Implementation of Flexibility Sources and Advanced Control Algorithms for Supporting Modern Power Systems with a High Share of Renewable Energy Sources

Work Package WP1

Development and Refinement of Physics-Based and/or Equivalent Electrical Models for Battery Energy Storage Systems

Deliverable D.1.1

Development and Experimental Validation of a Battery Charger/Discharger Laboratory Setup for Battery Model Assessment

Funding Instrument: Source 581 – The Recovery and Resilience Facility
Call: IP-UNIST-05
Project Start: 1 October 2025
Project Duration: 48 months

Beneficiary in Charge: FESB

Dissemination Level		
PU	Public	✓
CO	Confidential, only for the project members	

Funded by the European Union
NextGenerationEU

Deliverable Information

Document Administrative Information	
Project Acronym:	FLEXsys
Project Number:	IP-UNIST-05
Deliverable Number:	D.1.1.
Deliverable Full Title:	Development and Experimental Validation of a Battery Charger/Discharger Laboratory Setup for Battery Model Assessment
Deliverable Short Title:	Experimental Battery Laboratory Setup for Model Assessment
Document Identifier:	FLEXsys-D1.1.pdf
Beneficiary in Charge:	FESB
Report Version:	V1.0
Contractual Date:	23/09/2026
Report Submission Date:	15/10/26
Dissemination Level:	PU
Nature:	Report
Lead Author(s):	D. Jolevski
Co-author(s):	
Keywords:	Battery energy storage system; battery charger/discharger; laboratory setup; battery modeling; state of charge; state of energy; energy efficiency; battery state estimation; real-time control; digital signal processor.
Status:	final

Change Log

Date	Version	Author/Editor	Summary of Changes Made
23/09/2026	v1.0	D. Jolevski	Final - end

Table of Contents

Deliverable Information 2
Executive Summary 8
1 Introduction 9
1.1 Background and Motivation 9
1.2 Report Structure 11
2 The laboratory setup for Testing and Evaluating Battery System economic models 13
2.1 Description of the Real-Time DSP Software for the Bidirectional Battery Charger/Discharger 17
2.1.1 Functional block diagram and external interfaces 17
2.1.2 Real-time measurement and low-level converter control 18
2.1.3 Supervisory activity and state-machine control 19
2.1.4 Protection and fault handling 23
2.1.5 Battery-state monitoring, SOC estimation, and reference SOC/SOE calculation 24
2.1.6 Reference SOC and SOE integration used for comparison 24
2.1.7 Time-critical SOC calculation in battery_supervising() 25
2.1.8 Non-time-critical estimation and adaptation in battery_supervising_run() 27
2.1.9 OCV-based SOC correction during battery rest 29
2.1.10 SOH calculation and battery-state synchronization with the HMI 30
2.1.11 Interpretation of the SOC/SOE activity and state diagrams 31
2.1.12 HMI communication and experimental supervision 35
2.1.13 Local LED display and status indication 36
2.1.14 Interpretation of the complete control sequence 37
2.2 Human-Machine Interface for Control, Monitoring, and Experimental Automation of the Battery System 38
2.2.1 General purpose and role of the HMI 38
2.2.2 Software architecture 38
2.2.3 Establishment of communication with the DSP 41
2.2.4 Bidirectional exchange of experimental variables 42
2.2.5 Definition of experimental operating conditions 44
2.2.6 Manual operation of the charger/discharger 46
2.2.7 DSP controller-state supervision 47
2.2.8 Automated experimental recipes 49
2.2.9 Overall application activity 51
2.2.10 Real-time acquisition and visualization 53
2.2.11 Experimental data recording 54

2.2.12 Fault response and operational supervision 55
2.2.13 Role of the HMI in validation of the laboratory setup 56
3 Experimental data 58
3.1 Analysis of Measured and Calculated State of Charge and State of Energy 58
3.1.1 Determination of Measured State of Charge 59
3.1.2 Offline Calculation of State of Charge 60
3.2 Determination of Measured State of Energy 61
3.2.1 Offline Calculation of State of Energy 61
3.2.2 Experimental Comparison of Measured and Calculated SOE 63
3.2.3 Experimental Comparison of Measured and Calculated SOC 64
3.2.4 Influence of Efficiency Estimation Error 65
3.2.5 Verification of the Real-Time Algorithms and Laboratory Setup 65
3.3 Influence of Energy-Efficiency Correction on the Calculated State of Energy 66
3.3.1 SOE Correction for a Single Discharge–Charge Sequence 67
3.3.2 SOE Correction during Repeated Charge–Discharge Operation 69
3.3.3 Long-Term Influence of Energy Efficiency on SOE Estimation 69
3.3.4 Comparative Evaluation of the Efficiency-Corrected SOE Results 70
3.3.5 Sensitivity of SOE to the Accuracy of Energy Efficiency 71
3.3.6 SOC estimator verification and analysis of results 72

List of Figures

Figure 1. Laboratory setup 13
Figure 2 *Position of battery monitoring and SOC/SOE calculation in the overall DSP software architecture.* 17
Figure 3 . Activity/state diagram of the battery charger/discharger supervisory control implemented by the BatteryController state machine. 20
Figure 4 *Activity diagram of SOC calculation and estimation, showing the time-critical interrupt calculation and the non-time-critical capacity, efficiency, OCV-correction, SOH, and HMI-update tasks.* 32
Figure 5 State-oriented representation of the battery-monitoring algorithm 34
Figure 6 Conceptual architecture of the Python HMI and its integration with the DSP-based battery charger/discharger. The implemented serial protocol is Modbus ASCII. 39
Figure 7 Main window of the Python HMI application for battery charger/discharger control and monitoring. 40
Figure 8 Conceptual supervisory operating-state diagram of the HMI for manual and recipe-based operation. 46
Figure 9 Conceptual representation of DSP battery-controller states monitored by the HMI. Transition conditions shown schematically should be interpreted together with the implemented DSP control algorithm. 48
Figure 10 Conceptual activity diagram of the Python HMI application, from initialization and DSP connection to user interaction, measurement processing, and application termination. 52
Figure 11 Measured vs. Calculated SOC over time for mj5-18650-20260617 (no efficiency correction)... 59
Figure 12 Measured vs. Calculated SOC over time for mj2-20260716-1 (no efficiency correction)..... 60
Figure 13 Measured vs. Calculated SOC over time for mj3-20260717 (no efficiency correction). 60
Figure 14 Measured vs. Calculated SOE over time for mj5-18650-20260617 (no efficiency correction)... 62
Figure 15 Measured vs. Calculated SOE over time for mj2-20260716-1 (no efficiency correction)..... 62
Figure 16 Measured vs. Calculated SOE over time for mj3-20260717 (no efficiency correction). 63
Figure 17 Calculated SOE over time for mj5-18650-20260617 (with efficiency correction). 67
Figure 18 Calculated SOE over time for mj2-20260716-1 (with efficiency correction). 68
Figure 19 Calculated SOE over time for mj3-20260717 (with efficiency correction)..... 68
Figure 20 Measured SOC over time for mj2-20260716-1 used for SOC-estimator verification..... 72
Figure 21 Measured SOC over time for mj3-20260717 used for long-duration SOC-estimator verification. 73

List of Tables

No table of figures entries found.

List of Abbreviations

Executive Summary

This technical report details the development and experimental validation of a flexible laboratory setup designed to test battery systems and evaluate battery models for microgrid and economic analyses. The setup incorporates a bidirectional DC/DC converter, battery system, measurement equipment, a Texas Instruments digital signal processor (DSP), and a PC-based human-machine interface (HMI). The DSP enables real-time current and voltage control, supervisory charger and discharger operation, and comprehensive battery-state monitoring, including state of charge (SOC), state of energy (SOE), capacity, efficiency, state of health (SOH), and open-circuit-voltage (OCV)-based SOC correction. The HMI facilitates system control, automated experimental protocols, real-time visualization, and experimental data acquisition.

Experimental verification compares battery-state quantities calculated in real time by the DSP with independent offline calculations from recorded measurements. The results confirm the consistency of the implemented SOC and SOE calculations and demonstrate the importance of accounting for battery losses. In particular, neglecting energy efficiency produces cumulative SOE deviation during repeated charge–discharge operation, whereas introducing an experimentally determined efficiency correction substantially improves the stability and repeatability of the estimated energy state. The SOC experiments also show how OCV-based correction compensates for accumulated estimation errors during battery rest and how estimated battery capacity and SOH affect long-term SOC accuracy.

The experimental results verify that the developed laboratory setup provides a suitable platform for battery characterisation, control-algorithm verification, parameter identification, and evaluation of simplified battery models. Accurate representation of SOC, SOE, usable capacity, efficiency, and SOH is particularly important when the resulting battery model is used for microgrid simulation and economic optimisation, since errors in the estimated available battery energy can directly affect calculated grid-energy exchange, battery utilisation, operating costs, and optimal storage capacity.

1 Introduction

1.1 Background and Motivation

The increasing penetration of renewable energy sources, the electrification of transportation and industry, and the growing demand for flexible and reliable electrical networks have significantly increased the importance of energy storage systems. Among the available energy-storage technologies, battery energy storage systems (BESSs) are especially important because they can respond rapidly to changes in electrical power demand and supply. This report focuses on batteries' role in absorbing excess electrical energy, returning stored energy to the grid when needed, balancing power, supporting renewable-energy integration, and improving microgrid operational flexibility.

However, the growing importance of battery systems also introduces new technical and economic challenges. A battery energy storage system is a significant investment, and its performance depends on many electrical, thermal, control, and operational parameters. Consequently, designers cannot base battery capacity selection solely on nominal energy demand. Instead, the designer must consider the expected load profile, renewable generation, charging and discharging requirements, converter limitations, battery efficiency, degradation, operating constraints, and the required reliability level.

For this reason, battery system analysis has become an important component of modern microgrid design. An unnecessarily large battery increases the initial investment and may remain underutilised for much of its lifetime. On the other hand, a battery that is too small may be unable to provide the required power or energy, resulting in increased dependence on the grid, curtailment of renewable generation, or failure to satisfy the required operating conditions. The problem is therefore an optimisation problem in which the battery capacity and other microgrid components must be selected according to both technical and economic criteria.

A reliable mathematical model is crucial for addressing this problem, as it enables the accurate representation of battery system behaviour without the need to physically build every potential configuration. Simulation can therefore evaluate different battery capacities, operating strategies, renewable-energy penetration levels, and microgrid configurations before selecting the final hardware. This reduces development time and cost while allowing investigation of more configurations.

However, a mathematical model alone is not sufficient. Battery models necessarily involve simplifications and assumptions. Parameters such as internal resistance, open-circuit voltage, state of charge (SOC), temperature, efficiency, and degradation behaviour may vary with operating conditions. Therefore, experimental validation is necessary to determine whether the mathematical representation adequately describes the real system's behaviour.

This creates a direct relationship between mathematical modelling and experimental investigation. The mathematical model can predict system behaviour and help investigate potential microgrid configurations. At the same time, the laboratory setup can be used to obtain experimental data, identify model parameters, validate simulation results, and evaluate control algorithms under controlled operating conditions.

The proposed laboratory platform therefore aims to do more than provide a physical battery charging and discharging system. It should offer a flexible experimental environment in which the electrical hardware, low-level controllers, high-level control algorithms, human-machine interface (HMI), measurement system, and automation functions can be modified and extended to meet future research requirements. The platform's core components include a digital signal processor that implements low- and high-level control

algorithms to realise a programmable bidirectional DCDC power converter using a buck-boost topology. It is needed to charge and discharge batteries with constant current and charge with constant voltage.

A simplified economic objective can be expressed as:

$$\min J = C_{CAPEX} + C_{OPEX} + C_{deg} + C_{grid} - R_{energy} \quad (1.1)$$

where, C_{CAPEX} represents investment costs, C_{OPEX} operating costs, C_{deg} battery aging costs, C_{grid} electricity-related costs, and R_{energy} the economic benefits obtained from battery operation.

The optimal solution must compare these costs against the storage system's technical benefits. For example, increasing the battery capacity may reduce the amount of electricity purchased from the external grid. It may also reduce renewable-energy curtailment and increase the microgrid's ability to operate autonomously. However, after a certain point, additional battery capacity may provide only a small additional benefit while considerably increasing the capital investment.

The accuracy of the estimated energy-related revenue, R_{energy} , is critical in determining the reliability of the overall economic assessment of a battery energy storage system (BESS). In general, R_{energy} is not an independent parameter, but a quantity resulting from the interaction between the battery characteristics, the converter, the microgrid operating conditions, and the supervisory energy-management strategy. For a given time horizon T, it can be expressed in a general form as

$$R_{\text{energy}} = \int_0^T \left[p_{\text{sell}}(t)P_{\text{dis}}(t) - p_{\text{buy}}(t)P_{\text{ch}}(t)\right] dt \quad (1.2)$$

where $P_{dis}(t)$ and $P_{ch}(t)$ represent the actual battery discharge and charging power, respectively. In contrast, $p_{sell}(t)$ and $p_{buy}(t)$ represent the corresponding energy prices or economic values. The achievable values of $P_{dis}(t)$ and $P_{ch}(t)$, however, are constrained by the battery's dynamic behaviour, SOC, SOH, terminal voltage, current limits, temperature, efficiency, and converter limitations. Consequently, any systematic mistake in the battery model can propagate through the calculated energy trajectory as well as ultimately produce a significant error in R_{energy}. This shows that the accuracy of the economic objective function depends fundamentally on the underlying battery model's ability to reproduce the energy exchange realistically achievable by the physical system.

The supervisory control system is another, equally important component of this relationship. Let the supervisory controller be represented by a control law $u(t) = \mathcal{C}(x(t), r(t), w(t))$, where $x(t)$ represents the battery state variables, $r(t)$ the desired operating objectives, and $w(t)$ the external operating conditions, such as load demand, renewable generation, and electricity prices. The resulting control action determines the charging and discharging trajectory subject to the physical constraints of the battery and converter. The battery condition evolution can, in simplified form, be represented by

$$\dot{x}(t) = f(x(t), u(t), w(t), \theta), \quad (1.3)$$

where θ denotes the identified battery-model parameters. Therefore, R_{energy} is ultimately a function of both the battery model and the supervisory control strategy, i.e., $R_{energy} = F(f, \mathcal{C}, \theta, w)$. Accordingly, an economic optimisation based on an idealised battery model or an oversimplified supervisory strategy may identify an operating point or battery capacity that appears optimal in simulation but cannot achieve the assumed energy exchange in practice. Therefore, it is necessary to use a battery model that is accurate enough to reproduce the dominant electrical and energetic characteristics as remaining computationally effective for repeated optimisation.

For this reason, the recommended methodology should adopt a simplified but experimentally validated battery model, selecting model detail based on its influence on quantities relevant to the economic objective.

The purpose is not to duplicate every electrochemical phenomenon, but to accurately represent the dominant relationships between SOC, terminal voltage, current, power, efficiency, and usable energy over the operating region of interest. The mathematical model parameters must be obtained and validated using the experimental measurements from the laboratory setup, while the supervisory controller should reproduce the actual operating logic, including charge/discharge transitions, SOC constraints, power limits, and protection-related restrictions. In this way, the resulting framework delivers a direct link between physical battery behaviour and economic evaluation: the validated model determines the feasible energy trajectory, the supervisory controller determines how that trajectory is exploited, and the resulting energy exchange determines R_{energy}. This is particularly important for optimal battery sizing, since an error in the predicted energy benefit can shift the calculated optimum toward an unnecessarily large or insufficiently sized storage system. Thus, combining a computationally effective, simplified, experimentally validated battery model with an accurately represented supervisory control strategy is essential to obtain a credible, technically defensible evaluation of the economic objective function.

1.2 Report Structure

This report is structured to provide a systematic description and experimental evaluation of the developed laboratory setup for testing battery systems and assessing battery-system models intended for subsequent economic and microgrid studies.

Section 2 presents the laboratory setup for testing and evaluating battery-system economic models. It describes the principal hardware and software components of the setup, including the bidirectional DC/DC converter, measurement system, Texas Instruments digital signal processor (DSP), battery system, and PC-based human-machine interface (HMI). Particular attention is given to the interaction between the physical battery, power-electronic converter, real-time control system, and experimental supervision.

Section 2.1 provides a detailed description of the real-time DSP software developed for the bidirectional battery charger/discharger. It describes the software architecture, real-time measurement acquisition, low-level current and voltage control, supervisory state-machine control, protection and fault handling, and battery-state monitoring. The implementation of state-of-charge (SOC) and state-of-energy (SOE) calculations, capacity and efficiency estimation, open-circuit-voltage-based SOC correction, state-of-health estimation, communication with the HMI, and the overall control sequence are also presented.

Section 2.2 describes the Python-based HMI used for control, monitoring, and automation of the experimental setup. The section explains the software architecture and communication with the DSP, bidirectional exchange of experimental variables, definition of operating conditions, manual charger/discharger operation, controller-state supervision, and automated experimental recipes. Real-time acquisition, visualization and recording of experimental data, fault response, and the role of the HMI in experimental validation are also discussed.

Section 3 presents the experimental data used to evaluate the developed laboratory setup and the implemented battery-state calculation algorithms. The measured and independently calculated SOC and SOE are compared to verify the consistency of the real-time DSP implementation with offline calculations and to investigate the accumulation of estimation errors during repeated battery charging and discharging.

Section 3.1 focuses on the determination and comparison of measured and calculated SOC, while Section 3.2 extends the analysis to SOE. The experimental results are used to examine differences between charge-based and energy-based battery-state representations and to demonstrate the greater sensitivity of SOE to energy losses and efficiency assumptions.

Section 3.3 investigates the influence of battery energy-efficiency correction on SOE estimation. Experimental results for individual and repeated charge-discharge sequences are analysed to demonstrate how the inclusion and accuracy of the estimated energy efficiency affect the calculated SOE trajectory. The results provide experimental support for the development of simplified but sufficiently accurate battery-system models suitable for subsequent microgrid and economic studies.

2 The laboratory setup for Testing and Evaluating Battery System economic models

The laboratory setup realized at Faculty Electrical Engineering and Naval Architecture in Croatia, Split comprises a battery energy storage system designed to reproduce the essential electrical and control characteristics of a practical battery charging and discharging system. The setup, shown on **Figure 1** integrates the physical battery, a controlled DC/DC power converter, measurement sensors, a Texas Instruments digital signal processor (DSP), and a PC-based human-machine interface (HMI). This arrangement provides a controlled and repeatable environment for investigating battery behaviour, implementing practical control strategies, acquiring experimental data, and validating battery-system models used for economic assessment.

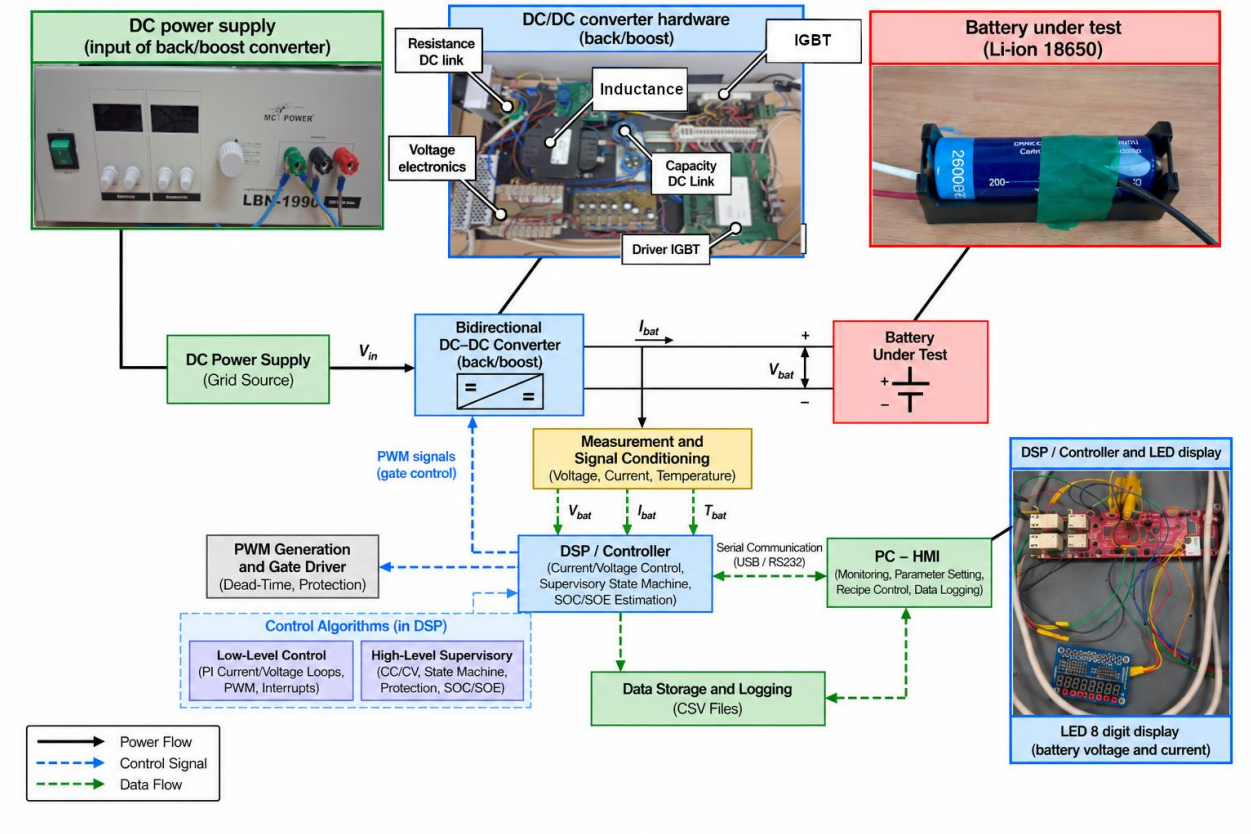


Figure 1. Laboratory setup

The energy-transfer subsystem is based on a controlled DC/DC converter that provides the electrical interface between the battery or the batteries' pack and the external DC system. A half-bridge power-electronic topology is employed as the main switching stage, enabling controlled bidirectional energy exchange. During charging, electrical energy is transferred from the DC side towards the battery, whereas during discharging the energy flow is reversed. The converter therefore determines the electrical operating point imposed on the battery and represents an essential part of the overall battery-system dynamics.

In contrast to an ideal laboratory source, the controlled DC/DC converter allows the battery to be operated through a regulated electrical interface representative of practical battery systems. The requested current, voltage, or power is realized through the switching operation of the converter and the corresponding control

algorithms. Consequently, the experimentally measured battery response incorporates the interaction between the inherent battery dynamics, power-electronic interface, and digital control system.

The real-time control and estimation functions are implemented on a Texas Instruments DSP, which represents the central computational unit of the system. The DSP performs real-time acquisition and processing of the measured electrical and thermal quantities and executes the low-level control algorithms, battery-state estimation, and supervisory control. The use of a dedicated DSP enables the control algorithms to operate with defined sampling and execution intervals, which is necessary for accurate and repeatable control of the power-electronic converter.

The low-level control system contains the current and voltage control algorithms required to regulate the operating point of the DC/DC converter. The current controller regulates the battery current according to the reference generated by the supervisory control system, while the voltage controller regulates the battery voltage when voltage-controlled operation is required. Depending on the selected operating mode, the voltage-control loop can determine the required current reference for the current-control loop. The resulting control action is converted into PWM signals that control the semiconductor switches of the half-bridge converter. These control algorithms are implemented using bare-metal real-time execution, with hardware interrupts generated at predefined time intervals and closely synchronized with the generation of the PWM signals for the power converters.

In addition to the converter control algorithms, the DSP performs real-time battery-state estimation. In particular, an SOC estimation algorithm is implemented using the available voltage and current measurements and the selected estimation methodology. The estimated SOC is continuously updated during battery operation and is made available to the supervisory control system. This allows the estimated battery state to be considered when determining permissible operating conditions and transitions between different operating modes.

The supervisory control algorithm provides the higher-level operational logic of the battery system. Its purpose is to determine the active operating state and coordinate the operation of the low-level controllers according to the requested operating condition and the applicable battery and system constraints. Typical operating states include idle, charging, discharging, constant-current charging, constant-voltage charging, charge termination, and fault conditions.

The supervisory controller is responsible for implementing the practical sequence of battery operation. During charging, for example, the system can initially operate in constant-current mode. When the battery reaches the specified maximum voltage, the supervisory algorithm changes the operating condition to constant-voltage charging. The charging process is subsequently terminated when the charging current decreases below the specified cutoff value. Similarly, discharge operation can be terminated when the minimum allowable battery voltage, minimum SOC, or another predefined operating limit is reached.

The transition between charging and discharging is also controlled by the supervisory algorithm. Therefore, changing from a discharge reference to a charge reference does not simply represent a change in the direction of the current. The supervisory system first evaluates the operating conditions and determines whether the transition is permitted, after which the appropriate references and control modes are activated. This approach allows the laboratory system to reproduce the operational behaviour of practical battery charging and discharging equipment.

The measurement subsystem provides the electrical and thermal information required for converter control, battery-state estimation, supervisory operation, protection, and experimental characterization. Battery current is measured using a dedicated current sensor located in the battery current path. The current measurement provides the feedback signal required by the current controller and is also used for SOC estimation and the determination of transferred charge and energy.

Battery voltage is measured using a dedicated voltage-sensing circuit. Because the battery voltage is generally not directly compatible with the input range of the DSP analogue-to-digital converter, appropriate signal conditioning is used to adapt the measured voltage to the required input range. The voltage measurement is used for voltage regulation, SOC estimation, identification of charging and discharging limits, detection of charging-state transitions, and protection functions.

Battery temperature is monitored using a dedicated temperature sensor positioned to provide a representative measurement of the battery thermal condition. Temperature information is used by the supervisory control system to prevent operation outside the permitted temperature range. It also provides important information for experimental characterization because battery electrical characteristics, capacity, resistance, and efficiency can vary with temperature.

The measured electrical quantities also provide the basis for determining important energetic characteristics of the battery system. Battery voltage and current are used to determine instantaneous power, while their integration over time provides the transferred charge and energy. These quantities are subsequently used to evaluate charging and discharging behaviour, energy efficiency, losses, and usable battery capacity.

A PC-based HMI is implemented to provide convenient configuration, monitoring, experiment control, and data management. The HMI enables the operating parameters of the battery system to be configured without requiring modification of the embedded DSP software for each individual experiment. This is particularly important when a large number of experimental tests are required for battery characterization and model validation.

The configuration interface allows the operator to define the principal operating parameters, including the selected operating mode, charging and discharging references, voltage and current limits, and other experiment-specific settings. The HMI also provides real-time visualization of measured and estimated quantities, including battery voltage, current, temperature, SOC, and the active operating state.

An important function of the HMI is the definition of experimental **recipes**. A recipe defines a sequence of operating conditions and reference values that can be executed automatically by the DSP. For example, a recipe may define an initial charging period at a specified current, a subsequent transition to voltage-controlled operation, a discharge period, and a final idle state. More complex sequences can also be defined, including changes in current reference, changes between charging and discharging, predefined pauses, voltage limits, and other operating conditions.

The recipe functionality provides an important advantage for experimental characterization because predefined operating profiles can be repeated under the same conditions. This improves the repeatability of the experiments and enables direct comparison between experimental measurements and model simulations. It also allows the response of the system to different operating profiles to be systematically investigated.

The HMI additionally enables the operator to start and stop experiments and to request changes in the battery operating condition. This includes requests to change from discharging to charging or from charging to discharging. However, the actual transition remains under the control of the supervisory algorithm implemented in the DSP. The HMI therefore defines the requested operating condition, while the DSP determines whether the requested transition can safely be executed according to the battery voltage, current, SOC, temperature, and other applicable operating constraints.

The HMI also provides data-acquisition and storage functionality. The measured and calculated variables can be transferred to the PC and recorded together with timestamps and operating-state information. The recorded dataset can include battery voltage, battery current, temperature, estimated SOC, voltage and current references, operating state, power, transferred charge, and energy. The experimental data can

subsequently be used for data processing, visualization, parameter identification, and validation of the battery-system model.

The laboratory setup therefore provides an integrated environment for controlled battery operation and experimental characterization. The physical battery provides the inherent electrical and energetic dynamics, while the controlled DC/DC converter realizes the required energy exchange with the external electrical system. The Texas Instruments DSP provides the real-time implementation of current and voltage control, SOC estimation, and supervisory operating logic, while the measurement subsystem supplies the feedback information required for closed-loop operation. The PC-based HMI enables system configuration, recipe definition, monitoring, experiment control, and systematic recording of the resulting experimental data.

An important characteristic of the setup is that the experimentally observed response represents the behaviour of the complete controlled battery system, rather than the response of the battery considered independently. The physical battery dynamics interact with the power-electronic converter and the control algorithms implemented in the DSP. Consequently, the experimental measurements provide information about both the inherent battery characteristics and the operational dynamics introduced by the control system. This combined behaviour forms the experimental basis for developing and validating a simplified but sufficiently accurate battery-system model for subsequent economic analysis, battery capacity sizing, and evaluation of energy exchange in a microgrid or other power-system application.

2.1 Description of the Real-Time DSP Software for the Bidirectional Battery Charger/Discharger

The embedded program implements the real-time control and supervisory functions of a bidirectional battery charger/discharger based on a Texas Instruments C2000 DSP. The software integrates measurement acquisition, current and voltage control, PWM generation, converter mode selection, battery-state supervision, protection, communication with the PC-based human-machine interface (HMI), and local visualization. This organization makes the laboratory setup suitable not only for executing charge and discharge experiments, but also for modifying low-level control laws and high-level supervisory logic when alternative battery models or operating strategies are investigated.

From a functional perspective, the software can be divided into three interacting layers. The first is the real-time converter-control layer, executed by interrupt-driven routines and responsible for ADC acquisition, current/voltage control, and PWM duty-cycle generation. The second is the supervisory layer, implemented as a state machine that determines whether the converter is inactive, charging in constant-current (CC) mode, charging in constant-voltage (CV) mode, discharging in CC mode, or located in a completed or protection state. The third layer comprises experimental supervision and communication, through which reference values, limits, commands, measurements, SOC/SOE quantities, controller state, and alarms are exchanged with the PC HMI.

2.1.1 Functional block diagram and external interfaces

Figure 2 presents the position of battery monitoring and SOC/SOE calculation within the overall software architecture and summarizes the principal interaction between the HMI, DSP software, real-time interrupt tasks, power stage, sensors and battery. The C2000 DSP forms the central real-time control platform and connects the HMI, supervisory logic, low-level controllers, measurement chain, protection functions, and bidirectional DC-DC power stage. The PC HMI provides the experimental operator with access to charge/discharge references, the CV voltage reference, discharge cut-off voltage, current and temperature limits, and Start/Stop/Reset commands. In the opposite direction, measured battery voltage, current, power and temperature, together with SOC/SOE, controller state and alarm information, are made available for monitoring and data acquisition.

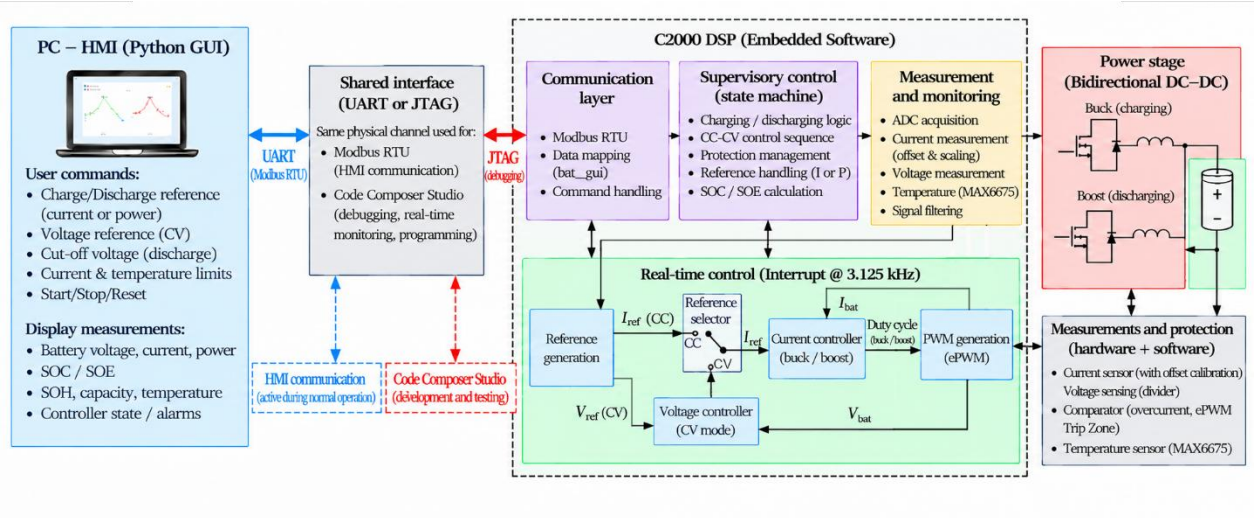


Figure 2 *Position of battery monitoring and SOC/SOE calculation in the overall DSP software architecture.*

The laboratory interface is additionally used during software development and commissioning. The communication path used for HMI operation is associated with the same physical debugging connection through which Code Composer Studio accesses the DSP over the JTAG interface. Consequently, the experimental setup supports two complementary operating functions through the shared development/communication connection: Modbus-based data exchange with the Python HMI during normal experimental operation and Code Composer Studio access for program loading, debugging and real-time observation during development and testing. This distinction is important: the C source directly implements the UART/Modbus communication, whereas the JTAG/Code Composer Studio path is a property of the laboratory hardware and development configuration rather than of the Modbus application protocol itself.

2.1.2 Real-time measurement and low-level converter control

The principal low-level control routine is executed as an interrupt service routine. At the beginning of each control cycle, the battery-voltage and battery-current ADC results are acquired. The current measurement is converted from ADC counts into amperes using the calibrated sensor offset and scaling factors. State-dependent correction terms are subsequently applied to compensate the average-current displacement associated with switching ripple in buck and boost operation.

The same interrupt contains separate current controllers for the charging and discharging directions. When the buck converter is enabled, the measured battery current is directly used by the buck current controller and its output determines the buck PWM duty cycle. During boost operation, the sign of the measured current is inverted before it is supplied to the boost current controller, thereby maintaining a consistent positive internal control convention. The voltage controller is evaluated in the same interrupt; its output becomes the current reference when the supervisory state machine enters CV charging. This realizes a cascaded CC-CV charging structure in which the voltage loop limits the current as the battery approaches the prescribed charging voltage.

The corresponding interrupt-level acquisition and controller calculations are summarized in **Code excerpt 1**.

```
493  __interrupt void INT_STRUJA_BAT_1_ISR(){
494
495
496      battery_voltage_adc = ADC_readResult(ADCARESULT_BASE, ADC_SOC_NUMBER1);
497
498      battery_current_adc = ADC_readResult(ADCARESULT_BASE, ADC_SOC_NUMBER0);
499
500
501      current_sensor_A = (battery_current_adc * V_REF/4095.f -
offset_current_sensor_v)*3.6711f*1.05f*9.0f/SENZOR_STRUJNI_BROJ_ZAVOJA;
502
503
504      //Correction of current to be average value
505      if(
506              BatteryController_CHARGING_CC == BatteryController_state
507          ||  BatteryController_CHARGING_CV == BatteryController_state
508      ){
509
510          current_sensor_A += CORRECTION_OFFSET_RIPPLE_BUCK;
511
512      }else if(BatteryController_DISCHARGING_CC == BatteryController_state){
513
514          current_sensor_A += CORRECTION_OFFSET_RIPPLE_BOOST;
515
516      }
517
518
519
520
521
```

```
522     if(buckConverterEnabled && !boostConverterEnabled){
523         //Regulator struje buck converter
524         uk2 = DCL_runPID_C1(&current_controller_buck, rk2, current_sensor_A, lk2);
525         setDutyCylceBuckConverter(uk2);
526 
527     }else if(boostConverterEnabled && !buckConverterEnabled){
528         //Regulator struje boost converter
529         float32_t current = -current_sensor_A;
530         uk2 = DCL_runPID_C1(&current_controller_boost, rk2, current, lk2);
531         setDutyCylceBoostConverter(uk2);
532 
533     }
534 
535 
536         //Regulator napona za CV mod
537     uk = DCL_runPID_C1(&voltage_controller, rk, yk, lk);
538 
```

Code excerpt 1 ADC acquisition and real-time buck/boost current and voltage control.

The interrupt also checks the ADC interrupt-overflow flag. An overflow indicates that the required execution timing has not been maintained. In this case, the program invokes the safety-stop routine, disables both converter directions and remains in a stopped condition. The timing check therefore converts a missed real-time deadline into a deterministic safe response rather than allowing the power converter to continue with potentially stale control information.

The implemented real-time deadline supervision and deterministic safety response are shown in **Code excerpt 2**.

```
549     //Resetira se ako bi se moglo detektirati je li trajanje ovog koda bilo duze nogo cycle time
550     ADC_clearInterruptStatus(STRUJA_BAT_BASE, ADC_INT_NUMBER1);
551 
552     //Detect if overflow is detected for ISR of PID
553     if(ADC_getInterruptOverflowStatus(STRUJA_BAT_BASE, ADC_INT_NUMBER1)){
554 
555         ADC_clearInterruptOverflowStatus(STRUJA_BAT_BASE, ADC_INT_NUMBER1);
556         //Ako je preskocen sample time od regulatora tada se iskljucuje sve u funkciji safety_stop()
557         safety_stop();
558 
559     }
1728 void safety_stop(){
1729 
1730     //do turn of device
1731 
1732     disableBuckConverter();
1733     disableBoostConverter();
1734 
1735     while(1);
```

Code excerpt 2 Real-time deadline supervision and safety stop.

2.1.3 Supervisory activity and state-machine control

The activity/state diagram in **Figure 3** represents the high-level operating logic implemented by the BatteryController_state switch statement. After power-up, the controller enters INIT. Both buck and boost converters are disabled, the status indication represents the initialization condition, and the system waits for a Start command. After Start, the controller enters RUN, which acts as a neutral decision state. A positive current or power request initiates charging, whereas a negative reference initiates discharging. This intermediate state provides a controlled mechanism for changing the direction of power flow rather than directly switching between buck and boost operation.

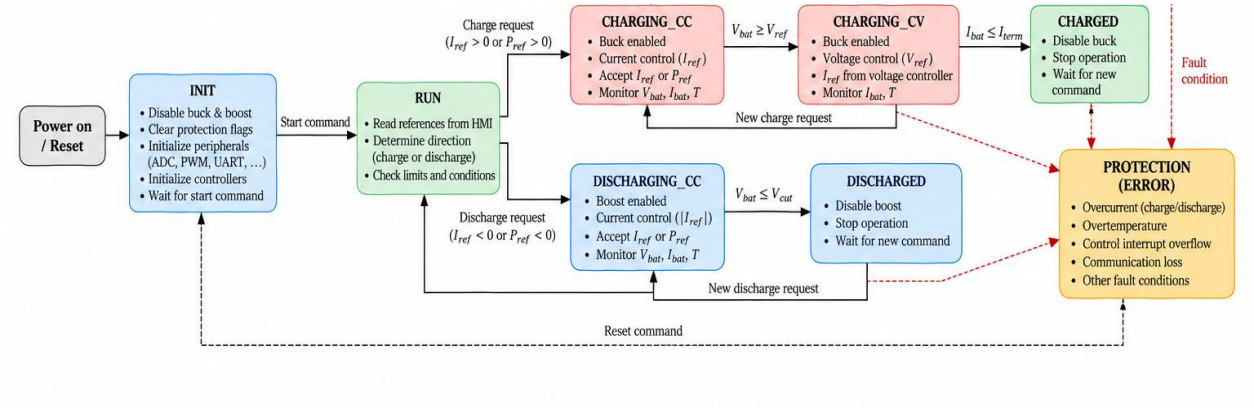


Figure 3 . Activity/state diagram of the battery charger/discharger supervisory control implemented by the BatteryController state machine.

Figure 3 presents the supervisory state diagram of the bidirectional battery charger/discharger and shows how the high-level control algorithm coordinates the operating mode of the power converter. In contrast to the low-level current and voltage controllers, which directly determine the converter duty cycle, the supervisory controller determines which operating condition is permitted at a given instant. The state machine therefore establishes the logical connection between the operating references received from the PC HMI, the measured battery variables, the requested direction of power transfer, and the low-level control loops.

After initialization, the controller is placed in the INIT state, in which active power transfer is disabled and the control system is brought to a defined initial condition. Following an operating command, the controller can enter the RUN state, which provides an intermediate supervisory condition from which the requested charging or discharging mode is selected. The use of this intermediate state is particularly useful in a bidirectional converter because it prevents an uncontrolled direct transition between two opposite directions of power flow and allows the associated controller variables and converter-enable signals to be prepared before an active operating mode is entered.

When charging is requested, the supervisory algorithm enters the constant-current charging state, CHARGING_CC. In this state, the current controller regulates the battery current according to the charging reference supplied by the higher-level control or HMI. At the same time, the battery voltage is continuously monitored. The constant-current state is therefore maintained only while the voltage remains below the prescribed charging-voltage limit. This organization separates the task of current regulation from the supervisory decision that determines when the battery has reached the operating region in which voltage limitation becomes dominant.

When the measured battery voltage reaches the specified charging-voltage reference, the state machine initiates the transition from CHARGING_CC to CHARGING_CV. In the constant-voltage state, the voltage controller determines the current demand supplied to the inner current-control loop, while the converter continues to regulate the battery operating point through the low-level current controller. The charging current consequently decreases as the battery approaches the fully charged condition. The state transition therefore implements the characteristic CC-CV charging sequence while preserving the cascaded organization of the low-level control structure.

The supervisory logic also allows the charging operating point to respond to changes introduced from the HMI or from a higher-level experimental procedure. Consequently, the state machine is not limited to a rigid, one-directional sequence that can only progress from CC to CV. If the requested operating point changes sufficiently, the controller can return to the appropriate current-controlled condition. This flexibility is

important for the developed laboratory setup because reference values can be changed manually or through an automated recipe while an experiment is in progress.

Completion of charging is identified when the termination criterion is satisfied and the controller enters the CHARGED state. At this point, active charging is terminated and the supervisory algorithm records that a fully charged condition has been reached. In addition to its direct role in charger operation, this state provides an event used by the battery-monitoring algorithm described in the following section. In particular, the fully charged event defines a cycle boundary required for the evaluation of accumulated charge quantities and charge-efficiency estimation.

When discharge operation is requested, the controller enters the constant-current discharging state, DISCHARGING_CC. The converter is operated in the direction required to extract energy from the battery, while the current-control loop regulates the requested discharge current. During this process, the battery voltage is continuously compared with the defined discharge cut-off voltage. Once the cut-off condition is reached, active discharging is disabled and the controller enters the DISCHARGED state. Similar to the CHARGED state, the DISCHARGED state also provides a well-defined event for the battery-monitoring functions, particularly for capacity-learning procedures that require a recognized full-discharge condition.

Figure 3 further shows that the normal operating states are complemented by dedicated protection and recovery paths. Abnormal operating conditions, such as excessive current or battery temperature, cause the supervisory controller to leave the normal charging or discharging sequence and enter the corresponding protection condition. In this way, protection is incorporated directly into the state-machine logic rather than being treated only as a numerical limitation applied to a reference signal. Recovery from a protection state requires the appropriate reset sequence before normal converter operation can resume.

The Stop command provides an additional supervisory mechanism for terminating an active operating mode and returning the system to a non-active condition. Together with the RUN state and the protection transitions, this ensures that starting, stopping, changing the direction of power flow, and recovering from abnormal conditions are coordinated by the high-level controller. Such coordination is particularly important in the laboratory setup, where operating points can be modified repeatedly during experimental testing and where the converter must remain in a predictable state after each user or protection event.

Overall, **Figure 3** illustrates the hierarchical organization of the implemented control system. The supervisory state machine determines what operating mode is permitted and when a transition between modes should occur, whereas the low-level current and voltage controllers determine how the requested electrical operating point is achieved by the converter. This separation provides the laboratory setup with the required flexibility to modify high-level charging and discharging sequences, transition criteria, termination conditions, and protection logic independently of the low-level converter-control algorithms. It therefore supports experimental evaluation of different battery supervisory strategies without requiring changes to the underlying power-electronic hardware.

The implemented CC charging reference generation and the transition to CV operation are shown in **Code excerpt 3**.

```
1298                case BatteryController_CHARGING_CC:
1299
1300
1301                    if(!GUI_CONTROLS.control_bits.bits.current_power_ref){
1302                        //Referenca se zadaje pomocu HMI-a
1303                        current_ref_A = GUI_REFERENCES.battery_current_A_ref;
1304                    }else{
1305                        //Referenca je snaga baterije, a zadaje se preko field od struje
1306                        if(GUI_MEASURE.napon_baterije_V > MIN_VOLTAGE_POWER_REF)
```

```
1307                            current_ref_A = GUI_REFERENCES.battery_power_W_ref /
GUI_MEASURE.napon_baterije_V;
1308                        else
1309                            current_ref_A = 0.0f;
1310                    }
1311
1312                    if(     CURRENT_REF_SELECTED == GUI_CONTROLS.control_bits.bits.current_power_ref &&
GUI_REFERENCES.battery_current_A_ref <= 0.0f
1313                        ||  POWER_REF_SELECTED == GUI_CONTROLS.control_bits.bits.current_power_ref &&
GUI_REFERENCES.battery_power_W_ref <= 0.0f
1314                    ){
1315
1316                        BatteryController_state = BatteryController_RUN;
1317                        disableBuckConverter();
1318
1319                    }
1320
1321
1322                    if(GUI_REFERENCES.voltage_battery_V_ref <= GUI_MEASURE.napon_baterije_V){
1323
1324                        //Izlaz regulatora se izjednacava s referentnom vrijednosti regulatora
1325                        //struje (blagi prijelaz iz CC u CV)
1326                        voltage_controller.i10 = current_ref_A;
1327                        //Integriranje ulazne pogreske se ukljucuje
1328                        lk = true;
1329
1330                        BatteryController_state = BatteryController_CHARGING_CV;
1331                    }
```

Code excerpt 3 CC charging reference generation and transition from CC to CV.

In CHARGING_CV, the output of the voltage controller is explicitly assigned as the battery-current reference. Charging terminates when the measured current falls below the configured termination-current threshold, after which the controller enters CHARGED. The code also permits a return from CV to CC when a newly selected current or power reference becomes more restrictive than the present operating point. Thus, the charging logic is not a fixed one-way sequence; it can respond to modified experimental references while the test is running.

The CV current-reference generation, termination criterion, and possible return to CC operation are summarized in **Code excerpt 4**.

```
1367                case BatteryController_CHARGING_CV:
1368
1369                    current_ref_A = uk;
1370
1371                    //Promjena, tj. prelazak iz punjenja u praznjenje (prolazi se krosz stanje RUN)
1372                    if(         CURRENT_REF_SELECTED == GUI_CONTROLS.control_bits.bits.current_power_ref
1373                            &&  GUI_REFERENCES.battery_current_A_ref <= 0.0f
1374                        ||      POWER_REF_SELECTED == GUI_CONTROLS.control_bits.bits.current_power_ref
1375                            &&  GUI_REFERENCES.battery_power_W_ref <= 0.0f
1376                    ){
1377
1378                        BatteryController_state = BatteryController_RUN;
1379                        disableBuckConverter();
1380
1381                    }
1382
1383                    //Uvjet za prestanak punjenja
1384                    if(GUI_MEASURE.struja_baterije_A < GUI_REFERENCES.current_term_cc_A){
1385                        BatteryController_state = BatteryController_CHARGED;
1386                    }
1387
1388                    //Prelazak iz CV moda punjenja u CC mod
1389                    if(         GUI_REFERENCES.battery_current_A_ref < GUI_MEASURE.struja_baterije_A
1390                            &&  CURRENT_REF_SELECTED == GUI_CONTROLS.control_bits.bits.current_power_ref
1391                        ||
```

```
                                GUI_REFERENCES.battery_power_W_ref < GUI_MEASURE.snaga_baterije_W
                            &&  POWER_REF_SELECTED == GUI_CONTROLS.control_bits.bits.current_power_ref
                    ){

                        BatteryController_state = BatteryController_CHARGING_CC;

                    }
```

Code excerpt 4 . CV charging, termination criterion, and return to CC.

During DISCHARGING_CC, the boost converter is used to regulate the magnitude of the discharge current. As in charging, the reference may be specified either as current or power. Discharging is terminated when the measured battery voltage falls below the user-defined cut-off voltage, at which point the state changes to DISCHARGED and the boost converter is disabled. A non-negative reference causes the controller to return to RUN, from which a subsequent charging request can be processed safely.

The implemented constant-current discharge logic and voltage cut-off transition are shown in **Code excerpt 5**.

```
                case BatteryController_DISCHARGING_CC:

                    if(GUI_CONTROLS.control_bits.bits.current_power_ref == CURRENT_REF_SELECTED){
                        //Upravljanje strujom pomocu field za struju na GUI od HMI-a
                        current_ref_A = GUI_REFERENCES.battery_current_A_ref;
                    }else{
                        //Upravljanje snagom pomocu filed za snagu na GUI od HMI-a
                        if(GUI_MEASURE.napon_baterije_V > MIN_VOLTAGE_POWER_REF)
                            current_ref_A = GUI_REFERENCES.battery_power_W_ref /
GUI_MEASURE.napon_baterije_V;
                        else
                            current_ref_A = 0.0f;
                    }

                    if(GUI_REFERENCES.cutoff_V > GUI_MEASURE.napon_baterije_V){

                        BatteryController_state = BatteryController_DISCHARGED;
                        disableBoostConverter();

                    }

                    if(     GUI_CONTROLS.control_bits.bits.current_power_ref == CURRENT_REF_SELECTED &&
GUI_REFERENCES.battery_current_A_ref >= 0.0f
                        ||  GUI_CONTROLS.control_bits.bits.current_power_ref == POWER_REF_SELECTED &&
GUI_REFERENCES.battery_power_W_ref >= 0.0f
                    ){
                        BatteryController_state = BatteryController_RUN;
                        disableBoostConverter();
                    }
```

Code excerpt 5 Constant-current discharge and cut-off-voltage transition.

2.1.4 Protection and fault handling

Protection is integrated at both hardware-assisted and software-supervisory levels. One-shot ePWM Trip Zone events are used to detect overcurrent conditions separately for charging and discharging, while battery temperature is compared with the protection threshold supplied through the supervisory parameters. These events force the state machine into dedicated overcurrent or overtemperature states. The corresponding reset procedure returns the current reference to zero, disables the converters, clears the relevant protection status, returns the controller to INIT, and resets controller integrator states. The software therefore requires an explicit recovery action before normal operation is resumed.

The safety concept also includes prevention of simultaneous buck and boost operation, monitoring of the real-time control interrupt, and supervision of the PC communication. Together, these mechanisms separate normal state transitions from abnormal operating conditions and ensure that faults do not simply alter the numerical reference but instead cause an explicit transition toward a non-power-producing state.

2.1.5 Battery-state monitoring, SOC estimation, and reference SOC/SOE calculation

The battery-state functionality is implemented at two different computational levels and must be distinguished from the simple SOC/SOE integration retained in the main program for experimental comparison. The simple integration represents an older reference approach: measured current and instantaneous battery power are integrated at a fixed one-second interval without the adaptive corrections implemented by the battery-monitoring module. The principal SOC monitoring and estimation algorithm is instead contained in the battery_supervising() and battery_supervising_run() functions. Their separation reflects the different timing requirements of the underlying calculations.

The separation between time-critical and non-time-critical battery-monitoring tasks can be identified from the software organization presented in **Figure 2**. The time-critical part of the battery-state algorithm is executed within the one-second interrupt through battery_supervising(), where measurement-dependent calculations requiring a precisely defined sampling interval are performed. These operations primarily include coulomb counting, accumulation of the transferred charge quantities, and the corresponding SOC update, thereby ensuring that the numerical integration is synchronized with the interrupt time base. In contrast, battery_supervising_run() is executed within the main program loop and performs event-driven or slowly varying operations that do not require deterministic execution at each sampling instant. These include capacity learning, charge-efficiency adaptation, open-circuit-voltage-based SOC correction after a defined rest period, SOH calculation, and synchronization of the estimated battery-state variables with the PC HMI. Thus, the software organization separates sampling-dependent battery-state calculations from slower estimation, adaptation, and communication-related tasks.

2.1.6 Reference SOC and SOE integration used for comparison

The main program retains a simple integration-based calculation of SOC- and SOE-related quantities. When the controller is in CHARGING_CC, CHARGING_CV or DISCHARGING_CC and the magnitude of battery current exceeds 0.02 A, current is integrated over the one-second sampling period and stored in GUI_MEASURE.SOC, while measured battery power is integrated and stored in GUI_MEASURE.SOE. In this reference calculation no efficiency, capacity adaptation, OCV correction or other battery-state estimation mechanism is applied. Its principal purpose in the experimental setup is therefore to provide a transparent baseline against which the behaviour of the more advanced battery-monitoring method can be evaluated.

The reference one-second SOC/SOE integration and the call to the advanced battery monitor are shown in **Code excerpt 6**.

```
753 //Calculation of SOC and SOE, this interrupt is called every 1 second by theCPUTIMER1
754 __interrupt void INT_Calculation_SOC_SOE_ISR(void){
755
756     const float32_t sample_time_s = 1.0f;
757
758     if( (BatteryController_DISCHARGING_CC == BatteryController_state
759         || BatteryController_CHARGING_CV == BatteryController_state
```

```
760       || BatteryController_CHARGING_CC == BatteryController_state)
761       ){
762       if(struja_baterije_sensor_A_f > 0.02f || struja_baterije_sensor_A_f < -0.02f){
763           GUI_MEASURE.SOC += struja_baterije_sensor_A_f*1.0f/3600.0f;
764           GUI_MEASURE.SOE += GUI_MEASURE.snaga_baterije_W*1.0f/3600.0f;
765
766       }
767
768   }
769
770 #ifdef TEST_BATTERY_MONITORING
771
772    battery_supervising(
773    &battery_monitor,
774    sample_time_s,
775    current_test_battery_monitoring,
776    voltage_test_battery_monitoring,
777    BatteryController_state_test);
778
779 #else
780
781
782
783    battery_supervising(
784    &battery_monitor,
785    sample_time_s,
786    struja_baterije_sensor_A_f,
787    napon_baterije_sensor_v_f,
788    BatteryController_state);
```

Code excerpt 6 Reference one-second SOC/SOE integration and call to the advanced battery monitor.

2.1.7 Time-critical SOC calculation in battery_supervising()

The principal sampling-dependent SOC calculation is implemented in battery_supervising(), which is explicitly intended to be called from the interrupt routine. The function stores the latest battery voltage and current and calculates the incremental transferred charge, delta_Q, from the absolute measured current and the known sampling period. During CC or CV charging, this increment is accumulated in both Q_charged and Q_in_cycle, while SOC is increased using the charging-efficiency coefficient eta_c and the presently estimated battery capacity C_bat. During CC discharging, the same charge increment is

accumulated in Q_discharged and Q_out_cycle and is subtracted from SOC. The resulting SOC is limited to the physical interval from 0 to 100%, after which DOD is calculated as 100% minus SOC.

Placing these operations in the periodic interrupt is significant because coulomb counting is directly dependent on the accuracy of the integration interval. A deterministic one-second execution period reduces timing-related integration error and provides a consistent basis for comparison between experiments. The function therefore represents the time-critical core of the advanced SOC monitoring method, whereas slower correction and learning mechanisms are intentionally excluded from the interrupt.

The time-critical coulomb-counting and SOC update implemented in battery_supervising() are shown in **Code excerpt 7**.

```
75 void battery_supervising(
76     BatterySupervising * const bat_sup,
77     const float32_t period_s,
78     float32_t battery_current_A,
79     float32_t battery_voltage_V,
80     BatteryControllerState controller_state
81     ){
82
83
84         bat_sup->battery_voltage_V = battery_voltage_V;
85
86         bat_sup->battery_current_A = battery_current_A;
87
88         float32_t delta_Q = fabsf(battery_current_A) * period_s /3600.0f; //Ah
89
90         if(battery_current_A > 0.02f || battery_current_A < -0.02f){
91
92             if(     BatteryController_CHARGING_CC == controller_state
93                 ||  BatteryController_CHARGING_CV == controller_state
94             ){
95
96                 bat_sup->Q_charged += delta_Q;
97
98                 bat_sup->Q_in_cycle += delta_Q;
99
100                bat_sup->SOC += delta_Q * bat_sup->eta_c
101                 * 100.0f / bat_sup->C_bat;
102
103            }else if(BatteryController_DISCHARGING_CC == controller_state){
```

```
104
105             bat_sup->Q_discharged += delta_Q;
106
107             bat_sup->Q_out_cycle += delta_Q;
108
109             bat_sup->SOC -= delta_Q * 100.0f/bat_sup->C_bat;
110
111         }
112
113         //Zasicenje SOC-a
114         if(bat_sup->SOC > 100.0f)
115             bat_sup->SOC = 100.0f;
116         else if(bat_sup->SOC < 0.0f)
117             bat_sup->SOC = 0.0f;
118
119         bat_sup->DOD = 100.0f - bat_sup->SOC;
```

Code excerpt 7 Time-critical coulomb counting, efficiency correction, SOC saturation and DOD calculation.

2.1.8 Non-time-critical estimation and adaptation in battery_supervising_run()

The battery_supervising_run() function complements the interrupt-based calculation and is executed from the main program loop. Its operations are primarily state- and event-dependent rather than sampling-period-dependent. When a transition into the fully discharged state is detected and at least one preceding full-charge event has occurred, the accumulated charge quantities are used to estimate a measured battery capacity. The stored capacity C_bat is then updated gradually rather than being replaced instantaneously, thereby introducing a slow adaptation of the capacity used by the SOC calculation.

A transition into the fully charged state establishes a known SOC boundary by setting SOC to 100%. The same event is used to evaluate the relationship between charge transferred into and out of the battery and to update the charging-efficiency parameter. The cycle accumulators are subsequently reset. These event-based operations provide the link between the short-term coulomb-counting calculation and the longer-term adaptation of battery parameters.

The event-driven capacity-learning and charging-efficiency operations are summarized in **Code excerpt 8**.

```
220 void battery_supervising_run(BatterySupervising * const bat_sup, BatteryControllerState controller_state){
221
222
223     bool battery_discharged = BatteryController_DISCHARGED == controller_state;
224
225     /*
226         Detekcija potpuno ispraznjene baterije
```

```
227  */
228  //Kada je baterija potpuno ispraznjena tada se moze odrediti nova vrijednost
229  //kapaciteta i inicijalizirati SOC
230  if(!bat_sup->past_battery_discharged && battery_discharged ){
231
232      //Learning capacity of battery
233      if(bat_sup->least_one_time_fully_charged){
234
235          float32_t measured_capacity = bat_sup->Q_out_cycle - bat_sup->Q_in_cycle;
236
237          //Korekcija kapaciteta baterije
238          bat_sup->C_bat = 0.95f * bat_sup->C_bat + 0.05f * bat_sup->eta_c * measured_capacity;
239
240
241
242      }
243
244
245
246  }bat_sup->past_battery_discharged = battery_discharged;
247
248
249  bool battery_charged = BatteryController_CHARGED == controller_state;
250
251  /*
252      Detekcija potpuno pune baterije
253  */
254
255
256  if(!bat_sup->past_battery_charged && battery_charged){
257
258      bat_sup->least_one_time_fully_charged = true;
259
260      bat_sup->SOC=100.0f;
261
262      if(bat_sup->Q_in_cycle > 0.1f && bat_sup->Q_out_cycle > 0.1f){
263
```

```
264         bat_sup->eta_c = bat_sup->Q_out_cycle / bat_sup->Q_in_cycle;
265
266         bat_sup->eta_c = bat_sup->eta_c * 0.95f + bat_sup->eta_c * 0.05f;
267
268     }
269
270     //Resetiranje brojaca
271     bat_sup->Q_in_cycle = 0.0f;
272     bat_sup->Q_out_cycle = 0.0f;
273
274 }bat_sup->past_battery_charged = battery_charged;
```

Code excerpt 8 Event-based capacity learning and charging-efficiency update.

2.1.9 OCV-based SOC correction during battery rest

Coulomb-counting methods accumulate errors caused by sensor offset, parameter uncertainty and imperfect efficiency representation. The implemented algorithm therefore introduces an additional SOC correction when the battery remains in a non-power-producing state. The software identifies rest conditions through the CHARGED, DISCHARGED, RUN and INIT controller states. After a rest period of approximately 30 minutes, the measured battery voltage is interpreted through an OCV-SOC lookup characteristic. Linear interpolation between neighboring lookup points provides an OCV-derived SOC estimate, which is blended with the existing SOC estimate using a 95%/5% weighting. The correction is deliberately gradual so that the OCV estimate does not cause an abrupt discontinuity in the monitored SOC.

The OCV-based SOC correction after the prescribed rest interval is shown in **Code excerpt 9**.

```
278     Correction of SOC by using OCV(SOC) characteristics, correction 5% is applied if battery at
279     rest more than 30 min
280
281 */
282
283 if(       BatteryController_CHARGED == controller_state
284         || BatteryController_DISCHARGED == controller_state
285         || BatteryController_RUN == controller_state
286         || BatteryController_INIT == controller_state
287 ){
288
289     uint32_t time = millis();
290
291     //If time of 30 min passed than the rest_start is updated with newer value and SOC
292     //is corrected with 5% value of SOC according the characteristics OCV(SOC)
```

```
293     if(!bat_sup->ocv_corrected && time - bat_sup->rest_start > 1800000UL){
294
295         //bat_sup->ocv_corrected = true;
296         bat_sup->rest_start = millis();
297
298         float32_t soc_ocv = get_soc_from_ocv(bat_sup->battery_voltage_V);
299
300         bat_sup->SOC = bat_sup->SOC * 0.95f + soc_ocv * 0.05f;
301
302     }
303
304
305 }else{
306     //If battery is charged or dicharged, rest_start is updated continously, if battery is at rest,
307     //rest_start is updated and used to count time
308     bat_sup->rest_start = millis();
309     //bat_sup->ocv_corrected = false;
```

Code excerpt 9 **OCV-based SOC correction after the prescribed rest interval.**

2.1.10 SOH calculation and battery-state synchronization with the HMI

The estimated battery capacity is also used to calculate SOH as the ratio between the currently estimated capacity C_bat and the nominal capacity C_bat_n. The resulting SOC, SOE variable, charging-efficiency coefficient, estimated and nominal capacities, cycle-charge accumulators and cumulative charge quantities are transferred to the GUI measurement structure. The module further supports restoration of previously stored battery-state variables from the PC when the corresponding initialization command is received. This mechanism is important for long-duration experiments because the internal battery-state estimate can be preserved across interruptions of the PC connection or controller operation.

The SOH calculation and synchronization of battery-state variables with the HMI are shown in **Code excerpt 10**.

```
312 bat_sup->SOH = 100.0f * bat_sup->C_bat / bat_sup->C_bat_n;
313
314 //Azuriranje stanja baterije preko modbus-a na PC racunalu
315
316 GUI_MEASURE.battery_state.SOC_BAT_MONINTOR = bat_sup->SOC;
317 GUI_MEASURE.battery_state.SOE_BAT_MONITOR =  bat_sup->SOE;
318 GUI_MEASURE.battery_state.eta_c =  bat_sup->eta_c;
319 GUI_MEASURE.battery_state.C_bat =  bat_sup->C_bat;
320 GUI_MEASURE.battery_state.Q_in_cycle = bat_sup->Q_in_cycle;
```

```
321  GUI_MEASURE.battery_state.Q_out_cycle = bat_sup->Q_out_cycle;
322  GUI_MEASURE.battery_state.Q_charged = bat_sup->Q_charged;
323  GUI_MEASURE.battery_state.Q_discharged = bat_sup->Q_discharged;
324  GUI_MEASURE.battery_state.C_bat_n = bat_sup->C_bat_n;
325  GUI_STATUS.battery_state.bits.least_one_time_fully_charged = bat_sup->least_one_time_fully_charged;
326
327
328
329
330
331  //Azuriranje vrijednost s vrijednostima spremljeni na PC racunalu
332  //u trenutku iskljucenja konkecije s racunalom
333  //Na ovaj nacin je realizirano cuvanje zadnjih vrijednosti
334  //stanja baterije
335
336  connected2pc = is_modbus_busy();
337  if(connected2pc){
338
339     if(GUI_CONTROLS.control_bits.bits.init_state_of_battery){
340        bat_sup->SOC = GUI_REFERENCES.battery_state.SOC_BAT_MONINTOR;
341        bat_sup->SOE = GUI_REFERENCES.battery_state.SOE_BAT_MONITOR;
342        bat_sup->eta_c = GUI_REFERENCES.battery_state.eta_c;
343        bat_sup->C_bat = GUI_REFERENCES.battery_state.C_bat;
344        bat_sup->Q_in_cycle = GUI_REFERENCES.battery_state.Q_in_cycle;
345        bat_sup->Q_out_cycle = GUI_REFERENCES.battery_state.Q_out_cycle;
346        bat_sup->Q_charged  = GUI_REFERENCES.battery_state.Q_charged;
347        bat_sup->Q_discharged = GUI_REFERENCES.battery_state.Q_discharged;
348        bat_sup->C_bat_n = GUI_REFERENCES.battery_state.C_bat_n;
349        bat_sup->least_one_time_fully_charged = GUI_CONTROLS.battery_state.bits.least_one_time_fully_charged;
350
351        GUI_CONTROLS.control_bits.bits.init_state_of_battery = false;
```

Code excerpt 10 SOH calculation and transfer/restoration of battery-state variables through the HMI.

2.1.11 Interpretation of the SOC/SOE activity and state diagrams

Figure 4 presents the complete organization of the SOC monitoring and estimation procedure and, in particular, clarifies the separation between the time-critical calculations executed in battery_supervising() and the non-time-critical calculations executed in battery_supervising_run(). The blue part of the diagram represents the one-second interrupt path, whereas the green part represents the main-loop processing.

This separation is important because only calculations that depend directly on a precisely defined sampling interval are placed in the interrupt, while parameter adaptation, state-dependent corrections, and communication-related updates are performed outside the time-critical routine.

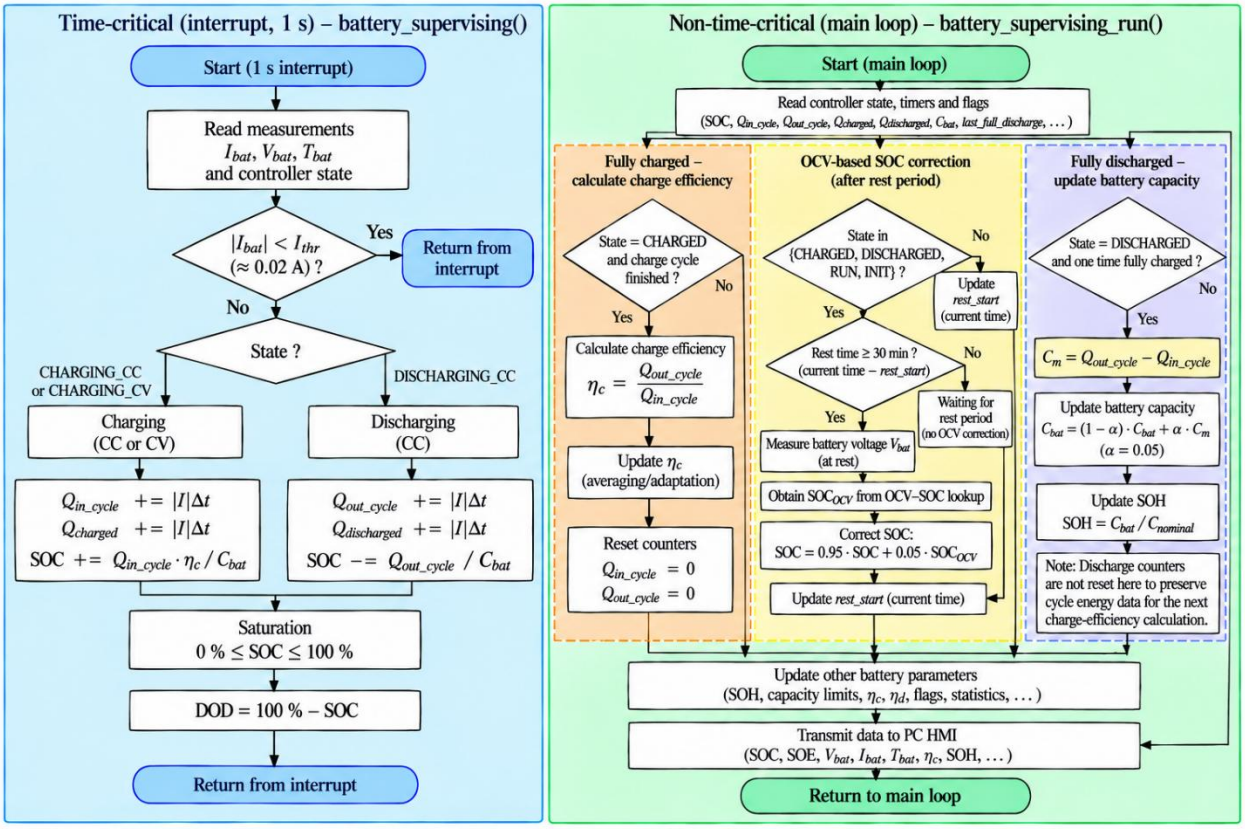


Figure 4 *Activity diagram of SOC calculation and estimation, showing the time-critical interrupt calculation and the non-time-critical capacity, efficiency, OCV-correction, SOH, and HMI-update tasks.*

The software structure of the battery-monitoring algorithm is presented in **Figure 4**. The left part represents the time-critical battery_supervising() routine executed by the one-second interrupt, whereas the right part represents battery_supervising_run(), which is executed in the main program loop. This organization separates calculations that depend directly on a deterministic sampling interval from event-driven estimation, adaptation, and synchronization tasks.

In the time-critical part, battery voltage, battery current, temperature, and the controller state are acquired at the defined sampling instant. When the magnitude of battery current exceeds the threshold, the algorithm identifies whether the battery is charging or discharging from the controller state. During charging, the transferred charge is accumulated in Q_in_cycle and Q_charged, and SOC is increased using the charging-efficiency coefficient and the current battery-capacity estimate. During discharging, Q_out_cycle and Q_discharged are accumulated and SOC is reduced accordingly. The resulting SOC is constrained to the physical interval from 0 to 100%, after which DOD is calculated.

The non-time-critical part contains three principal event-driven functions. The first is associated with a fully charged condition. When a charge cycle is completed, the charge efficiency is calculated from the charge transferred out of and into the battery according to η_c = Q_out_cycle / Q_in_cycle. The estimated efficiency can subsequently be adapted or averaged before it is used by the interrupt-based SOC calculation. After

completion of this processing, the cycle counters are reset, preparing the monitoring algorithm for the next complete cycle.

The middle branch of **Figure 4** represents the OCV-based SOC correction. OCV correction is permitted only while the controller is in a rest-compatible state, namely CHARGED, DISCHARGED, RUN, or INIT. If this condition is not fulfilled, the rest-start time is updated and no OCV correction is performed. If the battery remains in a permitted rest state for at least 30 minutes, the measured battery voltage is interpreted through the OCV-SOC lookup characteristic to obtain SOC_OCV. The SOC estimate is then corrected gradually according to SOC = 0.95·SOC + 0.05·SOC_OCV, limiting the influence of an individual voltage-based correction.

The right-hand branch represents battery-capacity learning associated with the fully discharged condition. The capacity update is performed only when the controller reaches DISCHARGED and at least one preceding fully charged condition has been registered. This requirement ensures that the accumulated cycle quantities correspond to an appropriate charge-discharge history before they are used for capacity identification.

For a valid fully discharged event, the measured capacity C_m is first determined from the difference between the accumulated discharge and charge cycle quantities, C_m = Q_out_cycle − Q_in_cycle. This measured value is not assigned directly as the new battery capacity. Instead, the stored capacity estimate is updated using a learning factor α = 0.05 according to C_bat = (1 − α)·C_bat + α·C_m. The 5% learning factor produces a gradual adaptation and reduces the sensitivity of the capacity estimate to measurement noise or variations in an individual cycle.

After the capacity estimate has been updated, SOH is calculated from the ratio between the estimated battery capacity and the nominal battery capacity. Importantly, the discharge-related counters are not reset in the fully discharged branch. Their accumulated information is retained so that it remains available for the subsequent fully charged event, where Q_out_cycle and Q_in_cycle are required for calculation of the charge-efficiency coefficient.

Finally, the estimated battery-state variables and parameters are synchronized with the remaining battery-monitoring data and transmitted to the PC HMI. Consequently, **Figure 4** illustrates that the implemented battery monitor combines deterministic coulomb counting with event-based efficiency estimation, capacity learning, OCV-based SOC correction, and SOH calculation while assigning each operation to an execution context appropriate to its timing requirements.

The state-dependent interaction between normal SOC estimation, rest-based OCV correction, boundary-event parameter updates, and fault handling is summarized in **Figure 5**. The diagram complements **Figure 4** by showing which supervisory conditions activate each battery-monitoring function rather than repeating the internal computational sequence.

Figure 5 is organized around NORMAL OPERATION, in which SOC is estimated by coulomb counting and the accumulated charge quantities, operating state, rest condition, boundary events, and fault conditions are monitored. START / INITIALIZATION enters this state after the battery parameters, estimator variables, counters, and flags have been initialized or restored from the PC. The arrows leaving NORMAL OPERATION therefore represent conditions that temporarily activate a specific correction, adaptation, or protection function; the return arrows indicate completion of that function and continuation of normal monitoring.

The blue branch represents rest-based OCV correction. A transition from NORMAL OPERATION to REST / OCV CORRECTION is permitted only when the controller is in a rest-compatible state (INIT, RUN, CHARGED, or DISCHARGED), the battery current is sufficiently small, and the required rest interval has

elapsed. The OCV-derived SOC value is then used for the gradual correction described previously. After the correction is completed, or when active operation resumes, the algorithm returns to NORMAL OPERATION. The arrow direction is important because rest detection triggers the correction; the correction itself does not create the rest condition.

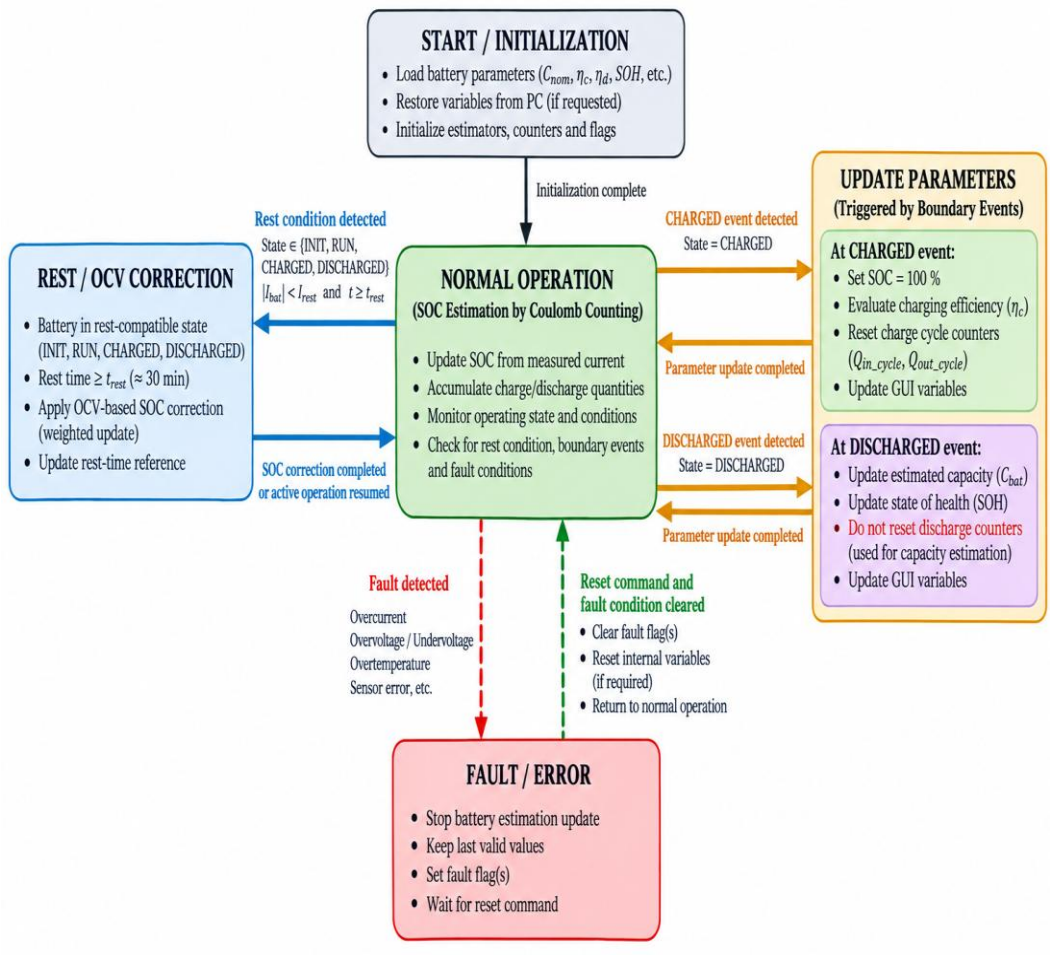


Figure 5 State-oriented representation of the battery-monitoring algorithm

The orange branch distinguishes the two boundary events because they perform different parameter updates. A CHARGED event sets SOC to 100%, evaluates the charging-efficiency coefficient from the accumulated cycle quantities, updates the GUI variables, and then resets both cycle counters Q_in_cycle and Q_out_cycle. The reset is therefore associated with the fully charged event, after the accumulated information has been used for the efficiency calculation.

A DISCHARGED event activates the capacity-learning branch only when the required preceding fully charged condition has been registered. At this boundary, the accumulated cycle quantities are used to update the estimated battery capacity, and the resulting capacity estimate is used to update SOH. In contrast to the CHARGED event, Q_in_cycle and Q_out_cycle are not reset at the DISCHARGED event. Their values are retained because they are still required when the subsequent CHARGED event is reached and the charging-efficiency calculation is performed.

The red and green dashed arrows describe fault handling separately from the normal estimation branches. Detection of an overcurrent, voltage-related abnormal condition, excessive temperature, sensor error, or

another relevant fault transfers the monitoring process from NORMAL OPERATION to FAULT / ERROR. The downward red arrow therefore represents fault activation. Return to NORMAL OPERATION is shown by the upward green arrow and is allowed only after the reset command has been issued and the fault condition has been cleared. This direction prevents the diagram from implying that fault detection itself returns the system to normal operation.

The figure consequently emphasizes the supervisory context of the battery monitor. Continuous coulomb counting remains the normal estimation process, while OCV correction is enabled by a validated rest condition, efficiency estimation and counter reset are associated with the CHARGED boundary, capacity and SOH adaptation are associated with the DISCHARGED boundary without resetting the cycle counters, and abnormal conditions suspend the normal sequence until recovery. This state-oriented view complements **Figure 4**. It explains where the calculations are executed, whereas **Figure 4** explains when the corresponding estimation and adaptation functions are activated.

2.1.12 HMI communication and experimental supervision

The program maintains dedicated GUI reference, control and measurement structures. The main loop continuously transfers measured temperature and controller state to the GUI measurement structure and calculates battery power from measured current and voltage. Conversely, the voltage reference, current or power command, maximum charge/discharge currents, cut-off voltage, termination-current threshold and protection parameters are obtained from the HMI-side data structures. This data organization provides a clear separation between real-time controller variables and externally adjustable experimental parameters.

UART communication is configured without the hardware FIFO because the implemented Modbus library was designed for non-FIFO operation. The Modbus communication therefore serves as the application-level link between the DSP and Python HMI. During normal experiments, this path supports parameter changes and measurement acquisition without recompiling the embedded program. During development and commissioning, Code Composer Studio can use the JTAG debugging path available through the shared laboratory interface for code loading, breakpoints, variable inspection and real-time debugging. These two functions should be regarded as complementary rather than as the same protocol. Modbus is used for the experimental HMI data exchange, whereas JTAG provides development and debugging access.

The principal GUI measurement updates and externally supplied controller references are illustrated in **Code excerpt 11**.

```
1072      //GUI azuriranje parametara
1073
1074      GUI_MEASURE.battery_temperature_c = battery_temperature_c = MAX6675_getCelsius(&battery_temp);
1075
1076      if(trigger_1Hz) MAX6675_read(&battery_temp);
1077
1078      GUI_MEASURE.state_of_controller = BatteryController_state;
1079
1080      GUI_MEASURE.snaga_baterije_W = GUI_MEASURE.struja_baterije_A * GUI_MEASURE.napon_baterije_V;
1081
1082      //SOC-SOE racunanje
1083      /*
1084      if( (BatteryController_DISCHARGING_CC == BatteryController_state
1085          || BatteryController_CHARGING_CV == BatteryController_state
1086          || BatteryController_CHARGING_CC == BatteryController_state)
1087          && trigger_5s){
1088          if(struja_baterije_sensor_A_f > 0.02f || struja_baterije_sensor_A_f < -0.02f){
1089              GUI_MEASURE.SOC += struja_baterije_sensor_A_f*5.0f/3600.0f;
1090              GUI_MEASURE.SOE += GUI_MEASURE.snaga_baterije_W*5.0f/3600.0f;
1091
```

```
        }
    }
    */
    //Resetiranje SOC-SOE
    if(GUI_CONTROLS.control_bits.bits.reset_soc_soe){
        GUI_CONTROLS.control_bits.bits.reset_soc_soe = 0;
        GUI_MEASURE.SOC = 0.0f;
        GUI_MEASURE.SOE = 0.0f;
    }


    rk = GUI_REFERENCES.voltage_battery_V_ref;
    yk = napon_baterije_sensor_v;
```

Code excerpt 11 GUI measurement update and controller references.

2.1.13 Local LED display and status indication

In addition to the PC HMI, the laboratory setup contains a local LED display that provides immediate observation of the principal battery electrical quantities. The filtered battery voltage and current are first transferred to GUI_MEASURE.napon_baterije_V and GUI_MEASURE.struja_baterije_A. Both floating-point values are then converted into binary-coded decimal data by float2bcd(). The battery voltage is written into one part of the eight-digit display buffer and the battery current into the remaining part. Consequently, the operator can observe the actual battery voltage and current locally during both charging and discharging without relying exclusively on the PC HMI.

The display is refreshed periodically together with the status-diode information. The code also calls turn_on_led() before write_led() at the 1 Hz trigger, which is intended to re-enable the display if electrical contact has temporarily been lost. A separate LED is used for driver-error indication, while the status-diodes variable communicates controller and protection conditions. The local display therefore has two experimental roles: quantitative visualization of battery voltage and current and immediate qualitative indication of operating or fault status.

The local presentation of measured battery voltage and current on the LED display is shown in **Code excerpt 12**.

```
#ifndef TEST_BATTERY

            GUI_MEASURE.napon_baterije_V =  napon_baterije_sensor_v_f;
            GUI_MEASURE.struja_baterije_A = struja_baterije_sensor_A_f;

#else
            if(trigger_1Hz){
                GUI_MEASURE.napon_baterije_V = test_battery_run(&tst_bat, current_ref_A);
                GUI_MEASURE.struja_baterije_A = current_ref_A;
            }
#endif



            float2bcd(bcd, GUI_MEASURE.napon_baterije_V, 2, 4);

            float2bcd(bcd, GUI_MEASURE.struja_baterije_A, 2, 0);
```

```
1706
1707
1708            //LED4 se koristi za prikaz je li error drivera
1709            //Ako je uključena tada je error inace je iskljucena
1710            error_driver = !GPIO_readPin(_ERR_DRIVER);
1711            GPIO_writePin(LED4, !error_driver);
1712
1713            if(trigger_1Hz){
1714                //Ponovno se ukljucuje display u slucaju ako je nestao kontakt tada ce opet
1715                //uspostavom kontakta nastaviti svijetliti i prikazivati vrijednosti
1716                turn_on_led();
1717                write_led(bcd, status_diodes);
1718            }
```

Code excerpt 12 Local LED display of measured battery voltage and current.

2.1.14 Interpretation of the complete control sequence

The complete software sequence can be interpreted as a closed experimental loop. The operator defines an operating point through the HMI; the supervisory state machine determines the permitted converter mode; the low-level interrupt acquires the electrical variables and computes the required PWM duty cycle; the power stage transfers energy to or from the battery; and the resulting voltage, current, power and temperature are returned to the monitoring functions. At a slower rate, battery-state quantities are integrated and made available to the HMI. Protection functions continuously constrain this loop and can force the converter into a safe state whenever electrical, thermal, communication or timing conditions are violated.

This architecture is particularly suitable for battery-model validation and microgrid-oriented studies because the hardware power path is retained while the control behaviour remains software configurable. Current-controller parameters, voltage-controller parameters, charging/discharging references, CC-CV transition logic, termination criteria, cut-off limits, protection thresholds and battery-state algorithms can be modified or replaced without changing the general laboratory topology. The same setup can therefore be used to investigate how different supervisory algorithms and battery models influence measured energy exchange, charging efficiency, SOC/SOE evolution and, ultimately, the accuracy of higher-level economic battery models.

2.2 Human-Machine Interface for Control, Monitoring, and Experimental Automation of the Battery System

This section describes the Python-based human-machine interface (HMI) developed for the DSP-based battery charger/discharger laboratory setup. The description is based on the implemented hmi.py application and focuses on its supervisory role, software architecture, DSP communication, experimental automation, real-time visualization, data recording, and interaction with the embedded battery-controller state machine.

2.2.1 General purpose and role of the HMI

A dedicated HMI was developed in Python to provide supervisory control, monitoring, visualization, and experimental automation of the DSP-based battery charger/discharger. The application represents the highest software layer of the laboratory setup and establishes the connection between the operator and the real-time control algorithms implemented on the DSP. Its primary purpose is not to replace the real-time controller, but to provide a flexible environment through which operating conditions can be defined, experiments can be automated, internal battery variables can be monitored, and measured data can be stored for subsequent analysis.

This distinction is particularly important for the proposed laboratory setup. The time-critical converter control and battery charging/discharging state logic remain implemented on the DSP, whereas the PC application performs supervisory functions. The HMI provides access to current and voltage references, charging and discharging mode, maximum current limits, constant-voltage termination current, and selection between current- and power-based references. Simultaneously, the application receives battery voltage, current, power, state of charge (SOC), state of energy (SOE), battery temperature, and the current state of the DSP controller.

Such separation between the supervisory and real-time control levels is beneficial for experimental battery research. Modification of experimental scenarios does not require modification of the fundamental HMI structure or the time-critical execution of the DSP controller. Different charging/discharging profiles can therefore be applied while maintaining deterministic execution of the converter control algorithms on the dedicated embedded platform.

The overall position of the HMI within the battery laboratory setup and the separation between the PC-based supervisory layer and the DSP-based real-time control layer are illustrated in **Figure 6**.

Figure 6 presents the functional position of the Python HMI within the complete laboratory setup. The operator interacts with the PC application to define references, operating modes and experimental sequences. The HMI exchanges commands and measured quantities with the DSP through the serial Modbus interface, while the DSP remains responsible for the deterministic control of the DC/DC converter and the battery charging/discharging process. The figure therefore emphasizes the separation between supervisory PC functions and real-time embedded control.

2.2.2 Software architecture

The HMI was implemented using the Tkinter graphical user-interface library, while Matplotlib is embedded into the application for real-time visualization. Serial communication is implemented using the PyModbus library, and the communication and data-acquisition process is separated from the main graphical interface through a dedicated Python thread. A thread-safe queue transfers acquired measurements from the communication thread to the GUI.

Conceptually, the software can be divided into four interacting functional layers: the graphical interface, supervisory application logic, communication and acquisition, and visualization/data-storage functions. This organization is important because serial communication can involve waiting periods that would otherwise block the graphical event loop. By executing hardware communication in a dedicated thread, the application can continuously communicate with the DSP while maintaining an interactive user interface.

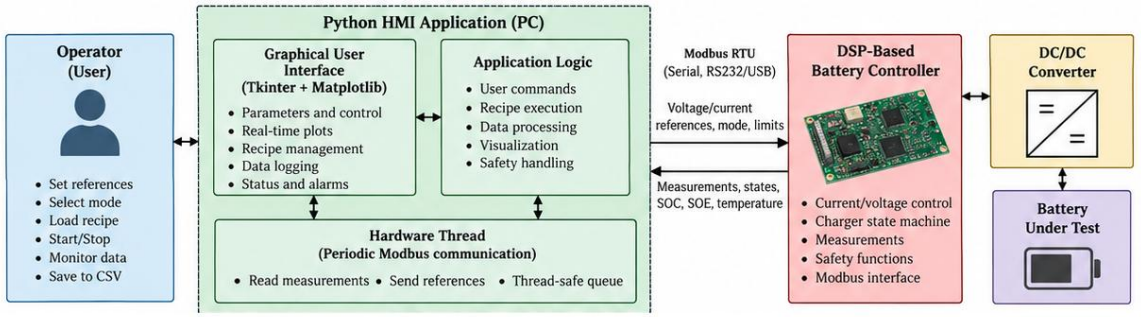


Figure 6 Conceptual architecture of the Python HMI and its integration with the DSP-based battery charger/discharger. The implemented serial protocol is Modbus ASCII.

The HardwareThread periodically reads measurements from the DSP, transmits modified reference quantities, and places newly acquired samples into a thread-safe queue. The main Tkinter thread subsequently retrieves the samples from this queue, updates numerical indicators, stores historical values, records selected data to CSV, evaluates the reported DSP state, and refreshes the plots. This architecture provides an important separation between data acquisition and data presentation, so the communication process is not directly dependent on the time required for graphical rendering.

The thread-safe queue enables reliable data transfer between the hardware communication thread and the graphical interface. It prevents synchronization problems by transferring complete measurement packets between the two independently running processes. This separation also prevents communication delays from directly blocking GUI operation.

The implemented graphical interface used for parameter definition, DSP monitoring, experimental control, and real-time visualization is shown in **Figure 7**.It shows the implemented Battery Cycler / Scope graphical interface during an actual operating condition. The upper part of the window contains the serial connection controls and the principal experimental setpoints, including the voltage and current references, maximum charging and discharging currents, cutoff voltage, CV termination current and power reference. Charge/discharge mode and current/power reference selection are available directly from the same interface. The Actual section provides immediate feedback from the DSP, including battery voltage, current, power, temperature, SOC, SOE and the active controller state. In the operating point shown, the DSP reports the CHARGING_CV state, illustrating that the HMI also exposes the internal supervisory state of the embedded charger controller.

The lower part of **Figure 7** acts as a continuously updated experimental scope. The three traces display battery voltage, battery current and SOC over the selected time window. The figure illustrates how transient changes in current and voltage can be observed together with the corresponding evolution of SOC. Controls located between the numerical indicators and plots provide Run/Stop operation, recipe execution, sampling-rate and time-window selection, CSV recording and SOC/SOE reset. Consequently, a single interface combines experiment configuration, execution, monitoring and data acquisition.

The corresponding Python implementation of the software dependencies, communication thread, and principal HardwareThread initialization is shown in **Code excerpt 13**.

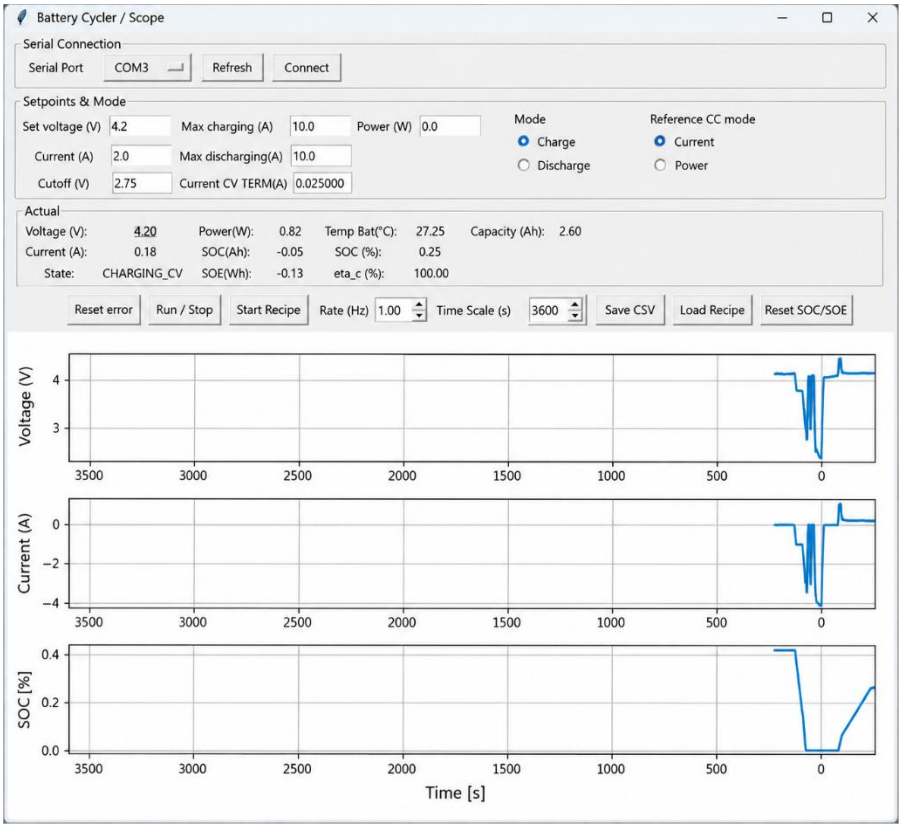


Figure 7 Main window of the Python HMI application for battery charger/discharger control and monitoring.

hmi.py — lines 12–76

```
12  import tkinter as tk
13  from tkinter import filedialog, messagebox
14  import threading
15  import time
16  import csv
17  import queue
18  import random
19
20  import matplotlib.pyplot as plt
21  from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
22
23
24  import struct
25
26  import sys
27
28  from pymodbus.client import (
29      ModbusSerialClient
30  )
31  from pymodbus.datastore import ModbusSequentialDataBlock
32  from pymodbus.datastore import ModbusServerContext
33
34  from pymodbus import FramerType
35
36  import serial.tools.list_ports
37
38
39  from enum import Enum
40
41  def get_serial_ports():
42      ports = serial.tools.list_ports.comports()
43      return [p.device for p in ports]
```

```
 44
 45
 46
 47  def toFloat(x):
 48
 49
 50
 51      byte = x.to_bytes(length=4, byteorder="big")
 52
 53      float_val = struct.unpack(">f", byte)[0]
 54      return float_val
 55
 56  # ========================
 57  # HARDWARE / CONTROL THREAD
 58  # ========================
 59  class HardwareThread(threading.Thread):
 60      def __init__(self, q, sample_rate, power_enabled, mode,
 61                                  set_v, set_c, set_max_discharging_i, set_max_charging_i , set_cv_current_end ,
limits, modbus_client):
 62          super().__init__(daemon=True)
 63          self.q = q
 64          self.sample_rate = sample_rate
 65          self.power_enabled = power_enabled
 66          self.mode = mode
 67          self.set_v = set_v
 68          self.set_c = set_c
 69          self.limits = limits
 70          self.set_max_charging_i = set_max_charging_i
 71          self.set_max_discharging_i = set_max_discharging_i
 72          self.set_cv_current_end = set_cv_current_end
 73
 74          self.sim_voltage = 11.5
 75          self.sim_current = 0.0
 76          self.modbus_client = modbus_client
```

Code excerpt

Code excerpt 13 Imported libraries and the HardwareThread structure used for communication and acquisition. Source lines 12-76 of hmi.py.

Code excerpt 13 presents the basic software structure used to separate the graphical interface from communication with the DSP. The imported modules provide the Tkinter GUI, Matplotlib visualization, serial-port access, Modbus communication, threading and thread-safe data exchange. The HardwareThread class is created as a daemon thread and receives access to the sampling rate, operating mode, voltage and current references, current limits, CV termination current and Modbus client. The register addresses used for references and measured quantities are also defined in this part of the program.

The use of a dedicated hardware thread is important because Modbus communication can be performed independently of the Tkinter event loop. Acquired data are transferred to the GUI through a queue rather than by directly modifying graphical objects from the communication thread. This provides a clear separation between communication/acquisition and visualization functions.

2.2.3 Establishment of communication with the DSP

The application automatically obtains the available serial communication ports and allows the operator to select the appropriate interface. After the connection command is issued, a Modbus serial client is configured using the selected serial port, a baud rate of 9600 bit/s, a communication timeout of 0.1 s, and the Modbus ASCII framer.

A successful connection initiates a predefined safe startup sequence. The HMI first sends a command that places the charger/discharger in its disabled condition. Only after this action is the hardware communication thread created and started. The current-based reference is subsequently selected as the initial reference mode. Establishment of PC-DSP communication therefore does not automatically energize the converter; connection and converter activation are treated as separate operations.

The implementation of serial-port selection, Modbus ASCII connection establishment, safe converter initialization, and acquisition-thread startup is shown in Code excerpt 2.

hmi.py – lines 249–303

```
249     def refresh_ports(self):
250       self.available_ports = get_serial_ports()
251       menu = self.port_menu["menu"]
252       menu.delete(0, "end")
253 
254       for port in self.available_ports:
255           menu.add_command(
256               label=port,
257               command=lambda p=port: self.selected_port.set(p)
258           )
259 
260       if self.available_ports:
261           self.selected_port.set(self.available_ports[0])
262 
263 
264 
265 
266 
267     def connect_modbus(self):
268         if not self.selected_port.get():
269             messagebox.showerror("Error", "No serial port selected")
270             return
271 
272         # Close previous connection if any
273         try:
274             self.hw.stop()
275             self.modbus_client.close()
276         except:
277             pass
278 
279         self.modbus_client = ModbusSerialClient(
280             framer=FramerType.ASCII,
281             port=self.selected_port.get(),
282             baudrate=9600,
283             timeout=0.1
284         )
285 
286         if self.modbus_client.connect():
287             messagebox.showinfo("Connected", f"Connected to {self.selected_port.get()}")
288             # Turn off charger/discharger safely
289             self.modbus_client.write_coils(address=1, values=[True], device_id=0xaf)
290             self.hw = HardwareThread(
291                         self.data_queue,
292                         sample_rate=lambda: self.sample_rate.get(),
293                         power_enabled=self.power_enabled,
294                         mode=lambda: self.mode.get(),
295                         set_v=lambda: self.voltage_ref,
296                         set_c=lambda: self.current_ref,
297                         set_max_charging_i=lambda: self.maximum_charging_current_val,
298                         set_max_discharging_i=lambda: self.maximum_discharging_current_val,
299                         set_cv_current_end = lambda: self.cv_current_end_val,
300                         limits=self.limits,
301                         modbus_client = self.modbus_client
302                     )
303             self.hw.start()
```

Code excerpt 14 Serial-port handling, Modbus ASCII client configuration, safe connection sequence, and hardware-thread startup. Source lines 249-303 of hmi.py.

Code excerpt 14 shows how communication with the DSP is established. The application first detects available serial ports and allows the operator to select the required interface. When Connect is selected, a Modbus serial client is configured with the selected port, a baud rate of 9600 bit/s, a 0.1 s timeout and the Modbus ASCII framer.

After successful connection, the HMI initially sends the command that places the charger/discharger in the disabled condition. A HardwareThread is then created and supplied with callable access to the current GUI parameters. Consequently, the communication thread always obtains the most recent setpoints without directly controlling the graphical widgets. This initialization sequence ensures that communication is established before active converter operation is requested.

2.2.4 Bidirectional exchange of experimental variables

Communication between the PC and DSP is bidirectional. In the DSP-to-HMI direction, the application periodically obtains battery temperature, battery power, SOC, SOE, controller state, battery current, and battery voltage. Each floating-point quantity is reconstructed from two 16-bit Modbus registers.

In the opposite direction, the HMI transfers experimental references and selected operating parameters to the DSP. The principal continuously managed quantities include the current reference, voltage reference, constant-voltage termination current, and maximum charging current. The application converts Python floating-point values into the corresponding 32-bit binary representation and separates them into two 16-bit words before transmission.

References are generally retransmitted only when their value has changed, with an initial transmission being forced when communication begins. This reduces unnecessary Modbus transactions while ensuring initialization of the DSP references. The sign of the current reference is determined according to the selected operating mode: a positive current magnitude entered through the HMI is converted to a negative current reference in discharge mode. This enables a consistent operator-facing sign convention while the software applies the convention required by the DSP.

The implemented bidirectional data exchange, including DSP measurement acquisition and transmission of battery-control references, is presented in **Code excerpt 15**.

hmi.py – lines 129–215

```
129          #Read temp of battery
130                result  =  self.modbus_client.read_input_registers(address=self.ADDRESS_TEMP_BAT  ,  count=2,
device_id=0xaf)
131          val = result.registers[0] | result.registers[1]<<16;
132          temp_bat = toFloat(val)
133
134          #Reading power of controller
135           result = self.modbus_client.read_input_registers(address=self.ADDRESS_MEASURE_POWER_BAT , count=2,
device_id=0xaf)
136          val = result.registers[0] | result.registers[1]<<16;
137          actual_power = toFloat(val)
138
139          #Reading SOC
140            result = self.modbus_client.read_input_registers(address=self.ADDRESS_MEASURE_SOC_BAT , count=2,
device_id=0xaf)
141          val = result.registers[0] | result.registers[1]<<16;
142          SOC = toFloat(val)
143
144          #Reading SOE
145            result = self.modbus_client.read_input_registers(address=self.ADDRESS_MEASURE_SOE_BAT , count=2,
device_id=0xaf)
146          val = result.registers[0] | result.registers[1]<<16;
147          SOE = toFloat(val)
148
149          #Reading state of controller
150            result = self.modbus_client.read_input_registers(address=self.ADDRESS_STATE_CONTROLLER, count=2,
device_id=0xaf)
151          state_of_controller = result.registers[0] | result.registers[1]<<16;
152
153
154            #print("run")
155          result = self.modbus_client.read_input_registers(address=self.ADDRESS_MEASURE_CURRENT_BAT, count=2,
device_id=0xaf)
156          val = result.registers[0] | result.registers[1]<<16;
157          c = toFloat(val)
158
159            #Read measured voltage of battery
160         result = self.modbus_client.read_input_registers(address=self.ADDRESS_MEASURE_VOLTAGE_BAT , count=2,
device_id=0xaf)
161          val = result.registers[0] | result.registers[1]<<16;
162          v = toFloat(val)
163
164
165            #Get reference value of current from field setpoints and mode
166          ref_current_controller = self.set_c()
167          if self.mode() == "DISCHARGE":
168            ref_current_controller = -ref_current_controller
169
170
171            #Only send new value to charger/discharger DSP
172          if self.ref_current_controller_past_val != ref_current_controller or self.firstTime:
173            #Get bytes of float
174            bytes_of_float = struct.pack(">f", ref_current_controller)
175            #Write this bytes in integer format
176            ref_current_integer = struct.unpack(">I", bytes_of_float)[0]
177            ref_current_lsw = ref_current_integer & 0xffff
178            ref_current_msw = ref_current_integer>>16 & 0xffff
179            #Sending this value to charger/discharger DSP
```

```
180                         self.modbus_client.write_registers(address=self.ADDRESS_REF_CURRENT,  device_id=0xaf,
values=(ref_current_lsw,ref_current_msw))
181           self.ref_current_controller_past_val = ref_current_controller
182
183           ref_voltage_controller = self.set_v()
184           if self.ref_voltage_controller_past_val != ref_voltage_controller or self.firstTime:
185             #Get bytes of float
186             bytes_of_float = struct.pack(">f", ref_voltage_controller)
187             #Write this bytes in integer format
188             ref_voltage_integer = struct.unpack(">I", bytes_of_float)[0]
189             ref_voltage_lsw = ref_voltage_integer & 0xffff
190             ref_voltage_msw = ref_voltage_integer>>16 & 0xffff
191             #Sending this value to charger/discharger DSP
192                         self.modbus_client.write_registers(address=self.ADDRESS_REF_VOLTAGE,  device_id=0xaf,
values=(ref_voltage_lsw,ref_voltage_msw))
193           self.ref_voltage_controller_past_val = ref_voltage_controller
194   #Novo
195           cv_current_end = self.set_cv_current_end()
196           if self.cv_current_end_past_val != cv_current_end or self.firstTime:
197             #Get bytes of float
198             bytes_of_float = struct.pack(">f", cv_current_end)
199             #Write this bytes in integer format
200             cv_current_end_integer = struct.unpack(">I", bytes_of_float)[0]
201             cv_current_end_lsw = cv_current_end_integer & 0xffff
202             cv_current_end_msw = cv_current_end_integer>>16 & 0xffff
203             #Sending this value to charger/discharger DSP
204                       self.modbus_client.write_registers(address=self.ADDRESS_CV_CURRENT_END,  device_id=0xaf,
values=(cv_current_end_lsw,cv_current_end_msw))
205           self.cv_current_end_past_val = cv_current_end
206
207
208           maximum_charging_current = self.set_max_charging_i()
209           if self.maximum_charging_current_past_val != maximum_charging_current:
210             bytes_of_float = struct.pack(">f", maximum_charging_current)
211             maximum_charging_current = struct.unpack(">I", bytes_of_float)[0]
212             maximum_charging_current_lsw = maximum_charging_current & 0xffff
213             maximum_charging_current_msw = maximum_charging_current>>16 & 0xffff
214             self.modbus_client.write_registers(address=self.ADDRESS_MAXIMUM_CHARGING_CURRENT, device_id=0xaf,
values=(maximum_charging_current_lsw,maximum_charging_current_msw))
215           self.maximum_charging_current_past_val = maximum_charging_current
```

Code excerpt 15 Periodic acquisition of DSP measurements and transmission of current, voltage, CV termination, and charging-current references. Source lines 129-215 of hmi.py.

Code excerpt 15 contains the central bidirectional communication cycle between the HMI and DSP. The hardware thread reads battery temperature, power, SOC, SOE, controller state, current and voltage from predefined DSP registers. Floating-point quantities are transmitted through two 16-bit Modbus registers; therefore, the received register words are combined and converted back into floating-point form before being passed to the GUI.

The same code transmits current, voltage, CV termination current and charging-current parameters to the DSP. Before transmission, each floating-point reference is converted into its 32-bit representation and separated into two 16-bit words. The current reference is automatically assigned a negative sign in discharge mode, allowing the operator to enter current as a positive magnitude. Previous values are stored so that references are normally retransmitted only when they change.

At the end of each acquisition cycle, the timestamp, measured variables and DSP controller state are placed into the thread-safe queue. The thread then waits according to the selected sampling rate. This code therefore represents the main communication bridge between the embedded real-time controller and the supervisory PC application.

2.2.5 Definition of experimental operating conditions

The graphical interface provides direct access to the parameters required for battery experiments. The operator can specify the voltage reference, current reference, cutoff voltage, maximum charging current, maximum discharging current, CV termination current, and power reference. Charge or discharge operation can be selected independently. The HMI additionally permits selection of either current or power as the reference quantity associated with the corresponding DSP control mode.

Input processing is used to maintain a consistent sign convention. Negative values entered for quantities representing magnitudes are converted to their absolute values, while the appropriate sign is applied during communication with the DSP. Selected parameters, including cutoff voltage and current limits, are written to the DSP and can be read back from the corresponding holding registers. This makes the HMI suitable for repeated experimental characterization because the operating point can be modified without recompiling the DSP program.

The Python implementation of the HMI fields used to define the principal experimental operating conditions is shown in **Code excerpt 16**.

hmi.py — lines 424–479

```
424          top = tk.LabelFrame(root, text="Setpoints & Mode")
425          top.pack(fill='x', padx=10, pady=4)
426
427          tk.Label(top, text="Set voltage (V)").grid(row=0, column=0)
428          self.voltage_entry = tk.Entry(top, textvariable=self.set_voltage, width=8)
429          self.voltage_entry.grid(row=0, column=1)
430          self.voltage_entry.bind("<Return>", self.check_number)
431
432
433          tk.Label(top, text="Current (A)").grid(row=1, column=0)
434          self.current_entry = tk.Entry(top, textvariable=self.set_current, width=8)
435          self.current_entry.grid(row=1, column=1)
436          self.current_entry.bind("<Return>", self.check_number)
437                           #tk.Entry(top,  textvariable=self.set_current,  width=8,  validate="focusout",
validatecommand=self.set_c).grid(row=1, column=1)
438
439          # Mode selection aligned
440          mode_frame = tk.Frame(top)
441          mode_frame.grid(row=0, column=6, rowspan=2, padx=20, sticky="n")
442
443          tk.Label(mode_frame, text="Mode").pack(anchor="w")
444                  tk.Radiobutton(mode_frame,  text="Charge",  variable=self.mode,  value="CHARGE",  command  =
self.toggle_mode).pack(anchor="w")
445              tk.Radiobutton(mode_frame, text="Discharge", variable=self.mode, value="DISCHARGE", command =
self.toggle_mode).pack(anchor="w")
446
447          # Mode selection aligned
448          current_power_select_frame = tk.Frame(top)
449          current_power_select_frame.grid(row=0, column=8, rowspan=2, padx=20, sticky="n")
450
451          tk.Label(current_power_select_frame, text="Reference CC mode").pack(anchor="w")
452                 tk.Radiobutton(current_power_select_frame,  text="Current",  variable=self.reference_cc_mode,
value="CURRENT_CC_REF", command=self.toggle_reference_cc).pack(anchor="w")
453                  tk.Radiobutton(current_power_select_frame,  text="Power",  variable=self.reference_cc_mode,
value="POWER_CC_REF", command=self.toggle_reference_cc).pack(anchor="w")
454
455
456          tk.Label(top, text="Cutoff (V)").grid(row=2,column=0)
457          self.cutoff_voltage_entry = tk.Entry(top, textvariable=self.cutoff_voltage, width=8)
458          self.cutoff_voltage_entry.grid(row = 2, column = 1)
459          self.cutoff_voltage_entry.bind("<Return>",self.check_number)
460
461          tk.Label(top, text= "Max charging (A)").grid(row=0, column=2)
462          self.max_charge_current_entry =tk.Entry(top, textvariable=self.maximum_charging_current, width=8)
463          self.max_charge_current_entry.grid(row=0, column=3)
464          self.max_charge_current_entry.bind("<Return>", self.check_number)
465
466          tk.Label(top, text= "Max discharging(A)").grid(row=1, column=2)
467              self.max_discharge_current_entry =tk.Entry(top, textvariable=self.maximum_discharging_current,
width=8)
468          self.max_discharge_current_entry.grid(row=1, column=3)
469          self.max_discharge_current_entry.bind("<Return>", self.check_number)
470
471          tk.Label(top, text= "Current CV TERM(A)").grid(row=2, column=2)
472          self.cv_current_end_entry =tk.Entry(top, textvariable=self.cv_current_end, width=8)
473          self.cv_current_end_entry.grid(row=2, column=3)
474          self.cv_current_end_entry.bind("<Return>", self.check_number)
475
476          tk.Label(top, text= "Power (W)").grid(row=0, column=4)
477          self.power_ref_entry =tk.Entry(top, textvariable=self.power_ref, width=8)
478          self.power_ref_entry.grid(row=0, column=5)
479          self.power_ref_entry.bind("<Return>", self.check_number)
```

Code excerpt 16 HMI fields for voltage/current references, charge-discharge mode, current/power reference selection, limits, and CV termination current. Source lines 424-479 of hmi.py.

Code excerpt 16 shows the implementation of the principal experimental setpoints in the graphical interface. Entry fields are connected to variables representing voltage, current, cutoff voltage, maximum charging and discharging currents, CV termination current and power reference. These values define the operating conditions that are subsequently transferred to the DSP.

The same part of the interface provides radio buttons for charge/discharge selection and for choosing whether the CC operating reference is based on current or power. Callback functions are associated with these selections so that changes made by the operator can be communicated to the DSP. This organization allows the experimental operating point to be modified without changing or recompiling the embedded control program.

2.2.6 Manual operation of the charger/discharger

The HMI provides a Run/Stop control for manual activation of the converter. Internally, the application maintains a power_enabled event representing the requested operating condition. When Run is activated, the event is set and a Modbus coil command is transmitted to place the converter in its running condition. When Stop is selected, the event is cleared and the corresponding command returns the converter to the disabled/initial condition.

Manual operation therefore follows the supervisory sequence: parameter definition -> mode selection -> converter enable -> DSP-controlled battery operation -> converter disable. The HMI defines what the system should do, whereas the DSP determines the real-time execution of the converter and its charging/discharging state transitions.

The relationship between manual operation, automated recipe execution, and the disabled operating condition of the HMI is summarized in **Figure 8**.

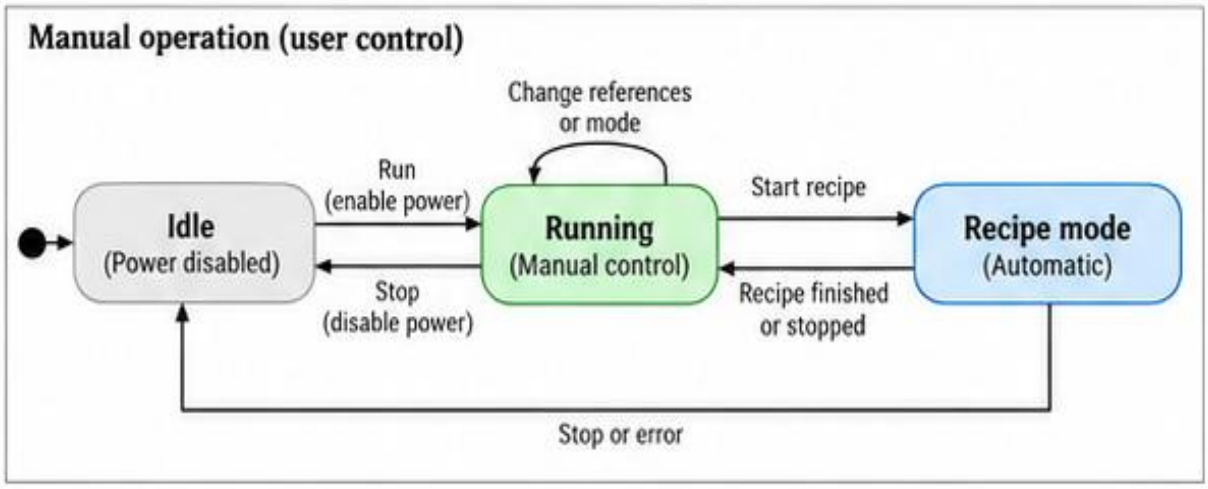


Figure 8 Conceptual supervisory operating-state diagram of the HMI for manual and recipe-based operation.

Figure 8 summarizes the supervisory operating modes of the HMI. In the idle condition the converter is disabled, while manual Run operation enables direct execution using the references selected by the operator. Recipe mode introduces an automated supervisory path in which the references and operating mode are changed according to the loaded time-dependent sequence. Stop commands or reported error conditions return the system toward a disabled condition. The diagram is intended to describe the PC-side supervisory logic rather than the high-frequency control states executed inside the DSP.

The implementation of manual Run/Stop operation and the corresponding converter enable and disable commands is shown in **Code excerpt 17**.

hmi.py – lines 851–882

```
851      def turn_on_converter(self):
852          self.modbus_client.write_coils(address=0, values=[True], device_id=0xaf)
853
854
855      '''
856        Iskljucenje pretvaraca, tj. dovodenje ga u stanje Init
857      '''
858      def turn_off_converter(self):
859          self.modbus_client.write_coils(address=1, values=[True], device_id=0xaf)
860
861
862
863      # =========================
864      # CONTROLS
865      # =========================
866      def toggle_power(self):
867        if self.recipe_running:
868            self.run_var.set(0)
869        else:
870
871            if not self.power_enabled.is_set():
872                self.power_enabled.set()
873                #self.modbus_client.write_registers(address=0, values=[1], device_id=0xaf)
874                print("Turn on converter")
875                self.turn_on_converter()
876                self.run_var.set(1)
877
878            else:
879                print("Turn off converter")
880                self.power_enabled.clear()
881                self.turn_off_converter()
882                self.run_var.set(0)
```

Code excerpt 17 Converter Run/Stop commands and manual power-enable logic. Source lines 851-882 of hmi.py.

Code excerpt 17 implements the manual Run/Stop functionality. The functions turn_on_converter and turn_off_converter encapsulate the Modbus coil commands used to request the DSP Run and disabled/Init conditions. The toggle_power function connects these commands to the graphical Run/Stop button and simultaneously updates the local power_enabled event.

The function also checks whether recipe execution is active. When an automated recipe is running, manual Run operation is prevented. This avoids simultaneous manual and automatic control of the converter enable command and establishes a simple supervisory priority between the two operating modes.

2.2.7 DSP controller-state supervision

An important feature of the HMI is its direct visualization of the internal state of the DSP battery controller. The implemented state table contains INIT, RUN, CHARGING_CC, CHARGING_CV, DISCHARGING_CC, CHARGED, DISCHARGED, OVERCURRENT_CHARGE, and OVERCURRENT_DISCHARGE.

The numerical controller state received through Modbus is converted into the corresponding textual state and presented to the operator. This provides considerably more experimental information than simple observation of voltage and current because the researcher can correlate measured quantities with the actual operating state of the embedded controller.

During a charging experiment, for example, measured current and voltage trajectories can be interpreted together with the transition from CHARGING_CC to CHARGING_CV, while completion of charging can be associated with the CHARGED state. This capability is particularly relevant when validating battery models that incorporate supervisory charging logic. The Python HMI monitors these DSP states; it does not implement the CC-CV transition itself. The DSP therefore remains the real-time control platform, while the PC application remains the experimental supervisory platform.

The DSP battery-controller states monitored by the HMI and their relationship to charging, discharging, completed-operation, and overcurrent conditions are illustrated in **Figure 9**.

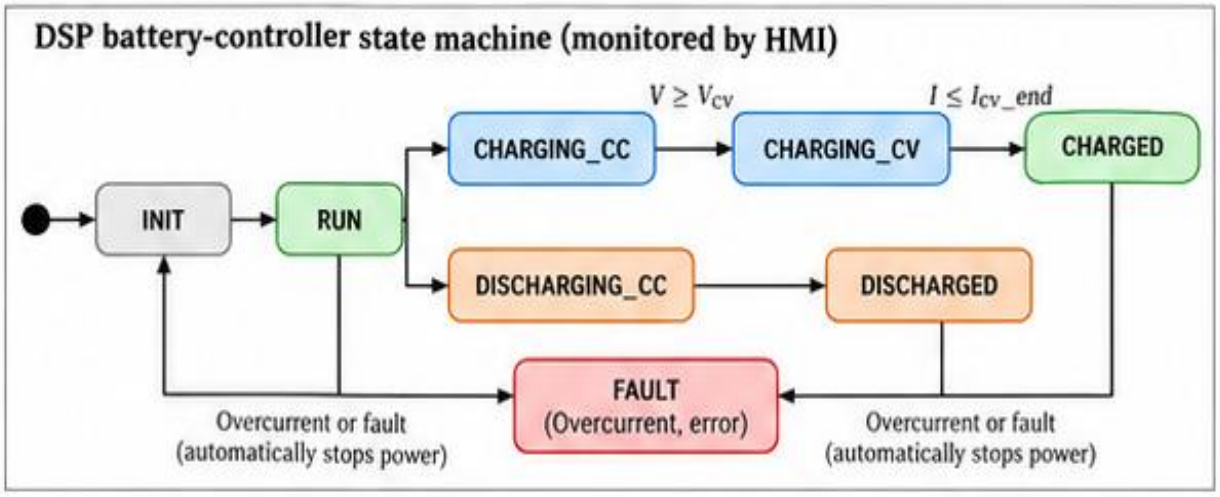


Figure 9 Conceptual representation of DSP battery-controller states monitored by the HMI. Transition conditions shown schematically should be interpreted together with the implemented DSP control algorithm.

Figure 9 presents the principal DSP battery-controller states that are monitored and displayed by the HMI. The application maps the numerical state received through Modbus to descriptive states such as INIT, RUN, CHARGING_CC, CHARGING_CV, DISCHARGING_CC, CHARGED and DISCHARGED. Overcurrent charging and discharging states are additionally used by the HMI to initiate a supervisory shutdown response. The figure therefore links continuous electrical measurements with the discrete operating state of the embedded battery controller.

The mapping of numerical DSP states to descriptive controller states, together with GUI data processing and the overcurrent response, is shown in **Code excerpt 18**.

hmi.py – lines 750–805

```
750      BatteryController_state_table = [   'INIT'              ,
751                                          'RUN'               ,
752                                          'CHARGING_CC'       ,
753                                          'CHARGING_CV'       ,
754                                          'DISCHARGING_CC'    ,
755                                          'CHARGED'           ,
756                                          'DISCHARGED'        ,
757                                          'OVERCURRENT_CHARGE',
758                                          'OVERCURRENT_DISCHARGE']
759      # =========================
760      # GUI LOOP
761      # =========================
762      def gui_loop(self):
763          while not self.data_queue.empty():
764              item = self.data_queue.get()
765
766              if item[0] == 'ALARM':
767                  messagebox.showerror("ALARM", "Safety limit exceeded! Output disabled.")
768                  continue
769
770              t, v, c, self.state_of_controller, actual_power, SOC, SOE, temp_bat = item
771              self.actual_voltage.set(round(v, 3))
772              self.actual_current.set(round(c, 3))
773              self.actual_power.set(actual_power)
774              self.actual_soc.set(SOC)
775              self.actual_soe.set(SOE)
776              self.temp_bat.set(temp_bat)
777
778              self.times.append(t)
779              self.voltages.append(v)
780              self.currents.append(c)
781              self.SOCs.append(SOC)
782              self.SOEs.append(SOE)
783              self.powers.append(actual_power)
784              self.temp_bats.append(temp_bat)
785
786              if self.csv_file:
```

```
                with open(self.csv_file, 'a', newline='') as f:
                    #"Time[s]", "Voltage[V]", "Current[A]", "Power[W]", "SOC", "SOE", "State", "Power_ref",
"Current_ref", "temp_bat_c"
                                csv.writer(f, delimiter=',').writerow([t, v, c, actual_power, SOC, SOE,
self.state_of_controller, self.reference_cc_mode.get(), self.set_current.get(), self.power_ref.get(), temp_bat])

            self.actual_state.set(self.BatteryController_state_table[self.state_of_controller])



                                                    if       self.power_enabled.is_set()    ==    True    and
(self.BatteryController_state_table[self.state_of_controller]        ==         'OVERCURRENT_CHARGE'         or
self.BatteryController_state_table[self.state_of_controller] == 'OVERCURRENT_DISCHARGE'):
                print("Turn off converter")
                self.power_enabled.clear()
                self.turn_off_converter()
                self.run_var.set(False)
                self.run_btn["state"] = "disabled"
                self.recipe_btn["state"] = "disabled"
                self.recipe_var.set(False)
                                                elif    self.run_btn["state"]    ==    "disabled"    and    not
(self.BatteryController_state_table[self.state_of_controller]          ==         'OVERCURRENT_CHARGE'         or
self.BatteryController_state_table[self.state_of_controller] == 'OVERCURRENT_DISCHARGE'):
                self.run_btn["state"] = "normal"
                self.recipe_btn["state"] = "normal"
```

Code excerpt 18 DSP controller-state table, measurement processing, state display, and overcurrent response. Source lines 750-805 of hmi.py.

Code excerpt 18 presents the processing of the DSP controller state and the acquired measurement packet. A state table maps the numerical state received from the DSP to descriptive labels such as INIT, RUN, CHARGING_CC, CHARGING_CV, DISCHARGING_CC, CHARGED and DISCHARGED. The GUI loop retrieves measurements from the queue, updates the displayed voltage, current, power, SOC, SOE and temperature, and appends the values to the historical arrays used for visualization.

The controller state is displayed together with the continuous measurements. This is particularly useful during experimental validation because electrical trajectories can be directly associated with the active embedded control state, for example with the transition between constant-current and constant-voltage charging.

The same excerpt contains the HMI response to the OVERCURRENT_CHARGE and OVERCURRENT_DISCHARGE states. When either state is reported while the converter is enabled, the HMI clears the local enable condition, sends the converter-off command and disables the Run and Recipe controls. Thus, fault detection remains associated with the DSP, while the HMI provides an additional supervisory response.

2.2.8 Automated experimental recipes

In addition to manual operation, the HMI incorporates a recipe mechanism for automatic execution of predefined experiments. Experimental sequences are stored in CSV format. Each recipe entry contains six principal fields: time, voltage, current, power, mode, and reference type.

When a recipe is loaded, its rows are converted into a sequence of internally stored operating points. Starting the recipe sets the recipe-running state, initializes the experiment start time, enables the power condition, invokes recipe processing, and activates the converter.

During recipe execution, the application calculates the elapsed time from the beginning of the recipe and searches the recipe for the most recent step whose specified time is less than or equal to the elapsed experiment time. The voltage, current, power, operating mode, and reference type associated with that step then become the active HMI commands. The recipe routine is scheduled again every 200 ms using the Tkinter event mechanism.

$$t_{elapsed} = t_{current} - t_{recipe,start}$$

This implementation is useful for battery experimentation because complex operating profiles can be defined externally without modifying the HMI source code. A researcher can construct repetitive charge/discharge cycles, stepwise current profiles, transitions between operating conditions, or other time-dependent experiments by modifying a CSV recipe. An important implementation detail is that the current version selects recipe steps according to their absolute time from the beginning of the recipe, rather than interpreting the time field as the duration of each individual step.

The implementation of CSV recipe loading and time-dependent automatic execution of experimental operating points is shown in **Code excerpt 19**.

hmi.py – lines 893–973

```
893      def toggle_recipe(self):
894          if not hasattr(self, 'recipe_steps') or not self.recipe_steps:
895              messagebox.showwarning("Recipe", "No recipe loaded")
896              self.recipe_var.set(0)
897              return
898
899          if self.recipe_var.get():
900              self.recipe_running = True
901              self.run_var.set(0)
902              self.power_enabled.set()
903              self.recipe_start_time = time.time()
904              self.run_recipe()
905              self.turn_on_converter()
906          else:
907              self.recipe_running = False
908              self.turn_off_converter()
909
910
911      def load_recipe(self):
912          path = filedialog.askopenfilename(filetypes=[('CSV', '*.csv')])
913          if not path:
914              return
915
916          self.recipe_steps = []
917
918          with open(path, newline='') as f:
919              reader = csv.DictReader(f)
920              for row in reader:
921                  print(row)
922                  self.recipe_steps.append({
923                      'time': float(row['Time']),
924                      'voltage': float(row['Voltage']),
925                      'current': float(row['Current']),
926                      'power':  float(row['Power']),
927                      'mode': row['Mode'].strip().upper(),
928                      'reference': row['Reference'].strip().upper()
929                  })
930              print("Recipe:")
931              print(self.recipe_steps)
932          '''
933          with open(path) as f:
934              reader = csv.DictReader(f)
935              for row in reader:
936                  print(row)
937                  self.set_voltage.set(float(row['Voltage']))
938                  print(self.set_voltage)#Test
939                  self.set_current.set(float(row['Current']))
940                  self.mode.set(row['Mode'])
941                  break
942          '''
943
944
945      def run_recipe(self):
946          if not self.recipe_running:
947              return
948
949          elapsed = time.time() - self.recipe_start_time
950
951          step = self.recipe_steps[0]
952          for s in self.recipe_steps:
953              if elapsed >= s['time']:
954                  step = s
955
956          self.reference_cc_mode.set(step['reference'])
957
958          self.set_voltage.set(step['voltage'])
959
960          self.set_current.set(step['current'])
961
```

```
962          self.power_ref.set(step['power'])
963
964          number = self.set_current.get();
965          if number < 0.0:
966            self.set_current.set(abs(number))
967          self.current_ref = self.set_current.get()
968
969
970
971          self.mode.set(step['mode'])
972
973          self.root.after(200, self.run_recipe)
```

Code excerpt 19 Recipe start/stop logic, CSV recipe loading, elapsed-time evaluation, and automatic setpoint updates. Source lines 893-973 of hmi.py.

Code excerpt 7 shows the implementation of automated experimental recipes. The load_recipe function reads a CSV file and converts each row into a structured step containing time, voltage, current, power, operating mode and reference type. Experimental profiles can therefore be prepared externally and reused without modifying the HMI source code.

When recipe execution begins, the application records the recipe start time, sets the recipe-running flag and activates the converter. During run_recipe, elapsed time is calculated from the start of the sequence. The code then selects the latest recipe step whose specified time has been reached and transfers its voltage, current, power, mode and reference selection to the active HMI variables.

The recipe function schedules its next execution after 200 ms using Tkinter's after mechanism. It therefore does not block the GUI while waiting for the next step. In the current implementation, the Time field represents absolute elapsed time from the beginning of the recipe rather than the duration of an individual step.

2.2.9 Overall application activity

At application startup, the graphical interface and internal variables are initialized. The operator selects a serial interface and establishes communication with the DSP. Following successful connection, the acquisition thread runs independently of the graphical event loop. User actions can modify references and operating modes, enable or disable the converter, load or execute recipes, select a data-recording file, reset selected DSP quantities, and terminate the application.

During operation, newly acquired measurements are transferred through the queue and processed by the GUI loop. The displayed values, historical arrays, controller-state indication, CSV data stream, and plots are updated from the same measurement packet. This establishes a consistent supervisory path from the physical battery experiment to operator visualization and data storage.

The complete application workflow, from initialization and DSP connection through acquisition, user interaction, visualization, and termination, is summarized in **Figure 10**.

Figure 10 describes the overall activity of the PC application. After initialization, the operator selects the serial port and establishes communication with the DSP. Successful communication starts the independent acquisition thread. During normal execution, operator commands and recipe actions modify references and modes, while newly acquired samples are passed to the GUI through the thread-safe queue. The GUI then updates numerical indicators, plots and optional CSV recording. This cyclic organization continues until the application is closed.

The main GUI processing loop responsible for receiving queued measurements, updating displayed quantities, recording data, and scheduling subsequent updates is shown in **Code excerpt 20**.

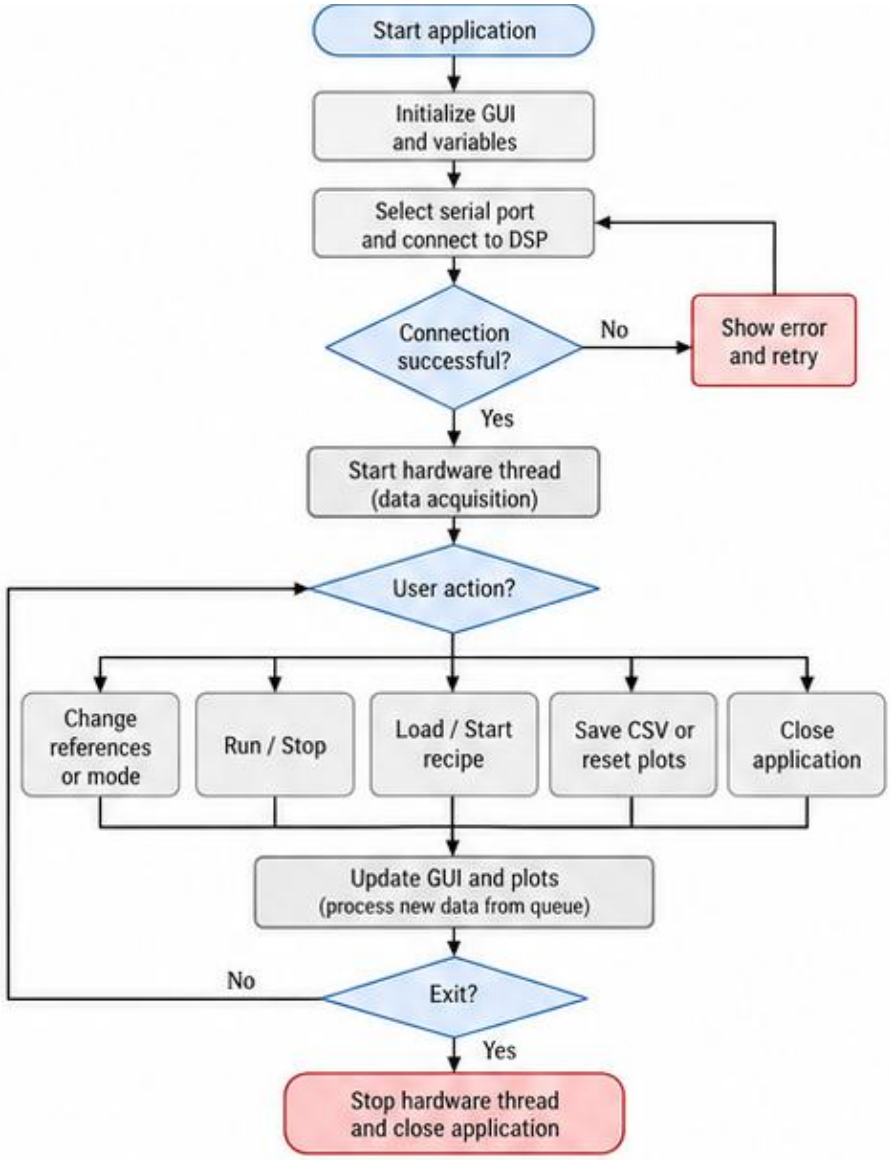


Figure 10 Conceptual activity diagram of the Python HMI application, from initialization and DSP connection to user interaction, measurement processing, and application termination.

hmi.py — lines 762–811

```
762     def gui_loop(self):
763         while not self.data_queue.empty():
764             item = self.data_queue.get()
765 
766             if item[0] == 'ALARM':
767                 messagebox.showerror("ALARM", "Safety limit exceeded! Output disabled.")
768                 continue
769 
770             t, v, c, self.state_of_controller, actual_power, SOC, SOE, temp_bat = item
771             self.actual_voltage.set(round(v, 3))
772             self.actual_current.set(round(c, 3))
773             self.actual_power.set(actual_power)
774             self.actual_soc.set(SOC)
775             self.actual_soe.set(SOE)
776             self.temp_bat.set(temp_bat)
```

```
777
778             self.times.append(t)
779             self.voltages.append(v)
780             self.currents.append(c)
781             self.SOCs.append(SOC)
782             self.SOEs.append(SOE)
783             self.powers.append(actual_power)
784             self.temp_bats.append(temp_bat)
785
786             if self.csv_file:
787                 with open(self.csv_file, 'a', newline='') as f:
788                     #"Time[s]", "Voltage[V]", "Current[A]", "Power[W]", "SOC", "SOE", "State", "Power_ref",
"Current_ref", "temp_bat_c"
789                                 csv.writer(f, delimiter=',').writerow([t, v, c, actual_power, SOC, SOE,
self.state_of_controller, self.reference_cc_mode.get(), self.set_current.get(), self.power_ref.get(), temp_bat])
790
791             self.actual_state.set(self.BatteryController_state_table[self.state_of_controller])
792
793
794
795                                                 if      self.power_enabled.is_set()   ==   True   and
(self.BatteryController_state_table[self.state_of_controller]       ==        'OVERCURRENT_CHARGE'        or
self.BatteryController_state_table[self.state_of_controller] == 'OVERCURRENT_DISCHARGE'):
796                 print("Turn off converter")
797                 self.power_enabled.clear()
798                 self.turn_off_converter()
799                 self.run_var.set(False)
800                 self.run_btn["state"] = "disabled"
801                 self.recipe_btn["state"] = "disabled"
802                 self.recipe_var.set(False)
803                                             elif   self.run_btn["state"]   ==   "disabled"   and   not
(self.BatteryController_state_table[self.state_of_controller]        ==        'OVERCURRENT_CHARGE'        or
self.BatteryController_state_table[self.state_of_controller] == 'OVERCURRENT_DISCHARGE'):
804                 self.run_btn["state"] = "normal"
805                 self.recipe_btn["state"] = "normal"
806
807
808
809
810         self.update_plot()
811         self.root.after(200, self.gui_loop)
```

Code excerpt 20 Main GUI processing loop, queue consumption, numerical display updates, CSV logging, and periodic GUI scheduling. Source lines 762-811 of hmi.py.

Code excerpt 20 presents the main GUI data-processing loop. Measurement packets generated by HardwareThread are removed from the queue and unpacked into timestamp, voltage, current, controller state, power, SOC, SOE and battery temperature. The values are used both to update the numerical indicators and to extend the historical arrays required by the plots.

If CSV recording is enabled, the same sample is written to the experimental file together with the controller state and reference information. The displayed and recorded quantities therefore originate from the same acquired data packet, which supports consistency between online observation and subsequent offline analysis.

After the available queue data have been processed, the plotting function is called and gui_loop schedules itself again after 200 ms. This event-driven organization allows GUI processing, communication and recipe execution to operate without a blocking loop in the main Tkinter thread.

2.2.10 Real-time acquisition and visualization

After acquisition, each data packet transferred through the queue contains a timestamp together with voltage, current, controller state, power, SOC, SOE, and battery temperature. These values are used both for numerical presentation and for building the historical measurement arrays used by the graphical scope.

The sampling rate can be adjusted through the graphical interface. In the hardware thread, the interval between successive acquisition cycles is approximately defined by the inverse of the selected sampling frequency:

$$T_s = 1 / f_s$$

The GUI itself is refreshed independently every 200 ms. The Modbus acquisition frequency and graphical refresh frequency are therefore conceptually different quantities. The embedded plotting system currently consists of three vertically arranged graphs displaying battery voltage, battery current, and SOC. The time axis is implemented as a moving window, with the most recent sample corresponding to zero and historical samples represented by negative relative time. The user can select the displayed time interval. In this way, the HMI behaves similarly to a low-frequency digital scope adapted to battery experiments.

The implementation of the moving-window real-time plots for battery voltage, current, and SOC is presented in **Code excerpt 21**.

hmi.py – lines 816–846

```
816      def update_plot(self):
817          self.ax[0].clear()
818          self.ax[1].clear()
819          self.ax[2].clear()
820
821
822          if self.times:
823              now = self.times[-1]
824              tmin = now - self.time_window.get()
825
826              x = [t - now for t in self.times if t >= tmin]
827              v = [v for t, v in zip(self.times, self.voltages) if t >= tmin]
828              c = [c for t, c in zip(self.times, self.currents) if t >= tmin]
829              soc = [soc for t, soc in zip(self.times, self.SOCs) if t >= tmin]
830              soe = [soe for t, soe in zip(self.times, self.SOEs) if t >= tmin]
831              power = [power for t, power in zip(self.times, self.powers) if t >= tmin]
832
833
834              self.ax[0].plot(x, v)
835              self.ax[1].plot(x, c)
836              self.ax[2].plot(x, soc)
837              self.ax[1].set_xlim(-self.time_window.get(), 0)
838
839          self.ax[0].set_ylabel("Voltage (V)")
840          self.ax[1].set_ylabel("Current (A)")
841          self.ax[2].set_ylabel("SOC[Ah]")
842          self.ax[2].set_xlabel("Time (s)")
843          self.ax[0].grid(True)
844          self.ax[1].grid(True)
845          self.ax[2].grid(True)
846          self.canvas.draw_idle()
```

Code excerpt 21 Moving-window real-time visualization of voltage, current, and SOC. Source lines 816-846 of hmi.py.

Code excerpt 21 implements the real-time plotting function. At every update, the plotting axes are cleared and the beginning of the displayed time interval is calculated from the newest sample and the time-window value selected by the operator. Only data belonging to this interval are extracted for visualization.

The selected timestamps are expressed relative to the newest sample, producing a moving-window display in which the latest point is located at zero seconds. Battery voltage, current and SOC are plotted on separate axes with their corresponding physical units and grid lines. The use of draw_idle requests an efficient refresh of the embedded Matplotlib canvas and allows the plots to operate as a low-frequency scope for battery experiments.

2.2.11 Experimental data recording

The HMI provides direct storage of experimental measurements in CSV format. When data recording is activated, every processed sample is written together with its timestamp, voltage, current, power, SOC, SOE, controller state, selected reference type, current and power references, and battery temperature.

This function establishes a direct connection between the physical experiment and subsequent offline battery-model analysis. The same experiment can therefore be monitored in real time while simultaneously generating a structured dataset for numerical processing. For the laboratory setup, this is particularly important because quantities calculated by the DSP, such as SOC and SOE, can subsequently be compared against independently calculated offline quantities. The HMI therefore provides the data-acquisition pathway required for verification of the real-time implementation of the battery-state algorithms.

The complementary CSV mechanisms used for measurement recording and experimental recipe definition are shown in **Code excerpt 22**.

hmi.py – lines 885–929

```
885      def choose_csv(self):
886          self.csv_file = filedialog.asksaveasfilename(defaultextension='.csv', filetypes=[('CSV', '*.csv')])
887          if self.csv_file:
888              with open(self.csv_file, 'w', newline='') as f:
889                  csv.writer(f).writerow(self.BatteryController_state_table)
890                  #"Time[s]", "Voltage[V]", "Current[A]", "Power[W]", "SOC", "SOE", "State", "Ref[0:I,1:P]"
891                    csv.writer(f).writerow(["Time[s]", "Voltage[V]", "Current[A]", "Power[W]", "SOC", "SOE",
"State", "Reference", "Power_ref", "Current_Ref", "Battery_temperature_C"])
892
893      def toggle_recipe(self):
894          if not hasattr(self, 'recipe_steps') or not self.recipe_steps:
895              messagebox.showwarning("Recipe", "No recipe loaded")
896              self.recipe_var.set(0)
897              return
898
899          if self.recipe_var.get():
900              self.recipe_running = True
901              self.run_var.set(0)
902              self.power_enabled.set()
903              self.recipe_start_time = time.time()
904              self.run_recipe()
905              self.turn_on_converter()
906          else:
907              self.recipe_running = False
908              self.turn_off_converter()
909
910
911      def load_recipe(self):
912          path = filedialog.askopenfilename(filetypes=[('CSV', '*.csv')])
913          if not path:
914              return
915
916          self.recipe_steps = []
917
918          with open(path, newline='') as f:
919              reader = csv.DictReader(f)
920              for row in reader:
921                  print(row)
922                  self.recipe_steps.append({
923                      'time': float(row['Time']),
924                      'voltage': float(row['Voltage']),
925                      'current': float(row['Current']),
926                      'power':  float(row['Power']),
927                      'mode': row['Mode'].strip().upper(),
928                      'reference': row['Reference'].strip().upper()
929                  })
```

Code excerpt 22 CSV measurement-file creation and CSV-based experimental recipe parsing. Source lines 885-929 of hmi.py.

Code excerpt 22 illustrates the two CSV-based mechanisms used by the HMI. The first is measurement recording. When the operator selects a destination file, the application creates the CSV file and writes headings for time, voltage, current, power, SOC, SOE, controller state, reference selection, reference values and battery temperature. Measurement rows are subsequently appended during GUI processing.

The second mechanism is recipe loading. A CSV recipe is read row by row and its fields are converted into the variables required for automated operation. Using CSV for both experimental input and output simplifies preparation of test profiles and subsequent processing of the acquired battery data in external numerical-analysis tools.

2.2.12 Fault response and operational supervision

The application reacts to overcurrent states reported by the DSP. If the converter is enabled and the controller enters either OVERCURRENT_CHARGE or OVERCURRENT_DISCHARGE, the HMI clears its power-enabled state, transmits the converter-off command, resets the Run indication, disables Run and Recipe controls, and clears the recipe selection. Once the controller is no longer reporting either overcurrent state, the controls are enabled again. A separate Reset Error command is also implemented through a dedicated Modbus coil.

This behaviour should be distinguished from general software-side protection. Although the application contains a dictionary of nominal safety limits, the supplied HardwareThread does not itself perform general comparison of voltage, current, and temperature against those values. The clearly implemented HMI protection response in the present version is primarily based on the overcurrent states reported by the DSP. The principal real-time protection should therefore be regarded as belonging to the DSP/controller layer rather than being attributed to Python.

The DSP reset commands exposed through the HMI for error reset and SOC/SOE reinitialization are shown in **Code excerpt 23**.

hmi.py – lines 556–579

```
556          self.MODBUSADDRESS_RESET_ERROR = 4
557          self.MODBUSADDRESS_CURRENT_POWER_REF = 5
558          self.MODBUSADDRESS_REF_CURRENT = 0
559          self.MODBUSADDRESS_RESET_SOC_SOE = 6
560
561
562
563      def reset_soc_soe(self):
564       self.modbus_client.write_coils(address=self.MODBUSADDRESS_RESET_SOC_SOE, values=[True], device_id=0xaf)
565
566
567      def toggle_mode(self):
568        if self.mode.get() == "CHARGE":
569          print("mode charge")
570                                              self.modbus_client.write_registers(address=12,    device_id=0xaf,
values=HardwareThread.convert_float_to_bytes(self.power_ref.get()))
571        elif self.mode.get() == "DISCHARGE":
572                                              self.modbus_client.write_registers(address=12,    device_id=0xaf,
values=HardwareThread.convert_float_to_bytes(-self.power_ref.get()))
573
574
575      def reset_error(self):
576
577        #Zanimljivo ovo dole komentirano za poboljasanje debug
578        #print(f"ResetButton code at line: {sys.exc_info()[2].tb_lineno}")
579        self.modbus_client.write_coils(address=self.MODBUSADDRESS_RESET_ERROR, values=[True], device_id=0xaf)
```

Code excerpt 23 Reset commands and DSP error-reset interface. Source lines 556-579 of hmi.py.

Code excerpt 11 shows the supervisory reset commands available from the HMI. The reset_soc_soe function writes to the Modbus coil associated with reinitialization of the SOC/SOE calculation, while reset_error sends the command used to clear the DSP error condition.

The HMI does not itself perform these reset operations. Instead, it sends a request to the embedded controller, where the corresponding real-time action is executed. This is consistent with the overall architecture in which the PC provides supervisory commands while the DSP remains responsible for internal control and state management.

2.2.13 Role of the HMI in validation of the laboratory setup

The developed HMI provides the experimental flexibility required to use the battery charger/discharger as a configurable laboratory platform for battery-model validation. As described in the preceding sections, the time-critical control functions remain implemented on the DSP, whereas the HMI provides supervisory access to operating references, experimental sequences, monitoring, and data recording. This separation enables experimental operating conditions to be systematically modified without altering the deterministic execution of the real-time control algorithms.

This capability is particularly important for the experimental validation of battery models intended for subsequent microgrid studies. Different charging and discharging conditions can be systematically imposed through the HMI while the resulting battery voltage, current, power, State of Charge (SOC), State of Energy (SOE), temperature, and controller state are continuously monitored and recorded. The resulting experimental datasets can subsequently be used to evaluate the battery response under different operating conditions and to compare the behaviour predicted by the battery model with that observed in the physical system.

The HMI therefore constitutes an integral component of the experimental validation procedure, providing a systematic link between the definition of experimental operating scenarios, their execution on the DSP-controlled physical system, the acquisition and recording of the resulting measurements, and the subsequent offline evaluation and validation of the battery model. In this manner, the developed laboratory setup supports repeatable and systematically defined experiments required for battery-model parameter identification, verification, and further development, while maintaining the physical operating constraints and supervisory behaviour of the battery-converter system.

3 Experimental data

The laboratory setup described in the previous section was developed to provide a flexible experimental platform for the investigation, implementation, and validation of battery-system models and control algorithms. Before the setup can be applied to more advanced studies of battery integration in microgrid environments, its principal functions relevant to battery modelling must be experimentally evaluated. Therefore, this section focuses on the experimental verification of those parts of the laboratory setup that directly influence the acquisition, calculation, and interpretation of battery operating data.

Particular attention is given to the capability of the developed system to perform controlled charging and discharging while continuously acquiring the electrical quantities required for battery-state determination. The experimental investigation consequently considers the measured battery voltage and current, the execution of the charging and discharging sequences, and the calculation of state variables such as State of Charge (SOC) and State of Energy (SOE). Comparison between quantities calculated in real time by the implemented DSP algorithms and those reconstructed independently from the recorded experimental data is used to verify the consistency of the measurement, control, and calculation procedures.

The experimental analysis is subsequently extended to investigate the influence of battery efficiency on the calculated state variables. This aspect is particularly important for SOE, since an energy-based battery representation must account for the difference between the electrical energy supplied during charging and the energy that can subsequently be recovered during discharge. Repeated charge–discharge experiments are therefore used to identify the consequences of neglecting these losses, evaluate the introduction of experimentally determined efficiency, and establish the requirements for more advanced estimation of battery capacity, efficiency, and SOC.

The purpose of the experimental results presented in this section is therefore not limited to characterization of the investigated battery. More importantly, they are used to **evaluate the developed laboratory setup as a tool for further battery-model development and validation**. Verification of controlled battery operation, measurement acquisition, real-time state calculation, efficiency representation, and battery-state estimation establishes the experimental basis required for subsequent development of simplified but sufficiently accurate battery models. Such experimentally validated models are particularly important for microgrid studies, where battery behaviour influences energy exchange with the grid, storage utilization, energy-management decisions, operating costs, and the assessment of appropriate battery capacity.

3.1 Analysis of Measured and Calculated State of Charge and State of Energy

The experimental evaluation of the developed battery system includes the analysis of both the **State of Charge (SOC)** and the **State of Energy (SOE)**. These two quantities provide complementary information about battery operation. SOC describes the evolution of the charge accumulated in the battery, whereas SOE provides an energy-based representation that additionally accounts for the battery terminal voltage. Their experimental evaluation is important not only for subsequent battery modelling but also for verification of the hardware and software implementation of the laboratory setup described in the previous section.

A particularly important objective of the presented experiments is the verification of the **real-time SOC and SOE calculation algorithms implemented in the Texas Instruments DSP**. During battery operation, the DSP processes the acquired electrical measurements and calculates the corresponding battery-state quantities in real time. Independently, the same experimentally acquired voltage, current, and time data are processed offline to calculate SOC and SOE. Comparison of the quantities determined by the real-time DSP

implementation with those obtained independently through offline calculation therefore provides a direct means of verifying the implemented algorithms and the complete measurement and data-processing chain.

For this initial verification, neither the real-time measured quantities nor the corresponding offline calculations include an explicit correction for battery charging and discharging efficiency. This is intentional, since the primary objective at this stage is to verify that the real-time algorithms implemented in the DSP correctly perform the required numerical calculations and reproduce the results obtained independently from the recorded experimental data. The consequences of neglecting efficiency can subsequently be identified from the evolution of SOC and SOE during repeated charging and discharging cycles and used as the basis for further improvement of the battery model.

3.1.1 Determination of Measured State of Charge

The measured SOC used in the experimental analysis is obtained from the SOC quantity calculated in real time during operation of the battery laboratory system. The real-time algorithm implemented in the DSP processes the measured battery current and continuously updates the charge-state information. Consequently, the term *measured SOC* used in **Figures 11 – 13** refers to the SOC trajectory recorded from the real-time laboratory system and used as a reference for comparison with the independently performed offline calculation.

Before comparison, the recorded SOC data are processed to establish a consistent reference for the different experimental datasets. The SOC trajectory is first shifted relative to the selected experimental time origin. An additional offset is subsequently applied so that the minimum SOC value within the considered experimental interval corresponds to zero. This normalization changes only the reference level and does not modify the relative variations produced during charging and discharging.

The SOC quantity presented in the experimental figures is expressed in terms of accumulated charge in ampere-hours. It therefore directly represents the charge balance used by the implemented algorithm. If required, this quantity can subsequently be normalized with respect to the defined battery capacity to obtain conventional SOC expressed as a fraction or percentage.

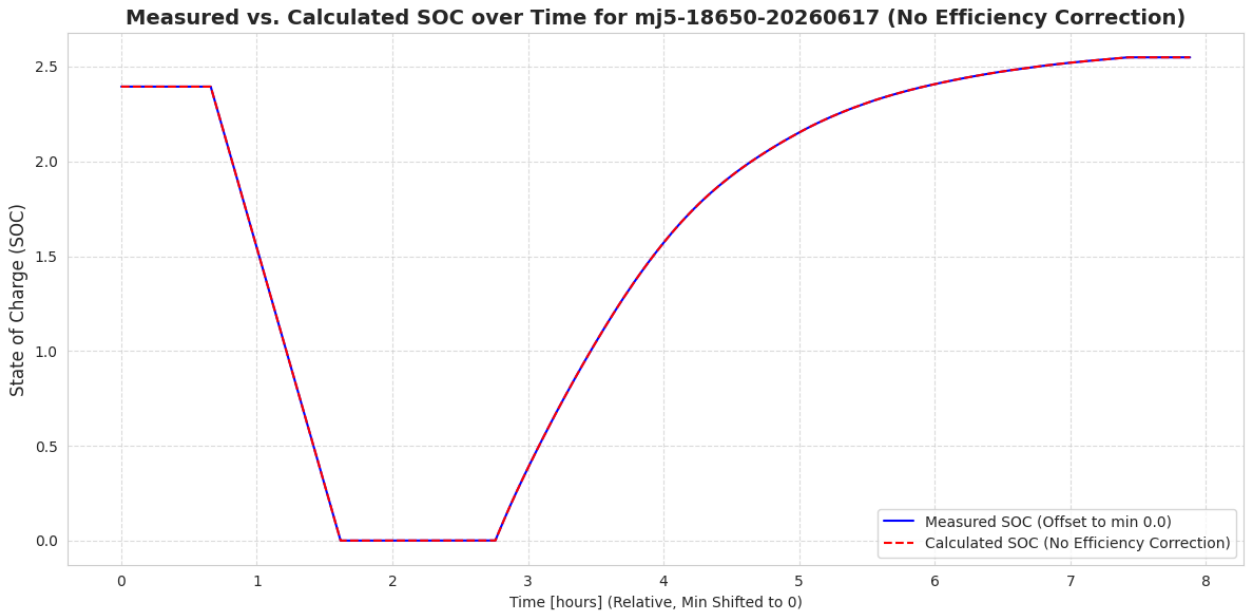


Figure 11 Measured vs. Calculated SOC over time for mj5-18650-20260617 (no efficiency correction).

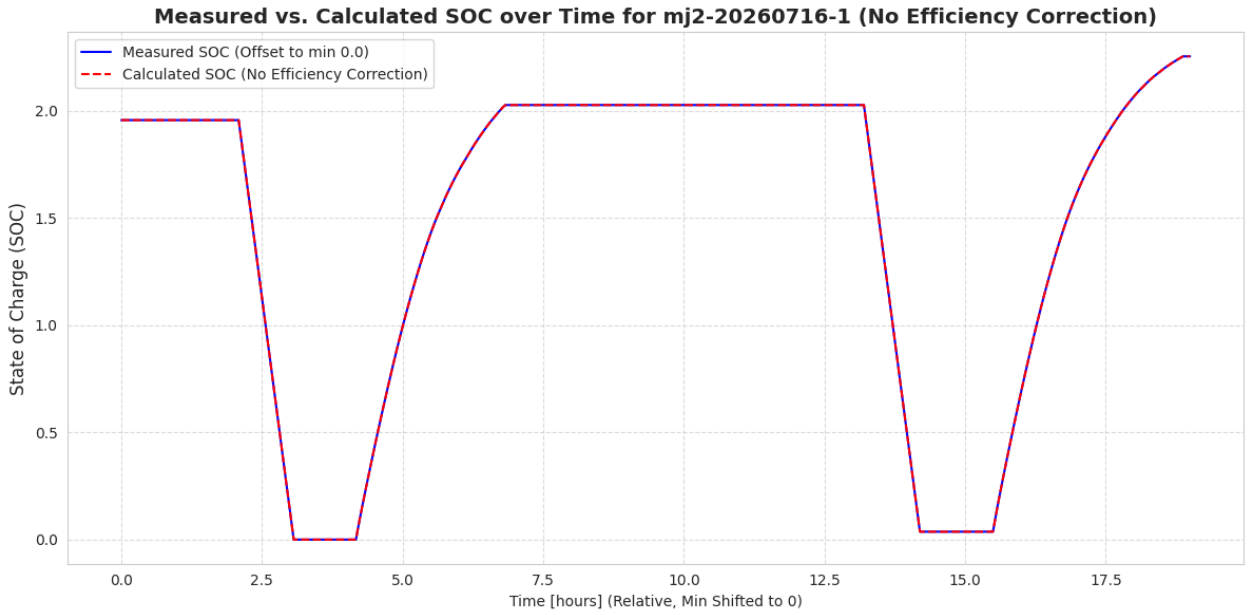


Figure 12 Measured vs. Calculated SOC over time for mj2-20260716-1 (no efficiency correction).

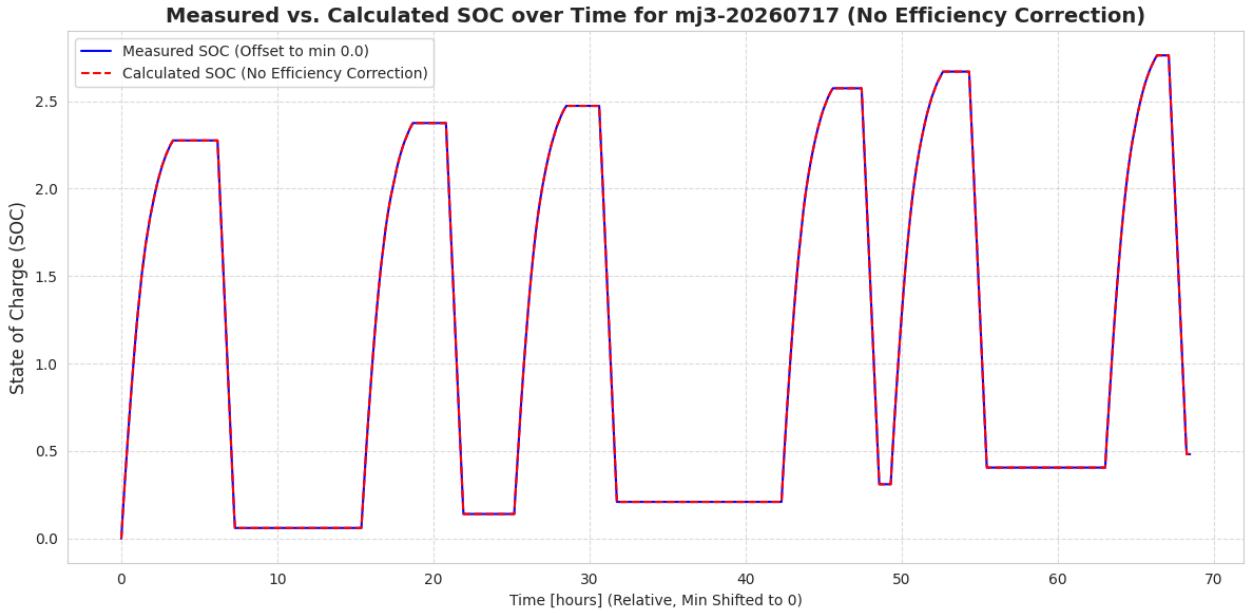


Figure 13 Measured vs. Calculated SOC over time for mj3-20260717 (no efficiency correction).

An important characteristic of the real-time calculation is that **no explicit coulombic-efficiency correction is applied at this stage**. The DSP algorithm therefore evaluates the battery charge state directly from the measured current throughput. This implementation provides a baseline charge-balance model whose behaviour can be independently verified using the recorded experimental data.

3.1.2 Offline Calculation of State of Charge

For verification of the real-time DSP algorithm, an independent SOC calculation is performed offline using the battery current recorded during each experiment. The purpose of this calculation is to determine whether

the charge-state evolution produced in real time by the DSP can be reproduced independently from the experimental measurements.

Prior to numerical integration, the recorded battery current is smoothed using a moving-average procedure to reduce the influence of measurement noise and short-duration fluctuations. The elapsed time between consecutive measurement points is obtained from the original time information, allowing the calculation to account for the actual sampling intervals present in the experimental dataset.

For each measurement interval, the transferred electrical charge is determined from the measured current and the corresponding elapsed time. These incremental charge contributions are accumulated over the complete experimental interval and converted to ampere-hours. The resulting trajectory, shown on **Figures 11 – 13** therefore represents the cumulative battery charge obtained independently through offline numerical integration.

As in the real-time DSP implementation, **no coulombic-efficiency correction is introduced into this calculation**. Charging and discharging currents are integrated directly according to their measured values. Consequently, both the real-time and offline calculations are based on the same fundamental assumption and can be directly compared for verification purposes.

Before comparison, the calculated SOC is aligned with the recorded real-time SOC using a common reference. This procedure modifies only the initial reference level and does not change the temporal variation of the calculated quantity. Any subsequent difference between the two trajectories therefore provides an indication of differences between the real-time and offline implementations.

3.2 Determination of Measured State of Energy

The measured SOE used in the experimental analysis is similarly obtained from the SOE quantity generated during real-time operation of the laboratory system. The corresponding algorithm uses the acquired electrical quantities to continuously determine the energy-state evolution of the battery. The resulting SOE is recorded together with the other experimental variables and subsequently compared with an independent offline energy calculation.

As for SOC, the recorded SOE is normalized before comparison. The SOE trajectory is referenced to the selected experimental time origin and shifted so that its minimum value within the analysed interval corresponds to 0 Wh. This establishes a common reference between the different experimental datasets without changing the magnitude of the energy variations occurring during battery operation.

The real-time SOE calculation also **does not include an explicit battery-efficiency correction** in the experiments presented in **Figures 14 – 16**. The calculated state therefore represents the cumulative energy balance obtained from the electrical quantities without correcting the charging and discharging energy according to the actual battery energy efficiency.

This characteristic is particularly important for interpreting the experimental results. The progressive change in the SOE reference observed during repeated battery operation is therefore not an error introduced by the offline processing alone. The same fundamental behaviour is present in the real-time SOE because both calculations intentionally neglect the energy losses occurring in the physical battery.

3.2.1 Offline Calculation of State of Energy

An independent SOE trajectory is calculated offline from the measured battery voltage and current. In contrast to SOC, which is determined from current integration, SOE is obtained from the integration of electrical power and therefore depends simultaneously on battery current and terminal voltage.

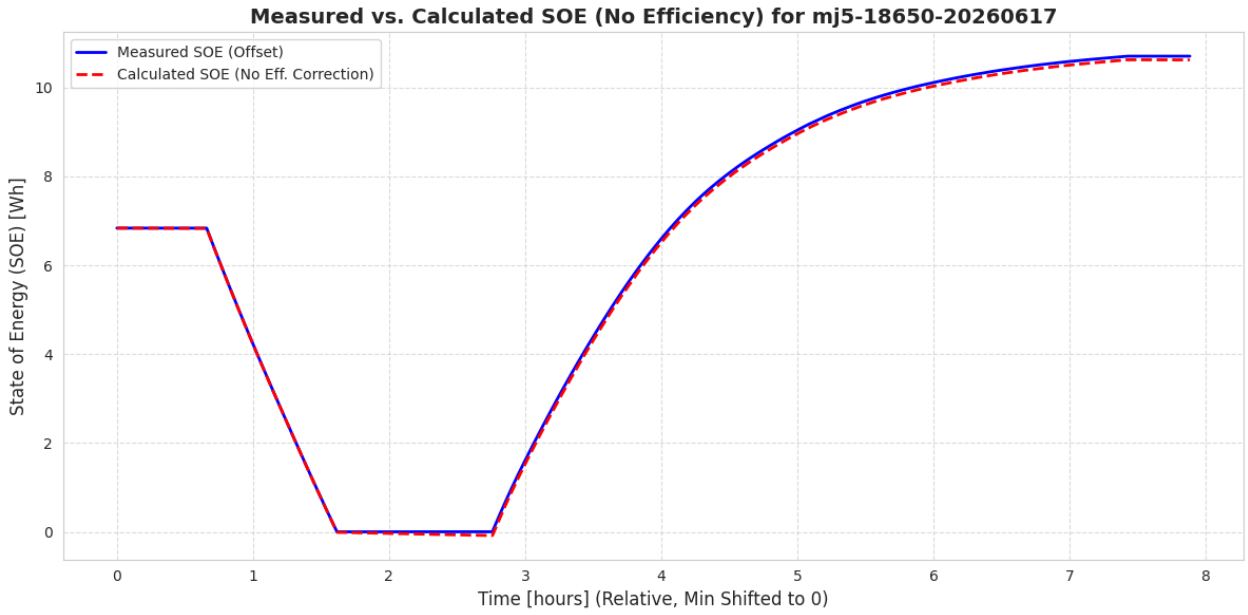


Figure 14 Measured vs. Calculated SOE over time for mj5-18650-20260617 (no efficiency correction).

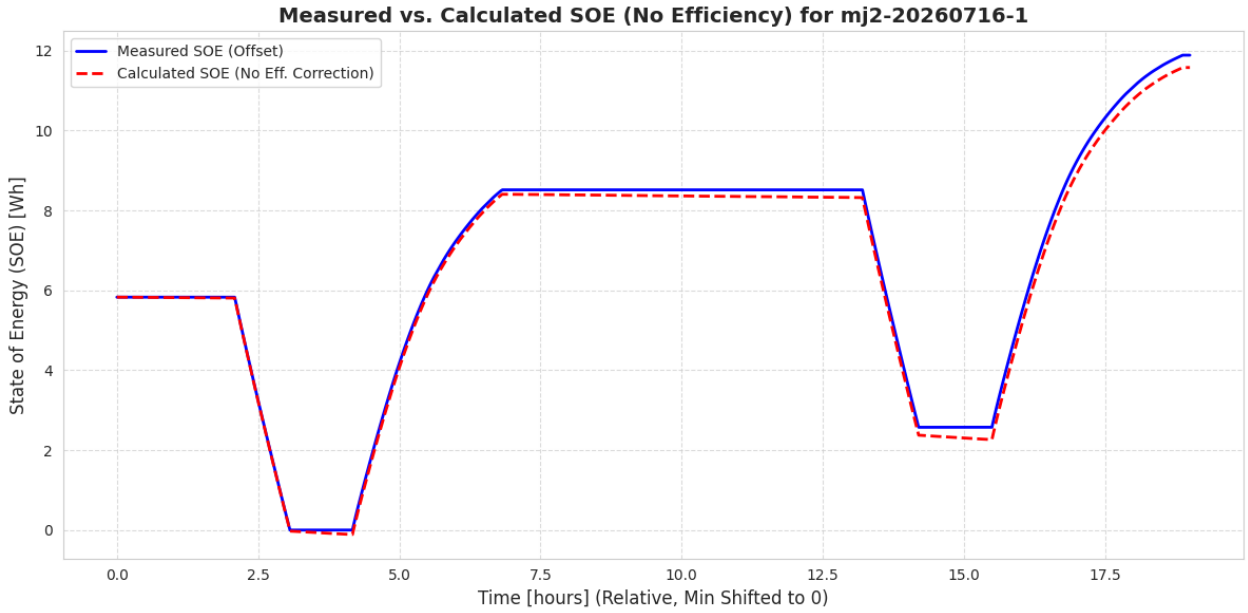


Figure 15 Measured vs. Calculated SOE over time for mj2-20260716-1 (no efficiency correction).

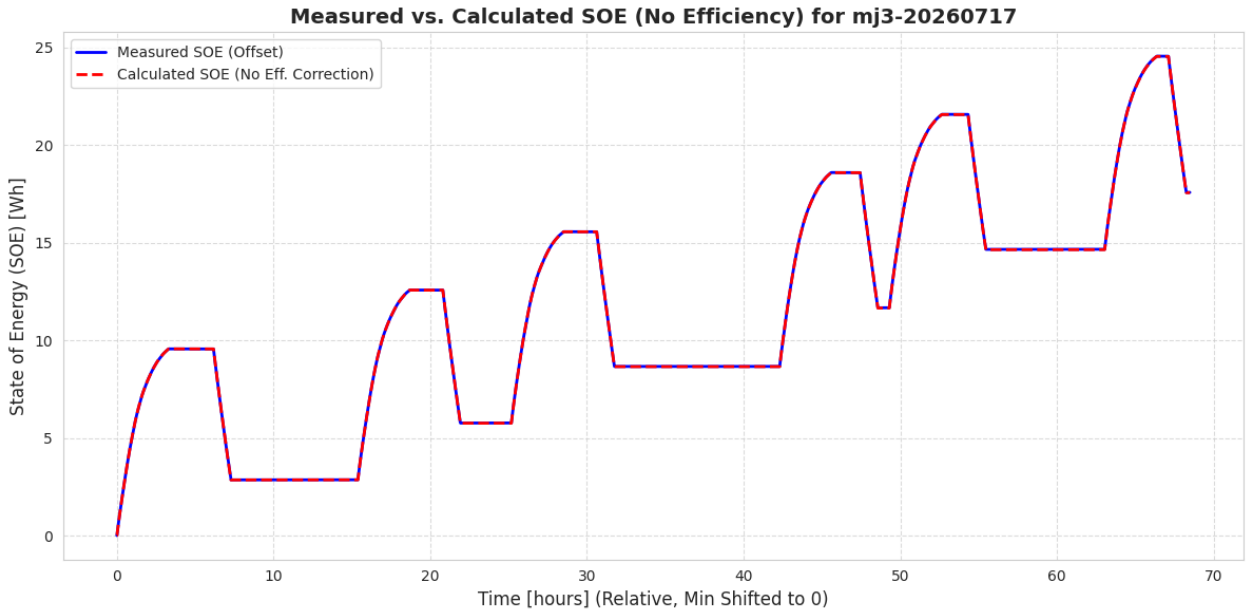


Figure 16 Measured vs. Calculated SOE over time for mj3-20260717 (no efficiency correction).

Before calculation, a predefined current threshold is applied to reduce the influence of measurement noise during periods in which the battery is effectively inactive. The battery power is subsequently determined from the measured voltage and current. The elapsed time between consecutive measurement points is obtained from the experimental time information, and the corresponding incremental energy is determined for each sampling interval. The individual energy contributions are accumulated and converted to watt-hours to obtain the calculated SOE trajectory.

As in the real-time DSP implementation, the offline SOE calculation is intentionally performed **without applying a charging or discharging efficiency correction**. The electrical energy entering and leaving the battery is therefore accumulated directly from the measured terminal power.

The calculated SOE is subsequently aligned with the real-time measured SOE at the selected reference point. This alignment establishes a common initial condition but does not alter the subsequent energy variation. The comparison therefore provides a direct verification of whether the real-time DSP algorithm reproduces the energy integration obtained independently from the recorded voltage, current, and time data.

3.2.2 Experimental Comparison of Measured and Calculated SOE

The comparison between the real-time measured SOE and the independently calculated offline SOE is presented in **Figures 14 – 16**. The results demonstrate that the offline calculation closely reproduces the principal dynamic behaviour of the SOE generated by the DSP. The discharge, charge, and idle intervals are consistently represented in both trajectories, demonstrating correct implementation of the energy-state calculation and consistency between the real-time and offline processing.

Figure 14 presents a relatively simple discharge–charge sequence. The measured and calculated SOE trajectories follow the same overall behaviour, including the decrease in energy during discharge, the approximately constant value during the idle period, and the subsequent increase during charging. The close correspondence between the two trajectories provides an initial verification of the real-time SOE algorithm.

The verification is extended in **Figure 15**. to an operating sequence containing multiple charging, discharging, and idle intervals. Again, the calculated SOE reproduces the changes observed in the real-

time SOE, including the transitions between the different operating periods. The experiment therefore demonstrates that the implemented algorithm continues to provide consistent results under a more complex sequence of battery operating conditions.

The long-duration experiment presented in **Figure 16** provides the most demanding verification. The battery is subjected to repeated charging and discharging operations over approximately 72 h. The overall evolution of the SOE generated by the DSP is reproduced by the independently calculated trajectory throughout the experiment. Consequently, the results provide experimental verification of the ability of the implemented system to perform continuous SOE calculation during extended and repeated battery operation.

At the same time, **Figures 14 – 16** reveal an important characteristic of the SOE evolution. Following repeated charging and discharging operations, the SOE does not return to the same value after each nominal cycle but progressively shifts towards a higher value. This behaviour is present because **battery efficiency is not included in either the real-time or offline SOE calculation**.

During charging, electrical energy is supplied to the battery and included in the calculated energy balance. However, because the physical battery is not lossless, the same amount of electrical energy cannot subsequently be recovered during discharge. Part of the charging energy is dissipated through internal resistance, polarization, electrochemical processes, and heat generation. Consequently, the magnitude of the energy recovered during discharge is lower than the energy previously supplied during charging. When efficiency is not explicitly incorporated into the SOE calculation, this difference remains in the accumulated energy balance and produces a **positive residual SOE after each charge–discharge sequence**. Repeated cycles therefore result in a progressive increase or positive shift of the SOE.

This result is particularly important because the increasing SOE should not be interpreted as a physical increase in the usable energy capacity of the battery. Instead, it represents the consequence of applying an energy-balance algorithm without correcting for the non-reversible energy conversion of the physical battery. The experimentally observed drift therefore demonstrates why battery efficiency must be incorporated when the SOE algorithm is subsequently extended from implementation verification to accurate battery-state and economic modelling.

3.2.3 Experimental Comparison of Measured and Calculated SOC

The corresponding verification of the SOC algorithm is presented in **Figures 11 – 13**. As with the SOE analysis, both the SOC obtained from the real-time DSP implementation and the independently calculated SOC are determined without applying an explicit efficiency correction.

In **Figure 11**, the real-time and offline calculated SOC trajectories exhibit very close agreement throughout the discharge, idle, and charging intervals. The correspondence between the two quantities confirms that the current integration implemented in the DSP is reproduced by the independent offline calculation.

The more complex operating sequence presented in **Figure 12** provides further verification. The measured and calculated SOC remain practically coincident during multiple charging and discharging operations, demonstrating that the implemented real-time algorithm correctly accumulates the measured battery current even when the operating state changes repeatedly.

The long-duration results presented in **Figure 13** provide particularly strong verification of the SOC implementation. Despite numerous charging, discharging, and idle periods over approximately 72 h, the independently calculated SOC closely follows the SOC generated in real time by the DSP. This confirms the numerical consistency and stability of the implemented charge-integration algorithm over extended operation.

As observed for SOE, however, the SOC itself exhibits a progressive shift over repeated charge–discharge sequences. Since no coulombic-efficiency correction is applied, the algorithm does not compensate for the difference between the charge transferred into the battery and the charge that can subsequently be recovered. The physical battery does not provide perfectly reversible charge transfer, and therefore a residual difference can remain after a complete charging and discharging sequence. Repetition of this process results in the accumulation of the residual difference and consequently in a progressive shift of the calculated charge state.

The close agreement between the real-time and offline SOC trajectories therefore does not mean an accurate battery model can neglect efficiency. Instead, it demonstrates that **both calculations consistently reproduce the same charge-balance algorithm**. This distinction is essential: the experiment verifies the algorithm's implementation, while the progressive SOC behaviour identifies a limitation of the underlying simplified battery representation that must be addressed by introducing an appropriate efficiency correction.

3.2.4 Influence of Efficiency Estimation Error

The observed behaviour also has an important implication for the subsequent introduction of efficiency into the battery model. Including an efficiency coefficient is necessary to compensate for the difference between the quantities supplied during charging and those recoverable during discharge. However, the accuracy of the corrected state estimate will depend directly on the accuracy with which the corresponding efficiency has been determined.

If the assumed efficiency differs from the actual battery efficiency, each charging or discharging interval introduces a small systematic error into the calculated state. Since SOC and SOE are cumulative quantities, these errors are retained and propagated into subsequent operating intervals. Repeated battery cycling can therefore transform a relatively small error in the assumed efficiency into a significant long-term state-estimation error.

This effect is expected to be more pronounced for SOE than for SOC. SOC is primarily sensitive to uncertainty in coulombic efficiency, whereas SOE is affected by the complete energy-conversion process and therefore includes the combined influence of current, terminal voltage, and energy losses. Consequently, **an error in the estimated efficiency produces a greater sensitivity in SOE estimation**, making SOE more error-prone than SOC when the efficiency representation is inaccurate.

This observation is particularly relevant when a simplified model is intended for long-term simulation. A small error that may appear negligible during one charging or discharging event can accumulate over a large number of cycles. Therefore, the efficiency parameters used in the final model should be experimentally identified and validated over representative operating conditions.

3.2.5 Verification of the Real-Time Algorithms and Laboratory Setup

An important outcome of the presented experiments is the verification of the complete implemented battery laboratory system. The experiments do not evaluate only the characteristics of the physical battery; they simultaneously test the measurement subsystem, real-time DSP implementation, data acquisition, numerical integration procedures, and interaction between the physical battery and the controlled power-electronic system.

The close correspondence between the SOC generated by the real-time DSP algorithm and the SOC independently reconstructed from the recorded current demonstrates the correct implementation of the real-time charge-integration procedure. Similarly, the agreement between the real-time SOE and the SOE independently reconstructed from measured voltage and current verifies the implementation of the corresponding real-time energy calculation. The successful reproduction of charging, discharging, idle

intervals, and repeated operating sequences further demonstrates that these algorithms remain operational during transitions between the different battery operating states.

The long-duration experiments are particularly relevant for this verification. The results demonstrate that the laboratory system can continuously acquire and process battery measurements and calculate the corresponding state variables during repeated operation extending over several days. Therefore, the experimental results provide verification not only of individual numerical algorithms but also of the **integrated operation of the hardware and software laboratory setup**.

At the same time, the experiments clearly identify the next required model improvement. The observed increase in SOC and particularly SOE over repeated charge–discharge sequences demonstrates the consequence of neglecting battery efficiency. This behaviour does not invalidate the implemented real-time algorithms; rather, because the independent offline calculations reproduce the same behaviour, it confirms that the algorithms operate according to their defined mathematical formulation. The experimentally observed drift then provides the physical motivation for extending this formulation by incorporating appropriate charging and discharging efficiencies.

Consequently, the developed laboratory setup and the implemented DSP algorithms can be considered **experimentally verified for the investigated operating conditions and suitable as a basis for further battery-system studies**. The platform can subsequently be used to identify efficiency parameters, evaluate improved SOC and SOE algorithms, investigate different charging and discharging conditions, and validate increasingly detailed battery models.

3.3 Influence of Energy-Efficiency Correction on the Calculated State of Energy

The results presented in the previous sections demonstrate that the real-time and offline SOC and SOE calculations provide consistent results when the same calculation methodology is applied. At the same time, repeated charge–discharge operation revealed a progressive increase in the calculated state when battery efficiency was not considered. This effect was particularly pronounced for SOE, since the electrical energy supplied during charging is greater than the energy that can subsequently be recovered during discharge. Consequently, direct integration of the electrical power without accounting for this energy imbalance results in the accumulation of a positive residual after successive charge–discharge sequences.

To investigate and compensate for this effect, the SOE calculation was extended by introducing the experimentally determined energy efficiency of the battery. The purpose of the efficiency correction is to account for the non-reversible character of battery energy conversion and thereby prevent the complete electrical energy supplied during charging from being interpreted as an equivalent increase in recoverable battery energy. The corrected SOE was then compared with the SOE obtained using the original calculation without efficiency correction.

The comparison was performed for the same three experimental datasets considered in the preceding SOE analysis. The energy-efficiency coefficients applied in the corrected calculations are 64.65% for dataset mj5-18650-20260617, 67.77% for dataset mj2-20260716-1, and 70.15% for dataset mj3-20260717, as indicated in **Figures 17–19**. The experiments progressively increase in duration and complexity, from a single discharge–charge sequence to repeated operation over an extended period. This progression makes it possible to evaluate not only the immediate influence of the efficiency correction, but also its cumulative effect on long-term SOE estimation.

3.3.1 SOE Correction for a Single Discharge–Charge Sequence

The influence of the energy-efficiency correction for dataset mj5-18650-20260617 is presented in **Figure 17**. The figure compares the SOE trajectory obtained without efficiency correction with the trajectory calculated using an energy efficiency of 64.65%. Because both trajectories are derived from the same measured voltage, current, and time data, their separation directly illustrates the effect of introducing the experimentally identified energy-loss representation.

At the beginning of the experiment, the two trajectories are aligned and remain approximately constant during the initial idle interval. During discharge, both decrease from about 6.9 Wh to approximately 0 Wh and remain close to one another. This agreement is expected because the principal modification in the corrected formulation is associated with the treatment of energy supplied during charging. The discharge interval therefore provides a common reference from which the influence of the efficiency correction can be evaluated.

A pronounced separation appears during the subsequent charging interval. Without efficiency correction, the full electrical energy supplied at the battery terminals is accumulated in the SOE state, causing the uncorrected trajectory to rise to approximately 10.7 Wh. This final value is substantially higher than the approximately 6.9 Wh level that preceded discharge. The uncorrected integration therefore produces a positive residual of about 3.8 Wh over a nominal discharge–charge sequence, even though the battery has returned to a comparable operating state.

When the 64.65% energy-efficiency coefficient is applied, the charging contribution is reduced to represent the fraction of supplied terminal energy associated with recoverable battery energy. The corrected SOE consequently rises more slowly and returns to approximately 6.9 Wh at the end of charging, close to its pre-discharge value. The result demonstrates that the experimentally identified efficiency largely compensates for the charging–discharging energy imbalance that otherwise appears as artificial state accumulation.

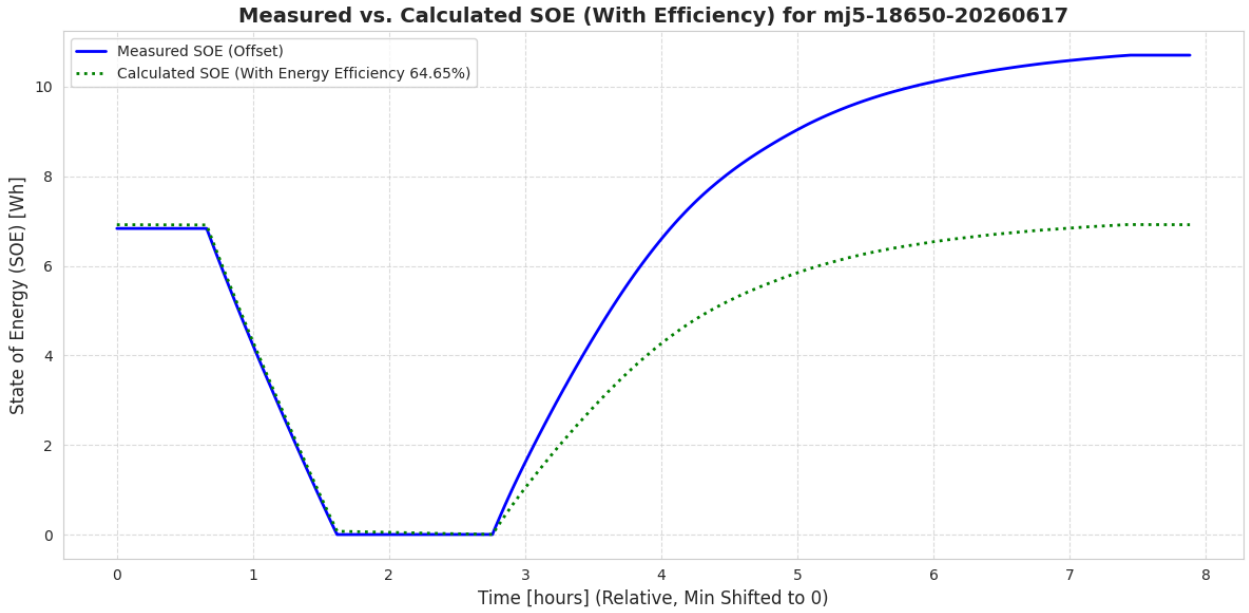


Figure 17 Calculated SOE over time for mj5-18650-20260617 (with efficiency correction).

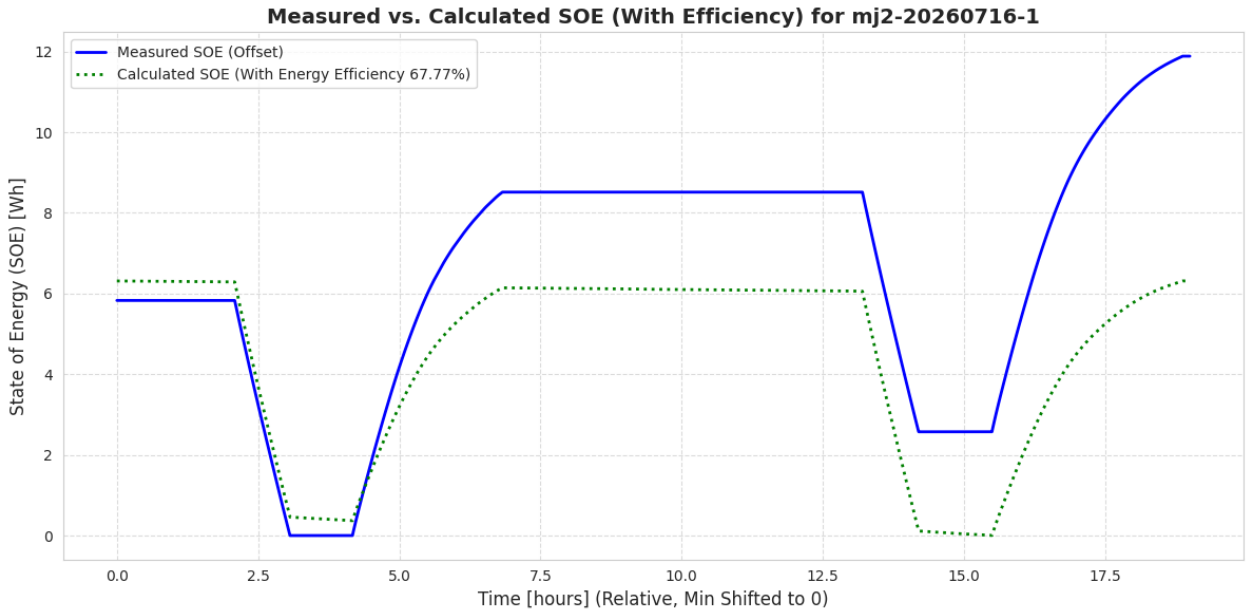


Figure 18 Calculated SOE over time for mj2-20260716-1 (with efficiency correction).

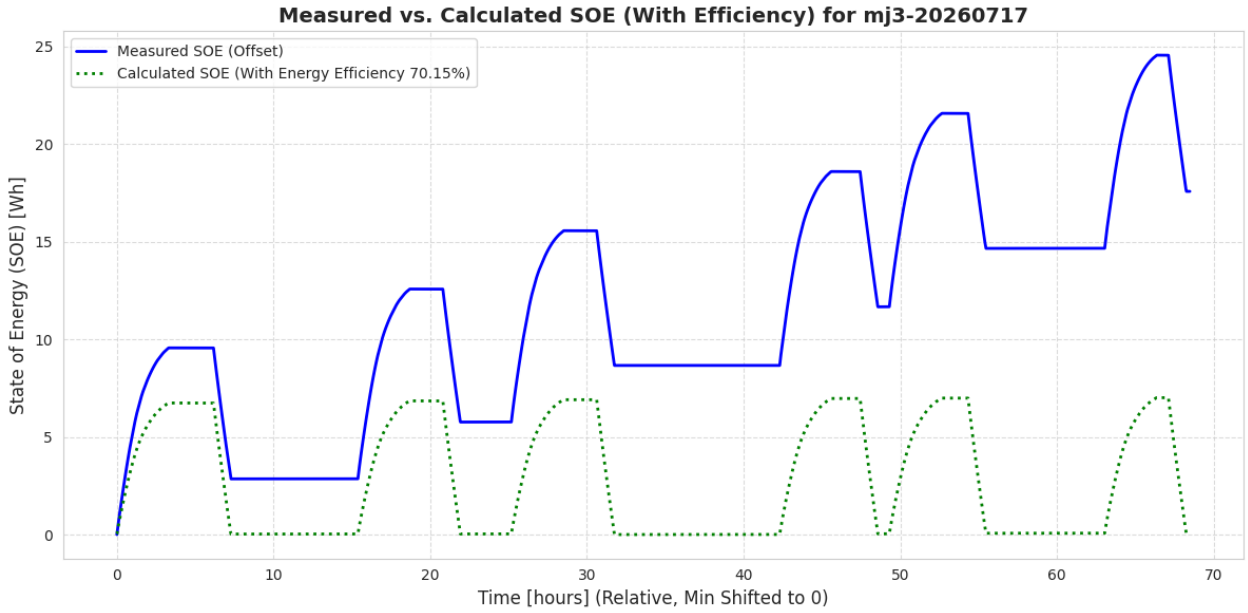


Figure 19 Calculated SOE over time for mj3-20260717 (with efficiency correction).

The approximately 3.8 Wh difference between the corrected and uncorrected trajectories at the end of the sequence is therefore not caused by different experimental data, but by the mathematical representation of battery losses. **Figure 17** provides a clear single-cycle demonstration that SOE is highly sensitive to the treatment of efficiency and that direct terminal-power integration cannot, by itself, be interpreted as a repeatable estimate of recoverable battery energy.

Figure 17 therefore provides the first direct verification that introducing the experimentally determined energy efficiency can substantially suppress the positive SOE shift observed when battery losses are neglected.

3.3.2 SOE Correction during Repeated Charge–Discharge Operation

The effect of efficiency correction becomes more evident when the battery undergoes more than one charge–discharge sequence. **Figure 18** presents dataset mj2-20260716-1, for which an energy efficiency of 67.77% is applied. The experiment contains two major discharge–charge sequences separated by an extended idle interval, allowing the cumulative character of the uncorrected SOE error to be observed directly.

During the first discharge, the uncorrected trajectory decreases from approximately 5.8 Wh to 0 Wh, while the efficiency-corrected trajectory approaches the same lower reference. During the following charge, however, the trajectories diverge: the uncorrected SOE reaches approximately 8.5 Wh, whereas the corrected SOE reaches approximately 6.1 Wh. The corrected upper level is close to the initial energy level, while the uncorrected trajectory already contains a substantial positive residual.

The second discharge–charge sequence makes the cumulative effect especially clear. The uncorrected SOE begins the second sequence from the elevated state produced by the first sequence and, after the second charging interval, rises to approximately 11.8 Wh. In contrast, the corrected trajectory again falls close to 0 Wh during discharge and subsequently returns to approximately 6.3 Wh during charging. Thus, the corrected calculation remains associated with a repeatable lower-to-upper energy excursion, whereas the uncorrected state progressively translates upward.

The uncorrected upper SOE therefore increases from approximately 5.8 Wh initially to about 8.5 Wh after the first charge and about 11.8 Wh after the second. The corresponding corrected upper levels remain close to approximately 6 Wh. This comparison demonstrates that the efficiency-related error is not a fixed offset: it is generated during active energy transfer and is carried into subsequent operation because SOE is a cumulative state variable.

Figure 18 consequently demonstrates how a relatively modest per-cycle energy imbalance can become a significant long-term state-estimation error. Each cycle adds a residual to the uncorrected integral, whereas the efficiency-corrected formulation prevents comparable accumulation and preserves a substantially more repeatable estimate of the battery energy excursion.

The extended idle intervals provide an additional useful observation. Once established, the difference between the trajectories remains essentially unchanged while the battery is inactive; the divergence is generated primarily during charging and discharging rather than by elapsed time. This behaviour supports the interpretation that the drift originates from the representation of battery energy conversion, not from a numerical time-integration instability.

3.3.3 Long-Term Influence of Energy Efficiency on SOE Estimation

The long-term significance of energy-efficiency correction is most clearly demonstrated by dataset mj3-20260717 in **Figure 19**. The experiment extends over approximately 68 h and contains several charging, discharging, and idle intervals. The corrected calculation shown in the updated figure uses an energy-efficiency coefficient of 70.15%. This long-duration sequence is particularly valuable because any systematic imbalance in a cumulative state variable becomes progressively amplified as the number of energy-transfer events increases.

The uncorrected SOE exhibits a pronounced upward displacement over the experiment. Successive charging intervals raise the upper SOE level from approximately 9.5 Wh to about 12.5 Wh, 15.5 Wh, 18.5 Wh, 21.5 Wh, and finally approximately 24–25 Wh. At the same time, the lower levels reached after discharge also move upward. The entire apparent operating window is therefore translated toward progressively larger SOE values, despite the fact that the physical battery capacity has not increased.

This behaviour is the cumulative consequence of integrating terminal energy without accounting for conversion losses. More electrical energy is supplied during charging than can subsequently be recovered during discharge. If the complete charging energy is added to the state while the smaller recovered energy is removed during discharge, the difference remains as a positive residual. Repetition of this process explains both the increasing maxima and the increasing post-discharge minima observed in the uncorrected trajectory.

The efficiency-corrected trajectory exhibits fundamentally different behaviour. Its charging intervals repeatedly reach an upper level of approximately 6.8–7.1 Wh, while the major discharge intervals return the calculated state close to 0 Wh. Rather than translating upward with successive cycles, the corrected SOE repeatedly occupies approximately the same energy range. The updated **Figure 19** therefore provides strong evidence that the 70.15% correction removes the dominant systematic component of the long-term SOE drift for this dataset.

The contrast between the two trajectories becomes increasingly pronounced with experimental duration. By the final charging interval, the uncorrected SOE approaches approximately 25 Wh, whereas the corrected state remains near 7 Wh. Such a large divergence is physically important: the uncorrected value would imply an apparent energy state several times larger than the repeatable energy excursion observed for the same battery. The corrected result is therefore substantially more consistent with a bounded battery-energy state.

The remaining differences between individual corrected cycles are comparatively small, but they should not be interpreted as proof that a single constant efficiency is universally valid. Battery energy efficiency can vary with current, SOC range, temperature, rest periods, and other operating conditions, while measurement uncertainty and differences in cycle endpoints can also affect the identified value. **Figure 19** should therefore be interpreted as verification that efficiency correction removes the dominant cumulative bias, while also motivating further identification of efficiency as a function of operating point.

3.3.4 Comparative Evaluation of the Efficiency-Corrected SOE Results

The combined analysis of **Figures 17–19** demonstrates a consistent influence of energy-efficiency correction across all three datasets. Although the operating sequences differ in duration and complexity and the applied efficiencies are 64.65%, 67.77%, and 70.15%, respectively, the same fundamental result is obtained: neglecting losses causes a positive residual to accumulate in SOE, whereas the corrected calculation remains associated with a substantially more repeatable energy range.

For the single discharge–charge sequence in **Figure 17**, the uncorrected SOE finishes approximately 3.8 Wh above its initial upper level, while the corrected state returns close to that level. In **Figure 18**, the residual from the first sequence is carried into the second, causing the uncorrected upper state to rise from about 8.5 Wh to about 11.8 Wh, whereas the corrected upper state remains near 6 Wh. **Figure 19** extends the same mechanism over several cycles, where the uncorrected state eventually approaches 25 Wh while the corrected charging peaks remain close to 7 Wh.

These results show that the principal benefit of the efficiency correction is not merely a reduction in numerical magnitude. It changes the cumulative state evolution by preventing non-recoverable charging losses from being retained as usable battery energy. The improvement becomes increasingly important as the duration of operation and the number of charge–discharge events increase.

The corrected trajectories also provide a useful criterion for model assessment. A physically meaningful simplified SOE model should remain compatible with the repeatable energy range of the battery under comparable operating conditions. Persistent cycle-to-cycle translation of the entire SOE range, as observed

in the uncorrected calculation, is therefore a clear indication of systematic model bias rather than an actual increase in stored-energy capacity.

The different applied efficiencies—64.65%, 67.77%, and 70.15%—further indicate that a single universal efficiency value should not automatically be assumed across all operating conditions. The variation between datasets is itself relevant model information and supports further experimental characterization of efficiency as a function of operating point and test conditions.

3.3.5 Sensitivity of SOE to the Accuracy of Energy Efficiency

The combined experimental results demonstrate that, although both SOC and SOE calculations reproduce the corresponding real-time DSP results, the influence of neglected battery losses differs between the two state variables. SOC is obtained from current integration and consequently represents primarily the charge balance of the battery, whereas SOE is determined from the integration of electrical power and therefore depends on both battery current and terminal voltage. Differences between the charging and discharging voltage, together with the associated internal battery losses, are consequently reflected more directly in the calculated energy balance.

The results presented in **Figures 17 – 19** further demonstrate that the accuracy of SOE estimation depends directly on the accuracy of the applied energy-efficiency coefficient. Because SOE is a cumulative quantity, a systematic error in the efficiency representation introduces a corresponding error during each energy-transfer interval and is retained in the subsequent state estimate. Although such an error may be small over a single charging or discharging event, its cumulative influence becomes increasingly significant during repeated operation.

This effect is particularly evident in the long-duration experiment presented in **Figure 19**. An inaccurately selected efficiency coefficient would prevent the corrected SOE from remaining within the repeatable energy range of the battery, producing a progressive deviation whose direction and magnitude depend on whether the effective energy conversion is overestimated or underestimated. Long-duration cycling therefore provides a particularly sensitive experimental basis for evaluating the adequacy of the selected efficiency representation.

The relative sensitivity of SOC and SOE can be further evaluated by comparing **Figures 13** and **16**, which correspond to the same long-duration experimental dataset and therefore represent identical operating conditions. Since SOC and SOE are expressed in different physical units, their absolute variations cannot be directly compared; instead, the relative cumulative drift provides a more appropriate measure. In **Figure 13**, the characteristic upper SOC level increases from approximately 2.27 Ah to 2.76 Ah, corresponding to an increase of approximately 21.6%. In **Figure 16**, the corresponding upper SOE level increases from approximately 9.5 Wh to 24.5 Wh, representing an increase of approximately 157.9%. The relative cumulative drift of SOE is therefore approximately 7.3 times greater than that observed for SOC under the same experimental operating sequence.

This quantitative comparison confirms that SOC is less sensitive than SOE to errors associated with the representation of battery efficiency. The greater sensitivity of SOE results from its dependence on the complete electrical energy-transfer process, including both current and terminal voltage, rather than on charge transfer alone. Consequently, accurate experimental identification of the energy-efficiency coefficient is particularly important when SOE is used in long-duration battery simulations and, subsequently, in microgrid operational and economic studies.

The variation in the experimentally identified efficiency coefficients further indicates that a single constant efficiency may not be sufficient over the complete battery operating range. Additional characterization under different current levels, SOC ranges, temperatures, and cycling conditions is therefore required to determine

the appropriate efficiency representation. This is particularly important for battery models intended for microgrid and economic studies, since systematic errors in SOE can lead to incorrect estimates of available battery energy and consequently affect predicted grid-energy exchange, battery utilization, operating costs, and storage-capacity assessment.

3.3.6 SOC estimator verification and analysis of results

Figure 20 presents the SOC estimated by the real-time battery-monitoring algorithm for dataset mj2-20260716-1. The same dataset was previously used in **Figure 12** to verify the agreement between the real-time DSP calculation and the independently reconstructed offline SOC. **Figure 20** is therefore used here to examine additional characteristics of the implemented estimator, particularly the influence of OCV-based correction during rest and voltage-based discharge termination.

A small decrease in SOC can be observed during the rest intervals in **Figure 20**. This behaviour results from the OCV-based correction described in Section 2.1.9. During relatively short charge–discharge cycles, small errors can accumulate in the coulomb-counting estimate, particularly while the estimated battery capacity differs from the actual available capacity. During sufficiently long rest periods, the OCV-based correction gradually compensates for this error. The small changes observed during rest therefore represent correction of the SOC estimate rather than physical removal of charge from the battery. As the estimated battery capacity converges toward its actual value, the corresponding correction required during rest is expected to decrease.

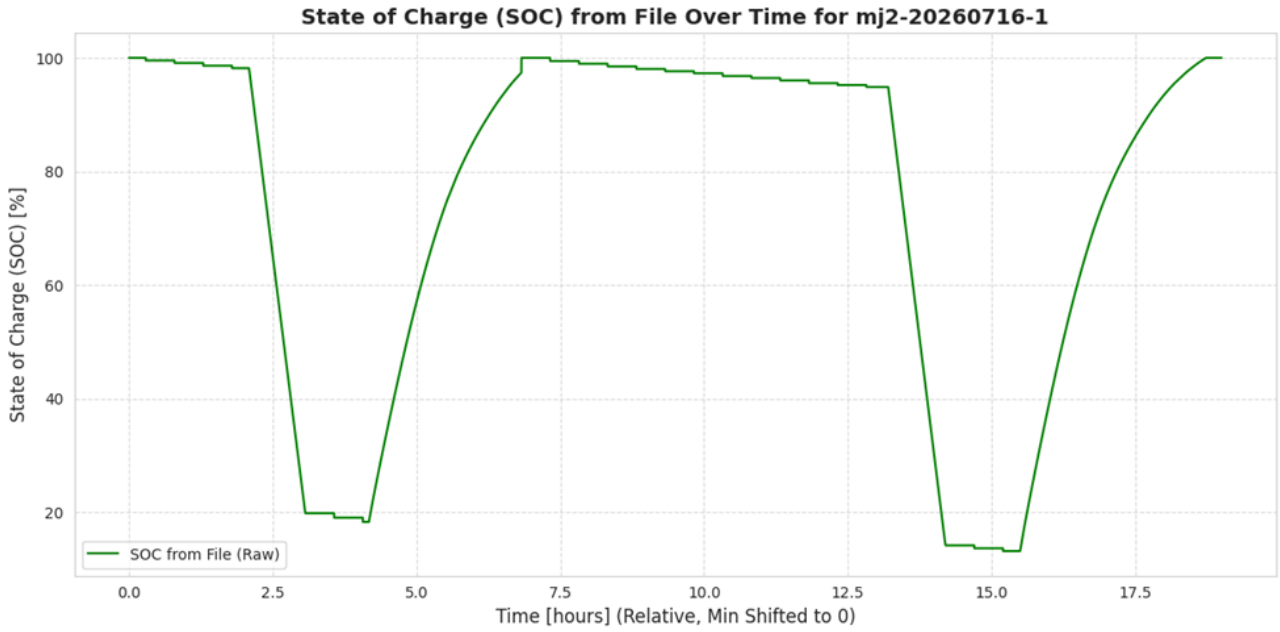


Figure 20 Measured SOC over time for mj2-20260716-1 used for SOC-estimator verification.

Figure 20 also shows that the estimated SOC remains above 0% when discharge is terminated. The supervisory controller stops discharging when the battery terminal voltage reaches the predefined cutoff voltage rather than when the estimated SOC reaches zero. At higher discharge current, the increased voltage drop under load causes this limit to be reached earlier, reducing the usable capacity available before discharge termination. The non-zero minimum SOC is therefore consistent with the interaction between the battery operating condition and the imposed voltage constraint.

Figure 21 extends the analysis to dataset mj3-20260717 and provides verification of the estimator over approximately 70 h of repeated charging, discharging, and rest intervals. Comparison with **Figure 13**

confirms that the real-time estimator remains consistent with the independently reconstructed SOC over the extended operating sequence. The repeated return to similar upper and lower SOC levels indicates stable long-duration operation without pronounced cumulative drift. The small SOC reductions during rest correspond to the same OCV-correction mechanism identified in **Figure 20**, while the non-zero SOC minima observed during successive discharge cycles are consistent with the voltage-based discharge termination described for **Figure 20** and with the corresponding long-duration SOC behaviour shown in **Figure 13**.

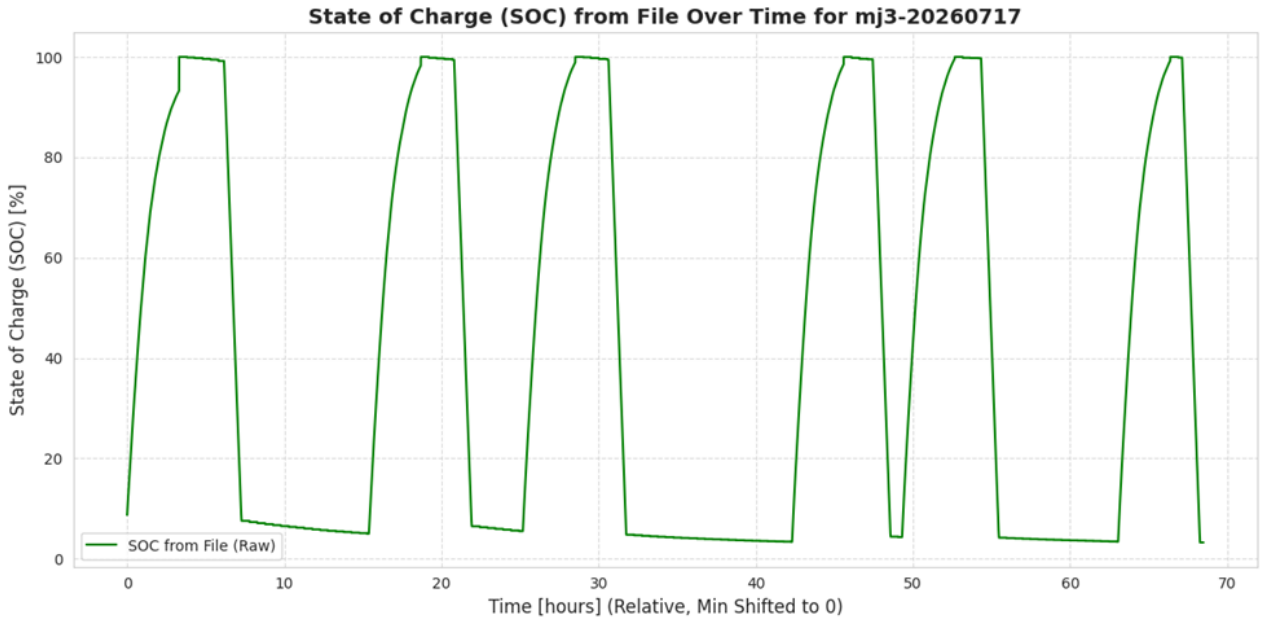


Figure 21 Measured SOC over time for mj3-20260717 used for long-duration SOC-estimator verification.

Taken together, **Figures 20** and **21** demonstrate stable operation of the implemented SOC estimator during both short and extended experimental sequences. The observed SOC trajectories show that accumulated coulomb-counting error is progressively compensated during sufficiently long rest periods through OCV-based correction, while the minimum SOC reached during discharge is constrained by the battery-voltage cut-off rather than by the estimator reaching 0%. The consistency of these behaviours with the corresponding verification results in **Figures 12** and **13** supports the use of the implemented estimator for subsequent battery-model validation under repeated cycling and rest conditions.