SOFTWARE DEVELOPMENT OF A MULTILEVEL BATTERY SYSTEM FOR AN ELECTRIC AIRCRAFT

P. Panchal*, W. Bliemetsrieder†, N. Sorokina†, S. Myschik*
* Institute for Aeronautical Engineering, † Department of Electrical Engineering,
University of the Bundeswehr Munich, 85577 Neubiberg, Germany

Abstract

This paper presents the development of safety-critical battery controller software for an electrically powered glider in compliance with aerospace standards. The development process involves several mandatory tasks, such as requirements-based testing, ensuring traceability, and performing software verification and validation. The embedded software consists of multiple layers, which are discussed in this paper in the context of a multilevel battery control system. The multilevel battery system is designed for two primary applications: an electric glider and an unmanned lift-to-cruise vehicle. The functional algorithm for the battery control system is developed using MATLAB and Simulink, enhanced by the process-oriented build tool, mrails, ensuring adherence to aerospace standards. The functional code is subsequently linked to the operating system layer through the middleware layer. The A53 core of the S32G2 hardware, developed by NXP, is used for the battery main control application. A custom certifiable real-time operating system, µC/OS-II, providing time and space partitioning, hosts the application. Real-time testing of the components will be conducted on a modular hardware-in-the-loop test bench. This paper aims to present the integration of model-based functional code into the operating system in the context of safety-critical systems.

Keywords

DO-178C; DO-331; model-based software; battery control; process-oriented, s32g2, safety-critical

1. INTRODUCTION

In modern aircraft, numerous systems and functions that were traditionally mechanical are now increasingly controlled by software, as part of a broader trend toward more advanced avionics and fly-by-wire systems such as auto-pilot, brake-by-wire, landing gear control, FADEC, etc. Software plays a crucial role in such safety-critical systems, where failures can result in significant harm, including injury, environmental damage, or financial loss. These systems are prevalent in sectors like aviation, automotive, healthcare, nuclear power, and industrial automation. Many of these systems incorporate safety-critical software, which demands as much attention—if not more—than the hardware itself. In the aerospace sector, numerous examples exist, such as flight control systems, flight management systems, collision avoidance systems, autopilot systems, and mission control systems. Failures in such software have caused fatal accidents, eroding public confidence in air travel. However, certification authorities continually update guidelines to ensure the safety and security of this software. Not all software in these systems is equally critical, and a scale known as Design Assurance Level (DAL) is used to classify the levels of criticality. This scale ranges from A to E, where A represents the highest level of criticality and E denotes no criticality.

A set of standards are defined to ensure the software is safe enough for the humans and the environment. RTCA (Radio Technical Commission for Aeronautics) introduced the DO-178C software standard [1], which outlines a set of objectives that must be met based on the DAL levels. This standard provides guidelines for software development, verification, and validation throughout its lifecycle. DO-178C includes supplements for specific scenarios, such as DO-331 [2], which focuses on model-based development. These guidelines cover the development process from software requirements and architecture to code generation, verification, and validation.

Similarly, DO-254 [3] applies to hardware development and provides necessary guidance. Together, the software and hardware form an embedded system designed to meet specific system requirements, illustrating the connection between system development and the development of software and hardware components. To ensure the overall safety of system development, SAE International issued ARP4754 (Aerospace Recommended Practices) [4], a guideline emphasizing safety in the development of civil aircraft and systems.

Strict methodologies imposed by the standards reduces the flexibility of incorporating changes in requirements at a later stage, for example, adding a new feature after certification is expensive and efforts consuming. To address this challenge, a process-oriented build tool, mrails, has been developed and successfully utilized in several complex flight control and avionics software development projects at both the Institute of Flight System Dynamics at the Technical University of Munich and the Institute for Aeronautical Engineering at Universität der Bundeswehr München [6,7]. The mrails tool supports the development of safety-critical applications using a model-based approach with MATLAB, Simulink, and Stateflow, covering various stages of software development [8].

This particular research focuses on the further development and integration of the functional software for a multilevel battery control system. The software is integrated in a certifiable real-time operating system as an user application.

doi: 10.25967/630363

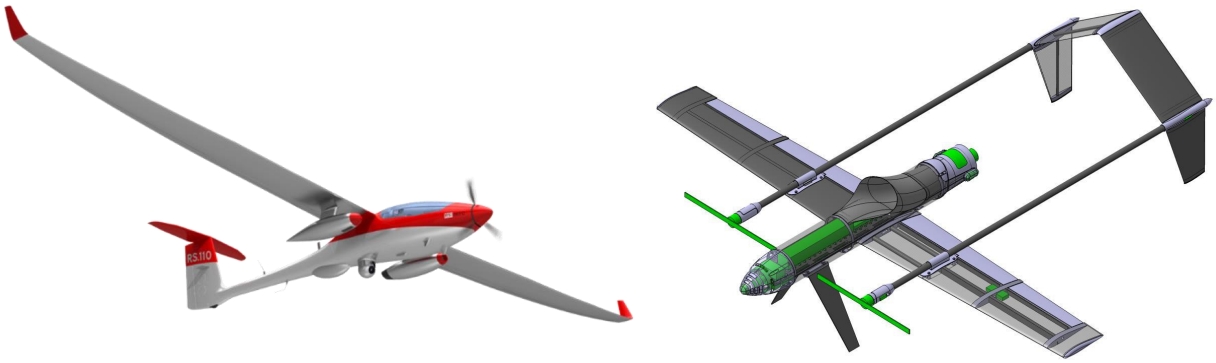

FIG 1. Left: ReinerStemme Aero RS 110 [5], Right: Lift-to-Cruise eVTOL Demonstrator

The main aim is to setup the workflow of integrating model-based generated code as an user application which can be later loaded on the flash memory of the hardware. The paper is organized into following sections: section 2 explains the aircraft applications for which the battery control system is being developed, whereas section 3 describes the multilevel battery control software. Section 4 provides an overlook of the software development and verification toolchain implemented in this research. Section 5 explains the integration of functional software into the µC/OS-II operating system.

2. AIRCRAFT SYSTEMS

The multilevel battery system will be applied to two different aerial systems, as shown in Fig. 1, an electric glider and an unmanned lift-to cruise vehicle (eVTOL). The electric glider has an maximum take-off weight of approx. 820 kg, max. 80 kW propulsion power generated by EMRAX 268 with a clipped 4-blade Helix propeller [9], a wingspan of 20 m and a maximum glide ratio of 43.

The aircraft is intended to employ a novel propulsion system involving the multilevel battery system, as shown in Fig. 2. The eVTOL system is under development and is intended to be used for demonstrating the capabilities of a novel propulsion system comprising a ducted e-Fan with counter-rotating blades. The eVTOl has a 2.1-meter wingspan and an electric propulsion system comprising two propellers and one impeller. It has been specifically designed for observation missions and is equipped with thermal and optical camera systems onboard. The central propulsion unit can rotate up to 90°, while the propellers have deflection angles ranging from 0° to 105° for hovering flight. In hover mode, yaw and roll control are managed by the propellers, while pitch control involves all propulsion units. The vehicle can reach a cruising speed of 30 m/s, using either the propellers or impeller, allowing flexibility for extended observation scenarios at lower airspeeds.

3. MULTILEVEL BATTERY SYSTEM

Multilevel systems are a class of battery management systems, such as the BM3 system discussed by Baver-tis [10, 11]. The half-bridge system is based on an integrated 2-switch inverter topology with MOSFET switches [12]. This topology offers several advantages, including flexible interconnections between battery cells to optimize efficiency, match load voltage requirements, extend battery lifespan, and enhance system fault tolerance. The module operates in two states: serial and bypass. Bypassing defective cells improves battery pack longevity and serves as a safety feature. The dynamic configuration of the modules enables the realization of these advantages. Figure 3 provides an overview of the battery pack and its controllers. The multilevel battery management system comprises a main controller, several BM3 modules, and individual cell controllers for each module. Together, the main controller and cell controllers form the battery controller. The main controller gathers crucial information, such as the current state of each cell (temperature, voltage, and current), error states from the switches, the battery pack's current output, and the DC voltage required by the motor controller. Based on these inputs, the main controller determines the necessary connection configuration for the BM3 modules and generates

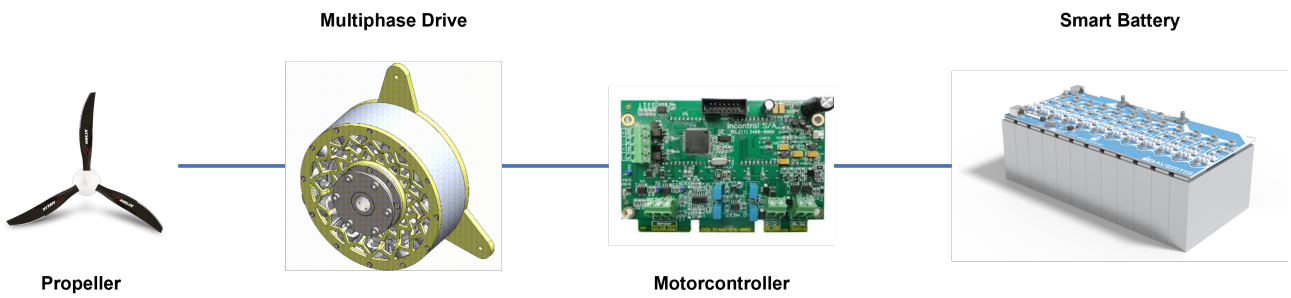

FIG 2. Electric Drive Train System

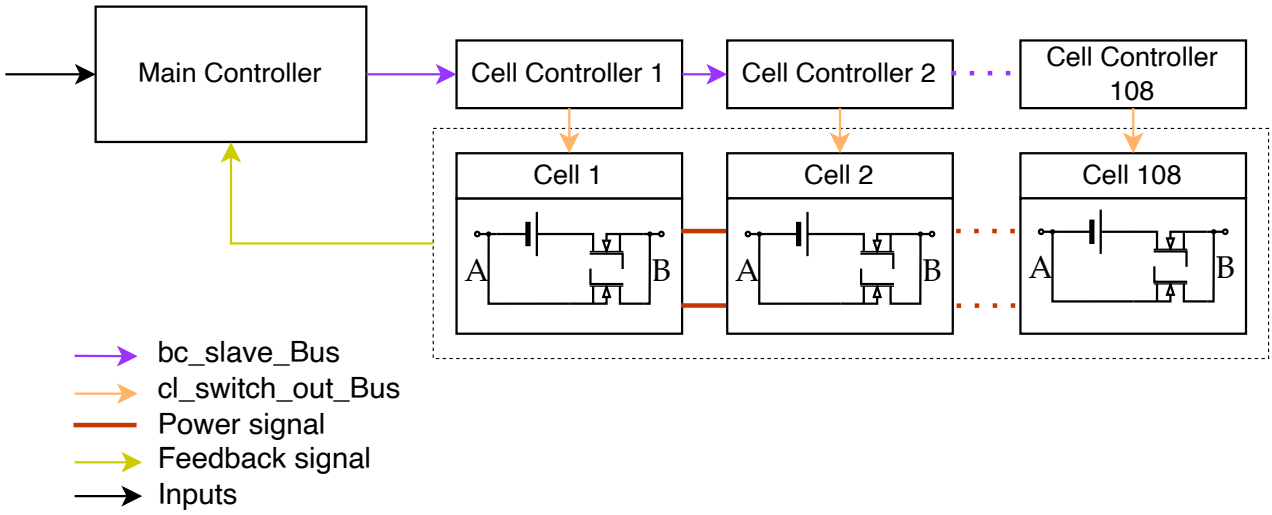

FIG 3. Top-Level Battery Controller Architecture

a configuration array that specifies the configuration values for each module. Development of the battery cell controller software is discussed by the author in the conference paper [13]. This configuration is transmitted to the first cell controller through the 'bc_slave_Bus', which also includes the module ID. As previously mentioned, there are two possible states: a series state, represented by the value 1, and a bypass state, represented by 0. The configuration is then forwarded via the 'cl_switch_out_Bus'.

4. SOFTWARE DEVELOPMENT TOOLCHAIN

The safety-critical battery controller software employs a process-oriented build tool alongside custom and commercial tools to ensure compliance. Figure 4 illustrates the V-model of development, indicating necessary tools at each stage. Implementing the V-Model enhances software safety, enabling verification and validation at different stages, aligning with DO-178C/DO-331 standards. This approach aids in early defect detection and supports model-based development, maximizing systematic benefits and fostering collaboration within distributed teams. Design models utilize the process-oriented build tool, mrails, automating tasks and code generation while aligning with DO-331 guidelines. The toolchain prioritizes agility for requirement changes and testing, with source management and continuous integration capabilities.

The process begins with defining stakeholder or customer requirements, which in turn shape the project's objectives and system requirements. Siemens Polarion PLM [14] is used to document system requirements, software and hardware requirements, test cases, development plans, as well as change control and issue tracking. Hardware development follows a similar process as software, though it is beyond the scope of this research. High-level software requirements, derived from system requirements, are linked to the software architecture model created using MathWorks System Composer. Low-level software requirements are represented by design models developed

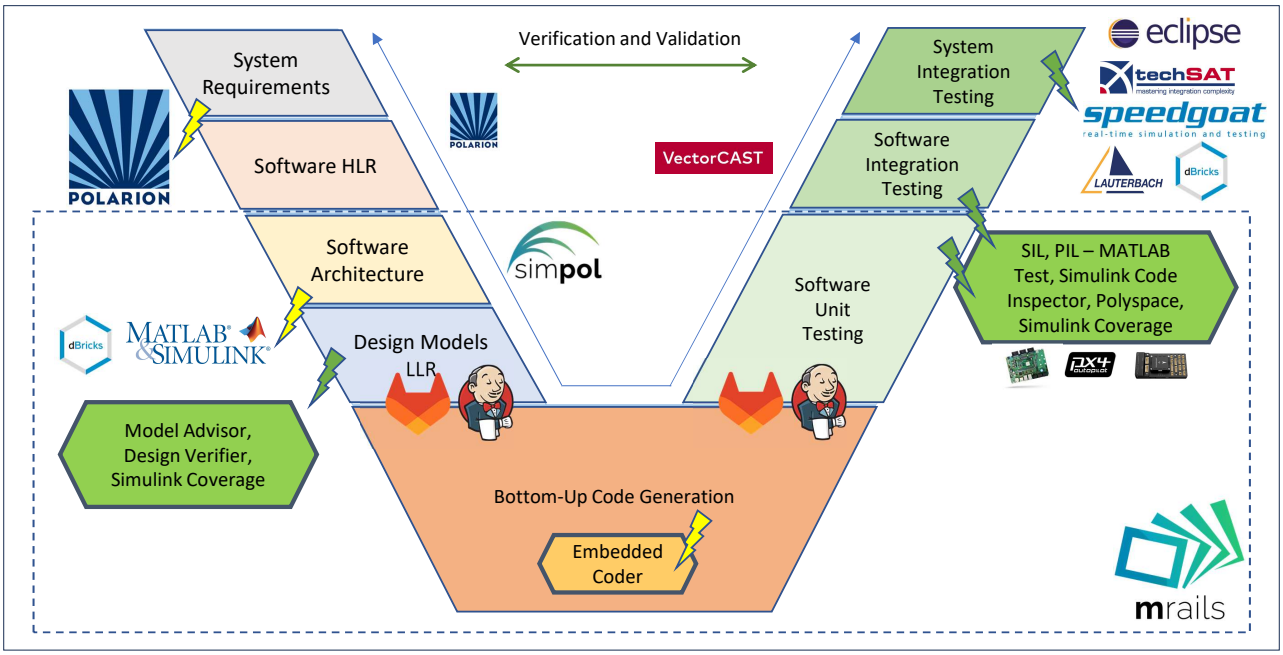

FIG 4. Software Development V-Model with Tools

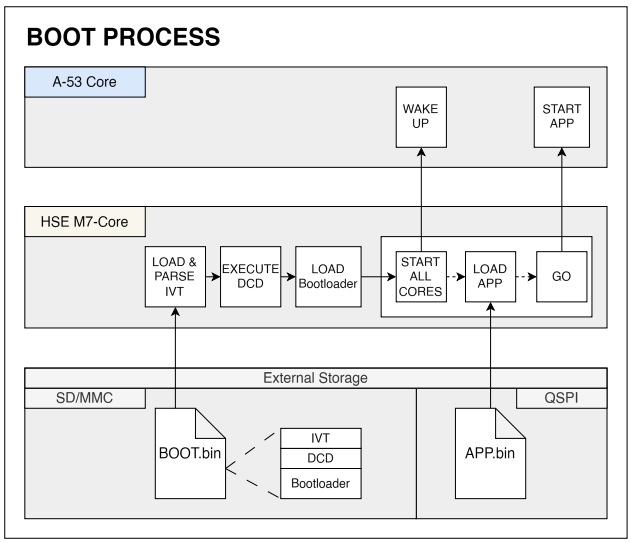

FIG 5. S32G2 Boot Sequence

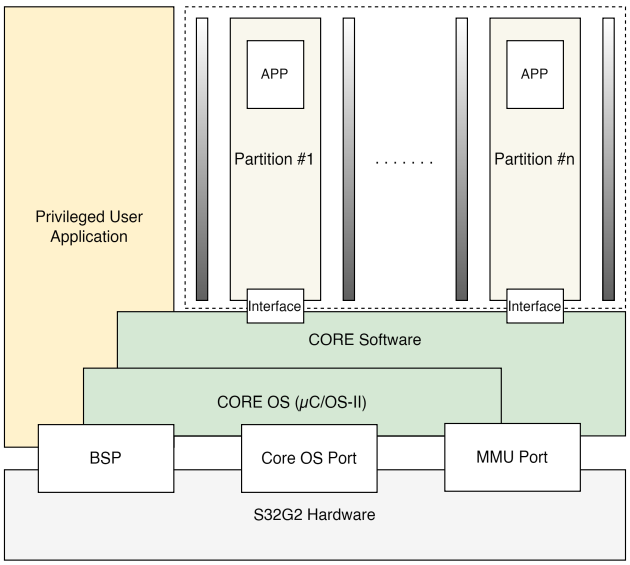

FIG 6. RTOS Software Architecture with Partitioning [18]

in MATLAB/Simulink. The design models are created using the process-oriented build tool, mrails, which provides a modular, model-based development framework in MATLAB/Simulink. This tool automates tasks such as generating code for the functional part using Embedded Coder, in compliance with DO-331 guidelines, and performing static model analysis. For the application software, low-level requirements are documented in text form in Polarion, while the corresponding code is manually written in Eclipse IDE [15]. After code generation, static testing is carried out with Polyspace, and simulation testing is performed with Simulink Tests for the functional part and VectorCAST [16] for the application part. The functional software is then integrated into the application framework within the Eclipse project and tested in real-time using a hardware-in-the-loop (HIL) testbench. Interfaces between software and hardware are managed using dBricks [17], an interface management tool. Various tools ensure traceability throughout the process, from requirements to testing.

5. BATTERY CONTROLLER SOFTWARE DEVELOPMENT

As shown in Fig. 3, the battery controller system consists of two components: the battery main controller (BMC) and the battery cell controllers (BCC). The BCC software is developed using the same development workflow but for different hardware. The BCCs use the STM32L431 hardware device with an Arm Cortex-M4 core. The functional software for the BCC is integrated into the hardware as a bare-metal application. This research primarily focuses on the development of the BMC software and its architecture. Following subsections explains the complete software stack starting from the booting till the application code.

5.1. S32G2 Application Boot Process

Developed by NXP, S32G2 hardware is a sophisticated processor that offers robust safety features and high performance. The S32G2 consists of four Arm Cortex-A53 cores and three Arm Cortex-M7 lockstep cores. Due to the safety-critical application,the hardware consists of several steps to finally get the application running. It must be noted that, for the BMC application, single-core is used to avoid the memory interference issue of multicore technology. There are various types of boot processes available for the S32G2 [19], Fig. 5 explains the implemented booting sequence for the development board S32G274A from MicroSys [19, 20]. A custom board is being developed in parallel, which will be used for the production purpose. The S32G2 contains a booting mechanism with a possibility of performing either a secure boot or a non-secure boot, with the help of the HSE (Hardware Security Engine) firmware. In a non-secure mode, the firmware passes control to the customer software running on the processor outside of HSE subsystem as shown in Fig. 5, whereas in the secure mode, the boot mechanism passes control to the HSE firmware which is running on the HSE_H subsystem. In this research, the non-secure booting mechanism is selected which directly triggers the A-53 core.

In this research, a custom U-Boot (Universal Boot Loader) is used provided by the manufacturer of the development board [21]. This bootloader is loaded from the external storage, a micro-SD card, and serves the purpose of configuring the pins, SoC, clock and the SDRAM. As soon as the board is powered on, the HSE reads the boot image (BOOT.bin), copies the code image to the SRAM and starts the core responsible for running the application, A-53 in this

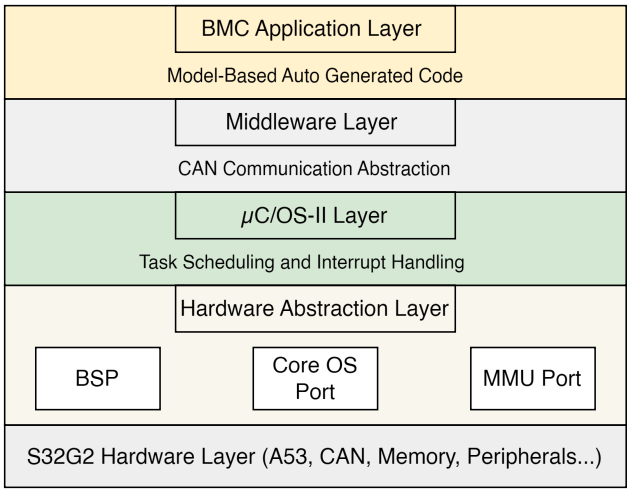

FIG 7. BMC Software Stack

case. If the bootloader is not available, the HSE doesn't start any of the cores and all the cores remain in reset [22]. The U-boot then loads the application binary which is flashed on the QSPI into the SRAM at the start address for the A-53 core along with the program counter of the processor.

5.2. BMC Software Architecture

The application loaded from the QSPI on the SRAM is explained in this subsection. The A53 core of the S32G2 hardware device is used for the Battery Management Controller (BMC) application [20]. Software development for this device is supported by S32 Design Studio [15], which is also utilized in this research. A certifiable version of µC/OS-II, along with the necessary I/O drivers (UART, CAN and Ethernet) for this hardware, is provided by an external vendor. Figure 6 shows the general embedded software architecture of application binary loaded by the bootloader. This architecture is referenced by a proprietary document of Embedded Office GmbH [18]. At the foundation is the S32G2 Hardware Layer, which includes the A53 cores, CAN controllers, memory, and various peripherals that form the physical components of the system. This layer interacts directly with the hardware components, providing the computational resources and communication interfaces required for the application.

Above the hardware layer is the Hardware Abstraction Layer (HAL), which consists of the BSP (Board Support Package), the Core OS Port, and the MMU Port. This layer abstracts the hardware specifics, offering an interface that simplifies hardware access for higher-level software. The BSP initializes and manages essential hardware components such as the CAN controller, UART, and timers, while the Core OS Port ensures that the real-time operating system (µC/OS-II) can function on the S32G2 hardware. The MMU (Memory Management Unit) port handles memory protection and mapping, critical for managing the address space effectively in a multitasking environment.

5.2.1. Real-Time Operating System

The µC/OS-II layer sits atop the hardware abstraction layer, providing the real-time operating system functionalities. This includes task scheduling, interrupt handling, and timing services that enable the real-time execution of tasks and the deterministic performance required by safety-critical systems like the BMC. It ensures that various tasks, including CAN communication and battery management operations, meet their real-time constraints.

Memory-Management Unit: The µC/OS-MMU extension provides an environment where applications can be implemented with the insurance, that no user application can disturb or corrupt any other user application. This insurance is called "Time and Space Partitioning" and is realized by the MMU extension [18].

5.2.2. Middleware

Middleware layer provides communication abstractions that simplify the interaction between the application and lower-level hardware, such as CAN controller. This layer simplifies interaction with the communication bus by abstracting lower-level details, enabling the application to focus on processing data rather than managing communication intricacies. To understand the middleware layer, it is necessary to develop a base knowledge of the software call hierarchy. The Fig. 8 demonstrates the call hierarchy of the Battery Management Controller (BMC) application within the µC/OS-II real-time operating system. It begins with the main() function, which initializes the system, setting up essential hardware components and preparing the environment. Following system initialization, the Board Support Package (BSP) is initialized through BSPInit(), establishing the necessary hardware abstractions. The PARStart() function then starts the operating system and sets up the Memory Management Unit (MMU) for efficient memory handling. After the system and OS setup, the application is created in privileged mode through AppStartup(), allowing the BMC to access critical resources. The application starts

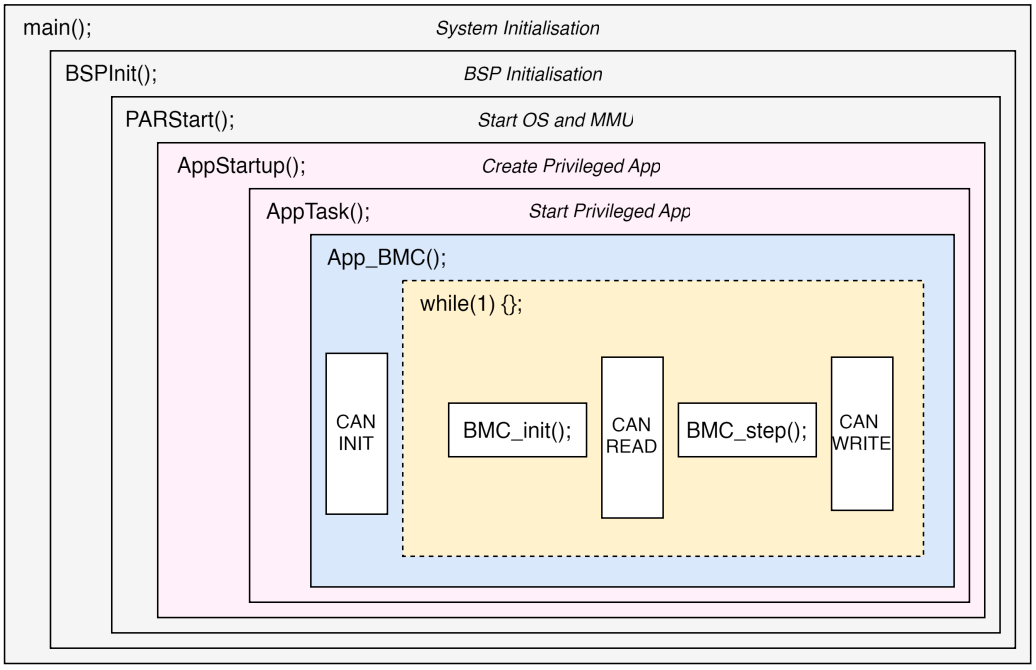

FIG 8. BMC Application Call Hierarchy in µC/OS-II

running via AppTask(), which transitions control to the main BMC logic in App_BMC(). Inside the App_BMC() function, the BMC application runs in an infinite loop (while(1)), performing core tasks like initializing the battery management system (BMC_init()), processing CAN messages (read and write), and executing its main processing step (BMC_step()). This call hierarchy ensures that the BMC runs continuously and interacts efficiently with the system's CAN network to monitor and control battery operations in real time.

The Fig. 9 represents the workflow of the Battery Management Controller (BMC) application, showcasing the integration between the CAN drivers and middleware. The process begins with the initialization of the CAN controllers, queue, and the installation of a callback function that is triggered when a CAN message is received. Upon receiving a message, the callback reads the message from the CAN driver and checks if the message queue is full. If the queue is not full, the message is enqueued, and the queue counter is incremented. If the queue is full, the message is discarded.

Once the system is initialized, the BMC enters a continuous main loop where it checks the queue for messages. If the queue is not empty, messages are dequeued, the queue counter is decremented, and the messages are processed by assigning their data as inputs to the BMC. When the queue is empty, the BMC continues with its control logic, executing its step function to process the system's current state and compute outputs. These outputs are used to create CAN frames, which are transmitted back through the CAN driver.

Interface Management Tool: The tool dBricks is used to manage the interfaces within the component. It integrates with Simulink, allowing data buses to be imported from the data dictionary to Bricks and also to be exported into the Simulink. The tool contains the standard communication protocols, such as ARINC 429 and 825, with the help of which the port contents of CAN controllers can be defined. A code generator is available to translate the data from ICD, such as dBricks, into code files with the help of templates. This automatic code generation facilitates the middleware code development process. Currently, the processing of CAN frames is written manually, however, in future the code generator will be used to generate encoding and decoding of the CAN frames to communicate with the application code.

5.3. BMC Functional Software

At the top of the stack lies the BMC Application Layer, which contains the model-based auto-generated code (BMC initializing and step functions in Fig. 9). This code is generated by the Embedded Coder using MATLAB/Simulink, which is then integrated into the embedded software environment. The application logic governs the management and monitoring of the battery system, leveraging inputs from CAN messages to execute real-time control algorithms. The outputs of this layer are used to create CAN frames that are transmitted to the cell controllers and the motor controller.

Figure 10 illustrates the top-level Simulink model used for the code generation of the BMC. It consists of two components (model references): ma_soc and ma_config. The SOC model receives the state of charge, current,

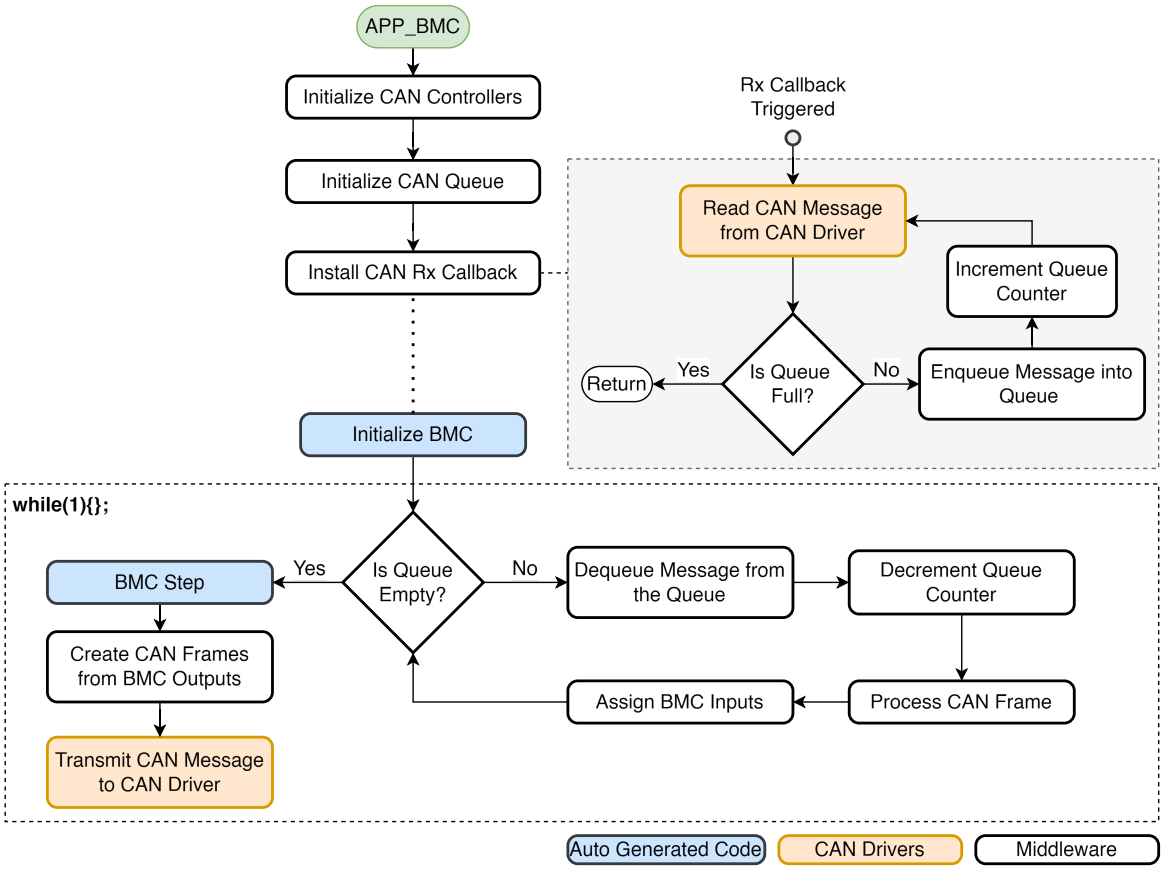

FIG 9. BMC Functional Code and Middleware Code Integration

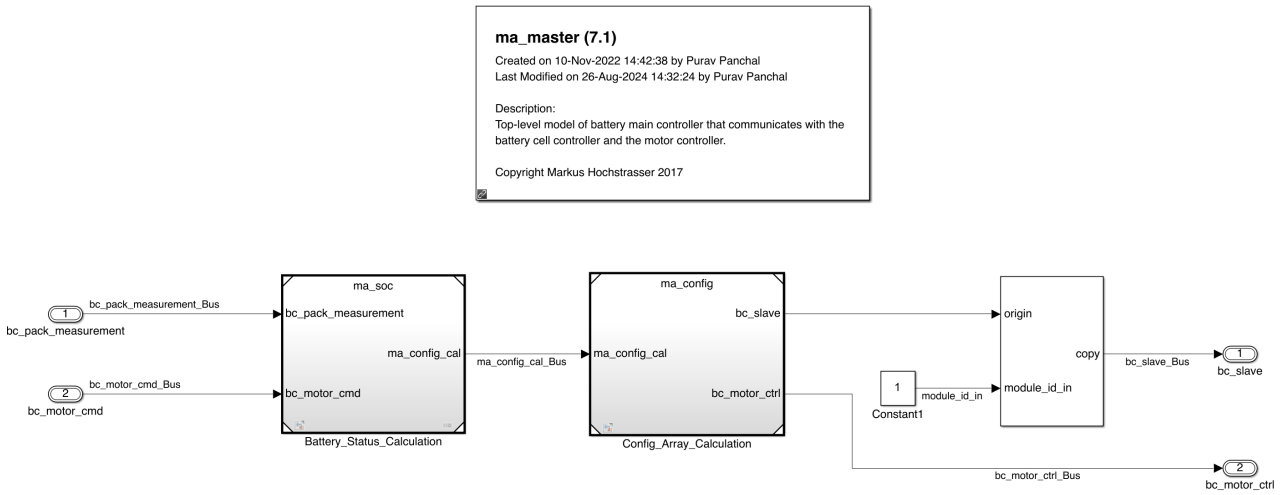

FIG 10. Top-Level BMC Simulink Model

and voltage data from all 108 battery cells, as well as the required voltage command from the motor controller. It then calculates the number of cells that need to be active to meet the requested voltage command. This information is passed to the second model, which determines which specific cells should remain active based on their state of charge. The configuration index is then sent to the battery cell controllers, which activate the cells according to the configuration array. The actual battery pack voltage and state of charge are communicated back to the motor controller. The CAN buses operate in duplex mode, meaning the BMC sends and receives data from the BCCs on one bus. Similarly, a second bus communicates with the motor controller, transmitting and receiving voltage commands and battery pack status.

The process-oriented build tool, mrails, is used to assist the designing and code generation process by its automated jobs. The generated code is then packaged and integrated into the S32DS project. Figure 11 shows the step function code snippet generated by the Embedded Coder.

```
/* Model step function */
void ma_master_step(void)
{
  /* local block i/o variables */
  bc_slave_Bus b_bc_slave_Bus;

  /* ModelReference: '<Root>/Battery_Status_Calculation' incorporates:
   *  Inport: '<Root>/bc_motor_cmd'
   *  Inport: '<Root>/bc_pack_measurement'
   *  Outport: '<Root>/ma_config_cal'
   */
  ma_soc(&ma_master_U.bc_pack_measurement_Bus_ekcy,
         &ma_master_U.bc_motor_cmd_Bus_ktdj, &ma_master_Y.ma_config_cal);

  /* ModelReference: '<Root>/Config_Array_Calculation' incorporates:
   *  Outport: '<Root>/bc_motor_ctrl'
   *  Outport: '<Root>/ma_config_cal'
   */
  ma_config(&ma_master_Y.ma_config_cal, &b_bc_slave_Bus,
            &ma_master_Y.bc_motor_ctrl);

  /* Outport: '<Root>/bc_slave' incorporates:
   *  BusAssignment: '<Root>/BusAssign'
   */
  ma_master_Y.bc_slave = b_bc_slave_Bus;

  /* BusAssignment: '<Root>/BusAssign' incorporates:
   *  Constant: '<Root>/Constant1'
   *  Outport: '<Root>/bc_slave'
   */
  ma_master_Y.bc_slave.module_id_in = 1;
}
```

FIG 11. BMC Step Function Generated By Embedded Coder

6. CONCLUSION

In this paper, a comprehensive overview of the software development process for the Battery Main Controller (BMC) in a multilevel battery management system is presented. The development began with a review of aircraft systems and their relevance to the multilevel battery system, highlighting the critical role of safety and efficiency in these contexts. The software development toolchain is explained, detailing the tools and methodologies used throughout the project, including model-based design and code generation using Simulink, supported by tools such as S32 Design Studio for the S32G2 hardware platform.

The Battery Controller Software Development section delved into the key aspects of implementing the BMC software on the S32G2 hardware, with a focus on the S32G2 application boot process, the software architecture of the BMC, and the functional software that manages key operations. The integration of a certifiable real-time operating system, µC/OS-II, is explained with the memory-management unit, emphasizing the importance of reliability and safety in embedded systems for battery management. Furthermore, the role of the middleware is explained, illustrating how these components facilitate the CAN communication between the application software developed by Simulink and the underlying hardware. The middleware is currently manually developed and follows a certain workflow of enqueuing and dequeuing the CAN messages to extract the frames and assign data to the input of the BMC step function. Similarly, CAN frames are created from the output of the step function and then written to the hardware.

Through the structured explanation of each software component, the paper demonstrates how the BMC software ensures safe and efficient battery management. The combination of advanced hardware, real-time operating systems, and model-based development tools has enabled a robust, scalable solution for managing the complex requirements of multilevel battery systems, particularly in aerospace applications. This development framework provides a foundation for further research and optimization of battery management systems, addressing future challenges in energy storage, safety, and performance in real-time embedded environments.

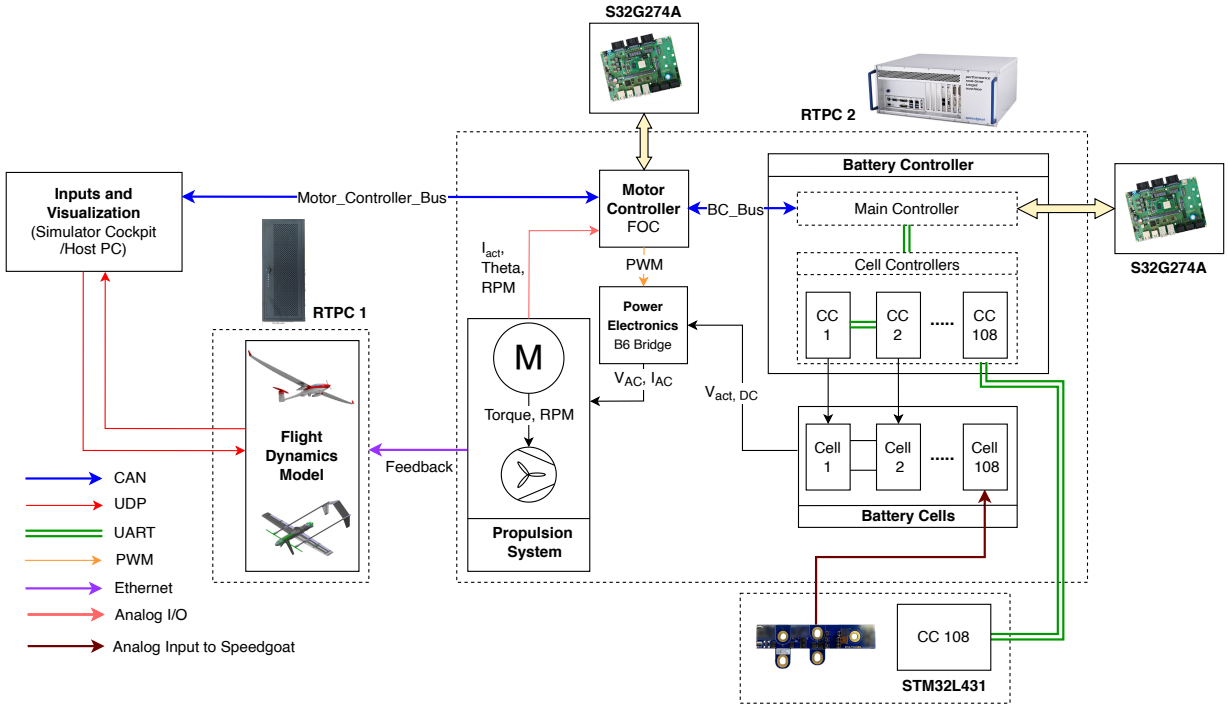

FIG 12. Hardware-In-The-Loop Testbench

7. FUTURE WORK

This research has established a preliminary basis for the development of other safety-critical applications for the project ELAPSED. The BMC itself is yet to be tested in hardware-in-the-loop (HIL) simulation as shown in Fig. 12. The functional requirements of the BMC will be then verified using the HIL simulations. The current HIL setup offers a powerful platform for simulating real-time battery management scenarios. Future work could focus on expanding the complexity and scope of simulations to incorporate more advanced fault injection, extreme operating conditions, and thermal management strategies. This would allow for deeper validation of the BMC's robustness and resilience in critical situations, such as cell failure or overcurrent scenarios. Additionally, the integrated flight dynamics models would enable closer simulation of actual in-flight conditions for battery and propulsion systems.

Along with the BMC application, other applications such as fault handling, charging, discharging will be integrated as tasks in partitions, leveraging the time and space partitioning provided by the MMU. The communication between the BMC and cell controllers via UART and CAN buses is functional; however, optimizing the latency and bandwidth usage of these communication protocols is to be performed. More efficient data handling between the BMC, BCCs and the motor controller can reduce communication overhead, especially during high-frequency control updates, ultimately improving response times and energy efficiency in real-time operation. Additionally, the current standard CAN will be migrated to the aerospace standard ARINC 825.

In real-time embedded systems like the BMC application, ensuring that tasks meet their deadlines is crucial for system reliability and safety. WCET analysis is a method used to determine the maximum time a task or piece of code can take to execute under worst-case conditions. This is particularly important for safety-critical systems, where missing a deadline could lead to system failure or unsafe conditions. The tools for performing static and dynamic WCET analysis is already been procured which will assist in determining the critical execution time, ensure real-time performance, optimize resource usage and provide safety.

ACKNOWLEDGEMENTS

This research and project ELAPSED are funded by dtec.bw – Digitalization and Technology Research Center of the Bundeswehr. dtec.bw is funded by the European Union – NextGenerationEU [23].

Contact address:

purav.panchal@unibw.de

References

[1] DO-178C. Software Considerations in Airborne Systems and Equipment Certification. Standard, RTCA, 2011.

[2] DO-331. Model-Based Development and Verification Supplement to DO-178C and DO-278A. Standard, RTCA, 2011.

[3] DO-254/ED-80. Design Assurance Guidance for Airborne Electronic Hardware. Standard, RTCA, EUROCAE, 2000.

[4] SAE. Arp 4754: Certification considerations for highly-integrated or complex aircraft systems, 2023.

[5] Rs.aero projects: Light sports aircraft elfin 20, rs.100 series, rs.500. Website, 2024. https://reinerstemme.aero/projects/.

[6] Markus Hochstrasser, Stephan Myschik, and Florian Holzapfel. A process-oriented build tool for safety-critical model-based software development. *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, 2018. DOI: 10.5220/0006605301910202.

[7] Markus Tobias Hochstrasser. *Modular model-based development of safety-critical flight control software*. Dissertation, Technische Universität München, München, 2020.

[8] Konstantin Dmitriev, Shanza Ali Zafar, Kevin Schmiechen, Yi Lai, Micheal Saleab, Pranav Nagarajan, Daniel Dollinger, Markus Hochstrasser, Florian Holzapfel, and Stephan Myschik. A lean and highly-automated model-based software development process based on do-178c/do-331. In *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, pages 1–10, 2020. DOI: 10.1109/DASC50938.2020.9256576.

[9] Carbon fiber propellers - helix-carbon gmbh. Website, 2024. https://helix-propeller.de/.

[10] Bavertis. 3 in 1 power electronics. https://bavertis.com/, Jul 2022.

[11] Manuel Kuder, Julian Schneider, Anton Kersten, Torbjoern Thiringer, Richard Eckerle, and Thomas Weyh. Battery modular multilevel management (bm3) converter applied at battery cell level for electric vehicles and energy storages. In *PCIM Europe digital days 2020; International Exhibition and Conference for Power Electronics, Intelligent Motion, Renewable Energy and Energy Management*, pages 1–8, 2020.

[12] Anton Kersten, Manuel Kuder, Emma Grunditz, Zeyang Geng, Evelina Wikner, Torbjörn Thiringer, Thomas Weyh, and Richard Eckerle. Inverter and battery drive cycle efficiency comparisons of chb and mmsp traction inverters for electric vehicles. In *2019 21st European Conference on Power Electronics and Applications (EPE '19 ECCE Europe)*, pages P.1–P.12, 2019. DOI: 10.23919/EPE.2019.8915147.

[13] Purav Panchal, Nina Sorokina, Stephan Myschik, Konstantin Dmitriev, and Florian Holzapfel. Application of a process-oriented build tool to the development of a bm3 slave controller software module. *DGLR DLRK 2022 Forum*, Sept. 2022. DOI: 10.25967/570308.

[14] Siemens. Application Lifecycle Management (ALM), Requirements Management, QA Management | Polarion - Software — polarion.plm.automation.siemens.com. https://polarion.plm.automation.siemens.com/, 2004. [Accessed 28-Oct-2022].

[15] NXP. S32 Design Studio IDE — nxp.com. https://www.nxp.com/design/software/development-software/s32-design-studio-ide:S32-DESIGN-STUDIO-IDE, 2023. [Accessed 27-10-2023].

[16] Vector. VectorCAST - Software Test Automation for High Quality Software. https://www.vector.com/int/en/products/products-a-z/software/vectorcast/, 2023. [Accessed 27-10-2023].

[17] dBricks. dBricks — peerss.ru. https://peerss.ru/en/dbricks/, 2023. [Accessed 27-10-2023].

[18] Efficiency in safety software development. Website, 2024. https://www.embedded-office.com/.

[19] NXP Semiconductors. S32g2 reference manual, 2021. Rev. 5. https://www.nxp.com/products/processors-and-microcontrollers/s32-automotive-platform/s32g-vehicle-network-processors/s32g2-processors-for-vehicle-networking:S32G2.

[20] MicroSys. Sbc s32g274a arm architecture | MicroSys — microsys.de. https://www.microsys.de/en/products/sbc/arm-architecture/miriac-sbc-s32g274a/, 2023. [Accessed 22-May-2023].

[21] U-boot / u-boot · gitlab. Website, 2024. https://source.denx.de/u-boot/u-boot.

[22] Architecture-specific notes > arm cortex > how-to guides > cold start with nxp s32g/s32r45x cortex-m7 and a53 core. Website, 2024. https://www.isystem.com/downloads/winIDEA/help/nxp-s32r45-g2-3-cold-start.html.

[23] DTEC. Electric Aircraft Propulsion – die zukunft der flugzeugantriebe, 2021. Accessed: 2022-04-11. https://dtecbw.de/home/forschung/unibw-m/projekt-elapsed.