ZEISS INTERVIEW

Temperature Monitor
Software Architecture Document (SAD)

CONTENT OWNER: Diego Alejandro Parra Guzman

DOCUMENT NUMBER:	RELEASE/REVISION:	RELEASE/REVISION DATE:
• ZEISS_ITRW_001	• DRAF	• 20/02/2025

All future revisions to this document shall be approved by the content owner prior to release.

Public

Table of Contents

Public

List of Figures

Public

List of Tables

Public

1 Documentation Roadmap

This section introduces the SAD for a BareMetal temperature monitor application. The temperature monitor is an application that react to changes in the temperature. The monitor is runs directly on an embedded hardware. Thus, it is equipped with two temperature sensors, also with an EEPROM Memory that provides configuration and three LEDs for indicating the status of the temperature.

The architecture described here resulted from a problem described below:

We ZEISS are looking for a temperature monitor application. This monitor is equipped with two sensors each operating at different resolution: Sensor 1, has a resolution of 1°C per digit while Sensor 2 has a resolution of 0,1°C. Both temperature sensors cannot be used at the same time, therefore, an EEPROM memory provides a configuration that indicates which sensor should be used at specific point of time. The temperature should be measured with a period of (10Khz or eq. 100µs). The monitor is connected to 3 LEDs. The Red LED represent a critical state, it shall be on if the temperature is >=105°C, or <5°C. The Yellow LED represent a warning state, it shall be on if the temperature is >=85°C. Similarly, the Green LED shall be on, only when the temperature is <85°C and >5°C.

1.1 How the documentation is organized

The architecture document is organized into the following sections:

- Section 1.2 Propose and scope of the SAD: this subsection explains the purpose and scope of the SAD.
- Section 1.3 Architecture Background.
- Section 1.4 Architecture Views.
- Section 1.5 Viewpoint Definitions.

Public

1.2 Purpose and Scope of the SAD

This SAD specifies the software architecture for **implementing a monitoring application.** This architecture is characterized for the following quality attributes:

- **Real time: A target system is affected by drastic temperature changes during its normal runtime operation.** To accurately track these changes, the monitor component must sample and report the temperature with a period of <=100us, and with a very low jitter (ideally 0). Crucially, at each sample, , the system shall indicate the current temperature range within the same timeframe i.e., within 100us. This indication can be achieved by using three LEDs, each representing a different temperature range. **If the time to sample, determine the range, and update the LEDs takes longer than 100us, could lead to a system malfunction, potentially resulting in economical loses.**

- **Modifiability: Changes in the architecture should be characterized by lower cost and complexity.** These changes may involve any aspect of the system, including sensors, the underlying platform (hardware, peripherals, and software), or the communication protocols used for interacting with external systems. The need for modifications is often driven by stakeholders, evolving project requirements, or planned system upgrades. Each requested change involves a formal process of planning, development, testing, and deployment to ensure proper integration and functionality.

Depending on a concrete scenario the following Quality attributes might be desirable. However, they are not part of the architecture.

- **Availability:** If the target system presents a high risk for human lives. The monitor component must exhibit high availability. This means a continuous operation, 24 hours a day, 7 days a week, throughout the entire lifespan of the target system. To ensure this, an external system or dedicated hardware must be capable of detecting malfunctions, and initiate a corrective response within a specified timeframe.

- **Energy Efficiency:** With the increasing adoption of IoT devices, energy has become a critical factor in embedded hardware architectures. While energy efficiency must be balanced with performance and availability, it also requires efficient resource management and allocation. When energy consumption crosses a certain threshold, the system may allocate or reallocate a defined set of applications so that the maximum or average consumption load remains within a specified range.

- **Safety:** Safety refers to a system's ability to avoid states that cause or lead to infrastructural damage or human losses. A safety system provides strategies for detecting and recovering from unsafe states, thereby preventing or minimizing the risk of harm. An unsafe state can be caused by the system itself, including its software, hardware, or

Public

peripherals, as well as by the surrounding environment, which may behave in an unsafe manner. Once an unsafe state is recognized, the system can apply recovery maneuvers and continue operating, or, in the worst case, transition to a safe state that requires human intervention.

- **Security:** Depending on the levels of confidentiality, integrity, and availability, the system should be evaluated. Similar to the safety case, it is important to model system threats and vulnerabilities. Attacks can originate either from outside or inside the organization. The source of the attack can be either a human or another system. During and after a potential attack, the system shall ensure that confidentiality and integrity are maintained.

  Based on the problem's description and quality attributes, A layered architecture fit conveniently since we can group functionality into independent layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage must be unidirectional. Layers completely partition a set of software, and each partition is exposed through a public interface.

  In addition to these quality attributes, the proposed architecture follows three iterations:

  - **Iteration 1 (Monolitic).** The first iteration proposes a Minimal architecture to achieve the requested Quality Attributes.

  - **Iteration 2 (HAL Layer).** The second iteration proposes improvement, **introducing a hardware abstraction layer (HAL) to increase** modularity and isolating low level functionality from the application.

  - **Iteration 3 (SCHED Layer).** The third iteration introduces a scheduling layer to improve the real time characteristics of the system.

1.3 Architecture Background

In this section a complete argumentation and reasoning for architecture decisions is provided.

1.3.1 Baseline Architecture Pattern

The architecture quality attributes suggest using a layered architecture pattern for the following reasons:

- Changes in low-level components such as Drivers, Frameworks, or Firmware should not affect the normal development or execution of the application.

Public

- Parts of the system should be exchangeable. Components should be able to be replaced by alternative implementations without affecting the rest of the system.

- To help understandability and maintainability. Each component should be grouped in layers, sharing a common set of responsibilities.

The layer architecture pattern depicted in figure 1, consists of a number of abstraction levels that has to be selected according to an abstraction criteria. However, having too many abstractions might impose an unnecessary overhead. Layers provide concrete interfaces, such that low level layers offer services to upper layers. The layered approach supports system evolution, providing better control to internal functionality of individual system components.

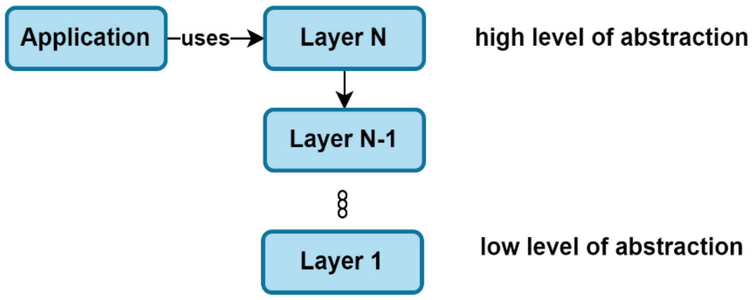


Figure 1 Layer pattern

1.3.2 Achieving Quality Attributes

1.3.2.1 Realtime

Ensuring real-time is a complex task, as it depends on the number of computations involved, their interactions with other task, and the resource allocation methods used to assign these computations to physical resources. The mapping of a single task to a computational resource is principally influenced by three factors: the execution time, the periodicity of the computation, and the number of resources blocked during its execution. Similar factors like computation chains, resource dependences and communication might also impact the resource allocation. This complexity is more evident in constrained devices, with limited processing capabilities and small set of peripherals.

The execution time of a computation depends on the number of logical operations that involved in the computation, as well as the system capacity to handle these operations efficiently. Furthermore, as the utilization of resources approaches their maximal capacity, the system degrades significantly, which can drastically affect the computation's execution time.

Public

The existing solution for achieving Realtime properties is to introduce an scheduling algorithm that takes into account not only task's characteristics but also the available resources. For this specific use cases a priority based scheduling algorithm is sufficient to achieve the desired level of prioritization and response time.

1.3.2.1.1 Priority Based Scheduling

Earliest deadline first (EDF) is an optimal schedule algorithm for scheduling a set of periodic tasks with known deadlines. The algorithm operates by selecting the task with the earliest deadline from the task set, assigning priorities according to the deadlines of their current request. Then the task is enqueuing it into an execution queue. Each task is executed within a designated time window, or timeslice, which represents the period during which the task is allowed to run. If a task does not complete within its timeslice, it is preempted, allowing the scheduler to reassign the processor to the next task with the nearest deadline. The length of the timeslice is critical, as it helps balance overall system performance with individual process responsiveness.

Figure 2 is a representation of the EDF algorithm, it consists of a Waiting Queue, containing the task in waiting state. A taskset, containing the task that will be executed, and a Running Queue which are task under execution.

```
Require: $W \Leftarrow WaitQueue\ S \Leftarrow Taskset, Q \Leftarrow RunningQueue$
  while True do
    $T \Leftarrow GetNextActTask(S)$
    $QueueTask(T, Q)$
    $ExecuteNextTask(Q)$
    $CleanupTask(S)$
    if $QueueEmpty(W)$ then break;
    else$UpdateTaskSet(W, S)$
    end if
  end while
```

Figure 2. EDF Algorithm

EDF Algorithm can theoretically achieve a utilization factor of 100% i.e., The EDF algorithm is an optimal dynamic priority scheduling policy in the sense that a process set is schedulable if its CPU utilization is no larger than 100.

Public

1.3.2.2 Modifiability

To achieve modifiability in software architecture, two key strategies can be adopted. The first strategy involves minimizing the number of responsibilities assigned to individual modules, thereby increasing cohesion. The lower the number of responsibilities, lower the probability that a given change will affect other modules. The second strategy involves reducing dependencies between independent modules. Encapsulation is a effective technique for achieving this. By introducing an explicit interface to an element, all access to the element is routed through this interface, eliminating dependencies on its internal workings. As a result, changes to one element are less likely to propagate to others. The architecture design will adopt cohesion by splitting the architecture in modules with minimal responsibilities. Additionally, encapsulation will be used as a strategy to minimize dependencies between modules. In this regard, the layer pattern permits the system to be divided in such a way that modules can be developed and evolved separately with a little interaction among the parts. Each layer consists of a collection of modules with explicit responsibilities, together with a common interface used for exposing modules to higher layers.

In our architecture design, we will implement these strategies by enhancing cohesion through the division of the architecture into modules with minimal responsibilities. Additionally, encapsulation will be employed to minimize dependencies between modules. To facilitate this, the layer pattern will be used, allowing the system to be divided in a way that supports separate development and evolution of modules with minimal interaction between them. Each layer will consist of a collection of modules with explicit responsibilities, along with a common interface for communicating with higher layers. This structured approach ensures modifiability while maintaining a clear and organized system architecture.

Public

2 Architecture Views

This section provides a complete description of the system's architecture, detailing its individual components, their interactions, and their properties. The architectural views presented in this description uses the standard UML 2.0. These views are organized according to few iterations as described in the following table.

Table 1 Architecture Views per Iteration

	Viewpoints	**Description**
Iteration 1	• Modular Diagram (Figure 3) • Sequential Diagram (Figure 4) • Execution Diagram (Figure 5)	Architecture for a BareMetal application. This application initiates not only the business logic, but also system drivers.
Interation 2	• Modular Diagram (Figure 7). • Sequential Diagram (Figure 8). • Execution Diagram (Figure 9). • Implementation Diagram (Figure 10).	This iteration introduces a hardware abstraction layer (HAL). The primary purpose of implementing a HAL is to enhance both the modifiability of the system and to manage interdependencies between its components.
Interation 3	• Architecture layer distribution HAL, Middleware (Figure 11). • Modular Diagram (Figure 12). • Sequential Diagram (Figure 13). • Execution Diagram (Figure 14). • Implementation Diagram (Figure 15).	This iteration introduces a middleware layer to offer primary services to improves the system response time. In this context, we introduce a scheduling service.

Public

2.1.1 Iteration 1

The initial architectural design focuses on providing a minimal implementation that addresses core quality attributes through a two-layer structure. The architecture comprises an application layer and a low-level driver layer, each serving a distinct purpose.

As depicted in Figure 3, the application layer contains the primary application logic, while the low-level driver layer implements microcontroller drivers and provides direct access to physical resources. This design strategy grants the application comprehensive control over system resources, offering maximum flexibility in resource management.

However, this approach has significant limitations. Its effectiveness depends on two critical assumptions: first, that the application remains free from performance bottlenecks, and second, that the hardware configuration remains consistent over time. These constraints create potential vulnerabilities in the system's scalability and responsiveness.

A critical weakness emerges when introducing additional applications. As multiple applications compete for limited resources, the system's response time can deteriorate dramatically. This resource contention can severely compromise the overall system performance, making the architecture less robust in dynamic or multi-tasking environments.

The key trade-off in this architectural iteration is between direct resource control and system flexibility, highlighting the need for more sophisticated resource management strategies in subsequent design iterations.

Public

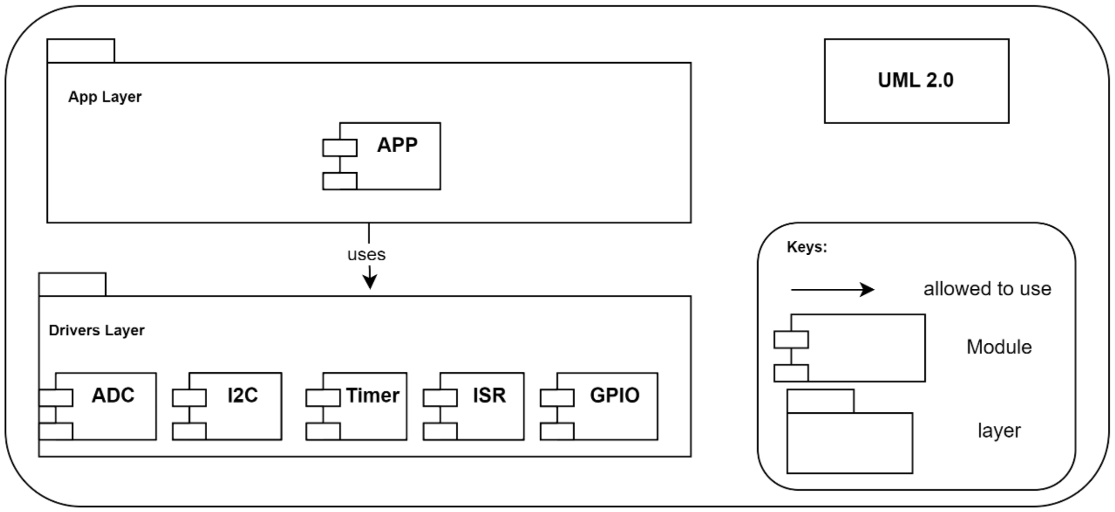
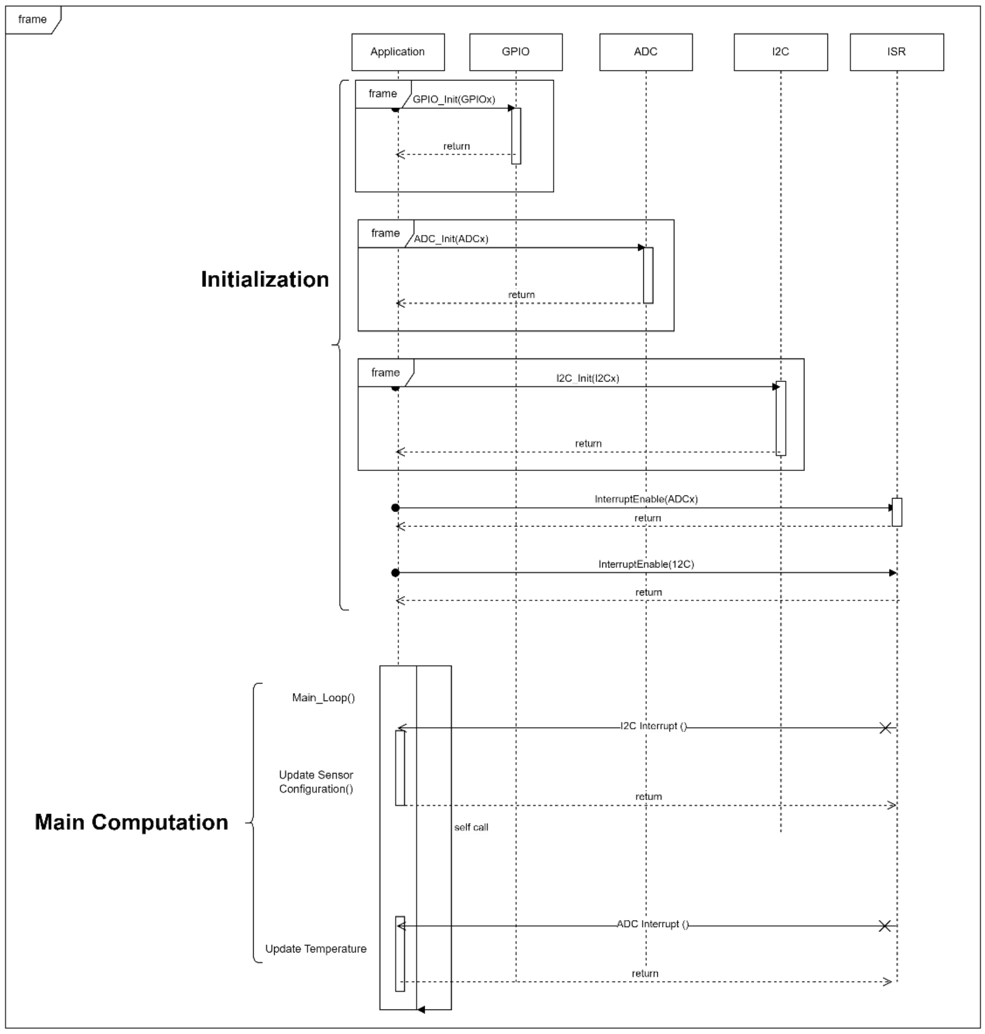


Figure 3. Modular Diagram for a Temperature Monitor



Public

Figure 4 Sequential Diagram For Temperature Monitor

The Modular View shows the distribution between application and drivers into two layers. The sequence diagram complements this view, as depicted in figure 4, by showing the initialization and the main computation procedures carried by the monitor application. As initialization, we refer to configuring GPIOs, ADC and I2C. This is achieved by a direct instantiation of peripherals in the application, as well as interrupt handling. The main computation verifies if there is a new sensor configuration from the EEPROM, and then based on the temperature value, updates the LED connected to the output GPIOs.

Public

Figure 5 illustrates the temperature monitor application from a logical perspective. As depicted in the sequential diagram, the application begins with an initialization phase where peripherals are set up and the sensor configuration is selected. After the initialization, the application enters the main loop, during which it computes the temperature and updates the LED array. It is important to note that asynchronous interrupts may occur at any time during this phase; these interrupts are responsible for updating the sensor configuration and the temperature value.

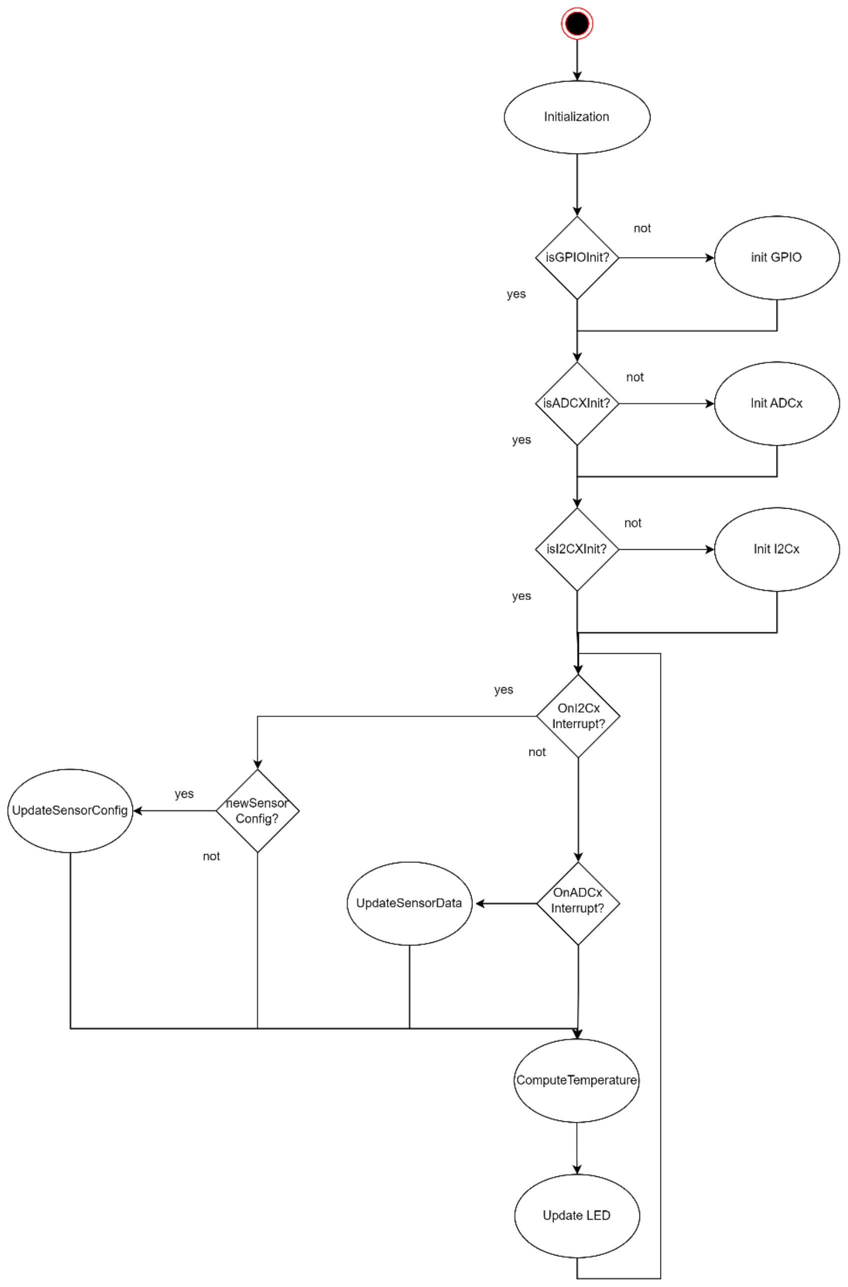


Figure 5 Logical Execution Diagram For Temperature Monitor

Public

2.1.1.1 Elements, Relations, Properties, and Constraints

The modular view illustrates the principal elements of the architecture these are represented as a following:

2.1.1.1.1 Architecture Elements

- **Application Module:** This component encapsulates the core business logic of the system and operates as a free-running element that interfaces with platform drivers. Its execution is event-driven, primarily triggered by interruptions from sensor inputs. When a new temperature value is sampled, the module immediately evaluates whether the reading falls within predefined specific ranges. Based on the temperature assessment, the module dynamically controls a set of LEDs, turning them on or off to provide visual feedback or indicate system status.

- **ADC Module, I2C Module, GPIO Module, ISR Module:** These elements represent low-level hardware driver modules that provide abstraction and control for specific physical resources in the system. Specifically, these modules include an Analog-to-Digital Converter (ADC) for converting analog signals to digital data, a Generic Input/Output (GPIO) module for managing digital pin configurations, an Inter-Integrated Circuit (I2C) hardware module for serial communication between integrated circuits, and an Interrupt Service Routine (ISR) module for handling and managing system interrupts.

2.1.1.1.2 Element Properties

The application module comprises of two functional components: an initialization function and a periodic monitoring section. The initialization function is responsible for configuring and instantiating the hardware modules, setting up the necessary parameters and connections for system operation. The periodic section serves as the core runtime logic, continuously monitoring temperature readings and dynamically controlling the LED outputs based on the detected temperature values.

Based on specification, we have derived the following hardware requirements.

Table 2 Hardware Requirements For Temperature Monitor

Hardware Module	**Requirements**
ADC	**Sampling Period**: 10kHz, 100us **Resolution**: 12Bits Num GPIO: 1. Analog Input

Public

I2C	**Mode:** Master **ClockSpeed:** >=400 kHz **Ack**: Enable
GPIO	**Num GPIOs: Set of 3bits.** **Mode:** Digital Output

2.1.1.1.3 Constrains

The implementation of these elements is contingent upon the specific platform chosen for implementation. Consequently, selecting a new platform may necessitate a complete reimplementation of these elements. Furthermore, the core business logic must execute within the sample period; otherwise, the system risks losing sampling values.

2.1.1.2 Conclusion Iteration 1

The first iteration introduces layers as a mechanism for achieving modifiability. Moreover, the use of interruptions ensures a constant sample of temperature values. However, the solution will only work under the following conditions:

1. To ensure accurate temperature sampling, it is crucial that the temperature calculation does not exceed 10 microseconds (us). Failure to meet this deadline poses a risk of missing temperature samples, as illustrated in Figure 6. The figure presents two scenarios: (a) and (b). In Scenario (a), two tasks are depicted: t1, which represents the temperature sampling function, and t2, responsible for processing the sampled value, calculating the real temperature, and controlling the LEDs. Task t1 occurs every 10 us and takes only 2 us to read the sensor value and store it in a data structure. Meanwhile, task t2 has a period of 10 us and an estimated execution time of 8 us. In this scenario, the combined computation time for both tasks does not exceed 10 us, thereby avoiding any risk of missing temperature samples. In contrast, Scenario (b) shows task t2 taking 12 us to complete its computation, resulting in missed samples (2 and 4). To guarantee the reliability of this solution, it is essential to ensure that the completion time for both tasks t1 and t2 does not exceed 10 us. Exceeding this deadline renders the system prone to failures.

Public

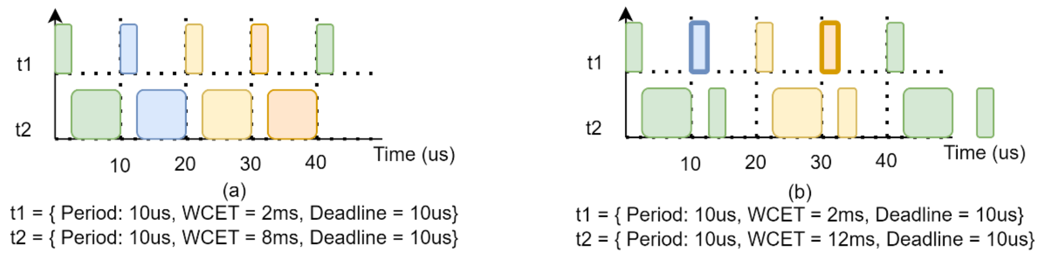


Figure 6 Timing Diagram for Temperature Monitor

2. Changes in the system´s drivers will have a drastic impact in the application development. The current architecture reveals a strong dependency between system drivers and the application, as drivers are directly instantiated from the application layer. Changes in the drive will demand require modifications in the application layer. This side effect suggests the needs for a better abstraction such that multiple hardware modules can be easily integrated, with minimal changes in the application.

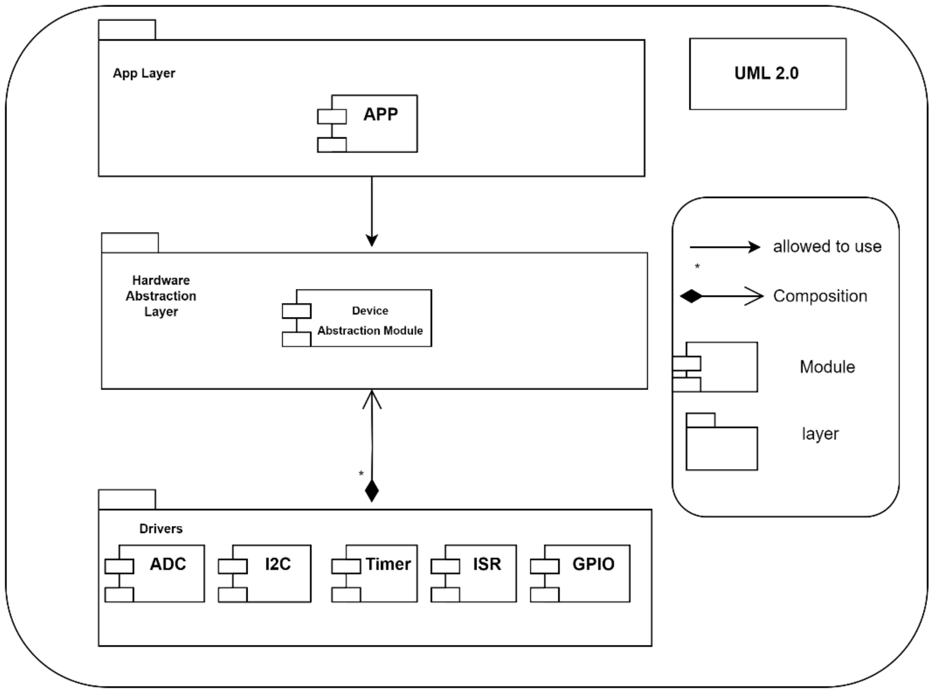
2.1.2 Iteration 2

This second iteration is focused on improving modifiability of the architecture for this we introduce a hardware abstraction layer (HAL). This layer separates the drivers from applications, making the application independent of the underline hardware. A hardware abstraction layer provides a interface for interacting with different hardware devices avoiding details of how the hardware is implemented.

As depicted in figure 7, a hardware abstraction layer has the potential to implement drivers for multiple devices, making easier to port the application to different platforms. With this implementation, changes in the application will be independent on the underline architecture.

Public

Figure 7 Module Diagram for Temperature Monitor.



The introduction of a Hardware Abstraction Layer (HAL) simplifies the development process by managing all driver-related operations within this layer. This allows developers to focus on application logic without needing to understand the intricate details of specific hardware devices. As a result, applications can be implemented more easily and ported seamlessly across different devices. Additionally, updates or changes to the HAL are often compatible with multiple versions of the same software components, ensuring broader stability and compatibility. Figure 8 illustrates these improvements in a sequential diagram, showing how the application interacts with the HAL. Notably, this interaction eliminates the need for direct driver dependencies further enhancing the application portability, modularity, maintainability, and modifiability.

Public

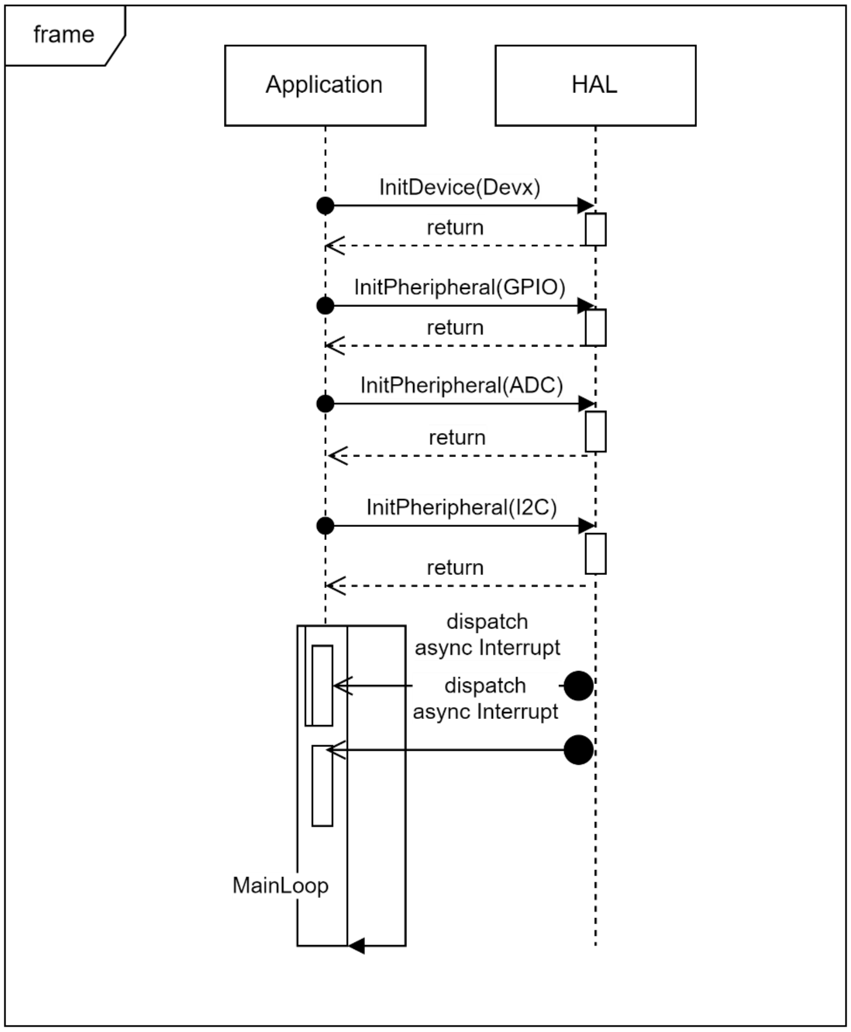


Figure 8 Sequential Diagram For Temperature Monitor

The use of a HAL layer introduces a small improvement in the application logic. This abstraction layer eliminates the need for applications to initialize drivers directly. Instead, the layer handles driver initialization internally. As a result, the application requires only few calls to be fully initialized, reducing complexity and improving efficiency. Figure 9 reflect this improvement in the logical execution diagram.

Public

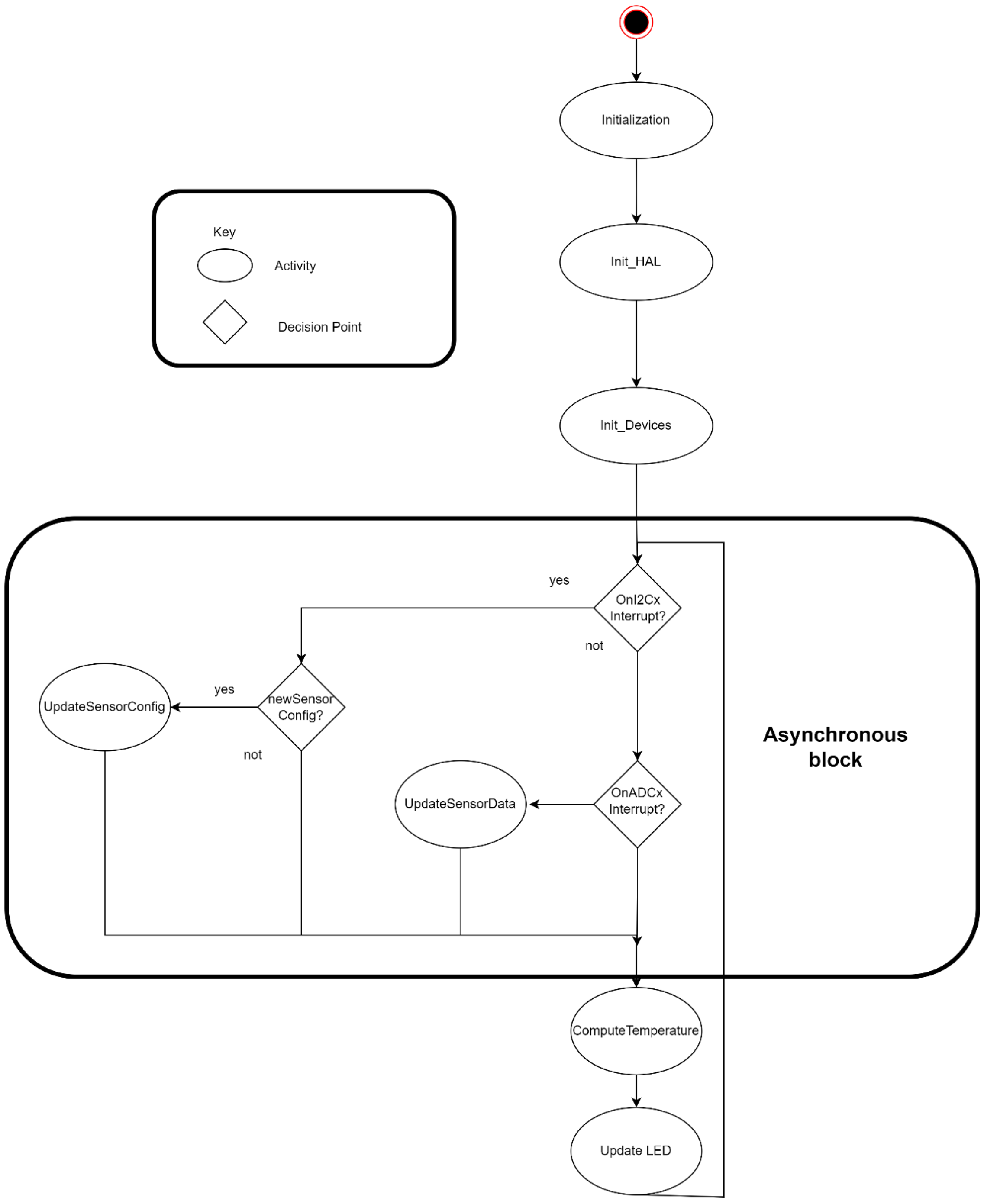


Figure 9 Logical Execution Diagram For Temperature Monitor

Figure 10 presents the implementation diagram for the Hardware Abstraction Layer (HAL). The HAL's primary function is to implement and expose a consistent interface that upper software layers can use to interact with hardware. This layer utilizes a common Device Interface to

Public

instantiate and manage multiple devices, abstracting away hardware-specific details. Each device can then instantiate one or more device drivers, the number of which is determined by the configuration of the underlying physical hardware. This design allows for flexible hardware support.

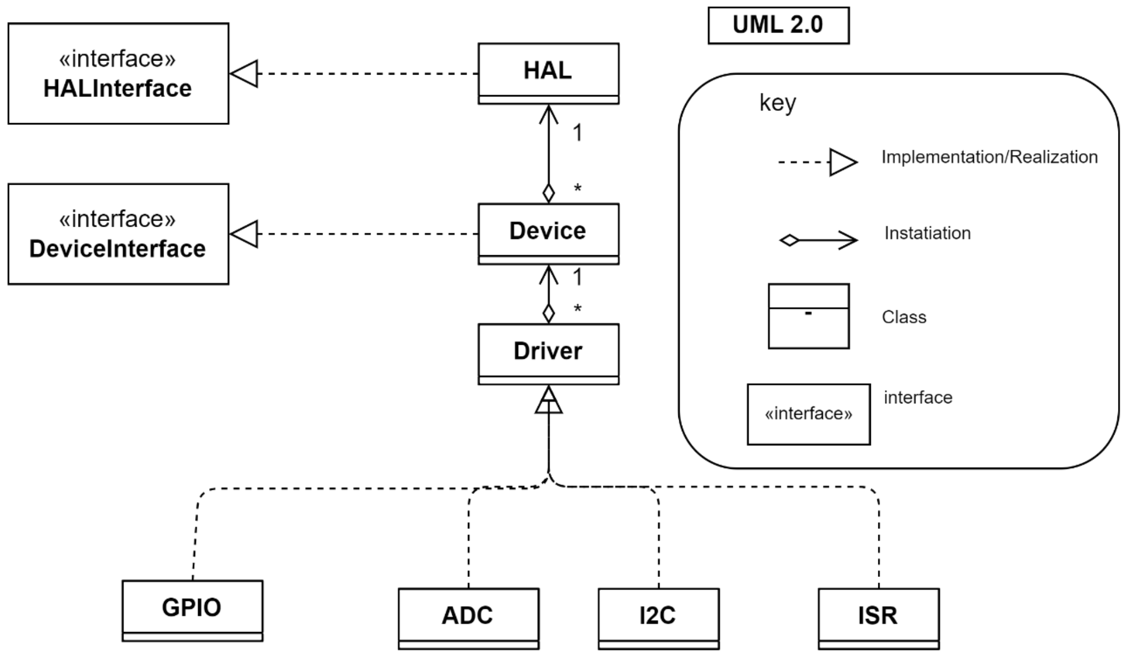


Figure 10 HAL implementation diagram for a Temperature Monitor.

2.1.2.1 Elements, Relations, Properties, and Constraints

The modular view illustrates the principal elements of the architecture at iteration 2. These are represented as a following:

2.1.2.1.1 Architecture Elements

- **Application Module:** This component encapsulates the core business logic of the system and operates as a free-running element that interfaces with the HAL layer. Its execution is event-driven, primarily triggered by interruptions from sensor inputs. When a new temperature value is sampled, the module immediately evaluates whether the reading falls within predefined specific ranges. Based on the temperature assessment, the module dynamically controls a set of LEDs, turning them on or off to provide visual feedback or indicate system status.

- **Device Abstraction Module:** A device abstraction module encapsulates platform elements, including drivers, initialization routines and event handlers. This module is part of the Hardware abstraction layer that can instantiate multiple devices and provides a common interface to the application module.

Public

- **ADC Module, I2C Module, GPIO Module, ISR Module:** These elements represent low-level hardware driver modules that provide abstraction and control for specific physical resources in the system. Specifically, these modules include an Analog-to-Digital Converter (ADC) for converting analog signals to digital data, a Generic Input/Output (GPIO) module for managing digital pin configurations, an Inter-Integrated Circuit (I2C) hardware module for serial communication between integrated circuits, and an Interrupt Service Routine (ISR) module for handling and managing system interrupts. These modules are instantiated by a device module, which contains routines for initialization and manager resources.

2.1.2.1.2 Element Properties

In addition to the Application and Driver Modules previously described, the Device Abstraction Module introduces a critical architectural approach that separates hardware-specific implementation from the core application logic. By eliminating the need for direct hardware driver implementation within the application, this module enables seamless platform compatibility and significantly simplifies system design. The abstraction layer not only facilitates easier porting across different hardware platforms but also reduces the overall code complexity by centralizing driver-related functionality, thereby streamlining code maintenance and enhancing the system's modularity.

2.1.2.1.3 Constrains

the core business logic must execute within the sample period of 100us; otherwise, the system risks losing sampling values.

2.1.2.2 Conclusion Iteration 2

This second iteration allows us to group system drivers in a single layer that abstract those to upper layers. It enables portability, hardware independence, and reduced complexity, making it easier to develop, maintain, and port software to different platforms. Additionally, it reduces development time, ensures compatibility, and facilitates easier hardware upgrades, making it easier to integrate with other systems and reducing the complexity of software development.

2.1.3 Iteration 3

The primary goal of Iteration 3 is to enhance the system's real-time capabilities by integrating a middleware layer. This layer introduces essential services, such as scheduling algorithms to ensure precise timing properties and remote monitoring functionalities for improved application oversight. These additions enable the system to handle time-critical tasks more effectively while providing greater flexibility and control over application behavior.

Public

As illustrated in Figure 11, the middleware layer operates at the same level of abstraction as the Hardware Abstraction Layer (HAL). While the HAL provides access to physical hardware resources, the middleware layer offers a set of common services that applications can utilize. Such services include communication, data exchange, scheduling. The major advantage of a middleware is its adaptability to changing requirements and its integration to various independent platforms. In addition, the middleware can offer robust security features, such as authentication, authorization, and encryption, to protect sensitive data and prevent unauthorized access.

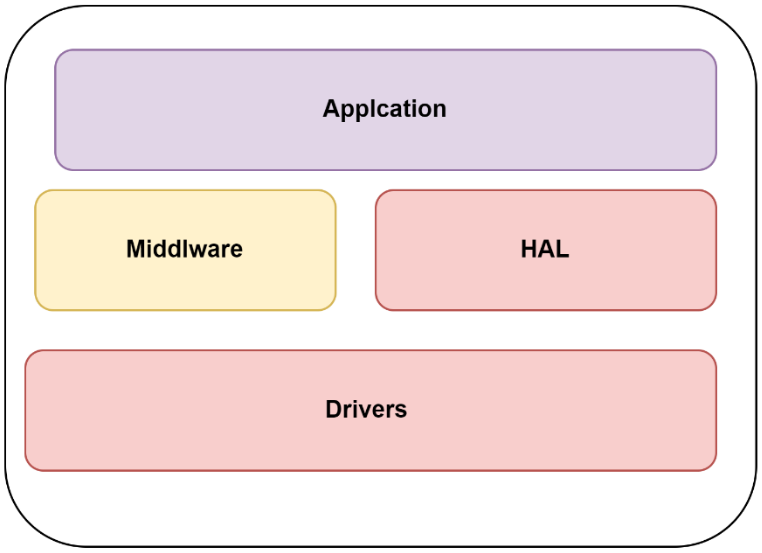


Figure 11 Layered Architecture: Middleware and Hardware Abstraction Layer (HAL)

Figure 12 illustrates the Modular view of a system that includes a middleware layer. The layer implements an Earliest Deadline First (EDF) scheduling algorithm. In contrast to the previous iteration, where application tasks could execute freely without interruption, the EDF algorithm restricts execution by preempting tasks that exceed their deadline. This ensures that only tasks with earlier deadlines are executed, prioritizing timely completion over uninterrupted execution. In addition, it is possible to detect earlier timing anomalies in the application, which is important to ensure availability.

Public

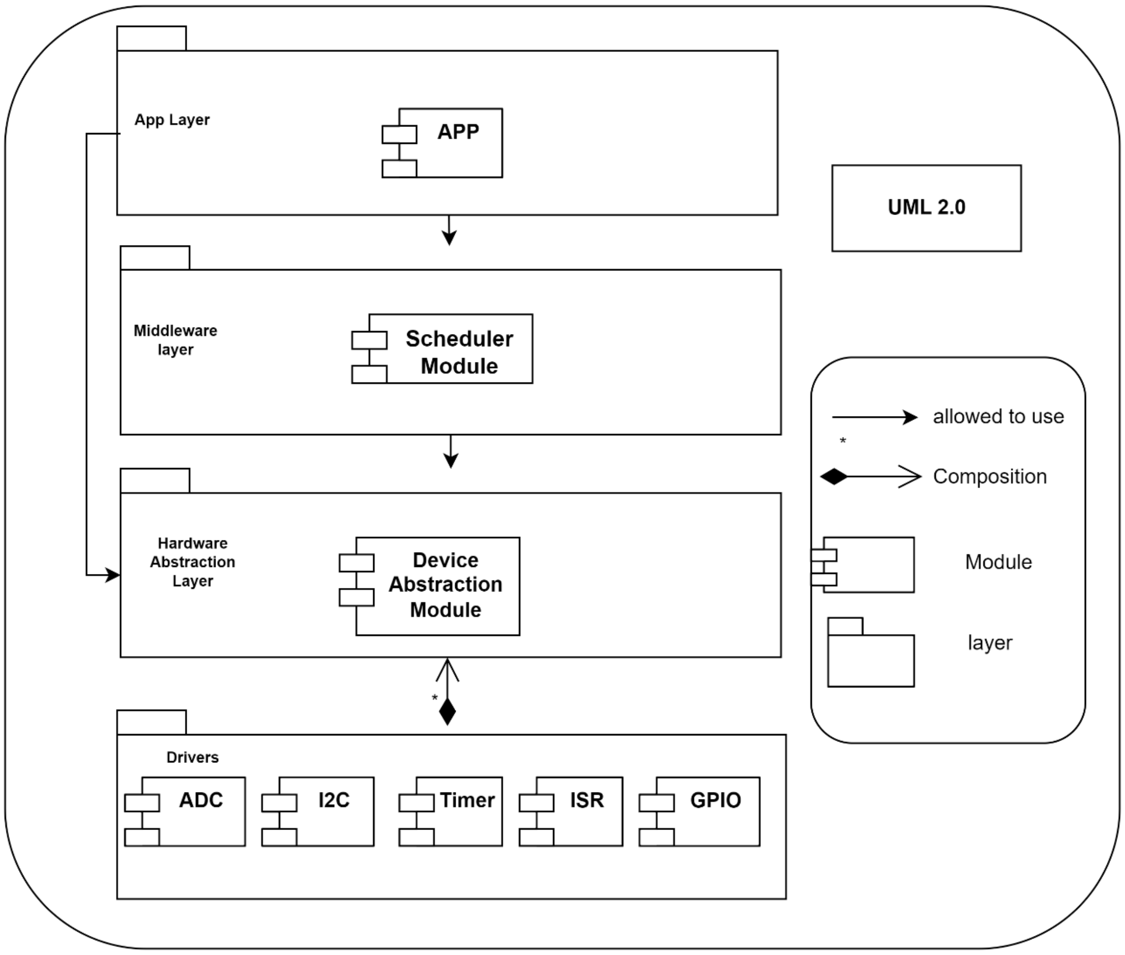


Figure 12. Modular Diagram Temeperature Monitor.

As previously introduced, the middleware layer allows for easy adaptation to evolving requirements. For example, capabilities like Availability, Energy Efficiency, Safety, and Security can be implemented and managed effectively within this layer. Consider, for instance, the addition of a monitoring component. This component could continuously evaluate the application's state, detecting potential safety violations. and trigger a rapid response, such as restarting the application.

Figure 13 show a prototype middleware that instantiate two services: A scheduling and Communication services. The schedule service is designed with a flexible scheduler interface, enabling the middleware to support multiple scheduling algorithms. This approach enhances the system's adaptability by allowing easy integration and swapping of different scheduling strategies.

Public

Similarly, the communication service is built around a communication interface, facilitating the implementation of diverse communication protocols and patterns like service-oriented communication.

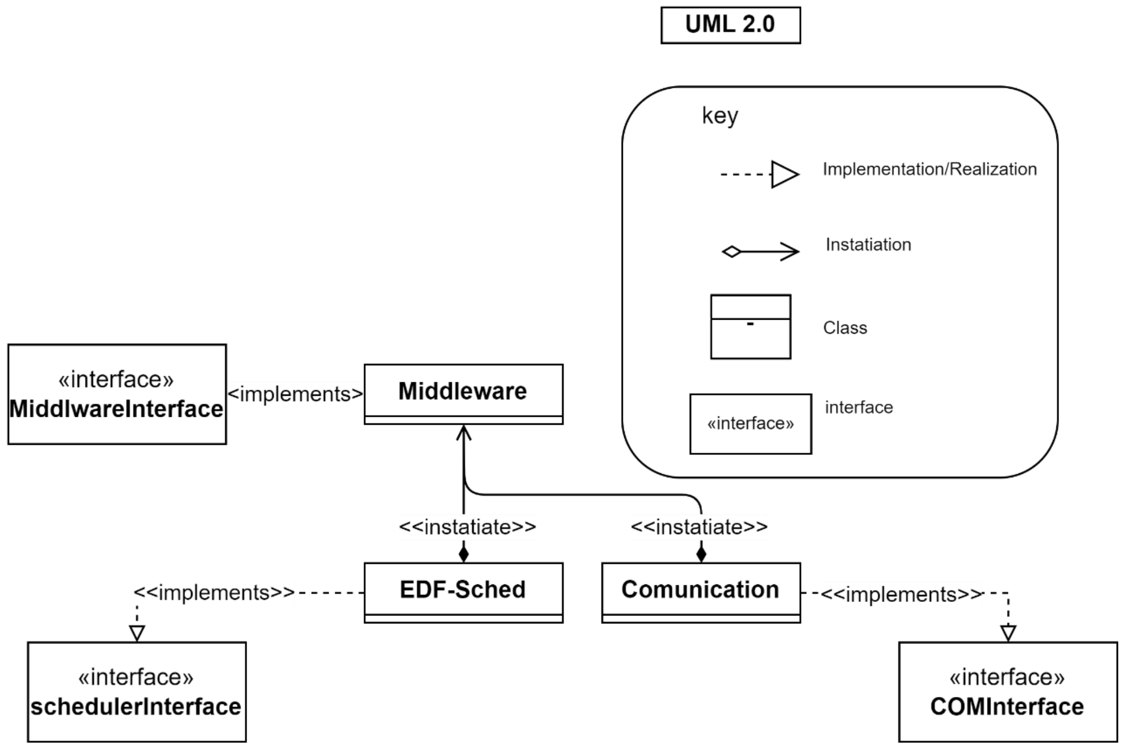


Figure 13 Middleware implementation diagram for a Temperature Monitor.

The integration of services within the middleware layer enhances the application workflow by allowing developers to express business logic more efficiently with fewer lines of code. Additionally, these services can significantly improve application performance. For example, consider an application running without scheduling mechanisms. In such cases, CPU usage can reach 100%, potentially causing system bottlenecks. However, implementing a scheduler service allows the application to run in specific time slots, maintaining CPU usage below 100% and leaving processing capabilities available for other system activities. This optimized resource management leads to better overall system performance and responsiveness.

The improvement in application workflow is illustrated in the sequential diagram shown in Figure 14, which demonstrates the interaction between the application and the scheduling service. The process consists of two main phases. During the initialization phase, the application registers two essential functions with the scheduler service: an initialization function and a periodic function. The initialization function, which is executed only once at startup, is responsible for initializing system peripherals. In contrast, the periodic function, which contains the application's business logic, is executed repeatedly according to preconfigured timing parameters (period and deadline).

Public

The scheduler service manages these executions, ensuring that the periodic function runs at the specified intervals.

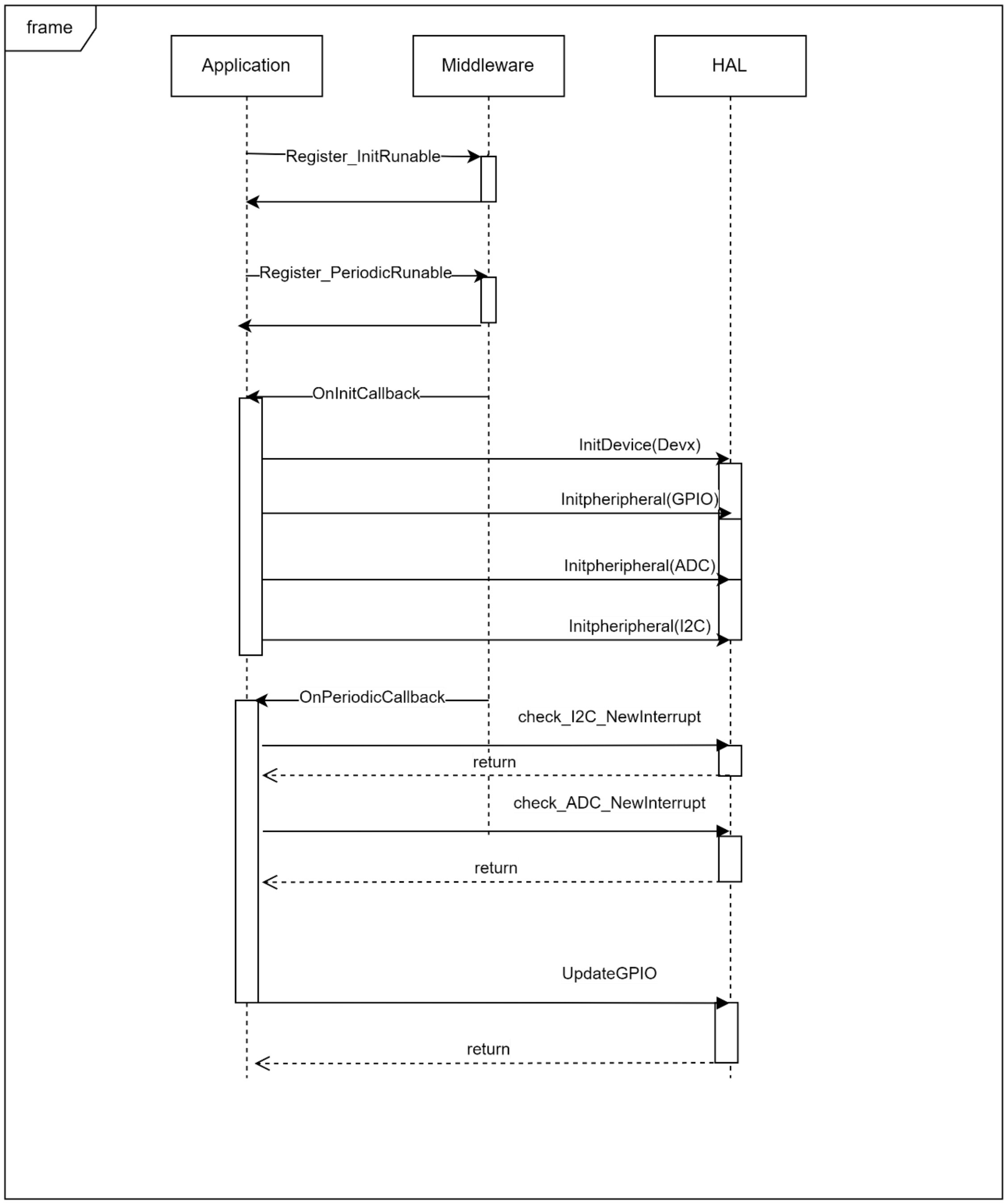


Figure 14 Sequential Diagram For Temperature Monitor

Public

2.1.3.1 Elements, Relations, Properties, and Constraints

2.1.3.1.1 Architecture Elements

- **Application Module:** This component encapsulates the core business logic of the system and operates as a free-running element that interfaces with the HAL layer. Its execution is event-driven, primarily triggered by interruptions from sensor inputs. When a new temperature value is sampled, the module immediately evaluates whether the reading falls within predefined specific ranges. Based on the temperature assessment, the module dynamically controls a set of LEDs, turning them on or off to provide visual feedback or indicate system status.

- **Device Abstraction Module:** A device abstraction module encapsulates platform elements, including drivers, initialization routines and event handlers. This module is part of the Hardware abstraction layer that can instantiate multiple devices and provides a common interface to the application module.

- **Scheduling Module:** This Module allows the architecture to support different scheduling algorithms to improve the response time of the application. Other services can also be included such as monitoring, or communication.

- **ADC Module, I2C Module, GPIO Module, ISR Module:** These elements represent low-level hardware driver modules that provide abstraction and control for specific physical resources in the system. Specifically, these modules include an Analog-to-Digital Converter (ADC) for converting analog signals to digital data, a Generic Input/Output (GPIO) module for managing digital pin configurations, an Inter-Integrated Circuit (I2C) hardware module for serial communication between integrated circuits, and an Interrupt Service Routine (ISR) module for handling and managing system interrupts. These modules are instantiated by a device module, which contains routines for initialization and manager resources.

2.1.3.1.2 Element Properties

The scheduling module is an additional element introduced to optimize system performance and resource utilization. This module ensures precise execution of the application by managing task timing specifications, effectively reducing CPU consumption and improving real-time system attributes. By intelligently allocating computational resources and controlling the execution sequence of tasks, the scheduling module enhances overall system efficiency and responsiveness.

2.1.4 Conclusion Iteration 3

Iteration 3 aims to enhance the system's real-time capabilities by introducing a middleware layer. This middleware provides critical services that improve system performance and responsiveness.

Public

Specifically, it implements advanced scheduling algorithms to ensure precise timing properties. By minimizing resource contention and optimizing the application's workflow, the middleware layer significantly enhances overall system efficiency and reliability.

Public

3 Stakeholder Representation

This section outlines the stakeholder roles considered in the development of the architecture documented within this Software Architecture Document (SAD). For each stakeholder, Table 3 lists their role and responsibilities.

Table 3 Stake Holder Roles and Responsabilities.

Stakeholder	Role	Responsibility
End User	User Validator	The end user is responsible of validating system properties: Specifically, Computation Time and response time.
Project Manager	Time, resource and budget administrator.	The Project manager should ensure that the architecture can be implemented on schedule and to budget constraints.
System Architect	System Architecture Desing and Implementation	The system architect should control the design and implementation of the architecture.
Developer	Implementer of architecture design decisions	The developer should define strategies to achieve architectural goals.
Testing Engineer	Technical Validator	Evaluate Software implementations against architecture design goals.

Public

4 Relations Among Views

Each of the views specified in Section 2 provides a different perspective and design handle on a system, and each is valid and useful. Although the views give different system perspectives, they are not independent. Elements of one view will be related to elements of other views. In this section we describe the relationship among the views chosen to represent the architecture.

4.1 General Relations Among Views

The architecture view utilizes four distinct diagram types to illustrate different aspects of the system's design and behavior. These diagrams are:

Module View: This diagram depicts the relationships between system elements, whether direct or indirect. Specifically, it emphasizes a layered architecture where independent layers interact through a common interface, promoting modularity and decoupling.

Sequential Diagram: This diagram focuses on the dynamic interactions between system elements. It illustrates how these elements communicate through function calls, which serve as the mechanism for requesting functionality across different modules. This highlights the temporal order of interactions.

Logical Execution Diagram: This diagram provides a granular view of the application's execution. It breaks down the execution process into a series of logical steps, offering a detailed understanding of how the application operates internally.

Implementation Diagram: The implementation diagram provides a detailed description of component implementation. It illustrates the relationships among classes and interfaces, along with their respective associations.

In essence, the Module View provides a static, structural overview, the Sequential Diagram highlights dynamic interactions, and the Logical Execution Diagram offers a detailed trace of the application's execution flow. Together, these diagrams provide a comprehensive understanding of the system's architecture.

Public

5 Referenced Materials

CONTENTS OF THIS SECTION: This section provides citations for each reference document. Provide enough information so that a reader of the SAD can be reasonably expected to locate the document.

Barbacci 2003	Barbacci, M.; Ellison, R.; Lattanze, A.; Stafford, J.; Weinstock, C.; & Wood, W. *Quality Attribute Workshops (QAWs)*, Third Edition (CMU/SEI-2003-TR-016). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. <http://www.sei.cmu.edu/publications/documents/03.reports/03tr016.html>.
Bass 2003	Bass, Clements, Kazman, *Software Architecture in Practice*, second edition, Addison Wesley Longman, 2003.
Clements 2001	Clements, Kazman, Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison Wesley Longman, 2001.
Clements 2002	Clements, Bachmann, Bass, Garlan, Ivers, Little, Nord, Stafford, *Documenting Software Architectures: Views and Beyond*, Addison Wesley Longman, 2002.
IEEE 1471	ANSI/IEEE-1471-2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, 21 September 2000.

5.1 Acronym List

API	Application Programming Interface; Application Program Interface; Application Programmer Interface
ATAM	Architecture Tradeoff Analysis Method
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
CORBA	Common object request broker architecture

Public

COTS	Commercial-Off-The-Shelf
EPIC	Evolutionary Process for Integrating COTS-Based Systems
IEEE	Institute of Electrical and Electronics Engineers
KPA	Key Process Area
OO	Object Oriented
ORB	Object Request Broker
OS	Operating System
QAW	Quality Attribute Workshop
RUP	Rational Unified Process
SAD	Software Architecture Document
SDE	Software Development Environment
SEE	Software Engineering Environment
SEI	Software Engineering Institute Systems Engineering & Integration Software End Item
SEPG	Software Engineering Process Group
SLOC	Source Lines of Code
SW-CMM	Capability Maturity Model for Software
CMMI-SW	Capability Maturity Model Integrated - includes Software Engineering
UML	Unified Modeling Language

Public

Appendix A Appendices

Public