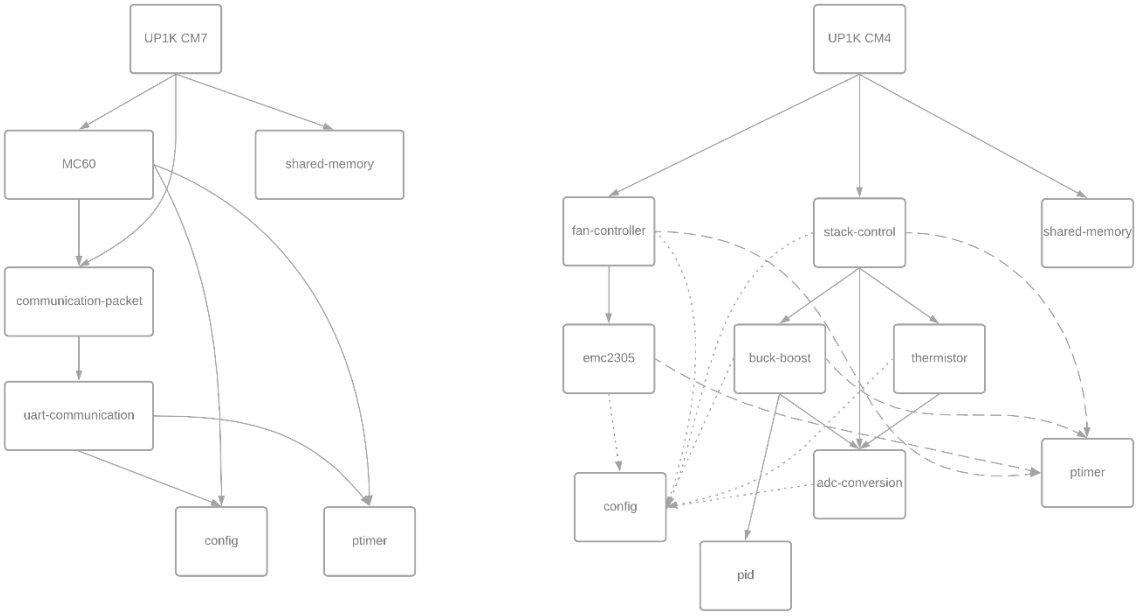

Figure 6.8 UP1K projects dependency diagram for both Cortex M7 (left) and Cortex M4 (right) projects

6.4.3 Software layer chart

The embedded software can be demonstrated as layered architecture software by ignoring some parts of the program (e.g. RTOS and pid module). In this case, Figure 6.9 illustrates the layers. Each part that is on top of some other sections shows that it is using all the below programs. For instance, the *fan-controller* module in the Cortex M4 program is created based on the *emc2305* module, which is based on the HAL driver. Also, they use an SMBUS peripheral of the MCU.

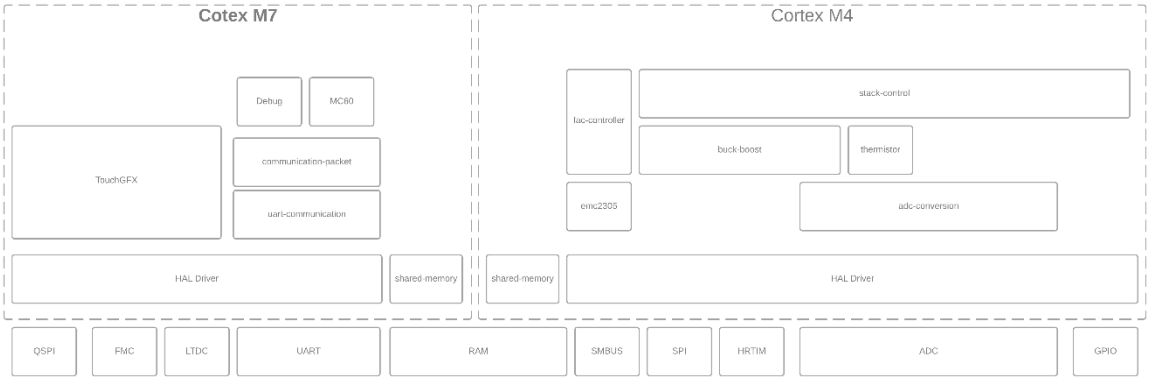

Figure 6.9 UP1K software layers diagram

6.5 UP200 Software

6.5.1 Directory structure

The UP200 project directory includes 5 sub-directories.

Drivers folder includes CMSIS and HAL drivers. These are the important libraries to communicate with the peripherals and control hardware.

Middleware directory includes FreeRTOS and FATFS as Third-Party middlewares.

Modules contains the modules that are needed in the UP200 Software. It is explained in more details in the next item.

Core directory consists of the source and header files of the project. The source files are located in the Src folder, and the header files are placed in the Inc folder. It also includes a Startup directory which provides for the startup file configurations.

Debug folder is the last directory that the compiler stores the binary output files in it.

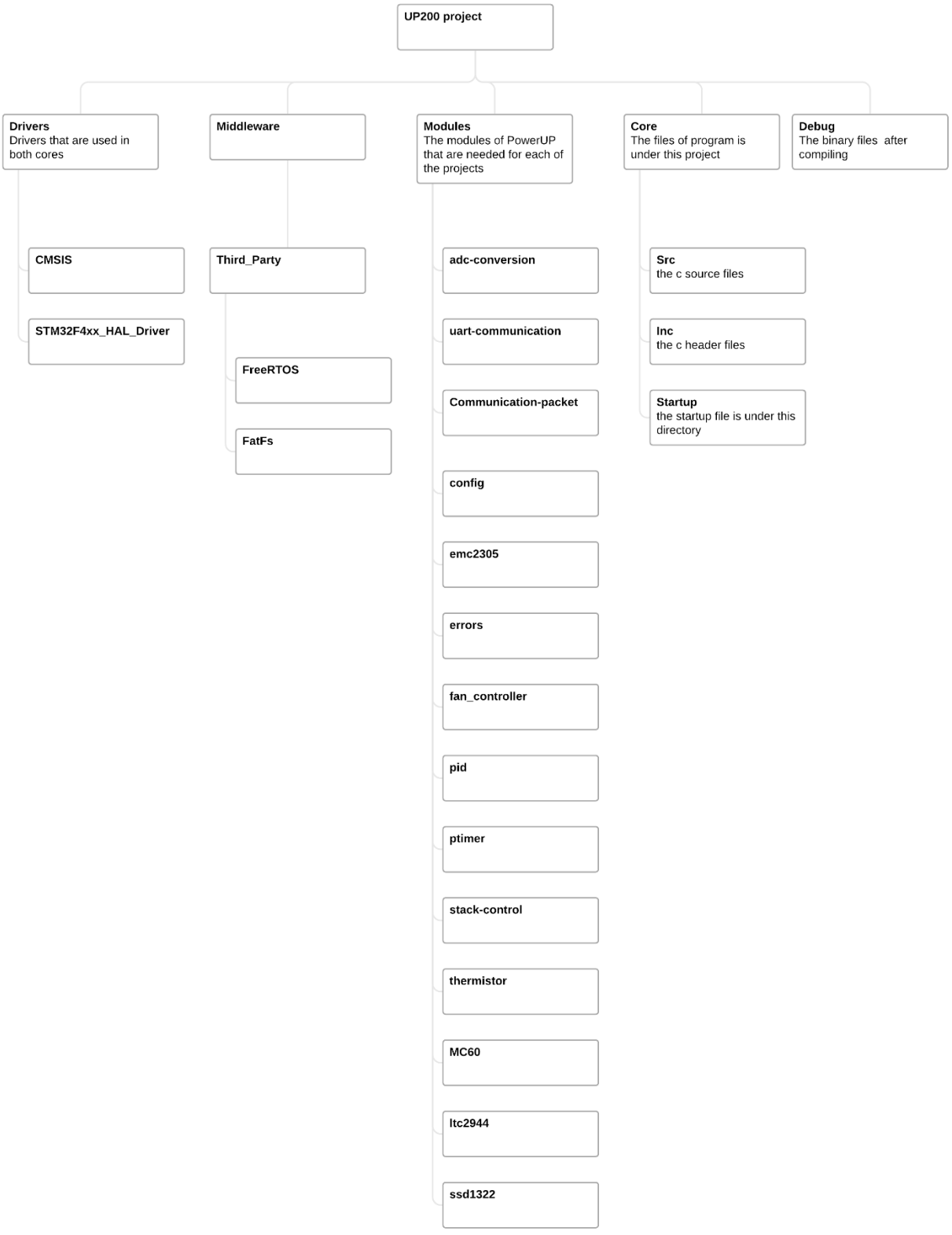

Figure 6.10 UP200 directory map diagram

6.5.2 Modules and dependencies

The modules in UP200 project use each other for their functionality. Figure 6.11 shows the dependencies. If the *config* and *ptimer,* which are used in many other modules, be ignored, they can be explained with no complexity. The mc60 module includes the communication packet, and it includes *uart-communication* module. The *fan-controller* is on top of *emc2305*. The *stack-control* uses *thermistor*, *adc-conversion*, and *ltc2944* modules. Also, *thermistor* module uses *adc-conversion* module. In the end, *ssd1322* module is used in the program for showing the values on the screen. Furthermore, it should be mentioned as the errors module is used in all the modules, it is removed from the diagram to ease understanding.

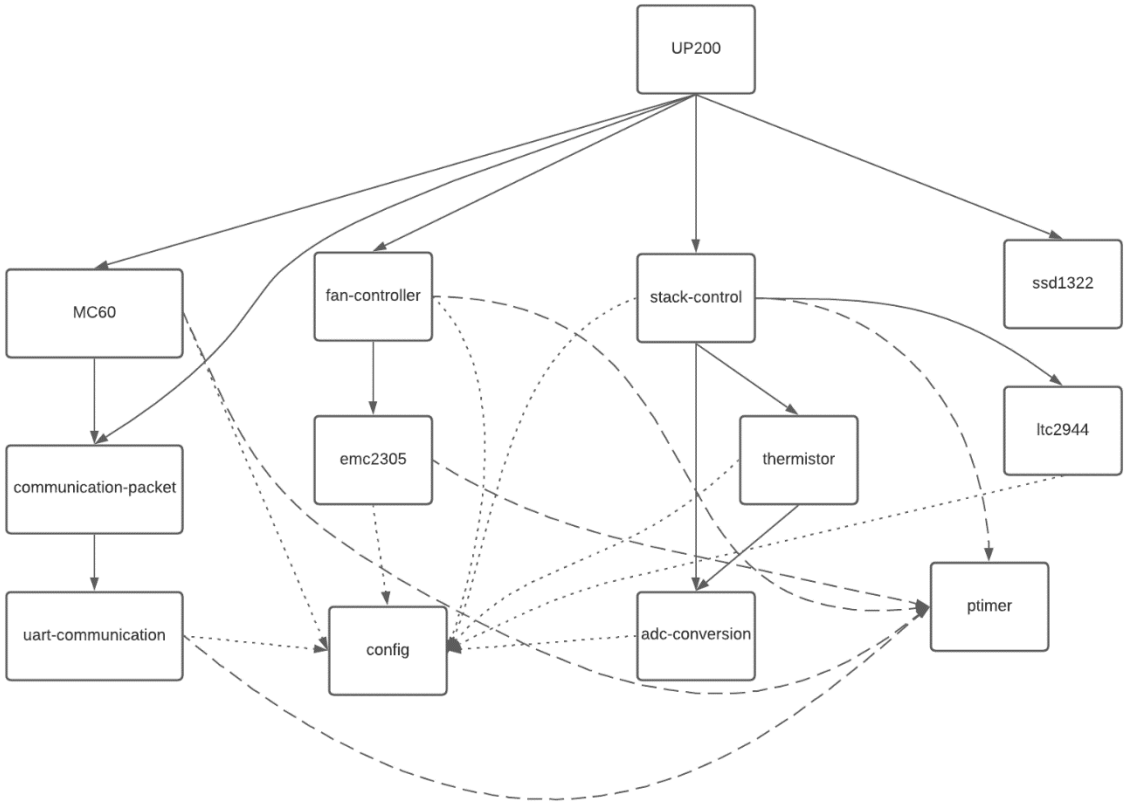

Figure 6.11 UP200 project dependency diagram

6.5.3 Software layer chart

Figure 6.12 illustrates a simplified software layers diagram of the UP200 software project. In this figure, each module is shown on top of other modules, drivers, and peripherals. In each column, it can be checked that how each module is created based on another one. For example, debug and mc60 modules are based on communication-

packet module which is created on top of uart-communication module and it uses HAL driver to control UART.

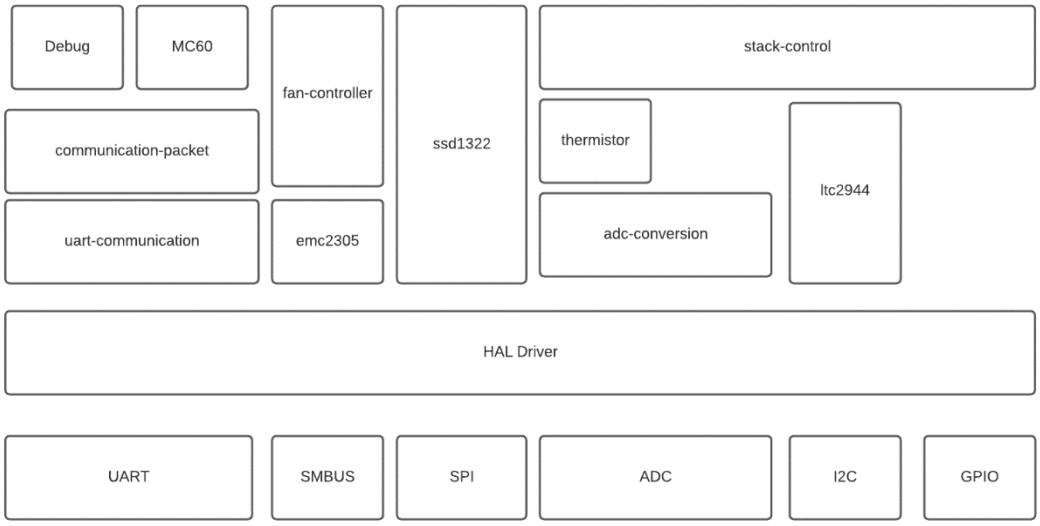

Figure 6.12 UP200 software layers diagram

SUMMARY

In conclusion, for using Hydrogen as fuel, there are several types of fuel cell and applications. The thesis goal is developing a portable Hydrogen fuel cell generator product. For Hydrogen fuel cell products, the PEM fuel cell is the most proper type of generator with low weight and low operation temperature.

The thesis covered a comparison between different types of fuel cells. The connections, specifications, and configurations of the chosen PEM fuel cell are explained in details. Moreover, it includes the operation modes, process conditions and controlling methods of the fuel cell. For this purpose, the hardware that controls the fuel cell is shown. Also, the different versions of the hardware and the reasons for substitution with the new versions are described. It shows how the hardware increases the performance and features of the generator.

The outcome of the thesis is an embedded software architecture and implementation that can control a hydrogen fuel cell stack and its output power. As the firmware is implemented in a modular method, each part of the program is tested module by module. The lab test showed acceptable performance while running the program. These modules are architected and/or written by the author and described in details. The modular design of software caused to use of each part of the software separately in different generators. It can be used as a software platform, even for future fuel cell generators. The thesis also covered the software configuration in the toolchain for all the modules that were needed. On the other hand, it shows the third-party middlewares that helped to increase the speed of development. All the diagrams and flow charts that show how the software works are added to the thesis for more readability.

Although it is tried to have the most efficient software, there are many ways to increase its performance in future. The most important part that helps this performance growth is creating a customized HAL driver with the least overhead in the process. Furthermore, as an external communication module is used in the products, many features are possible to add by changing the module to a 4G module in the future. The IoT implementation of this product can be one of the best options. On the other hand, some development is still in progress on the generator. One of them is the implementation of Modbus communication for syncing the generators together and also to PLC. It will be implemented over RS-485, which is one of the standard communication protocols for Modbus. The other important development is a secure bootloader for the user to update the firmware by UART and also Bluetooth. In that case, the user can upgrade to the new firmware without any technical knowledge.

Selleks, et kasutada vesinikku kütusena, on olemas erinevaid kütuseelementide tüüpe ja rakendusi. Selle lõputöö eesmärk on arendada välja kaasaskantav vesiniku kütuseelementidel põhinev generaator. Vesiniku kütuseelementide seast on kõige kasulikum kasutada polümeerelektrolüüt-kütuseelemente (PEKE), kuna seda tüüpi kütuseelemendid on kõige kergemad ja töötavad madalal temperatuuril.

Käesolevas lõputöös on käsitletud erinevate kütuseelementide võrdlus. Samuti on lõputöös täiendavalt selgitatud valitud PEKE ühendused, tehnilised andmed ja erinevad konfiguratsioonid. Kütuseelemendi töörežiimide, protsessi tingimuste ja juhtimismeetodite selgitamiseks on välja toodud selle juhtimiseks tarviliku riistvara disain. Illustreerimaks riistvaralahenduste mõju generaatori efektiivsusele, on lõputöös välja toodud ja täpsemalt lahatud ajas tehtud muutused riistvara versioonides.

Lõputöö väljund on generaatori vesinikkütuseelemendite ja nende väljundvõimsuse juhtimiseks mõeldud sardsüsteemi tarkvara ja selle arhitektuuri arendamine ja rakendamine. Tänu sardvara modulaarsusele, on võimalik programmi erinevaid osasid teineteisest sõltumatult testida. Laborikatsetused näitasid programmi rahuldavat võimekust. Sardvara moodulite arhitektuur on loodud ja sardvara on kirjutatud lõputöö autori poolt ja on lõputöös täpsemalt kirjeldatud. Samuti võimaldab selle modulaarne disain kasutada sardvara osasid eraldiseisvatena teistes generaatorites – seda saab kasutada tarkvaraplatvormina. Lõputöös on kirjeldatud tarkvara konfigureerimist tööriistaahelas. Samuti on välja toodud kõik kolmandate osapoolte kirjutatud vahetarkvarad, mille aitasid tarkvara kirjutamist kiirendada. Lõputöö sisu paremaks selgitamiseks on lisatud ka diagramme ja vooskeeme.

Kuigi lõputöös kirjeldatud sardvara on üritatud arendada nii efektiivseks kui võimalik, on siiski veel võimalusi selle võimekuse parandamiseks. Kõige suuremat mõju efektiivsuse parandamisel annaks riistvara abstraktsioonikihi draiveri muutmine nõnda, et lisakulud arvutusvõimekusele oleksid minimaalsed. Tänu asjaolule, et kommunikatsiooniks kasutatakse eraldiseisvat moodulit, saab tulevikus vajaduse korral see moodul vahetada välja 4G võrku kasutava mooduli vastu. Generaatori ühendamine asjade internetti on üks paremaid võimalusi selle kasutusvaldkonda laiendada olukordadesse, kus kasutaja igapäevaselt generaatoriga kokku ei puutu. Üks võimalus selleks oleks rakendada Modbus suhtlust generaatorite omavaheliseks sünkroniseerimiseks ja programmeeritava loogikakontrolleriga (PLC) ühendamiseks. Seda lahendust saaks kasutusse võtta RS-485 abil, mis on üks standardne suhtlusprotokoll Modbusile. Generaatori kasutusmugavust suurendaks turvaline alglaadur, mis võimaldaks kasutajal uuendada sardvara üle UARTi või Bluetoothi. See võimaldaks kasutajal muuta tarkvara ilma tehnilisi teadmisi omamata.

LIST OF REFERENCES

[1] "Analysis of the control strategies for fuel saving in the hydrogen fuel cell vehicles | Elsevier Enhanced Reader." https://reader.elsevier.com/reader/sd/pii/S0360319917348711?token=52CF255135293DD474A45191186F0653571279E051235DD080D26FD5517BF3BF462C4900938238D0C25B7F10F38FE7AF&originRegion=eu-west-1&originCreation=20210606193557 (accessed Jun. 06, 2021).

[2] M. Ball and M. Weeda, "The hydrogen economy - Vision or reality?," *Int. J. Hydrogen Energy*, vol. 40, no. 25, pp. 7903–7919, Jul. 2015, doi: 10.1016/j.ijhydene.2015.04.032.

[3] Y. Manoharan *et al.*, "Hydrogen Fuel Cell Vehicles; Current Status and Future Prospect," doi: 10.3390/app9112296.

[4] A. Baroutaji, T. Wilberforce, M. Ramadan, and A. G. Olabi, "Comprehensive investigation on hydrogen and fuel cell technology in the aviation and aerospace sectors," *Renew. Sustain. Energy Rev.*, vol. 106, pp. 31–40, May 2019, doi: 10.1016/j.rser.2019.02.022.

[5] A. Ajanovic and R. Haas, "Prospects and impediments for hydrogen and fuel cell vehicles in the transport sector," *Int. J. Hydrogen Energy*, vol. 46, no. 16, pp. 10049–10058, Mar. 2021, doi: 10.1016/j.ijhydene.2020.03.122.

[6] P. D. Thesis, "Dries Verstraete The Potential of Liquid Hydrogen for long range aircraft propulsion SCHOOL OF ENGINEERING," Cranfield University, 2009. Accessed: Jun. 08, 2021. [Online]. Available: http://dspace.lib.cranfield.ac.uk/handle/1826/4089.

[7] "What are the Pros and Cons of Hydrogen Fuel Cells? - TWI." https://www.twi-global.com/technical-knowledge/faqs/what-are-the-pros-and-cons-of-hydrogen-fuel-cells (accessed Jun. 08, 2021).

[8] "11 Big Advantages and Disadvantages of Hydrogen Fuel Cells – Green Garage." https://greengarageblog.org/11-big-advantages-and-disadvantages-of-hydrogen-fuel-cells (accessed Jun. 08, 2021).

[9] "Control strategies for high-power electric vehicles powered by hydrogen fuel cell, battery and supercapacitor | Elsevier Enhanced Reader." https://reader.elsevier.com/reader/sd/pii/S0957417413001449?token=908C09

3BE4FF084CD3F72D498BB45B2EA579FE1B9E9DC185125C94F225EE76E99CCC209A7DC23F15C67E2C1B86568933&originRegion=eu-west-1&originCreation=20210606205520 (accessed Jun. 06, 2021).

[10] Y. Haseli, "Maximum conversion efficiency of hydrogen fuel cells," *Int. J. Hydrogen Energy*, vol. 43, no. 18, pp. 9015–9021, May 2018, doi: 10.1016/j.ijhydene.2018.03.076.

[11] "A Basic Overview of Fuel Cell Technology." https://americanhistory.si.edu/fuelcells/basics.htm (accessed Jun. 08, 2021).

[12] G. F. McLean, T. Niet, S. Prince-Richard, and N. Djilali, "An assessment of alkaline fuel cell technology," *Int. J. Hydrogen Energy*, vol. 27, no. 5, pp. 507–526, May 2002, doi: 10.1016/S0360-3199(01)00181-1.

[13] "Fuel Cell Systems - Google Books." https://books.google.ee/books?hl=en&lr=&id=s9QFCAAAQBAJ&oi=fnd&pg=PA1&dq=alkaline+fuel+cell+efficiency&ots=q11HfO6AZQ&sig=g7qtDtycNj6gCPNWR49jm2oWieg&redir_esc=y#v=onepage&q=alkaline fuel cell efficiency&f=false (accessed Jun. 08, 2021).

[14] D. DeFelice, "NASA - Fuel Cells: A Better Energy Source for Earth and Space."

[15] "ProQuest Ebook Central - Reader." https://ebookcentral.proquest.com/lib/tuee/reader.action?docID=1044921 (accessed Jun. 08, 2021).

[16] B. Ghorbani, M. Mehrpooya, and S. A. Mousavi, "Hybrid molten carbonate fuel cell power plant and multiple-effect desalination system," *J. Clean. Prod.*, vol. 220, pp. 1039–1051, May 2019, doi: 10.1016/j.jclepro.2019.02.215.

[17] A. Lanzini *et al.*, "Dealing with fuel contaminants in biogas-fed solid oxide fuel cell (SOFC) and molten carbonate fuel cell (MCFC) plants: Degradation of catalytic and electro-catalytic active surfaces and related gas purification methods," *Progress in Energy and Combustion Science*, vol. 61. Elsevier Ltd, pp. 150–188, Jul. 01, 2017, doi: 10.1016/j.pecs.2017.04.002.

[18] M. Mehrpooya, S. Sayyad, and M. J. Zonouz, "Energy, exergy and sensitivity analyses of a hybrid combined cooling, heating and power (CCHP) plant with molten carbonate fuel cell (MCFC) and Stirling engine," *J. Clean. Prod.*, vol. 148, pp. 283–294, Apr. 2017, doi: 10.1016/j.jclepro.2017.01.157.

[19] M. Marefati, M. Mehrpooya, and M. B. Shafii, "A hybrid molten carbonate fuel cell and parabolic trough solar collector, combined heating and power plant with carbon dioxide capturing process," *Energy Convers. Manag.*, vol. 183, pp. 193–209, Mar. 2019, doi: 10.1016/j.enconman.2019.01.002.

[20] A. Mehmeti, F. Santoni, M. Della Pietra, and S. J. McPhail, "Life cycle assessment of molten carbonate fuel cells: State of the art and strategies for the future," *J. Power Sources*, vol. 308, pp. 97–108, 2016.

[21] S. V. M. Guaitolini, I. Yahyaoui, J. F. Fardin, L. F. Encarnacao, and F. Tadeo, "A review of fuel cell and energy cogeneration technologies," in *2018 9th International Renewable Energy Congress, IREC 2018*, May 2018, pp. 1–6, doi: 10.1109/IREC.2018.8362573.

[22] S. Wilailak *et al.*, "Thermo-economic analysis of Phosphoric Acid Fuel-Cell (PAFC) integrated with Organic Ranking Cycle (ORC)," *Energy*, vol. 220, p. 119744, Apr. 2021, doi: 10.1016/j.energy.2020.119744.

[23] "Phosphoric Acid Fuel Cells - an overview | ScienceDirect Topics." https://www.sciencedirect.com/topics/chemical-engineering/phosphoric-acid-fuel-cells (accessed Jun. 09, 2021).

[24] "Platinum Electrocatalysts for Phosphoric Acid Fuel Cells | Johnson Matthey Technology Review." https://www.technology.matthey.com/article/26/3/118-120/ (accessed Jun. 09, 2021).

[25] H. Vaghari, H. Jafarizadeh-Malmiri, A. Berenjian, and N. Anarjan, "Recent advances in application of chitosan in fuel cells," *Sustain. Chem. Process.*, vol. 1, no. 1, p. 16, 2013, doi: 10.1186/2043-7129-1-16.

[26] R. E. Rosli *et al.*, "A review of high-temperature proton exchange membrane fuel cell (HT-PEMFC) system," *Int. J. Hydrogen Energy*, vol. 42, no. 14, pp. 9293–9314, Apr. 2017, doi: 10.1016/j.ijhydene.2016.06.211.

[27] T. Zhang, P. Wang, H. Chen, and P. Pei, "A review of automotive proton exchange membrane fuel cell degradation under start-stop operating condition," *Applied Energy*, vol. 223. Elsevier Ltd, pp. 249–262, Aug. 01, 2018, doi: 10.1016/j.apenergy.2018.04.049.

[28] D. Banham *et al.*, "Ultralow platinum loading proton exchange membrane fuel cells: Performance losses and solutions," *J. Power Sources*, vol. 490, p. 229515,

Apr. 2021, doi: 10.1016/j.jpowsour.2021.229515.

[29] A. C. Fărcaş and P. Dobra, "Adaptive Control of Membrane Conductivity of PEM Fuel Cell," *Procedia Technol.*, vol. 12, pp. 42–49, 2014, doi: 10.1016/j.protcy.2013.12.454.

[30] S. Hussain and L. Yangping, "Review of solid oxide fuel cell materials: cathode, anode, and electrolyte," *Energy Transitions*, vol. 4, no. 2, pp. 113–126, Dec. 2020, doi: 10.1007/s41825-020-00029-8.

[31] L. Fan, B. Zhu, P. C. Su, and C. He, "Nanomaterials and technologies for low temperature solid oxide fuel cells: Recent advances, challenges and opportunities," *Nano Energy*, vol. 45. Elsevier Ltd, pp. 148–176, Mar. 01, 2018, doi: 10.1016/j.nanoen.2017.12.044.

[32] R. M. Ormerod, "Solid oxide fuel cells," *Chemical Society Reviews*, vol. 32, no. 1. Royal Society of Chemistry, pp. 17–28, Dec. 18, 2003, doi: 10.1039/b105764m.

[33] E. D. Wachsman and K. T. Lee, "Lowering the temperature of solid oxide fuel cells," *Science*, vol. 334, no. 6058. American Association for the Advancement of Science, pp. 935–939, Nov. 18, 2011, doi: 10.1126/science.1204090.

[34] K. Kendall and M. Kendall, *High-Temperature Solid Oxide Fuel Cells for the 21st Century: Fundamentals, Design and Applications: Second Edition*. Elsevier Inc., 2015.

[35] S. A. Saadabadi, A. Thallam Thattai, L. Fan, R. E. F. Lindeboom, H. Spanjers, and P. V. Aravind, "Solid Oxide Fuel Cells fuelled with biogas: Potential and constraints," *Renewable Energy*, vol. 134. Elsevier Ltd, pp. 194–214, Apr. 01, 2019, doi: 10.1016/j.renene.2018.11.028.

[36] P. Arunkumar, M. Meena, and K. S. Babu, "A review on cerium oxide-based electrolytes for ITSOFC," *Nanomater. Energy*, vol. 1, no. 5, pp. 288–305, Sep. 2012, doi: 10.1680/nme.12.00015.

[37] D. Hart, S. Jones, and J. Lewis, "The Fuel Cell Industry Review 2020."

[38] "OpenPose - Installation." https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/doc/installation/README.md#compiling-and-running-openpose-from-source (accessed Jan. 06, 2021).

[39] X. Liu, K. Reddi, A. Elgowainy, H. Lohse-Busch, M. Wang, and N. Rustagi,

"Comparison of well-to-wheels energy use and emissions of a hydrogen fuel cell electric vehicle relative to a conventional gasoline-powered internal combustion engine vehicle," *Int. J. Hydrogen Energy*, vol. 45, no. 1, pp. 972–983, Jan. 2020, doi: 10.1016/j.ijhydene.2019.10.192.

[40] T. Wilberforce, A. Alaswad, A. Palumbo, M. Dassisti, and A. G. Olabi, "Advances in stationary and portable fuel cell applications," *Int. J. Hydrogen Energy*, vol. 41, no. 37, pp. 16509–16522, Oct. 2016, doi: 10.1016/j.ijhydene.2016.02.057.

[41] "Electric generator - H2sys." https://www.h2sys.fr/en/electric-generator/ (accessed Jun. 08, 2021).

[42] "GreenBox 2 | H2Planet - Re-evolution started - Hydrogen & fuel-cell experience." https://www.h2planet.eu/en/landing_page/greenbox_1 (accessed Jun. 08, 2021).

[43] "Product Center-Jiangsu Yanchang Sunlaite News Energy Co.,Ltd." http://en.sunlaite.com/product_show.php?id=35 (accessed Jun. 08, 2021).

[44] "Internal PowerUP Energy Technologies document," Tallinn.

[45] "UP1K | PowerUP Energy Technologies." https://www.powerup-tech.com/products/up1k (accessed Jun. 09, 2021).

[46] "STM32H745xI/G Datasheet - production data," no. STM32H745XI. 2019, [Online]. Available: https://www.st.com/resource/en/datasheet/stm32h745xi.pdf.

[47] "MC60&M66&M66 R2.0&M66-DSBT Application Note," no. MC60. 2021, [Online]. Available: https://www.quectel.com/download/quectel_mc60m66m66-r2-0m66-ds_bt_application_note_v1-3/.

[48] "Micron Serial NOR Flash Memory 3V, Multiple I/O, 4KB, 32KB, 64KB, Sector Erase," no. MT25QL512ABB. 2019, [Online]. Available: https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/nor-flash/serial-nor/mt25q/die-rev-b/mt25q_qlkt_l_512_abb_0.pdf?rev=0ef0faa5f7b645d7bc11c30bfd27505b.

[49] "SDR SDRAM MT48LC4M32B2 – 1 Meg x 32 x 4 Banks," no. MT48LC4M32B2. 2016, [Online]. Available: https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/128mb_x32_sdram.pdf?rev=a4b9962d86784413b3cfa348b78a1360

.

[50] "SC7283 720x544 System-On-Chip Driver for 480RGBx272 TFT LCD | Datasheet," no. SC7283. 2018, [Online]. Available: https://www.orientdisplay.com/pdf/SC7283.pdf.

[51] "STM32H745I-DISCO STM32H750B-DK Data brief," no. STM32H745I-DISCO. 2019, [Online]. Available: https://www.st.com/resource/en/data_brief/stm32h745i-disco.pdf.

[52] "EMC2301/2/3/5 datasheet." Microchip, 2021.

[53] R. Fan, "SMBus Quick Start Guide," 2012.

[54] "DRV8876 H-Bridge Motor Driver With Integrated Current Senseand Regulation." Texas Instrument, 2021.

[55] "UP200 | PowerUP Energy Technologies." https://www.powerup-tech.com/products/up200 (accessed Jun. 09, 2021).

[56] "STM32F427xx STM32F429xx datasheet." ST, 2018.

[57] "SSD1322 Advance Information." Solomon Systech Limited, 2010.

[58] "ACS770xCB." ALLEGRO microsystems, 2019.

[59] "LTC4020." LINEAR TECHNOLOGY, 2016.

[60] "LTC2944." LINEAR TECHNOLOGY, 2017.

[61] E. White, *Making Embedded Systems*, First Edit. O'Reilly Media, Inc., 2011.

[62] "STM32CubeIDE - Integrated Development Environment for STM32 - STMicroelectronics." https://www.st.com/en/development-tools/stm32cubeide.html#overview (accessed Jun. 09, 2021).

[63] "Eclipse desktop & web IDEs | The Eclipse Foundation." https://www.eclipse.org/ide/ (accessed Jun. 09, 2021).

[64] "Description of STM32F4 HAL and low-layer drivers." ST, 2020.

[65] "FatFs - Generic FAT Filesystem Module." http://elm-chan.org/fsw/ff/00index_e.html (accessed Jun. 04, 2021).

[66] “Framebuffer | TouchGFX Documentation.” https://support.touchgfx.com/docs/4.15/basic-concepts/framebuffer (accessed Jun. 05, 2021).