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*—Text...**

***Index Terms*—keyword1, keyword2, keyword3, keyword4, keyword5**

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-domain 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 in order 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-over-Thread, 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 developed in the late 1970s. 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].

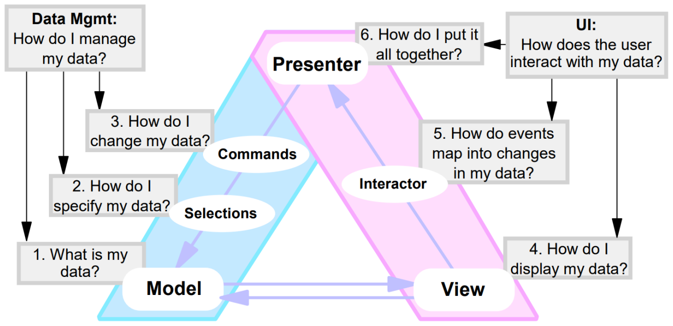
Data Mgmt:
How do I manage my data?
6. How do I put it all together?
UI:
How does the user interact with my data?
Presenter
3. How do I change my data?
Commands
5. How do events map into changes in my data?
Interactor
2. How do I specify my data?
Selections
1. What is my data?
4. How do I display my data?
Model
View

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 other 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.

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,

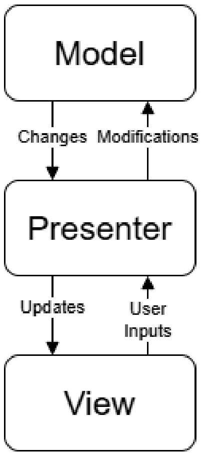

Fig. 2. Data flow in the MVP programming model (mod. from [7], [8]).

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. [9]

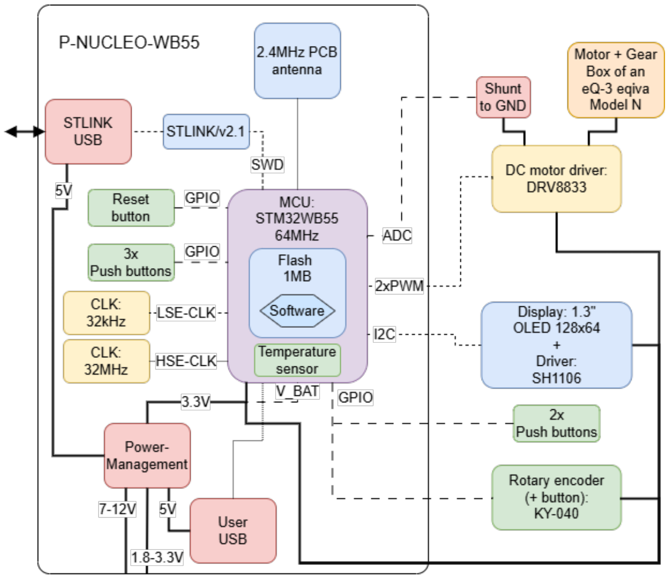

Fig. 3. Block diagram of the thermostat's prototype hardware for software development and testing [9].

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. [9]

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. [9]

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. [9]

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 must provide contextual button hints on every page, indicating the function assigned to each physical button. [9]

IV. Technical Design

This section presents the software architecture and the adapted design pattern used for the UI implementation. The design addresses the requirements from section III by establishing a modular, task-based structure that separates hardware interaction, control logic, and user interface concerns.

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-over-Thread 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*

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 [10]. This separation is the primary reason for choosing MVP over MVC as the basis for the thermostat's UI architecture.

However, classical MVP relies on rich, automated UI widgets (Interactors) that generate high-level semantic events—an assumption rooted in desktop frameworks such as those in Windows or Taligent's CommonPoint [11]. In the 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.

While the LVGL graphics library does offer an encoder input device abstraction [12], 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). For this reason, native LVGL input callbacks are not used. Instead, a dedicated *Router* component is introduced to receive input events from the InputTask and forward them to the active presenter, enabling full control over gesture interpretation.

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

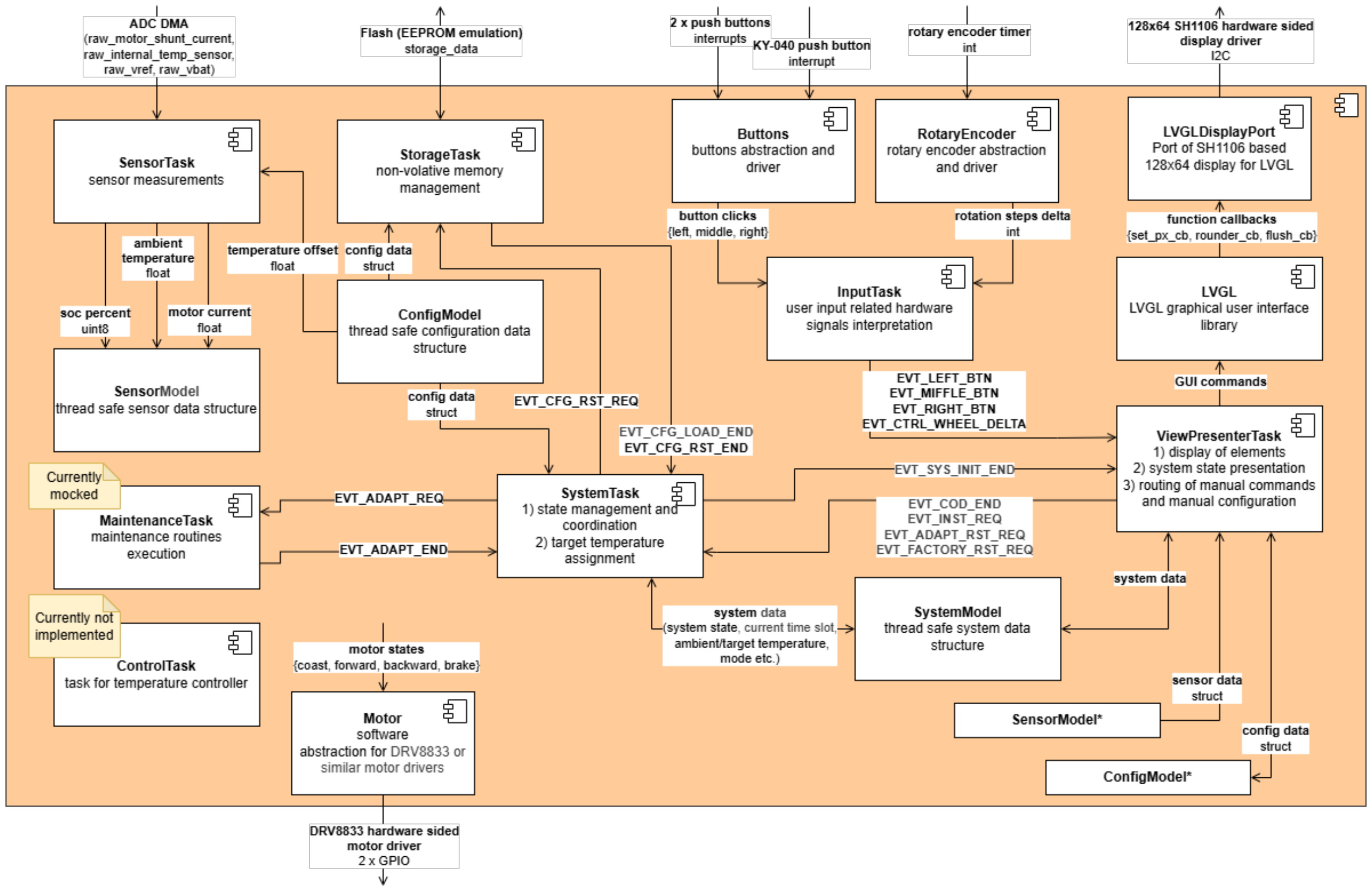
ADC DMA (raw_motor_shunt_current, raw_internal_temp_sensor, raw_vref, raw_vbat)
Flash (EEPROM emulation) storage_data
2 x push buttons interrupts
KY-040 push button interrupt
rotary encoder timer int
128x64 SH1106 hardware sided display driver I2C
SensorTask sensor measurements
StorageTask non-volative memory management
Buttons buttons abstraction and driver
RotaryEncoder rotary encoder abstraction and driver
LVGLDisplayPort Port of SH1106 based 128x64 display for LVGL
button clicks {left, middle, right}
rotation steps delta int
function callbacks {set_px_cb, rounder_cb, flush_cb}
ambient temperature float
temperature offset float
config data struct
InputTask user input related hardware signals interpretation
LVGL LVGL graphical user interface library
soc percent uint8
motor current float
ConfigModel thread safe configuration data structure
SensorModel thread safe sensor data structure
config data struct
EVT_CFG_RST_REQ
EVT_LEFT_BTN
EVT_MIFFLE_BTN
EVT_RIGHT_BTN
EVT_CTRL_WHEEL_DELTA
GUI commands
EVT_CFG_LOAD_END
EVT_CFG_RST_END
ViewPresenterTask 1) display of elements 2) system state presentation 3) routing of manual commands and manual configuration
Currently mocked
EVT_SYS_INIT_END
MaintenanceTask maintenance routines execution
EVT_ADAPT_REQ
SystemTask 1) state management and coordination 2) target temperature assignment
EVT_COD_END
EVT_INST_REQ
EVT_ADAPT_RST_REQ
EVT_FACTORY_RST_REQ
EVT_ADAPT_END
system data
Currently not implemented
ControlTask task for temperature controller
motor states {coast, forward, backward, brake}
system data (system state, current time slot, ambient/target temperature, mode etc.)
SystemModel thread safe system data structure
sensor data struct
SensorModel*
config data struct
Motor software abstraction for DRV8833 or similar motor drivers
ConfigModel*
DRV8833 hardware sided motor driver 2 x GPIO

Fig. 4. Architecture of the radiator thermostat software.

(COD). This requires navigation to be partially state-driven rather than purely user-driven as in classical MVC or MVP. The Router addresses this by reading the current system state from the SystemModel to determine which page to display.

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

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

2) 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.

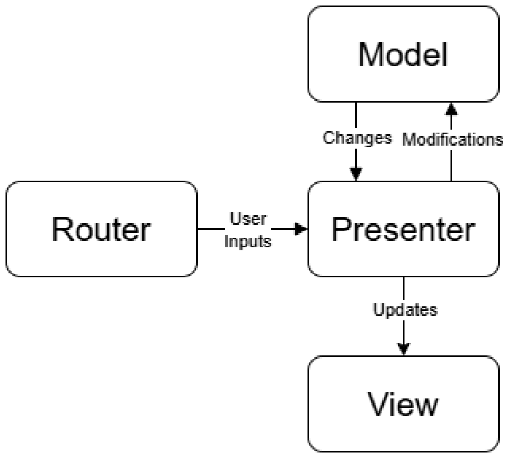

Fig. 5. Data flow in the R-MVP programming model.

3) 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, main challenges encountered during implementation are discussed, along with the solutions adopted to address them.

A. Interaction of ViewPresenterTask with SystemTask

Implementation of a paritally 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 and views, as well as for forwarding input events. A big amount of this code could be reduced by using a language with advanced abstraction features, such as Rust, which is well-suited for embedded systems and offers "zero cost" abstractions (e.g., traits, generics, and pattern matching). 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 and libraries.

VI. Verification and Results

Text…

A. Test Environment

Text…

B. Results

Text…

VII. Conclusion

Text…

References

[1] Statistisches Bundesamt (Deutschland). (2025, 07) 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

[2] Umweltbundesamt (Deutschland). (2022, 03) Treibhausgasemissionen stiegen 2021 um 4,5%. [Online]. Available: https://www.umweltbundesamt.de/presse/pressemitteilungen/treibhausgasemissionen-stiegen-2021-um-45-prozent

[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," 05 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, 10 2017, pp. 1–46.

[8] Wikipedia contributors. (2024) Modelviewpresenter — 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., 01 2026, version 1.0.

[10] Chris Ramsdale. (2010, 03) Building MVP apps: MVP Part I. [Online]. Available: https://www.gwtproject.org/articles/mvp-architecture.html

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

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

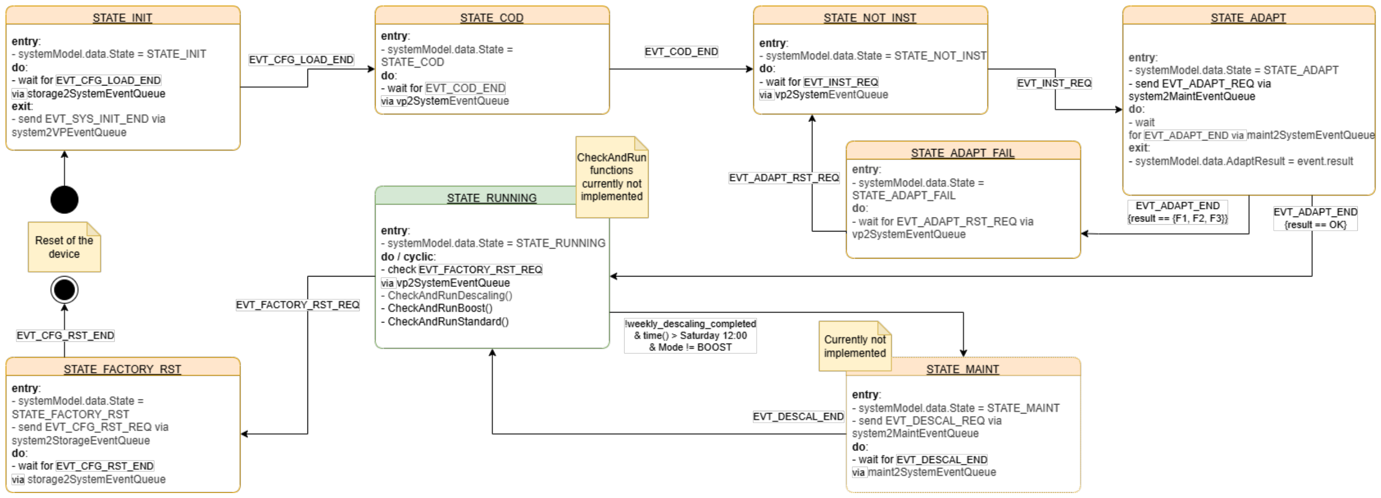
STATE_INIT
entry:
- systemModel.data.State = STATE_INIT
do:
- wait for EVT_CFG_LOAD_END via storage2SystemEventQueue
exit:
- send EVT_SYS_INIT_END via system2VPEventQueue
EVT_CFG_LOAD_END
STATE_COD
entry:
- systemModel.data.State = STATE_COD
do:
- wait for EVT_COD_END via vp2SystemEventQueue
EVT_COD_END
STATE_NOT_INST
entry:
- systemModel.data.State = STATE_NOT_INST
do:
- wait for EVT_INST_REQ via vp2SystemEventQueue
EVT_INST_REQ
STATE_ADAPT
entry:
- systemModel.data.State = STATE_ADAPT
- send EVT_ADAPT_REQ via system2MaintEventQueue
do:
- wait for EVT_ADAPT_END via maint2SystemEventQueue
exit:
- systemModel.data.AdaptResult = event.result
EVT_ADAPT_END {result == {F1, F2, F3}}
EVT_ADAPT_END {result == OK}
STATE_ADAPT_FAIL
entry:
- systemModel.data.State = STATE_ADAPT_FAIL
do:
- wait for EVT_ADAPT_RST_REQ via vp2SystemEventQueue
EVT_ADAPT_RST_REQ
CheckAndRun functions currently not implemented
STATE_RUNNING
entry:
- systemModel.data.State = STATE_RUNNING
do / cyclic:
- check EVT_FACTORY_RST_REQ via vp2SystemEventQueue
- CheckAndRunDescaling()
- CheckAndRunBoost()
- CheckAndRunStandard()
!weekly_descaling_completed & time() > Saturday 12:00 & Mode != BOOST
Currently not implemented
STATE_MAINT
entry:
- systemModel.data.State = STATE_MAINT
- send EVT_DESCAL_REQ via system2MaintEventQueue
do:
- wait for EVT_DESCAL_END via maint2SystemEventQueue
EVT_DESCAL_END
EVT_FACTORY_RST_REQ
Reset of the device
EVT_CFG_RST_END
STATE_FACTORY_RST
entry:
- systemModel.data.State = STATE_FACTORY_RST
- send EVT_CFG_RST_REQ via system2StorageEventQueue
do:
- wait for EVT_CFG_RST_END via storage2SystemEventQueue

Fig. 6. State machine of the SystemTask