

A State-Indexed Execution Model for Embedded Systems

Matúš Nečas

University of Žilina

Faculty of Electrical Engineering and Information Technology

Univerzitná 8215/1, 010 26 Žilina, Slovakia

matus.necas@feit.uniza.sk

Roman Budjac

University of Žilina

Research Centre

Univerzitná 8215/1, 010 26 Žilina, Slovakia

roman.budjac@uniza.sk

Juraj Ďudák

Slovak University of Technology in Bratislava

Faculty of Materials Science and Technology in Trnava

Jána Bottu 25, 917 24 Trnava, Slovakia,

University of Žilina

Research Centre

Univerzitná 8215/1, 010 26 Žilina, Slovakia

juraj.dudak@stuba.sk

Abstract. *This work introduces the State-Indexed Application Loop (SIAL), a lightweight implementation pattern for microcontroller-based systems. SIAL decomposes application logic into a fixed set of indexed states, where each state encapsulates both computation and successor selection through a single activation function. Rather than introducing a fundamentally different finite-state-machine formalism, SIAL organizes state-oriented application logic by embedding successor selection directly into individual states, while the Scheduler performs indexed state selection and invocation. The model preserves deterministic sequential execution over a statically allocated state space while remaining compatible with interrupt-driven asynchronous inputs via a shared system context. The resulting organization provides a uniform structure for state-oriented control flow with constant-time indexed state selection. The presented properties are analytical and structural; quantitative performance advantages require implementation-specific experimental validation.*

Keywords. control flow design, embedded systems, finite-state machines, software architecture, state-based execution

1 Introduction

Microcontrollers have become an integral part of modern electronic systems and represent a fundamental computing platform for a wide range of applications (Hee et al., 2021). Their usage spans from simple data acquisition and processing devices, through wearable electronics and body sensing systems, industrial automation applications, to complex embedded solu-

tions equipped with user interfaces, wireless communication, and cloud connectivity (Lestariningsih et al., 2019; Madhu and Vyjayanthi, 2018; Zavitsanou et al., 2016). With the increasing performance of modern hardware, the complexity of software deployed on these platforms continues to grow.

The development of embedded applications requires knowledge across multiple domains. In addition to low-level programming and hardware peripheral handling, it involves analog and digital signal processing, communication protocol implementation, data flow management, user interface design, and coordination of concurrent activities. In recent years, machine learning techniques, including computer vision and neural networks, have also been integrated into embedded systems, further increasing the demands on modularity, maintainability, and scalability of application firmware (Sanchez-Iborra and Skarmeta, 2020).

Applications with user interfaces represent a particularly challenging category. Such systems must ensure smooth interaction between the user and the device through physical controls, touch screens, or their combination (Necas et al., 2026). They typically operate in multiple modes or screens between which the user dynamically navigates. Each mode requires dedicated logic for input processing, validation and update of application data, control of displayed content, and handling of events generated by both the user and the system.

The increasing complexity of embedded applications places high demands on software architecture design (Hofmeister et al., 2000). Poorly designed program structures may reduce clarity of control logic, increase maintenance costs, and limit extensibility (Ganssle, 2008). For this reason, various architectural

approaches are used in practice, ranging from simple main-loop-based designs (superloop), through finite-state machine (FSM) implementations, to event-driven frameworks and real-time operating systems (White, 2024).

Each of these approaches has specific advantages and limitations. Finite-state machines provide an explicit representation of operational modes and transitions, but their implementation may take different forms, including *centralized switch statements*, *transition tables*, and *function-pointer-based state handlers*. The choice of implementation therefore affects how state behavior and transition logic are organized in the source code. Event-driven architectures provide a different approach by distributing execution across event handlers and dispatching mechanisms, while real-time operating systems introduce scheduling and concurrency abstractions that are not required by all embedded applications (Michelson, 2006).

For a class of microcontroller applications characterized by a fixed set of operational modes, sequential execution, and statically allocated resources, a lightweight organization of state-oriented application logic can be preferable to a more general execution framework. In such systems, the main design concern is not necessarily concurrency or dynamic event dispatching, but maintaining a clear relationship between the currently active application mode, the code executed in that mode, and the selection of the subsequent mode.

This work presents the *State-Indexed Application Loop* (SIAL), a lightweight implementation pattern addressing this organization problem. SIAL structures application logic as a statically allocated collection of indexed states. Each state exposes a uniform `activate()` operation that performs state-specific processing and returns the identifier of the next state. The scheduler is intentionally kept minimal: it selects the active state by index, invokes its activation operation, and stores the returned state identifier for the next execution step.

The contribution of this work is not a new finite-state-machine semantics. Instead, SIAL defines a specific implementation organization for state-oriented embedded applications, emphasizing static allocation, uniform state activation, and local successor selection. The paper describes the resulting execution model formally and discusses its relationship to conventional FSM implementation techniques, as well as its applicability, advantages, and limitations in resource-constrained microcontroller systems.

2 Related Work

Existing approaches for managing software complexity in embedded systems can be broadly categorized according to their execution model, including event-driven frameworks, real-time operating systems, and

state-oriented or hybrid architectures. The main objective of these approaches is to improve modularity and scalability of application logic compared to the classical superloop paradigm.

Event-driven frameworks represent one of the dominant design strategies. QP™ Real-Time Embedded Frameworks (RTEFs) implement a hierarchical state machine model combined with an event-driven execution mechanism. System behavior is modeled through active objects that communicate via event queues, enabling explicit representation of system dynamics and separation of concerns. While this improves modularity and scalability, it introduces an additional event management layer that can obscure direct execution flow and increase architectural complexity (Quantum Leaps, LLC, 2014).

A similar event-driven principle is widely used in graphical user interface libraries such as LVGL (Light and Versatile Graphics Library). In this case, application logic is decomposed into callback functions bound to user interface events (LVGL Kft., 2024). This enables flexible and portable UI development across platforms. However, distributing logic across callbacks can reduce traceability of execution flow and make global reasoning about system behavior more difficult.

Another class of solutions is represented by real-time operating systems such as FreeRTOS and Zephyr. These systems adopt a scheduler-based execution model in which application logic is decomposed into concurrent tasks. Execution is coordinated using synchronization primitives and inter-task communication mechanisms. While this improves modularity and enables concurrency, it also introduces complexity related to scheduling, synchronization, and shared resource management (Barry and FreeRTOS Team, 2023).

In addition to these approaches, state-machine-based extensions integrated into RTOS environments, such as the Zephyr State Machine Framework, combine explicit state modeling with operating system scheduling. This enables structured representation of application behavior while maintaining real-time capabilities (The Zephyr Project Contributors, 2025). However, application logic is distributed across both the state-machine layer and the underlying RTOS infrastructure, which increases conceptual fragmentation.

Overall, existing approaches represent different strategies for addressing the limitations of the superloop architecture. Event-driven systems emphasize modularity through indirect execution, RTOS-based systems introduce concurrency and scheduling abstractions, while state-machine frameworks provide structured behavioral modeling at the cost of additional architectural layers. SIAL instead focuses on the organization of state-oriented application logic at the implementation level, without requiring an event dispatcher, event queue, or operating system scheduler.

Recent work continues to explore lightweight software architectures and state-machine-based ap-

proaches for resource-constrained embedded systems. These approaches emphasize structured state representation, modular control flow, and reduced runtime infrastructure, particularly for microcontroller-oriented applications. Such developments demonstrate the continued relevance of state-oriented execution models in embedded software, while also highlighting the variety of implementation techniques available for realizing them (Ali and Vitek, 2022; Babiuch and Foltyn, 2024; Ball et al., 2024).

SIAL is positioned within this context as a lightweight implementation pattern that focuses specifically on state-local execution and statically indexed control flow rather than introducing a fundamentally new finite-state-machine formalism.

2.1 Conventional FSM Implementation Techniques

Finite-state machines can be implemented using several conventional software structures. A switch-based implementation commonly represents the active state using an enumerated value and evaluates the corresponding state behavior through a centralized switch statement. A table-driven implementation stores transition information explicitly in a table indexed by the current state and input condition. Another common approach uses function pointers to associate each state with a handler function, allowing the active handler to be selected through an indexed structure.

SIAL is structurally related to the function-pointer approach. The distinction is primarily organizational rather than semantic: SIAL defines a uniform state abstraction in which the state activation operation performs both state-specific processing and successor selection, while the scheduler remains responsible only for indexed state selection and invocation. SIAL is therefore positioned as an implementation pattern for state-oriented control flow rather than as a fundamentally different finite-state-machine formalism.

3 Materials and Methods

The *State-Indexed Application Loop* (SIAL) is a lightweight implementation pattern for microcontroller-based systems. It is derived from the classical superloop architecture, which represents one of the most widely used execution paradigms in embedded applications.

3.1 Baseline Execution Model: Superloop

In the superloop model (as shown in Fig. 1), the application is structured into an initialization phase (*Setup Logic*) followed by an infinite cyclic execution loop (*Cyclic Logic*). Within this loop, all application logic is executed sequentially, while asynchronous behavior is typically handled through interrupts.

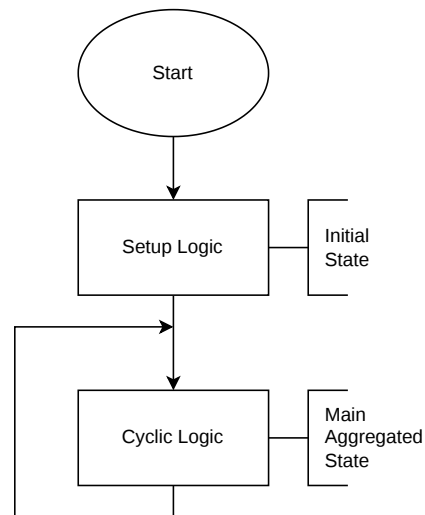


Figure 1. Superloop execution model

Although this model is widely adopted due to its simplicity, deterministic execution flow, and low runtime overhead, it does not scale well in systems with multiple operational modes. As system complexity increases, the application logic becomes concentrated within a single execution block, leading to a monolithic structure with strong coupling between functional components. From a software engineering perspective, this corresponds to a single aggregated state without explicit decomposition into independent behavioral units.

To address these limitations, SIAL introduces a decomposition of the monolithic execution loop into a set of explicitly defined states with localized runtime logic and transition behavior.

3.2 State-Driven Decomposition and RACe Execution Rule

SIAL is based on the idea that the monolithic control logic of a superloop-based system can be decomposed into a set of explicitly defined states. Each state represents a distinct operational mode of the system and encapsulates the functionality associated with that mode.

This decomposition transforms the application structure from a single centralized control flow into a collection of self-contained functional units. As a result, system behavior is no longer expressed as a single cyclic logic, but as an interaction between explicitly defined states.

To realize this decomposition, SIAL uses the RACe (*Run And Choose Next*) execution rule, a term introduced in this work to describe the state activation and successor-selection mechanism used by SIAL.

Within the RACe execution rule, each state executes its associated functionality and determines its successor based on the current system context. Computation and successor selection are therefore integrated into a single state-local operation.

This organization avoids the need for a separate centralized transition-handling layer and places application-specific successor selection within the corresponding state implementation, while the scheduler remains responsible for selecting and invoking the active state.

3.3 System Architecture

The SIAL architecture consists of two tightly connected components: the *Scheduler* and the *State* abstraction, as illustrated in Fig. 2.

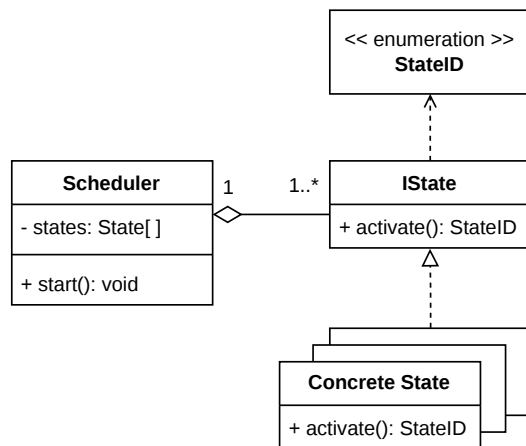


Figure 2. Architecture of the SIAL framework

The State abstraction is defined through the interface *IState*. Each state encapsulates a self-contained portion of application logic and exposes a single activation method, denoted as *activate()*. This method serves as the execution entry point of the state and provides a unified operation for state-specific processing and successor selection.

The activation method processes the current system context and returns the identifier of the next state, thereby implementing the RACe execution rule at the software level. In this sense, each state contains its own successor-selection logic as part of its execution.

The Scheduler is responsible for controlling the execution flow of the system. It maintains the identifier of the currently active state and invokes the corresponding state through the *IState* interface (as shown in Fig. 3). After each execution, the Scheduler updates its internal state index based on the value returned by the active state. This creates a strict execution cycle in which sequencing is centralized in the Scheduler, while application-specific successor selection is distributed across states.

3.4 State Lifecycle Semantics

Each state may be activated repeatedly while it remains the current state. A self-transition therefore corresponds to repeated activation of the same state without leaving its operational mode. A transition to a dif-

ferent state terminates the current state activation sequence and makes the target state the next active state. One-time initialization and cleanup operations are not implicitly provided by the *activate()* interface; they are explicitly implemented by the application, for example by detecting the first activation or the transition into a state. Similarly, re-entry into a previously visited state is treated as a new activation sequence, allowing the application to explicitly perform any required re-initialization.

This lifecycle definition intentionally avoids imposing separate *entry*, *do*, and *exit* methods. Their functionality can instead be represented by application-level conditions within *activate()*, or by auxiliary state-local mechanisms when required. Consequently, SIAL defines the execution structure but does not prescribe a specific policy for initialization, cleanup, or re-entry behavior.

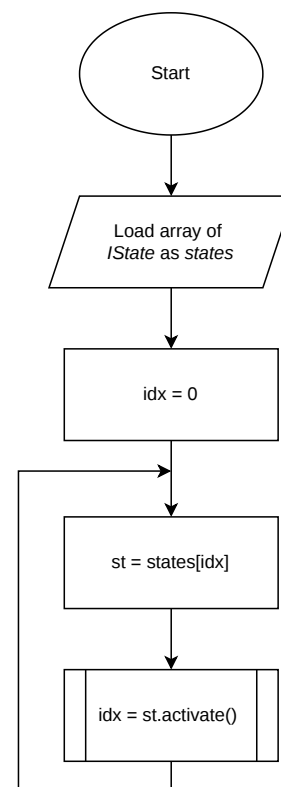


Figure 3. SIAL execution flow

3.5 Illustrative Example: Comparative Execution Models

To illustrate the organization of state-oriented application logic in SIAL and its relationship to conventional FSM implementations, consider a typical multi-state embedded application with human interaction (Fig. 4). The system operates in four distinct modes: Booting (*B*), Calibration (*C*), Measurement (*M*), and Failure (*F*).

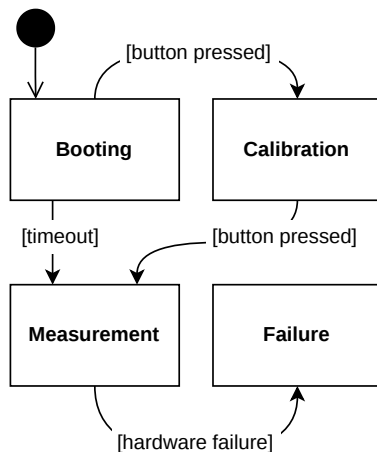


Figure 4. An example of typical embedded four-state system

The system reacts to a set of discrete events, namely timeout (*tm*), button press (*bp*), and hardware failure (*hf*). The resulting transition structure can be represented by the partial transition relation in Table 1.

Each state in the system corresponds to a specific operational mode and defines its internal behavior and possible successor states.

The Booting state (*B*) represents the system initialization phase, during which hardware and software components are brought into a valid operational configuration. Upon successful initialization, the system transitions either to the Calibration state (*C*) or directly to the Measurement state (*M*), depending on the system context and startup conditions.

The Calibration state (*C*) is responsible for establishing reference parameters required for accurate operation. Once calibration is completed, the system proceeds to the Measurement state (*M*), which represents the primary functional mode of the application.

The Measurement state (*M*) implements the core application logic, continuously processing incoming data and performing system-specific computations. During operation, it monitors for abnormal conditions; in the case of a detected hardware failure (*hf*), the system transitions to the Failure state (*F*).

The Failure state (*F*) represents an error-handling or recovery-oriented mode in which normal operation is suspended. Depending on the implementation, it may remain active until external intervention or recovery conditions are satisfied.

In a conventional FSM, the transition relation can be represented explicitly through a transition function or table, while the state-specific behavior may be organized separately or combined with transition handling depending on the implementation technique. SIAL follows the latter organization by associating state processing and successor selection with the active state through its `activate()` operation. The resulting structure is typically represented by a transition rela-

tion such as the one shown in Table 1.

state	B	C	M	F
B		bp	tm	
C			bp	
M				hf
F				

Table 1. Partial transition relation between system states

Event-driven architectures, such as GUI-based systems, implement transitions through event callbacks. While this reduces the need for an explicit transition table, it introduces a separation between event handling and cyclic application logic (e.g., measurement loops), often requiring additional coordination mechanisms to maintain consistent system behavior, typically realized through shared or global state variables.

FSM implementations can organize transition logic and state-specific processing in different ways depending on the selected implementation technique. Switch-based and table-driven approaches commonly provide an explicit representation of transition logic, while function-pointer-based implementations can combine state processing and successor selection within individual state handlers. As the number of states and transitions grows, these different representations involve different trade-offs in terms of code organization, traceability, and separation of concerns. In event-driven designs, application behavior is distributed across event handlers and other execution mechanisms, which may require additional coordination to maintain consistent state behavior.

In contrast, the SIAL implementation pattern integrates state behavior and successor selection into a single state-local activation operation. Each state evaluates the system context and determines its successor directly, resulting in a uniform per-state abstraction regardless of the number of possible successor states. In the illustrative four-state example, this corresponds to four state implementations, each exposing the same `activate()` interface, while the Scheduler provides a common mechanism for selecting and invoking the active state.

3.6 Formal Model

The organization of SIAL can be described using the classical finite-state machine model as a reference. The purpose of the formalization is to describe how state-specific processing and successor selection are combined within the SIAL implementation pattern.

The behavior of classical deterministic finite-state machines is defined as

$$FSM = (S, E, \delta, s_0), \quad (1)$$

where S is a finite set of states, E is a set of input

events, $\delta : S \times E \rightarrow S$ is a centralized transition function, and s_0 is the initial state (Lee and Yannakakis, 1996).

The system evolution is given by

$$s_{k+1} = \delta(s_k, e_k), \quad (2)$$

where $e_k \in E$ denotes the event observed at step k .

SIAL distributes the state-specific successor selection across the state implementations. The system is defined as

$$SIAL = (S, A, s_0), \quad (3)$$

where $A = \{a_0, a_1, \dots, a_{n-1}\}$ is a set of state-specific activation functions, and each function a_i is realized in software through the corresponding `activate()` method of a state.

Each activation function is defined as

$$a_i : X \rightarrow X \times S, \quad (4)$$

where X denotes the system context. The output therefore consists of the evolved system context and the identifier of the successor state.

In this model, the system context X represents the complete set of observable runtime information available to the application at a given moment. This includes external inputs (e.g., user events or sensor readings), internal state variables, and values updated asynchronously, typically through interrupt service routines or peripheral interfaces.

The global execution semantics are defined as

$$(X_{k+1}, s_{k+1}) = a_{s_k}(X_k), \quad (5)$$

where X_k is the system context at step k .

The resulting transition relation between states is given by

$$T = \{(s_i, s_j) \mid \exists X, X' : a_i(X) = (X', s_j)\}, \quad (6)$$

which defines a directed graph over S induced by the possible successor states of the activation functions.

4 Results

The *State-Indexed Application Loop* (SIAL) was evaluated from a structural and execution-semantics perspective. The resulting properties follow directly from the organization of the implementation pattern and describe state selection, execution structure, and memory organization. They should be interpreted as analytical properties rather than experimentally measured performance results.

The SIAL implementation uses a statically allocated indexed state space

$$S = \{0, 1, \dots, n-1\}. \quad (7)$$

Each state corresponds directly to an entry in a fixed array structure, where the array index serves as the unique state identifier. System execution is driven by the currently active state s_k . At each step, the corresponding activation function processes the current system context and returns the successor state (5),

The system evolution is therefore governed by the discrete update rule

The resulting execution trace forms a discrete sequence in the state space S

$$s_0, s_1, s_2, \dots \quad (8)$$

where each successor state is selected by the activation function of the currently active state.

A direct consequence of the indexed state organization is that selecting the active state does not require searching through the state set. The scheduler therefore performs indexed state selection in constant time,

$$\tau_{\text{scheduler}} = O(1). \quad (9)$$

This result applies to the state-selection operation performed by the Scheduler and does not describe the execution time of `activate()`, which depends on the application-specific processing implemented by each state.

The analysis also shows that SIAL provides a uniform implementation structure for states. Each state exposes one `activate()` operation that combines state-specific processing with successor selection. The number of possible successor states therefore does not change the interface or the scheduler structure. However, the corresponding decision logic remains part of the state implementation and may increase its internal complexity as the number of possible transitions grows.

From a memory-organization perspective, the state collection can be allocated statically as a fixed array. The scheduler requires only the current state index and the state collection, without requiring dynamically allocated state objects or a runtime registration mechanism. This provides compatibility with embedded applications where static memory allocation is preferred.

The resulting properties are summarized in Table 2. The table describes structural characteristics derived from the respective execution approaches; the qualitative entries are therefore design assessments rather than experimentally measured performance values.

5 Discussion and Conclusions

The *State-Indexed Application Loop* (SIAL) provides a state-centric implementation pattern for embedded systems in which both computation and successor selection are encapsulated within individual states. This organization shifts application-specific control-flow decisions from a separate transition-handling structure to individual state implementations, while the Scheduler

Table 2. Structural characteristics of different execution approaches

Property	FSM	Event-driven	SIAL
Explicit state representation	Yes	Partial	Yes
State-local processing	Depends	Depends	Yes
State-local successor selection	Depends	Depends	Yes
Indexed state selection	Depends	No	Yes
Static memory allocation compatibility	Yes	Depends	Yes
Separate transition representation	Depends	Depends	No
External event dispatcher	No	Typically	No
Scheduler-based concurrency	No	Depends	No
Asynchronous event handling	Depends	Yes	Via shared context
Dynamic runtime extensibility	Depends	Yes	No

remains responsible for selecting and invoking the active state.

In contrast to event-driven architectures, which rely on external dispatching mechanisms such as callback registration, event queues, or observer-based models, SIAL does not require an intermediate event-handling layer for its basic execution. While event-driven systems provide flexibility in highly asynchronous environments by decomposing behavior into event-triggered handlers, this may distribute application logic across multiple components and execution mechanisms.

By embedding successor selection directly into state execution, SIAL establishes a direct relationship between the current system context and the next state. This reduces the separation between state-specific processing and transition selection and provides a uniform organization of state-oriented control flow.

A key structural property of the SIAL implementation pattern is its reliance on static memory allocation and a fixed state space, which aligns well with microcontroller-based systems where dynamic allocation is typically avoided due to fragmentation risks and non-deterministic memory behavior. Unlike classical object-oriented State design patterns, which may depend on dynamic polymorphism and heap-based allocation, SIAL can be implemented using statically defined state structures.

These design choices introduce an inherent trade-off: while state-local organization can simplify the distribution of application logic, encapsulating both computation and successor selection within states can increase per-state complexity, particularly in systems with dense or highly interconnected transition structures. In such cases, complexity is shifted from a separate transition representation to individual state implementations, which may reduce separation of concerns compared to FSM implementations where transitions are explicitly externalized.

On the level of execution semantics, SIAL follows a strictly sequential scheduling model and does not provide hardware-supported concurrency mechanisms such as preemptive multitasking or RTOS-based execution. Nevertheless, asynchronous inputs can be incor-

porated through interrupt-driven updates of the shared system context, provided that access to shared data satisfies the required atomicity and synchronization assumptions.

Although the implementation presented in this work uses an object-oriented abstraction based on state interfaces, the underlying organization is not bound to a specific programming paradigm and can also be implemented in procedural environments.

The results presented in this work establish the main structural and execution properties of SIAL, including indexed state selection, static state organization, and state-local successor selection. These properties describe the intended characteristics of the implementation pattern; quantitative performance advantages such as reduced runtime overhead, memory consumption, or execution time require validation through implementation-specific measurements.

The scalability of the indexed execution mechanism should also be distinguished from the scalability of the application structure itself. Even for systems containing hundreds of states, indexed state selection remains $O(1)$; however, the number of state implementations and the amount of state-local control logic increase with system size. Consequently, large and densely connected state spaces may become more difficult to debug and reason about, particularly when transitions occur between states that are distant in the overall application structure. In such systems, explicit state documentation, naming conventions, and transition tracing mechanisms may be required to maintain readability and debugging efficiency.

Overall, SIAL is positioned as a lightweight implementation pattern for embedded applications that emphasizes static structure, sequential execution, and locality of control flow. It is particularly applicable to systems with well-defined operational modes, where predictable sequential execution and static resource organization are important design considerations. Its main limitation is the reduced suitability for highly asynchronous or concurrent applications, where event-driven frameworks or real-time operating systems may provide more appropriate execution mechanisms.

Acknowledgments

This research was supported by the project VEGA through the Research on motion data analysis methods for applications in diagnosis and therapy of gnostically relevant symptoms under Grant 1/0147/25.

References

- Ali, K., & Vitek, J. (2022). *Lipics, volume 222, ecoop 2022, complete volume* (Vol. 222). Schloss Dagstuhl – Leibniz-Zentrum für Informatik. <https://doi.org/10.4230/2022>
- Babiuch, M., & Foltyniek, P. (2024). Implementation of a universal framework using design patterns for application development on microcontrollers. *Sensors*, 24(10). <https://doi.org/10.3390/s24103116>
- Ball, T., Halleux, J. " d., Devine, J., Hodges, S., & Moskal, M. (2024). Jacdac: Service-based prototyping of embedded systems. *Programming Language Design and Implementation (PLDI)*. <https://www.microsoft.com/en-us/research/publication/jacdac-pldi2024/>
- Barry, R., & FreeRTOS Team. (2023). *Mastering the freertos™ real time kernel: A hands-on tutorial guide* (Version 1.1.0) [Available online]. Amazon Web Services. <https://www.FreeRTOS.org>
- Ganssle, J. (2008). *The art of designing embedded systems*. Newnes.
- Hee, Y. H., Ishak, M. K., Asaari, M. S. M., & Seman, M. T. A. (2021). Embedded operating system and industrial applications: A review. *Bulletin of Electrical Engineering and Informatics*, 10(3), 1687–1700. <https://doi.org/10.11591/eei.v10i3.2526>
- Hofmeister, C., Nord, R., & Soni, D. (2000). *Applied software architecture*. Addison-Wesley Professional.
- Lee, D., & Yannakakis, M. (1996). Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8), 1090–1123. <https://doi.org/10.1109/5.533956>
- Lestariningsih, T., Artono, B., Hidayatullah, N., & Kusbandono, H. (2019). Microcontroller and android hmi based water level and control system. *EAI Endorsed Transactions on Internet of Things*, 5(17).
- LVGL Kft. (2024). *Lvgl — light and versatile graphics library documentation* (v9.0) [Dostupné online; navštívené jún 2026]. LVGL Kft. <https://docs.lvgl.io/9.0/>
- Madhu, G. M., & Vyjayanthi, C. (2018). Implementation of cost effective smart home controller with android application using node mcu and internet of things (iot). *2018 2nd International Conference on Power, Energy and Environment: Towards Smart Technology (ICEPE)*, 1–5. <https://doi.org/10.1109/EPETSG.2018.8659128>
- Michelson, B. M. (2006). Event-driven architecture overview. *Patricia Seybold Group*, 2(12), 10–1571.
- Necas, M., Dudak, J., Gaspar, G., Husar, P., & Gurin, D. (2026). Embedded digital instrumentation for automated assessment of sensory perception. *IEEE Access*, 14, 75933–75944. <https://doi.org/10.1109/ACCESS.2026.3693773>
- Quantum Leaps, LLC. (2014, May). *Qp/c++ and the qt gui framework* (Application Note No. Document Revision K). Quantum Leaps, LLC. Chapel Hill, NC. <http://www.state-machine.com/qt>
- Sanchez-Iborra, R., & Skarmeta, A. F. (2020). Tinyml-enabled frugal smart objects: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, 20(3), 4–18. <https://doi.org/10.1109/MCAS.2020.3005467>
- The Zephyr Project Contributors. (2025). *Zephyr project documentation (latest), version 4.2.99* [Accessed: 2025-09-02]. The Linux Foundation. <https://docs.zephyrproject.org/latest/index.html>
- White, E. (2024). *Making embedded systems*. " O'Reilly Media, Inc."
- Zavitsanou, S., Chakrabarty, A., Dassau, E., & Doyle, F. J. (2016). Embedded control in wearable medical devices: Application to the artificial pancreas. *Processes*, 4(4). <https://doi.org/10.3390/pr4040035>