Software engineering methods and algorithms for a CAN Bus based battery management system with data recording functionality used for AI training

Rene Santeler, and Thomas Gadner (supervisor)

***Abstract*—In this thesis, methods, components, and implementations from professional embedded software engineering are applied to the development of a battery management system (BMS). Modern society relies heavily on essential safety-critical systems. A BMS is important to ensure the safe and reliable execution of these systems.**

Therefore, the primary goal is to develop a robust BMS platform that is capable of supporting the integration of advanced algorithms in the future. Additionally, the BMS is based on a comprehensive theoretical framework, detailing the methodologies, principles, and tools essential for the effective execution of real-time, resource-limited development projects. Emphasis is placed on the design, implementation, and testing of microcontroller-based software using the C programming language. The crucial components of the BMS are designed to enhance battery safety, performance, and longevity, thereby improving efficiency and reducing waste. The developed prototype features a non-blocking loop design, various safety mechanisms and hot-swap capability. Moreover, an effective balancing and battery state algorithm is included, along with an adapted data recording feature to generate real training data for future AI algorithm development. A user interface and an extensive CAN-based external communication interface are utilized for system control and data access.

All described components have been implemented and tested successfully. The positive results underscore the robustness and reliability of the developed BMS. The outcomes of this paper provide valuable insights into the practical application of embedded software engineering principles and lay the foundation for future advancements in battery management technology.

***Index Terms*—battery management system, embedded system development, hot-swap, CAN, AI training data recording, diagnostic, cell balancing**

I. Introduction

THE development of embedded software stands at the core of numerous modern safety-critical applications, ranging from automotive systems to portable electronics. Battery management systems (BMS) are particularly important for safeguarding the functionality and longevity of devices that depend on battery power. As our reliance on these systems grows, the safety, sustainability and performance of such systems become increasingly important, which makes the adoption of structured and professional embedded software development methodologies essential. This paper documents the application of professional embedded software engineering methods in the development of a BMS. [1] [2]

Rene Santeler studies at the MCI, e-mail: sr4666@mci4me.at

II. Previous Work

A. Hardware

The hardware platform used for this project was previously developed by the Infineon AG. Figure 1 and 2 show the components of the system. [3] [4]

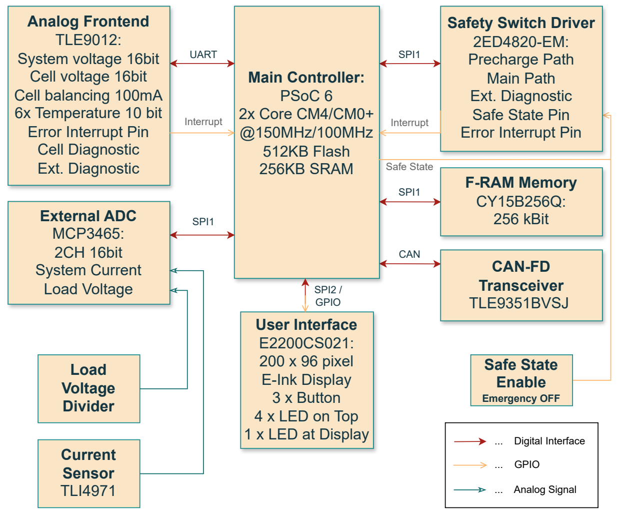


Fig. 1. Each block represents a hardware peripheral, its tasks and most important capabilities. The connections between blocks show the communication/signal type. [3] [4]

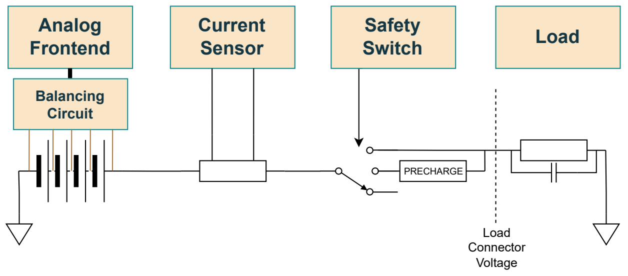


Fig. 2. The simplified schematic of the system shows how the main components are connected [3]

The most important parts are the analog frontend and the safety switch driver. The analog frontend TLE9012 is used for monitoring, balancing and diagnostic of the cells. It is connected to the battery main terminals as well as all tabs between the series cells. The safety switch driver 2ED4820 incorporates many features that are designed to increase performance and keep the system safe and expandable. To control the connection of the battery to the load and allow recharging, an intelligent switch driver with vast safety and diagnostic functionality is used. Both components provide interrupt pins to detect errors and enter a safe state without using the digital bus. [3] [4]

B. Preceding project

Prior to the development of this system, a project to develop a wireless BMS concept demonstrator (BMS via Radio [5] [6]) was realized and basic BMS features were implemented. Additionally, a simple Coloumb-Counting-based battery state estimation library was developed [7]. Parts of this project are used as a starting point for this BMS.

III. Objectives

The objective of the project is to develop a stable expandable BMS platform that can later be used for the implementation of advanced algorithms. To do that, multiple features must be implemented. For safety reasons, the switch must be deactivated if system parameters like temperature, voltage or current are outside defined thresholds. For this purpose, a diagnostic system is needed which also performs logical, peripheral and processing error checks. The soft switch-on of capacitive loads must be controlled by a precharge mechanism. To allow multiple systems to switch the supply between each other, a hot-swap scheme is also needed. Communication between the systems and the load/charger shall be implemented on the CAN messaging protocol. Runtime data storage must be implemented to provide training data for later implementation of AI-based algorithms. Also, a simple but suitable balancing and battery state algorithm must be implemented to provide all basic BMS functionality. For user interaction, a simple user interface incorporating a display, buttons and LEDs is needed.

IV. Concept

A. Design Principles

During the implementation of the main loop, three key design principles are followed:

Non-blocking Software Design: The main loop is engineered to prevent any form of blocking. Using state machines, actions proceed at full speed until they need to pause, at which point they change states rather than waiting idly. When the action that triggered the wait is finished (e.g. peripheral ready), the action resumes and transitions to the next state. This ensures that critical tasks are executed with minimal delay.

Event-driven Execution: Non-critical actions within the loop are executed at specific intervals or as a result of other actions. This allows actions that do not require continuous operation, such as UI updates or communication routines, to run only when it is necessary. Dependencies between actions (e.g. one action requiring fresh data from another) are managed through flags that signal when new data is available, ensuring the system is reactive to changes and conserving computational resources by avoiding redundant calculations.

Timeout Protection: To guarantee reliability and safety, critical operations are always guarded by timeouts. If a task exceeds its expected duration, the timeout triggers, reverting to a previous state and attempting the operation again. This mechanism prevents prolonged system hang-ups due to stalled tasks, such as delayed responses from hardware components, thereby maintaining system integrity.

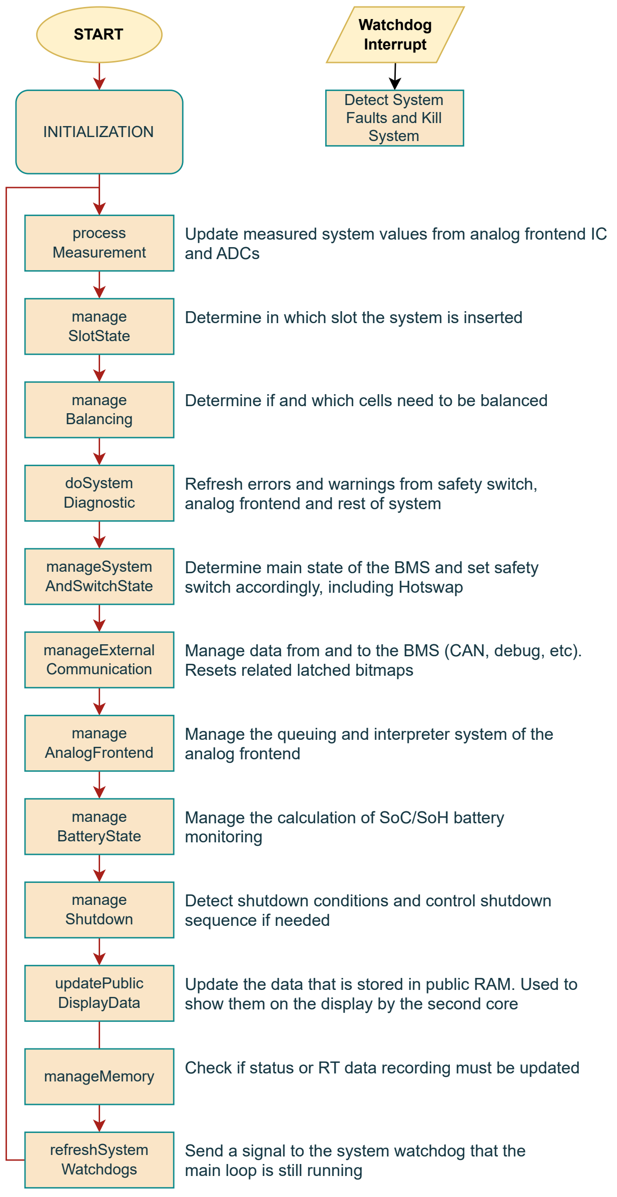

START
Watchdog Interrupt
Detect System Faults and Kill System
INITIALIZATION
process Measurement
Update measured system values from analog frontend IC and ADCs
manage SlotState
Determine in which slot the system is inserted
manage Balancing
Determine if and which cells need to be balanced
doSystem Diagnostic
Refresh errors and warnings from safety switch, analog frontend and rest of system
manageSystem AndSwitchState
Determine main state of the BMS and set safety switch accordingly, including Hotswap
manageExternal Communication
Manage data from and to the BMS (CAN, debug, etc). Resets related latched bitmaps
manage AnalogFrontend
Manage the queuing and interpreter system of the analog frontend
manage BatteryState
Manage the calculation of SoC/SoH battery monitoring
manage Shutdown
Detect shutdown conditions and control shutdown sequence if needed
updatePublic DisplayData
Update the data that is stored in public RAM. Used to show them on the display by the second core
manageMemory
Check if status or RT data recording must be updated
refreshSystem Watchdogs
Send a signal to the system watchdog that the main loop is still running

Fig. 3. Detailed main loop flowchart.

B. Main Loop and System Watchdogs

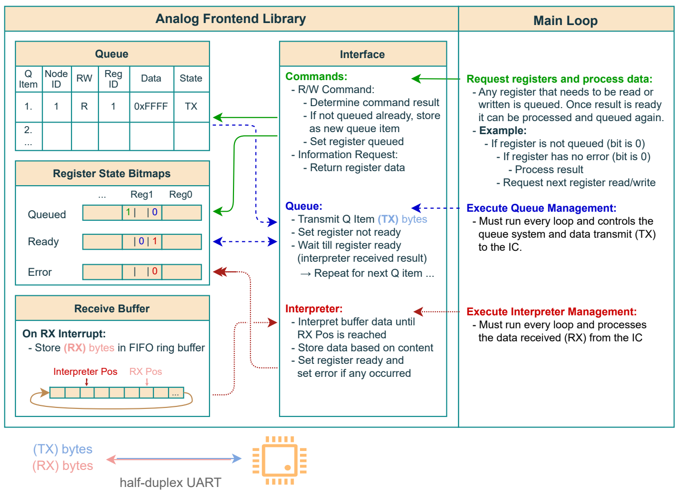
The system's microprocessor contains two cores: the first core is dedicated to the BMS functionalities including performing measurements, executing diagnostics, determining system state, controlling switches, and carrying out housekeeping tasks (as depicted in figure 3). The second core is allocated for managing the user interface, which operates at a slower rate. Detailed explanations of some key functions are given in the next passages.

For fault, 2 types of watchdogs are implemented: a software watchdog with a short interval and a hardware watchdog with a long interval. The software watchdog is designed for soft system shutdowns when the shutdown button is pressed for an extended time or if a hot-swap takes too long. It can also force a hard shutdown if the main loop becomes unresponsive. The hardware watchdog, embedded in the microcontroller hardware, performs a hard reset of the system if it got triggered. This provides a fail-safe even when the system's cores are non-operational. Both watchdogs require regular 'kicking' within their intervals to prevent triggering a shutdown or reset. [8]

C. Analog Frontend Interfacing

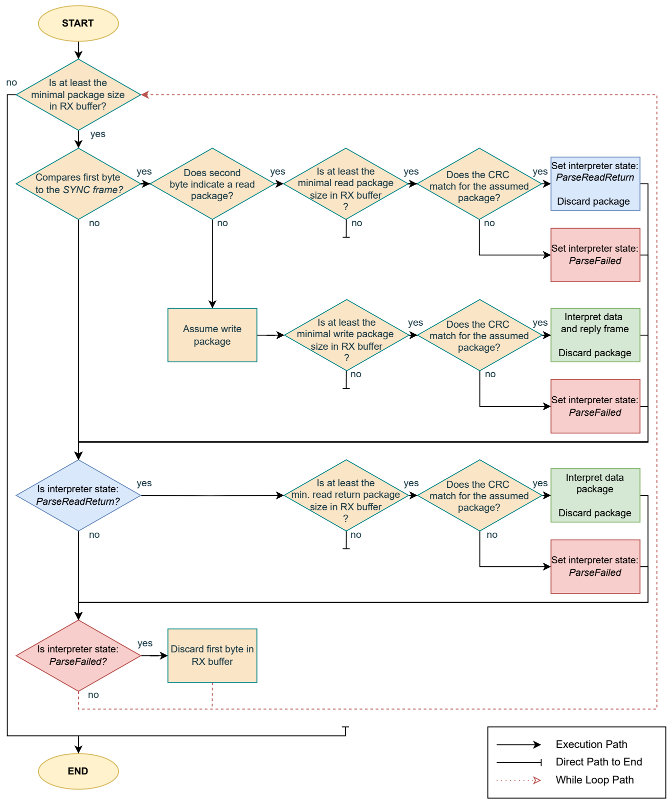
The Analog Frontend Library is engineered to enhance system performance, maintain safety and allow for scalability through a half-duplex communication protocol that supports daisy-chaining of devices. It distinguishes between internal and externally-triggered measurements, using CRC-protected frame layouts for reliable communication, and offering optimization features like multi-read for commonly accessed registers. The library manages over 180 diverse fields for system functions. [9] [10]

In order to comply with the design principles in section IV-A, the library provides a non-blocking interface composed of three main elements: commands, queuing and an interpreter (see figure 4). Commands, which can be executed repeatedly throughout system operation, prepare and queue actions. The queuing system processes these sequentially to comply with the half-duplex nature, preventing simultaneous data transmission and reception. The interpreter monitors and processes complete data packets on the bus.



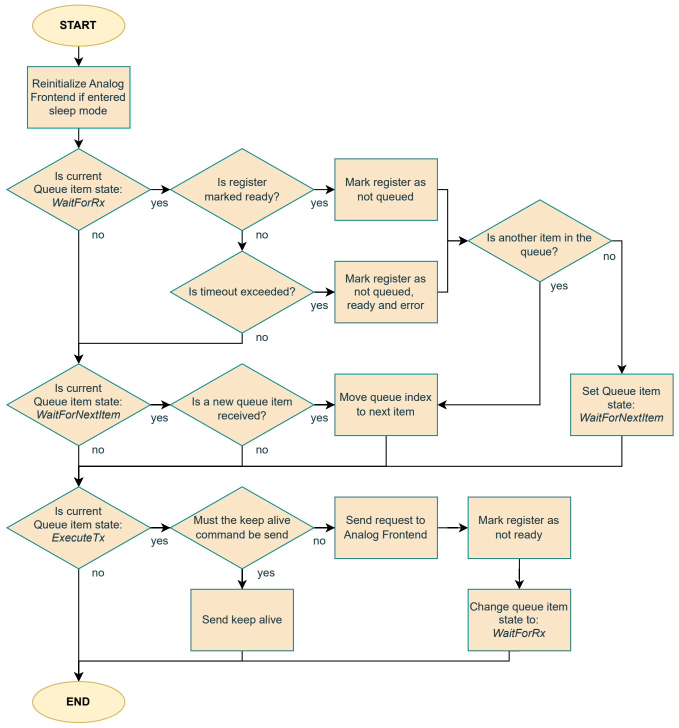
Analog Frontend Library
Main Loop
Queue
Q Item | Node ID | RW | Reg ID | Data | State
1. | 1 | R | 1 | 0xFFFF | TX
2. ...
Register State Bitmaps
... Reg1 Reg0
Queued 1 | 0
Ready 0 | 1
Error 0
Receive Buffer
On RX Interrupt:
- Store (RX) bytes in FIFO ring buffer
Interpreter Pos
RX Pos
Interface
Commands:
- R/W Command:
- Determine command result
- If not queued already, store as new queue item
- Set register queued
- Information Request:
- Return register data
Queue:
- Transmit Q Item (TX) bytes
- Set register not ready
- Wait till register ready (interpreter received result)
→ Repeat for next Q item ...
Interpreter:
- Interpret buffer data until RX Pos is reached
- Store data based on content
- Set register ready and set error if any occurred
Request registers and process data:
- Any register that needs to be read or written is queued. Once result is ready it can be processed and queued again.
- Example:
- If register is not queued (bit is 0)
- If register has no error (bit is 0)
- Process result
- Request next register read/write
Execute Queue Management:
- Must run every loop and controls the queue system and data transmit (TX) to the IC.
Execute Interpreter Management:
- Must run every loop and processes the data received (RX) from the IC
(TX) bytes
(RX) bytes
half-duplex UART

Fig. 4. Overview of the analog frontend library.



START
Is at least the minimal package size in RX buffer?
no
yes
Compares first byte to the SYNC frame?
yes
Does second byte indicate a read package?
yes
Is at least the minimal read package size in RX buffer ?
yes
Does the CRC match for the assumed package?
yes
Set interpreter state: ParseReadReturn
Discard package
no
Set interpreter state: ParseFailed
Assume write package
Is at least the minimal write package size in RX buffer ?
Does the CRC match for the assumed package?
Interpret data and reply frame
Discard package
Set interpreter state: ParseFailed
Is interpreter state: ParseReadReturn?
Is at least the min. read return package size in RX buffer ?
Does the CRC match for the assumed package?
Interpret data package
Discard package
Set interpreter state: ParseFailed
Is interpreter state: ParseFailed?
Discard first byte in RX buffer
END
Execution Path
Direct Path to End
While Loop Path

Fig. 5. Flowchart of the interpreter system implemented by the library.



START
Reinitialize Analog Frontend if entered sleep mode
Is current Queue item state: WaitForRx
yes
no
Is register marked ready?
yes
no
Mark register as not queued
Is timeout exceeded?
yes
no
Mark register as not queued, ready and error
Is another item in the queue?
no
yes
Set Queue item state: WaitForNextItem
Is current Queue item state: WaitForNextItem
Is a new queue item received?
Move queue index to next item
Is current Queue item state: ExecuteTx
Must the keep alive command be send
Send request to Analog Frontend
Mark register as not ready
Send keep alive
Change queue item state to: WaitForRx
END

Fig. 6. Flowchart of the queuing system implemented by the library.

These components operate autonomously and communicate through register state bitmaps that indicate queue status, errors and readiness. Commands compose the read/write register operations, verifying register availability before queuing and handling errors. Results of operations are stored in shadow registers, with specialized functions translating the raw data into usable formats.

The queuing system (figure 6) ensures orderly data flow, alternating between transmission execution (ExecuteTx) and waiting for reception (WaitForRx). It can be timed out if necessary. If all queue items are finished, the queue moves to a waiting state (WaitForNextItem).

The interpreter is tailored to handle variable-sized communication packets, identifying expected sizes from the initial bytes and using CRC validation for integrity. Communication starts with a consistent SYNC byte and an ID byte that distinguishes the node and read/write type, followed by an ADDRESS byte indicating the target register. Write operations follow with DATA and CRC bytes, while read operations finish directly with a CRC byte, to which the IC responds with ID, DATA and CRC bytes. For multi-read operations, configuration at startup is essential to define the expected packet size for each register requested. [9] [10]

D. Measurement

The main loop cycle initiates with a measurement phase, reading amongst others cell and battery voltages via the analog frontend and current/load voltages through an external ADC. The analog frontend measurements are periodic and current measurements are synchronized with these triggers. This is shown in figure 7 and 8;

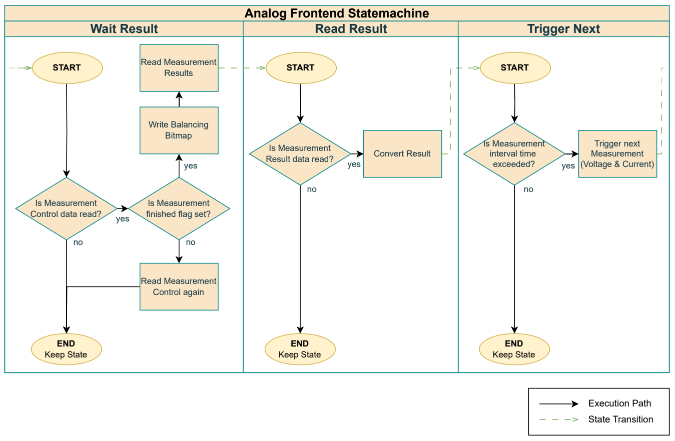

Analog Frontend Statemachine
Wait Result
Read Result
Trigger Next
START
Read Measurement Results
Write Balancing Bitmap
yes
Is Measurement Control data read?
yes
Is Measurement finished flag set?
no
no
Read Measurement Control again
END Keep State
START
Is Measurement Result data read?
yes
Convert Result
no
END Keep State
START
Is Measurement interval time exceeded?
yes
Trigger next Measurement (Voltage & Current)
no
END Keep State
Execution Path
State Transition

Fig. 7. Flow/Statechart of the analog frontend measurement.

Upon triggering the analog frontend measurement, the state machine enters the WaitResult state, doing nothing until the measurement completes. Subsequently, a multi-read of essential registers is queued, transitioning the system into the ReadResult state, where it waits for the multi-read completion. After processing and storing the results, the system writes the balancing bitmap, derived from an algorithm. This is needed here because any measurement stops the balancing. It must be re-enabled at the earliest possible point to keep the balancing efficiency high. The cycle concludes by returning to TriggerNext, waiting out the remainder of the interval before recommencing.

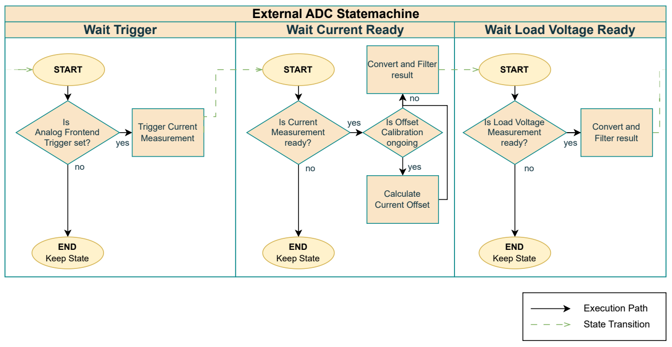

External ADC Statemachine
Wait Trigger
Wait Current Ready
Wait Load Voltage Ready
START
Is Analog Frontend Trigger set?
yes
Trigger Current Measurement
no
END Keep State
START
Convert and Filter result
Is Current Measurement ready?
yes
Is Offset Calibration ongoing
no
yes
Calculate Current Offset
no
END Keep State
START
Is Load Voltage Measurement ready?
yes
Convert and Filter result
no
END Keep State
Execution Path
State Transition

Fig. 8. Flow/Statechart of the external ADC measurement, used for current and load voltage.

Simultaneously, the external ADC triggers its current measurement process before entering the WaitCurrentReady state. Here, the ADC data is continuously polled until it is ready for use. During the initial setup, the current offset is also computed at this stage. The value is then converted, filtered and the system proceeds to the WaitLoadVoltageReady state, adopting an analogous approach for load voltage readings.

E. Diagnostic

The diagnostic system depicted in figure 9 is engineered to identify, log and manage errors, thereby defining the decision base for system state assessment and switch control. It continuously cycles through requesting, evaluating and clearing debug registers across all peripheral ICs using state machines. Errors and warnings are consolidated into bitmaps for streamlined communication and application. [1]

Diagnostic operations involve monitoring three primary error sources: safety switch driver registers, analog frontend registers and internal system checks. Each cycle involves refreshing these sources to capture the latest statuses. The associated errors are cleared from and, if detected again, reinstated in the error bitmap. Errors on the ICs are reset automatically where feasible.

The error bitmap is latched to preserve errors across cycles, ensuring they are retained for subsequent actions, such as external communication. Any detected errors trigger a BMS ERROR status, leading to the deactivation of the safety switch. Direct fault interrupts from peripheral ICs immediately initiate a safe state and deactivate outputs. Conversely, if no errors are found, the safe state is released.

Finally, the system's red LED behavior is determined: it remains on while errors exist and turns off when cleared. The LED serves as an immediate error indicator, while latched errors are displayed until reset, ensuring transient errors are logged even if the LED is no longer lit. If there are no errors and the State of Charge (SoC) drops below a critical level, the LED blinks.

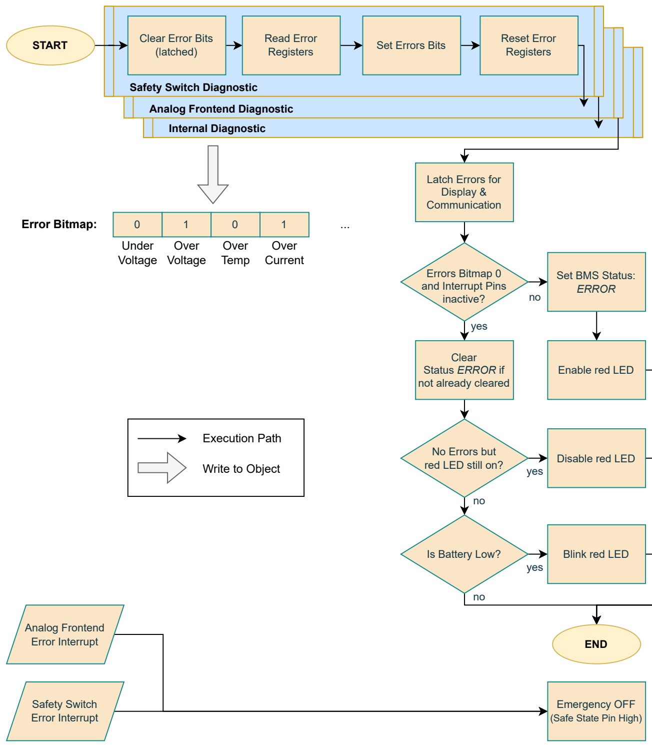

START
Clear Error Bits (latched)
Read Error Registers
Set Errors Bits
Reset Error Registers
Safety Switch Diagnostic
Analog Frontend Diagnostic
Internal Diagnostic
Error Bitmap:
0
1
0
1
...
Under Voltage
Over Voltage
Over Temp
Over Current
Latch Errors for Display & Communication
Errors Bitmap 0 and Interrupt Pins inactive?
no
yes
Set BMS Status: ERROR
Clear Status ERROR if not already cleared
Enable red LED
No Errors but red LED still on?
yes
no
Disable red LED
Is Battery Low?
yes
no
Blink red LED
END
Execution Path
Write to Object
Analog Frontend Error Interrupt
Safety Switch Error Interrupt
Emergency OFF (Safe State Pin High)

Fig. 9. The overview of the diagnostic concept.

F. System, Switch and Hot-Swap Control

The system must determine its operational state based on diagnostic outcomes and manage the safety switch (see figure 10). When detecting an ERROR state, it deactivates the switch and resets relevant components. If there are no errors, the BMS transitions its state to either DISCHARGING or CHARGING, depending on the detected circumstances. It then evaluates conditions for a potential hot-swap, which may also induce a system shutdown if the swap fails or concludes.

The key process during normal operation is the BMS Role state machine, determining the responsibilities of the BMS as either the primary power provider (MAIN BMS) or a participant in the hot-swap process (GIVER BMS or TAKER BMS). The coherences of the roles are provided in figure 11. Initially, the first BMS assumes the MAIN role, while the subsequent BMS takes on the TAKER role, remaining idle until a handover is initiated. The MAIN BMS controls the soft power of the switch by utilizing the precharge channel with a supplementary state machine. This switch-on state machine tracks the conditions for the switch-on and starts the precharge if needed. Once started, the full switch-on will commence after certain conditions are reached. Here, a minimal and maximal time sets up a boundary in which the change rate of the load voltage is traced to find the best switch on point.

If a swap is justified, The MAIN BMS transitions to the GIVER role, beginning a CAN communication handshake to transfer power supply responsibilities to the TAKER. During the hot-swap, the GIVER first confirms the TAKER's presence and prompts the load to reduce power demand to protect the hardware. Subsequently, the GIVER instructs the TAKER to activate its precharge path, which mitigates high compensation currents that could otherwise trigger safety features or cause damage (swap phase). The finish phase, triggered by the TAKER makes the GIVER power down its paths, signals completion to the TAKER, and shuts down. The TAKER then fully engages its main path, notifying the load that power restrictions can be lifted, and adopts the MAIN role.

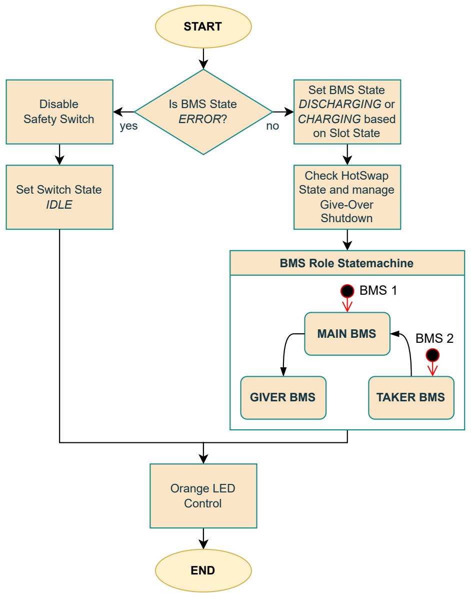


Fig. 10. Simplified flowchart of system and switch control.

Throughout the hot-swap process, timeouts are continuously monitored. If a timeout occurs, both the GIVER and TAKER will shut down to ensure system safety.

G. Data Recording

The F-RAM memory in the system is separated into two segments as can be seen in figure 12. The first segment records system status data such as the BMS's last State of Charge (SoC) and State of Health (SoH), updating only when new values diverge from those already stored. The second segment is allocated for real-time (RT) data accumulation during system operation.

Data selection for recording is tailored to parameters typically used in training AI algorithms for battery state prediction [11], emphasizing compactness by storing values in groups of two little-endian formatted fixed-point bytes. To achieve resource-efficient storage, only the interval between data points is recorded, which also serves to indicate recording stops when set to zero.

Current levels are not stored directly. Instead, the change in charge between data points is used, with the current calculable by dividing this charge difference by the corresponding time interval. A mechanism compensates for charge loss during the conversion from floating to fixed-point, ensuring overall charge accuracy over time upon data evaluation.

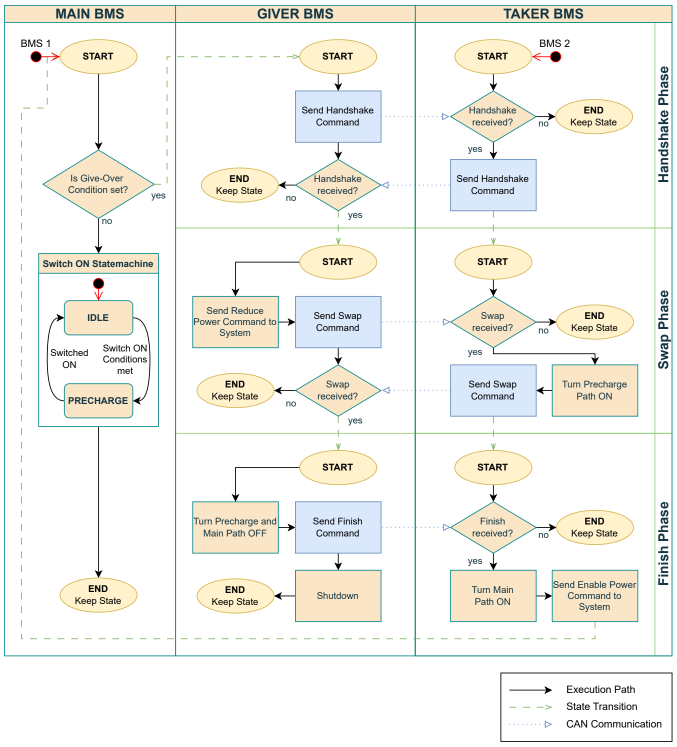


Fig. 11. Overview of the BMS Roles and how they change and interact with each other throughout the hot-swap phases.

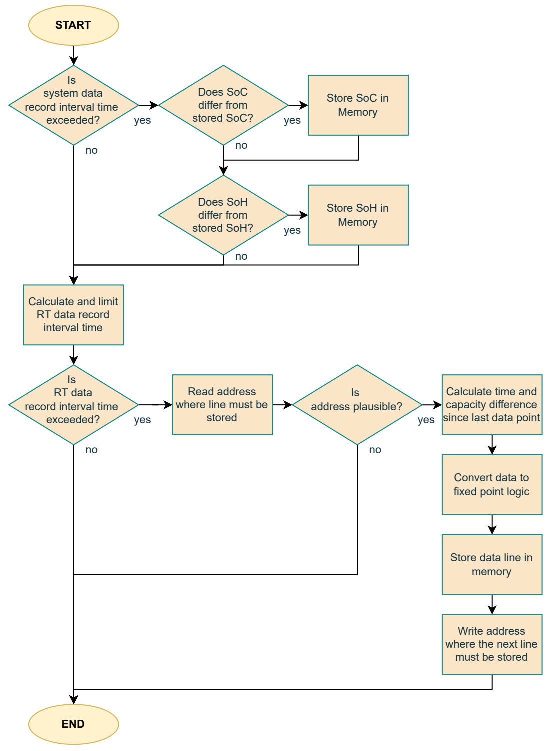

START
Is system data record interval time exceeded?
yes
no
Does SoC differ from stored SoC?
Store SoC in Memory
Does SoH differ from stored SoH?
Store SoH in Memory
Calculate and limit RT data record interval time
Is RT data record interval time exceeded?
Read address where line must be stored
Is address plausible?
Calculate time and capacity difference since last data point
Convert data to fixed point logic
Store data line in memory
Write address where the next line must be stored
END

Fig. 12. Flowchart of the data recording of system status and the real-time data. Both mechanisms act at different intervals.

Temperature readings use the average from sensors near the cells, while block voltage and imbalance figures are stored outright, totaling 10 bytes needed per record.

Recording RT data optimally involves capturing a complete battery cycle before the storage is full. The load conditions dictate the minimal sampling rate required: high current draws need shorter sampling intervals, whereas lower loads permit extended intervals. A dynamic sampling time estimation adjusts the sampling rate in real time, calculating it based on the predicted system runtime and remaining storage capacity to ensure a full cycle is documented, regardless of load variability.

H. *CAN Communication Interface*

The system communicates via CAN messages, allowing the exchange of status information, requests, and commands. To prevent message blockages due to bus errors, a re-initialization process is triggered by specific errors.

To ensure moderate bus usage, a minimum interval between outgoing messages is enforced, with status information and requests processed exclusively by the MAIN BMS. Additionally, CAN messages are employed during the hot-swap process, in the roles of either GIVER or TAKER.

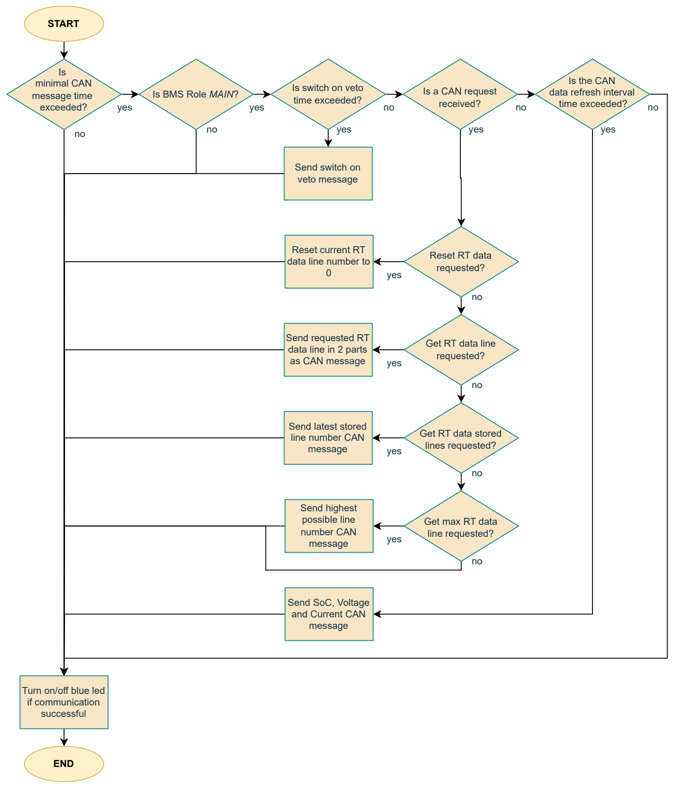

START
Is minimal CAN message time exceeded?
yes
no
Is BMS Role MAIN?
Is switch on veto time exceeded?
Is a CAN request received?
Is the CAN data refresh interval time exceeded?
Send switch on veto message
Reset RT data requested?
Reset current RT data line number to 0
Get RT data line requested?
Send requested RT data line in 2 parts as CAN message
Get RT data stored lines requested?
Send latest stored line number CAN message
Get max RT data line requested?
Send highest possible line number CAN message
Send SoC, Voltage and Current CAN message
Turn on/off blue led if communication successful
END

Fig. 13. Flowchart of the communication via CAN messages in BMS role *MAIN*. Only here, requests and continuous status updates are processed.

When conditions permit message transmission, three types of messages, as depicted in figure 13 are prioritized: SwitchOnVeto Messages from the MAIN BMS, responses to incoming requests and system status data sending. The

SwitchOnVeto Message has the highest priority, signaling other BMS units that a MAIN BMS is providing the load power and that no other must switch on. This is sent regularly. During the intervals, the system addresses responses to requests, followed by status data if no other message type needs to be done earlier.

Request responses reply to the requested data with the same command ID, usually without needing parameters. An exception to this is the RT_Data_GetLines command, which specifies StartIndex and StopIndex in two-byte increments. After validation, the requested data lines are transmitted, and split into two parts due to CAN message size constraints.

V. Results

To verify the system requirements, a series of tests and recordings are performed. [8] Recorded real-time data is visualized with the balancing target shown as a black line on the imbalance plot (relevant only in charge mode).

A. Over-Temperature Test

Discharging a full system on a constant load of $5\,\mathrm{A}$ triggers the over-temperature error at $60\,^{\circ}\mathrm{C}$ after $32\,\mathrm{min}$.

B. Voltage Limit Test

To assess the over/under-voltage error detection, the system is powered by a variable voltage supply in/decreasing the voltage until the threshold is exceeded. Both over and under-voltage errors triggered as expected.

C. Current Limit Test

Equipped with cells and connected to a digital variable load, the system's target current is manually increased until an over-current error is triggered at $5.9\,\mathrm{A}$ (figure 14), reducing the current to the BMS draw.

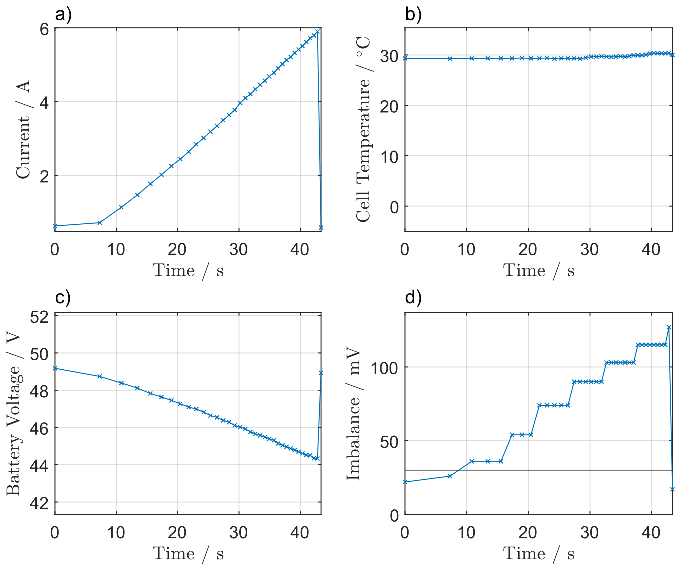


Fig. 14. A discharge cycle where the current is increased from $0\,\mathrm{A}$ to $7\,\mathrm{A}$ until an over-current error is triggered. Each measurement point is marked to visualize the increase in the sampling rate with the rising current.

D. Battery Cycle Test

As a simple test, a charge and discharge cycle is recorded. The data coherence between tests is evaluated to be good, though precise hardware benchmarks and battery process evaluations were not performed. The charge test (figure 15) showed an imbalance reduction through balancing systems, while the discharge test (figure 17 and 16) highlighted voltage drops and rising temperatures.

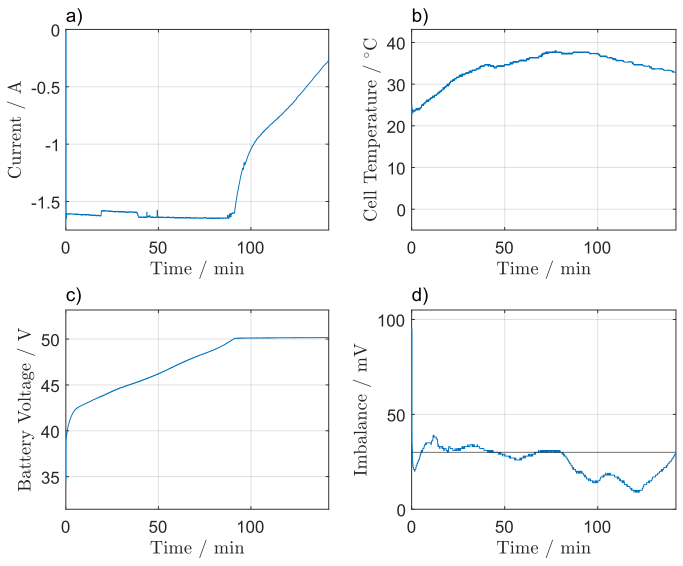


Fig. 15. A charge cycle where $3104\,\mathrm{mAh}$ are loaded into the battery in $142\,\mathrm{min}$ with a maximum charge current of $1.7\,\mathrm{A}$.

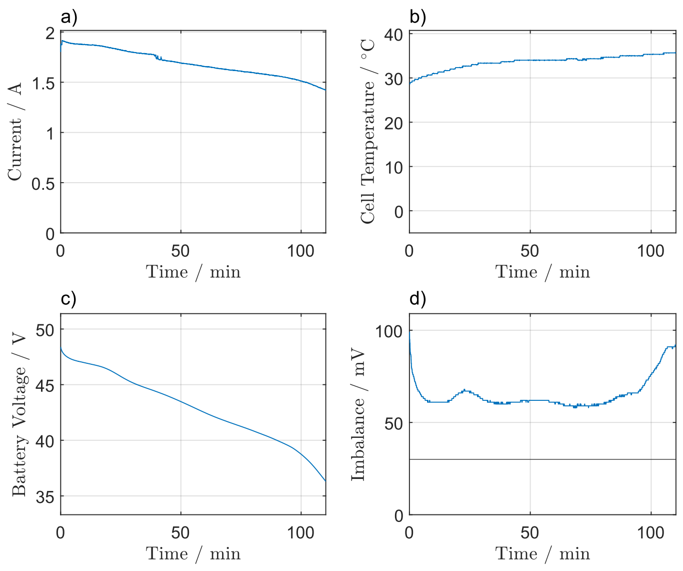


Fig. 16. A discharge cycle where $3103\,\mathrm{mAh}$ are taken from the battery in $110\,\mathrm{min}$. The load is set up to simulate a constant resistance of $26.6\,\Omega$. Therefore, the current drops with the voltage.

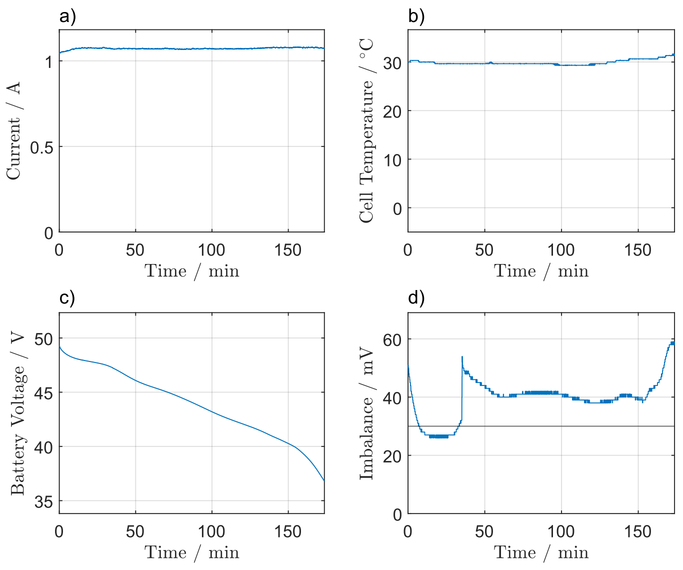


Fig. 17. A discharge cycle where 3103 mAh are taken from the battery in 173 min. The load is set up to keep the current constant at 1 A.

E. Precharge Evaluation

To test the precharge, the switch-on of a capacitive load is examined. The voltage change rate is set to the detectable minimum at 0.01 V/ms. In multiple tests, the minimal usable precharge time to not trigger any safety mechanisms is found and compared to the said rate. The initial current peak and the oscillations after switch-on are reduced greatly (figure 18).

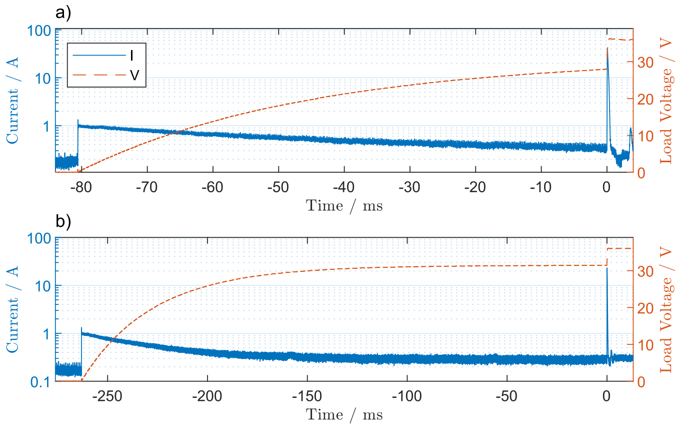


Fig. 18. Switch-on of a load with 915 µF and 330 Ω where current (solid line) and voltage (dashed) is measured. Plot (a) uses the minimal precharge time where switch-on is possible. Plot (b) shows the switch-on using the minimal detectable voltage change rate of the system.

F. Software Review

The software is designed to be fast, safe, reliable, extensible and maintainable. Speed is improved through a set of design principles and loop strategies. Safety is ensured by utilizing hardware capabilities and watchdogs. Extensibility is achieved by separating functional parts into dedicated files. Code maintainability is enhanced by centralizing settings and constants, and thorough documentation. The cycle time is measured to be around 1 to 2 ms.

VI. Conclusion

Presented is a comprehensive design, implementation and evaluation of a battery management system (BMS) in C, aiming to enhance robustness and functionality through a modular approach utilizing embedded software engineering principles. The implementation focuses on the system design, integration of multiple safety features, hot-swap capability, CAN-based communication interface and runtime data storage, which ensure reliability and safety. The evaluation chapter presents performance measurements and testing results, demonstrating the effectiveness of the implemented concepts. The work successfully develops a robust BMS platform for future evaluations of advanced battery management algorithms.

References

[1] S.Yang, X.Liu, S.Li, and C.Zhang, "Advanced Battery Management System for Electric Vehicles," Springer, 1 2023. [Online]. Available: https://link.springer.com/book/10.1007/978-981-19-3490-2

[2] H.Gabbar, A.Othman, and M.Abdussami, "Review of Battery Management Systems (BMS) Development and Industrial Standards," Technologies, vol. 9, no. 2, pp. 1–23, 2021. [Online]. Available: https://doi.org/10.3390/technologies9020028

[3] Infineon Technologies, "Infineon Mobile Robot (IMR) battery management system power," 04/2024. [Online]. Available: www.infineon.com/dgdl/Infineon-UG092042_mobile_Robot_IMR_battery_management_system_power_using_DEMO_IMR_BMSPWR_V1-UserManual-v01_00-EN.pdf

[4] Infineon Technologies, "Infineon Mobile Robot (IMR) battery management system control," 04/2024. [Online]. Available: www.infineon.com/dgdl/Infineon-UG091835_mobile_robot_IMR_battery_management_control_using_DEMO_IMR_BMSCTRL_V1-UserManual-v02_00-EN.pdf

[5] R. Santeler, M. Schmidt, P. Temsamani, "BMS via Radio - Documentation," MCI EAL in cooperation with Infineon AG, 2023.

[6] P.Temsamani, "Battery management via radio," 2022.

[7] O.Demirci, S.Taskin, E.Schaltz, and B. A.Demirci, "Review of battery state estimation methods for electric vehicles - Part I: SOC estimation,"Journal of energy storage, vol. 87, p. 111435, 2024. [Online]. Available: https://doi.org/10.1016/j.est.2024.111435

[8] A. S.Berger, "Embedded Systems Design: An Introduction to Processes, Tools and Techniques," CMP Books, 2001.

[9] Infineon Technologies, "TLE9012DQU Li-ion battery monitoring and balancing IC," 01/2022. [Online]. Available: www.infineon.com/dgdl/Infineon-TLE9012DQU-DataSheet-v01_00-EN.pdf

[10] Infineon Technologies, "TLE9012DQU, TLE9015DQU," 01/2022. [Online]. Available: www.infineon.com/dgdl/Infineon-Infineon-TLE9012DQU_TLE9015DQU-UM-v01_00-EN-UserManual-v01_00-EN.pdf

[11] E.Chemali, P. J.Kollmeyer, M.Preindl, and A.Emadi, "State-of-charge estimation of Li-ion batteries using deep neural networks: A machine learning approach," Journal of power sources, vol. 400, pp. 242–255, 2018. [Online]. Available: https://doi.org/10.1016/j.jpowsour.2018.06.104

Rene Santeler is a full system developer with a focus on embedded software development. After a decade-long career in IT, he studied mechatronics at the MCI where he also worked as part of the Emerging Applications Laboratory team in cooperation with Infineon.