UNIVERSITÀ
DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Informatica

Elaborato finale

Battery Management System Development

Issues and applications in a Formula SAE electric race car

Supervisore
Roberto Passerone

Laureando
Matteo Bonora

Anno accademico 2021/2022

Contents

1 Background **4**
- 1.1 Formula SAE . . . 4
  - 1.1.1 Competitions . . . 4
- 1.2 E-Agle Trento Racing Team . . . 5

2 Introduction **7**
- 2.1 Tractive System . . . 7
- 2.2 Batteries . . . 7
  - 2.2.1 Battery Pack . . . 7
  - 2.2.2 Battery Management System . . . 10
- 2.3 Issues and objectives . . . 11
  - 2.3.1 Battery Management . . . 11
  - 2.3.2 Module Balancing . . . 12
  - 2.3.3 Error Management . . . 12

3 Cell Balancing **13**
- 3.1 Balancing techniques . . . 14
  - 3.1.1 Strategy . . . 14
- 3.2 Algorithm . . . 15
  - 3.2.1 Imbalance computation . . . 15
  - 3.2.2 Cell Selection . . . 15
  - 3.2.3 Implementation . . . 16
  - 3.2.4 Pseudo-code . . . 17

4 Error Management **19**
- 4.1 Error Definition . . . 19
  - 4.1.1 Types of errors . . . 19
- 4.2 Timeouts . . . 20
  - 4.2.1 Timekeeping . . . 20
  - 4.2.2 Timer Scheduling . . . 21
- 4.3 Implementation . . . 21
  - 4.3.1 Error setting and resetting . . . 22
  - 4.3.2 Scheduling . . . 22
  - 4.3.3 Logging . . . 23

5 Experimental results **25**
- 5.1 Balancing . . . 25
- 5.2 Error management . . . 25

6 Conclusions and future works **27**
- 6.1 Cell Balancing . . . 27
- 6.2 Error Management . . . 27
- 6.3 State of charge estimation . . . 27

Bibliography **27**

A Discharge energy **29**

B Balancing test **30**

C Error management test **32**

Abstract

This thesis covers my work inside E-Agle Trento Racing Team from 2019 to 2021 as the lead firmware developer on the high-voltage battery management system for Fenice, the electric race car planned to compete in the 2022 Formula Student season. The paper overviews some key software components of the BMS that were introduced with my developments.

The first chapter introduces the competition environment and presents the team. The race organization and structure are explained to better understand the requirements of the car and the battery pack.

The introduction presents a simple overview of the powertrain of Fenice. It continues by explaining the basic behavior of batteries under load along with the physical and electrical structure of a battery pack. Requirements are then discussed on the capacity and maximum power of the pack. Finally, the BMS architecture is overlaid and the role of each component is briefly explained.

The cell balancing algorithm is the topic of the third chapter. First, the problem is laid out with help from data acquired on the older car: Chimera Evoluzione. The balancing algorithm is then introduced along with all the requirements and constraints that justify the software design. Significant focus is put on the analysis of complexity of the algorithm.

A crucial element of the BMS, error management, is then analyzed. The chapter explains constraints imposed by the rulebook and the approach taken to respect them while having a scalable and efficient error tracking system that emphasizes timing accuracy and scheduling.

In conclusion, the results of the cell balancing algorithm are shown with experimental data that validate the algorithm and its implementation. Some possible issues are identified and a fix is proposed.

The error management implementation is shown working on real hardware and the timing functionality is tested with the use of a logic analyzer.

Future advancement of the BMS in the form of state of charge estimation is then proposed, explaining the benefits of such a feature.

1 Background

1.1 Formula SAE

Formula SAE is an international design competition founded by the American Society of Automotive Engineers (SAE) in 1980, in which university students develop, build and race an open-wheel, single-seater race car. The car must adhere to the Formula SAE rules that govern all the aspects of the car and the competition.

In Europe, Formula Student Germany (FSG) is the main governing body. FSG releases the rulebook [1] that is used in all European competitions and it organizes the biggest competition of the season. Many more Formula SAE events are held all around Europe and the world.

1.1.1 Competitions

A Formula SAE competition consists of static and dynamic events in which industry experts judge the cars and the teams on several different aspects by awarding points for each event.

Tech Inspections

At the start of the competition technical inspections are carried out to verify that each car is respecting the rulebook and is thus eligible to participate in the dynamic events. The inspection is carried out as follows:

- **Pre-inspection**: safety hardware, driver equipment, tires, and rims are checked for compliance with the SAE and FIA rules.
- **Accumulator inspection**: the battery pack's adherence to the SAE rules is verified by checking its insulation and data acquisition methods. The pack is then sealed for the rest of the competition.
- **Electrical inspection**: all electrical components on the car are checked for compliance by verifying every component's datasheet. Furthermore, electrical grounding is checked all across the car.
- **Mechanical inspection**: It is verified that the car's dimensions are within rulebook specification. All strength samples for the chassis materials are shown to the officials.
- **Tilt test**: the car with the driver is put on an inclined platform with a lateral angle of 60 degrees. The car must adhere to the platform's surface and no fluids must leak.
- **Rain test**: the car is sprinkled with water for 120s to check proper humidity insulation of electronic components. The car must be on during the test and must remain powered for an additional 120 s.
- **Brake test**: the driver accelerates the car, switches it off and performs an emergency brake test. All four wheels must lock and the car must not deviate its path.

If the car passes every step of the tech inspections, it gets certified and can participate in the dynamic events.

Static Events

In static events, the car and the team are evaluated on theoretical points including design processes, team organization, business ideas, and more. Static events are divided into three separate categories:

- **Engineering design** (150 pts.): The judges evaluate the whole design process of the car, from the early stages of evaluation to the realization and validation of components. Each team's department has to justify all choices made in the design phase with data collected from simulations, previous experience, or QFD methodology.
- **Cost and manufacturing** (100 pts.): The team provides an engineering and a manufacturing BOM in which the cost of each component is reported. Team members have to explain the cost of every component, material, and manufacturing process.
- **Business plan presentation** (100 pts.): In the business plan presentation judges pretend to be potential investors and the team has to make an enticing presentation of its product and the business model in which the team plans to mass-produce the prototype car with studies on costs and profitability. Innovative and original ideas are usually rewarded.

Dynamic Events

Dynamic events value the overall performance of the car compared to its competitors. The car is put through four different tests that each assesses a different aspect of performance:

- **Skidpad** (75 pts.): the car must complete two laps of a figure 8 track, sometimes on wet tarmac.
- **Acceleration** (75 pts.): the car must do a standing start and complete a straight 75 m long track as fast as possible.
- **Autocross** (100 pts.): a single lap of an autocross track with a length lower than 1,5 km must be completed in the least time possible. The track contains slaloms, straights, and sharp turns.
- **Endurance and efficiency** (425 pts.): the car must race for a total length of 22 km around a track similar to the autocross layout.

For each of the previous events, points are calculated based on the time taken to complete the race. For the Endurance event efficiency is also taken into account.

1.2 E-Agle Trento Racing Team

E-Agle Trento Racing Team is the Formula SAE team of the University of Trento. The team was created in 2016, and was immediately involved in the creation of electric race cars. Since its inception, the team realized two Formula SAE cars: Chimera and Chimera Evoluzione. Instead of following an evolutionary approach with the third car, the team decided to design the vehicle from scratch. This choice came from the experience with the two previous cars: some inconvenient design flaws and the will to innovate were the main driving forces of this decision.

The redesign was completed in 2019 with production planned to end in time for the 2020 racing season; unfortunately, due to the COVID-19 pandemic, all competitions were canceled and the team was forced to work away from the workshop. In 2021 production was resumed and the car is planned to compete in the second half of the 2021 season and continue for 2022.

The team is divided into four main divisions, each in charge of a different assembly of the car:

- **Mechanical Team**: is in charge of the design, manufacturing and testing of every mechanical component of the car including the chassis, suspension components, aerodynamics and more.
- **Dynamics and Modeling Team**: the team develops the suspension kinematics and the whole car dynamics based on simulations of a virtual model of the car. By using the same model the team then creates the control logic of the vehicle that includes the traction control and torque vectoring algorithms.
- **Electronics Team**: Fenice relies on custom-designed electronics for most of its features. The team's job is to design and test all the electronic components, from the car's wiring to the high-voltage battery pack, from the printed circuit boards to the complex battery management system inside the high-voltage battery.

- **Software Team**: The award-winning cockpit HMI embedded in the steering wheel is fully custom built by the software team along with the telemetry system that provides data logging and live telemetry. In addition, the team is responsible for developing the control boards designed by the electronics division.

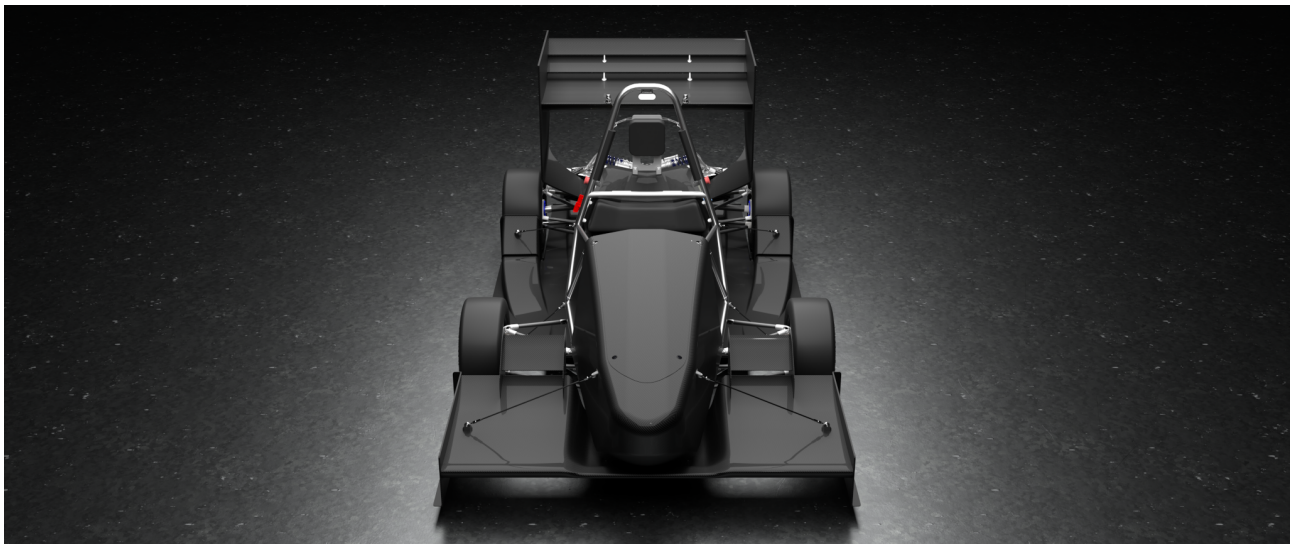

Figure 1.1: A render of Fenice

2 Introduction

2.1 Tractive System

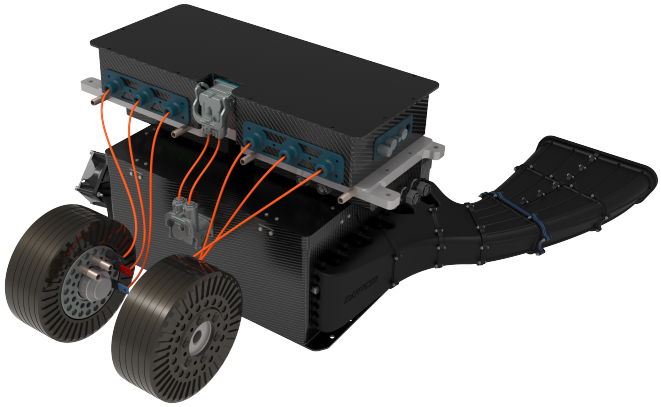

Figure 2.1: Chimera Evoluzione's tractive system with the battery pack below the two inverters that are connected to the motors.

Tractive system refers to all the high-voltage components of the car. It includes the battery pack, the inverters and the electric motors that drive the wheels of the car. As with Chimera Evoluzione, Fenice is powered by two independent three-phase permanent-magnet motors that drive the rear axle. Each motor is controlled by a motor controller that converts the direct current from the battery into a three-phase alternating current for the motors. By varying the output frequency and current, the controllers can set the motor's speed and torque. Motors and controllers are water-cooled while the battery is air-cooled.

2.2 Batteries

A battery is an electrical energy storage system that relies on chemical reactions to generate a voltage difference. The main properties of a battery are: nominal voltage, internal resistance, energy capacity and maximum discharge rate.

The voltage of a battery is influenced by many factors including: state of charge, temperature and applied load. The open-circuit voltage (OCV) of a Lithium-Ion battery cell is 4.2V at 100% state of charge and 3.0V at 0%. When a load is applied to a cell, the voltage drops according to Ohm's law: $V_{dropped} = R_{internal} * I_{load}$. The higher the current drawn, the higher the voltage drop. In high power applications, this phenomenon can significantly reduce the usable energy of the battery.

2.2.1 Battery Pack

A battery pack is a group of cells connected in series and parallel to increase the electrical characteristics of the pack. Arranging the cells in series means that the current will only travel down a single path, passing through every cell. In this case, the voltage of the pack is equaled to the sum of every series.

In a parallel arrangement, the current travels down multiple paths, splitting across more cells. This increases the maximum current output of the battery, but the potential across all cells is equalized. The structure of a battery pack is decided considering its specific application. For example, in a high power context, higher voltage batteries are more desirable because for the same power draw, less current is required, reducing Joule effect heat losses across conductors. In high current applications,

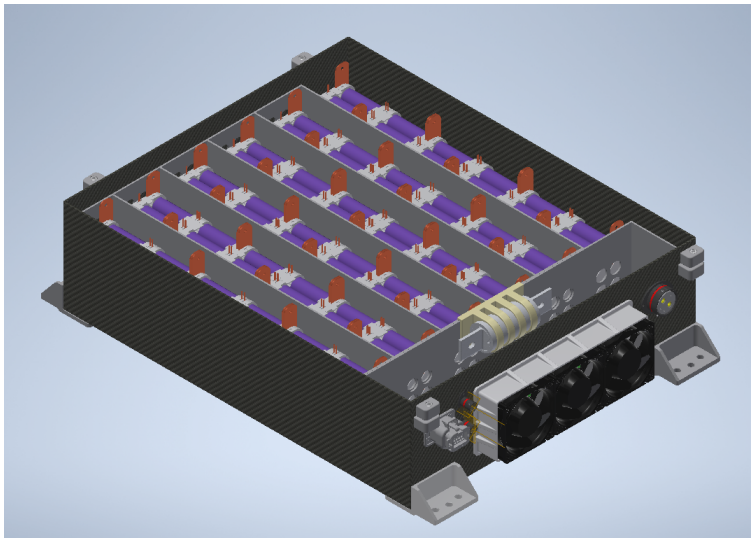

Figure 2.2: 3D model of Fenice's battery pack

more cells in parallel could be arranged, increasing the total energy storage and battery weight as a consequence.

As mentioned above, a Formula SAE car's battery should be sized to complete a 22km endurance course while also being able to output the maximum amount of power for the acceleration event. The battery should also be optimized for weight and have a low center of gravity. Furthermore, the rulebook limits maximum voltage to 600V [1, EV 4.1.1] and power to 80kW [1, EV 2.2.1], so the resulting battery will have as many cells in series as permitted and as few parallels as needed to reach the required power and capacity target.

Fenice's battery features 108 cells in series and 4 in parallel (108s4p), for a total of 432 cells and a nominal voltage of 388.8V. Higher voltages could be reached by rewiring the pack in a 144s3p configuration, but the pack would have a maximum voltage of 604.8V, exceeding the 600V limit. Adding more cells leads to increased weight, volume and excess capacity.

The high power requirement of the pack is fulfilled by the use of high-discharge rate cells, amounting to 45A in Fenice's case [11]. This results in a total output of 180A of continuous discharge current. As a consequence, The maximum theoretical power output of the whole pack approaches 70kW and the energy capacity amounts to 6.2kWh.

Naming hierarchy

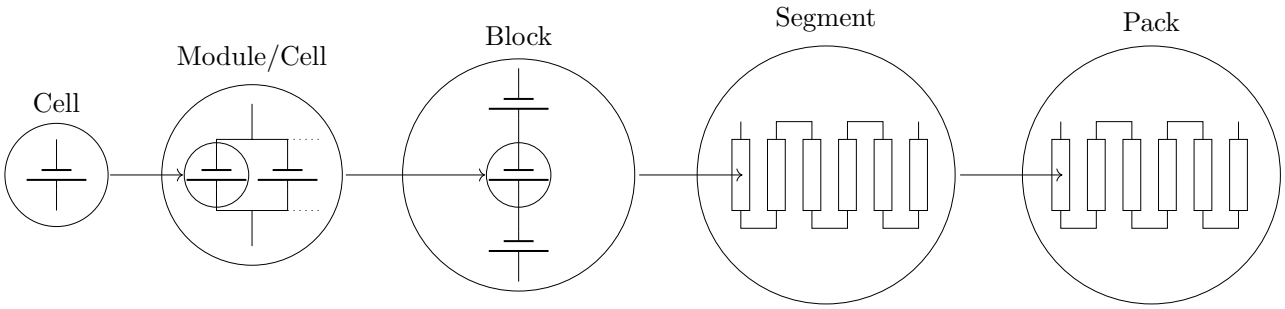

Figure 2.3: Battery pack elements naming scheme

As explained in Figure 2.3, the pack is subdivided into physical blocks that can be summarized as follows:

- The **cell** is the basic unit of the pack.

- Four cells in parallel form a **module** (also called cell, since the voltage is the same as a single cell).

- **Blocks** are a series of three modules, held together to form an actual block.

- The rulebook requires the separation of the pack into **segments** with precise physical and electrical characteristics [1, EV 5.3.2]. In this case, a segment is a series of six blocks, totaling a maximum voltage of 75.6V ($4.2V * 3\ modules * 6\ blocks$) and 1.2kWh of energy, below the limit of 120V and 1.6 kWh mandated by the rules.

- In conclusion, the whole **pack** is composed of six segments connected in series.

Battery insulation

To be able to safely shut off the output of the battery pack, two normally-open Accumulator Isolation Relays (AIR) [1, EV 5.6] are located at both poles of the pack and can connect the internal battery terminals from the external connector located on the battery container.

The terminals of the battery pack must be always insulated from the rest of the low voltage system. For this reason, an insulation monitoring device (IMD) continuously checks for insulation between both the battery poles and the car's ground. The AIRs must disconnect if the insulation falls below a resistance greater than $^{500\Omega}/_{V_{max}}$, where V_{max} is the maximum pack voltage. The IMD is part of a broader circuit that drives the AIRs, called shutdown circuit.

Shutdown Circuit

The shutdown circuit carries the power that drives the coils of both AIRs. Figure 2.4 shows the block schema of the circuit with all the safety switches and their normal state. The most important switches are:

- **LVMS** (Low-voltage master switch): Cuts low voltage power to the entire car. It is used as a power switch.

- **BSPD** (Brake switch plausibility device): a non-programmable circuit that opens when the brake and accelerator pedal are pressed simultaneously.

- **AMS** (Accumulator Management System): switch opened by the BMS in case of internal errors.

- **Shutdown Buttons**: Mushroom safety buttons located in the cockpit and on both sides of the car.

- **TSMS** (Tractive system master switch): similar to the LVMS, the TSMS is used to arm the tractive system before powering it on.

Being a critical and complex part of the car, problems with the shutdown circuit are common. In Chimera Evoluzione, there was no easy way of knowing the state of the shutdown circuit, so troubleshooting was hard and took considerable time. For Fenice, the circuit has been improved by adding 16 feedback connections to the BMS. Feedbacks are inputs on the microcontroller that are directly connected to various parts of the shutdown circuit to aid troubleshooting. Depending on its internal state, the BMS expects a certain set of feedbacks to be high and another set to be low; If some feedback signal is in an unexpected state, an error is triggered and the error management system takes over control.

For example: if the BMS is in the *Idle* state, it expects the AIRs to be powered off. If the feedback for the AIR state is high an error is triggered, and if the problem persists for a set amount of time the pack goes into a critical error state.

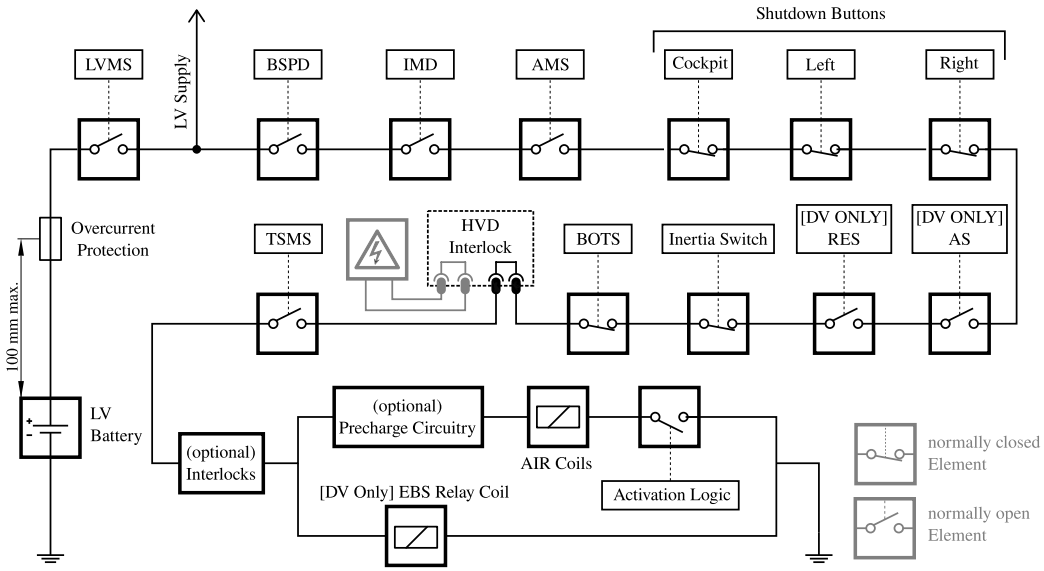

Figure 2.4: Shutdown circuit block diagram [1, EV 6.1.2]

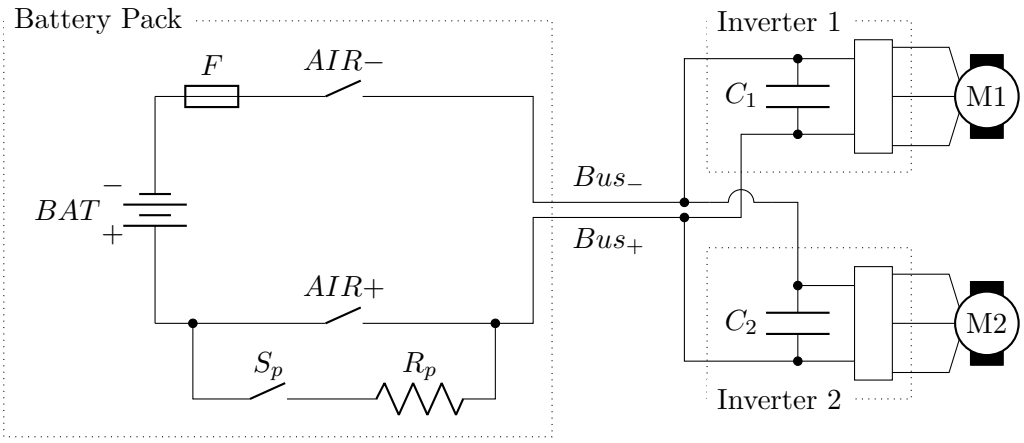

Figure 2.5: Tractive system schema highlighting the pre-charge circuitry

Pre-charge

If the power-on procedure of the pack simply involved putting both AIRs to their closed position, a potentially dangerous in-rush current would charge the capacitors on the inverters very quickly. To prevent over-current, the pre-charge procedure is performed [12]. Figure 2.5 shows the components of the pre-charge circuit.

First, the negative AIR and S_p are closed. The resulting circuit has R_p as the current-limiting resistor. After Bus and BAT voltages have equalized, the BMS commands the positive AIR to close. This completes the pre-charge procedure and the battery is now ready to be used.

2.2.2 Battery Management System

Battery management is a collection of operations that ensure the safety and efficiency of the battery pack. A basic battery management system should constantly measure cell temperatures and voltages along with the total pack current output and check that each of those values is within specification. If anomalies are detected, the battery should be disconnected immediately via the AIRs.

More advanced BMS implementations communicate their state of operation with other devices and turn the battery on or off according to external and internal signals.

Moreover, during battery charging, the BMS should handle the charge current curve to maximize charging speed while keeping the battery in a safe state. Cell balancing is also an important feature.

As the consequences of software errors on the BMS are potentially catastrophic, the code must

adhere to strict standards and should be tested thoroughly. Furthermore, reliable error management should be present to abide by the rulebook and ease troubleshooting.

Hardware

The need to measure a large number of voltages and temperatures scattered around the battery pack leads to a decentralized structure for the BMS components. Two types of logic boards are involved, namely the **Mainboard** and the **Cellboard**.

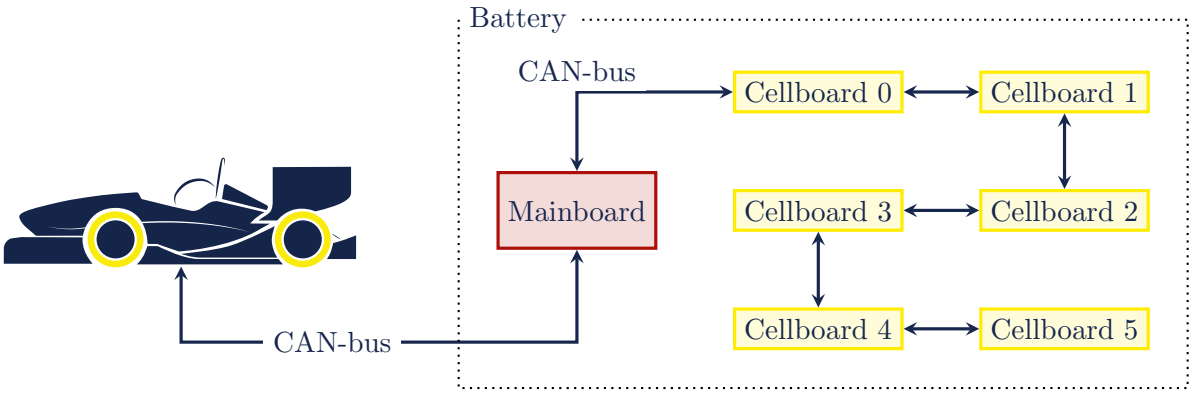

Figure 2.6: BMS hierarchy

Mainboard

The Mainboard is the central control unit of the BMS. It contains a microcontroller that handles two CAN-bus lines for internal and external communications, peripherals such as insulated ADCs, EEPROMs, serial ports, an SD-card and more. The mainboard is responsible for the actuation of the AIRs and contains the shutdown and pre-charge circuits. It also communicates voltages, temperatures, currents, battery status, warnings and errors to the rest of the car via CAN-bus. An integrated serial command-line interface and internal logging are included to help with troubleshooting.

Cellboard

Cellboards are dedicated to the measurement of module voltages and cell temperatures. The reading of voltages is handled by a specialized battery management chip that communicates via SPI to the onboard microcontroller. Temperatures are acquired with sensors glued to the busbars and are processed on board. All the data collected from the boards are sent via an internal CAN-bus to the mainboard.

Cellboards are also in control of cell balancing. They can activate and deactivate discharging for any module through the battery management chip, depending on commands from the mainboard and internal states.

The rulebook mandates that cellboards must be isolated from the mainboard; For this reason, the cellboards are self-powered by stepping down the battery segment's voltage. The communication is carried out with an isolated CAN-bus transceiver. Power for the insulated side of the transceiver coming from the mainboard also provides a power-on signal to the supply of the cellboard. In this way, if the mainboard is not powered, the cellboards don't draw current from the battery pack.

There are a total of six cellboards on Fenice's battery pack, one for every battery segment.

2.3 Issues and objectives

2.3.1 Battery Management

Work began by analyzing Chimera Evoluzione's BMS software, which ran on a slightly different architecture. While the software was functioning, it did not follow a consistent and modular structure and

thus was not easily maintainable. A complete rewrite gives the possibility of implementing a more advanced and robust software architecture that is more maintainable and supports future improvements and additions while providing a better separation of features.

2.3.2 Module Balancing

Cells are not perfectly identical and can have slight variations in internal resistance between each other. These differences mean that after some use, modules can start to deviate in voltage between one another [10]. If a cell has a higher voltage than average, the pack can only be charged to the voltage of said cell, while the remaining modules don't get fully charged. The same applies during discharge: if a module is more discharged than others, it constraints the maximum depth of discharge to when it reaches its cutoff voltage, while other cells are still above it and could theoretically be discharged further.

Chimera Evoluzione did not have the hardware required to mitigate the issue. This meant that cell voltages were slowly diverging, lowering usable capacity and affecting range. Cell balancing hardware is present on Fenice's BMS with some slight limitations.

The objective is to achieve the best cell balancing performance the hardware can offer.

2.3.3 Error Management

Error management is one of the more important features of a BMS. In Chimera Evoluzione errors were handled by hand with custom logic for every error type. Moreover, errors were only reported through the CAN network but were not logged internally or communicated through serial to a debugging computer. All these factors made on-track troubleshooting cumbersome and slow. Debugging issues were common amongst enclosed microcontrollers in Chimera Evoluzione. For this reason, significant effort was made towards the implementation of efficient and effective error management and communication system. A centralized framework should handle all error types without distinction and logging should be integrated as well as CAN communication. The framework should be testable with offline unit tests to validate performance and stability. With this solid base, the management system can then be used to log a large variety of errors, ranging from safety-critical errors to minor issues with peripherals, enhancing online and offline troubleshooting for every situation.

3 Cell Balancing

As discussed in the introduction, the problem of battery balancing can reduce net battery capacity. To realize the extent of the losses, Chimera Evoluzione's battery pack has been analyzed. Chimera Evoluzione did not feature cell balancing, but could still measure individual cell voltages and has been in use for one year without being manually balanced. Figure 3.1 shows measured open-circuit voltages of the whole battery pack.

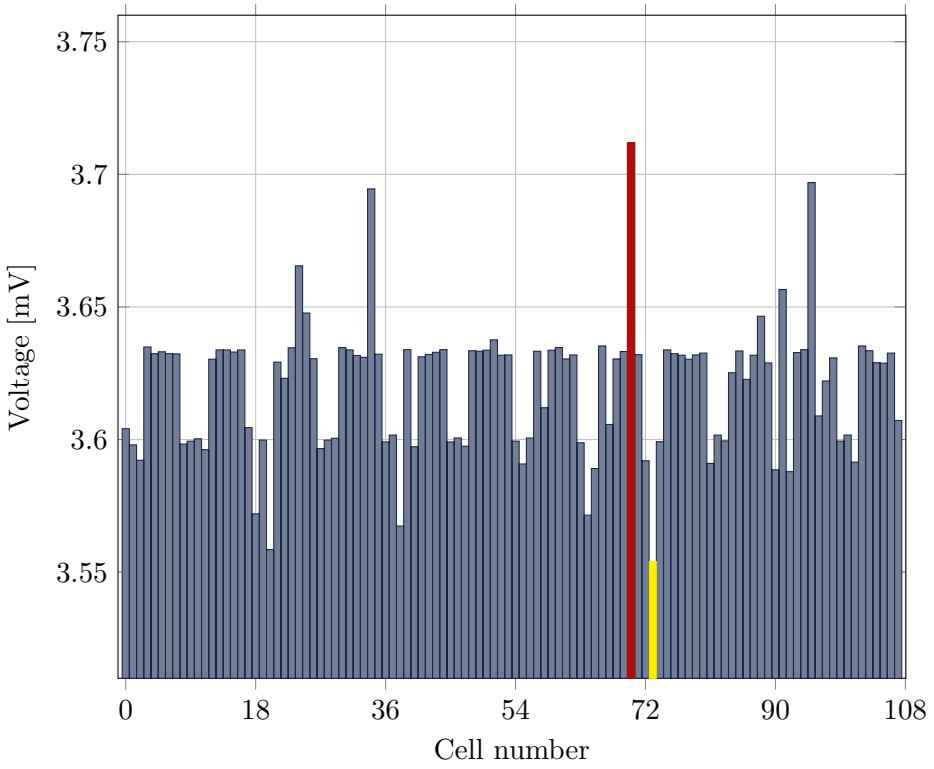

Figure 3.1: Unbalanced Chimera Evoluzione cells.

As can be seen in the chart, The difference between the highest-voltage cell ($V_{max} = V_{70} = 3.712V$) and the lowest-voltage cell ($V_{min} = V_{73} = 3.554V$) is $V_{delta} = V_{max} - V_{min} = 0.158V$. The difference between the average voltage and the minimum and maximum voltage is respectively $V^0_{delta} = V_{avg} - V_{min} = 0.066V$ and $V^1_{delta} = V_{max} - V_{avg} = 0.091V$.

If the pack in those conditions is discharged until V_{73} reaches the cut-off voltage of $V_{low} = 3.000V$, the average cell voltage would be $V^0_{avg} = V_{low} + V^0_{delta} = 3.066V$. Similarly, if the battery is charged until V_{70} reaches $V_{high} = 4.200V$, the average voltage could only reach $V^1_{avg} = V_{high} - V^1_{delta} = 4.109V$.

Chimera Evoluzione's battery pack is a 108s6p design, with each cell rated at 9.36 Wh nominal. Considering the discharge capacity curve for a single cell at 5 A Appendix A, the capacity lost between V_{high} and V^1_{avg} can be approximated to 0.2 Wh, or 2.1% of a cell's energy. The low-end capacity between V_{low} and $V^0_{avg} = 3.0V$ is close to 0.2 Wh as well. Assuming a total pack nominal energy capacity of $E_{total} = 6065.3Wh$, the energy loss amounts to $E_{lost} = E_{total} * 4.2\% = 255Wh$. Assuming a usable capacity of E_{total} during an endurance event the average energy consumption of the car should remain below $^{E_{total}}/_{22km} = 275^{Wh}/_{km}$ without a safety margin. If a 10% margin is considered, the maximum consumption is $C_{max} = 248^{Wh}/_{km}$. It can then be assumed that the cost of an unbalanced battery pack is $\Delta S = ^{E_{lost}}/_{C_{max}} = 1km$ of range in an endurance event.

With the use of cell balancing, it is expected to get imbalance values below 10 mV between all

cells, making losses irrelevant.

3.1 Balancing techniques

There are different methods to balance a battery pack; They can be summarized into passive balancing, energy transfer and individual charge control [10]. Passive balancing is the simplest technique to implement: it only needs a switched shunt resistor in parallel to each battery module that dissipates the excess energy from each cell. The main disadvantages are the great heat generation that requires active cooling and the reduced overall efficiency of the battery pack that loses some of its energy through the balancing process.

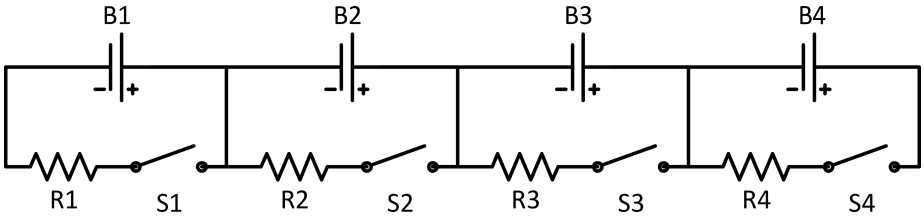

Figure 3.2: Passive balancing wiring schema

More efficient techniques exist under the name of active balancing. As the name suggests, excess energy is not wasted using these methods. Energy transfer is a balancing strategy that can move energy between cells of the pack. For this to work, every cell must be interconnected with each other with some circuitry. One of the simplest transfer circuits is made of capacitors that can be alternately connected in parallel to two adjacent cells. Both poles of each capacitor can be switched at the same time to be connected to either one of the two cells, charging the capacitor from one cell and discharging it on the other, using the capacitor as a bucket to transfer energy between the cells. This is the simplest active balancing method, but its simplicity has the disadvantage of being slow to transfer energy between distant cells [10]. More capacitor tiers can be added to provide more transfer paths at the cost of scalability.

Other systems with increased complexity involve the use of transformers, buck-boost converters or more complicated methods to transfer energy between each cell with more flexibility.

In the use case of a Formula SAE car, where the battery goes through full charge and discharge cycles, the disadvantages of a passive solution are less relevant, while the increased complexity of active balancing results in more weight and failure points. If done only during charging, passive balancing doesn't affect the running efficiency of the car, as it doesn't discharge cells when the car is running. Active balancing could provide efficiency advantages if the battery doesn't get fully charged often, as happens in road cars. For these reasons, a passive balancing design was chosen.

More advanced balancing algorithms base the imbalance computation on the state of charge of the cell instead of cell voltage [9]. This preliminary study focuses on voltage values as a reference, since the state of charge estimation is not yet implemented in the BMS.

The balancing hardware is located on the cellboards and is composed of the battery management chip that drives external transistors which can connect each battery module in parallel with a power resistor. The resulting circuit discharges the module at a rate of $400mA$ [4].

3.1.1 Strategy

A design constraint in the balancing circuitry on the cellboards prevents two adjacent cells to be discharged simultaneously. To work around this problem an algorithm has been developed to select the optimal set of cells that can be discharged at the same time.

When enabled, the cell balancing algorithm runs periodically following the state machine shown in Figure 3.3: first, the algorithm computes the set of cells to discharge (*Comp*); The balancing ICs on the cellboards are then instructed to start the discharge process on selected cells. The discharge period lasts 30 seconds at the end of which there is a cooldown period of 10 seconds, which lets the voltages stabilize and the discharge resistors cool down. After cooldown, the algorithm is executed again. If no cells need discharging anymore, the state machine goes back to the *Off* state. The balancing state

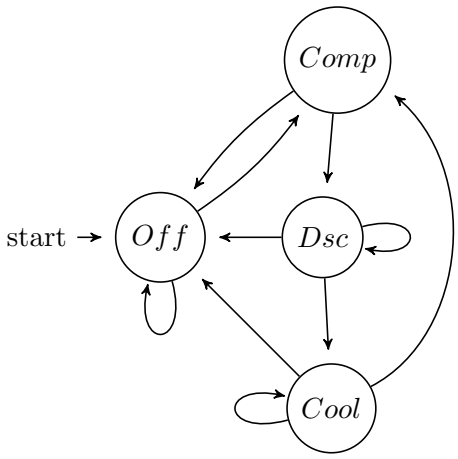

Figure 3.3: Balancing state machine

machine can be stopped at all times by external events.

3.2 Algorithm

The algorithm is made of two parts: the first computes the amount of imbalance of each cell compared to the minimum voltage, while the last selects the optimal combination of cells that can be discharged continuously.

3.2.1 Imbalance computation

The imbalance computation algorithm assigns an integer value $I[i]$ to each cell that equals the difference between the ith cell and the lowest voltage cell plus a threshold. Since passive balancing can only discharge the batteries, the minimum voltage in the pack is the target voltage.

$$I[i] = voltages[i] - (min(voltages) + threshold)$$

int[ ] imbalance(**int**[ ] *voltages*, **int** n, **int** *threshold*)

$I =$ **int**$[0 \dots n-1]$
$min\_voltage = \mathsf{min}(voltages)$
for $i = 0 \rightarrow\ n$ **do**
  $I[i] = \mathsf{max}(0, voltages[i] - (min\_voltage + threshold))$
return I

The practical implementation cuts the imbalance to values greater than zero, as cells with negative imbalance don't need to be discharged.

3.2.2 Cell Selection

The hardware limitation discussed above poses the interesting challenge of finding the optimal combination of compatible cells that need to be discharged. The optimal combination is the one that discharges all the overcharged cells in the least amount of cycles, avoiding neighboring cells in the same discharge cycle. The problem is similar to a canonical optimization problem that can be solved efficiently with Dynamic Programming [2].

The problem can be formulated as follows:

Given a vector of imbalances $I[]$ of size n, return the subset of non-adjacent cells (with positive imbalance) that maximizes total imbalance.

By using the imbalance of each cell as a priority value, the algorithm prefers highly unbalanced cells.

Let $Cells[i]$ be a subset of the first $i \leq n$ indexes that respect the above requirements. Given this definition, $Cells[n]$ is the solution to the problem.

Recursive step

First, only cells with a positive imbalance ($I[i] > 0$) are considered. For these cells, two options are available:

- if i is selected:
$$Cells[i] = \{i\} \cup Cells[i-2]$$
In this case, cell $i-1$ is a neighbor of i and has to be discarded, thus $i-2$ is the closest cell that can be picked.
- if i is skipped, then $Cells[i] = Cells[i-1]$

The choice to select i or not is made by comparing the resulting set for both options and picking the one with higher imbalance:

$$Cells[i] = highest(Cells[i-1], \{i\} \cup Cells[i-2]) \tag{3.1}$$

highest is a function that sums the imbalance of the two sets and returns the set with the highest value.

Base cases

The recursion is completed by the introduction of two base cases:

$$Cells[0] = \emptyset \tag{3.2}$$

$$Cells[1] = \begin{cases} \emptyset & I[i-1] \leq 0 \\ \{0\} & I[i-1] > 0 \end{cases} \tag{3.3}$$

Recursive function

The complete solution can be expressed with the following recursive equation:

$$Cells[i] = \begin{cases} \emptyset & i = 0 \text{ or } (i = 1,\ I[i-1] < 0) \\ \{i-1\} & i = 1,\ I[i-1] > 0 \\ Cells[i-1] & i \geq 2,\ I[i-1] \leq 0 \\ highest(Cells[i-1], \{i\} \cup Cells[i-2]) & i \geq 2,\ I[i-1] > 0 \end{cases}$$

3.2.3 Implementation

A very efficient approach to this problem is to find and save in a helper array ($C[]$) the maximum sum of imbalances for all subsets of n and compatible cells, and then reconstruct the solution by walking back from it. This variation benefits from not relying on complex data structures such as sets, and it reduces computational complexity by not needing the custom *highest* function for each cell that could result in an algorithm with quadratic computational complexity.

Computing the maximum

Let $C[i]$ be the maximum total imbalance from compatible cells that can be obtained with the first i cells.

$C[n]$ is the solution to the problem.

Recursive Step For each cell i, two options can be considered:

1. If cell i is discarded, cell $i-1$ can be selected:
$$C[i] = C[i-1]$$

2. If cell i is selected, cell $i-1$ has to be discarded, but $i-2$ can be selected:

$$C[i] = I[i-1] + C[i-2]$$

To maximize the total imbalance, the highest of the two possibilities is chosen:

$$C[i] = \max(C[i-1],\ I[i-1] + C[i-2])$$

By adding the base cases, the complete recursion can be defined as follows:

$$C[i] = \begin{cases} 0 & i = 0 \\ \max(0, I[0]) & i = 1 \\ \max(C[i-1], C[i-2] + I[i-1]) & i \geq 2 \end{cases}$$

3.2.4 Pseudo-code

Since *imbalance*() returns an array of non-negative values, there is no need to check if a certain imbalance is negative. For this reason the *exclude*() function does not theck the values of $I[]$.

```
int exclude (int[] I, int n)
    int[] C = int[0...n+1]
    C[0] = 0
    C[1] = I[0]
    for i = 2 → n+1 do
        C[i] = max(C[i-1], C[i-2] + I[i-1])
    return C[n]
```

Reconstructing the set

The *exclude()* function describes a correct albeit partial solution to the problem since the set of cells to discharge is not returned. However, as discussed before, the set of selected cells can be reconstructed by analyzing the $C[]$ array in a top-down fashion.

The *solution()* recursive function starts from the $C[]$ array returned by *exclude()* and finds the cells that have been selected by comparing the imbalance of each cell with the relative increase in imbalance from the previous elements of the $C[]$ array. For example, if $C[i]$ is equals to $C[i-1]$ it can be assumed that cell $i-1$ has not been selected, otherwise $C[i]$ would be greater than $C[i-1]$. In this case the output set remains unchanged and the function is called with a decremented i.

If the two values are different the only possibility is for cell $i-1$ to have been selected, so $\{i-1\}$ is inserted in the output set and *solution()* is called again with i decremented by 2. However, if i equals to 1 it cannot be decremented by 2 as it will go negative. For this particular case the set $\{i-1\}$ is returned.

When i reaches 0, the empty set is returned, as a starting point for the stack of pending function calls that will fill it.

```
set solution (int[] D, int i)
    if i == 0 then
        return {∅}
    else if D[i] == D[i-1] then
        return solution(D, i-1)
    else
        if i > 1 then
            return solution(D, i-2) ∪ {i-1}
        return {i-1}
```

Finally, a wrapper function *balance()* calls all the above routines in the right order and with the

right parameters. The variable *seg* defines how many segments the pack is made of. In Fenice, this number corresponds to the number of cellboards.

While *solution()* could simply be called once, this distinction is necessary to handle the case in which two imbalanced neighboring cells that are managed by two different cellboards need to be discharged. Although they are neighbors, these two cells are not affected by the hardware limitation of the cellboards, as they are not sharing the same balancing circuit. By calling *solution()* for each subset of cells this property is respected.

```
set balance (int[] voltages, int n, int seg, int threshold)
    int[] I = imbalance(voltages, threshold)
    if I = ∅ then
        return ∅
    exclude(I, n)
    sol = ∅
    for i = 0 → seg do
        sol = sol ∪ solution(I[i * (n/seg)], n/seg)
    return sol
```

Complexity

It's easy to see that *imbalance()*, *exclude()* and *solution()* are all belonging to the linear time complexity set $\Theta(n)$. Even if *solution()* is called *seg* times, the size of data is also smaller by the same value, and with *seg* being a constant, the function still has a complexity of $\Theta(n)$.

$$\sum_{i=0}^{seg} \Theta(n/seg) = \Theta(n)$$

The complete algorithm inherits the complexity from its components, thus can be categorized as $\Theta(n)$ as well.

Distribution

The modular nature of the algorithm enables the possibility of running different parts in separate systems. The mainboard is the only device that has track of all the voltages together, so it's the only one that can run *imbalance()*. The list of imbalanced cells is then sent to the cellboards which execute the remaining *exclude()* and *solution()* functions for their subset of cells.

4 Error Management

The rulebook puts lots of emphasis on the safety of the tractive system and tries to regulate every safety-concerning aspect of it. The battery management system is one of the safety components strictly regulated by the rules. To better meet the imposed requirements, a complete and detailed error management system has been built. In previous cars, BMS-related errors were hard to understand and troubleshoot. Significant effort has been made to make errors more specific and easier to comprehend.

Despite the decentralized nature of the BMS, error management is centralized in the mainboard. Errors triggered by the cellboards are transmitted back to the mainboard via CAN-bus and are then handled like any other error.

4.1 Error Definition

An error has a *state* property that can assume two values: *warning* or *fatal*. The *fatal* state indicates a safety-critical problem in some parts of the battery that needs immediate action. When a fatal error occurs, the battery must disconnect itself from the high voltage bus by opening the two AIRs.

The rules mandate that after a fatal error, the shutdown circuit should stay latched in the off state and can only be restored by a reset switch. A *warning* is not a critical issue but can become one overtime after a delay.

For example, an error in communicating with the cellboards starts off as a warning, since a single communication failure is not considered critical. After some time, if the communication is not reestablished, the warning becomes a fatal error that needs to trigger the shutdown procedure.

However, not all warnings can turn into errors: A minor issue such as a failure in the communication with the on-board EEPROM is not a safety concern, thus it is only useful as a warning of some non-critical malfunction in the system.

An error instance is defined by a data structure called *error_t*. The tuple composed of *type* and *index* parameters uniquely identifies an error instance. The *type* defines the error's typology from a list of possible errors: it identifies the type of problem.

```
typedef struct {
        bool active;
        error_type type;
        uint8_t index;
        error_state state;
        uint32_t timestamp;
} error_t;
```

Listing 1: *error_t* struct

The *index* variable is mainly used to distinguish errors of the same type if more than one instance can happen simultaneously. It is used on errors that involve cellboards or single cells, such as voltage and temperature errors. For example, if cells number 12 and 63 are in under-voltage, then two errors with the tuple $\{type = \texttt{CELL\_UNDER\_VOLTAGE}, index = 12\}$ and $\{type = \texttt{CELL\_UNDER\_VOLTAGE}, index = 63\}$ will be activated.

The *state* parameter indicates whether the instance is an error or a warning and *timestamp* stores the timestamp at which the error occurred.

4.1.1 Types of errors

The BMS error manager handles the following errors:

- `CELL_LOW_VOLTAGE`: A cell's voltage has dropped below the `LOW_VOLTAGE` threshold.

- `CELL_UNDER_VOLTAGE`: A cell's voltage has dropped below the `MIN_VOLTAGE` threshold.
- `CELL_OVER_VOLTAGE`: A cell's voltage has risen over the `MAX_VOLTAGE` threshold.
- `CELL_HIGH_TEMPERATURE`: A cell's temperature is over the `HIGH_TEMPERATURE` value.
- `CELL_OVER_TEMPERATURE`: A cell's temperature has reached the `MAX_TEMPERATURE` value.
- `OVER_CURRENT`: The current delivered by the pack exceeded a specified limit.
- `CAN`: There was an error in the CAN-bus communication to the car.
- `ADC_INIT`: The on-board ADC that measures pack and bus voltages has not been initialized successfully.
- `ADC_TIMEOUT`: Communication efforts with the ADC resulted in a timeout.
- `INT_VOLTAGE_MISMATCH`: The internal voltage measured with the ADC and the voltage resulted by the sum of all the cell's voltages differs by a significant amount.
- `CELLBOARD_COMMUNICATION`: The CAN-bus communication with a cellboard was not possible.
- `CELLBOARD_INTERNAL`: Internal error in a cellboard.
- `FEEDBACK`: One or more feedbacks for the shutdown circuit did not match the expected state.

4.2 Timeouts

> The BMS must switch off the TS via the shutdown circuit if critical voltage, temperature or current values [...] persistently occurs for more than:
>
> - 500 ms for voltage and current values
> - 1 s for temperature values
>
> ([1, EV 5.8.6])

Rule EV 5.8.6 cited above generally states that the delay in the power-off reaction to an error condition is deemed safe. This information can be helpful to the BMS, as it gives margin to the enforcement of error conditions. However, to take advantage of this possibility, the error management would have to keep track of the trigger and expiration time of every error. To ensure timekeeping reliability, hardware timers are used to handle the expiration logic.

4.2.1 Timekeeping

Every error type has an associated timeout delay variable, in the form of an array, that for each type, has a time value defined by the rules. If the error can not be fatal, its timeout is set to a magic number that is ignored by error management functions.

```
const uint32_t timeouts[ERROR_COUNT]={
        [ERROR_CELL_LOW_VOLTAGE]      = UINT32_MAX,
        [ERROR_CELL_UNDER_VOLTAGE]    = 500,
        [ERROR_CELL_OVER_VOLTAGE]     = 500,
        [ERROR_CELL_HIGH_TEMPERATURE] = UINT32_MAX,
        [ERROR_CELL_OVER_TEMPERATURE] = 1000,
        [ERROR_OVER_CURRENT]          = 500,
        [ERROR_CAN]                   = 500,
        [ERROR_ADC_INIT]              = UINT32_MAX,
        [ERROR_ADC_TIMEOUT]           = UINT32_MAX,
        [ERROR_INT_VOLTAGE_MISMATCH]  = UINT32_MAX,
        [ERROR_CELLBOARD_COMM]        = 500,
```

```
        [ERROR_CELLBOARD_INTERNAL]      = 500,
        [ERROR_FEEDBACK]                = UINT32_MAX
};
```

When an error occurs, a hardware timer is set to trigger at the expiration time of the error, which equals $now + timeouts[error\_type]$. If the error is unset before the timer triggers, the timer is stopped and the error never becomes fatal. If the timer triggers, the TS is immediately shut off, as mandated by [1, EV 5.8.6].

The above explanation doesn't consider that the amount of hardware timers on a microcontroller is limited and is lower than the number of errors that can occur simultaneously. To overcome this issue a scheduling algorithm can be employed to reduce the number of used timers while keeping track of every error.

4.2.2 Timer Scheduling

All active errors can be managed by only one timer because, at any given time, the only error to keep track of is the one that expires before any other error. To select the sooner-to-expire error, all errors are organized in a priority queue sorted by expiration time. The hardware timer always triggers at the expiration time of the first item in the queue. If a new error is inserted in the queue and comes on top, the timer is reset to its expiration time. Otherwise, if the head error gets removed, the timer is reset to the expiration time of the second error. If the queue is empty the timer is stopped.

The example in Figure 4.1 shows the behavior of the scheduler when two errors overlap in time. At T+0 the timer is not running as there are no active errors. At time s_0, E_0 activates and the timer is set to trigger at time $e_0 = s_0 + timeouts[E_0.type] = 1100ms$.

200ms after s_0, E_1 activates. At this point, the scheduler has to decide which error to track. The choice is made by picking the smallest time e_x at which error x expires. In the example, e_1 is smaller than e_0, so the timer is set to trigger at $e_1 = s_1 + timeouts[E_1.type] = 800ms$. After 600ms, E_1 is deactivated. Since the timer is now tracking a deactivated error, an update must be made to the scheduler. The choice now falls on the only active error: E_0, so the timer is set back to expire at e_0. Finally, E_0 remains active for the whole duration of its timeout period and the timer triggers at e_0, shutting the battery pack down.

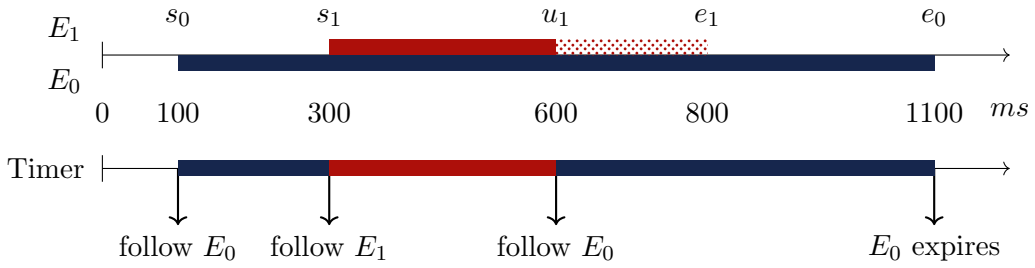

Figure 4.1: Error scheduling example

4.3 Implementation

Instances of active errors are stored in a double-linked list [3] in which elements are sorted by expiration time. The error management library declares all the possible error pointers as arrays that can be used externally to the linked list to find whether a specific error is enabled without searching the linked list itself. This gives a significant advantage since the most performed operation during the normal execution of the library is the check of the activation state of an error.

The insertion and removal of the items from the list have a time complexity of $O(n)$ in the case the error has the lowest priority of all. The higher time complexity gets compensated by the size of the priority queue, which is expected to be in the order of tens of elements at most.

The priority queue is provided with a comparator function 2 that is used to sort the errors by expiration time. The *delta* variables contain the amount of time after the given error expires.

```
int8_t error_compare(error_t a, error_t b) {
    uint32_t a_delta = a->timestamp + error_timeouts[a->id] - HAL_GetTick();
    uint32_t b_delta = b->timestamp + error_timeouts[b->id] - HAL_GetTick();

    if(a_delta < b_delta) return 1;
    if(a_delta == b_delta) return 0;
    return -1;
}
```

Listing 2: *error_compare()* function

4.3.1 Error setting and resetting

To abstract the management of error scheduling, two functions are exposed that handle error setting and resetting. *error_set()* is called whenever an error is detected. A timestamp parameter is requested to indicate the exact error occurrence time; The timestamp can be used for errors that have a delayed trigger signal: for example, an error triggered on a cellboard must travel on the CAN-bus and be processed by the mainboard before *error_set()* is called. If the cellboard and mainboard's timestamps are synchronized and the occurrence timestamp is provided along with the error message, the transmission delay can be calculated and subtracted from the error timestamp.

```
bool error_set(error_id id, uint8_t offset, uint32_t timestamp) {
    error_t *error = ERROR_GET(id, offset); // Get error from external pointers set
    if (!error->active) {
        error->active = true;
        error->timestamp = timestamp;

        if (llist_insert_priority(er_list, error) != LLIST_SUCCESS) {
            return false;
        }

        if (error_equals(llist_get_head(er_list), error)) {
            error_set_timer(error);
        }
    }
    return true;
}
```

Listing 3: *error_set()* function

When *error_set()* is called and the provided error is already active, the function exits immediately. Otherwise, the error is activated and its timestamp updated. Then the error is inserted in the priority queue; If the error comes up first in the queue, the timer is reset to track it.

Whenever an error is not happening *error_reset()* is called. If the error is not active the function exits without doing time consuming operations. If the error is active and is the first in the priority queue, then the timer is reset to track the second error in the queue. Finally, the error is deactivated and removed from the queue.

4.3.2 Scheduling

Timer scheduling logic is done inside *error_set()* and *error_reset()* by calling *error_set_timer()* which sets the hardware timer up. Initially, the timer is stopped, then if a *NULL*-valued error is given to the function, or the error does not expire, the timer gets disabled. Otherwise, the timer's output compare register is set to the desired value. The output compare register (*CCR1*) is checked against the timer's counter (*CNT*) and when the two matches, an interrupt is triggered [8]. The timer's prescaler is set so that the counter increases every $100\mu s$.

```
bool error_reset(error_id id, uint8_t offset) {
    error_t *error = ERROR_GET(id, offset);
    if (error->active) {
        // If the first error is being removed
        if (error_equals(llist_get_head(er_list), error)) {
            // reset the timer to the second error
            error_t *tmp = NULL;

            llist_get(er_list, 1, (llist_node *)&tmp);
            error_set_timer(tmp);
        }

        if (llist_remove_by_node(er_list, error) != LLIST_SUCCESS) {
            return false;
        }

        error.active = false;
        return true;
    }
    return false;
}
```

Listing 4: *error_reset()* function

4.3.3 Logging

The centralized approach to error management is ideal for logging data. The BMS has two means of logging error activity.

Internal

Internal logging is done to a Micro SD card located on the mainboard. The SD card is formatted in the FAT32 filesystem so that logs can be easily organized and read from a computer.

```
[timestamp] Subsystem:
[123.456] Error: ERROR_CELL_LOW_VOLTAGE index: 45 activated
[130.123] Error: ERROR_CELL_UNDER_VOLTAGE index: 45 activated
[130.623] Error: ERROR_CELL_UNDER_VOLTAGE index: 45 expired
[130.623] FSM: transitioning to BMS_STATE_HALT
[145.000] Error: ERROR_CELL_UNDER_VOLTAGE index: 45 deactivated
[150.012] Error: ERROR_CELL_LOW_VOLTAGE index: 45 deactivated
[150.012] FSM: transitioning to: BMS_STATE_IDLE
```

Listing 6: Example log

When the BMS starts a new text file is created. Inside the file every line represents an event. Every event starts with the timestamp followed by the subsystem that generated the error. As can be seen in Listing 6, the logging functionality is used in multiple subsystems of the BMS. In the example, an undervoltage situation is shown. At first cell 45 drops under the `LOW_VOLTAGE` threshold and activates the corresponding warning. Seven seconds later cell 45 drops below `UNDERVOLTAGE` and triggers the `CELL_UNDER_VOLTAGE` error. Being a fatal error, this situation starts the error timer that interrupts after 500ms and triggers a battery shutdown, transitioning the FSM to the *Halt* state. After the load to the battery has been disconnected, the voltage starts to rise and the errors get deactivated. After the last error is inactive, the FSM transitions to the *Idle* state and is ready to accept commands again.

```
bool error_set_timer(error_t *error) {
    // Stop the timer
    HAL_TIM_Base_Stop_IT(&htim_err);

    if (error != NULL && error->state == STATE_WARNING && error_timeouts[error->id] < SOFT) {
        // Increase the output compare counter period register
        htim_err.Instance->CCR1 += get_timeout_delta(error) * 10;

        // Start the timer
        HAL_TIM_Base_Start_IT(&htim_err);
        // Enable output compare signal generation
        HAL_TIM_OC_Start_IT(&htim_err, TIM_CHANNEL_1);

        return true;
    }

    return false;
}
```

Listing 5: *error_set_timer* function

The plain text format shown in 6 was devised to be easily readable and scriptable. It is common to all the boards so that standardized tools can be used to filter or plot data.

External

External logging is done by the telemetry system via the CAN-bus. Every time an error is set or reset a series of CAN messages containing error information is sent to the bus. Together with the telemetry system, the steering wheel interface intercepts the error messages and shows error information directly to the pilot and to track engineers.

The first message encodes error types as a bitset and is used to communicate the general state of errors on the BMS. If all errors are inactive, no further messages are sent. If some errors are active, a message containing the error type and index is sent for each active error. This completes the information about the error and gives the telemetry system useful details about the problem.

5 Experimental results

Production delays made full-scale testing of the battery infeasible. However, the main features of the BMS have been tested separately and encouraging data has been collected.

5.1 Balancing

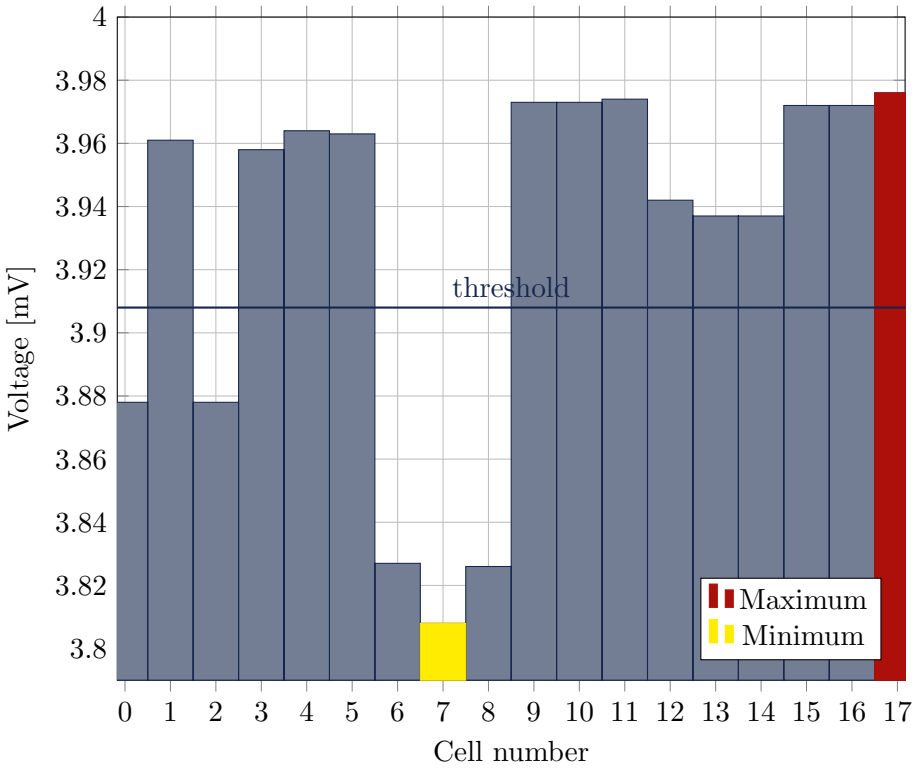

Figure 5.1: Unbalanced Fenice segment

To test the balancing algorithm a battery pack segment was connected to a cellboard. Every segment cell was charged individually to different voltages to create an imbalanced situation. Figure 5.1 shows the initial state of the segment with a total imbalance of $V_{17} - V_7 = 3.976V - 3.808V = 0.168V$. For this test, the balancing settings are set to have a maximum imbalance of $100mV$. With these settings all cells with a voltage above $V_7 + 100mV = 3.908V$ need to be discharged, in this case cells 1, 3, 4, 5, 9, 10, 11, 12, 13, 14, 15, 16 and 17 are all above the threshold voltage.

The test setup only involved a single cellboard, disconnected from the mainboard. The balancing algorithm and state machine ran on the cellboard only relying on its internal voltage measurements as input. A serial console was connected to a computer for monitoring and data gathering.

After starting the state machine the algorithm correctly computed the cells to balance and proceeded to the discharge state. The serial console showed the cells that were being discharged and after the predefined 30 seconds of discharge, the state machine went into the cooldown state. After 10 seconds a new set of cells was computed and balancing began again.

5.2 Error management

Unit testing was used to validate the priority queue part of the system. The timer scheduling part has been tested using a logic analyzer to verify the scheduling and timing accuracy.

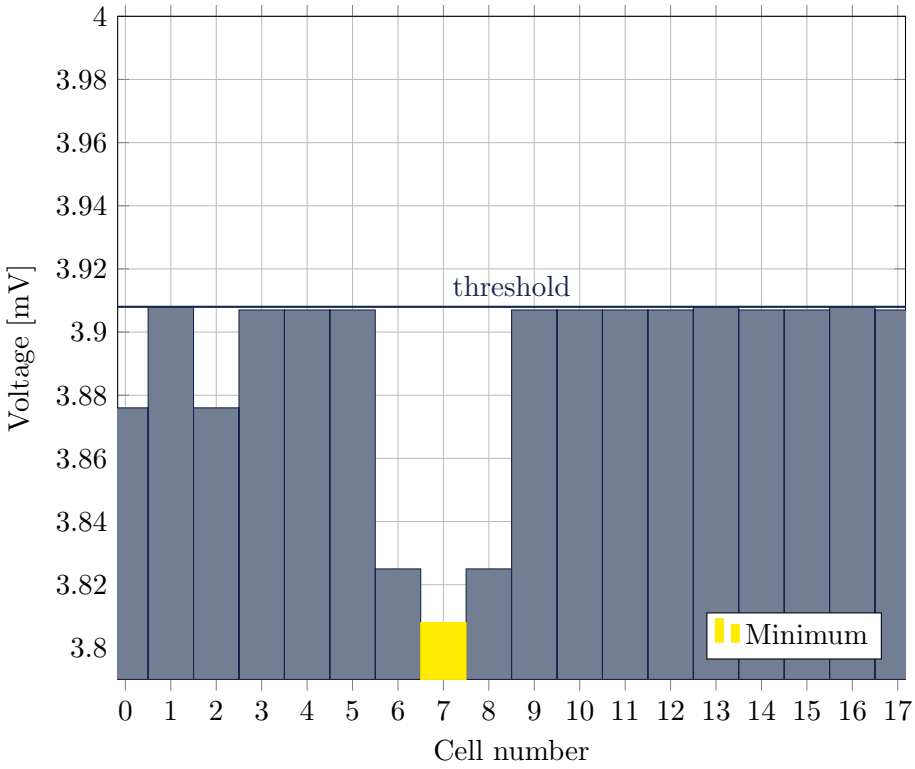

Figure 5.2: Fenice segment after balancing

In Figure 5.3 test results can be seen. The test scenario is analog to the one presented in Figure 4.1 in which two errors happen within the same timeframe. The two small drops of the *Timer* channel signal when the timer switches from following E_0 to E_1 and vice versa. When the timer triggers the `Fatal` signal can be seen going high.

The timing shown in was very consistent across multiple tests albeit being higher than expected. Given the high repeatability of the unwanted extra time, all timeout values have been shortened to remain within the rulebook specification.

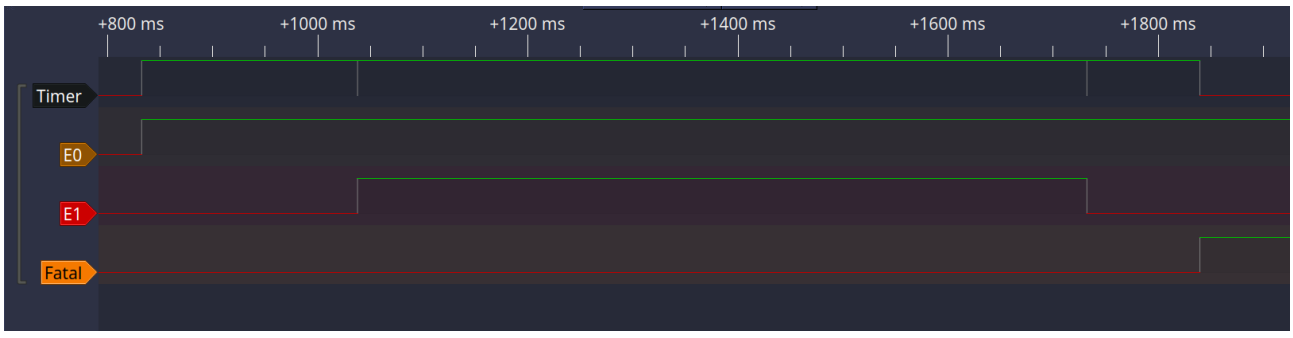

Figure 5.3: Error timing test. E_1 is reset before triggering, and the timer switches back to E_0.

6 Conclusions and future works

6.1 Cell Balancing

After one hour of balancing the voltages on the segment were converging to the wanted maximum value of 3.908V. In the following hour, the algorithm was able to balance the pack to the situation seen in Figure 5.2 the state machine went into the *Off* state as expected. More tests are being conducted with different balancing thresholds and discharging periods to assess the reliability of the algorithm.

Potentially harmful behavior can happen if, inside the discharging period, a cell's voltage drops below the minimum voltage cell. This occurrence can start a loop that alternately discharges all cells until undervoltage protections kick in. To avoid this problem short discharging periods are preferred along with not using too small values for the threshold. A future solution would monitor voltages continuously and stop the discharge when the voltage falls inside the threshold, without risking over-discharge the cell.

6.2 Error Management

Overall, the system works as expected and it already helped in troubleshooting while developing the firmware. There are concerns about the dynamic nature of the linked list library used as a data store. While the library is well tested and has proven to work as designed, it is generally agreed that in safety-critical applications static allocation is preferred.

There are plans to convert the error management system to a static storage solution by creating arrays of error instances that cover all the possible errors. There is a minor concern about the memory occupied by this solution since all possible errors are always allocated in RAM. However, given the characteristics of the mainboard's microcontroller [7], RAM is plenty for this application and no problems are expected with the current amount of errors. In any case, if memory issues arise they will be noticed at compile-time, since the whole codebase is statically allocated, so catastrophic failures at runtime are prevented.

6.3 State of charge estimation

Work is underway to implement a state of charge (SoC) estimation algorithm based around cell open-circuit voltage estimation [5]. An important component of many commercial battery management systems is a state of charge estimation. This feature tries to guess the amount of remaining energy in each battery cell by analyzing the load current applied to the battery and its voltage to estimate the internal resistance at each instant. If the internal resistance is known, the open-circuit voltage (OCV) of the cell can be calculated using Ohm's law. With this data, SoC can be easily estimated by comparing the OCV to a voltage vs. energy function to get the remaining energy in the cell. The function can be found on the manufacturer's datasheet or can be measured with a controlled discharge of single cells.

This complex feature will be the basis of other advanced functionalities such as range estimation that can prove useful during the endurance event and power limits on lower values of SoC to prevent undervoltages and keep the battery running as long as possible.

Bibliography

[1] Formula student germany 2020 rules v1.0. https://www.formulastudent.de/fileadmin/user_upload/all/2020/rules/FS-Rules_2020_V1.0.pdf, 2020.

[2] Montresor A. Bertossi A. *Algoritmi e Strutture di Dati*. Città Studi Edizioni, third edition, 2014.

[3] Ruffini S. Bonora M. Dynamic double linked list implementation in c. https://github.com/eagletrt/micro-libs/tree/master/llist.

[4] Ruffini S. Bonora M. Fenice's high voltage battey management system. https://github.com/eagletrt/fenice-bms-hv.

[5] Chang Yoon Chun, Gab-Su Seo, Sung Hyun Yoon, and Bo-Hyung Cho. State-of-charge estimation for lithium-ion battery pack using reconstructed open-circuit-voltage curve. In *2014 International Power Electronics Conference (IPEC-Hiroshima 2014 - ECCE ASIA)*, pages 2272–2276, 2014.

[6] lygte info.dk. Sony us18650vtc5 review. https://lygte-info.dk/review/batteries2012/Sony%20US18650VTC5%202600mAh%20%28Green%29%20UK.html.

[7] ST Microelectronics. Stm32f446re datasheet. https://www.st.com/resource/en/datasheet/stm32f446re.pdf, 2021.

[8] ST Microelectronics. Stm32f446re reference manual. https://www.st.com/resource/en/reference_manual/rm0390-stm32f446xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf, 2021.

[9] Zachary Bosire Omariba, Lijun Zhang, and Dongbai Sun. Review of battery cell balancing methodologies for optimizing battery pack performance in electric vehicles. *IEEE Access*, 7:129335–129352, 2019.

[10] Jian Qi and Dylan Dah-Chuan Lu. Review of battery cell balancing techniques. In *2014 Australasian Universities Power Engineering Conference (AUPEC)*, pages 1–6, 2014.

[11] Samsung SDI. Samsung inr21700-40t datasheet. https://www.dnkpower.com/wp-content/uploads/2019/02/SAMSUNG-INR21700-40T-Datasheet.pdf, 12 2020.

[12] Sensata Technlogogies. *How to Design a Precharge Circuit for Hybrid and Electric Vehicle Applications*.

Appendix A Discharge energy

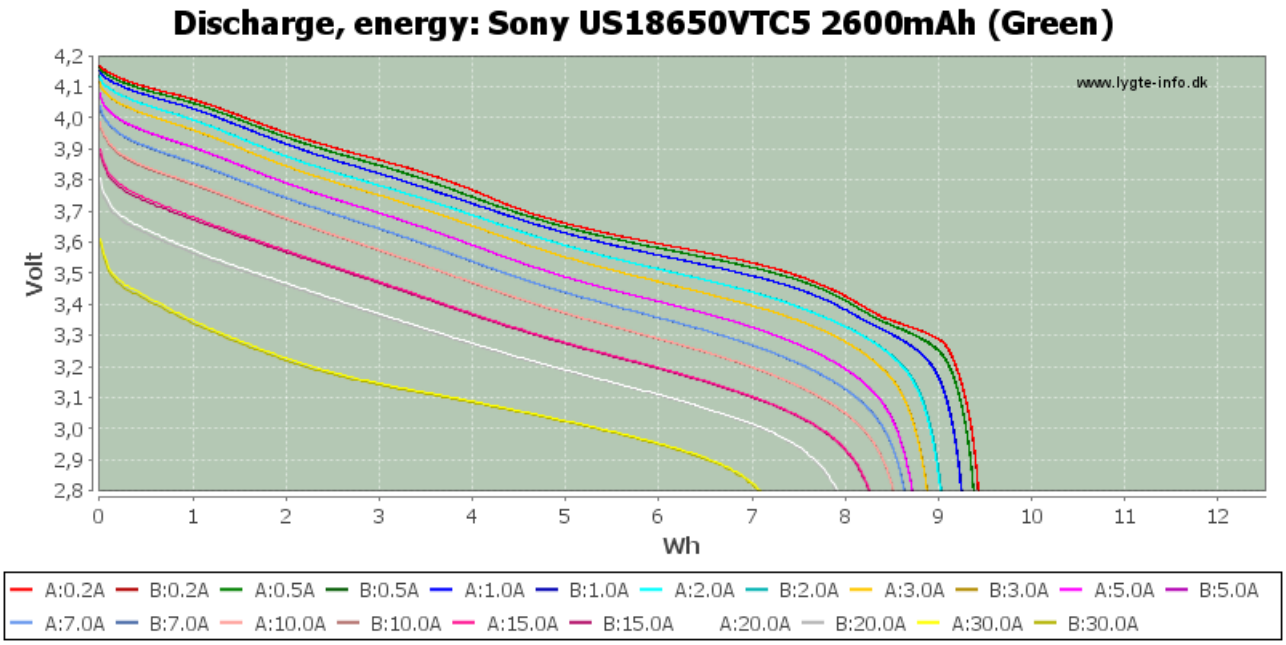

Figure A.1: Discharge voltage vs. energy for Sony US18650VTC5 cells [6]

Appendix B Balancing test

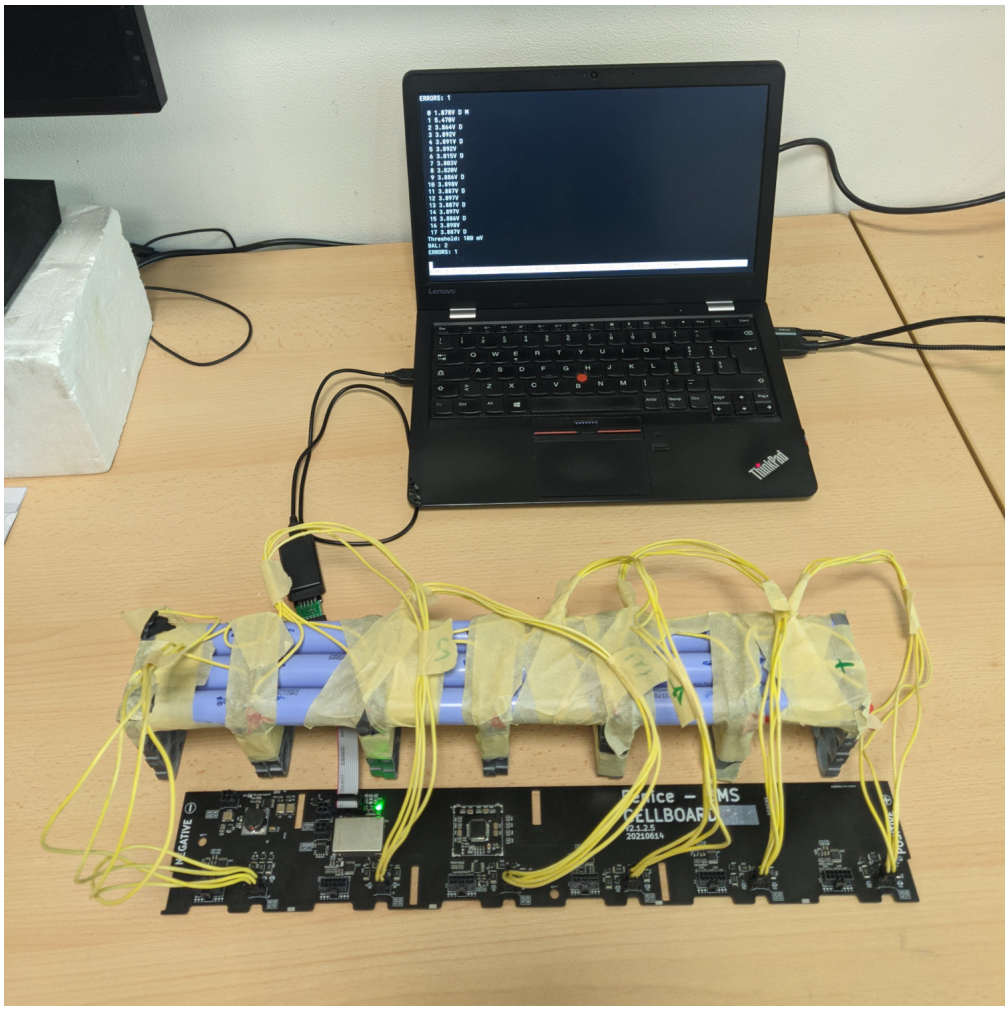

Figure B.1: Cell balancing test setup.

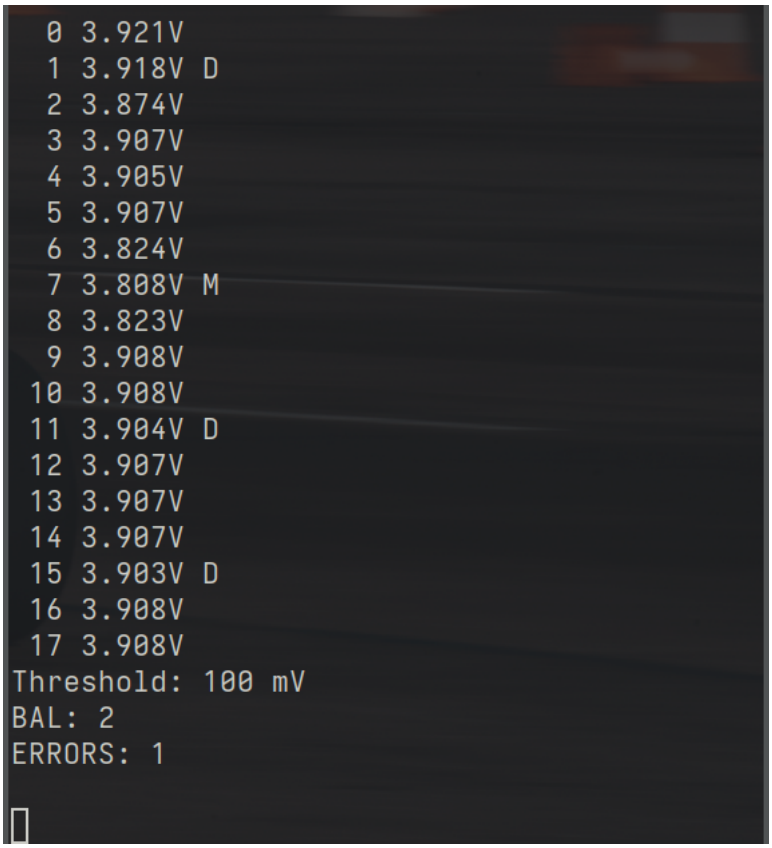

Figure B.2: Serial console during balancing. `M` indicates the minimum voltage, cells marked with `D` are being discharged

Appendix C Error management test

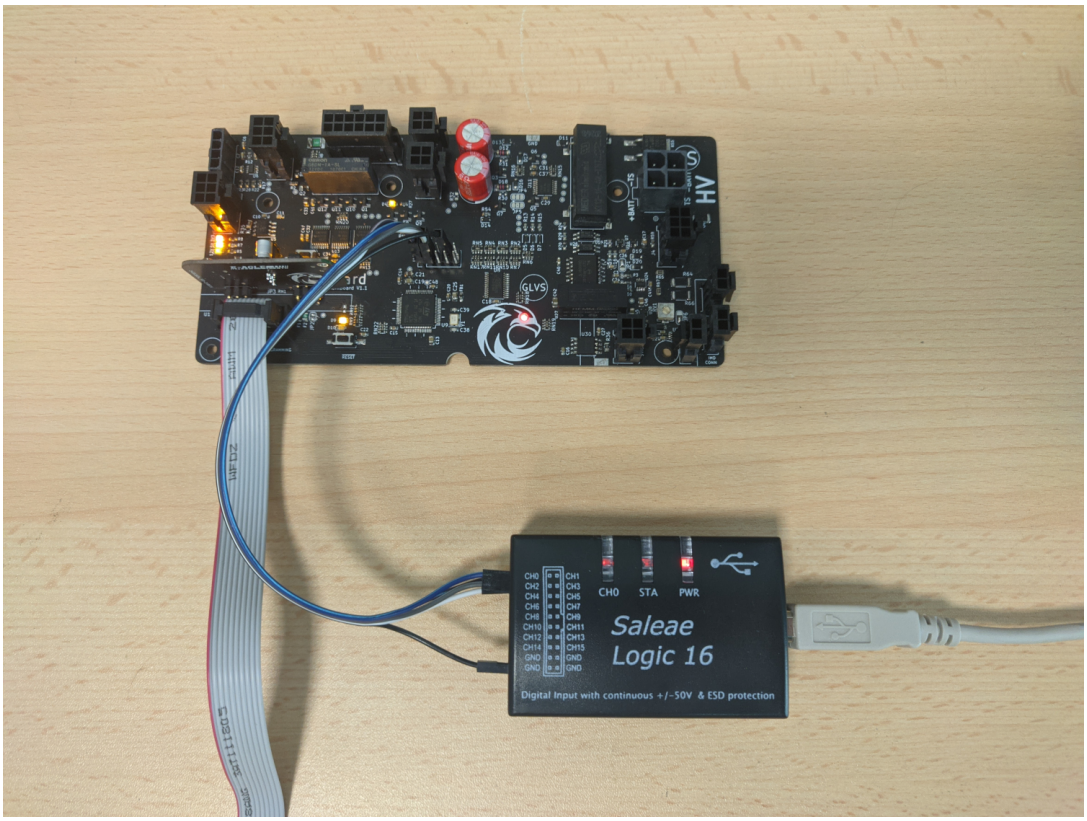

Figure C.1: Error test setup with the mainboard and a logic analyzer.

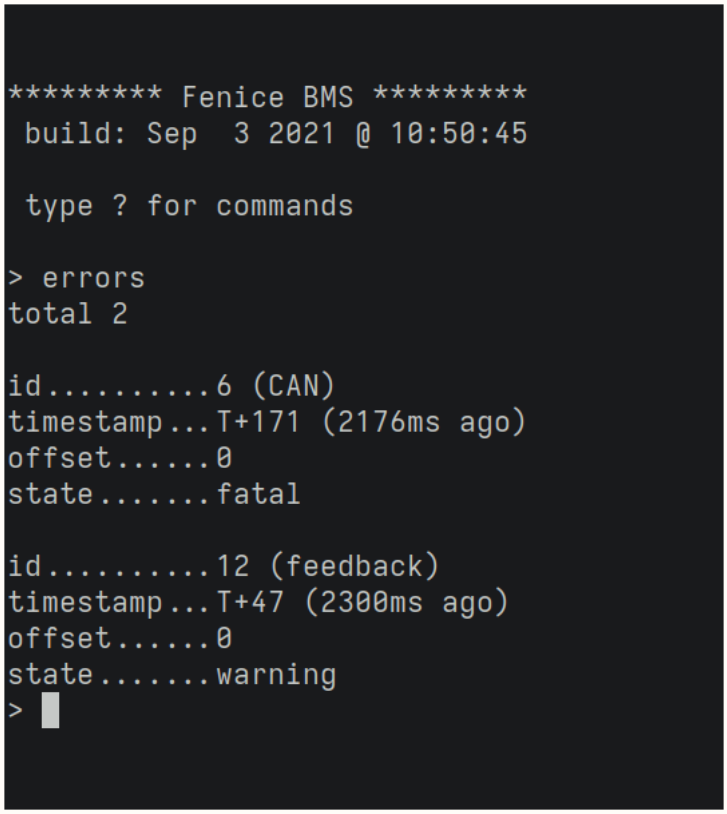

Figure C.2: Serial console reporting active errors.