Decoupling UI Logic in Embedded Systems: Technical Design of R-MVP-based Thermostat Software

Alexander Menzel
Department of Electrical Engineering
Fulda University of Applied Sciences
Fulda, Germany
alexander.menzel@et.hs-fulda.de

***Abstract*—Intelligent heating control can significantly reduce residential energy consumption, yet the market is dominated by proprietary solutions with limited transparency for research and development. This paper presents the software architecture for a radiator thermostat prototype, developed as part of the interdisciplinary MiraTherm project at Fulda University of Applied Sciences. The central contribution is the Routed-Model-View-Presenter (R-MVP) design pattern, an adaptation of the classical MVP pattern for resource-constrained embedded systems. R-MVP introduces a dedicated Router component that centralizes page navigation and input event forwarding, enabling state-driven UI transitions without relying on the widget-level input abstractions assumed by classical MVP. The software is implemented in C using FreeRTOS on an STM32WB55 microcontroller and employs the LVGL graphics library for display rendering. A modular, task-based architecture separates hardware abstraction, control logic, and user interface into distinct FreeRTOS tasks communicating via event queues and mutex-protected shared data structures. Verification through integration testing and source code inspection confirms that all functional requirements are met and the architecture successfully decouples UI logic from hardware drivers and core system logic.**

***Index Terms*—embedded software architecture, Model-View-Presenter, design pattern, radiator thermostat, smart home, FreeRTOS**

I. Introduction

Heating private living spaces is one of the most significant sources of CO_2 emissions. In Germany, a substantial portion of annual greenhouse gas emissions originates from this sector [1], [2]. While intelligent heating control and smart home systems offer an average energy saving potential of between 8 and 19% [3], the market is currently dominated by proprietary solutions. Consequently, there is a lack of open-source projects that can serve as a foundation for research and development of smart heating controllers.

This paper presents the software development for a radiator thermostat prototype, realized as part of the interdisciplinary "MiraTherm Radiator Thermostat" project at Fulda University of Applied Sciences. The overarching project aims to create a complete device, encompassing mechanics, electronics, and control algorithms.

The primary objective of this work is to establish a solid software foundation for the thermostat's microcontroller-based hardware. While the long-term vision includes control algorithms and wireless connectivity (e.g., Matter-over-Thread), the central contribution of this paper lies in the architectural design of the application and its User Interface (UI). Specifically, we propose the application of the Routed-Model-View-Presenter (R-MVP) design pattern to decouple UI logic from hardware drivers and core system logic. This approach is used to implement basic consumer functions considering constraints of an embedded system and of a verbose programming language.

II. Background

This section provides the necessary context for this work. First, the domain of smart radiator thermostats is introduced. Then, the Model-View-Presenter (MVP) design pattern, which forms the architectural basis for the software, is described.

A. Smart Radiator Thermostats

A radiator thermostat is a device mounted on a heating radiator valve to control the flow of hot water, thereby regulating the room temperature. Traditional thermostatic radiator valves (TRVs) use a wax or liquid-filled capsule that expands and contracts with temperature changes, mechanically adjusting valve position. Digital radiator thermostats replace this mechanism with an electronically controlled system, typically consisting of a microcontroller unit (MCU), a DC motor with a gearbox for valve actuation, a user interface (buttons, rotary encoder, display), and a motor driver circuit.

Smart radiator thermostats extend the digital concept with wireless connectivity, such as Wi-Fi, Bluetooth Low Energy (BLE), or emerging standards like Matter, enabling integration into smart home ecosystems and remote control via mobile applications. Common consumer functions of such devices include manual temperature adjustment, mode selection (e.g., auto, manual, boost, vacation), weekly schedule programming, an open-window detection function, a periodic valve decalcification program, and others [4], [5]. They are most commonly battery-powered and are expected to operate for extended periods, typically around two years on two AA batteries [4], [5].

The software presented in this paper uses the eQ-3 eqiva Bluetooth Smart Radiator Thermostat [4] as a functional reference for defining its scope.

B. MVP Pattern

The Model-View-Presenter (MVP) pattern is a derivative of the classic Model-View-Controller (MVC) pattern [6]. MVP is described in [6] as a generalization of MVC that decomposes an application's design into two fundamental domains: *Data Management* and *User Interface*.

In the Data Management domain, a developer must address three sub-questions:

1) *What is my data?* — answered by the **Model**, which encapsulates the application's data and its access methods;
2) *How do I specify my data?* — answered by **Selections**, which represent abstractions for identifying subsets of the model's data;
3) *How do I change my data?* — answered by **Commands**, which represent operations that can be performed on selections [6].

In the User Interface domain, three further questions are posed:

4) *How do I display my data?* — answered by the **View**, which renders a representation of the model's data (Views need not be graphical.);
5) *How do events map into changes in my data?* — answered by **Interactors**, which handle user-initiated actions such as mouse clicks, keystrokes, or physical input like turning a dial;
6) *How do I put it all together?* — answered by the **Presenter**, which coordinates all other elements by providing the business logic that maps user gestures onto the appropriate commands for manipulating the model [6].

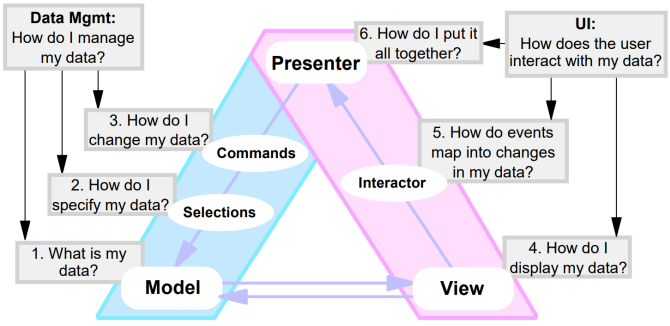

Fig. 1. The six design questions of the MVP programming model and their corresponding abstractions [6].

A key insight from [6] is that these abstractions need not all be employed at once. A developer can start with only a subset of the MVP elements—such as the Model, View, and Presenter—and introduce others if the application requires their respective benefits, such as undo/redo capability, scriptability, or advanced input handling.

This partial adoption is particularly relevant for the embedded context of this work, where the constrained environment and the use of a verbose programming language favor a minimal yet well-structured architecture. Accordingly, the design of an application can deliberately omit the Command, Selection, and Interactor abstractions, retaining only the core Model, View, and Presenter triad as the foundation of its UI architecture.

Figure 2 illustrates data flow in this simplified programming model. The View and Presenter are decoupled from each other, with the Presenter serving as the sole mediator between the View and the Model.

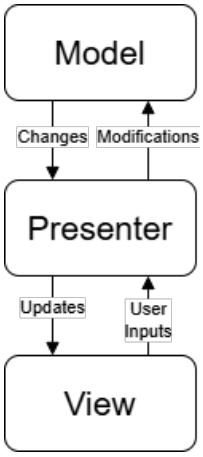

Fig. 2. Data flow in the MVP programming model (mod. from [7], [8]).

III. Requirements

The software requirements for the thermostat prototype are defined in [9] as a separate specification document. This section provides a condensed overview of the key functional and non-functional requirements that drive the technical design.

The functional requirements specify the integration of hardware peripherals shown in Figure 3. A display, a rotary encoder, three buttons, and a motor driver must be connected to the STM32WB55 MCU on its development board. Measurement of temperature, motor current, and supply voltage (`V_BAT`) must be performed.

The thermostat must support three operational modes: *Auto*, *Manual*, and *Boost*. In Auto mode, the target temperature follows a configured daily schedule; in Manual mode, the user sets it directly; and Boost mode provides a temporary override with a countdown display.

A Configuration on Device (COD) routine is required at startup, guiding the user through date/time setup, daily schedule programming, and a valve adaptation procedure.

The home display page must directly reflect the current thermostat state (e.g., current time, battery level, target temperature). A menu provides access to schedule changes, temperature offset configuration, and factory reset. All user settings must persist in non-volatile memory across power cycles.

The non-functional requirements mandate a modular architecture with hardware abstraction layers to allow easy replacement of individual hardware components (e.g., display, motor driver, temperature sensor) with minimal code changes. The UI

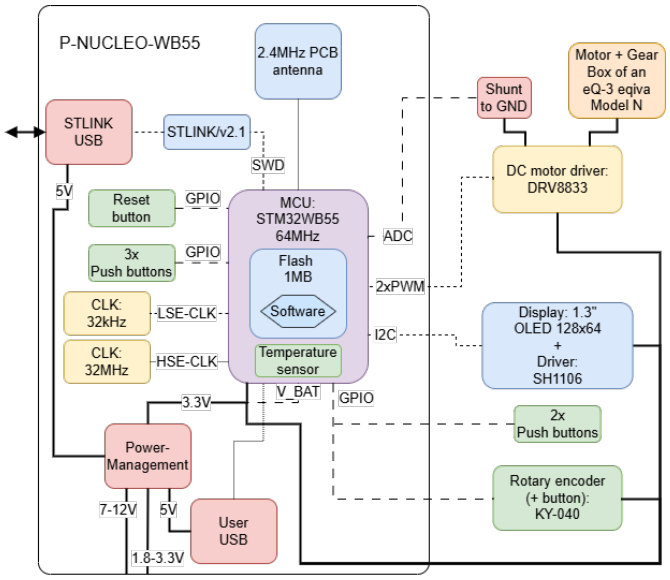

Fig. 3. Block diagram of the thermostat's prototype hardware for software development and testing.

must provide contextual button hints on every page, indicating the function assigned to each physical button.

IV. Technical Design

This section presents the software architecture and the adapted design pattern used for the UI implementation. Both were also described during the development process in [10]. The design addresses the requirements from section III by establishing a modular, task-based structure that separates hardware interaction, control logic, and user interface.

A. Overall Architecture

The software is written in C and uses FreeRTOS for real-time task scheduling and resource management. The architecture follows a modular design that separates hardware abstraction, core control logic, and UI management into distinct FreeRTOS tasks. Each task runs as a singleton and is started at system boot. The architecture is designed to enable future integration of control algorithms and the Matter standard with minimal effort. Figure 4 shows the overall architecture.

1) Synchronization and Data Structures: Tasks communicate using FreeRTOS event queues and mutex-protected data structures. All synchronization primitives are created and passed to tasks as parameters before the scheduler is started, preventing deadlocks and priority inversion.

Event queues provide one-way communication between task pairs for signaling and synchronizing state changes (e.g., initialization completion, adaptation requests). Bidirectional communication between two tasks uses two dedicated queues.

Shared data is organized into mutex-protected *Model* structures, each encapsulating a mutex together with the data it protects:

- **SensorModel**: Latest sensor readings (temperature, battery state of charge, motor current).
- **ConfigModel**: User-configurable settings, automatically persisted to non-volatile memory on every update.
- **SystemModel**: Volatile runtime data (system state, current operational mode, target temperature, active time slot, etc.).

2) Application Layer: The application layer consists of the following tasks:

- **SystemTask**: Central coordinator managing the high-level state machine. It orchestrates transitions between operational modes, assigns the target temperature, and communicates with other tasks via events (e.g., `EVT_SYS_INIT_END`, `EVT_ADAPT_REQ`).
- **InputTask**: Converts hardware signals from the rotary encoder and buttons into logical input events (e.g., `EVT_MENU_BTN`, `EVT_CTRL_WHEEL_DELTA`) forwarded to the ViewPresenterTask.
- **ViewPresenterTask**: Manages the UI using the R-MVP pattern (see subsection IV-B). It receives input events, accesses shared models, and renders UI elements via the LVGL graphics library.
- **SensorTask**: Periodically reads raw sensor data (via ADC/DMA), converts it to internal units, and updates the SensorModel.
- **StorageTask**: Manages non-volatile memory through an EEPROM emulation layer, ensuring configuration persistence.
- **MaintenanceTask**: Handles non-standard operations such as valve adaptation and descaling, triggered by commands from the SystemTask.

3) Hardware Abstraction Layer: A self implemented Hardware Abstraction Layer (HAL) provides interfaces for the display (LVGL display port for SH1106), the motor driver (IN1/IN2 interface for DRV8833), the rotary encoder (quadrature decoder), and the push buttons (interrupt-driven). This layer satisfies the replaceability requirement by isolating hardware-specific code from application logic.

B. R-MVP Pattern

1) Motivation for MVP over MVC: The classical MVC pattern does not enforce a strict decoupling between Model and View—the View can directly observe and access the Model, which makes it difficult to develop these components independently. MVP addresses this by introducing the Presenter as an intermediary: the View only communicates with the Presenter, and the Presenter mediates all access to the Model [11]. This separation is the primary reason for choosing MVP over MVC as the basis for the thermostat's UI architecture.

2) Challenges in Embedded Context: Classical MVP relies on rich, automated UI widgets (Interactors) that generate high-level semantic events—an assumption common to modern application frameworks, including desktop environments (Windows, Taligent's CommonPoint [12]), mobile platforms (Android [13]), and web frameworks [14]. In the bare-metal embedded context of this work, no such widget infrastructure is available. User input must therefore be handled manually, closer to the approach taken in MVC.

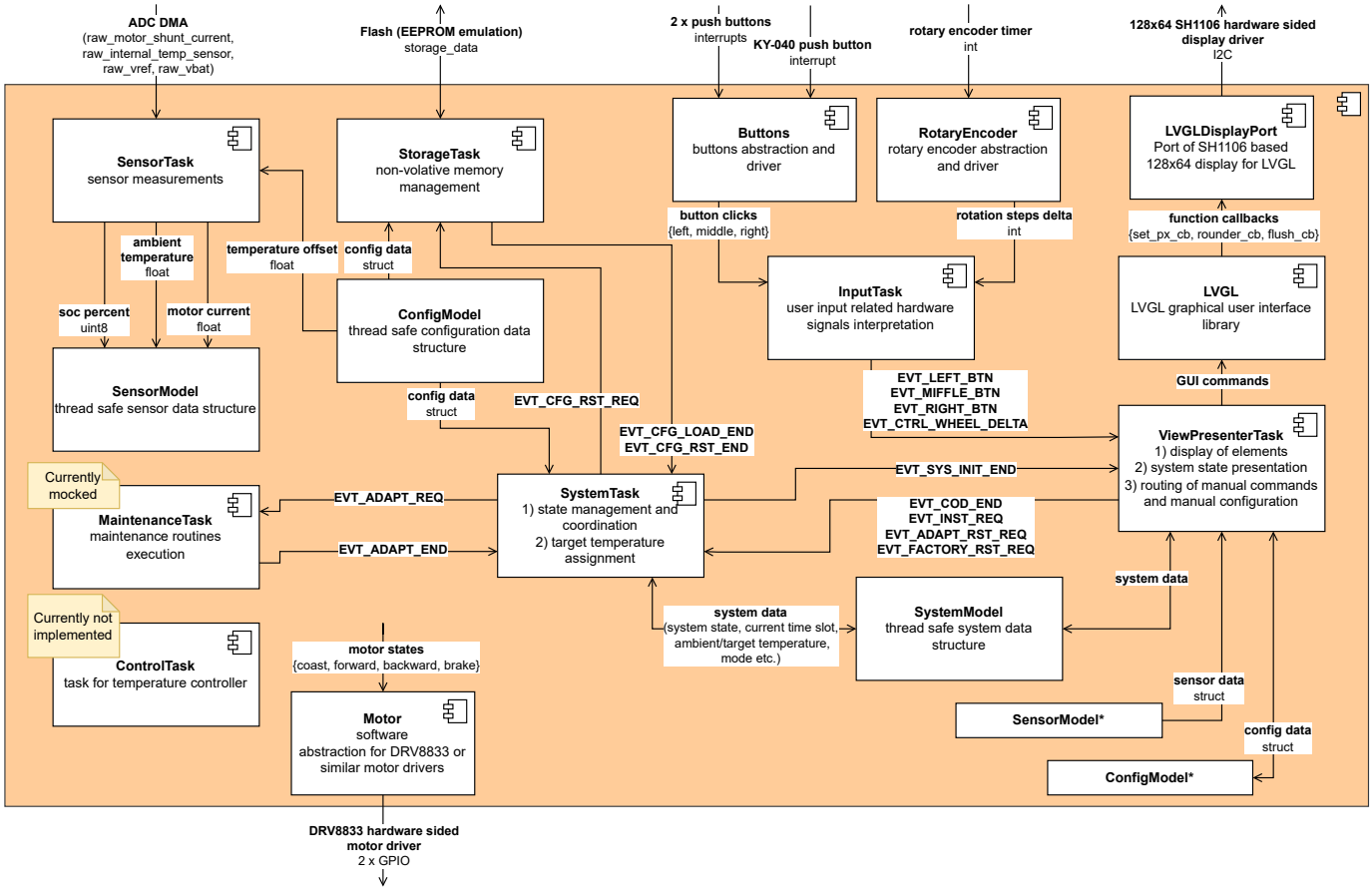

Fig. 4. Architecture of the radiator thermostat software.

While the LVGL graphics library does offer an encoder input device abstraction [15], the interaction model of this library is primarily designed for touchscreen or mouse/keyboard interfaces. Integration of an encoder in LVGL—based on widget focus groups, edit modes, and long-press transitions—proves too complicated for quick configuration routines of a simple thermostat. The predefined gesture semantics do not match the custom navigation behavior required by the application (e.g., context-dependent button actions, quick wizard-style page flows).

Furthermore, certain system state changes must provoke specific page transitions or restrict user interactions—for instance, during an ongoing adaptation procedure or while the SystemTask awaits completion of the Configuration on Device (COD). This requires navigation to be partially state-driven rather than purely user-driven as in classical MVC or MVP.

3) The R-MVP Solution: To address these challenges, the presented architecture introduces a dedicated *Router* component that receives input events from the InputTask and forwards them to the active presenter, enabling full control over gesture interpretation without relying on LVGL's native input callbacks. The Router also reads the current system state from the SystemModel to determine which page to display, supporting state-driven navigation alongside user-driven interactions.

The resulting pattern is referred to as Routed MVP (R-MVP). In the presented architecture, its entire logic is encapsulated within the ViewPresenterTask.

4) R-MVP Components: The pattern comprises four component types:

- **Router**: Manages screen navigation and page transitions based on the system state read from the SystemModel. It initializes and deinitializes presenters for each page, forwards input events from the InputTask to the currently active presenter, and can send events to the SystemTask to initiate state transitions.
- **Presenters**: Handle presentation logic and user interactions. They receive input events from the router, read and write shared models, and instruct views to update display elements. Presenters can be nested to implement multi-step wizard workflows.
- **Views**: Pure rendering components using LVGL, responsible solely for updating graphical elements on the display using data provided by the Presenters.
- **Models**: The shared, thread-safe data structures (SensorModel, ConfigModel, SystemModel) described in subsection IV-A, passed to the ViewPresenterTask as parameters.

For simplicity and to reduce abstraction overhead in the em-

bedded C environment, no explicit interfaces are used; presenters call views directly. Since page content differs significantly across screens, this does not introduce code duplication.

5) R-MVP Data Flow: The data flow follows a path through four stages:

1) The InputTask generates input events and sends them to the Router via an event queue.
2) The Router receives input events and forwards them to the currently active Presenter.
3) The active Presenter processes the event, reading from or writing to the shared Models (with mutex protection).
4) The Presenter instructs the corresponding View to render the updated data on the display.

This can be illustrated in Figure 5, which shows the data and control flow between the components of the R-MVP pattern. Comparing this flow with the classic MVP flow in Figure 2, it can be seen that only the path of user input has changed, which now leads from the Router to the Presenter instead of from the View.

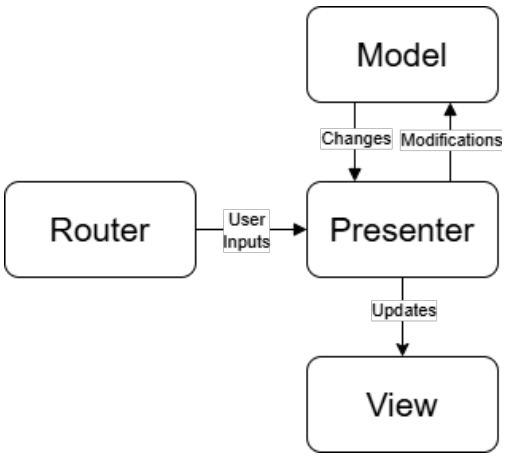

Fig. 5. Data flow in the R-MVP programming model.

6) Key Differences from Classical MVP: Three key deviations distinguish R-MVP from the classical pattern:

- **Centralized Router**: Instead of presenters managing navigation between views, a dedicated Router component centralizes page navigation logic and input event forwarding. Routing decisions are partially state-driven, based on the current system state read from the SystemModel, rather than being purely user-driven as in classical MVP.
- **Separated Input Handling**: Input events do not originate from the views (as they would via LVGL callbacks in a classical approach) but are received by the Router from the InputTask and forwarded to the active presenter. This bypasses LVGL's built-in input device model and enables custom gesture interpretation tailored to the rotary encoder and button hardware.
- **Hierarchical Structure**: Presenters are initialized and deinitialized by the Router based on the active page; nested presenters are initialized by their parent presenters; views are initialized by their corresponding presenters.

V. Implementation

In this section, the main challenges encountered during implementation are discussed, along with the solutions adopted to address them.

A. Interaction of ViewPresenterTask with SystemTask

Implementation of a partially state-driven UI requires tight synchronization between the ViewPresenterTask and the SystemTask in certain phases. Their interaction points can be clearly identified in the diagram of the SystemTask state machine, shown in Figure 6.

The ViewPresenterTask interacts with the SystemTask through two mechanisms: event queues for discrete notifications (e.g., `EVT_SYS_INIT_END`, `EVT_COD_END`) and direct reading/writing of the SystemModel for state-dependent routing and display updates.

During initialization, the ViewPresenterTask waits for `EVT_SYS_INIT_END` from the SystemTask before rendering. After that, the SystemTask waits for `EVT_COD_END` to signal the completion of Configuration on Device. Beyond initialization, the router reads the system state directly from the SystemModel to determine which pages to render, without requiring further events from the SystemTask.

B. Boilerplate Code in Router

Since a verbose programming language (C) is used, the Router contains a significant amount of boilerplate code for initializing and deinitializing presenters, as well as for forwarding input events. This could be substantially reduced by using a language with advanced abstraction features, such as Rust, which is well-suited for embedded systems [16] and typically provides zero-cost abstractions with no runtime overhead through monomorphization (e.g., generics and trait-based polymorphism) [17]. However, the decision to use C was made based on the target hardware's ecosystem and the desire for maximum compatibility with existing embedded development tools (e.g., X-CUBE-MATTER) and libraries (e.g., LVGL).

VI. Evaluation

The implemented software was verified against the requirements defined in [9] using two complementary methods: integration testing on the target hardware and source code inspection. The complete test scenarios, inspection criteria, and detailed results are documented in [18].

Integration testing was performed on the prototype hardware shown in Figure 3. The tests were divided into two groups: *driver tests*, which verify the correct operation of individual HAL components (see "Hardware Abstraction Layer" in subsection IV-A), and *integration tests*, which validate the application-level functionality including the R-MVP-based UI (subsection IV-B), the SystemTask state machine with its ViewPresenterTask synchronization (subsection V-A), and non-volatile storage. For requirements not practically verifiable

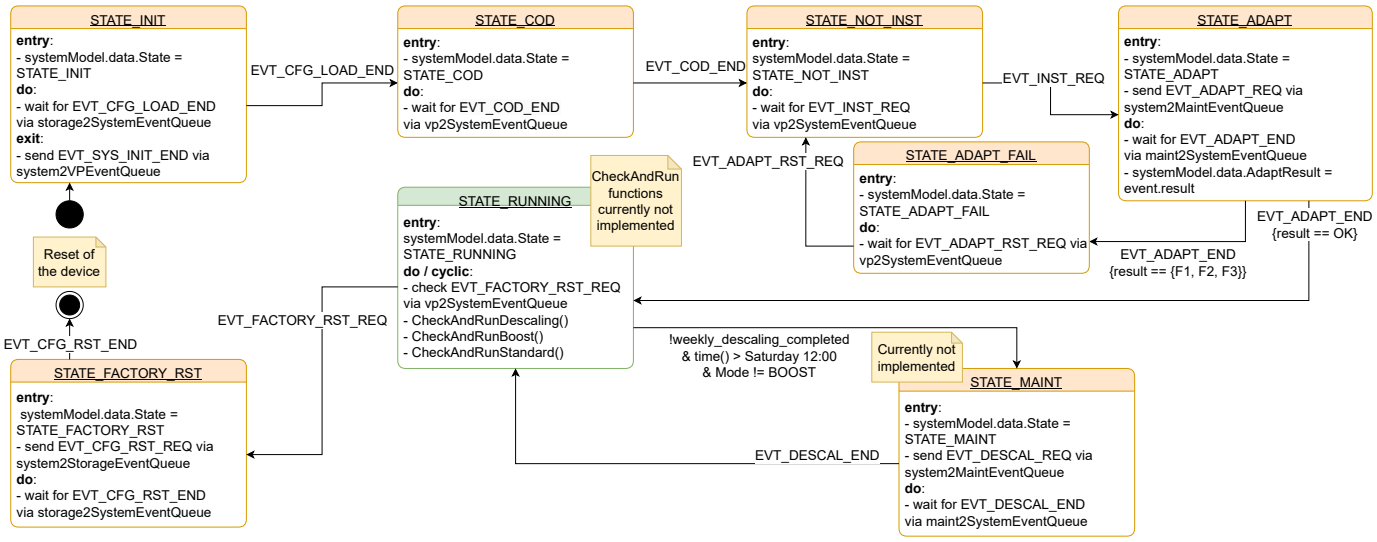

Fig. 6. State machine of the SystemTask

through functional testing, source code inspection was performed.

A. Results

All four driver tests and all eight integration tests passed successfully. Table I summarizes the results with their requirement coverage.

TABLE I
SUMMARY OF DRIVER AND INTEGRATION TEST RESULTS

ID	Name	REQ	Status
Driver Tests			
1	Display & Measurements	1.x, 5.x, 6.x, 8.x	Passed
2	Rotary Encoder	2.x	Passed
3	Buttons	3.x	Passed
4	Motor Driver	4.x, 5.x	Passed
Integration Tests			
5	Home Display Page	14, 31	Passed
6	Temp. Range & Valve States	20	Passed
7	Mode Switch	14, 17, 18, 31	Passed
8	COD: Date & Time	9, 10, 31	Passed
9	COD: Daily Schedule	9–13, 20	Passed
10	Boost Mode	17, 19.x, 31	Passed
11	Temperature Offset	15, 21, 23, 31	Passed
12	Factory Reset	15, 23, 24, 31	Passed

The inspection results are summarized in Table II. The automatic summer/winter time switching was fully verified, as the implementation correctly delegates to native STM32 HAL RTC functions. Hardware component replaceability was partially satisfied: dedicated abstraction layers exist for the buttons, rotary encoder, motor driver, and display, but ADC-based measurements and the internal temperature sensor lack own abstraction layers due to their coupling to a single DMA-driven ADC peripheral in the SensorTask.

In summary, all requirements were fully verified through testing or inspection, with the exception of hardware replaceability, which was partially met due to the aforementioned measurement code coupling. No functional defects were observed.

TABLE II
SUMMARY OF INSPECTION RESULTS

REQ	Name	Status
22	Summer/Winter Time Switching	Fully passed
30	Hardware Replaceability	Partially passed

VII. CONCLUSION

This paper presented the software design and implementation for a radiator thermostat prototype, with a focus on the Routed-Model-View-Presenter (R-MVP) pattern as an architectural approach for decoupling UI logic in embedded systems. The R-MVP pattern addresses specific challenges that arise when applying classical MVP in a bare-metal embedded context: the absence of rich widget infrastructures for input handling and the need for partially state-driven page navigation. By introducing a centralized Router component, the pattern achieves a clean separation between input processing, presentation logic, and rendering, while remaining practical within the constraints of a verbose programming language and limited hardware resources.

The modular, task-based architecture built around FreeRTOS enables clear separation of concerns across hardware abstraction, control logic, sensor management, and user interface. Verification through integration testing and source code inspection confirmed that all functional requirements were met, with the sole partial limitation being the hardware replaceability of ADC-based measurement code due to its coupling to a single DMA-driven peripheral.

The boilerplate overhead observed in the Router implementation highlights a trade-off inherent to applying structured design patterns in C. A migration to a language with stronger compile-time abstractions, such as Rust, could address this limitation while preserving or improving runtime efficiency.

Future work will extend the software with valve mainte-

nance, temperature control algorithms, and integration of the Matter communication standard for smart home interoperability.

REFERENCES

[1] Umweltbundesamt (Deutschland). (2022, March) Treibhausgasemissionen stiegen 2021 um 4,5%. [Online]. Available: https://www.umweltbundesamt.de/presse/pressemitteilungen/treibhausgasemissionen-stiegen-2021-um-45-prozent

[2] Statistisches Bundesamt (Deutschland). (2025, July) CO2-Emissionen beim Heizen binnen 20 Jahren um 12% gesunken. [Online]. Available: https://www.destatis.de/DE/Presse/Pressemitteilungen/Zahl-der-Woche/2024/PD24_05_p002.html

[3] M. Kersken, H. Sinnesbichler, and H. Erhorn, "Analyse der Einsparpotenziale durch Smarthome– und intelligente Heizungsregelungen," *Bauphysik*, vol. 40, no. 5, pp. 276–285, 2018.

[4] eQ-3 AG, "Operating Manual BLUETOOTH® Smart Radiator Thermostat UK eqiva CC-RT-M-BLE-EQ," May 2018.

[5] ELVjournal, "MAX! Heizkörperthermostat als ARR-Bausatz," *ELVjournal*, no. 2, 2012.

[6] M. Potel, "MVP: Model-View-Presenter - The Taligent Programming Model for C++ and Java," Taligent Inc., Tech. Rep., 1996.

[7] Q. Boucher, G. Perrouin, J.-M. Davril, and P. Heymans, *Engineering Configuration Graphical User Interfaces from Variability Models*. Springer, October 2017, pp. 1–46.

[8] Wikipedia contributors. (2024) Model-view-presenter — Wikipedia, the free encyclopedia. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93presenter&oldid=1264370651

[9] A. Menzel, "MiraTherm Radiator Thermostat Software Specification," Fulda University of Applied Sciences, Tech. Rep., January 2026, version 1.0. [Online]. Available: https://is.gd/mt_rt_sw_specs1

[10] Alexander Menzel. (2026) MiraTherm Radiator Thermostat Software Design. [Online]. Available: https://github.com/MiraTherm/miratherm-radiator-thermostat-documentation/blob/main/software/design/README.md

[11] Chris Ramsdale. (2010, March) Building MVP apps: MVP Part I. [Online]. Available: https://www.gwtproject.org/articles/mvp-architecture.html

[12] Bill Kratochvil. (2011) MVPVM Design Pattern - The Model-View-Presenter-ViewModel Design Pattern for WPF.

[13] Google LLC. (2025) Input events overview | Android Developers. [Online]. Available: https://developer.android.com/develop/ui/views/touch-and-input/input-events

[14] World Wide Web Consortium. (2024) UI Events: W3C Working Draft. [Online]. Available: https://www.w3.org/TR/uievents/

[15] LVGL Kft. (2025) LVGL 9.4 Documentation: Encoder. [Online]. Available: https://docs.lvgl.io/9.4/details/main-modules/indev/encoder.html

[16] Rust Embedded Working Group. (2025) The Embedded Rust Book. [Online]. Available: https://docs.rust-embedded.org/book/

[17] The Rust Project Developers. (2025) The Rust Reference: Monomorphization. [Online]. Available: https://doc.rust-lang.org/reference/items/generics.html#monomorphization

[18] Alexander Menzel. (2026) MiraTherm Radiator Thermostat Software Verification. [Online]. Available: https://github.com/MiraTherm/miratherm-radiator-thermostat-documentation/blob/main/software/verification/README.md