

POLITECNICO DI TORINO

Collegio di Ingegneria Informatica, del Cinema e Meccatronica



Master's Degree Thesis
Mechatronic Engineering

Optimization and Integration of Model-Based Algorithms for the Evaluation and Management of Dynamic Parameters of Lithium-Ion Batteries, Deployed on a Custom Automotive BMS

Supervisors

Prof. Massimo VIOLANTE

Eng. Simone PENNAVARIA

Candidate

Salma Yasser Mohamed

Abdelwahhab FARAG

July 2025

Summary

The efficient management of lithium-ion battery packs (Li-ion) is essential to ensure the correct performance, longevity, and safety of modern electric vehicles (EVs) and other high-power applications. This thesis focuses on the development, integration, and optimization of model-based algorithms within an automotive Battery Management System (BMS) to accurately estimate battery parameters and manage power limits. The research emphasizes the role of software algorithms, particularly the Extended Kalman Filter (EKF) for state-of-charge (SOC) estimation and the PI controller-based Power Limits algorithms for power management.

A model-based approach was employed to evaluate the dynamic parameters of Li-ion batteries, leveraging MATLAB Simulink for modeling and simulation, as well as the software tool Embedded Coder for code generation, optimization and deployment on a custom automotive BMS platform. Model-Based Design (MBD) is a widely adopted methodology for developing embedded software, especially in the automotive industry. Embedded Coder is a key tool that facilitates the transition from system models to production-ready C and C++ code. MBD focuses on using models throughout the development process, from requirements to testing, enabling simulation and analysis of system behavior without needing expensive hardware.

A rigorous testing and validation process was conducted to ensure the accuracy, reliability, and performance of the algorithms. This included Software-in-the-Loop (SIL) and Processor-in-the-Loop (PIL) simulations to verify the functional equivalence between the auto-generated code and the original Simulink models. The tests also provided valuable insights into execution time, memory usage, and the impact of model and code-level optimizations. Post-integration testing was performed to confirm the correct integration of the algorithms within the BMS's software architecture. This phase involved code development, integration of the CAN bus system, as well as performance analysis via the software tool CANalyzer.

In the final development phase, the firmware was deployed to the target and tested on a real battery pack under controlled laboratory conditions at BeonD S.r.l.'s

laboratory facility, enabling validation of the system's behavior in scenarios closely resembling real-world operation. The results demonstrate that automated code generation and optimization of model-based algorithms using MATLAB Simulink not only enhances algorithm performance but also simplifies integration into embedded systems. Optimization techniques were applied to improve the execution efficiency of the algorithms, leading to a significant reduction in the computation time.

Overall, this study highlights the critical role of combining advanced computational techniques with embedded systems to enable efficient and reliable battery operation. As the adoption of electric vehicles accelerates, continued research and innovation in intelligent battery management systems will be essential to improve performance, extend battery life, and support the long-term sustainability of electric mobility.

Table of Contents

List of Tables	VII
List of Figures	VIII
Acronyms	XI
1 Introduction	1
2 State of the Art	5
2.1 Role of the Battery-Management-System	5
2.1.1 Types of faults in the Li-ion battery system	6
2.2 Offline state estimation methods	7
2.2.1 The battery cell	7
2.2.2 The OCV test	8
2.2.3 Coulomb Counting	10
2.3 Online SOC estimation	10
2.3.1 Model-based state estimation	12
2.3.2 Equivalent-circuit models	12
2.3.3 The BCM	15
2.3.4 Extended Kalman Filter for state estimation	16
2.4 Power Limits and the State-of-Health	19
2.4.1 Power Limits	19
2.4.2 State-of-Health	24
3 Code Generation and Optimization	26
3.1 Code generation: Embedded coder	26
3.1.1 Configuration parameters for code generation	27
3.1.2 Code optimization objectives	28
3.2 Implementation of the code optimizations	32
3.2.1 Data management and version control	38

4	System Architecture	41
4.1	Laboratory test bench	41
4.1.1	The battery pack	41
4.1.2	Slave board	43
4.1.3	Shunt sensor	44
4.1.4	Master and slave controller	45
4.1.5	Contactors and safety net	46
5	Integration Phase	48
5.1	Module Structure	48
5.2	The code-base (BMS firmware)	49
5.3	Integration within BeonD's BMS	50
5.3.1	The Wrapper files	50
5.4	CAN bus	53
5.4.1	The development process of CAN	53
5.4.2	CANalyzer	56
5.5	Execution time measurement methods	56
6	Testing and Validation	60
6.1	Testing of algorithms	60
6.1.1	EKF vs. Coulomb Counter	61
6.1.2	SIL/PIL testing	63
6.1.3	Completion time results	66
6.1.4	Testing and validation of Power Limits	68
6.2	Testing and validation post-integration	72
6.2.1	Power limits execution times	72
6.2.2	Testing and validation using CANalyzer	73
7	Conclusions	81
	Bibliography	83

List of Tables

1.1	SOC estimation methods pros and cons	3
2.1	MOLICEL P45B cell characteristics	8
3.1	Embedded Coder's Build Directory	27
6.1	Average execution times of EKF functions obtained through PIL test	65
6.2	Completion times of various algorithms post-integration measured using the oscilloscope	67
6.3	Effect of optimization on completion times of EKF post-integration measured using the oscilloscope	68
6.4	PL10s execution time optimization results post-integration measured using RunTime Counter	73

List of Figures

1.1	SOC estimation methods	3
1.2	The phases of design, integration and testing	4
2.1	The algorithms implemented in the BMS	6
2.2	Look-up SOC-OCV table used in EKF used to estimate voltage V_0	9
2.3	Coulomb counter simulated on Simulink	11
2.4	General diagram of the model-based estimation approach coupled with a feedback mechanism	13
2.5	Third order resistor-capacitor ECM diagram	14
2.6	Look-up table for estimation of the ECM parameter R_1	15
2.7	Look-up table for estimation of the ECM parameter C_1	16
2.8	The BCM model implemented in Simulink	17
2.9	The EKF general overview	18
2.10	The EKF algorithm modelled on MATLAB Simulink	19
2.11	The Power Limits algorithm modeled in Simulink	21
2.12	The discrete PI controller implemented in Simulink	23
2.13	SOH algorithm modeled in Simulink	25
3.1	Code optimization objectives	28
3.2	Code generation advisor	30
3.3	Static Code Metrics Report	31
3.4	Code Interface Report	31
3.5	Removing root-level I/O initialization	34
3.6	Removing internal data zero initialization	34
3.7	Configuration parameters section showing the data type supports selected	35
3.8	Enabling and reusing local block outputs in PL10s source file	37
3.9	Simulink referenced configuration activated	39
3.10	Git extensions branches	40
4.1	BMS test benches in BALF	42

4.2	BALF's test bench lithium-ion battery packs connected to the slave boards	43
4.3	Slave board connected to the battery pack's cells	44
4.4	Isabellenhuetten shunt sensor	45
4.5	BEOND's PCB incorporating the master and slave controllers . . .	46
4.6	Pre-charge circuit	47
5.1	Module Structure	49
5.2	CAN interface hardware	57
5.3	Vector CANdb++ editor	57
5.4	Runtime Counter function used to measure execution time of EKF .	59
6.1	EKF vs. CC SOC estimation in Simulink, DCHG at 10A	61
6.2	EKF vs. CC SOC estimation in Simulink, CHG at 10A	62
6.3	EKF vs. CC SOC estimation in Simulink, CHG and DCHG random current profile	63
6.4	Quantitative data comparison between the EKF simulation and SIL, 50A DCHG current	64
6.5	Graphical data comparison between the EKF simulation and SIL, 50A DCHG current	65
6.6	Oscilloscope displays the pulse width indicating the algorithm's execution time	67
6.7	Simulink's Data inspector for Power Limits under a random current profile	69
6.8	Simulink's Data inspector for Power Limits under a 90 A constant current profile	70
6.9	Low integral gain	71
6.10	High Integral Gain	71
6.11	SIL simulation test of PL10s under a DCHG current	72
6.12	First trial tuning PID for power limits 1s and 10s under a DCHG random current	75
6.13	First trial tuning PID for power limits 1s and 10s under a CHG random current	76
6.14	CANalyzer results, $K_p=0.01$, $K_i=0.001$, 70 A DCHG current . . .	78
6.15	CANalyzer results, $K_p=0.01$, $K_i=0.0001$, 70 A DCHG current . . .	79
6.16	CANalyzer results, $K_p=0.01$, $K_i=0.001$, 70 A DCHG current . . .	80

Acronyms

Li-ion

Lithium-ion

ECM

Equivalent Circuit Model

OCV

Open Circuit Voltage

EKF

Extended Kalman Filter

CC

Coulomb Counter

Q

Total Capacity

SOC

State of Charge

SOH

State-of-Health

BMS

Battery Management System

PCB

Printed Circuit Board

NaN

Not-a-Number

ERT

Embedded Real-Time Target

Req

Equivalent Resistance

PL1s

Power Limits 1s Algorithm

PL10s

Power Limits 10s Algorithm

PL30s

Power Limits 30s Algorithm

CAN

Controller Area Network

V_t

Terminal Voltage

HEV

Hybrid-Electric Vehicles

PHEVs

Plug-in Hybrid-Electric Vehicles

EVs

Electric Vehicles

DCHG

Discharge

CHG

Charge

K_i

Integral Gain

K_p

Proportional Gain

ADC

Analog-to-Digital Converter

PWM

Pulse-Width Modulation

SPI

Serial Peripheral Interface

UART

Universal Asynchronous Receiver-Transmitter

GPIO

General-Purpose Input/Output pin

UDS

Unified Diagnostic Service

API

Application Programming Interface

Chapter 1

Introduction

This thesis explores the effective management of lithium-ion (Li-ion) battery packs, by implementing an automotive battery management system (BMS), which requires the integration of both software and hardware components. The hardware components consist of electronic circuits designed to ensure the safety of both the battery pack and its user, while also facilitating measurements such as battery cell voltages, electrical current, and temperature. The software components are responsible for estimating the battery state, managing power limits, and monitoring and coordinating the battery pack's operations. The primary focus of this thesis is on the software methods and algorithms involved in this management process.

To manage the operation of the battery pack, certain parameters must be known. This includes voltage, current, temperature, state-of-charge (SOC), state-of-health (SOH) and power limits. The SOC of the battery is an essential parameter for battery management; however, there is no direct way of measuring the SOC in real-time; therefore, complex algorithms such as the Extended Kalman Filter (EKF) are implemented to accurately estimate the SOC. Advanced battery management methods come with added costs, so not all applications use them. For inexpensive devices such as TV remotes, the cost of battery failure is low, making advanced management unnecessary. However, for mission-critical or large battery systems, where premature failure can be costly and unsafe, advanced management is justified. This thesis focuses on methods relevant to high-stakes applications where the cost of failure outweighs the additional expense of better management.

Electric vehicles can be categorized into three main types [1]:

1. Hybrid-electric vehicles (HEVs) use a combination of a gasoline engine and an electric motor, with a small battery pack to assist in acceleration and energy recovery during deceleration. These vehicles do not have an all-electric range and rely on the gasoline engine for recharging.

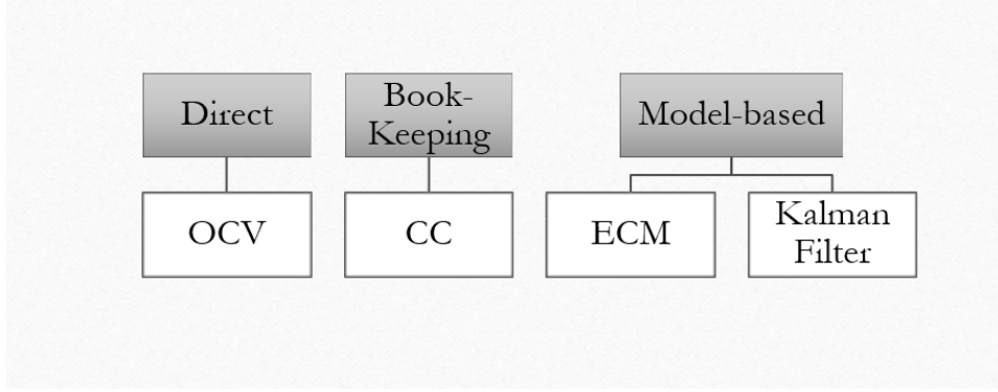
2. Plug-in hybrid-electric vehicles (PHEVs) have larger battery packs, allowing limited all-electric driving (15-30 km). They can be plugged into the grid for recharging and switched to hybrid mode once the battery is depleted.
3. Electric vehicles (EVs) or battery-electric vehicles (BEVs) rely solely on a battery-powered motor, without a gasoline engine. Their range is determined by battery capacity, typically ranging from under 200 to over 500 km.

Each vehicle operates using a different type of battery pack and requires different desired performance. The BMS is required to adapt to the different performance requirements of the vehicle and correctly manage the assigned battery pack.

The focus of this thesis is on rechargeable lithium-ion (Li-ion) batteries. The Li-ion battery offers several advantages over other types of batteries, including higher voltage (3.7V vs. 1.2V for NiMH or NiCd), higher energy density, fewer cells needed for certain applications, and lower self-discharge rates. However, they are more expensive and sensitive to overcharging, requiring additional protective circuitry. Although lithium ion cells are more expensive and more complex to manufacture, their price is expected to decrease as production scales. Li-ion batteries are becoming more and more popular because of their long lifespan, high energy and power density, in addition to the positive impact they have on the environment. However, failure in management of such batteries leads to a reduction in their lifespan, poor performance and potentially dangerous situations due to overheating or overcharging. Therefore, the management of Li-ion batteries is critical in order to use the battery to its full potential while ensuring safe operating conditions.

There are different methods that can be used for the SOC estimation, each with their own advantages and disadvantages. Figure 1.1 illustrates the different methods used for SOC estimation. Starting from the direct method and the simplest that is the OCV (open-circuit voltage) test, moving on to the Book-keeping method known as the Coulomb Counting (CC) and finally the online method implemented in the BMS that is the Model-based method derived from the Equivalent circuit model (ECM), complemented by the sophisticated adaptive filter known as the Extended Kalman Filter (EKF). This paper employs a third-order equivalent electrical circuit model to characterize the behavior of a lithium-ion battery. Each method will be discussed in detail in this paper. Table 1.1 outlines the benefits and drawbacks of such methods [2].

A BMS is an advanced solution designed to monitor and control battery packs and provide fault diagnosis. The STM32 microcontroller-based PCB, designed for automotive applications, serves as a central component in the BMS, integrating both master and slave boards while communicating over the CAN bus. The microcontroller is programmed by uploading BMS-specific firmware that governs battery management functions, enabling precise control and monitoring of the battery.

**Figure 1.1:** SOC estimation methods

SOC estimation methods	Pros	Cons
OCV	Simple, Accurate	Offline calculation only
CC	Simple	Cannot initialize SOC, Inaccurate
ECM	Online	Inaccurate
EKF	Online, Accurate	Complex

Table 1.1: SOC estimation methods pros and cons

Additionally, advanced algorithms are implemented within the BMS firmware to manage critical tasks, such as ensuring safety, protecting battery cells from damage during faults or abuse, extending battery life, as well as, maintaining optimal battery performance and ensuring the pack delivers or receives power within design specifications.

The process of designing, integrating, and testing a model involves several critical phases to ensure its robustness and effectiveness. It begins with the definition of mathematical models, in which the underlying principles and equations governing the system are formulated to accurately represent real-world behavior. Next, model-based design using MATLAB/Simulink enables the creation and simulation of the system's dynamics within a visual framework, allowing for iterative refinement and validation. Once the design is finalized, code generation and optimization using MATLAB/Embedded Coder converts the model into efficient, target-specific C code, ensuring compatibility and performance. The generated algorithm is then integrated into the Battery Management System (BMS) using development tools such as STM32CubeIDE, enabling seamless deployment on embedded hardware. Finally, the system undergoes rigorous testing and validation, utilizing tools such as CANalyzer for network-level analysis and the laboratory test bench for comprehensive hardware-in-the-loop testing, ensuring the model meets all functional and performance criteria. Figure 1.2 summarizes the various phases of design,

integration, and testing.

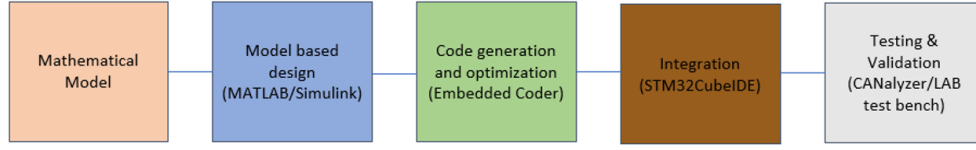


Figure 1.2: The phases of design, integration and testing

After the model-based design phase is over, we transition to the code generation and optimization phase in which the model is converted into code written in C language and then the process of optimization begins. Optimization is particularly useful in embedded systems where memory and processing resources are limited. Optimizing the code, reducing the startup time and increasing the execution speed significantly impacts system performance. Minimizing the execution time of algorithms in a BMS is critical for maintaining real-time performance, safety, power efficiency, and system reliability. It helps the system make quick decisions, preserve battery life, ensure safe operation, and reduce the cost and complexity of the hardware, all of which are essential in deploying an effective battery management system. Reduction in initialization time is especially important in real-time systems such as the BMS, where data need to be updated at a high rate and consequently enable the BMS to make decisions and take faster action resulting in improved performance, safety and prolonging of the battery's life.

Overall, the goal of the project is to perform the Code generation and optimization of advanced algorithms using the MATLAB® Simulink environment. This is achieved while defining a procedure that ensures smooth and quick integration of algorithms into the master BMS PCB designed by BeonD s.r.l, the company in which the thesis was carried out. The project is finalized by testing and validating the obtained results. A portion of the testing and validation was done using desktop tools. The remainder of the tests were done in BeonD's laboratory BALF (BeonD advanced laboratory facilities), a laboratory which houses several equipment for testing cells, modules, and battery packs and is home to custom battery management system projects.

Chapter 2

State of the Art

2.1 Role of the Battery-Management-System

The methods and algorithms discussed in this paper are implemented in a Battery Management System (BMS). The BMS is an embedded system combining specialized electronics, including hardware and software components used to serve specific functions. The focus will be mainly on the four core algorithms illustrated in Figure 2.1. The algorithms include the BCM (battery cell model) algorithm for modelling battery cells, the EKF based SOC algorithm for estimation of the battery's state of charge, the SOH algorithm for estimation of the battery's health and finally, the Power Limits algorithm for managing the power limits of the battery pack.

The primary objectives of a BMS are:

1. Ensuring operator safety by detecting unsafe conditions such as over-heating or over-charging and responding, by reducing or cutting off the current.
2. Protecting the battery cells from damage during failure or abuse, using either software or hardware to isolate faulty components.
3. Extending battery life by coordinating with the load controller by managing the battery's temperature and preventing over-charging or over-discharging based on the estimated SOC.
4. Maintaining the battery's ability to meet its functional design requirements, ensuring it can deliver or receive power within its rated capacity while maintaining the voltage and temperature within their specified limits and balancing the cells.

5. Measuring voltage, current and temperature as well as controlling contactors and the thermal management system.
6. Communicating with the user and other electronic components within the vehicle, reporting on SOC and power, and signaling to the user the end-of-life of the battery, based on the estimated SOH.



Figure 2.1: The algorithms implemented in the BMS

2.1.1 Types of faults in the Li-ion battery system

Lithium-ion batteries face various internal and external faults that can lead to performance degradation and serious consequences like thermal run-away, fires, or explosions. Key internal faults include overcharge, undercharge, internal short circuits, overheating, and accelerated degradation, while external faults involve issues such as sensor failures, cooling system malfunctions, and cell connection faults.

Overcharge can cause thermal runaway, gas buildup, and cathode damage due

to incorrect measurements or faulty charging systems, while undercharge leads to capacity loss, electrode corrosion, and potential short circuits. Internal and external short circuits trigger excessive heat and electrolyte decomposition, resulting in thermal runaway. Overheating accelerates degradation, causing swelling and potential explosions, while accelerated degradation shortens the battery's lifespan through material disintegration and lithium loss. Among these, thermal runaway is the most severe, caused by cumulative heat and pressure increases [3].

The BMS receives information about each cell within the battery pack, and ensures that voltage and temperature limits are respected, while also ensuring that the cells neither over-charge nor under-charge. For instance, if a cell reaches its maximum SOC the BMS commands the charging contactors to be opened to prevent the cell from over-charging. Effective fault detection and management strategies are vital for minimizing risks, ensuring safety, and maintaining long-term battery performance.

2.2 Offline state estimation methods

The SOC of a cell Z_k is defined as the ratio of its residual capacity to its total capacity as shown in Equation (2.1). The average lithium concentration stoichiometry is defined as shown in Equation (2.2) at time index k . However, there is no direct method for the measurement of lithium concentration within a battery cell. Therefore, the SOC must be somehow estimated based on the available measurable cell parameters [1].

$$\theta_k = \frac{C_{s,avg,k}}{C_{s,max}} \quad (2.1)$$

$$Z_k = \frac{\theta_k - \theta_{0\%}}{\theta_{100\%} - \theta_{0\%}} \quad (2.2)$$

This section presents two offline state estimation methods namely, the OCV test and coulomb counting. The advantages and the limitations of these methods are also discussed.

2.2.1 The battery cell

A battery cell is the smallest individual electrochemical unit, providing a voltage that depends on the specific chemicals and compounds used in its construction. In contrast, a battery pack is a collection of cells connected electrically.

The voltage of a cell depends on various factors, with the nominal voltage specified

by the manufacturer serving as a convenient reference value for its voltage class. The actual operating voltage can fluctuate above or below this nominal value, with most lithium-based cells having nominal voltages above 3V.

The nominal charge capacity of a cell indicates the amount of charge it can hold, measured in ampere-hours (A h) or milliampere-hours (mA h). Closely related to this, the C-rate represents the relative current measure of the cell, defined as the constant-current charge or discharge rate that the cell can sustain for 1 hour. This is calculated by multiplying the cell's nominal ampere-hour rating by 1 h^{-1} .

The cell used in this project is the Molicel P45B cell. Table 2.1 summarizes the cell characteristics such as its capacity, voltage, current and temperature. The high discharge current of value 45A is noteworthy as it showcases how powerful the cell is. In addition, the cell has an impressive life cycle therefore, making the Molicel P45B suitable for creating a long-lasting, high performing battery pack [4].

Cell characteristics	Value
Typical Capacity	4500 mAh
Charge voltage	4.2 V
Discharge voltage	2.5 V
Standard charge current	4.5 A
Standard discharge current	45 A
Charge temperature	0°C to 60°C
Discharge temperature	-40°C to 60°C

Table 2.1: MOLICEL P45B cell characteristics

2.2.2 The OCV test

The OCV test is a simple and accurate method to measure the SOC of the battery at a resting state. In a resting state, variables like temperature and hysteresis are eliminated and the terminal voltage indicates the battery's energy content; therefore, simplifying the SOC-OCV relationship. This method requires calibration through comprehensive characterization of SOC-OCV curves across different temperatures and aging conditions to account for their effects on voltage response. Despite its simplicity, the OCV test cannot be carried out in online applications as it requires the battery to arrive at a resting state to perform the SOC estimation. In addition, these methods can lead to significant estimation errors, especially in batteries with non-linear relationships between the SOC and OCV. However, data obtained from the OCV test can be used to create lookup tables where the SOC is a function of voltage and temperature. This is useful in estimating the parameters of the equivalent circuit model (ECM) and initializing the SOC which will be discussed

later in this paper. Figure 2.2 below shows one of the look-up tables implemented in model-based representation of EKF in MATLAB, used to estimate the voltage across one of the ECM's resistors based on the measured temperature and estimated SOC. Multiple SOC-OCV curves were created and the values are based on tests carried out in BALF at different temperatures ranging between 293-344 Kelvin (21-71 degree Celsius). Look-up tables will be discussed more in details later in this chapter, highlighting their important role in estimating cell parameters.

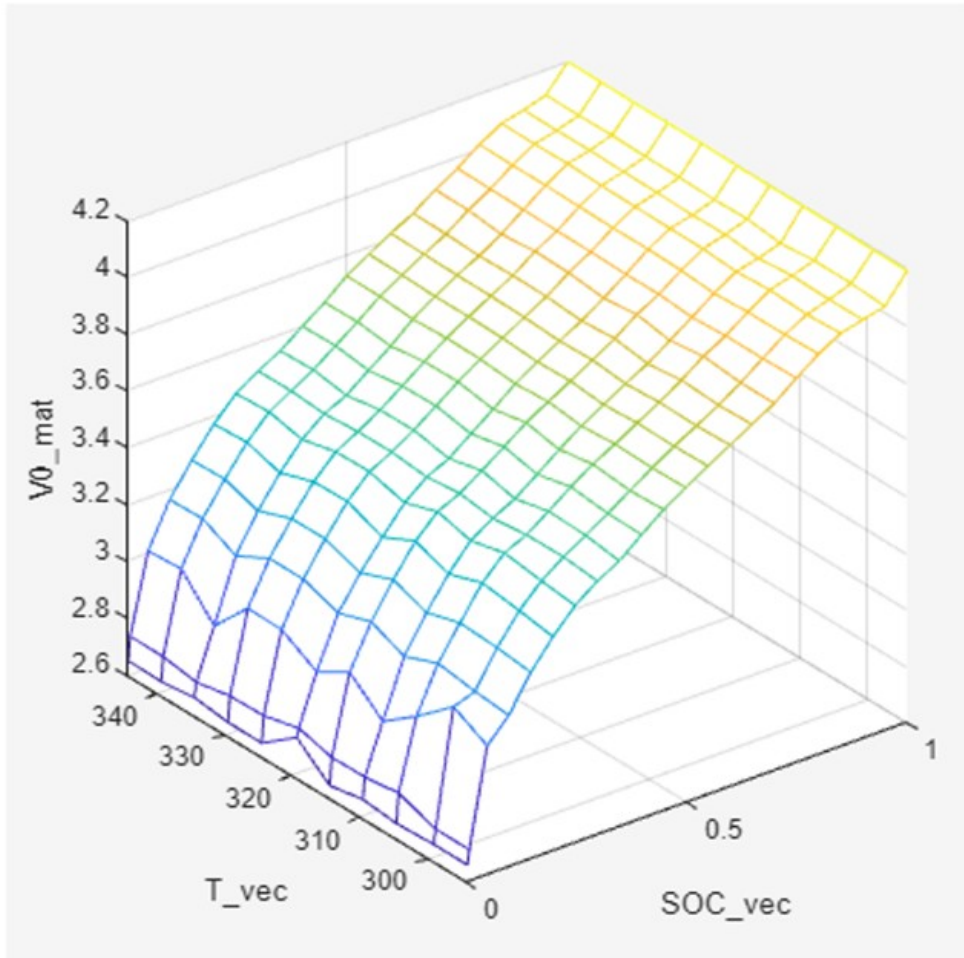


Figure 2.2: Look-up SOC-OCV table used in EKF used to estimate voltage V_0

2.2.3 Coulomb Counting

While it might seem logical to obtain the state of charge (SOC) from cell voltage, this is problematic due to several factors. Cell voltage is influenced by temperature, surface concentrations, and other factors, but SOC depends on average concentrations within the cell. Changes in voltage do not always correspond to changes in SOC. Resting a cell, temperature fluctuations, and hysteresis can alter voltage without affecting SOC. Although voltage can serve as an indirect indicator of SOC, it's not a reliable direct measurement. Instead, SOC is more directly tied to current flow, as it changes when current passes through the cell during charging or discharging. The relationship between SOC for cell i at time index k and the cell current i_k can be expressed via Equation (2.3). This method is known as coulomb counting. Where cell current is positive on discharge and negative on charge, $\eta_k^{(j)}$ is cell coulombic efficiency, $z_0^{(i)}$ is the initial SOC of cell i and $Q_k^{(j)}$ is the cell total capacity in ampere-seconds. The total capacity Q is determined via the total net ampere-hours discharged.

$$z_k^{(i)} = z_0^{(i)} - \sum_{j=0}^{i-1} \frac{1}{Q_k^{(j)}} \eta_k^{(j)} i_k \quad (2.3)$$

However, this method for estimating SOC has certain weaknesses. For example, inaccuracies in measuring the current can limit the accuracy at which the SOC is estimated. In addition, the value of Q is affected by the temperature as well as aging. As the cell ages the value of Q decreases and therefore it needs to be monitored and corrected over time. On the other hand, if the SOC is initialized incorrectly, the system will base its estimates on a flawed starting point, leading to inaccuracies throughout the estimation process. As the battery goes through more charge and discharge cycles, errors in SOC estimation tend to accumulate, further decreasing the accuracy of the prediction over time. Figure 2.3 shows how the coulomb counter was simulated in Simulink environment.

2.3 Online SOC estimation

Accurate battery modelling is a primary requirement of online SOC estimation for the simulation of battery dynamics. Model-based state estimation relies on mathematical techniques to predict battery cell behavior and system responses. It uses measurements of current, voltage, and temperature based on a battery model represented by an ECM to obtain the parameters necessary for the estimation of the state of charge (SOC). Since the ECM cannot simulate all the electrochemical processes in the battery needed for the estimation of the SOC, the ECM is coupled with the Extended Kalman Filter (EKF) to accurately predict and correct the

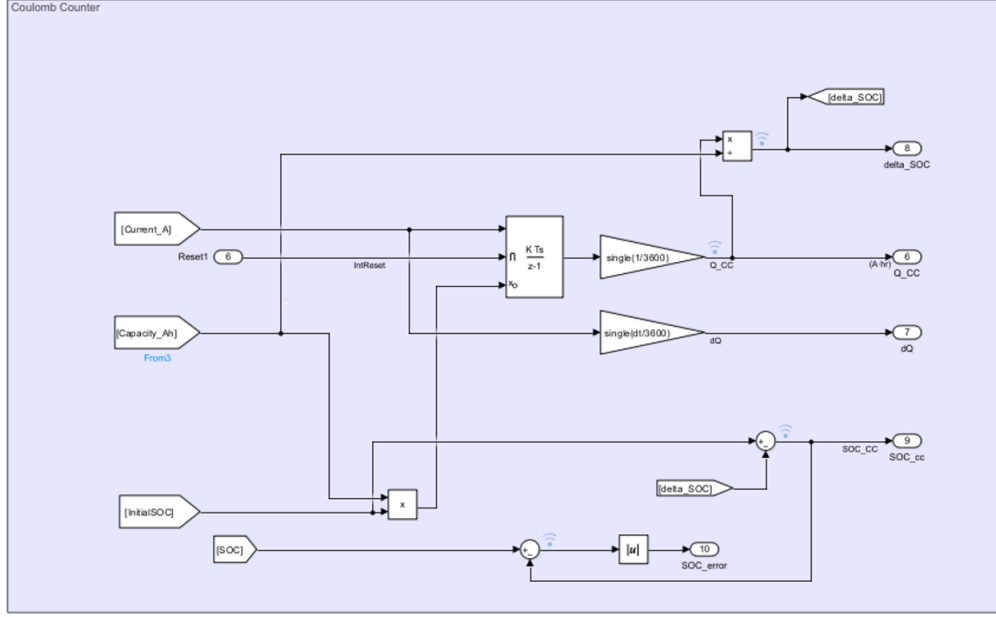


Figure 2.3: Coulomb counter simulated on Simulink

SOC value. The EKF integrates voltage and current data, accounting for noise and refining SOC estimates through feedback that compares predicted and actual voltages. The EKF is a sophisticated algorithm used for battery state estimation, designed to manage errors stemming from measurement noise, state-estimation inaccuracies, and modeling discrepancies. It adapts the classic Kalman filter to nonlinear systems by linearizing the model at each time step. Though EKF introduces approximations and has limitations, it remains effective for moderate non-linearities in practical applications.

Unlike the coulomb counting method, the model-based state estimation approach relies on mathematical techniques to accurately represent the battery cell's behavior and predict system responses. The EKF, while capable of handling non-linear systems, may encounter difficulties when applied to highly non-linear systems such as lithium-ion batteries. Particularly at low states of charge, the battery's behavior becomes increasingly non-linear, causing the EKF to produce less accurate state-of-charge estimations.

Despite these limitations, the EKF is considered a leading method for SOC estimation across various conditions. It operates as an optimal autoregressive data processing algorithm, providing minimum variance estimation through a recursive approach. Additionally, it can quantify the errors in its estimates [5].

2.3.1 Model-based state estimation

SOC estimation is the task of battery state estimation using battery parameters obtained from a real battery pack or an equivalent-circuit model of the battery pack cells. A combination of the measured current, voltage, temperature, and knowledge from a cell model is used to calculate estimates of SOC.

There are three approaches to estimate the SOC. The first approach is a voltage-based method that estimates using plots of SOC vs. OCV and assuming cell's terminal voltage to be approximately equal to the open circuit voltage. However, the results are too noisy. The second approach is a current-based method to estimate using the state of charge equation. If estimates are incorrect, there is no feedback mechanism to correct this error. Additionally, it is prone to bias, self-discharge, and leakage errors. Finally, the third and most accurate approach is a model-based estimation approach combining voltage-based and current-based methods to estimate the state of charge (SOC) and other internal states of a battery.

The model-based estimation approach includes both the true system that is the actual cell as well as the system model coupled with a feedback mechanism Figure 2.4. The input to the cell is the current while the output is the terminal voltage. The SOC, diffusion current and hysteresis voltages within the cell cannot be measured and therefore, they must be estimated. In addition, the current and voltage sensors experience noise causing inaccuracies in the measurement of the input and output. In the model-based estimation, the same input of the true system is used as in input for the model, the state is estimated and then the output of the system is predicted. The predicted output is compared to the true system's output measured by the voltage sensor and the difference is calculated. This difference is used in a feedback mechanism to update the estimated state of the model. The state estimate must be updated carefully accounting for the errors due to the sensor noise, the state-estimation errors and the modeling errors. The process of refining the estimate through feedback, makes the model estimation method more accurate than the simpler coulomb counting [1].

2.3.2 Equivalent-circuit models

A simple way to model a cell's operation is using electrical-circuit analogs to define a behavioral or phenomenological approximation to how a cell's voltage responds to different input-current stimuli. The input/output (current/voltage) behaviors of a lithium-ion cell are often well approximated by an equivalent circuit. When the model is being created, values of the resistance and capacitance (R_0 , C_1 , and R_1) are adjusted using an optimization procedure to make model predictions agree with the measured cell-test data. This process is known as system identification. The optimized parameter values are typically a function of state of charge and

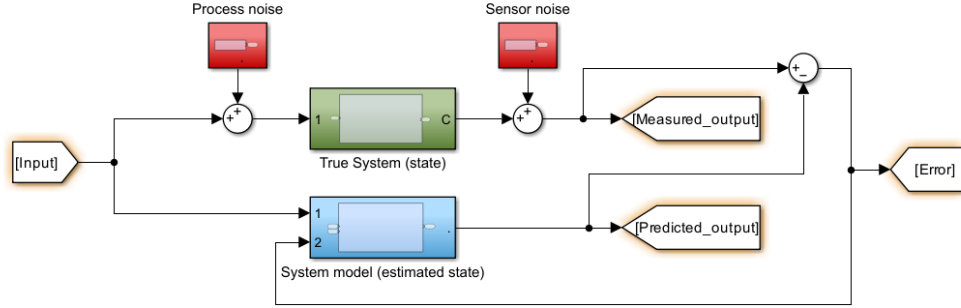


Figure 2.4: General diagram of the model-based estimation approach coupled with a feedback mechanism

temperature. Once the parameters are optimized, the discrete-time state space form of the battery model can be utilized in the EKF.

The approach begins by constructing a circuit model for a battery cell step by step, starting with its basic observed behaviour and refining it to minimize modelling errors. The process starts with the simplest model, representing the cell as an ideal voltage source, where the terminal voltage is constant and independent of load current or past usage. Although this initial model is unrealistic, it serves as a foundation, acknowledging that cells supply a predictable voltage known as open-circuit voltage (OCV) in an open-circuit condition. The ideal voltage source remains a key component in the final equivalent-circuit model, even as refinements are made to account for more complex behaviour. We proceed by implementing improvements to the simple cell model by considering the SOC dependence, equivalent series resistance and diffusion voltages.

One of the most common equivalent circuit models used is the 3RC ECM. It is made up of the battery represented by an OCV, in addition to an internal resistance R_0 and three parallel RC pairs as shown in Figure 2.5. The resistance R_0 models the instant battery response, while the RC networks models different battery dynamics. The OCV is a function of both the SOC and the temperature.

A precise lithium-ion battery model is essential for evaluating the suitability of the cells across various applications and analyzing their dynamic behavior. A detailed testing procedure was implemented to parameterize the model, involving extensive characterization experiments conducted under a wide range of operating conditions. The results were utilized to parameterize the proposed dynamic model of the Li-ion battery cell. Two of the already existing look-up tables obtained from thus tests are depicted in Figure 2.6 and Figure 2.7. The look-up tables mentioned were used to estimate the R_1 and C_1 parameters of the ECM respectively.

The equations forming the equivalent circuit model comprise of the SOC and the

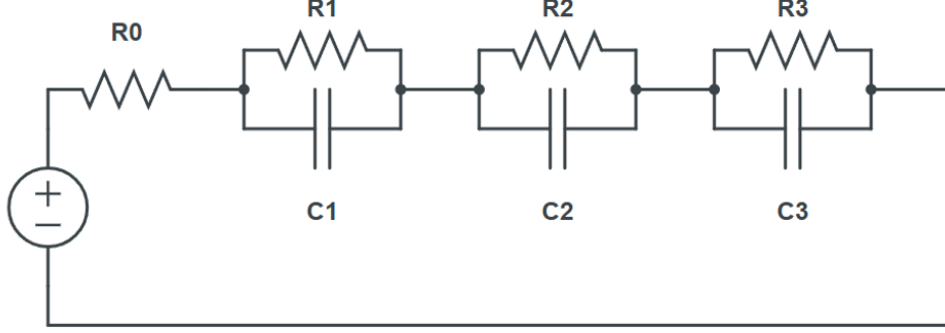


Figure 2.5: Third order resistor-capacitor ECM diagram

diffusion-resistor current. Two equations are used to describe the system, the state equation and the output equation. The state equation describes all the dynamic effects, while the output equation computes the voltage at discrete-time index k . To simulate battery-cell behaviour, the ECM voltage equation is evaluated, and the model state equation is updated once per sample interval. We conclude that the ECM looks similar to-but not identical to - a linear state-space system. Equation (2.4) below indicates the form of the linear state-space system. Where χ_{k+1} is the state vector and y_k is the system output representing the terminal voltage V_t [6].

$$\begin{aligned}\chi_{k+1} &= A_k \chi_k + B_k i_k \\ y_k &= C_k \chi_k + D_k i_k\end{aligned}\tag{2.4}$$

The non-linear nature of the SOC-OCV relationship represented by the matrix C , makes it difficult to obtain accurate estimation of the states by applying simple methods such as the classic Kalman filter. Additionally, despite the simplicity and fast computation of the offline algorithms, they do not take into account the aging of the battery cell, nor do they simulate other electrochemical processes in the battery that affect the SOC estimation. Therefore, the linear state-space system based on the ECM is coupled with an adaptive filter known as the Extended Kalman Filter (EKF) for accurate prediction and correction of the SOC value.

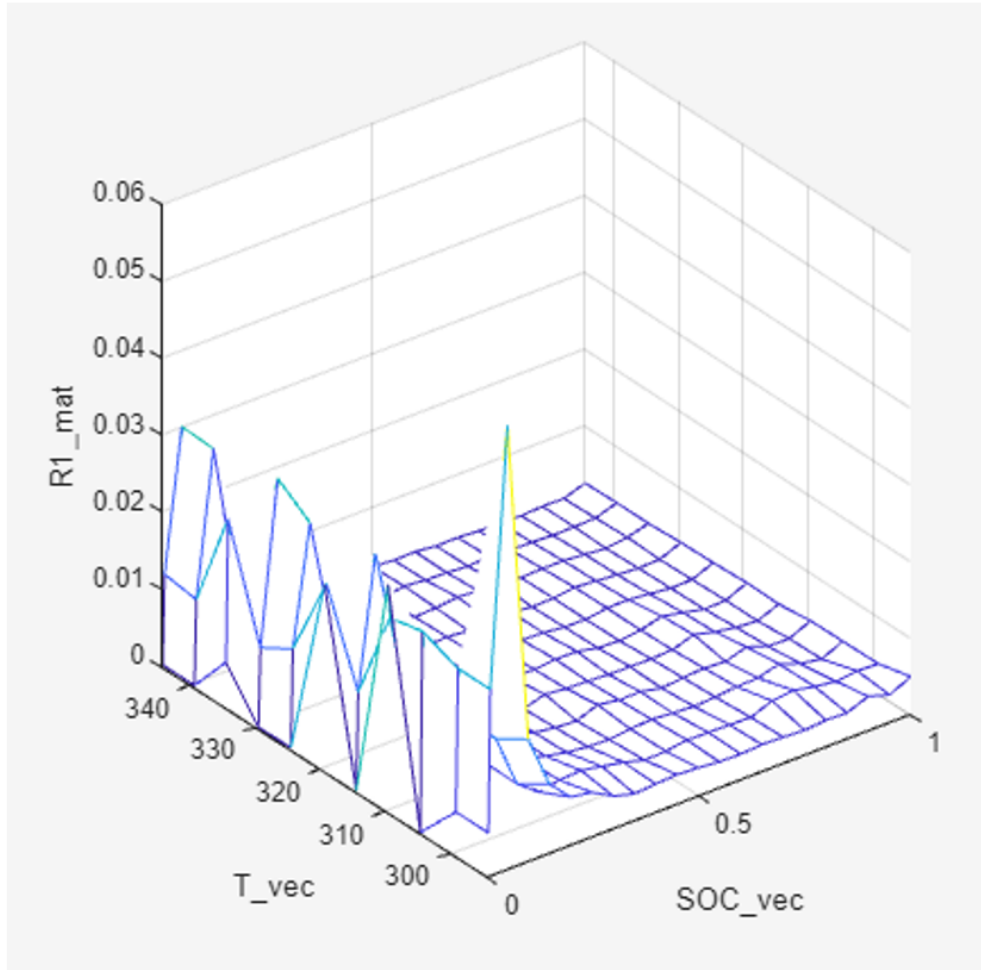


Figure 2.6: Look-up table for estimation of the ECM parameter $R1$

2.3.3 The BCM

The Battery Cell Model (BCM) modelled on MATLAB is founded upon the 3RC Equivalent Circuit Model (ECM). This ECM approach utilizes electrical circuit analogs and associated look-up tables to effectively represent the behavior of a single battery cell. By incorporating inputs of State of Charge (SOC), current, and temperature, the model dynamically estimates key cell parameters. Subsequently, the BCM calculates the expected cell voltage during both charging (CHG) and discharging (DCHG) conditions.

This model serves as a valuable tool for simulation and testing within the MATLAB/Simulink environment. It provides an estimated value for the terminal voltage

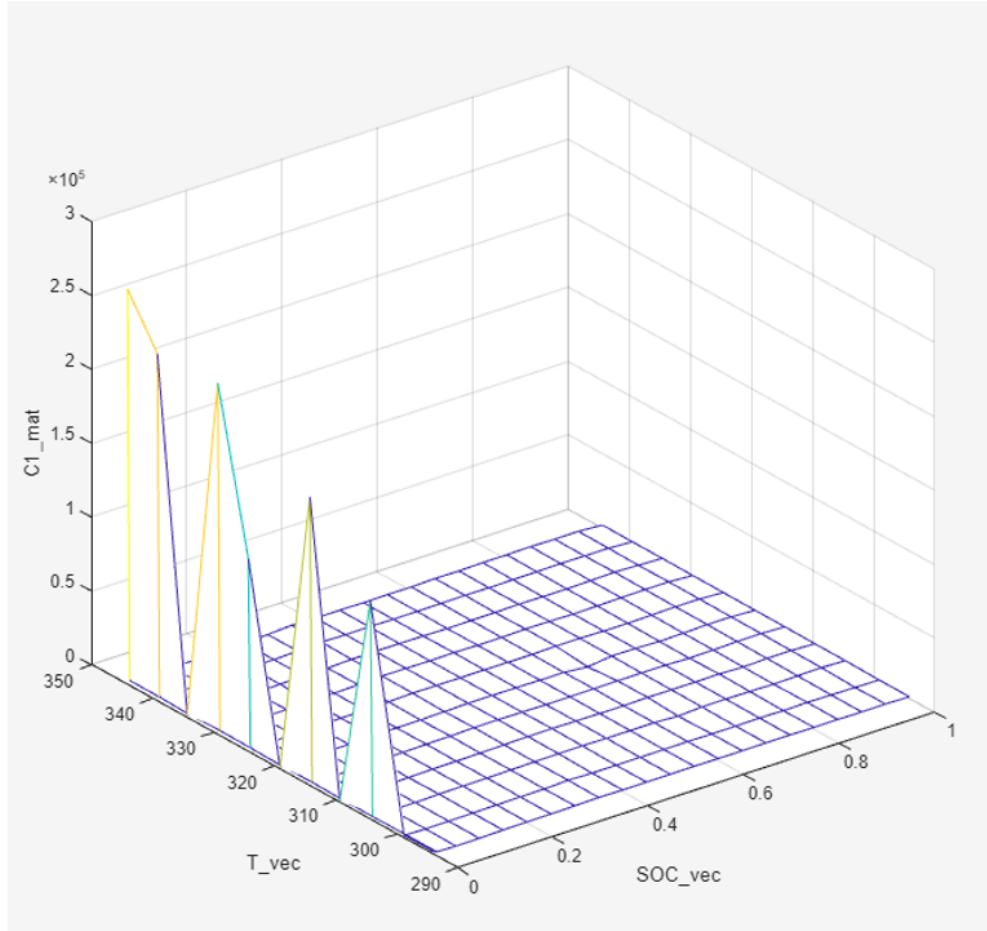


Figure 2.7: Look-up table for estimation of the ECM parameter $C1$

(V_t) that would be observed in an actual battery cell under the given conditions. Furthermore, the model facilitates the evaluation of power losses within the cell by calculating the voltage drops across the various components represented within the 3RC ECM. The BCM model is depicted in Figure 2.8 showing its inputs and outputs as implemented in Simulink.

2.3.4 Extended Kalman Filter for state estimation

When applying feedback in battery state estimation, it's important to account for various sources of error, including state-estimation errors, measurement noise, and modeling inaccuracies. The Kalman filter, an algorithm designed for optimal

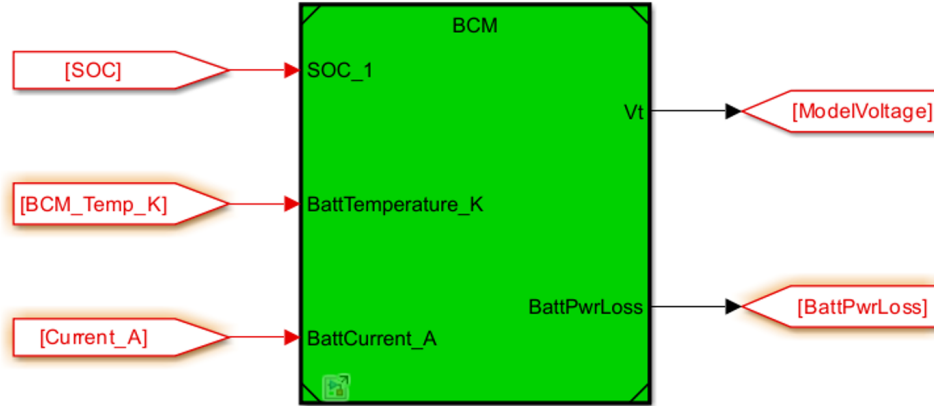


Figure 2.8: The BCM model implemented in Simulink

state estimation, helps manage these uncertainties. Though Kalman filters are typically derived under specific conditions, real-world applications often require modifications. Understanding the mathematical basis of the Kalman filter is crucial, particularly for BMS algorithm engineers, as they need to adapt the algorithm for practical use in environments where the original assumptions may not hold.

While the linear Kalman filter works optimally for systems that can be modeled in state-space form and assumes all noises are white and Gaussian, it doesn't directly apply to the nonlinear ECM. However, the probabilistic inference framework remains valid for nonlinear systems with Gaussian noise, though exact calculations aren't feasible. In such cases, approximation methods like the extended Kalman filter (EKF) are used, which performs linearization of the model at each time step. Despite its limitations, the EKF remains popular and effective when system nonlinearities are moderate.

Figure 2.9 shows the EKF's general overview [7]. In the case of SOC estimation, the C matrix is non-linear as the battery's SOC-OCV relationship is non-linear. Consequently, a Jacobian matrix is produced and is used in the correction step. This replaces the conversion matrix H typically used in the linear Kalman filter by a Jacobian matrix H_j and then, non-linear state transition and measurement functions are used for the prediction and correction respectively.

EKF subsystem blocks:

The EKF was designed by creating four subsystems to carry out the necessary operations required for the estimation of the SOC as shown in Figure 2.10. The

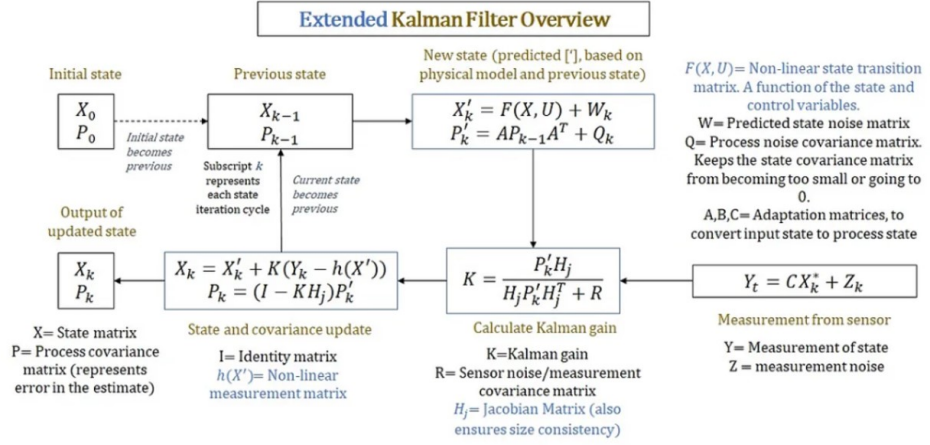


Figure 2.9: The EKF general overview

subsystems are organized as follows:

State-space Jacobian: Consists of multiple lookup tables used for the estimation of the ECM (RC-circuit) parameters such as the values of resistance, capacitance and OCV. Based on these values the matrices A , B and C that are part of the linear state-space system are created and output by the block.

Prediction: This block performs matrix multiplication procedures based on the obtained matrices, the estimated state vector X (SOC, V_1, V_2, V_3) and the covariance of the measurement error P to obtain prediction parameters required for the correction step. In other words, it is a time update step that projects the state and error covariance ahead.

Correction: Performs a measurement update. It uses the terminal voltage error Vt_{error} measured by subtracting the estimated voltage from the measured voltage, in addition to the prediction parameters and the C matrix to perform mathematical operations and update the estimated state vector and covariance of the measurement error.

Delay: Delays input signal that is the state vector by one unit.

scheduling over a near-future time horizon, followed by an update at low frequency to avoid sudden performance losses. Once the power limits are successfully computed, a control mechanism is required to control the output current to the load. In this case, a PI controller was implemented as part of the model-based algorithm for the computation of the power limits.

Three power limits algorithms were designed each with a different time window. The first algorithm (PL1s) provides the instantaneous power limits while the other two implement time-windows covering different time horizons of 10 seconds (PL10s) and 30 seconds (PL30s) within which the battery can safely operate at the calculated power limit. Once the time-window is elapsed the power limits are updated. The time-windows allow the measurement of a predictive non-instantaneous estimate of the power limits.

The reason for designing the same model with three different time-windows is to provide a flexible and adaptable solution for power management in electric vehicles. The instantaneous power limit provides a rapid response to immediate changes in power demand, which can be useful in testing regenerative braking. However, it may not be the most efficient or conservative approach, as it can lead to frequent and potentially excessive power peaks. To address this, longer time-windows of 10 and 30 seconds are introduced. These time-windows allow for a more predictive and conservative approach to power management, enabling the system to anticipate future power demands and adjust the power limits accordingly. By considering the longer-term effects of power usage, the system can better manage factors such as temperature and battery health, ultimately leading to improved overall performance and longevity. Figure 2.11 below depicts the Power Limits algorithm modeled in the Simulink environment, showcasing its inputs and outputs.

The power limits algorithm requires the SOC as estimated by EKF as an input. Additionally, it receives the measured temperature and current required by the load as inputs. Model parameters are also inputs to the power limits algorithm and this includes the values of the resistances and voltages required for the calculation of the power limits.

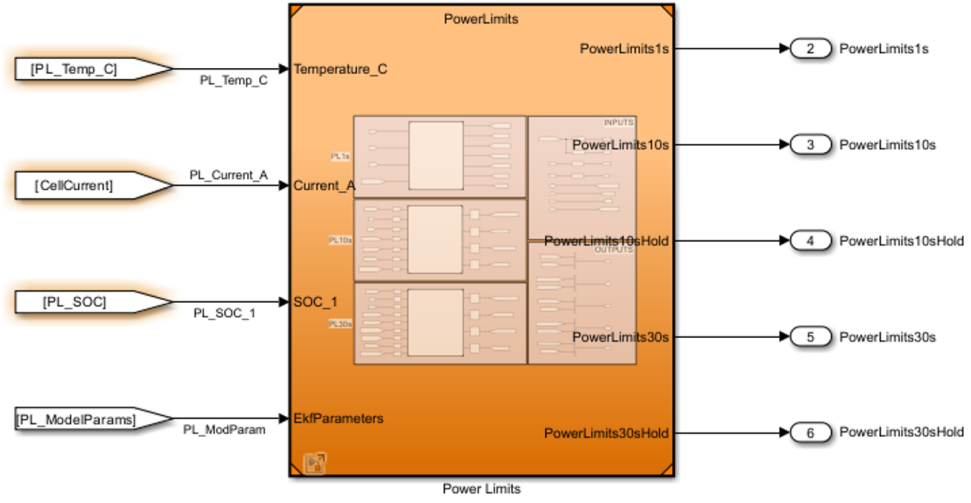


Figure 2.11: The Power Limits algorithm modeled in Simulink

Instantaneous power limits

Joule loss (also known as I^2R loss) refers to energy loss in electrical components due to resistance. It's critical in energy management because excessive losses reduce efficiency and can lead to overheating. Therefore, it is necessary to determine how much energy has been lost in the system due to resistance during a time window. The system can then adjust its power limit or other parameters to compensate for this loss.

The instantaneous power limits algorithm acts as the next step after the EKF algorithm has executed. It obtains the Open Circuit Voltage (OCV) and the equivalent resistance (R_{eq}), as estimated and corrected by the EKF at a given moment and uses these values to calculate the power limits for that specific instant. The value is then updated every second.

The algorithm checks various conditions, such as overvoltage (OV), undervoltage (UV), and thermal limits, to calculate the maximum allowable current for charging and discharging. Additionally, it takes into account the state of charge (SOC) to ensure safe operation based on the available charge in the battery, thus preventing over-charging and under-charging. It calculates the maximum allowable current for charging within a 1-second window based on the OV and the equivalent cell resistance Equation (2.5). Similarly, it calculates the maximum allowable current for discharging within a 1-second window based on UV and the equivalent cell resistance Equation (2.6). This ensures the battery pack does not exceed the

defined safe operating range. To obtain a more conservative, tunable behavior, a PI controller is implemented within the PL10s and PL30s algorithms, which is discussed in the next paragraph.

Equation (2.5)

$$I_{max,CHG} = \frac{V_{OV,lim} - OCV}{R_{Eq}} \quad (2.5)$$

Equation (2.6)

$$I_{max,DCHG} = \frac{OCV - V_{UV,lim}}{R_{Eq}} \quad (2.6)$$

PI controller in power limits

PID is the most prevalent form of feedback control for a wide range of real physical applications. It is typically used in control systems to maintain a certain target (like power, voltage, or temperature). PID is widely used as it is simple, efficient and effective in a wide array of applications. In the context of power limits, it is used for adjusting the current output based on feedback. This is necessary to ensure the system adjusts dynamically to maintain the power limit within acceptable levels.

A PID controller consists of three paths, a proportional, an integral and a derivative path. In the proportional path the error is multiplied by a constant K_p and in the integral path the error is multiplied by a constant K_i and is integrated. While, in the differential path the error is multiplied by a constant K_d and then differentiated. All three paths are then summed together to produce the controller output. The three constants are called gains and can be tuned to obtain the desired behavior. The proportional path mirrors the behaviour of the error in magnitude and direction. The integral sums the error and therefore it is used to remove constant errors such as the steady state error in the control system. Even if the error is small, eventually the summation of this error will be enough to adjust the controller's output. The integral path acts as a memory as it keeps a running total of the input over time, so as long as there's an error in the system the integral output will continue to change and together with the proportional path, they will work to drive the error to zero. The derivative on the other hand, takes into account the rate of change of the error. The faster the error changes the larger is the derivative's path. In other words, it quantifies how fast the output is closing in on the desired value.

However, the effect of the differential path might not be significant in the control process and therefore in some cases it is worth eliminating it. If the differential gain is set to zero then we have a simpler version of the PID controller and we refer to it with the letters of the remaining paths, so a PI controller. This simplification allows the controller to be easier to implement, and test while still meeting design requirements. Figure 2.12 shows simulink's discrete PI controller block as implemented in the 10s-discharge power limits algorithm.



Figure 2.12: The discrete PI controller implemented in Simulink

10s and 30s power limits

The purpose of the computation of the power limits is to ensure the system stays within a predefined power or energy threshold. This is crucial for protecting components and ensuring system stability. PL10s utilizes the power limits calculated by PL1s as the reference power. It additionally takes into consideration the power loss over time in order to enforce new, more conservative power limits. The result is to lower the power output or take other actions to ensure the system remains safe and efficient. The PL10s algorithm ties together different critical tasks needed for energy and power management in a system, ensuring the power stays within safe limits. The power limit set is valid for the duration of the 10-second time-window, such that if the user was to continuously request for 10 seconds the maximum power, equivalent to the power limit set, none of the safety limits would be crossed. Similarly, the 30-second power limits algorithm implements the same concept but instead it considers a longer, more conservative time-window of 30-seconds.

For each reference point in the 10-second window, the algorithm calculates the square of the current and multiplies it by the cell resistance to compute the Joule loss for that time step. The maximum allowable discharge and charge currents

for the 1-second window are used to clamp the 10-second current limit, and the same is done for clamping the 30-second power limit using the output of the PL10s. The reference for the 30-second PI controller is then updated using the clamped 10-second current limit.

Overall, the algorithms manage the power limit references over 10-second and 30-second windows by calculating current limits, Joule losses, and updating indices for the next time-step. The process ensures that the system stays within safe operational limits, adjusting for factors like resistance and cumulative energy losses, which are critical in managing power effectively in systems like the BMS.

2.4.2 State-of-Health

The state-of-health (SOH) is a comparison between the current condition of the battery and its nominal condition defined in its specifications. A battery cell that perfectly matches its specifications has a SOH of 100%. As the battery undergoes multiple charge and discharge cycles it starts to deviate from its specification; therefore, its SOH falls below 100%. A simple way to define the SOH of the battery cell is the ratio between the capacity of the current cell to the capacity of a new unused cell. As the battery ages its capacity decreases, in other words, the amount of charge a cell can hold for a given OCV is reduced. The value of SOH for which the battery must be discarded is arbitrary and depends on the user, but according to IEEE standard 1188.1996, once a Li-ion battery's SOH falls below 80% the battery is no longer usable and must be replaced. Multiple factors influence the SOH of a battery such as increase in cell resistance, decrease in capacity, number of charge/discharge cycles, self-discharge rate, and simply the time elapsed since the battery's production date [8]. Figure 2.13 below shows the SOH modeled in Simulink, showing the model inputs and outputs.

SOH is an important parameter to be considered within the BMS as it signals the end of life of the battery and is strongly intertwined with the SOC; however, this paper does not delve into the methodologies used for estimating the state of health. Instead, it focuses on the EKF and Power Limits algorithms and their integration, which will be discussed in detail in the next chapters.

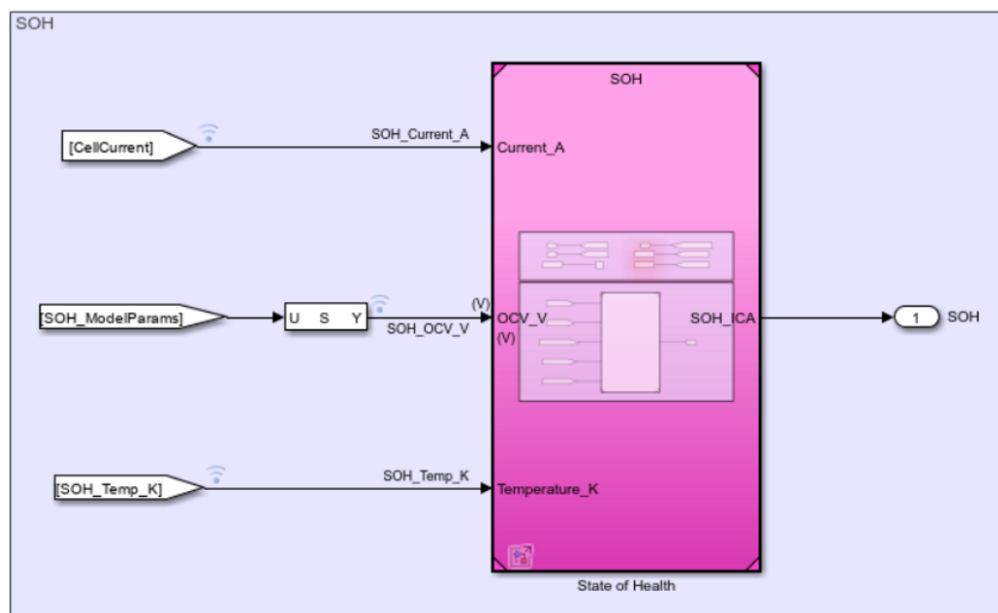


Figure 2.13: SOH algorithm modeled in Simulink

Chapter 3

Code Generation and Optimization

3.1 Code generation: Embedded coder

Embedded Coder® is one of MATLAB's add-on products. It automatically generates C/C++ code from a model for developing platform-specific applications. This automation eliminates the need for manual algorithm coding, reduces the risk of coding errors, and ensures consistent code quality. It also allows for easy reproduction of code, allowing for the optimization implemented to be tested easily. Embedded Coder extends MATLAB Coder and Simulink Coder with advanced optimizations for precise control of generated functions, files, and data. These optimizations improve code efficiency and facilitate integration. It employs processor-specific optimizations and define code generation patterns to enhance code readability, maintainability, and compliance with coding standards. The generated code can be validated through rigorous testing, including Software-In-the-Loop (SIL) and Processor-In-the-Loop (PIL) simulations. There is also the possibility to generate comprehensive reports that include metrics on code size, stack usage and execution time. Additionally, it utilizes code tracing capabilities, tracing the code back to the simulated model to facilitate debugging and analysis. Additionally, it includes complementary products to ensure compliance with industry standards such as ISO 26262, MISRA C/C++, and AUTOSAR. Overall, it is useful in developing production code that takes into account speed, simplicity and memory efficiency. The embedded Real-Time target can be chosen by the user. In this thesis the NUCLEO-f429ZI was selected and used as the development board for the application.

Embedded Coder creates a build directory to store the generated source code, along

with object files, a make-file, and other files produced during code generation. The generated files are shown in Table 3.1.

Embedded Coder's Build Directory	
File	Description
.c	Contains the entry points for all code that implements the model algorithm.
.h	Declares the model's data structures and provides a public interface to the model's entry points and data structures, including access to the real-time model data structure via accessor macros.
private.h	Holds local macros, local data required by the model and subsystems, and any externally defined data imported by the model. This file is included in the generated source files when needed.
types.h	Provides forward declarations for the real-time model and parameter data structures, which may be necessary for reusable function declarations.
rtwtypes.h	Defines data types, structures, and macros required by Embedded Coder, and is used by most other generated code modules.
ert_main.c	A default example of a main program generated by Embedded Coder.

Table 3.1: Embedded Coder's Build Directory

3.1.1 Configuration parameters for code generation

To successfully generate code the model must be configured to meet code generation requirements. First, the solver must be set to a fixed-step solver. In the configuration parameters, embedded coder must be chosen as the target by selecting ert.tlc for the system target file.

Additionally, based on the optimization objectives, certain optimization goals can be prioritized by selecting them through the code generation advisor as shown in Figure 3.1. In this example, MISRA C guidelines and execution efficiency were chosen as the main objectives. MISRA C guidelines are guidelines set for software development using C language. It is useful in producing a safe and reliable code, protecting against language aspects that can affect the application of embedded system. The code generation advisor then runs different checks and provides

suggestions based on the selected objectives. Finally, you can choose to generate code and build the model.

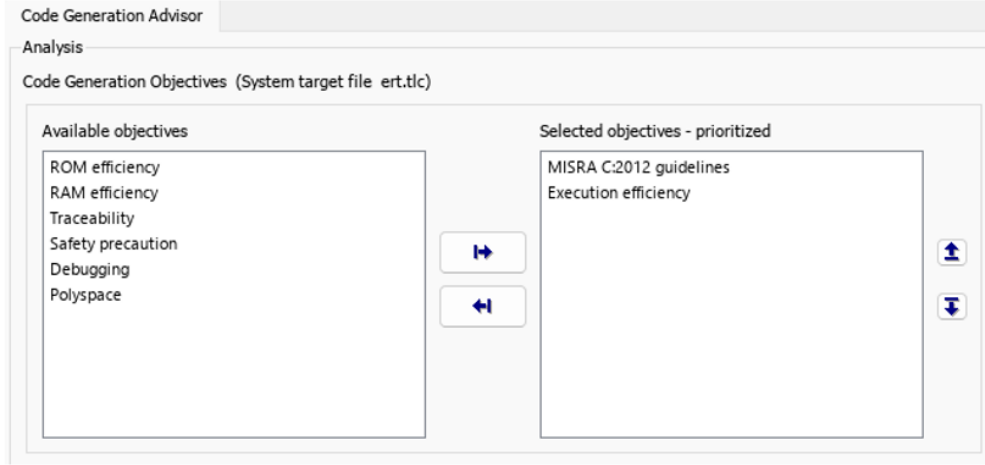


Figure 3.1: Code optimization objectives

All the files generated within the build directory can be visualized in the code pane. Code lines can be traced back to Simulink model elements. Modifications in the model leading to modifications in the code will be highlighted, and the number of modifications in each file will be visible between subsequent runs.

In the code mappings pane, inports, outports, signals, parameters and functions can be visualized. Names of entry-point functions can be modified and different signal storage classes can be selected. For instance, within the code mappings when using 'ImportedExtern', the `model_private.h` file declares the signal as an external variable. This ensures that the extern declaration is accessible to other files within the model. By defining model data externally, the generated code's memory footprint is reduced.

3.1.2 Code optimization objectives

One of the main objectives of code optimization is to reduce the execution time of the algorithm. The execution time of a task is the time interval taken by the system for the whole processing of this task between the start time and the completion time. Minimizing execution time ensures that the BMS can process data and make decisions such as detecting faults, controlling charging and discharging without delays, which is vital for system stability. A faster algorithm allows the BMS to respond more quickly to changes in battery conditions such as rapid changes in load. In addition, the BMS is responsible for managing critical safety functions such as preventing over-charging, over-discharging, and over-heating. Slower algorithms can

introduce risks if the system fails to detect and respond to unsafe conditions quickly enough. Reducing execution time ensures faster fault detection and mitigation, reducing the likelihood of catastrophic battery failures. On the other hand, the BMS is responsible for communication with other systems, performing diagnostics, and logging data. Minimizing the execution time of algorithms allows the BMS to handle multiple tasks without introducing delays or missing important events while maintaining proper scheduling.

Code Generation Advisor

When preparing for code generation, it is important to consider how application goals, like efficiency, execution time and debugging align with specific code generation settings. The Configuration Parameters settings control both the model's simulation behavior and the code that is generated.

The Code Generation Advisor, included in MATLAB, can be used to review the model before code generation or as part of the process itself. When reviewing the model in advance, you can choose which parts (model, subsystem, or referenced model) the advisor will inspect. When reviewing during code generation, the entire system is assessed. The advisor consults the 'Recommended Settings Summary for Model Configuration Parameters' to determine parameter values that align with the objectives.

By setting a code generation objective and running the Code Generation Advisor, the advisor outlines guidance on meeting that objective. While the Advisor itself doesn't alter the code, it suggests model changes that can be implemented before regenerating the code. After a model is modified and the code is regenerated, the code generation advisor includes comments noting the set objectives, the checks the advisor ran, and recommendations for optimizing parameters. Figure 3.2 shows the Code Generation Advisor window for PL10s after it carried out the checks and generated the recommended modifications.

Model Advisor

Another useful MATLAB tool is the Model Advisor that can be used to review the model or subsystem for conditions and configuration settings that may lead to inaccurate or inefficient simulations, helping to verify compliance with industry standards and guidelines. Using the Model Advisor promotes consistency in modeling practices across projects and teams.

After analyzing the model, the Model Advisor generates a report that highlights suboptimal settings, modeling techniques, and provides suggested improvements where applicable. For Embedded Coder, Model Advisor checks recommendations for C/C++ production code, identifying blocks not ideal for deployment and configuration parameters that may generate inefficient code.

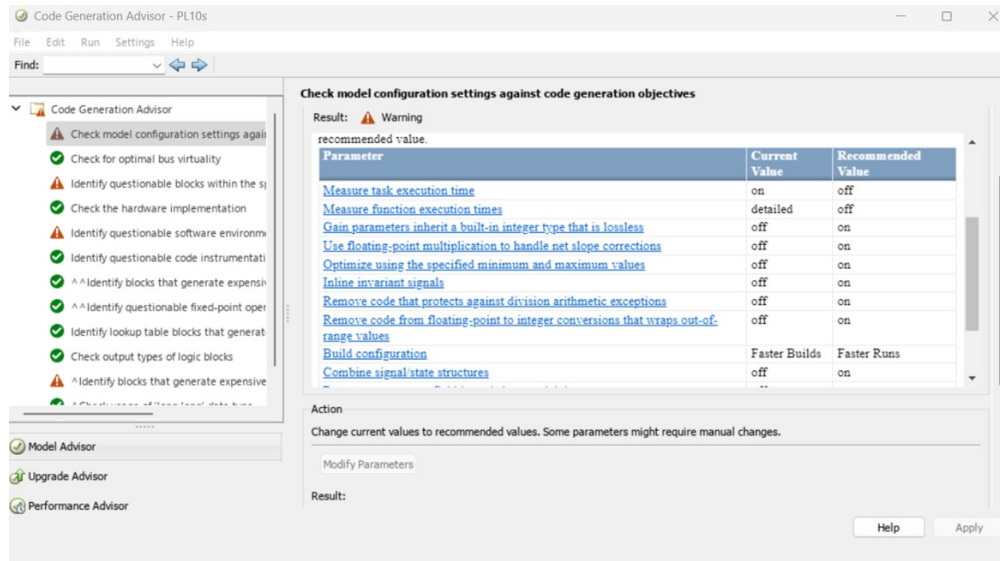


Figure 3.2: Code generation advisor

Overall, both the Code advisor and Model advisor were used to enhance the overall development efficiency, meet code generation objectives, comply with industry standards, detect potential errors and improve code quality and consistency across projects. Each suggested change is meticulously evaluated and implemented while preserving the algorithm's underlying logic.

Code generation Report

When the option to generate a report is enabled, the Code generation report can be generated along with the code generated by Embedded Coder as shown in Figure 3.3. The Static Code Metrics report is a section included in the Code generation report. It provides generated code statistics such as the number of files and lines of code in each file, as well as the number of lines of code and stack usage per function. It also includes information regarding the global constants in the generated code and their size, in addition to the function metrics such as stack size, number of inputs, number of outputs and number of locals. The Static Code Metrics report is a useful tool in investigating how different methods of optimization impact the code, in order to assess the applied optimizations and perform modifications on the model before deploying the code implementation on the target.

Another valuable report that can be included within the code generation report is the code interface report. This report serves as a cornerstone in streamlining the integration process by thoroughly documenting the model's entry-point functions

and interface data. As depicted in Figure 3.4, the report details the function interfaces, such as the `model_initialize`, `model_step`, and `model_terminate` functions.

Static Code Metrics Report

The static code metrics report provides statistics of the generated code. Metrics are estimated from static analysis of the generated code using the C data types specified in the "Device details" section of the **Configuration Parameter > Hardware Implementation** pane: **char** 8, **short** 16, **int** 32, **long** 32, **float** 32, **double** 64, **pointer** 32 bits. If your model contains a Variant block, the Static Code Metrics Report does not contain data for the inactive variant. Actual object code metrics might differ due to target specific compiler and platform settings. Consult the **Code Generation Advisor** for options to improve code efficiency.

Table of Contents

1. File Information

2. Global Variables

3. Function Information

1. File Information

hide

[-] Summary

Number of .c files : 1

Number of .h files : 5

Lines of code : 515

Lines : 1,650

[-] File details

File Name	Lines of Code	Lines	Generated On
PL10s.c	346	883	12/17/2024 3:26 PM
PL10s.h	86	518	12/17/2024 3:26 PM
rtwtypes.h	67	150	12/17/2024 3:26 PM
rtmodel.h	7	35	12/17/2024 3:26 PM
PL10s_private.h	5	31	12/17/2024 3:26 PM
PL10s_types.h	4	33	12/17/2024 3:26 PM

2. Global Variables

hide

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[*] PL10s_DW	1928	156	148
[*] PL10s_U	28	13	13
[*] PL10s_Y	24	6	6
[*] PL10s_B	16	82	82

Figure 3.3: Static Code Metrics Report

2.1 Initialize Functions
Initialize entry-point functions implement startup behavior. In a model, Initialize Function blocks represent startup behavior explicitly.
PL10s_initialize Initialization entry point of generated code <pre>#include "PL10s.h" void PL10s_initialize(void)</pre>
2.2 Terminate Functions
Terminate entry-point functions implement shutdown behavior. In a model, Terminate Function blocks represent shutdown behavior explicitly.
PL10s_terminate Termination entry point of generated code <pre>#include "PL10s.h" void PL10s_terminate(void)</pre>
2.3 Periodic Functions
Periodic entry-point functions implement model behavior that occurs at a fixed sampling rate. For a rate-based model, the code generator produces a model, Function Call Subsystem blocks that specify a sampling rate represent periodic functions.
PL10s_step Output entry point of generated code. Must be called periodically, every 1 seconds. <pre>#include "PL10s.h" void PL10s_step(void)</pre>

Figure 3.4: Code Interface Report

3.2 Implementation of the code optimizations

This section delves into some of the optimization techniques applied to the generated code, analyzing their impact on code footprint, memory usage, and execution time. Each optimization technique is examined in detail, discussing its implementation and the resulting trade-offs in terms of performance and resource consumption.

Memset to initialize float and doubles to 0.0

One of the code optimization methods used in Simulink involves enabling the use of `memset` to initialize float and doubles to 0.0. This option in MATLAB controls how the initialization of floating-point variables is handled when setting them to zero. Firstly, `memset` is a standard library function in C/C++ used to set a block of memory to a specific byte value, often zero. When applied to memory allocated for float and double variables, it writes the zero-bit pattern across the entire memory space used by those variables. For floating-point numbers, zero is represented by a specific bit pattern (all bits set to zero). If this option is enabled, `memset` simply fills the memory for float and double variables with this bit pattern (all zeroes), initializing them as zero values directly, without additional code. When the option is enabled, `memset` is used for setting memory for floating-point variables to zero, which is faster because it does it in one step for the entire memory block. This eliminates the need to generate extra initialization code for each floating-point variable. In addition, this approach applies the same initialization regardless of floating-point type, directly setting them to the zero-bit pattern without relying on floating-point operations. Disabling this option makes the code generate explicit initialization instructions for each floating-point variable. This means it writes 0.0 to each floating-point variable individually in code, which can add overhead and reduce performance slightly, especially if there are many such variables. Explicit initialization could also increase code size and might involve additional floating-point operations, which are generally more costly than a simple `memset` in terms of processing time.

Removing internal data zero initialization and root-level I/O zero initialization

Simulink's configuration parameters also offer the option to remove internal data zero initialization and to remove root-level I/O zero initialization. Internal data includes variables or data within functions, subsystems, or blocks that are not directly accessible from outside the model. Examples might include temporary variables used for calculations, intermediate data, or internal state data within the model. Root-Level I/O Data refers to the input and output data at the root level of the model, which is often interfaced with external systems or components.

Root-level I/O data could include inputs coming into the model from external sensors or controllers, as well as outputs that go from the model to other parts of the system. By default, MATLAB initializes both internal data and root-level I/O data to zero for safety. This ensures that if any data is accidentally uninitialized, it won't contain unpredictable values, which could lead to unreliable or unsafe behavior in the generated code; however, this might be a redundant process. In many embedded systems, the entire memory (RAM) is cleared to zero during the system's boot process. If the hardware or environment already guarantees that all memory starts at zero, explicitly setting values to zero in generated code becomes redundant, as the data is already zeroed out before code execution. Removing initialization code reduces the startup time, as the code no longer needs to iterate over each variable to set it to zero leading to increased execution speed. This is especially important in applications where initialization time impacts system performance, like the BMS which operates as a real-time systems. In addition, removing unnecessary initialization code reduces the size of the generated code which preserves the memory resources.

The result of removing root-level I/O zero initialization is shown in Figure 3.5. The lines highlighted in red indicate that the lines have been eliminated due to the optimization implemented. As a result, the lines that use the `memset` function to set the entire memory block of the input and the output structures to zero is removed.

Similarly, the result of removing internal data zero initialization is shown in Figure 3.6. The variable `xkalman_DW` is used to store internal data and line 498 is writing the value zero to each byte of the memory block using the `memset` function. This line is also removed as part of the optimization.

In STM32 the BSS section of memory is initialized. The BSS section is a memory region allocated for uninitialized global and static variables. The reset handler iterates through this section and sets each memory location to zero. The linker script, a configuration file used by the linker during the build process, defines the memory layout of the program. It specifies the location of the BSS section in RAM and ensures that it is placed in a region that is cleared during the reset process.

Data type support

A quick way to further optimize the code is to remove unnecessary data support through the configuration parameters in Simulink as shown in Figure 3.7. There is the option to include support for various data types. These options let you enable or suppress the generation of floating-point, non-finite, and complex numbers. Since the model requires the generation of floating-point numbers only, the other options can be deselected. Support for non-finite numbers option is disabled as well as support for complex numbers.

```

489  /* Model initialize function */
490  void Kalman_initialize(void)
491  {
492      /* Registration code */
493
494      /* external inputs */
495      (void)memset(&xKalman_U, 0, sizeof(ExtU_Kalman_t));
496
497      /* external outputs */
498      (void)memset(&xKalman_Y, 0, sizeof(ExtY_Kalman_t));
499

```

Figure 3.5: Removing root-level I/O initialization

```

489  /* Model initialize function */
490  void Kalman_initialize(void)
491  {
492      /* Registration code */
493
494      /* initialize error status */
495      rtmSetErrorStatus(xKalman_M, (NULL));
496
497      /* states (dwork) */
498      (void) memset((void *)&xKalman_DW, 0,
499                  sizeof(DW_Kalman_t));
500

```

Figure 3.6: Removing internal data zero initialization

Other options include support for absolute and continuous time. Certain blocks require the value of either absolute time or elapsed time such as time elapsed between two trigger events. These related options determine how the ERT target provides absolute or elapsed time values to blocks in the model. By default, the ERT target generates and maintains integer counters, if a block in the model requires absolute or elapsed time values. The target does not generate the counters if model blocks do not use time values. However, in this case none of the algorithms require absolute time support.

If support for the continuous-time option is selected, the ERT target supports code generation for continuous-time blocks. By default, this option is deselected, and the build process generates an error if any continuous-time blocks are present in the model. In this case, none of the models contain continuous-time blocks. For example, the integrator blocks used in power limits are all discrete-time integrator blocks; therefore, this option can be safely disabled. Removing unnecessary data support results in a more efficient code, since the generated code will not need to allocate memory for the counters or perform the necessary calculations to update

them.

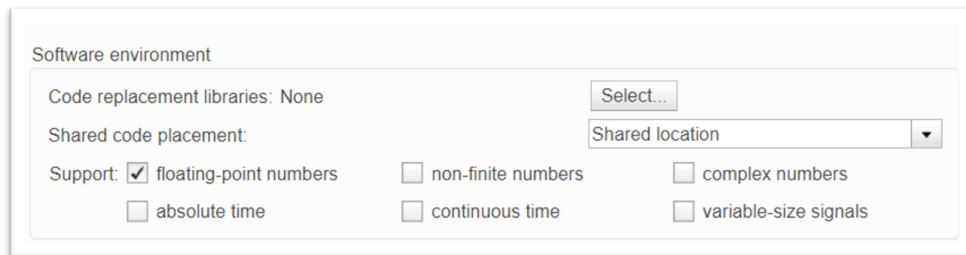


Figure 3.7: Configuration parameters section showing the data type supports selected

Conditional branch execution

In MATLAB Simulink, the "Conditional input branch execution" optimization option enhances the efficiency of models that use 'Switch blocks' by selectively executing only the required input branch. Simulink only evaluates the blocks required to compute the control input and the data input that will be selected by the Switch block at each time step. Based on the control input's value, Simulink determines which branch to execute and skips the other branches, saving computational resources.

When 'Conditional Input Branch Execution is enabled', Simulink generates code that only computes the active branches feeding into the switch block, based on the control signal. This means that in the generated code, only the necessary calculations are performed instead of computing both branches and then discarding the unused one. For example, without this optimization, the generated code would likely include both branches, calculating both paths every time the code runs through that section, even though only one branch is actually needed. With conditional execution enabled, the generated code will include logic to skip the unselected branches, reducing the computational load and making the code more efficient.

By only including code for the branch that is actively selected, this optimization reduces the code execution time since fewer instructions are executed. Additionally, if certain branches contain complex calculations or function calls, excluding them from the generated code may lead to a reduction in code size as well. For embedded systems or real-time applications, where both execution speed and memory usage are critical, this can be a significant advantage.

This optimization has some potential trade-offs. While enabling this option usually improves efficiency, there are cases where it may have limitations. For instance,

if the model is designed to have side effects in each branch such as updates to internal states or variables, conditional branch execution may not execute those side effects in inactive branches. Therefore, conditional branch execution should be carefully tested to ensure that the behavior meets application requirements.

Signal storage reuse

The ‘Signal storage reuse’ option in the optimization settings allows for more efficient memory usage in the generated code by reusing memory buffers for signals. This setting can have a significant impact on the memory usage of the generated code. Signal storage reuse allows the code generator to reuse memory locations for intermediate results and block outputs where possible. Instead of allocating separate memory for every signal or block output, memory can be shared across signals that are not simultaneously active, reducing overall memory consumption. This can be especially beneficial in embedded systems where memory is limited, as it minimizes the amount of RAM used for signal storage.

Selecting ‘Signal storage reuse’ also enables other options such as ‘local block outputs’, ‘Reuse local block outputs’, ‘Reuse global block outputs’, and ‘Eliminate superfluous local variables (expression folding)’. When the ‘Local block output’ option is enabled, the code generator declares block output signals as local variables within functions rather than global variables. Local variables are generally preferred in embedded systems, as they reduce global memory usage and can lead to faster access times, since local variables are typically allocated on the stack.

With ‘Reuse local block outputs’ option selected, the code generator reuses memory for local signals as much as possible. In other words, a single memory location can be shared by multiple signals, provided they are not used concurrently. This leads to a reduction in the number of unique memory locations needed, lowering RAM requirements. Similarly, with ‘Reuse global block outputs’, the code generator reuses memory for global signals whenever it is possible by sharing memory locations between signals. Figure 3.8 shows the effect that this optimization has on the code. The PL10s source file depicts how local block outputs such as ‘PL10s_Y.DCHG10sNotHold’ is enabled and reused throughout the code.

On the other hand, ‘Expression folding’ is an optimization technique that eliminates intermediate variables when they are unnecessary. Instead of creating separate variables for each part of a computation, it combines multiple expressions into a single calculation. This reduces the number of temporary variables needed, further optimizing memory usage and simplifying the generated code.

Reusing local variables and minimizing global memory usage allows the code to execute faster. Local variables, especially those stored in registers or on the stack, can be accessed faster than global variables. It also reduces the number of instructions in the generated code, potentially making the code execution faster

since fewer operations are needed to compute results. Furthermore, when less memory is used, the number of times the memory needs to be accessed is reduced. This results in improved performance, especially in RAM-constrained environments.

```

96  /* MinMax: '<Root>/Min6' incorporates:
97  *   DiscreteIntegrator: '<S44>/Integrator'
98  *   Gain: '<S49>/Proportional Gain'
99  *   Sum: '<S53>/Sum'
100  */
101  PL10s_Y.DCHG10sNotHold = 0.02F * rtb_Integrator_b + PL10s_DW.Integrator_DSTATE;
102
103  /* DeadZone: '<S36>/DeadZone' */
104  if (PL10s_Y.DCHG10sNotHold > 1.0F) {
105      rtb_Integrator_b = PL10s_Y.DCHG10sNotHold - 1.0F;
106
107  /* Switch: '<S34>/Switch1' incorporates:
108  *   Constant: '<S34>/Constant'
109  */
110      tmp = 1;
111  } else {
112      if (PL10s_Y.DCHG10sNotHold >= 0.0F) {
113          rtb_Integrator_b = 0.0F;
114      } else {
115          rtb_Integrator_b = PL10s_Y.DCHG10sNotHold;
116      }
117
118  /* Switch: '<S34>/Switch1' incorporates:
119  *   Constant: '<S34>/Constant2'
120  */
121      tmp = -1;
122  }

```

Figure 3.8: Enabling and reusing local block outputs in PL10s source file

Block reduction

Enabling ‘Block Reduction’ in Simulink’s configuration parameters helps streamline the model by optimizing specific types of blocks and removing unnecessary operations. This reduces memory usage, enhances execution speed, and improves the efficiency of the generated code, without altering the appearance or behavior of the model itself. This includes removal of ‘Redundant Type Conversions’, ‘Dead Code Elimination’ and removal of ‘Fast-to-Slow Rate Transition Blocks’ in a single-tasking system. Redundant type conversions occur when a value is converted from one data type to another and then back to the original type without any real benefit. Simulink identifies these unnecessary type conversions and removes them, reducing the number of operations required to execute the model. This results in fewer CPU cycles and reduced memory footprint, meaning faster and more compact code that improves execution efficiency. With dead code elimination, the model is analyzed to detect blocks or segments that have no effect on the output and thus,

they are removed from both the simulation and generated code. Consequently, this produces a leaner code.

As for the ‘Rate transition’ blocks, they are typically used in multitasking systems to manage data integrity when signals are transferred between blocks operating at different sample rates. However, in a single-tasking system (where all operations run at the same rate), these blocks are redundant. Simulink identifies and removes fast-to-slow rate transition blocks that are unnecessary in a single-tasking system since the rate synchronization is already inherent in the single-tasking operation. This reduces the number of function calls and eliminates unnecessary buffer allocations associated with rate transitions, resulting in faster code execution and lower memory requirements [4].

Tunable Parameters

Tunable parameters allow modification of parameter values during execution, without having to rebuild the code or re-flash the hardware. This is especially useful for adjusting calibration values and other parameters in real time while the system is running. On the flip side, the values of inlined parameters are hardcoded into the generated code. To modify any parameter, the code must be rebuilt and regenerated, which can be time consuming and inconvenient during testing and application. For instance, having the parameter related to the battery pack capacity set to “tunable”, allows the pack capacity value to be updated in real time without having to rebuild the code. Setting the default parameter behavior to tunable, causes the model parameters to appear in the generated code as global variables. This allows the parameters to be accessed by all parts of the program, not only inside a particular function. The parameters are then packed in a unique structure. Furthermore, the parameters are assigned to their values in a separate data file. In a nutshell, the usage of tunable parameters provides significant advantages in terms of flexibility, real-time tuning and debugging. It is especially beneficial during testing and calibration stages, or when the system behavior needs to be modified without recompiling the code.

3.2.1 Data management and version control

Data Dictionary

A great tool that proved to be useful in managing data and storing configurations for several Simulink models was the data dictionary. Data dictionaries can store global design data such as signals, parameters, or global data objects belonging to several Simulink models that use the base workspace. They can also store the model configurations which can then allow multiple models to easily share the same configurations.

The use of a data dictionary in Simulink offers several advantages. It helps prevent clutter and overwriting of variables in the base workspace by keeping the variables organized and separate. Additionally, it allows for sourcing data from different dictionaries for different models, eliminating conflicts that may arise from identical variable names. The data dictionary also provides built-in functionality for tracking changes and comparing different versions, making it easier to manage modifications. Furthermore, by referencing other data dictionaries, it supports the creation of a data hierarchy, improving both readability and memory efficiency.

Simulink also offers a robust mechanism for managing and sharing configuration parameters by using Data Dictionaries. Freestanding configuration sets, stored in the Data Dictionary, can be referenced by multiple models, allowing for centralized parameter management and easy updates. This approach facilitates hierarchical configuration management within models, enabling inheritance of configurations down the model hierarchy. Furthermore, by storing configuration sets within the Data Dictionary, users can change the parameter values in the configuration modifying the data dictionary file. Models that are connected to the data dictionary and its referenced configuration use the modified values without altering the model files themselves. This flexibility enhances model reusability and simplifies parameter updates across multiple projects. Figure 3.9 shows one of the referenced configurations stored in Simulink data dictionary. It is set to “active” and is shared by several models.

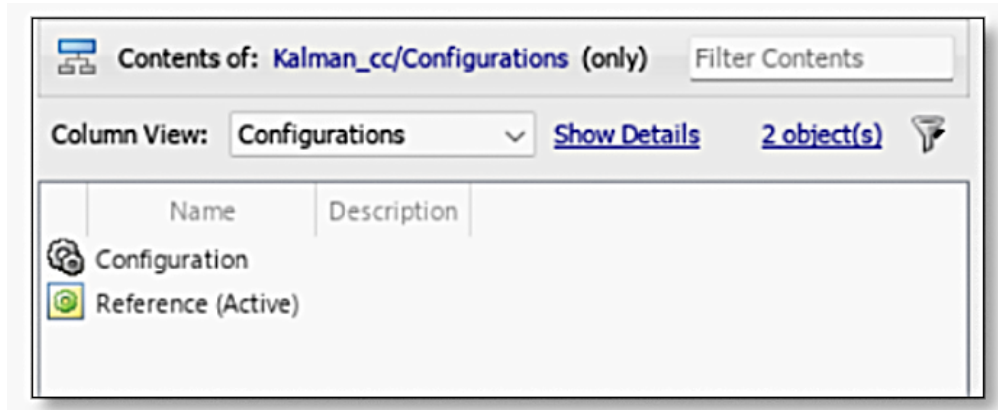


Figure 3.9: Simulink referenced configuration activated

Git and GitHub

One of the tools that were crucial in management and code development were Git and GitHub. GIT plays a pivotal role in version control, collaborative coding, and efficient project management. Git is an open-source version control tool that

can be installed locally on a personal computer. Git is useful in keeping track of the changes in code at specific points of time, thus creating a version history that maps the software development process. The code can also be shared with multiple users working on the same project, and all authorized users can keep track of the code in history and modify it asynchronously. GitHub on the other hand, is a service built to run Git in the cloud. GitHub allows users to create remote, public-facing repositories on the cloud. A repository, or "repo" for short, is the coding project files and the revision history for each file. GitHub allows the user to gain user authentication tools. This prevents remote users from accessing the local Git installation preventing them from taking control over the repository and commit history.

As an example, a repository on GitHub can be created by the user to store all the files, including current and past versions, then other collaborators working on the project can gain access to this 'repo' as well. Each user can create a branch (a separate development area), where the collaborators can work independently. Once the work is done, collaborators can make a pull request asking to merge the branches with the main branch. A very useful tool that aided the process of managing git repositories is Git extensions, shown in Figure 3.10. It enables the user to commit changes, manage branching and merging, track changes and compare source code [9].

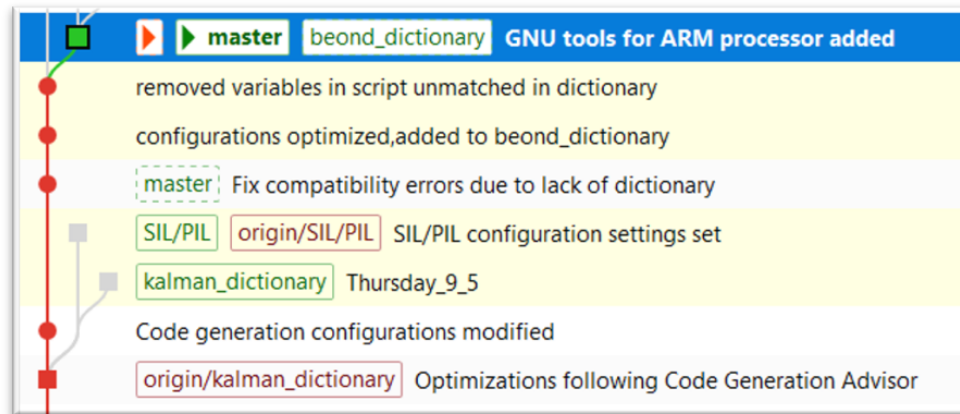


Figure 3.10: Git extensions branches

Chapter 4

System Architecture

4.1 Laboratory test bench

By taking a step backward and viewing things from a broader perspective, we can take a look at the system architecture. The system is made up of several electrical components both at the hardware and software levels. Starting with the main component, that is the battery pack managed and controlled by the BMS, programmed with the necessary firmware. Moving on to other essential supporting components such as the slave board and the shunt sensor. Finally, the components related to the interface system consisting of the CAN bus as well as the SPI for managing communication between components within the system.

Additionally, for ensuring safe conditions and correct functioning the system architecture comprises of multiple contactors, specifically it consists of the negative contactor, the positive contactor and the pre-charge contactor, as well as an emergency button that serves as a safety-net for opening the interlock. The contactors are connected to the inverter that delivers the requested current to the load. The current load is controlled via PC through a software designed by BeonD, which enables configuration of any desired current profile to be requested from or delivered to the battery. Figure 4.1 depicts part of the setup of the test bench located in BALF laboratory, showcasing some of the components that comprise the system including the BMS, contactors, safety-net and shunt sensor.

4.1.1 The battery pack

Recalling that the goal of this thesis is to integrate model-based algorithms and deploy them on a custom BMS, this brings us to the most essential and primary component within the system architecture, that is the battery pack. The battery

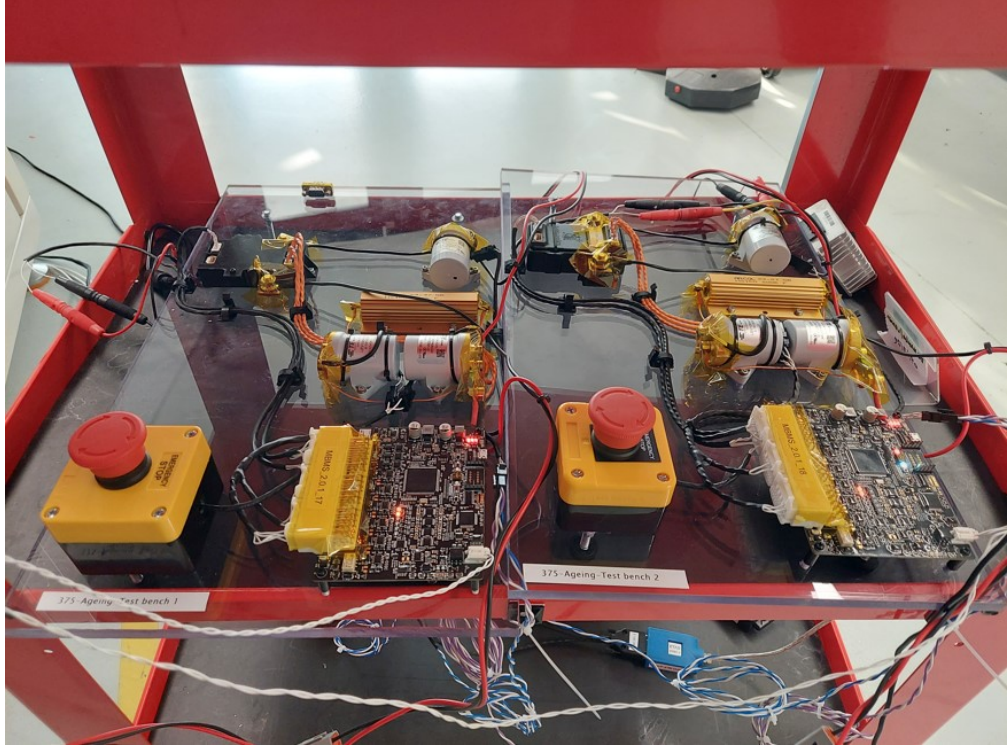


Figure 4.1: BMS test benches in BALF

used for carrying out the testing and validation in BALF laboratory was a lithium-ion battery pack made up of 14 series and 2 parallel battery cells as shown in Figure 4.2.

Each battery cell stores and discharges electrical energy through chemical reactions, converting chemical energy into electrical energy and vice versa when discharging. Cells can provide a range of voltages that vary depending on its SOC, temperature and is limited by the cell's nominal voltage. Every cell is designed with a specific nominal charge capacity, that is the quantity of charge in ampere-hours that it can hold.

To obtain a high-power battery, both the current and voltage in the battery pack must be high, while taking into account both safety and the economic aspect. The voltage range of a cell depends on the chemistry of the cell; therefore, it is fixed. Consequently, a high voltage battery pack would consist of multiple cells in series. Approximately, the battery pack voltage can be calculated as $v_{pack} = N_s \times v_{cell}$, where N_s is the number of cells in series. Regarding the current, each cell is designed to operate under a specific current limit. So, placing cells in parallel makes the pack current the sum of currents passing through cells in parallel, such as $i_{pack} = N_p \times i_{cell}$, where N_p is the number of cells in parallel. The choice of the

number of cells in parallel versus in series depends on the power requirements.



Figure 4.2: BALF's test bench lithium-ion battery packs connected to the slave boards

4.1.2 Slave board

The slave board (Figure 4.3) plays a crucial role in monitoring the health and performance of individual cells within the battery pack. Connected directly to the battery, it measures up to 24 cell voltages in series, enabling precise voltage monitoring. A passive balancing circuit is integrated into the slave to mitigate voltage imbalances between cells, ensuring optimal performance and extending battery life. The slave transmits this critical cell data to the Master BMS via an isolated SPI channel, ensuring secure and reliable communication. In addition to voltage monitoring, the slave also monitors cell temperatures and participates in cell balancing processes as directed by the Master BMS. By continuously gathering and transmitting this detailed cell-level information, the slave empowers the Master BMS to make informed decisions and optimize the overall battery management strategy [10].

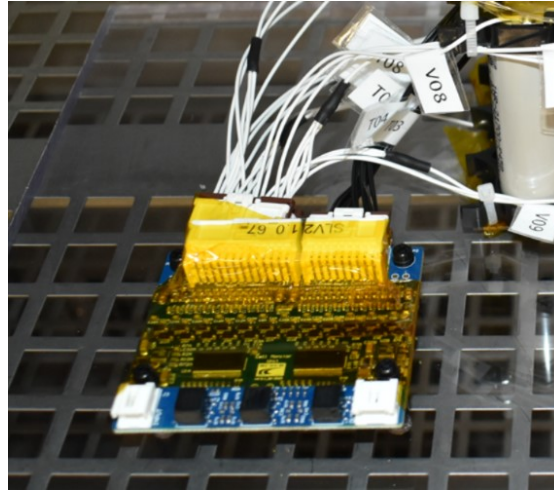


Figure 4.3: Slave board connected to the battery pack's cells

4.1.3 Shunt sensor

A shunt sensor is a precise, low-resistance component used to measure accurately the current. For precise measurements of current and voltage at critical points within the battery system such as across the contactors, the Isabellenhuette shunt sensor depicted in Figure 4.4 is used. The Isabellenhuette shunt sensor is a highly accurate and reliable device specifically designed for precise current and voltage measurement. The sensor utilizes a high-precision shunt resistor to measure the voltage drop across it, which is directly proportional to the current flowing through the circuit. This enables accurate current measurement, even at low current levels. Current shunts have several important characteristics; for instance, they do not introduce an offset at zero current, minimizing drift in measurements, though offsets may arise from accompanying electronics. On the downside, temperature variations can alter the shunt's resistance affecting accuracy, and it also results in minor energy losses. Therefore, the sensor incorporates temperature compensation circuitry to ensure accurate measurements across a wide operating temperature range. This is crucial in automotive applications where temperature fluctuations can significantly impact sensor performance [11].

The sensor is also equipped with galvanic isolation, which prevents potential ground loops and noise interference, ensuring accurate and reliable measurements. The sensor can transmit the measured current and voltage values, as well as other relevant data, to the microcontroller over the CAN bus while using a 16-bit ADC for generating digital signals [12].



Figure 4.4: Isabellenhuette shunt sensor

4.1.4 Master and slave controller

The Master Battery Management System (Master BMS) serves as the central hub for the entire battery pack, overseeing its operation and ensuring safe and efficient performance. It houses the core algorithms responsible for calculating critical battery parameters such as State of Charge (SOC), State of Health (SOH), and Power Limits. These algorithms enable the BMS to accurately assess the battery's current condition and predict its future behavior. Additionally, the Master BMS controls vital components like battery chargers and contactors, optimizing their operation to maintain the battery within its safe operating window. By continuously monitoring and adjusting these parameters, the Master BMS safeguards the battery from potential hazards such as over-charging, over-discharging, and excessive temperature fluctuations.

The PCB (Figure 4.5) is designed by BeonD to play the role of the master BMS. It incorporates the STM32 NUCLEO-F429ZI board that houses the main processing unit. It is a powerful and versatile device that offers a wide range of features and benefits for automotive BMS applications. It boasts a powerful ARM Cortex-M4 core, enabling efficient execution of complex algorithms and real-time processing of battery data. In addition, it provides a rich array of communication interfaces, including CAN, SPI, I2C, USART, and USB, facilitating seamless communication with various sensors, actuators, and other BMS components [13]. The integrated 12-bit ADCs enable accurate measurement of cell voltages, temperatures, and other analog signals, crucial for precise battery monitoring. It is also equipped with multiple timers and counters providing precise timing control for various tasks.

From the security and protection perspective, the STM32 incorporates secure memory and cryptographic hardware accelerators, ensuring data integrity and protection against unauthorized access.

In a nutshell, the STM32 features high performance, flexibility, robustness and security. By leveraging the capabilities of this microcontroller, a robust and efficient BMS can be developed ensuring safe and reliable operation of the battery pack [14].

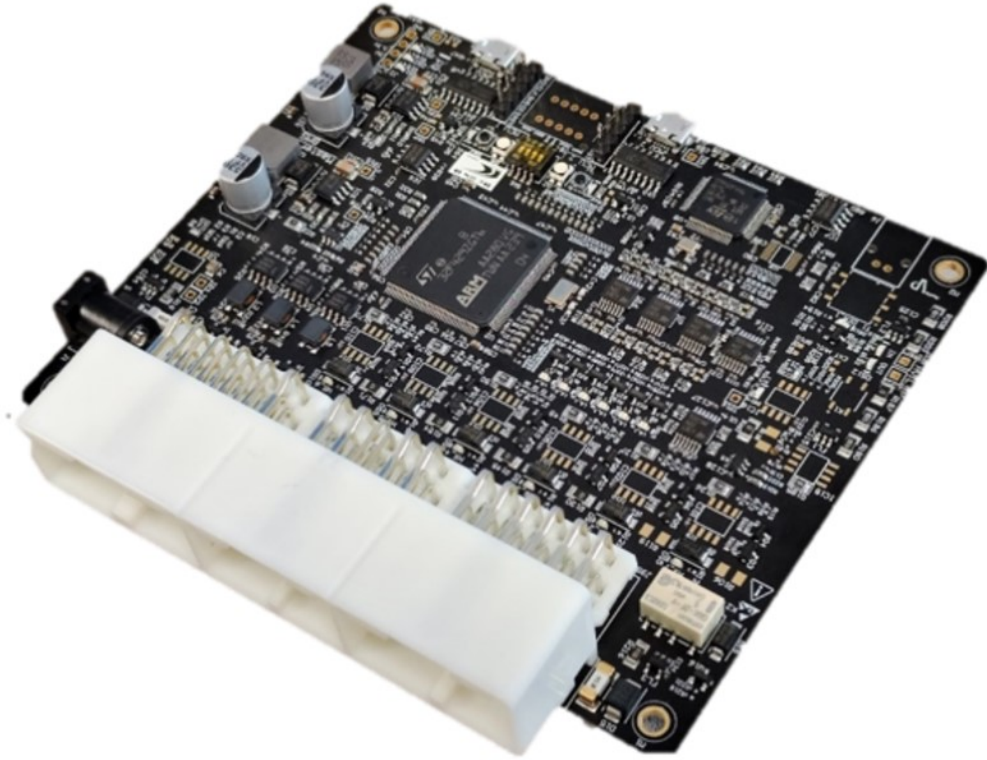


Figure 4.5: BEOND’s PCB incorporating the master and slave controllers

4.1.5 Contactors and safety net

Systems operating at higher voltages often encounter a significant challenge during initial power-up. This phenomenon is known as inrush current, and it is particularly pronounced in circuits with substantial capacitive loads. Inrush current can stress or even damage components if not properly managed. This issue is particularly critical in modern EVs, which operate at high voltages. Frequent on/off cycles throughout the day, characteristic of EV operation, exacerbate this problem by repeatedly subjecting the system to inrush currents.

The standard approach to mitigating inrush current is pre-charging. This technique aims to safeguard electrical and electronic components from damage, ensuring the long-term reliability and trouble-free operation of the vehicle and its systems. By preventing excessive current surges, pre-charging reduces the risk of safety hazards such as fire or electric shock. Furthermore, pre-charging creates a more stable environment for diagnostics allowing for early detection of potential problems before they escalate into more serious damage. As voltage rises to reach a steady

state, pre-charging is no longer needed. It can be taken out of the circuit, normally through some automatic method such as through a high-current relay such as a contactor that can disable the system when required [15].

The contactors within the battery are orchestrated by the BMS. The pre-charge circuit consists of a separate, smaller contactor connected in series with a resistor. These two components are then wired in parallel with the main contactor, along the positive side as shown in Figure 4.6. The resistor's role is to make the charging of the capacitor more gradual; therefore, allowing the voltage to rise relatively slowly and in a controlled manner. Once the voltage reaches steady state, the pre-charge is disabled.

The process in detail is as follows: Initially, the battery pack is disconnected from the load with all contactors in the open position. The negative contactor is activated first, connecting the battery pack's negative terminal to the load's negative terminal. Subsequently, the pre-charge contactor is activated. This connects a pre-charge resistor in series, limiting the initial current flow and allowing the battery pack to safely charge the capacitive load. Once the voltage difference between nodes V2 and V1 has sufficiently decreased within an acceptable time frame, and without exceeding voltage or temperature limits, the BMS closes the positive contactor. This directly connects the positive terminal of the battery pack to the load's positive terminal, bypassing the pre-charge resistor. Finally, the pre-charge contactor is opened.

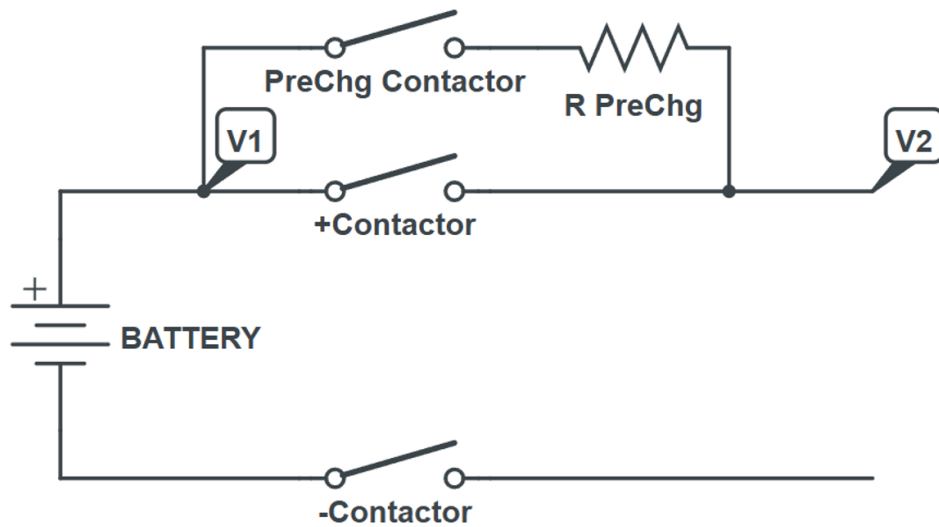


Figure 4.6: Pre-charge circuit

Chapter 5

Integration Phase

To successfully integrate the algorithms into the BMS it is crucial to have a clear understanding of the module structure and the interactions between the different layers of the system. A thorough analysis of the software architecture, particularly the application layer, is essential to identify and interpret the source code files that form the foundation of the system. Part of the integration process involves locating functions, structures, inputs and outputs, as well as understanding the role of the CAN bus in facilitating communication. Additionally, it is important to study how the operating system manages code execution and task scheduling. Careful planning and implementation of the source (.c) and header (.h) files are required to ensure proper initialization, execution, and abstraction of data. These files serve to connect algorithm inputs and outputs, coordinate their operation within the BMS, and ultimately transmit results over the CAN network for testing and validation, confirming that the algorithms have been seamlessly and effectively integrated into the system.

5.1 Module Structure

The module structure (Figure 5.1) serves various purposes in organizing and streamlining the software development process. The LLD/HAL (Low-Level Driver/Hardware Abstraction Layer) provides a standardized interface to the hardware, isolating higher-level software from hardware-specific details, enabling portability across different platforms, and handling direct interactions with hardware peripherals. The MCAL (Microcontroller Abstraction Layer) offers a standardized API (Application Programming Interface) to microcontroller peripherals, ensuring consistency across different microcontrollers and facilitating migration between families by abstracting peripheral controls. This includes drivers for standard peripherals like GPIO, ADC, PWM, and communication interfaces like CAN, SPI and UART.

The Service layer handles key functionalities such as CAN (Controller Area Network) for communication over the CAN bus, DIAG (Diagnostics) for implementing diagnostic protocols like UDS (Unified Diagnostic Service), and NvM (Non-volatile Memory) for managing long-term data storage in the non-volatile memory, ensuring persistence across power cycles. The Devices section provides drivers for complex devices that require specialized handling, such as sensors and actuators, and offers APIs that abstract these complexities. The Configuration section manages system and module configuration parameters, allowing customization and optimization for specific use cases. It typically includes tools for generating configuration files. Finally, the Application layer consists of Algo (Algorithms), which implement core logic and data processing, and FSM (Finite State Machine), which models and manages the application's operational states, ensuring predictable system behavior. This modular and organized structure contributes to a scalable and maintainable software architecture.

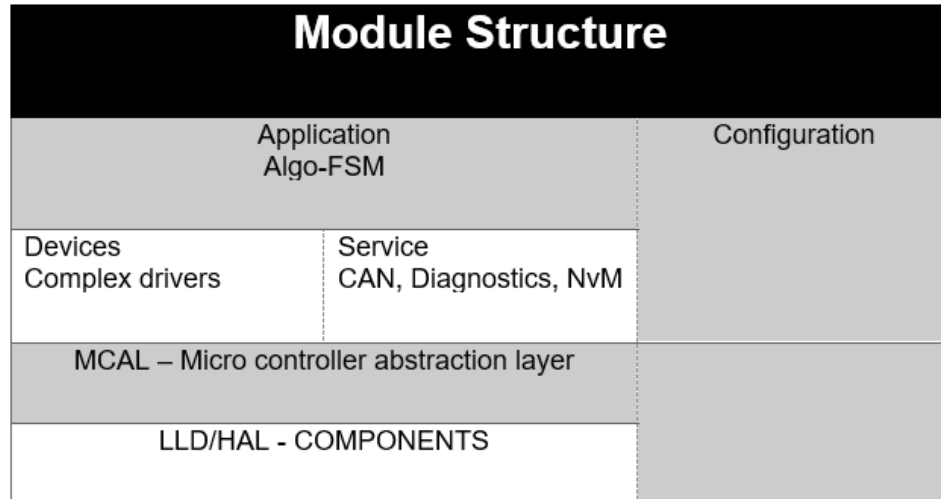


Figure 5.1: Module Structure

5.2 The code-base (BMS firmware)

The code-base is a collection of source code used to build the software system written in C language. Most of the code built is for the master BMS. It is made up of the repo/libraries that are mostly written by human programmers. It contains the ‘.c’ and ‘.h’ files which provide general access to the algorithm and not unrelated

to the architecture or the battery. The ‘cfg.c’ and ‘cfg.h’ files on the other hand are where the solution is adapted to the application.

To manage the code-base and the task scheduling the operating system FreeRTOS was used. FreeRTOS is the real-time operating system kernel used for embedded devices and is equipped with a scheduler. Tasks are created and are periodically called; each assigned a different priority. Tasks are assigned for the execution of every algorithm. Certain tasks are used to measure execution times, while other tasks are responsible for the CAN bus.

The code-base is managed using the development platform STM32CubeIDE with the help of the graphical tool STM32CubeMX that facilitates the configuration of STM32 microcontrollers and microprocessors. The BMS is then programmed using the STM32Cube programmer.

The goal of the integration phase is to carefully integrate the code generated for the model-based algorithms into the already existing code-base in a strategic manner such that the algorithms function in accord with the already existing BMS firmware allowing smooth flow of information in and out of the algorithms while simultaneously handling errors.

5.3 Integration within BeonD’s BMS

As part of the integration phase, several C-language source files were developed to embed the model-based algorithms into the existing BMS firmware. The handwritten .c and .h files were designed to contribute to the structure and functionality of the software system. To enable seamless integration of algorithms such as the Extended Kalman Filter and the Power Limits algorithm, a series of files were implemented. These files serve as interfaces between the algorithmic models and the broader BMS infrastructure, encapsulating initialization, parameter handling, fault detection, and CAN communication. By abstracting the logic of the algorithm and ensuring modular and maintainable code, these files played a critical role in bridging the gap between model-based development and embedded software implementation.

5.3.1 The Wrapper files

The wrapper files are part of the code-base and are handwritten in C. They serve as a crucial layer of abstraction and management within the embedded system. It is designed to encapsulate the core functionality of specific modules, such as the EKF for SOC estimation and the power limits algorithms. The wrapper file acts as an interface between the algorithm and the rest of the BMS system. It handles initialization, parameter updates, and fault detection, while also ensuring data persistence and providing debugging capabilities. It is also useful in providing

a structured interface for managing and controlling algorithms and integrating them with other system components and ensuring safe battery operation. In essence, these wrapper files are an essential part of integration as they enhance code modularity, improve maintainability, and simplify system integration by encapsulating complex functionalities and providing well-defined interfaces. The specific wrapper files within the code-base that were written for the purpose of integrating the model-based algorithms are discussed in detail in this section.

Wrapper file: BCM

The purpose of the wrapper file for the EKF, named BCM file, is to monitor and manage battery state parameters such as State of Charge (SOC), cell voltage, temperature, and pack capacity. Most importantly it includes the initialization function, which initializes the EKF model, as well as the main function. The main function operates as a state machine with phases for initialization, operational checks, normal operation, debug, and an idle state. The function evaluates battery parameters like voltage, current, and temperature, ensuring safe conditions. It updates SOC, adjusts system parameters, and detects faults such as over-voltage and under-temperature. In normal operation, it processes SOC updates and persists them in memory. In debug mode it supports testing. In addition, the file handles boundary checks for voltage, temperature, and SOC values. It uses real-time cell data to compute SOC via the step function and stores results in EEPROM, a type of non-volatile ROM, for persistent tracking. The debugging outputs are transmitted via CAN, to be visualized and monitored using the software tool CANalyzer.

The file starts by setting the Kalman Filter's temperature mode and calling the functions that initialize the SOC and the overall battery pack's capacity to prepare for the SOC estimation process. It then updates the battery's current measurement and based on the current flow direction (charging or discharging), the cell voltage and temperature are set accordingly. During discharging, the minimum cell voltage is used, while during charging, the maximum cell voltage is used. These values are obtained via the slave source code files. If the system is not in debug mode, normal operations are followed. However, if debug mode is enabled, a timeout is set, and when it expires, debug mode is deactivated. This allows for testing and diagnostics without interference with normal operations.

Next a state machine is used for system states. A function is set to use a state machine to manage different operational states of the system:

1. **The uninitialized state:** The system initializes SOC using data from EEPROM and checks the state of the battery's voltage, SOC, and other parameters to transition to a proper initialization state.

2. **The operational check state:** The system ensures all battery parameters are within valid ranges (e.g., voltage, temperature, SOC) and transitions to operational mode or debug mode if required.
3. **The operational state:** The Kalman filter is called by executing its step function to update the SOC estimate using the latest battery data. The SOC and OCV values are also stored in EEPROM.
4. **The debug state:** Similar checks are performed, but the system remains in a diagnostic mode for debugging purposes.

The Kalman step function is invoked in both operational and debug modes to perform the core Kalman filter algorithm, which refines the SOC estimate using current measurements and other data points. The SOC is updated and saved periodically based on the Kalman filter's output. The system ensures the SOC stays within defined limits, and if not, appropriate actions are taken (e.g., under or over-voltage conditions trigger safety or corrective mechanisms). Finally, the function responsible for CAN Communication sends debug information via the CAN bus, including state information, voltage errors, SOC, and other critical parameters for remote monitoring or diagnostics. The BCM file combines real-time battery data, a state machine for operational checks, and a Kalman filter to accurately estimate and manage the SOC of a battery pack, with the ability to communicate this data over a CAN bus for integration into a larger system.

Wrapper file: task_eng

The 'task_eng' wrapper file purpose is to manage periodic tasks, diagnostics, and CAN communication for an embedded system using FreeRTOS. It defines several periodic tasks (10 ms, 100 ms etc..) for operations like SOC calculation, diagnostics, insulation monitoring, and power limit management. The CAN-related tasks handle message transmission, reception, and error reporting across CAN1 and CAN2. The file also includes utilities for calculating task execution times, monitoring system performance, and dynamically enabling or disabling specific features such as diagnostics. It also facilitates runtime debugging by sending task metrics such as execution time and stack usage over CAN, ensuring robust real-time performance monitoring.

Wrapper file: PL10s_wrap

The wrapper file for the PL10s contains several functions and structures designed to manage and control the 10-second power limits algorithms for charging and discharging currents in a Battery Management System (BMS). It integrates various

libraries and headers for CAN communication, calibration, mathematical operations, and power limit management. The file initializes and runs the power limit model, processes inputs such as current, charge, discharge, and equivalent resistance. It outputs clamped charging and discharging limits to ensure they stay within predefined safety thresholds.

The main functions include the initialization function, input processing and execution of the step model as well as setting and clamping output limits for charging and discharging. The model calculates charging and discharging limits, clamping them based on predefined boundaries. Functions are used to manage publishing these limits in the structure used to store the available current value and sending relevant data over CAN communication.

Additionally, the file includes functions to set and retrieve values for the inputs and outputs, as well as utility functions for managing limits over a 10-second window, incorporating calibration constants. Overall, the wrapper provides a robust interface for managing and monitoring power limits while ensuring safe battery operation.

5.4 CAN bus

CAN bus (Controller Area Network communication bus) is a robust and reliable communication protocol that serves to manage the interconnection in vehicles between electronic components. CAN is equipped with differential signaling which provides noise immunity. Each node interconnected along the CAN bus has a unique identifier to determine the priority and content of the message. In addition, CAN makes use of bit-wise arbitration to determine the priority of the message. In this section, the role of CAN bus in the integration and testing phase is discussed in detail.

5.4.1 The development process of CAN

The development process of CAN proceeds through several key phases, each focused on distinct aspects of network design, hardware implementation, coding, and validation to ensure efficient and reliable communication across the network.

In the network design phase, the configuration of CAN nodes is established, including defining the total number of nodes, communication parameters, and CAN message specifications. CAN messages and signal definitions are typically specified using tools like Vector CANdb++, which centralizes message information in a database to maintain consistency across nodes and simplify network setup.

During the hardware design phase, decisions are made about the data-link layer and physical layer implementation. This includes setting the parameters for physical wiring, termination resistance, and bit timing. Choices made in this phase directly

affect how signals are interpreted and transmitted over the CAN bus.

The coding phase encompasses software-level implementation details for managing CAN transmission and reception. Transmission can follow either a message-based approach, where data is sent as entire messages, or a signal-based approach, which focuses on specific signals within messages. Reception management can be handled by either polling or interrupts. Polling involves the software periodically querying the CAN controller to check for incoming messages. In contrast, interrupt-driven reception allows the CAN controller to notify the ECU immediately when a message arrives, suspending other tasks if necessary. Additionally, reception can be configured as filtered or unfiltered, where filtered reception allows only specific messages to be received based on predefined criteria, thereby reducing processing load by discarding irrelevant messages.

Finally, the validation phase ensures the network's functionality through extensive testing. At the node level, each individual node's communication capability is verified. At the network level, tests confirm that all nodes interact seamlessly within the network. For validation, tools like Vector CANalyzer are commonly used for unit and system testing, allowing engineers to monitor and simulate communication on the CAN network.

The non-destructive CSMA/CA arbitration

In a CAN bus system, the non-destructive CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) arbitration or content-based arbitration is a method used to ensure that multiple nodes (devices) can communicate on the same bus without data loss or corruption due to collisions. This mechanism allows nodes to arbitrate for access to the bus without causing interference with each other. Before sending a message, a node will sense whether the CAN bus is free or busy. If the bus is free, the node will attempt to transmit its message. If the bus is busy, the node waits until the bus becomes available. Since all nodes have equal access to the bus, they can attempt to send messages once they detect that the bus is free. This is what allows multiple nodes to share the same communication medium. If multiple nodes attempt to send messages at the same time, CAN uses an arbitration process based on the message identifiers (IDs) to determine which node gets to transmit. The arbitration process is non-destructive, meaning that even if two or more nodes start transmitting simultaneously, only the node with the highest priority (the lowest message ID) will continue to transmit, and the others will back off. Each node transmits its message, and the CAN bus compares the bits of the message IDs bit by bit. As each node transmits, if it detects that a dominant bit (logical '0') has been transmitted by another node when it was trying to send a recessive bit (logical '1'), it will stop transmitting, allowing the node with the higher-priority message to continue sending.

CAN messages have a message identifier that specifies the content of the message and its priority. The lowest message identifier has the highest priority. The arbitration process is non-destructive because no data is lost during collisions. The node that loses the arbitration simply stops transmitting and waits for the bus to be free again. This ensures that no data corruption occurs during the arbitration process. This content-based arbitration makes CAN suitable for real-time and safety-critical systems, such as automotive applications, where certain messages must always take precedence over others.

CAN files development within the Code-base

Several files within the code-base are related to the development of CAN such as the CAN drivers and the CAN dbc files. The `can_com.c` file is part of the driver source file which implements functionalities for configuring, transmitting, and receiving messages via CAN interfaces. It supports two CAN controllers (CAN1 and CAN2) and includes queue management for handling message transmission and reception. Key functions include initializing CAN filters, triggering message transmission, and periodic processing for both received and transmitted signals. The file facilitates message composition, callback integration, and error handling while adhering to specific configurations like FIFO (First In, First Out) assignment and filter modes. It uses real-time OS queues for efficient message handling and ensures compatibility with the DBC (Database CAN) format. The remaining part of the file delves deeper into handling the transmission, reception, and manipulation of CAN messages. It includes functionality for constructing and sending CAN messages, processing transmit callbacks, and managing received messages. These functions employ message queues to buffer data, ensuring efficient communication despite limited immediate availability of CAN hardware resources. Additionally, utility functions for data integrity and signal processing, such as CRC (Cyclic Redundancy Check) computation and raw-to-physical signal conversion, enhance the robustness and accuracy of the CAN operations. Advanced bit-wise operations in various functions facilitate precise bit-level manipulations, catering to specific protocol requirements. The code also includes safeguards such as periodic error reporting and limits validation, ensuring system reliability.

The `can_dbc.c` file on the other hand, is responsible for defining several external variables and functions related to CAN message and signal handling. It plays a critical role in managing CAN message and signal interactions. It defines and exposes key data structures, including arrays for messages and signals associated with the two distinct CAN channels CAN1 and CAN2, segregated by their transmission (Tx) and reception (Rx) directions. The file provides utility functions to retrieve specific message or signal properties, such as IDs, DLCs (Data Length Codes), and

retransmission timings. Additionally, it includes mechanisms to invoke callback functions associated with signals, ensuring customizable and context-specific handling of CAN data. This modular implementation facilitates the flexible handling and verification of both CAN messages and signals across multiple channels.

It abstracts away the underlying hardware details and provides a higher-level API for developers to interact with the CAN bus. By using this file, developers can easily configure and manage CAN communication without needing to delve into the specifics of the hardware registers and protocols.

5.4.2 CANalyzer

CANalyzer is a software tool used to analyze the data traffic in serial bus systems. CANalyzer is used for the analysis, testing, simulation and diagnostic of data transmitted through the CAN bus. It provides features such as Trace and Graphics to visualize signals as well as logging of bus data and replay for offline analysis. An important feature known as the interactive generator (IG) can be used to send periodic messages such as an input current profile, simulating a load. This was essential in the testing and validation phase of the integrated algorithms.

Important data in the code-base are sent as messages through CAN bus. Each CAN channel can transmit a maximum of 8 bytes; therefore, large messages must be split and sent over more than one channel. Messages are sent to a queue where they are then transmitted based on priority. With the help of the features included in CANalyzer, the data can be visualized, and various tests can be done to validate the models and evaluate performance. CANalyzer requires a license and the VECTOR network interface to access CAN. The network interface hardware is shown in Figure 5.2. As explained previously, the CAN DBC file is a data description file or database that contains information for handling identification and translation of CAN messages and raw CAN data to physical values. The CANdb++ program acts as a user interface that allows the visualization of the database and facilitates the process of adding and modifying CAN messages. Figure 5.3 shows some of the signals added to the Vector CANdb++ editor, related to the integrated algorithms. The file can then be used by CANalyzer to graphically visualize the signals.

5.5 Execution time measurement methods

Measuring execution time of the algorithms implemented on the BMS is essential not only to confirm the capability of the BMS in handling such algorithms, but also for determining the influence of the implemented code optimizations on the execution time. In comparison to the PIL test results obtained using MATLAB, the execution time of the algorithms after integrating them into the BMS is expected



Figure 5.2: CAN interface hardware

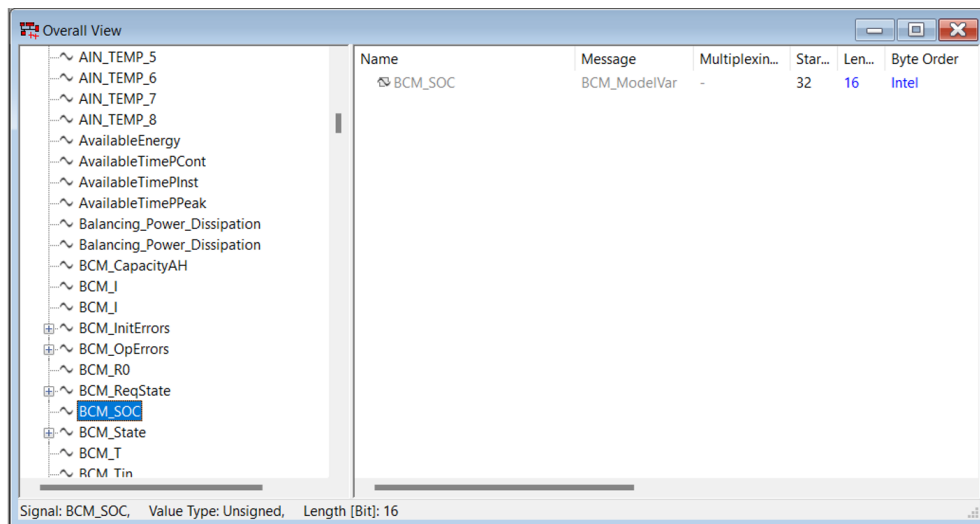


Figure 5.3: Vector CANdb++ editor

to be higher. This is because the PIL test does not take into account the wrapper file that integrates the algorithm to the rest of the code-base. In general, there are two methods for measuring the execution time of an algorithm after it has been integrated into the code-base. The first method is through the microcontroller's GPIO pin, and the second method is using the FreeRTOS runtime counter.

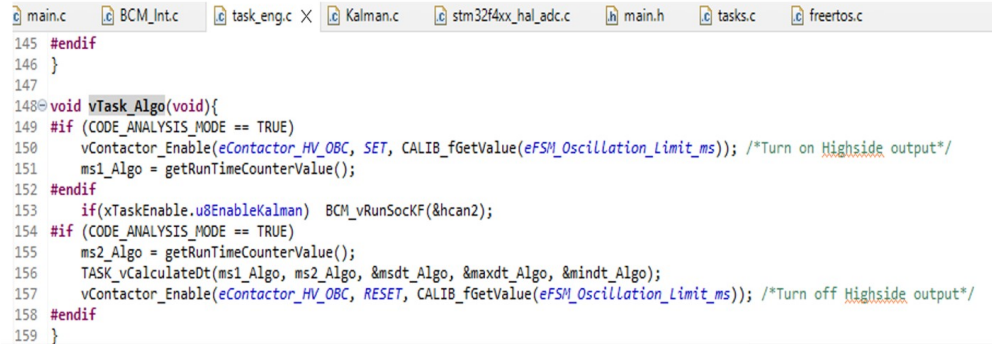
GPIO is a general-purpose input/output digital pin located on the STM32 Nucleo board. This pin can be connected to the oscilloscope to graphically display the varying voltage as a function of time resulting from the execution of the algorithm. Once the BMS is flashed, the rising and dropping of the voltage creates a pulse that can be visualized on the oscilloscope's display. Using the cursors, the pulse width can be measured thus indicating the execution time of the algorithm.

On the other hand, FreeRTOS can collect information on the current time. Using the run-time counter macro, the current time can be returned before and after the execution of the algorithm; therefore, calculating the completion time. To obtain the execution time of the algorithm, the interrupts are disabled and the difference in the times returned by the runtime counters is calculated. The function that was used to obtain the value of the run-time counter is named `getRunTimeCounterValue`. Figure 5.4 shows how the function is used to measure the execution time of the EKF algorithm. In line 151 the current time is returned, the algorithm is executed in line 153 and the current time is returned again in line 155. The two values obtained are then subtracted to calculate the time elapsed during the execution of the algorithm.

Both methods, GPIO-based and FreeRTOS-based, offer ways to measure the execution time of an algorithm on an STM32 microcontroller. However, they have different advantages and disadvantages. GPIO-based measurement provides a direct, hardware-based measurement of the algorithm's execution time and the measurement process itself does not introduce significant software overhead. However, the accuracy of the measurement depends on the oscilloscope's resolution and the precision of the trigger mechanism. In addition, external factors such as noise or electromagnetic interference can potentially affect the measurement. The FreeRTOS-based measurement allows for more precise measurements and complex analysis within the software environment. On the downside, the measurement process involves function calls and timer operations, which can introduce some overhead, especially for short execution times. Disabling interrupts during the measurement can affect the overall system behavior, especially in real-time systems and therefore the run-time counter cannot be used in this case.

Overall, both the GPIO-based and FreeRTOS-based methods were employed to measure the execution time of the algorithms. While both approaches yielded comparable results, the FreeRTOS-based method, specifically utilizing the `getRunTimeCounterValue` function, was primarily used to measure the execution time of algorithms as it was the faster method, since no hardware setup was required and it was sufficiently precise since the overhead was insignificant compared to the total execution time. The GPIO-based method, involving the oscilloscope, was

employed to verify the results, particularly when interrupts were disabled while testing prior to the integration phase.



```

145 #endif
146 }
147
148 void vTask_Algo(void){
149     #if (CODE_ANALYSIS_MODE == TRUE)
150         vContactor_Enable(eContactor_HV_OBC, SET, CALIB_fGetValue(eFSM_Oscillation_Limit_ms)); /*Turn on Highside output*/
151         ms1_Algo = getRunTimeCounterValue();
152     #endif
153     if(xTaskEnable.u8EnableKalman) BCM_vRunSocKF(&hcan2);
154     #if (CODE_ANALYSIS_MODE == TRUE)
155         ms2_Algo = getRunTimeCounterValue();
156         TASK_vCalculateDt(ms1_Algo, ms2_Algo, &msdt_Algo, &maxdt_Algo, &mindt_Algo);
157         vContactor_Enable(eContactor_HV_OBC, RESET, CALIB_fGetValue(eFSM_Oscillation_Limit_ms)); /*Turn off Highside output*/
158     #endif
159 }

```

Figure 5.4: Runtime Counter function used to measure execution time of EKF

Chapter 6

Testing and Validation

6.1 Testing of algorithms

Several tests were conducted to confirm the validity of the EKF and are discussed in detail in this section. First, to test the behavior of the EKF and its ability to correctly estimate the SOC, it was compared against the simpler CC (Coulomb Counter) algorithm. In ideal simulation conditions where sensor noise is ignored and under optimal battery health and cell capacity, the deviation between the EKF and CC should be insignificant, making it a good reference for validation of the EKF.

Another test that was carried out involves comparing the performance of the EKF post-optimization with the original baseline algorithm. This test is useful in confirming that the optimization process did not alter the underlying logic of the algorithm.

Additionally, SIL and PIL test simulations were performed using the SIL/PIL manager on MATLAB before the integration of the algorithms. These tests are vital in validating that the code generated from the models is equivalent to the Simulink model. Moreover, it is essential for providing insight into execution times and stack usage and for testing how the different optimizations applied affect the models and the generated code.

Finally, the algorithms must be tested post-integration to ensure that they are integrated correctly into the BMS and collectively work in harmony with the rest of the elements in the code-base. This can be tested with the help of the analysis software tool CANalyzer while connecting the CAN bus for communication and flashing the BMS implemented on the PCB. After performing rigorous testing and validation we can confidently deploy the firmware connected to a real battery pack and observe the behavior in conditions similar to real world conditions with the help of the equipment in Beond's laboratory BALF.

6.1.1 EKF vs. Coulomb Counter

A quick way to confirm the ability of the EKF to estimate the SOC correctly is to compare it to the much simpler model of the Coulomb counter. A parameter named SOC_error, was determined by calculating the difference between the EKF-estimated SOC value and the SOC obtained through Coulomb counting. A simulation carried out in Simulink, utilizing the Data Inspector tool for data logging and signal display, was conducted to evaluate the behavior testing under a 10A discharging constant current, then a 10A charging constant current and finally a random charging/discharging current.

Testing under a 10A discharge current, as shown in Figure 6.1, the difference between the SOC estimation of the EKF and the CC remained minimal, recording values below 0.0005, up until the SOC reached 30%. Below this SOC value, the error increased sharply reaching a value of 0.0022 at 5% SOC. Further discharge led to a more rapid divergence, culminating in a maximum error of 0.0073 at SOC values below 1%. This analysis highlights the potential for small estimation discrepancies at low SOC levels between EKF and CC.

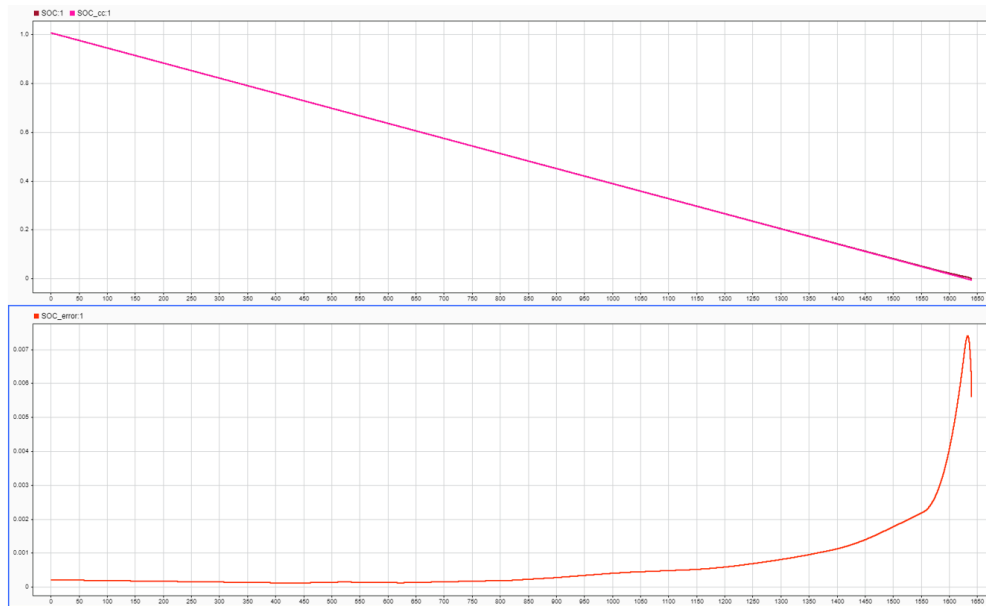


Figure 6.1: EKF vs. CC SOC estimation in Simulink, DCHG at 10A

The second test was conducted under a 10A charging current as illustrated in Figure 6.2. The difference between EKF and CC estimations remained below

0.000066 up to 70% SOC, after which it began to gradually increase. Beyond the 95% SOC mark, the difference rose sharply, reaching a maximum of 0.000245 at 100% SOC. Overall, the observed discrepancies between EKF and CC estimations were slightly lower during charging compared to discharging.

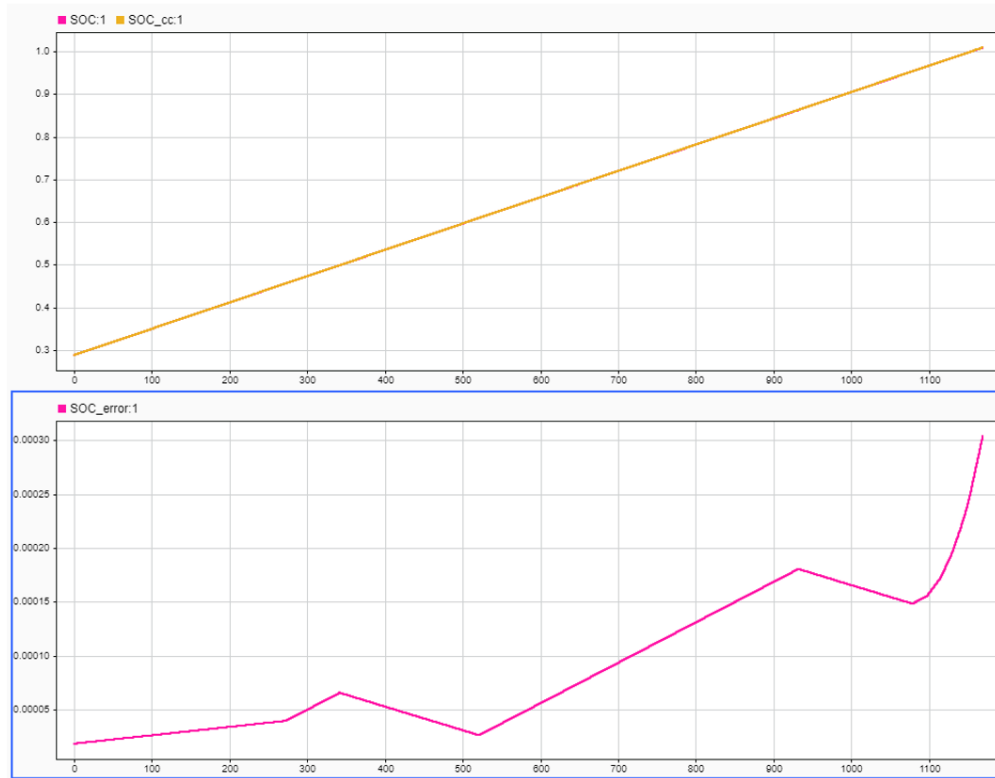


Figure 6.2: EKF vs. CC SOC estimation in Simulink, CHG at 10A

A random current profile, fluctuating between 60A (charging) and -20A (discharging), was used to test the EKF and CC Simulink models. Figure 6.3 shows how the estimation error remained below 0.0005 for SOC values below 30%. Minor peaks in the error were observed, coinciding with instances of charging current. A steep increase in error was observed below 25% SOC, arriving at a value of 0.003 for SOC levels below 1%.

In a nutshell, the two algorithms demonstrated comparable results with minimal deviation, which confirms the validity and correctness of the EKF algorithm in estimating the SOC. It is important to note that the deviation between the two algorithms would be much more significant under real-world testing conditions

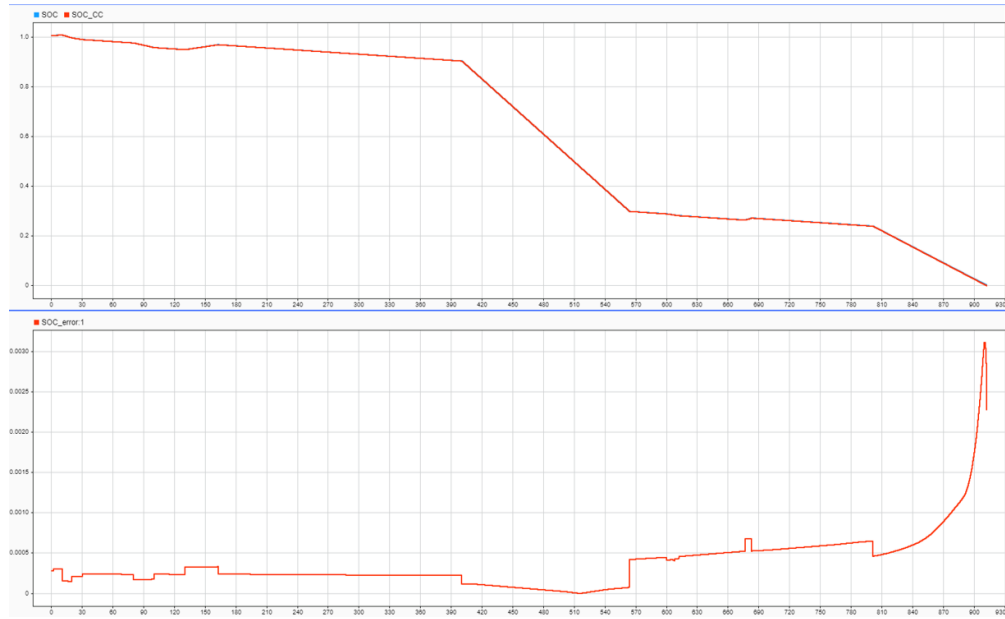


Figure 6.3: EKF vs. CC SOC estimation in Simulink, CHG and DCHG random current profile

where the tests are carried out for much longer periods and where other external factors are more likely to interfere with the results. In which case, the EKF proves to be much more powerful in estimating the SOC, displaying precise results comparable to the experimental SOC-OCV curves. In this simulation, the cell capacity remained constant and other conditions were ideal for the sole purpose of confirming the ability of the EKF to correctly estimate the SOC under different current profiles.

6.1.2 SIL/PIL testing

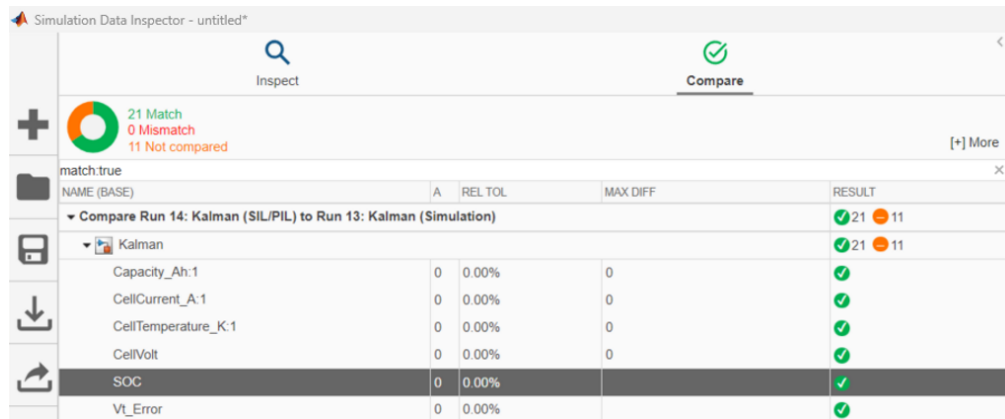
Two critical tests named SIL and PIL were carried out before the integration phase. A software-in-the-loop (SIL) simulation compiles and runs the generated code on the personal computer used to run the simulation. A processor-in-the-loop (PIL) simulation cross-compiles source code on the personal computer, and then downloads and runs the object code on the target processor. Both SIL and PIL simulations can be useful in testing whether the model and generated code are numerically equivalent and allow us to observe the code coverage. In addition, using PIL combined with the target hardware, code execution profiling can be performed. It is used to determine the size of stack memory and execution time that is required to run generated code on the target hardware. The PIL test can be executed to generate a stack usage profile and an execution time profile. The

profiles generated enable us to observe the effect of code optimizations and data input on stack usage, as well as execution time.

EKF SIL/PIL validation of generated code

MATLAB makes use of the SIL/PIL manager to carry out SIL and PIL tests using the code generated from the embedded algorithm. The SIL test compiles and runs the generated code on the PC and displays the results on MATLAB's data inspector. The data inspector is used to visualize simulation data such as signals, inputs and outputs and compare them. SIL is a faster and simpler way to verify the source code when compared to PIL since it does not require a target hardware as it simply runs the algorithm on the PC. Therefore, the SIL test was done first for verification of the generated code before connecting the target hardware and running PIL.

The data inspector logs data both from the normal Simulink simulation and the SIL simulation and compares them. A constant current input of 50A was fed into the model and the SIL verification was carried out. Figure 6.4 shows the quantitative data comparison between the simulation and SIL again showing a perfect match with a tolerance of zero as shown in the data inspector. Furthermore, Figure 6.5 shows a perfect match graphically between the SOC estimated in the simulation and the SOC estimated in SIL, obtained from the EKF. Therefore, confirming the validity of the source code in matching the logic of the model-based simulation. The test was repeated using various other current profiles, further confirming the conclusion.



NAME (BASE)	A	REL TOL	MAX DIFF	RESULT
Compare Run 14: Kalman (SIL/PIL) to Run 13: Kalman (Simulation)				
Kalman				21 Match, 0 Mismatch, 11 Not compared
Capacity_Ah:1	0	0.00%	0	✓
CellCurrent_A:1	0	0.00%	0	✓
CellTemperature_K:1	0	0.00%	0	✓
CellVolt	0	0.00%	0	✓
SOC	0	0.00%	0	✓
Vt_Error	0	0.00%	0	✓

Figure 6.4: Quantitative data comparison between the EKF simulation and SIL, 50A DCHG current

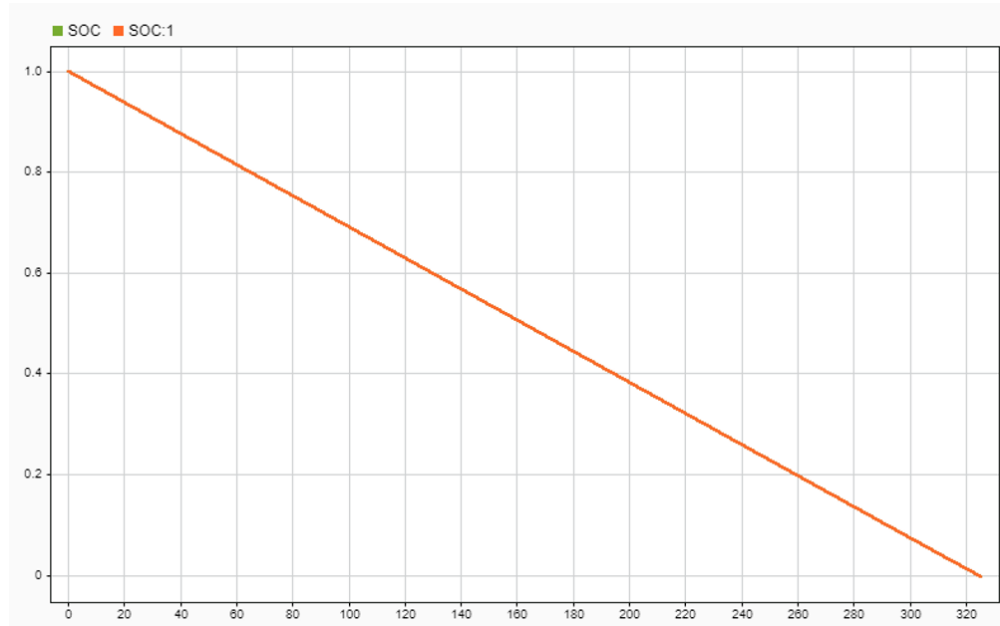


Figure 6.5: Graphical data comparison between the EKF simulation and SIL, 50A DCHG current

EKF SIL/PIL Execution Time and Stack Profiles

The SIL/PIL manager also provides a summary of the profiling data generating the Code Execution Profiling Report. It includes the task execution times of the entry-point functions of the algorithm such as the ‘initialize’ function, the ‘step’ function and the ‘terminate’ function. The execution time displayed is the time elapsed between calling the entry-point function and terminating it and so it includes the time spent calling other functions within the entry-point function.

The useful execution times report would be the one generated by the PIL simulation as it runs the code on the target processor ARM Cortex-M4, which is the processor of the STM32 Nucleo-F429ZI board. Therefore, it acts as a preliminary step between the model-based design phase and the integration phase for comparing the baseline code and the optimized code.

<i>PIL</i>	Average Execution Time [μs]		
Section Name	Baseline	Optimized	Difference
EKF_initialize	8	3.7	54%
EKF_step [0.1 0]	103.1	53.5	48%
EKF_terminate	0.9	0.39	56%

Table 6.1: Average execution times of EKF functions obtained through PIL test

Table 6.1 presents a comparison between average execution times of the key entry-point functions of the optimized and baseline EKF algorithms. The results demonstrate a significant performance improvement, with the optimized version achieving approximately a 50% reduction in execution time compared to the baseline.

6.1.3 Completion time results

In the previous section, the algorithms were tested independently by performing the PIL test and measuring the execution times confirming that the optimizations implemented have significantly reduced the execution times of the standalone algorithms. However, it is essential to test the algorithms after integrating them into the BMS while they execute along side multiple elements present in the BMS and understand their behavior as integrated algorithms rather than standalone algorithms. To measure the algorithms' completion times, a GPIO pin was toggled and observed using an oscilloscope. This method provided accurate results by measuring the pulse width generated during execution. The same technique was used to evaluate various algorithms, such as the EKF and the Power Limits (1 s, 10 s, and 30 s). The test was performed with interrupts enabled and then repeated after disabling them. Additionally, this method was used to assess the impact of EKF code optimization, by performing the test once with the integrated baseline EKF algorithm and once with the integrated optimized EKF algorithms and measuring the completion times.

Oscilloscope and GPIO pin for execution time measurements

As explained previously in Section 5.5, the GPIO pin was used to directly measure the algorithm's completion time with the help of the oscilloscope. Figure 6.6 demonstrates the results displayed on the oscilloscope screen when the firmware is flashed on the BMS. The cursors were used to measure the pulse width indicating the algorithm's completion time. In this example, the algorithm's completion time measures around 51.76 microseconds as seen on the display screen.

The same method using the GPIO pin was repeated to measure the completion time of different algorithms including the EKF, The 1 s, 10 s, and 30 s Power Limits as shown in Table 6.2. The test was repeated once for when the interrupts were enabled and again for when the interrupts were disabled. When interrupts are disabled, the algorithm is executed by the processor without interruption until it is completed. Disabling the interrupts; therefore, reduces the time measured by eliminating the overhead caused by context switching to handle ISRs (interrupt service routines). As a result, the algorithm runs without interference, so its execution time is deterministic. The interrupts are disabled only for testing purposes,

such that the execution time measurements reflect the ‘pure’ execution time of the function rather than the ‘completion time’, therefore excluding the influence of external interruptions. These results are important because they will be compared later with the optimized versions.



Figure 6.6: Oscilloscope displays the pulse width indicating the algorithm’s execution time

<i>Oscilloscope measurements</i>	Completion Time Post-Integration [μs]	
Algorithm (Baseline)	Interrupts Enabled	Interrupts Disabled
Kalman	128.32	102.3
EW1sPwrLim	3.81	2.08
EW10sPwrLim	466	168
EW30sPwrLim	455	174

Table 6.2: Completion times of various algorithms post-integration measured using the oscilloscope

Furthermore, the GPIO method was used to evaluate the EKF code optimization. As seen in Table 6.3, the completion time of the algorithm is significantly reduced post-optimization. The optimization mentioned in Section 3.2 related to the signal storage reuse may have had the most significant influence on the reduction in execution time. Due to this optimization, the CPU may save and restore certain parts of memory and registers during context switches. If memory usage is optimized, less

data needs to be saved and restored. In addition, sharing memory across signals means fewer memory accesses, which reduces contention between the EKF function and other tasks. These combined effects lead to a significant reduction (23.7%) in execution time when interrupts are enabled. It can also be observed that the pure execution time of the algorithm, measured when the interrupts are disabled, has been reduced by 49.4%. These results are consistent with the results obtained by the PIL test and confirm the positive influence that the implemented optimizations have on the execution efficiency of the EKF.

<i>Oscilloscope measurements</i>	EKF Completion Time Post-Integration [μs]		
	EKF Baseline	EKF Optimized	Difference
<i>Interrupts Enabled</i>	128.32	97.96	23.7%
<i>Interrupts Disabled</i>	102.3	51.76	49.4%

Table 6.3: Effect of optimization on completion times of EKF post-integration measured using the oscilloscope

6.1.4 Testing and validation of Power Limits

The power limits algorithm was rigorously tested and validated using a combination of simulation and hardware-in-the-loop techniques. Similar to the EKF, the SIL/PIL testing in MATLAB Simulink was employed to assess the algorithm's functional correctness and execution time. Stack profiles were analyzed to identify potential memory bottlenecks. Post-integration, the algorithm's execution time was measured using the FreeRTOS runtime counter, providing insights into its real-world performance.

In this case, the purpose of the test is to evaluate the efficiency of the code generation process. To do so, the MATLAB-generated C code was compared with a hand-written C implementation to measure both the execution time and the performance of the algorithm. Both versions were subjected to identical test cases, including random positive and negative current profiles, to assess their accuracy and speed. Comprehensive testing was conducted to verify the algorithm's adherence to various constraints. Pack temperature and temperature limits were monitored to ensure safe operation. Pack voltage and voltage limits were checked to prevent over-voltage and under-voltage conditions.

Data inspector testing of Power Limits Simulink model

As explained previously, the power limits algorithm obtains its inputs from the EKF, while the EKF requires measurements from the battery as inputs. Therefore, the BCM Simulink model was included in this segment of testing to simulate a battery cell providing the necessary inputs for the EKF. As shown in Figure 6.7 below, a random DCHG current (pink) ranging between 0-90 Amps, was imposed to test the Power Limits Simulink model. The output current limits for the instantaneous PL1s (red), PL10s (green) and PL30s (blue) algorithms were plotted over time, measured in seconds. The PL10s algorithm was also propagated through a sample-and-hold block (purple) and was plotted on the same graph. The PI parameters were carefully tuned to obtain a controller that responds fast, but with minimum oscillations. Since the DCHG current in this test is not aggressive, the three algorithms produce almost identical results with a small deviation from each other. In the beginning of the discharge cycle the SOC is high; therefore, the current limits are well above the requested current. At 265 seconds, the graphs intersect and at this point in time the full requested current cannot flow anymore and is limited by the output set by the Power Limits algorithm. The value of the current limit continues to decrease as the battery discharges and the SOC value drops.

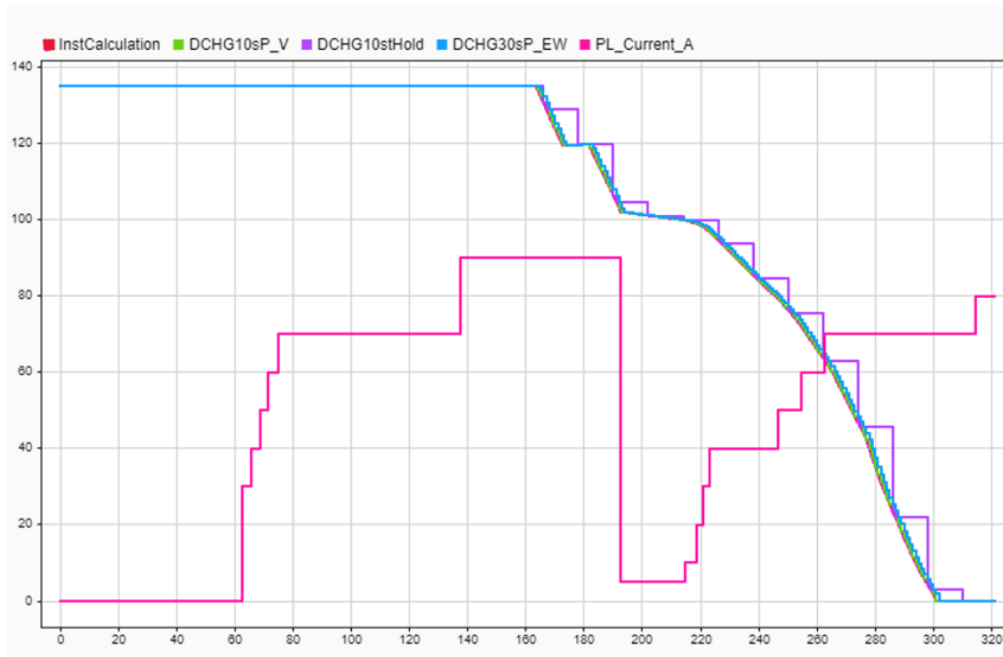


Figure 6.7: Simulink's Data inspector for Power Limits under a random current profile

Further testing was performed under more aggressive current profiles followed by tuning of the PI controller to obtain and asses different conservativeness in the estimation of the power limits and compare the PL10s algorithm with the PL1s algorithm. Figure 6.8 shows the power limits over time (in seconds) obtained by testing under a constant DCHG 90 A current. The deviation between the PL10s algorithms (orange, green, blue) and the PL1s (pink) is more significant under an aggressive DCHG current. The plot demonstrates how the PL10s is more conservative than the PL1s as all the plots lie below the PL1s plot indicating lower current limits.

It is important to note that the three outputs related to the PL10s algorithm present different behaviors due to the different PI parameters assigned to them. This demonstrates the effect of tuning the PI parameters on the behavior of the algorithms and the possibility of customizing the behavior to match the desired output.

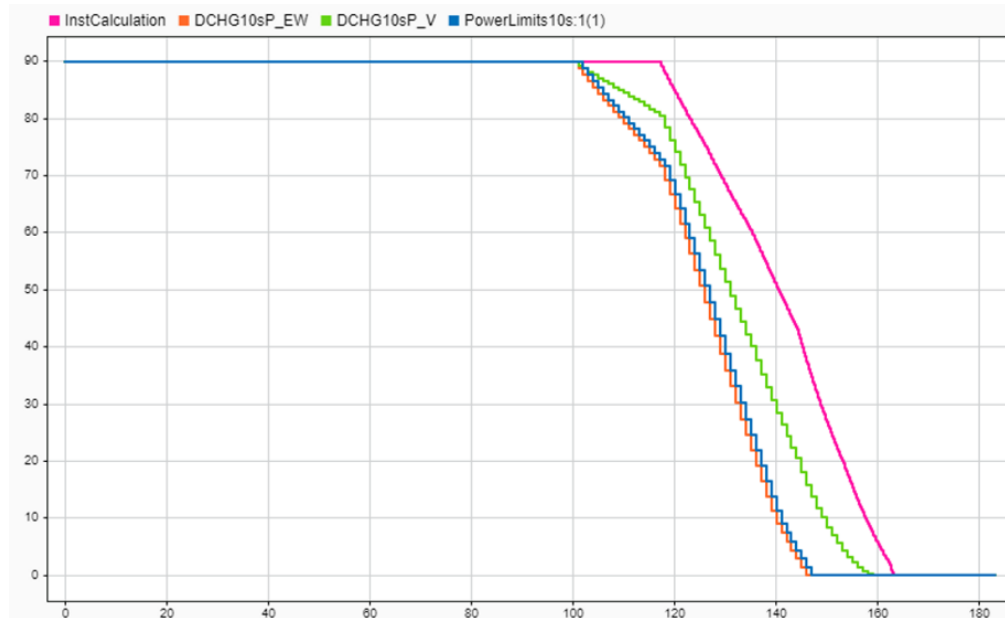


Figure 6.8: Simulink's Data inspector for Power Limits under a 90 A constant current profile

Next, further tests were performed by modifying the integral gain of the PI controller. As mentioned in sub-section 2.4.1, increasing the value of the integral gain of the PI controller, eliminates the steady-state error faster; however, it leads to increased oscillations and larger overshoot. This is depicted in Figure 6.9 and Figure 6.10 showing the effect of imposing higher and lower integral gains. The current

limit in Amps is plotted over time in seconds. It can be observed that by setting a low integral gain, the PL10s and PL30s algorithms experience a lower overshoot, with the PL30s being more conservative. The PL10s shows a less conservative behavior almost following the PL1s algorithm during the discharge cycle. A higher integral gain on the other hand, increases the overshoot causing an exaggerated conservative behavior, especially in the PL30s. In conclusion, careful tuning of the PI parameters is essential in obtaining a balanced behavior that preserves both the health of the battery while simultaneously meeting the performance demands and respecting the safety limits.

To delve more into the possible behaviors that can be obtained by tuning the PI parameters, further testing was done and is discussed in detail in sub-section 6.2.2.

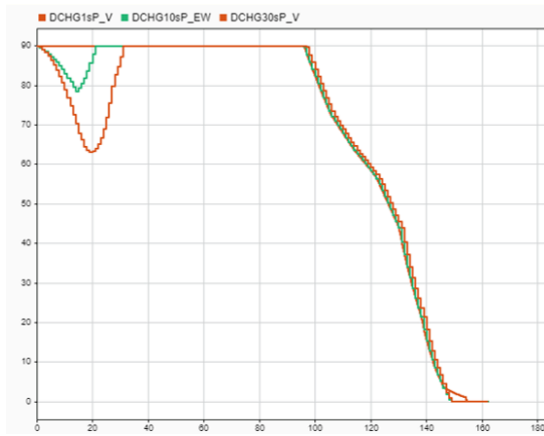


Figure 6.9: Low integral gain

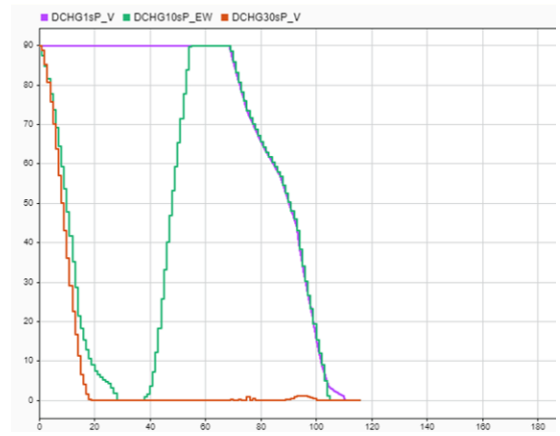


Figure 6.10: High Integral Gain

SIL testing and validation of power limits

As performed previously on the EKF, the SIL test was also performed to validate the generated code for the Power Limits and ensure that it behaves identically to the Simulink model. The Power Limits algorithm was fed with various input currents to perform tests under both DCHG and CHG conditions. The main input and output signals were logged including six different signals and compared.

Figure 6.11 below shows an example of a DCHG test carried out on the PL10s demonstrating a perfect match between all six signals, with a zero deviation from the simulated model. The upper graph plots one of the signal pairs, the DCHG power limits obtained through the PL10s Simulink model and the DCHG power limits obtained through the generated code. The lower graph indicates that the

difference between the results is equal to zero. It is important to note that the SIL simulation test does not compare intermediate signals logged inside the model, it compares only external input and output signals, which was sufficient to validate the generated code. Once the generated code has been validated, we can proceed with the integration process and continue with further testing and validation.

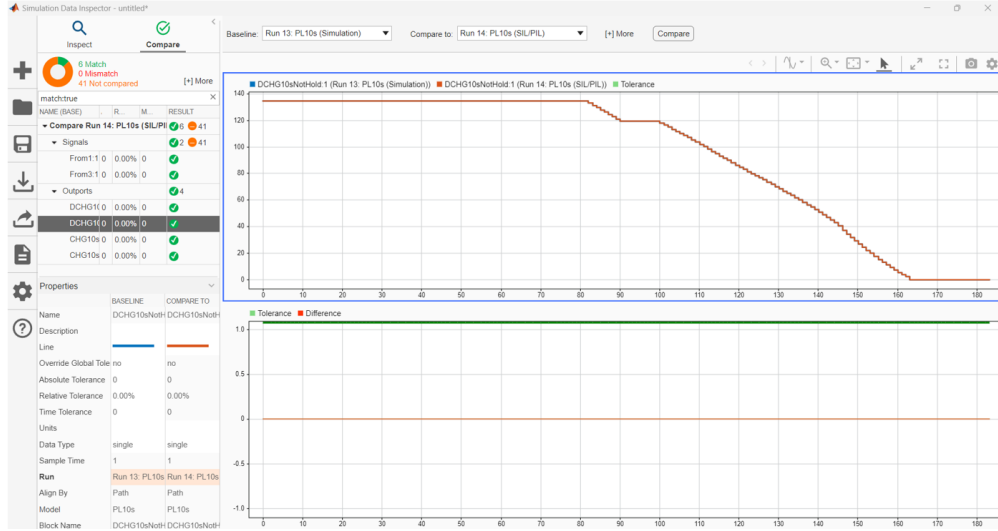


Figure 6.11: SIL simulation test of PL10s under a DCHG current

6.2 Testing and validation post-integration

6.2.1 Power limits execution times

The power limits algorithm is integrated into the BMS. By utilizing the FreeRTOS RunTime Counter, the completion time of the algorithm was measured while the interrupts are enabled. A comparison was then conducted between the completion time of the optimized and the non-optimized version of the same algorithm. In other words, the optimized code generated by MATLAB's Embedded Coder and the equivalent hand-written C implementation.

Table 1.1 presents these results and clearly demonstrates a significant performance advantage for the code generated by Embedded Coder. The Embedded Coder version exhibited a remarkable 77.68% faster execution speed compared to the hand-written implementation, highlighting the substantial impact of Embedded Coder on code optimization and performance. This significant performance gain underscores the value of leveraging the code generation capabilities of MATLAB for the development of efficient and optimized embedded systems.

Optimization of 10s Power Limits Algorithm	
Algorithm	Completion Time (μ s)
MATLAB Generated Code	104
Handwritten Code	466
DIFFERENCE	77.68%

Table 6.4: PL10s execution time optimization results post-integration measured using RunTime Counter

6.2.2 Testing and validation using CANalyzer

After the integration process was completed, multiple simulations were conducted with the help of CANalyzer, not only to confirm that the Power Limits algorithm is functioning correctly, but also to evaluate and tune the PI controller's behavior. Multiple tests were performed using different sets of PI parameters and the results were recorded and assessed.

The performance of the 10-second power limits algorithm (PL10s), generated using Embedded Coder, was compared against a simple 1-second power limits algorithm (PL1s) previously validated and implemented in hand-written C code. The tuning of the PI parameters was done directly within the code-base by modifying the source code of the integrated algorithm. Testing post-integration was done by flashing the firmware onto the micro-controller implemented in the BMS. The BMS was also connected to a CAN bus to visualize the data and perform testing and simulations using CANalyzer.

Test under random DCHG current

A random discharging current, ranging between 0 to 90 amps, was used to bring the SOC from 100% to 10%. Figure 6.12 top plot depicts the input discharging current (CurrentProfile), along with the output DCHG current limits PL1sDCHG and PL10sDCHGop, obtained by the PL1s algorithm and the optimized/Embedded Coder generated PL10s algorithm respectively. The bottom plot is the corresponding SOC obtained as an output of the optimized EKF algorithm under the same random current profile.

The DCHG PL10s algorithm using the first trial set of PI parameters, exhibited an aggressive but conservative behavior, responding more rapidly to changes in the input current profile compared to the DCHG PL1s, while maintaining lower magnitudes of current throughout the discharge cycle. It is observed that the general trend of the current limit decreases as the SOC decreases. At high DCHG input current, the PL10s algorithm preserves the battery by reducing the current limits and consequently the power limits. As intended, it maintained more conservative

power limits in comparison to PL1s resulting in lower overall current. This confirms that none of the battery limits are exceeded. However, the algorithm's behavior can be improved by reducing the overshoots. The PI controller's behavior can be adjusted to meet specific design requirements through appropriate tuning. This will be demonstrated and discussed in detail in this section.

Test under random CHG current

The two algorithms were again tested under a CHG current ranging between 0 to 30 Amps, taking the SOC from 70% to 100%. This test is done at higher SOC values since this is the critical range where the CHG current is de-rated by the algorithm. A first trial set of PI parameters was used for the PL10s PI controller. Figure 6.13 depicts the magnitude of the CHG current profile (AbsCurrentProfile) in the top plot, along with the CHG current limits for the PL1s (PL1sCHG) and the PL10s (PL10sDCHG). The bottom plot depicts the corresponding SOC. At lower SOC, the current limits maxed out at 27 A, which is the maximum value allowed for the current to reach as decided by the algorithm. Again as intended, the PL10s showed a more conservative approach than the PL1s. It produced a similar behavior to the PL1s at a lower SOC, however; above a SOC of 70, it maintained a lower CHG current. Both algorithms show a decreasing trend as a response to the constantly increasing SOC. The behavior of the CHG PI controller implemented in the PL10s can be tuned by varying the PI parameters to obtain the desired behavior.

Tuning the PI parameters and testing the Power Limits algorithms

It is important to test the algorithms by applying different PI parameters, comparing the behavior of the PI controller in the PL10s algorithms to the original PL1s, as well as comparing the behavior of the baseline algorithm to the optimized version and assess the differences.

To obtain a less aggressive behavior for the DCHG optimized PL10s (PL10sDCHGop), the proportional gain K_p was reduced to 0.01. The top plot in Figure 6.14 shows the results under a constant discharging current of 70 A (CurrentProfile) for both the optimized PL10s and the two non-optimized PL10s (PL10sDCHG) and PL1s (PL1sDCHG) algorithms handwritten in C. The bottom plot shows the corresponding SOC falling from around 70% to 0%. The reduction in the value of K_p leads to a less aggressive behavior in the PL10s algorithms as it results in a slower response time. Additionally, the two algorithms take longer to react to the initial current change when compared to PL1sDCHG, since reducing the value of K_p increases the rise time. It is also worth noting the slight deviation in the behavior of the PI controller in the PL10s handwritten in C PL10sDCHG in comparison to the optimized Embedded Coder generated algorithm PL10sDCHGop, since the logic

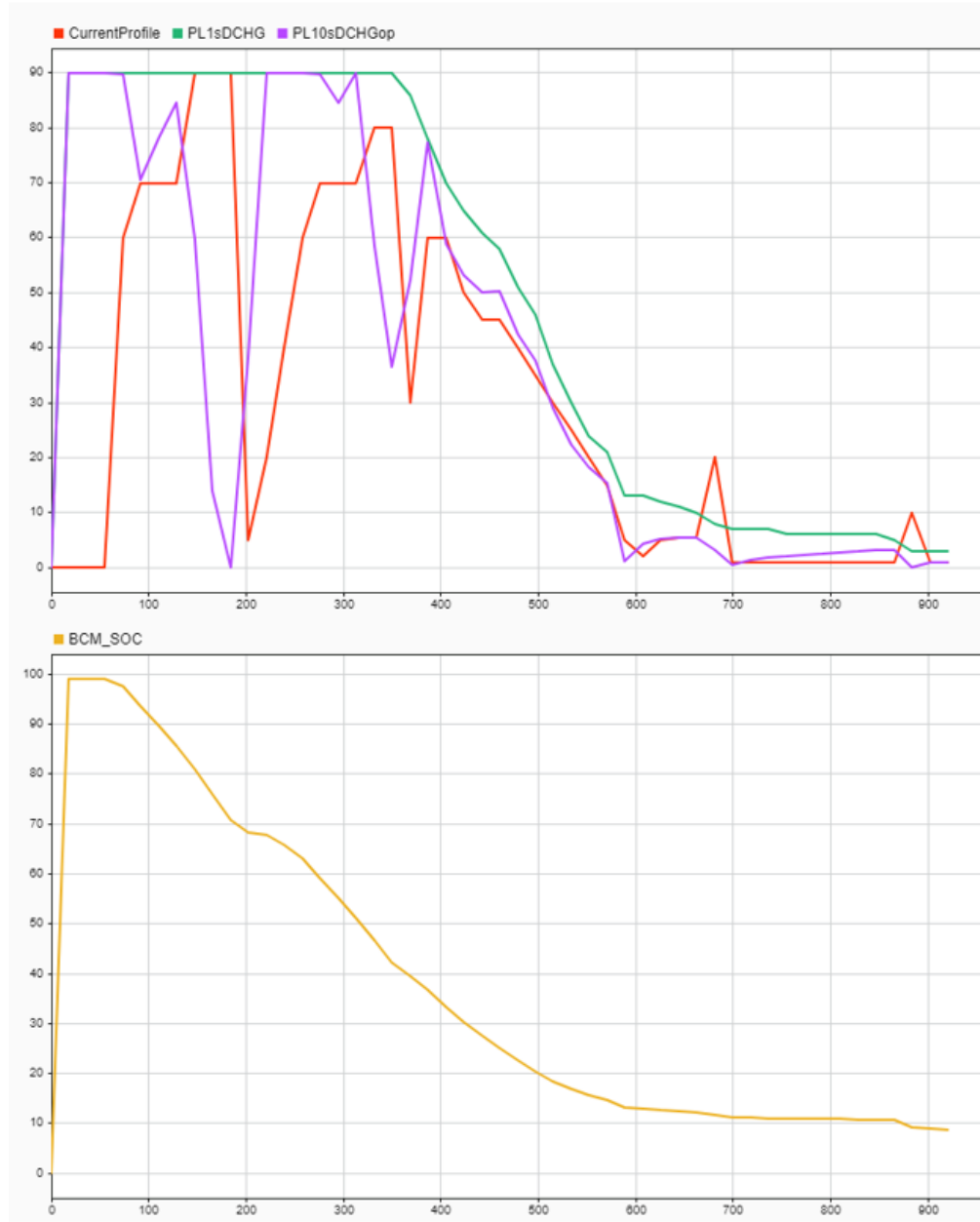


Figure 6.12: First trial tuning PID for power limits 1s and 10s under a DCHG random current

of the handwritten PI controller varies slightly in comparison to MATLAB's PI controller. However, the deviation is minimal and both algorithms do not exceed the power limits set by the PL1s algorithm.

This test not only validates that the optimized PL10s algorithm adopts a more

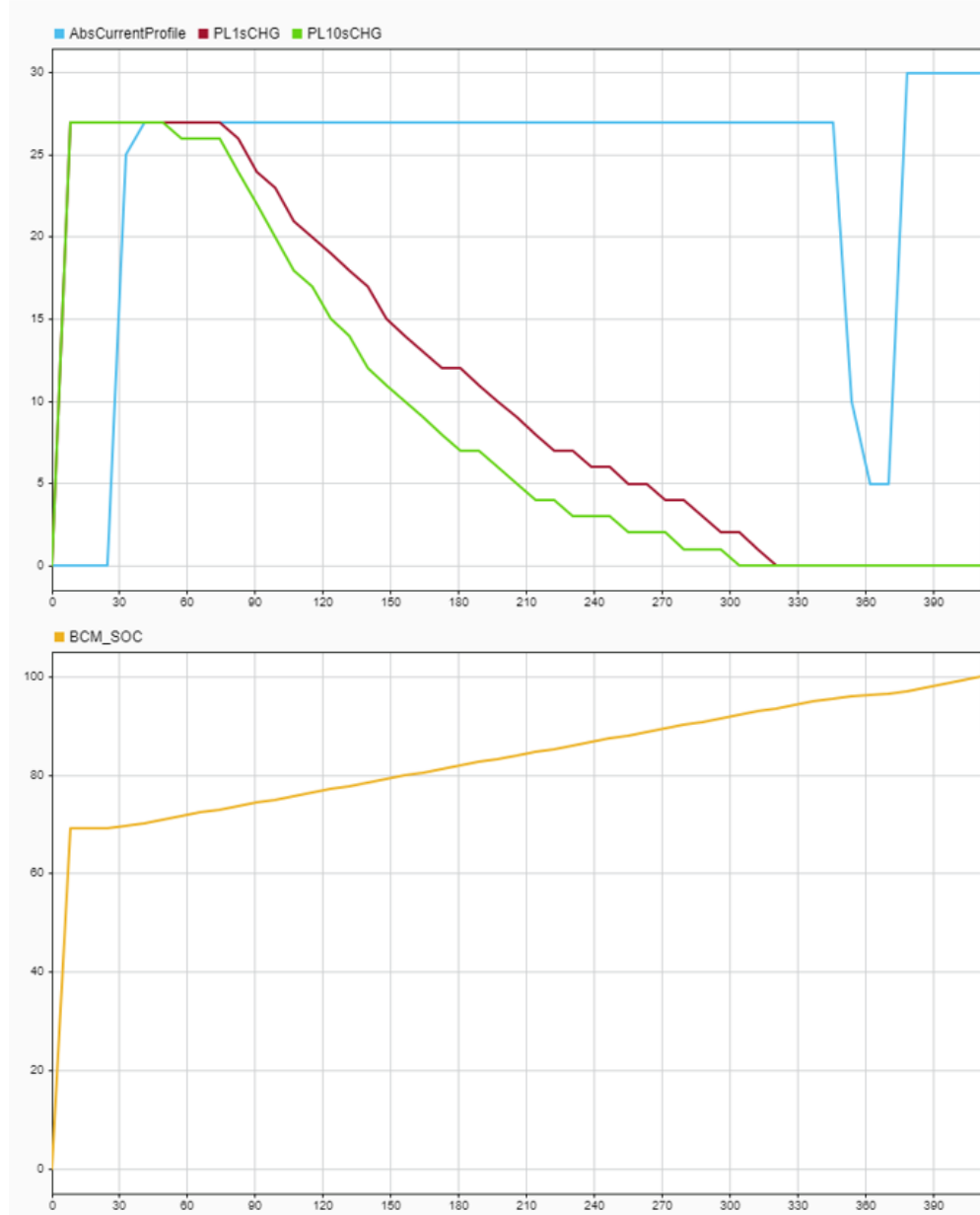


Figure 6.13: First trial tuning PID for power limits 1s and 10s under a CHG random current

conservative approach compared to the PL1s, but also highlights the effectiveness of the PI controller in regulating power limits. Moreover, it demonstrates the algorithm's adaptability through the straightforward tuning of the PI parameters.

In another test, the value of K_p was kept at 0.01 as before, while K_i was reduced to 0.0001 for the optimized PL10s. Values were kept the same for the non-optimized PL10s (PL10sDCHG). As shown in Figure 6.15, this reduction in K_i value resulted in a more conservative algorithm (PL10sDCHGop), since reducing the value of K_i leads to a slower elimination of the steady-state error.

A middle ground where the optimized PL10s algorithm is conservative, relatively non-aggressive and moderately fast at eliminating the steady-state error was achieved when K_p was set to 0.01 and K_i set to 0.001. The same test was carried out at a discharging current of 70 A and the results were plotted as shown in Figure 6.16.

These findings demonstrate the versatility of the PI controller in shaping the Power Limits algorithm's behavior. Through careful tuning of the PI parameters, the algorithm can be tailored to meet specific vehicle performance requirements while prioritizing battery health.

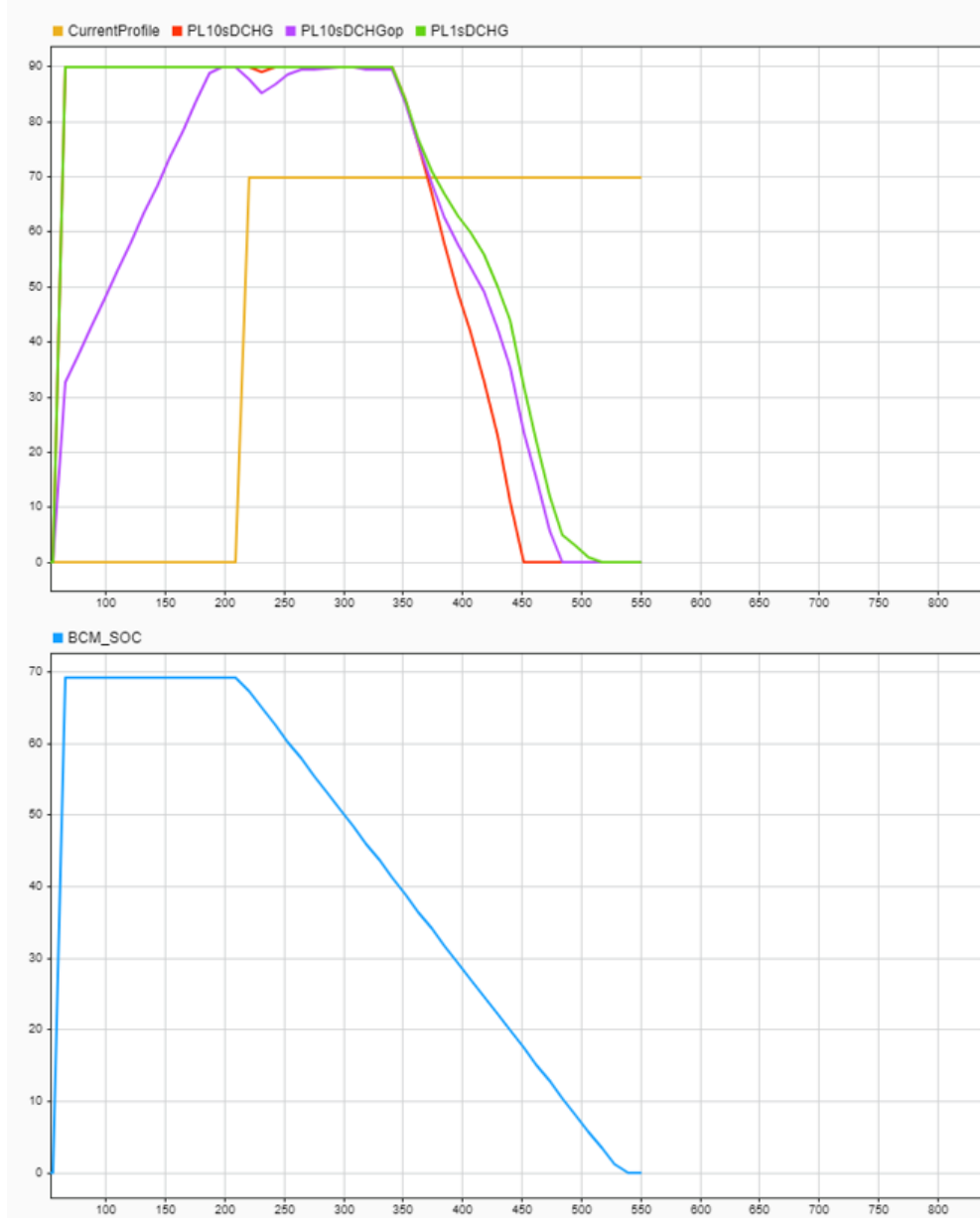


Figure 6.14: CANalyzer results, $K_p=0.01$, $K_i= 0.001$, 70 A DCHG current

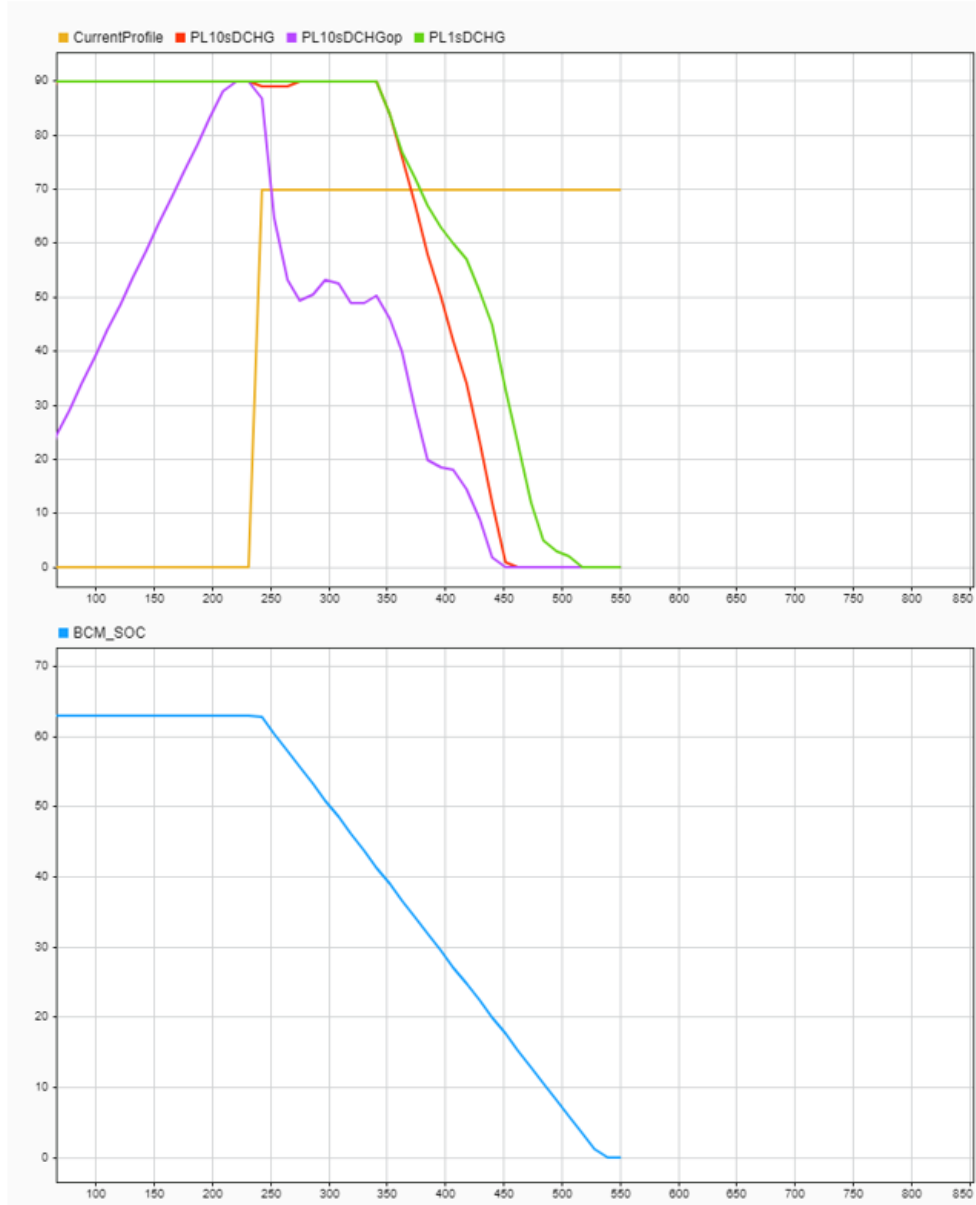


Figure 6.15: CANalyzer results, $K_p=0.01$, $K_i=0.0001$, 70 A DCHG current

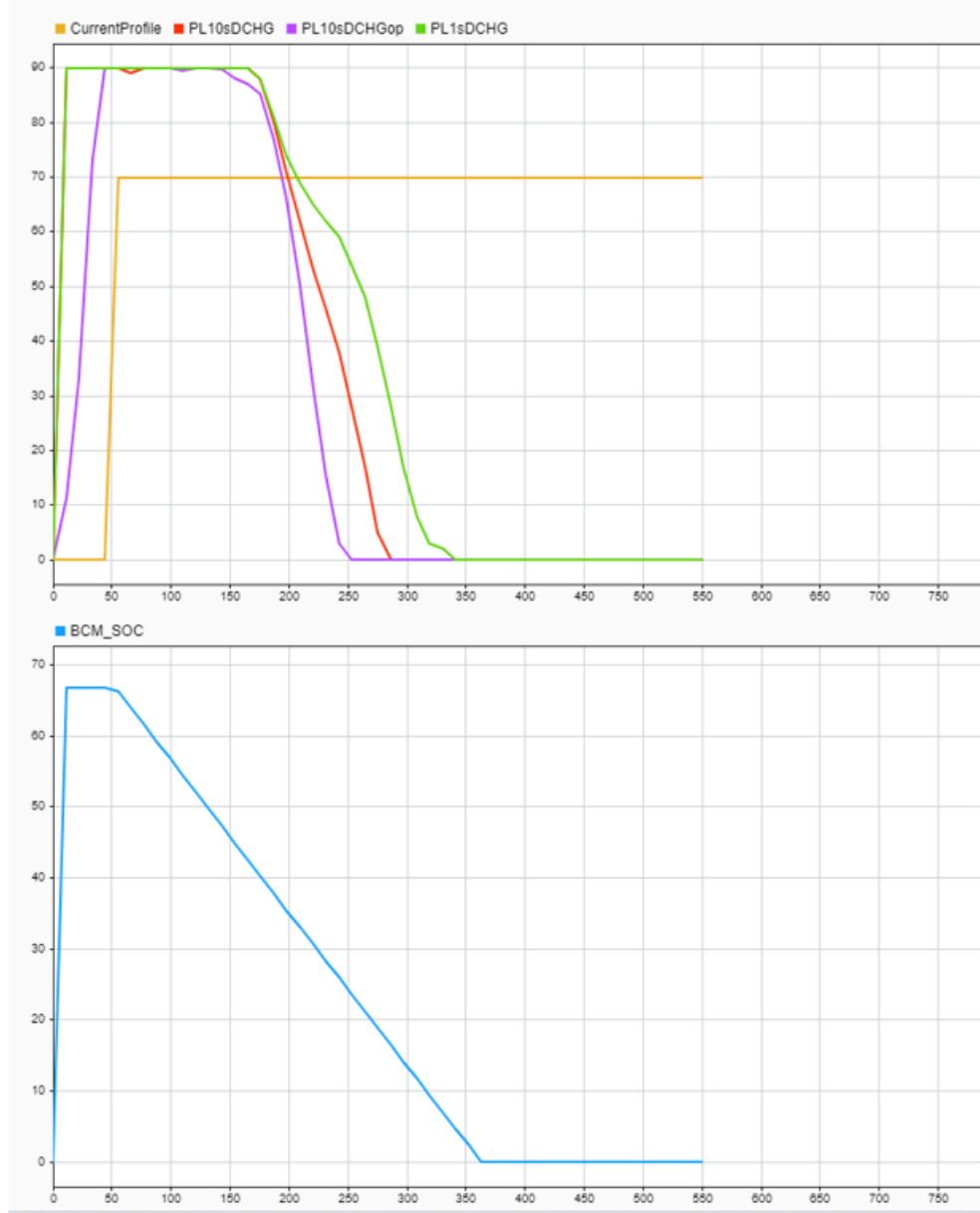


Figure 6.16: CANalyzer results, $K_p=0.01$, $K_i=0.001$, 70 A DCHG current

Chapter 7

Conclusions

As the adoption of electric vehicles (EVs) continues to rise, the demand for efficient and reliable battery management solutions has become more critical than ever. Lithium-ion (Li-ion) batteries, which power modern EVs, offer high energy density and longevity but require sophisticated monitoring and control systems to ensure safety, performance, and durability. The challenges associated with battery degradation, state estimation, and power management necessitate advanced software-driven solutions to optimize their operation. Without effective management, battery packs can suffer from reduced lifespan, performance inefficiencies, and safety risks. This highlights the crucial role of Battery Management Systems (BMS) in ensuring the safe, efficient, and reliable operation of Li-ion batteries, particularly in mission-critical applications where failure is not an option.

This thesis focused on the software aspects of the BMS, particularly the optimization and integration of key algorithms for state estimation and power management. The Extended Kalman Filter (EKF) was employed for accurate State-of-Charge (SOC) estimation, addressing the challenge of real-time SOC monitoring. Additionally, power limits algorithms were developed and integrated to dynamically regulate battery power output based on real-time conditions, improving both performance and longevity. Through extensive testing and optimization, the performance of these algorithms was significantly enhanced. The optimized EKF algorithm achieved a 50% reduction in execution time, improving real-time estimation capabilities. Moreover, the use of Embedded Coder in the code generation of the Power Limits algorithm gave rise to a remarkable 77.68% decrease in execution time compared to the hand-written code, emphasizing the advantages of automatic code generation for embedded systems.

Beyond the numerical improvements, the test results demonstrated the effectiveness of the PI controller in refining the power limits algorithm, allowing for a

customizable behavior to balance performance and battery health. These findings reinforce the importance of software-driven optimizations in BMS, as they not only enhance the operational efficiency of Li-ion batteries but also contribute to the overall reliability and viability of EV technology.

Possible future developments could focus on implementing a Model Predictive Control (MPC) strategy for managing power limits. This would involve the model-based design of the MPC algorithm, followed by code generation, system integration, and a thorough evaluation of the overall workflow, in addition to testing and validation of the results. In parallel, documenting the existing code-base is essential to support long-term software development. Generating structured code reports and maintaining clear, comprehensive documentation would streamline the integration of new models and support efficient, collaborative development across engineering teams.

In conclusion, this thesis discussed various advancements in battery management methodologies, including state estimation, power management, and code optimization. The results underscore the importance of integrating advanced computational methods with embedded systems to achieve efficient and safe battery operation. As EV adoption continues to grow, further research and development in intelligent battery management systems will be pivotal in enhancing performance, extending battery life, and ensuring the sustainability of electric transportation.

Bibliography

- [1] Gregory L. Plett. *Battery management systems: Volume II, Equivalent-Circuit Methods*. 2015 (cit. on pp. 1, 7, 12).
- [2] S. Zhang, C. Zhu, J. K. O. Sin, and P. K. T. Mok. «A Novel Ultrathin Elevated Channel Low-temperature Poly-Si TFT». In: 20 (Nov. 1999), pp. 569–571 (cit. on p. 2).
- [3] Michael Fowler Manh-Kien Tran. «A Review of Lithium-Ion Battery Fault Diagnostic Algorithms: Current Progress and Future Challenges». In: (2020) (cit. on p. 7).
- [4] MATHWORKS. *Model Configuration Parameters: Code Generation Optimization*. 2024. URL: <https://it.mathworks.com/help/rtw/ref/optimization-pane-general.html> (cit. on pp. 8, 38).
- [5] Zhenjie Cui & Weihao Hu & Guozhou Zhang & Zhenyuan Zhang & Z. Chen. «An extended Kalman filter based SOC estimation method for Li-ion battery». In: (2022) (cit. on p. 11).
- [6] Fauzia Khanum & Eduardo Loubach & Federico Duperly & Colleen Jenkins & Phillip J. Kollmeyer & Ali Emadi. «A Kalman Filter Based Battery State of Charge Estimation MATLAB Function». In: (2010) (cit. on p. 14).
- [7] Joshua Owoyemi. *Kalman Filter: Predict, Measure, Update, Repeat*. 2017. URL: <https://tjosh.medium.com/kalman-filter-predict-measure-update-repeat-20a5e618be66> (cit. on p. 17).
- [8] Davide Andrea. *Battery Management System for Large Lithium-Ion Battery Packs*. 2010 (cit. on p. 24).
- [9] Jamie Juviler. *What Is GitHub? (And What Is It Used For?)* 2021. URL: <https://blog.hubspot.com/website/what-is-github-used-for> (cit. on p. 40).
- [10] Ivor.p. *Master and Slave BMS*. 2023. URL: <https://www.batterydesign.net/master-slave-bms/> (cit. on p. 43).

- [11] Isabellenhütte. *Precision measurement technology with highly integrated, shunt-based digital current and voltage measurement sensors*. URL: <https://www.isabellenhuette.de/en/precision-measurement/applications#:~:text=A%20precise%20and%20high%2Dresolution,SoC%3B%20SoF%3B%20SoH> (cit. on p. 44).
- [12] Rhopoint Components. *Isabellenhütte IVT-S series*. 2020. URL: <https://www.rhopointcomponents.com/product/sensors/current-and-voltage-measurement/isabellenhutte-ivt-s-series/#:~:text=The%20shunt%2Dbased%20measurement%20method,drop%20into%20a%20digital%20signal> (cit. on p. 44).
- [13] STMicroelectronics. *STM32F427xx and STM32F429xx datasheet*. 2024. URL: <https://www.st.com/resource/en/datasheet/stm32f427vg.pdf> (cit. on p. 45).
- [14] Nordic Semiconductor ASA. *Technical Documentation: ST Nucleo F429ZI*. 2024. URL: https://docs.nordicsemi.com/bundle/ncs-2.4.3/page/zephyr/boards/arm/nucleo_f429zi/doc/index.html (cit. on p. 45).
- [15] Alan R. Earls. *Pre-Charge Circuits Lead to Safer EVs*, *ElectronicDesign*. 2024. URL: <https://www.electronicdesign.com/technologies/power/article/21280232/electronic-design-pre-charge-circuits-lead-to-safer-evs> (cit. on p. 47).