UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

MAURICIO DALL OGLIO FARINA

HARDWARE-INDEPENDENT FIRMWARE DEVELOPMENT METHODOLOGY

Porto Alegre
2024

MAURICIO DALL OGLIO FARINA

HARDWARE-INDEPENDENT FIRMWARE DEVELOPMENT METHODOLOGY

Thesis presented to Programa de Pós-Graduação em Engenharia Elétrica of Universidade Federal do Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Electrical Engineering.
Area: Control and Automation

ADVISOR: Prof. Dr. Edison Pignaton de Freitas

Porto Alegre
2024

MAURICIO DALL OGLIO FARINA

HARDWARE-INDEPENDENT FIRMWARE DEVELOPMENT METHODOLOGY

This thesis was considered adequate for obtaining the degree of Master in Electrical Engineering and approved in its final form by the Advisor and the Examination Committee.

Advisor: _____
Prof. Dr. Edison Pignaton de Freitas, UFRGS
Doctor by the University of Halmstad, Sweden and by the Universidade Federal do Rio Grande do Sul, Brazil

Examination Committee:

Prof. Dr. João Cesar Netto, UFRGS
Doctor by the Université Catholique de Louvain, Belgium

Prof. Dr. Leandro Buss Becker, UFSC
Doctor by the Universidade Federal do Rio Grande do Sul, Brazil

Prof. Dr. Carlos Eduardo Pereira, UFRGS
Doctor by the University of Stuttgart, Germany

Coordinator of PPGEE: _____
Prof. Dr. Jeferson Vieira Flores

Porto Alegre, July 2024.

ABSTRACT

Unlike other forms of development, the way firmware development is designed is somewhat outdated. It is usual to come across whole systems implemented in a cross-dependent monolithic way. In addition, the software of many implementations is hardware-dependent. Hence, significant hardware changes may result in extensive firmware implementation reviews that can be time-consuming and lead to low-quality ports, which may represent an important challenge for embedded system applications that undergo frequent evolution. To address this problem, this work proposes a firmware development methodology that allows reuse and portability while improving the firmware development life cycle. In addition, the typical mistakes of a novice software developer can be reduced by employing this methodology. Moreover, these developers can get up-to-speed with the project and improve their skills much faster by following the guidelines and architecture proposed in this work. The proposed methodology defines team member roles, development workflows and a multi-layer modular architecture. Additionally, it provides guidelines, rules, standards, conventions, and best practices for firmware development. To evaluate the proposed methodology, two case studies were conducted. First, a simple IoT system project was refactored for this methodology model and the results were compared with the legacy project. Second, the comparison between legacy and new project metrics of a more complex real-world project were analyzed. In both cases, the comparison demonstrated that the proposed methodology can substantially improve portability, reuse, modularity, and other firmware factors.

Keywords: Embedded Systems, Embedded Development, Firmware Architecture, Development Methodology.

RESUMO

Ao contrário de outras formas de desenvolvimento, a maneira como o desenvolvimento de firmware é realizada é um tanto desatualizada. Não é incomum encontrar sistemas inteiros implementados de forma monolítica e com dependências cruzadas. Além disso, o software de muitas implementações é dependente do hardware. Portanto, mudanças significativas no hardware podem resultar em extensas revisões de implementação de firmware que podem consumir tempo e levar a versões de baixa qualidade, o que pode representar um problema importante para aplicações de sistemas embarcados que evoluem com muita frequência. Para lidar com esse problema, este trabalho propõe uma metodologia de desenvolvimento de firmware embarcado que permite reutilização e portabilidade, ao mesmo tempo em que melhora o ciclo de vida do desenvolvimento de firmware. Além disso, ao empregar essa metodologia, os erros típicos de um desenvolvedor de software iniciante podem ser reduzidos. Além disso, esses desenvolvedores podem ter uma rápida rampa de aprendizagem do projeto e melhorar suas habilidades de desenvolvimento seguindo as diretrizes e arquitetura propostas neste trabalho. A metodologia proposta define funções dos membros da equipe, fluxos de trabalho de desenvolvimento e uma arquitetura modular de várias camadas. Além disso, ele fornece diretrizes, regras, padrões, convenções e melhores práticas para o desenvolvimento de firmware embarcado. Para avaliar a metodologia proposta, foram conduzidos dois estudos de caso. Primeiro, um projeto simples de sistema IoT foi refatorado para este modelo de metodologia e os resultados foram comparados com o projeto legado. Segundo, a comparação entre as métricas da versões legado e novo de um projeto real e mais complexo foram analisadas. Para ambos os casos, a comparação demonstrou que a metodologia proposta pode apresentar melhorias substanciais na portabilidade, reutilização, modularidade e outros aspectos do firmware.

Palavras-chave: Sistemas Embarcados, Desenvolvimento de Sistemas Embarcados, Arquitetura de Firmware, Metodologia de Desenvolvimento.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ABBREVIATIONS

3JAR	3 Joint Arm Robot
API	Application Programming Interface
ARCH	Architect
ARIC	Arm Robot Interface Class
ASMI	Average Simple Maintainability Index
CORE	Core Project
COUP	Module Coupling
CPU	Central Processing Unit
CS1	Case Study 1
CS2	Case Study 2
DEV	Developer
DMA	Direct Memory Access
ELOC	Effective Lines of Code
ELOCFI	Effective Lines of Code per File
ELOCFU	Effective Lines of Code per Function
ESP-DIF	Espressif IoT Development Framework
FIFO	First-In First-Out
GPU	Graphics Processing Unit
HAL	Hardware Abstraction Layer
I/O	Input/Output
I2C	Inter-Integrated Circuit
IC	Integrated Circuit
IDE	Integrated development environment
IoT	Internet of Things
IRQ	Interrupt Request
ISR	Interrupt Service Routine

LCD	Liquid Crystal Display
LEGP	Legacy Project
LOC	Lines of Code
LOCFI	Lines of Code per File
LOCFU	Lines of Code per Function
LTG	Luminator Technology Group
MAIN	Maintainer
MCU	Microcontroller Unit
MMU	Memory Management Unit
MNL	Maximum Nested Level
MPU	Microprocessor Unit
MVG	McCabe's Cyclomatic Complexity
NEWP	New Project
NFC	Novice Firmware Choice
NUMA	Non-Uniform Memory Access
OOC	Object Oriented C
OOP	Object Oriented Programming
OPS	DevOps
OS	Operating System
RAM	Random Access Memory
RT	Real-Time
RTOS	Real-Time Operating System
SMI	Simple Maintainability Index
SPECS	Target Project Domain Specifications
TARGET	Target Project
TEST	Tester
TELOC	Total Effective Lines of Code
TLOC	Total Lines of Code
UART	Universal Asynchronous Receiver / Transmitter

CONTENTS

1 INTRODUCTION

Application software developers have many frameworks and methodologies with ready-made solutions and tons of information on how to architect, structure, develop, and maintain their systems. In contrast, embedded software developers have to deal with different hardware platforms and vendor-directed frameworks that commonly result in projects with customized and specialized architectures that usually do not apply to other projects. In light of that, embedded software still requires improvements in the development process and methodology to provide a more flexible and robust firmware development process like other more mature software areas (FAHMIDEH *et al.*, 2022)(SOLOVEV; YULDASHEV, 2023).

The firmware that is currently being developed is written in a somewhat outdated manner. Each product development cycle is limited to no code platforms or reuse, with reinvention being a significant concern among development teams. A common example is when development teams opt to develop their own in-house scheduler instead of using an available RTOS (BENINGO, 2017). Similarly, teams may also develop custom protocols for their IoT devices rather than making use of available ones such as Matter (CONNECTIVITY STANDARDS ALLIANCE, 2024). Another problem is that most firmware is set in a cross-dependent monolithic way. As a result, it becomes tricky to debug, maintain, and introduce new features into the software while it becomes less reusable and portable. This type of undocumented change is common on open-source platforms, where each programmer changes features according to their preferences.

Novice developers often lack guidance when developing firmware, which, coupled with their inexperience, may compromise their understanding of the firmware architecture. Also, there exists a noticeable gap between the skill set expected by industries from embedded system professionals and the curriculum offered by the universities (DEEPA *et al.*, 2024). As a consequence, when attempting to add a new feature or fix a bug, these developers may introduce their changes in the wrong places or in a non-standard manner. As a result, bugs may be introduced in areas outside the expected code context, reducing the quality and maintainability of the firmware.

To mitigate this problem, it is necessary to establish a clear and standardized path for

firmware development, which can be followed by both novice and experienced developers. To address that, methodologies and frameworks have been objects of research over the years. They make it possible to abstract low-level details, allowing developers to focus on a specific functionality without dealing directly with matters of complexity, which enables the development of new applications. These advantages are possible because their development involves the employment of best practices in software engineering, such as the use of design principles for writing clean and high-quality code (MARTIN, 2017), applying design patterns, and defining a robust, scalable, and reusable architecture (FOWLER, 2012) that dispense with the most tedious tasks associated with MCU development (TREMAROLI, 2023).

Most proposals in the literature provide methods that depend on conventional techniques adapted from software engineering. However, since embedded software has notable differences from traditional software, researchers have been tasked with proposing methodologies that can specifically cover the firmware lifecycle aspects (GUERRERO-ULLOA; RODRÍGUEZ-DOMÍNGUEZ; HORNOS, 2023).

For example, even though many embedded devices are designed to last for a short time, they constantly face unexpected situations that may reduce their lifespan. One scenario is unforeseen circumstances that may require devices to adopt extra routines, shortening their battery lives. For example, in a Wireless Sensor Network, an IoT node close to the sinks may be overwhelmed by high traffic, resulting in energy overheads (HORSTMANNM *et al.*, 2023). Moreover, power failures are the norm rather than accidental (JIA *et al.*, 2022), and, thus, inevitably, all these devices will soon reach the end of their life cycle, some, early.

Firmware maintenance usually takes up a large portion of a product's life cycle and it is needed to correct design errors and to adapt the software to the never-ending needs of an evolving application scenario (KOPETZ; STEINER, 2022). Implementing automated testing may substantially reduce costs, but conducting automated tests for embedded systems is a considerable challenge, and in the case of disorganized firmware, it can become an unsustainable task.

At all events, device replacement is a "bound-to-happen" issue that must be addressed. However, at the time of replacement, the components used in the original devices may no longer be available. Several factors may drive companies to change their products to ensure their availability (FARINA; DOS ANJOS; DE FREITAS, 2023). For example, DUNN (2021) discusses the impacts of the COVID-19 pandemic on global supply chains, highlighting how the microchip shortage has forced companies to revise consolidated projects to make use of alternative components available on the market. However, in many situations, unless mechanisms that abstract and encapsulate platform-specific elements are supported, the software developed for a given platform will seldom be portable to a different one (FRÖHLICH; WANNER, 2008). In that direction, a methodology

that considers the specificity of embedded systems, such as hardware and communication mechanisms, would be a significant contribution to the industry (FERREIRA *et al.*, 2022).

Seeking solutions, the Hardware Abstraction Layer (HAL) is a well-established concept that has been widely used in embedded systems development and can help address those challenges. HALs simplify the underlying hardware usage by abstracting its inherent complexity in a more ergonomic API. As a result, they can enhance firmware portability, facilitate software reusability, and reduce overall development costs (SIMMANN; VEERANNA; KRIESTEN, 2024)(KRÖNING, 2023). However, even well-architected firmware may lose its reusability and portability if developers violate or bypass conventions or if inexperienced developers make unsupervised changes to the production code.

1.1 Objectives and Contribution

This work attempts to bridge those gaps by introducing a firmware development methodology to support teams in designing, developing, and maintaining firmware. By providing standardized modules based on different concepts and structuring them into a multi-layer hierarchical structure, the methodology defines a generic modular architecture that can decouple several parts of systems. As a result of that, problems become simpler to isolate, and new features can be easily introduced. Besides improving maintainability, the methodology also enables projects to be portable and share reusable modules.

However, achieving and maintaining high-quality firmware requires further care besides describing an architecture. Therefore, the proposed methodology also defines roles, workflows, guidelines, and standards that were designed to simplify and organize operational aspects, capacitate inexperienced developers, and promote collaboration. With a clear understanding of the development process and a facilitated comprehension of the system architecture, developers can become less prone to diverge from the architecture definitions, structures, and rules, which results in fewer mistakes and more consistent outcomes.

In the 1980s, Brooks and Bullet declared that *there is no silver bullet for software development* (BROOKS, 1987), which is still applicable today. The embedded world covers an extensive universe that requires different development approaches. The proposed methodology aims at a fraction of MCU-based systems where the firmware resources are less constrained and can be developed in a more generic format. For example, modern MCUs are designed to address the most common requirements for a given group of applications. Consequently, systems may require a given MCU based on the provided peripherals rather than a minimal processor power or memory size.

From managing limited resources to timing constraints of real-time systems, embedded software often presents particular requirements (PASSIG HORSTMANN; CON-

RADI HOFFMANN; FRÖHLICH, 2023). Projects that apply the proposed methodology are expected to present extra overhead compared with hardware-specific designs. However, this is acceptable since these systems are not expected to exist at the edge of the available MCU resources. By sacrificing some extra resources available, projects can then benefit from a more modular and abstract architecture.

The structure provided by the proposed methodology allows projects to be designed for mono-thread (Bare-Metal) or to make use of RTOS for multi-thread capabilities. Although RTOS can be applied, this methodology was not designed to address the implementation of safety-critical systems or hard real-time approaches. Finally, systems based on application processors or sophisticated SoM equipped with advanced features, including GPU, NUMA, or multiple heterogeneous core architectures, are beyond the scope of this approach.

The main research contributions of this study are as follows:

- Provide a firmware development methodology that improves processes and life cycle phases of embedded projects;
- Define a standardized and generic modular architecture for firmware that improves portability, reuse, and maintainability; and
- Establish development standards and lay down guidelines for firmware development to reduce the risk of mistakes being made by novice developers while not limiting or restricting the scope of advanced developers;

To evaluate the quality of the resulting code developed and the benefits introduced by this methodology, code metrics were used to compare two projects with their respective legacy versions. For the first project, an author's previous work was used to demonstrate the use of the methodology and evaluate results for a small-scope project. The second project is based on a real product, and its evaluation indicates the improvements of using the methodology in a large-scope project.

1.2 Work Organization

This work has 6 chapters:

Chapter 1 presents the contextualization of the theme and the objectives of the work.

Chapter 2 presents essential embedded development concepts that support the development of the proposed methodology.

Chapter 3 presents a directed study on related works, along with a comparison with the present work.

Chapter 4 presents the complete methodology structure, team roles, workflows and architecture.

Chapter 5 presents two case studies to apply and evaluate the methodology. The first case demonstrates the methodology and evaluates the results. The second case evaluates the results of a real-world system.

Chapter 6 presents the conclusions obtained with the development of the methodology and the implementations of the proposed case studies. Finally, directions for future work are discussed.

2 BACKGROUND CONCEPTS REVIEW

2.1 Firmware

Firmware can be defined as software that can instruct the hardware to perform functions (CROSSLEY, 2024). Once a device is turned on, it is the first piece of code that runs on the target hardware. In this process, the operations performed may differ according to each target. Still, the fundamental function of the firmware is to perform the bare minimal hardware initialization and either wait for a host-centric communication to initiate or hand off control to the high-level system software (BANIK; ZIMMER, 2022).

In the embedded system world, the growth of intelligent physical products such as Internet of Things (IoT), Industrial IoT (IIoT), and operational technology enabled the expansion of the definition of firmware. Although historically it was used for lower-level functionalities, in embedded systems, firmware may be a standalone software that bundles all the system's lower-level and higher-level functionalities. In that sense, firmware is also known as embedded software (CROSSLEY, 2024).

Firmware is not intended to be modified by the user. However, even though it is usually programmed into the device during the manufacturing process, firmware can be designed to be upgradable. Firmware updates can fix bugs, improve performance, and add new features to the device, all without the need to replace any hardware (THIRUMALAI, 2023).

2.2 Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects, which are instances of classes. A class defines a blueprint for an object, encapsulating data (attributes) and behavior (methods) within a single, cohesive unit (GRADY BOOCH ROBERT A. MAKSIMCHUK, 2007). This approach emphasizes the concepts of:

- **Encapsulation** refers to the bundling of data and methods within a single instance, and restricting access to some of its components.

- **Inheritance** allows classes to derive from other classes, promoting code reuse and creating a natural hierarchy between base and derived classes.

- **Polymorphism** enables objects of different classes to be treated as objects of a common super class. It allows one interface to be used for a general class of actions, making it easier to extend and maintain applications.

- **Abstraction** is the process of hiding the complex implementation details of an object and exposing only the necessary aspects, simplifying interaction with the object.

2.2.1 Object-Oriented C

Modern programming languages use OOP to allow modularity and decoupling. However, the C language is a procedural language that does not natively support it. To address that, Object-Oriented C (OOC) is an approach that provides a way to organize C programs using OOP principles (NESER; VAN SCHOOR, 2009).

While C does not have native support for classes and objects like C++, developers can implement OOP concepts by using structures (structs) to represent objects and function pointers to simulate methods (QUANTUM LEAPS LLC, 2020). In Object-Oriented C, a struct typically contains the object's attributes, and function pointers are used to point to functions that operate on those attributes, mimicking methods in traditional OOP languages.

Encapsulation can be achieved by defining structs and associated functions in a way that restricts direct access to data, allowing only designated functions to modify the state. Inheritance can be simulated by including a base struct within a derived struct, allowing the derived struct to access the base struct's members. Polymorphism can be achieved by using function pointers in the base struct that can be overridden in derived structs (AMINI, 2019).

2.3 Hardware Abstraction Layer

Microcontroller software tends to be designed for a highly specific task with little room for scalability or code reuse (TREMAROLI, 2023). To avoid this limitation, a separation between hardware-dependent and independent implementation is a highly desirable task. With that in mind, BENINGO (2017) considers the HAL, the most interesting firmware layer available to developers.

ECKER; MüLLER; DöMER (2009) defines that the HAL represents the thin software layer that depends entirely on the underlying target architecture. Structuring the hardware-dependent software in these well-defined layers, accessible through the application programming interface, is essential to support software flexibility and portability

on different hardware platforms. Similarly, BENINGO (2017) defines a HAL as an interface that provides the application developer with a standard function set that can be used to access hardware functions without an in-depth understanding of how the hardware works. Finally, ECOS (2024) describes a HAL as all the software directly dependent on the underlying hardware. Whenever the hardware architecture is changed, adjustments to the HAL are required.

The definition of generic HAL APIs for the target application domain makes it possible to start designing the software before the hardware is complete, thus enabling concurrent hardware and software design. The portability also facilitates the exchange of the software code and architecture exploration, e.g. experimenting with different types of processors to find an optimal target processor. The HAL includes two types of software code (ECKER; MüLLER; DöMER, 2009):

- Processor-specific software code, such as context switch, boot code or code for enabling and disabling the interrupt vectors.

- Device drivers, which represent the software code for configuring and accessing hardware resources, including MMU (Memory Management Unit), system timer, on-chip bus, bus bridge, I/O devices, and resource management, such as tracking system resource usage (check battery status) or power management (set processor speed).

2.4 Real-Time Systems

Typically, embedded systems are crafted with specific timing constraints in mind, including the need for deterministic responses to interrupts and efficient completion of interrupt processing. However, while these timing parameters serve as benchmarks, they do not ensure universal feasibility. Conversely, embedded systems tailored for real-time applications must adhere to exceptionally stringent timing criteria, guaranteeing response times to interrupts and timely completion of all requested services within predefined limits. This operational context closely resembles that of a true real-time system (WANG, 2017a).

Determinism is a fundamental characteristic of real-time systems that ensures operations are executed predictably within specified time constraints. It means that the timing behavior of the system is consistent and predictable, allowing jobs to be completed within their deadlines regardless of external conditions (SHIN; RAMANATHAN, 1994). This predictability is crucial in safety-critical environments where failure to meet timing requirements can lead to catastrophic outcomes.

Achieving real-time determinism involves careful design considerations, such as scheduling algorithms, resource management strategies, and hardware configuration. Fixed-

priority preemptive scheduling, where tasks are assigned fixed priorities and higher-priority tasks can interrupt lower-priority ones, is commonly used in real-time systems to enhance determinism (STANKOVIC, 1996). Also, the use of RTOS further supports the deterministic behavior of the system.

Even non-real-time embedded systems can face time constraints. Consider a scenario where a TV remote control takes over 5 seconds to transmit a command to a TV, followed by an additional 5-second delay as the embedded device within the TV switches channels. Naturally, such delays would lead to dissatisfaction. Consumers rightfully anticipate a TV to respond to remote control inputs within a 1-second timeframe. Nonetheless, these constraints primarily gauge system performance (WANG, 2017b).

Real-time systems are tasked with computing and delivering accurate results within a predetermined timeframe. Making it simpler, every task in a real-time system operates under a deadline, whether it's classified as hard or soft. Missing a hard deadline renders the result useless, regardless of its accuracy. Take, for instance, the airbag control system in automobiles. These systems are engineered to deploy airbags swiftly in the event of frontal impacts. Given the rapid changes in vehicle speed during a crash, airbags must inflate rapidly to mitigate the risk of occupants colliding with the vehicle's interior. Typically, the entire deployment and inflation process occurs within approximately 0.04 seconds, well within the prescribed limit of 0.1 seconds (WANG, 2017b).

2.4.1 Soft and Hard Real-Time Constraints

Numerous methods for categorizing real-time systems have been proposed and are currently utilized. One pragmatic approach revolves around time and criticality (COOLING, 2019). Typically, the timing constraint of a task is defined by its deadline, representing the moment at which its execution or service must be completed. Depending on the severity of missing a task deadline, the timing constraint can be classified as either hard or soft (FAN, 2015):

- A timing constraint is considered soft when the repercussion of a missed deadline is unfavorable but manageable. Even if a response arrives late, it remains beneficial as long as it falls within an acceptable range, meaning it occurs occasionally with a suitably low probability.

- A timing constraint is deemed hard when the outcome of a missed deadline is catastrophic. In such cases, a delayed response, or the completion of the requested task beyond the deadline, is not only useless but can also be entirely unacceptable.

Hard real-time systems operate under strict deadlines where the consequences of missing a deadline can be severe or even fatal. The system must ensure that every task meets its deadline without exception, and the timing of each task must be thoroughly analyzed

and verified during the design phase (INTERVALZERO, 2019)(SHIN; RAMANATHAN, 1994).

Soft real-time systems also require deterministic behavior, but they offer more flexibility. Missing a deadline in these systems may degrade performance or reduce the quality of service, but it does not result in system failure. For example, in multimedia streaming, slight delays in processing can lead to a drop in video quality or buffering, but the system continues to function. However, even in soft real-time systems, maintaining a high degree of determinism is essential to ensure a satisfactory user experience (KOPETZ; STEINER, 2022).

Different from soft and hard real-time, some systems can also be classified as near real-time. This type of system can be described as a high-response sensitivity system subjected to a negligible processing delay. Near real-time is usually used in cases where time-critical responses are required, but the determinism required by hard real-time systems is unnecessary (GOMES *et al.*, 2021).

2.4.2 Bare-Metal

Bare-metal software refers to software that runs directly on the hardware of a system without the intervention of an operating system or a higher-level software layer (OUALLINE, 2022). This type of software interacts directly with the hardware components, such as the CPU, memory, and peripherals, allowing for precise control over the system's resources. Bare-metal software is often used in embedded systems, real-time applications, and environments where performance, low latency, and minimal overhead are critical (BARR; MASSA, 2006). Since there is no operating system to manage resources, the developer must handle tasks like memory management, task scheduling, and interrupt handling, which can lead to highly efficient but complex code.

2.4.3 Real-Time Operating Systems

A general-purpose OS serves as system software, overseeing both user applications and the computer's hardware resources. It establishes regulations and programming interfaces, enabling programs to request OS services and interact with the system at large. On the other hand, an RTOS is fundamentally an OS comprising multiple software subsystems with its kernel at the nucleus (FAN, 2015).

Even though an RTOS may offer numerous services akin to those in general-purpose OS, an RTOS should be temporally predictable (deterministic) (KOPETZ; STEINER, 2022). In contrast to operating systems for personal computers, real-time systems deadlines must always be met (especially in hard real-time). For that, it is crucial to ensure that a well-designed RTOS, uniquely engineered with specialized scheduling algorithms tailored for real-time applications, is used (GRACIOLI *et al.*, 2013).

To adhere to rigorous timing demands, RTOSes are typically engineered with the fol-

lowing capabilities (WANG, 2017a):

1. Minimum interrupt latency: Interrupt latency measures the time interval from the instant an interrupt is received until the CPU initiates execution of the corresponding interrupt handler. To mitigate interrupt latency, an RTOS kernel should avoid blocking interrupts for extended periods. This typically necessitates support for nested interrupts within the system to ensure that the low-priority interrupt handling does not hinder the processing of high-priority interrupts.

2. Short critical regions: In all OS kernels, critical regions safeguard shared data objects and facilitate process synchronization. However, in the context of an RTOS kernel, it is imperative to keep these critical regions as brief as possible.

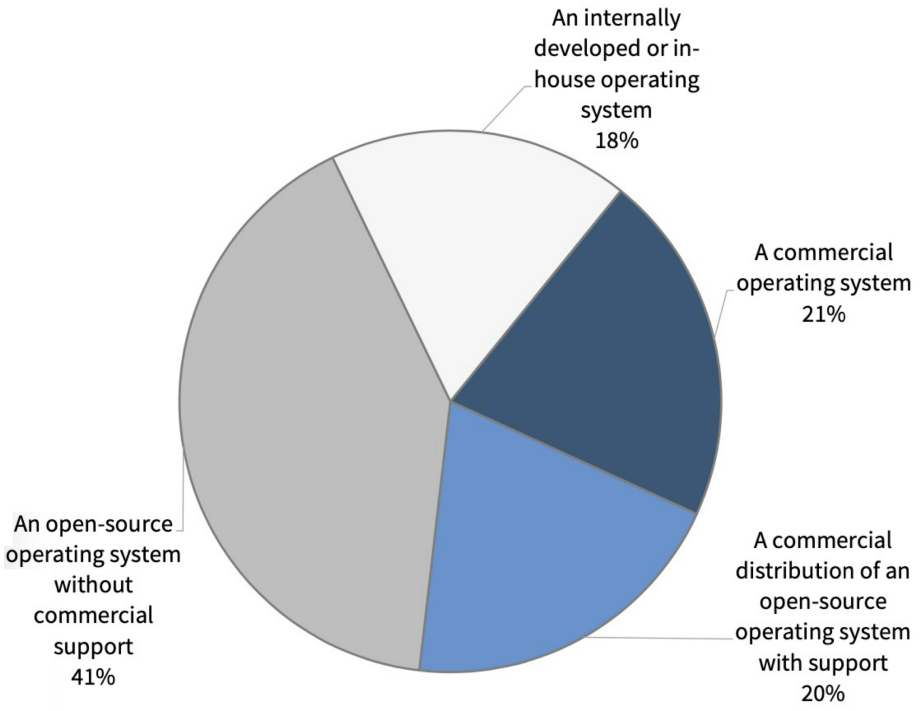
3. Preemptive task scheduling: Preemption in an OS context signifies that a task with a higher priority can interrupt and take precedence over a lower-priority task at any given moment. To effectively meet task deadlines, an RTOS kernel necessitates support for preemptive task scheduling. Additionally, minimizing the duration of task switching ensures efficient operation.

4. Advanced task scheduling algorithm: Preemptive scheduling is a crucial but not sole requirement for real-time systems. Without preemptive scheduling, high-priority tasks could encounter delays caused by low-priority tasks, rendering it unfeasible to meet task deadlines consistently. Nevertheless, even with preemptive scheduling, there's no assurance that tasks will consistently meet their deadlines. To address this challenge, the system must employ a suitable scheduling algorithm designed to achieve this objective.

As embedded systems evolved into more sophisticated and complex systems, RTOSes became widely used tools for embedded development. As of 2023, OSes are present in 74% of ongoing embedded projects and have a projection of being part of 86% in the near future (ASPENCORE, 2023). RTOS can be separated into four categories (Figure 1): open-source, commercial, commercial distribution of open-source, and in-house.

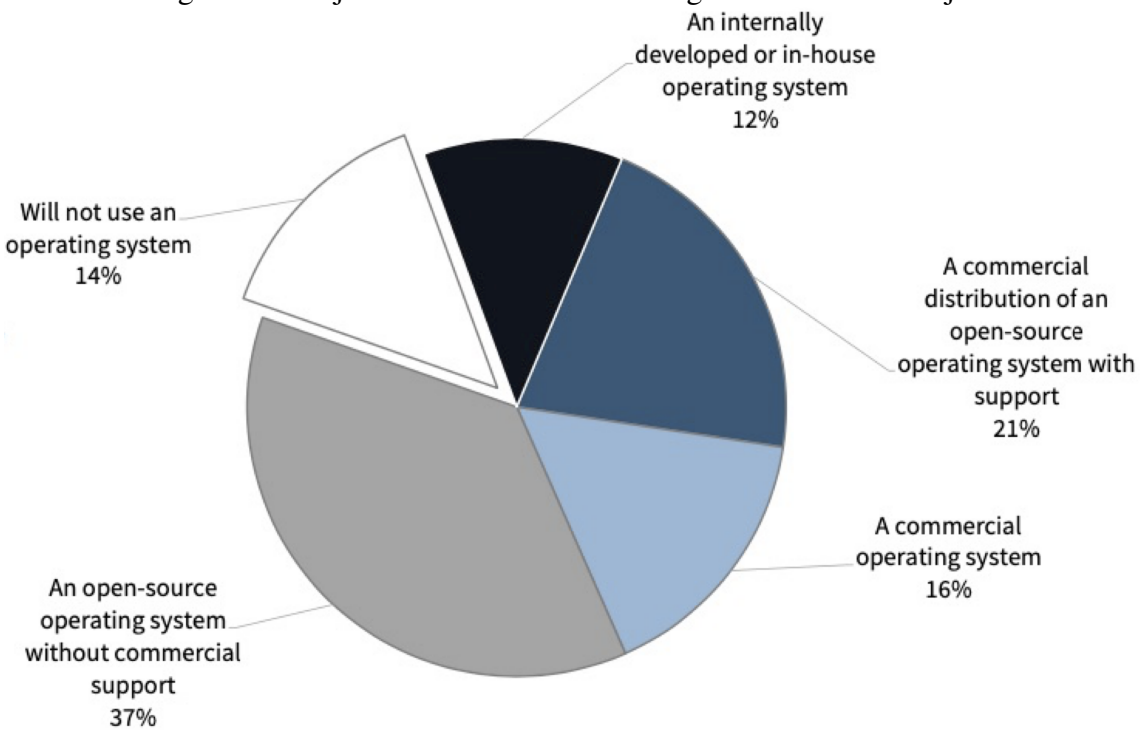
In recent years, big tech companies have acquired popular embedded RTOSes, boosting their popularity in the development community. For example, FreeRTOS and ThreadX were acquired by Amazon (AMAZON WEB SERVICES, 2024) and Microsoft (MICROSOFT, 2019), respectively, as the official embedded OS for their cloud platforms. As a result, companies become more susceptible to adopting open-source instead of in-house or commercial solutions (Figure 2). Finally, the top most popular embedded RTOSes used today (2023) are (ASPENCORE, 2023): FreeRTOS, RTX, ThreadX, Texas Instruments RTOS, and QNX.

Figure 1 – OS Used in Current Embedded Project



Source: ASPENCORE (2023)

Figure 2 – Projection of the future OS usage in Embedded Project



Source: ASPENCORE (2023)

2.5 Quality Attributes

Quality attributes, also known as non-functional requirements, are characteristics of a system that describe how well it performs its intended functions. While functional requirements specify what the system should do, quality attributes define how the system should behave, ensuring that it meets the needs of users, stakeholders, and developers in terms of performance, reliability, usability, and more (SOMMERVILLE, 2016). Quality attributes are often interrelated, and achieving a balance between them is essential for creating robust and efficient software. These attributes are typically evaluated using metrics to ensure that the developed software achieves the desired expectations throughout its lifecycle (ROGER; BRUCE, 2019).

2.5.1 Modularity

Modularity measures the degree to which the components of a system can be separated and recombined. In software development, modularity means designing a system in a way that it is divided into discrete, self-contained modules, each with a specific responsibility. High modularity allows developers to work on individual modules independently, making the code easier to understand, test, and maintain. It also enhances reusability since modules can often be reused in different parts of the system or even in different projects (GRADY BOOCH ROBERT A. MAKSIMCHUK, 2007).

2.5.2 Portability

Portability is the ability of software to run on different platforms or environments with little or no modification. High portability means that the software can be easily transferred from one system to another or from one type of hardware to another. Portability is achieved through platform-independent code. This quality attribute is especially important in today's diverse embedded environments, where firmware may need to operate on different MCUs (BENINGO, 2022). High portability reduces the time and cost associated with adapting software to new environments (COOLING, 2019).

2.5.3 Maintainability

Maintainability measures how easy a software can be modified to correct problems, include new features, or adapt to a changed environment. It includes several factors, such as code readability, documentation quality, and the simplicity of its design. High maintainability means that the software is easier to understand and modify, making it less prone to errors during modifications, leading to greater reliability and reduced downtime (SOMMERVILLE, 2016).

2.6 Code Evaluation Metrics

To evaluate quality attributes, software metrics have played an increasingly important role in software engineering searching for better source code quality. Metrics allow the measurement, evaluation, control, and improvement of software products and processes (DE BASSI *et al.*, 2018). Also, the validation of software metrics shows that metrics allow conclusions on the quality of software (BARKMANN; LINCKE; LÖWE, 2009). In this work, the following metrics were used to evaluate and compare the quality of source code produced using the proposed methodology:

2.6.1 Lines of Code

Historically, Lines of Code (LOC) has been broadly used as one of the most common metrics to measure the quality of a software project. Along with LOC, the Effective Lines of Code (ELOC) metric can be used to improve the analysis. ELOC is the number of lines of code that are not comments or preprocessors. LOC and ELOC can be used in multiple sub-metrics to provide a better understanding of the project's structure. For this analysis, the following sub-metrics were used:

- Lines of Code (LOC): Number of lines of code.
  - Lines of Code per File (LOCFI): Number of lines of code in each file.
  - Lines of Code per Function (LOCFU): Number of lines of code in each function.
  - Total Lines of Code (TLOC): Total number of lines of code in the project.
- Effective Lines of Code (ELOC): Number of lines of code that are not comments or preprocessors.
  - Effective Lines of Code per File (ELOCFI): Number of lines of code in each file.
  - Effective Lines of Code per Function (ELOCFU): Number of lines of code in each function.
  - Total Effective Lines of Code (TELOC): Total number of lines of code in the project.

Well-structured and readable code, typically with fewer LOC, is easier to maintain (LOUBSER, 2021). By being more readable, code becomes easier to understand, modify, and debug, and besides that, less ELOC also means less work to maintain (WILLENBRING; SINGH WALIA, 2021)(FOROUZANI; CHIAM; FOROUZANI, 2016).

Since high numbers of LOC and ELOC can reduce maintainability, projects with larger TLOC are more challenging to maintain due to the increased complexity involved in

understanding and modifying the code (IFTIKHAR *et al.*, 2023)(FOROUZANI; CHIAM; FOROUZANI, 2016). In contrast, smaller, well-defined functions (LOCFU) or modules (LOCFI) are typically easier to maintain and reuse since they can be more easily reused into other parts of the project or even be integrated into other projects (SOMMERVILLE, 2016). Finally, a higher comment density or ratio can indicate the code is well-documented, making it easier for other developers to understand, reuse, and maintain (FOROUZANI; CHIAM; FOROUZANI, 2016).

2.6.2 McCabe's Cyclomatic Complexity

McCabe's Cyclomatic Complexity (MVG) is a software metric used to measure the complexity of a program (FOROUZANI; CHIAM; FOROUZANI, 2016). It measures the number of linearly independent paths through a program's source code. This metric can be used to determine the quality of functions and evaluate the following factors (BENINGO, 2022):

- The expected minimum number of test cases required to test a function
- The risk associated with modifying a function
- The likelihood that a function may contain unknown bugs

Modularity is crucial in reusable designs since it allows software to be divided into smaller, more manageable parts. For that, MVG can help evaluate how modules are structured and how complex they are. High MVG values within a module might indicate that it has too many decision points, which is a characteristic of poor modular design. As a result, modularity gets degraded, making it challenging to adapt or reuse those modules in different contexts. In contrast, a well-modularized design typically exhibits lower MVG values within individual modules, making them easier to understand and modify independently and, therefore, more likely to be reused (FENTON; BIEMAN, 2014)(ROGER; BRUCE, 2019).

MVG can also provide assumptions about the maintainability of the code. As the MVG increases, there is a higher chance of bugs existing in the code. For that, Table 1 presents the relationship between MVG and the risk of bugs.

Higher MVG values can also lead to more difficult debugging and modification processes. For example, Table 2 presents the chance of injecting a bug in a function based on its MVG. Conversely, as the MVG decreases, the maintenance process can be done easier, and software becomes more reliable (FOROUZANI; CHIAM; FOROUZANI, 2016) (IFTIKHAR *et al.*, 2023).

Finally, a high MVG value might also indicate that code is tightly coupled with a specific environment, making it less portable (ROGER; BRUCE, 2019).

Table 1 – McCabe Cyclomatic Complexity value vs. the risk of bugs

Cyclomatic Complexity	**Risk Evaluation**
1-10	A simple function without much risk
11-20	A more complex function with moderate risk
21-50	A complex function of high risk
51 and greater	An untestable function of very high risk

Source: BENINGO (2022)

Table 2 – The risk of bug injection as the Cyclomatic Complexity rises

Cyclomatic Complexity	**Risk of Bug Injection**
1-10	5%
11-20	20%
21-50	40%
51 and greater	60%

Source: BENINGO (2022)

2.6.3 Maximum Nesting Level

The Maximum Nested Level (MNL) is a software metric that measures a function's maximum number of nested blocks. Similar to MVG, MNL can be used as a measure of a function's complexity. The MNL is calculated based on the number of indentation levels in a function.

The higher the MNL, the more complex the function is. Besides that, a high MNL can drastically reduce the readability of the code (JOHNSON *et al.*, 2019). As a result, higher values of MNL can degrade maintainability and demotivate reusability (FOROUZANI; CHIAM; FOROUZANI, 2016).

2.6.4 Simple Maintainability Index

The Simple Maintainability Index (SMI) is a metric that measures a function's maintainability based on the values of MVG and ELOCFU. SMI values range from 1 to 25. The lower the SMI, the easier it is to maintain a function (WILLENBRING; WALIA, 2024)(WILLENBRING; WALIA, 2022).

The SMI is calculated using the following formula:

$$SMI = MVG_{index} \times ELOCFU_{index} \tag{1}$$

Where MVG_{index} and $ELOCFU_{index}$ values are obtained from the Table 3:

Even though the SMI provides how maintainable a function is, it does not provide a clear understanding of the overall project's maintainability. The impacts of worsening a single function's maintainability (increasing the SMI) may increase the difficulty of

Table 3 – Simple Maintainability Index Range Values

Index	Cyclomatic Complexity (MVG)	Effective Lines of Code (ELOCFU)
1	0-7	0-124
2	8-11	125-249
3	12-19	250-499
4	20-49	500-999
5	50 and Greater	1000 and Greater

Source: METRIX++ (2024)

maintaining the entire project. To better analyze this metric, the following equation can calculate the project's SMI Average (ASMI):

$$ASMI = \frac{x \times y}{x + y} \tag{2}$$

Where:

- x is the product of all maintainability indexes. Which represents the overall effectiveness or performance of the system in terms of maintainability.
- y is the sum of all maintainability indexes. Which represents the total potential or capacity for maintainability improvements or efforts within the system.

Since the larger a project gets, the harder it becomes to maintain it, in ASMI, each one of the system's functions adds a portion of difficulty to the overall maintainability of the system. On it, functions with SMI values of 1 have a linear impact on ASMI, while others impact ASMI exponentially. Therefore, it is recommended that developers target as many (and ideally all) project functions to an SMI value of 1.

2.6.5 Module Coupling

The Module Coupling (COUP) metric measures the degree of interdependence between software modules, and it can be used to infer the complexity of the project's architecture. Lower COUP values between modules mean a module is less dependent on other parts of the system, making it easier to reuse in different contexts. Besides that, COUP also relates to maintainability, where lower COUP values indicate easier maintenance (FOROUZANI; CHIAM; FOROUZANI, 2016).

Unlike other programming languages, ANSI-C does not support any features that can natively define what a C language module is. Instead, by not providing a standardized structure, the language allows developers to freely organize projects how they see fit. Consequently, a module definition and structure may change from project to project, making it hard for metric tools to provide a generic approach for them. For that reason, Metrix++ was not used to extract COUP metrics, and instead, they were calculated manually.

3 RELATED WORKS

Several works are available in the literature addressing aspects of the proposed methodology. However, most of them do not provide a complete solution. Many of the contributions of those works were integrated or used as inspiration to address parts of the proposed methodology. In the following sections, each aspect of the proposed methodology will be reviewed individually.

3.1 C Code Standard

One of the main concerns of this methodology is to guarantee that all developers design their code in a similar way, so they can "speak the same language". Code standards are a great way to start this culture within a team since they are well-established rules that can be verified individually, making it easy to ensure that all developers follow the guidelines.

A development code standard can be defined as a set of established guidelines and best practices that guide the software development process. These standards are responsible for ensuring that all stages of development are conducted in a consistent and structured way. By following these standards, developers can work more efficiently, reduce errors, and ensure that the software meets both technical requirements and user expectations (SCHMIDT, 2000). Ultimately, development standards help teams collaborate more effectively and produce higher-quality software that aligns with industry norms and regulations (PARGAONKAR, 2023).

Usually, most C code standards are separated into three system categories:

- **Safety-Critical**: These standards are used to develop safety-critical systems, including medical devices, automotive, and aerospace. These standards are the most restrictive as they ensure code determinism and consistent behavior to prevent failures caused by undefined and unspecified behaviors (KALAYCI, 2023). For years, MISRA (2012) has defined the MISRA-C standard, which is one of the most well-established standards for safety-critical embedded systems.

- **Security-Critical**: These standards are used to develop security-critical systems,

such as cryptographic systems, secure communication, and secure storage. These standards emphasize eliminating security vulnerabilities and preventing cyber-attacks (KALAYCI, 2023). In this regard, the CERT-C standard is one of the most well-known standards (SEACORD, 2016).

- **General**: These standards are used to develop general-purpose systems that do not require safety or security concerns. These standards are less restrictive than the previous ones but still provide a higher code quality and maintainability. In this regard, GANSSLE (2004) defines a structural coding standard while BARR-C (BARR, 2018) defines a development focus standard.

Although critical standards ensure higher levels of code quality, they can be too overwhelming and extensive for general-purpose systems. Besides that, the concepts they present may be too advanced for novice developers. Also, their restrictive level may result in an unnecessary development time overhead. To reduce the complexity of critical standards, HOLZMANN (2006) proposes a simple set of rules for safety-critical development rather than an extensive standard. Next, to evaluate aspects of different standards, ANDERSON (2008) compares popular standards and how they can be combined with tools to ensure higher confidence levels of development.

Besides the system's categories, standards may be designed to address specific attributes and requirements. For example, standards such as MISRA (2012) and SEACORD (2016) are defined to ensure the safety or security of the systems to which they are applied. However, to ensure those requirements, these standards expect their users to be experienced developers with advanced knowledge of the C language and system development. In contrast, standards such as GANSSLE (2004) and HOLZMANN (2006) try to reduce complexity to be more suitable for novice developers.

Standards may also dictate rules to improve some quality attributes of systems. For example, standards such as BARR (2018), MISRA (2012), and SEACORD (2016) restrict the use of certain language features that may depend on architecture-specific characteristics to improve code portability. Besides that, those standards also define rules to reduce complex expressions and how code should structured to improve its readability.

Given the large number of well-established standards available, it becomes unnecessary to create a new one. However, a combination of multiple standards was defined to address the aspects of the proposed methodology. For that, BARR (2018) was used as the base standard, and it was adapted to fit some of the methodology requirements and to include items from the other mentioned standards. Besides that, C99 was used as the base language standard for all the reviewed works.

Table 4 compares the reviewed works and the proposed methodology.

Table 4 – Code Standard Works Comparison

Work	System Category	Readability	Portability	Novice Development
MISRA (2012)	Safety-Critical	Yes	Yes	No
HOLZMANN (2006)	Safety-Critical	No	No	Yes
SEACORD (2016)	Security-Critical	Yes	Yes	No
GANSSLE (2004)	General	No	No	Yes
BARR (2018)	General	Yes	Yes	Yes
This Work	General	Yes	Yes	Yes

3.2 Modularity

Several works in the literature seek to address aspects of firmware design to make it more portable and reusable. In the reuse field, many papers argue that modularity is the basis of many reusable architectures. For example, Dano analyzes the importance of reuse and modularity while suggesting activities that can maximize these factors (DANO, 2019). Next, BENINGO (2017) discusses the importance of modularity in portable firmware and describes the main characteristics and requirements of a modular design for embedded systems.

Intending to make firmware more modular, YUAN *et al.* (2021) introduces a component-based framework for embedded software that confines development to standalone components and thus spares the developer from the need to understand the system in its entirety. Furthermore, BENINGO (2022) applies the concepts presented in his previous work to demonstrate design techniques for developing embedded software. Next, MUDE; JOSHI (2023) evaluates the benefits of modular firmware development for a Genset Controller Unit. Designed for scalability, the results show that his modular approach has produced firmware with a smaller code size, allowing the implementation of other functionalities and reducing the product cost. Finally, HÄNISCH (2023) seeks a different approach by defining a modular architecture based on microservices. With independent and isolated processes, the proposed architecture maximizes flexibility, reduces complexity, and facilitates maintenance.

Modern programming languages use OOP to allow modularity and decoupling. However, the C language does not natively support it. In that regard, NESER; VAN SCHOOR (2009) addresses this question by introducing a framework to simulate OOP features in C language called OOC. The method does not allow a full OOP implementation. However, most core features can be imported similarly. Alternatively, QUANTUM LEAPS LLC (2020) also proposes a similar OOC framework that does not use preprocessors for emulating OOP features.

Multiple aspects of those works were applied in the proposed methodology to design

and enable modularity. The work of YUAN *et al.* (2021) and MUDE; JOSHI (2023) inspired the hierarchical design of the different methodology component types. Additionally, HÄNISCH (2023) provided insights for decoupling component dependencies and delimiting application module scopes. Next, NESER; VAN SCHOOR (2009) provided ideas for expanding the applicability of OOP in C language while the work of QUANTUM LEAPS LLC (2020) was integrated into the methodology to define the class module type. Finally, the rich work of BENINGO (2022) provided multiple insights into development pitfalls, modular design patterns, and other small aspects that were integrated into this work.

Table 5 compares the reviewed works and the proposed methodology.

Table 5 – Modularity Works Comparison

Work	Contribution to the Methodology			
	Object Oriented	Dependency Decoupling	Componentization	Microservices
YUAN *et al.* (2021)	Yes	No	Yes	No
BENINGO (2022)	Yes	No	Yes	No
MUDE; JOSHI (2023)	No	No	Yes	No
HÄNISCH (2023)	No	Yes	No	Yes
NESER; VAN SCHOOR (2009)	Yes	No	No	No
QUANTUM LEAPS LLC (2020)	Yes	No	No	No

3.3 Portability

Concerning portability, several studies have sought to employ methods to facilitate the implementation of portable firmware. For example, MARCONDES *et al.* (2006) adopts an application-oriented and component-based operating system that includes code portability between MCUs with different architectural sizes. In contrast, SUN; LI; MEMON (2017) provides a microservices-based framework for IoT where service and physical layers communicate through a common message broker medium. At the same time, many legacy codes may be required so that they can be ported in the future. In this context, M. GOMES; BAUNACH (2021) analyzes what the current portable IoT operating systems are like and the quality of the currently available ports.

Rehosting techniques allow firmware testing without an MCU or a device application process. When conducting the test, researchers must be able to port the current code to an emulator or a general application computer so that they can work out their routines. Even though the objective is not the same, the result of rehosting methods is very similar to that of portability since the ultimate goal is to port an existing code to a different host.

In this context, ZADDACH *et al.* (2014) recommends Avatar, a hardware-in-the-loop design for an event-based arbitration framework that orchestrates the communication be-

tween an emulator and a targeted physical device. Next, Pretender makes observations of the interactions between the original hardware and the firmware to automatically create models of peripherals that allow the execution of firmware in a fully emulated environment (GUSTAFSON *et al.*, 2019). In the same way FENG; MERA; LU (2020), finds a solution that involves abstracting various peripherals and handling firmware I/O on the fly based on automatically generated models, thus ensuring sufficient code coverage.

After identifying the challenges of rehosting firmware, LEE *et al.* (2023) identifies that existing rehosting techniques have limited applicability. To address that, the author proposes the VDEmu rehosting system that does not require path elimination and supports dynamic direct memory. As another approach, CLEMENTS *et al.* (2020) introduces the HALucinator framework. This high-level emulation method offers simple, analyst-created replacements that carry out the same task from the standpoint of the firmware. Subsequently, the authors expand HALucinator by adding a re-hosting support layer, significantly reducing device porting time for VxWorks (CLEMENTS *et al.*, 2021).

Table 6 – Portability Works Comparison

Work	Contribution to the Methodology			
	Hardware Abstraction Layer	Hardware Decoupled Development	Design Quality Aspects	Development Quality Aspects
MARCONDES *et al.* (2006)	Yes	No	Yes	Yes
SUN; LI; MEMON (2017)	Yes	No	Yes	Yes
M. GOMES; BAUNACH (2021)	No	No	Yes	No
ZADDACH *et al.* (2014)	No	Yes	No	No
GUSTAFSON *et al.* (2019)	No	No	No	Yes
FENG; MERA; LU (2020)	Yes	No	Yes	No
LEE *et al.* (2023)	No	No	No	Yes
CLEMENTS *et al.* (2020)	Yes	Yes	Yes	No
CLEMENTS *et al.* (2021)	Yes	Yes	Yes	No

Some of the techniques described in these previous works were used to inspire the design of a portable development structure for the proposed methodology. First, the proposed abstraction layer designed by MARCONDES *et al.* (2006) was integrated into the methodology structure to create a separation between hardware-specific and system-specific code. Next, the work of ZADDACH *et al.* (2014) inspired the use of an approach that restricts the use of hardware-dependent language features to achieve an architecture-agnostic code. Also, the work of M. GOMES; BAUNACH (2021) provided insights on design aspects that may improve or deteriorate portability. Finally, other works contributed with insights into development practices that may potentially result in less portable systems.

Table 6 compares the reviewed works and the proposed methodology.

3.4 Maintainability

To improve the maintainability of projects developed with this methodology, several studies have sought to identify the best practices and activities that can be employed to improve the quality of the code and reduce the chance of errors being made by novice developers or even experienced developers. To achieve this, MOTOGNA; VESCAN; ŞERBAN (2023) studies the quality attributes prioritized in embedded systems and identifies the best practices and activities they involve. Their research revealed that maintainability, along with safety, security, performance, and energy efficiency, are among the most important quality attributes in embedded systems development. Since maintainability is often cited in the literature as an essential feature of reusable and portable code, several studies offer solutions.

For this reason, SPRAY; SINHA (2018) integrates the knowledge of software architecture with the experience in designing embedded software from the Tru-Test Group to create an abstraction layered architecture and create code bases with improved long-term maintainability. In addition, WILLOCX *et al.* (2018) establish a layered IoT architecture to support the development of complex and maintainable IoT applications. By abstracting low-level implementation details, developers can focus on business logic without being experts in IoT sensor technology.

Furthermore, MAKHSHARI; MESBAH (2021) conducted a study to identify the most common challenges in IoT development and maintenance. The research revealed the most frequent and severe bugs, their correlations, and their root causes, allowing early fault prevention during development or easy detection and correction during the maintenance phase.

Finally, a violation of the design of reusable firmware is a notable cause of portability problems. Thus, the most common mistakes in the development process may result from insights taken to find solutions. HUBALOVSKY; SEDIVY (2010) examines the most common OOP mistakes made by both beginners and experienced programmers. Also, (STEWART, 1999) analyses the most frequent mistakes made in RT software development.

3.5 Similar Works

All the previously mentioned works seek to address specific areas of portable firmware development. However, none of them can reach a complete end-to-end solution. Besides that, other more generic development methodologies available in the literature, industry, and community are mainly designed to facilitate the development process by establishing,

structuring, and providing reusable and portable components that can be used to develop a given system (TREMAROLI, 2023).

In addition to methodologies, a wide range of works in the literature address the development process of embedded systems ((BENINGO, 2022), (HOBBS, 2019), (COOLING, 2019), (BANIK; ZIMMER, 2022), (GARCÍA TUDELA; MARÍN MARÍN, 2023), (WHITE, 2024), and (MARTIN, 2017)). Although they provide valuable insights into the complete development process, they do not provide or define a development methodology that structures and rules a complete system development lifecycle. At the time of writing this work, the author was not aware of any work that provides a complete solution that defines a firmware development methodology tailored to the specific domain and scope addressed herein.

4 THE PROPOSED METHODOLOGY

The proposed methodology is based on three core elements: well-defined team roles, adaptable workflows, and a robust firmware architecture. These elements were designed to boost teamwork, speed up operations, and ensure uniformity in the development process, making the proposed methodology flexible to handle various project demands.

With clear team roles, the proposed methodology aims to reduce confusion and improve efficiency. By assigning specific responsibilities, each team member can easily understand their tasks, enhancing the team's communication and encouraging a cooperative environment. Finally, defining roles and responsibilities allows projects to be executed and managed more smoothly.

The workflows are designed to be flexible and accomodate the needs of different development models such as Spiral, Waterfall, and Agile. This flexibility is important for keeping the methodology relevant and practical, adjusting to project changes and requirements.

The firmware architecture seeks modularity, portability, and maintainability. It is organized hierarchically with layers composed of different modules. Each module is defined by a particular development pattern to address different aspects of the system. This structure makes it easier to navigation and promotes consistent implementation across multiple projects, resulting in modules that are easy to understand and reuse in future projects.

The effectiviness of the proposed methodology is achieved by the sinergy between clear team roles, adaptable workflows, and a solid firmware architecture. Defined roles allows workflows to be executed efficiently, which is crucial for leveraging the benefits of the architecture. Altogether, these elements combine into a cohesive and flexible methodology that meets organizational and technical requirements for projects.

4.1 Team Roles

It is expected that teams be composed of developers of multiple levels of experience. In that sense, one of the main objectives of the proposed methodology is to train junior members while enabling them to collaborate productively on the project. For that, these

members will be assigned to roles that may require frequent interactions with senior members.

The methodology defines five roles that members can be assigned to:

- Architect (ARCH): Responsible for structuring and leading the team throughout the development life-cycle.
- Developer (DEV): Responsible for translating tasks into production code.
- Tester (TEST): Responsible for testing DEV submissions.
- Maintainer (MAIN): Responsible for solving internal and external reports of problems and bugs and requests for changes.
- DevOps (OPS): Responsible for providing solutions to improve the whole project life-cycle.

In this structure, multiple members can be assigned to the same role. Opposite to that, companies may have a limited number of developers to allocate to a given project. Therefore, members can assume multiple roles in the same team.

Architect

The ARCH is responsible for structuring and leading the team throughout the development life-cycle. This person must be the most (or one of the most) experienced member of the team. The ARCH should architect and guide all other members on the development of the ARCH's vision. Being the most required role by other members, the ARCH should never assume any other roles.

Before the project development can be started, the ARCH should gather all project requirements and structure a system that can address them. After that, this system should be translated into the methodology modular structure, and then deliverables should be established for each development phase. Finally, the ARCH should plan and assign tasks to other team members.

During the development process, the ARCH should help other members in understanding and developing their tasks. Basically, this individual should act as a facilitator to help other members overcome any problems they may encounter. The ARCH is also expected to assist the development of other team members' skills by providing on-demand training, pair programming sections, live reviews, and support to any other matter where they can be of help.

The final responsibility of the ARCH is to review and approve every change made to the production code. In this process, the ARCH should evaluate code aspects such as quality, readability, correctness and organization, in addition to ensuring that other

members are correctly following the methodology guidelines, structure, standards, and best practices.

In summary, the ARCH responsibilities are:

- Leading, managing and guiding the development team
- Structuring the project and translating it into tasks
- Supporting, training and empowering other members
- Reviewing the work of other members
- Enforcing methodology guidelines, structure, standards and best practices.

Developer

The main responsibility of DEVs is to translate tasks into production code. The outcome of their work will have a great impact on the quality of the developed system. Therefore, more experienced developers are expected to be assigned to this role.

To ensure that the project advances at the desired pace, DEVs should not receive or address external demand. The reason for that is that frequent multitasking and task priority shifting often lead to context switching as people try to manage different tasks. As a result, members may switch context at inopportune moments when they have maximum context about their current task and are in a state of flow, resulting in high task resumption costs and loss of productivity (KAUR *et al.*, 2020).

Finally, members in this role have extensive knowledge of the parts of the system they are involved in. Consequently, it is expected that these members also provide support and help in the skill development of less experienced team members.

Tester

The TEST role should be the entry point for new and inexperienced developers as these members will be able to study and evaluate higher-quality code created by DEVs. They will be required to fully understand what a given code does to execute their tasks. In this process, they will obtain a better understanding of the current state of the system, develop their programming skills and get used to all methodology aspects.

The main responsibility of these members is to offload DEVs from the overhead of extensive testing of their work. Testing represents almost 19% of the time developers spend dealing with code-related activities (GRAMS, 2019). Therefore, in this structure, DEVs can focus on the progress of the project, while TESTs verify the correctness of their work. However, this workflow does not exempt DEVs from testing their work. Instead, they should always verify that the expected behavior is correct before submitting their work for review.

The inexperience of TESTs should not disregard the fact that testing can be considered a risk-based activity. Testers must understand how to minimize a large number of tests into a manageable test set and make wise decisions about the risks that are important to test or the ones that are not (JAMIL *et al.*, 2017). Therefore, it is of great importance that the ARCH and DEV provide guidance to these members on what and how they should proceed with their activities.

Finally, TESTs are also responsible for release tests. Before each release, TESTs must validate the stability of the product as a whole. During this process, they should evaluate every feature and functionality to ensure that the product behaves as expected.

In summary, TESTs are responsible for:

- Testing DEV submissions
- Writing unit and automation tests
- Function and non-functional tests
- Integration and system tests
- Release tests

Maintainer

Maintenance is usually the longest part of the software life-cycle and may constitute a portion that surpasses more than 80% of overall costs in software development (ALMOGAHED *et al.*, 2023). However, it is usual for small companies to focus heavily on development while lacking a structured maintenance model altogether. When needed, maintenance tasks are solved on an ad hoc basis with the best effort available. These processes work well in a small context with occasional maintenance tasks, however, as the project grows, this model becomes no longer scalable and impacts the team's ability to advance and deliver new features (TYRKKÖ *et al.*, 2019).

To address that, as novice developers improve their skills and become more experienced, they can assume the role of MAIN. MAINs are responsible for receiving and solving internal and external reports of problems and bugs and requests for changes. When assuming this role, members can improve their communication and requirement gathering skills, since they will have frequent interactions with system users, clients, and so on.

Additionally, the MAIN role has the purpose of shielding DEVs from external demands. Maintenance represents almost 28% of the time developers spend dealing with code activities (GRAMS, 2019). Therefore, MAINs can reduce DEVs workload, allowing them to focus on other activities.

DevOps

The OPS is a role that does not directly participate in the development or maintenance workflows of a team. Instead, this role provides solutions to improve the whole project life-cycle.

DevOps can be defined as a software engineering methodology that aims to integrate the work of software development and software operations teams by facilitating a culture of collaboration and shared responsibility (BENINGO, 2022). Its main goal is to shorten feedback loops and the development cycle through collaboration, automation and frequent software releases (LWAKATARE *et al.*, 2016).

In general, embedded software teams have often overlooked DevOps, choosing to use more traditional development approaches due to hardware dependencies (BENINGO, 2022). Yet, the embedded domain can benefit from it to improve a great number of aspects. It is not the goal of this methodology to provide a complete DevOps methodology for embedded systems. Instead, it creates opportunities for OPS members to introduce new solutions that may be beneficial to the project.

4.2 Workflows

One of the aspects of this methodology is that it is not restricted to a given type of development philosophy. Teams can freely choose methodologies, such as Waterfall, Spiral, Agile, or others. Independent of the chosen methodology, ensuring that code standards, architecture structure, and other development conventions are respected is essential.

For that, code review is essential in ensuring code quality and reducing the likelihood of errors and bugs (BEN SGHAIER; SAHRAOUI, 2024). In addition, they can also promote teaching, bonding, and improving understanding within teams (ELDH, 2024). Besides ensuring the correctness of conventions, software testing is crucial for ensuring the quality and reliability of software products (WANG *et al.*, 2024).

Applying code reviews and testing processes is a fundamental factor in the pursuit of successful software products (HOMÈS, 2024). For those reasons, supported by the roles defined in Section 4.1, the proposed methodology defines two main workflows to introduce new code into production:

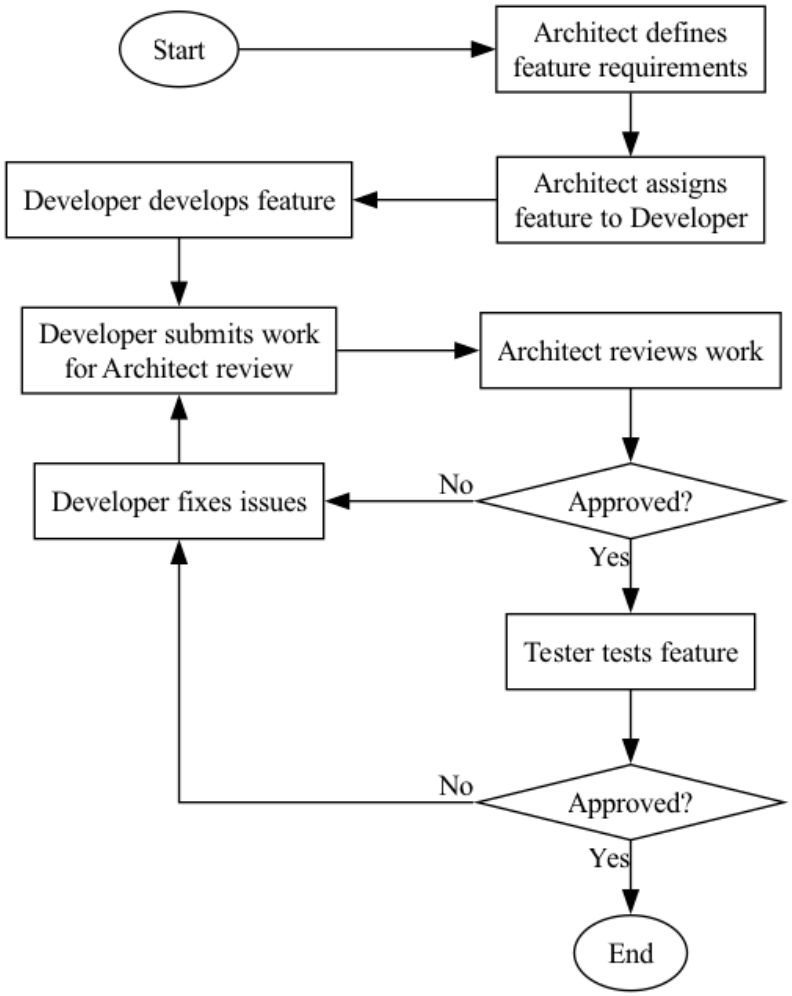
Development Workflow

The development workflow (Figure 3) consists of the processes in which new features and functionalities must be performed before getting into production.

Initially, the ARCH will define and describe the scope and requirements of a given new feature. Next, this professional will assign a DEV to implement it. After completion, the work developed by the DEV must be submitted to the ARCH for review. On approval, the ARCH will assign a TEST to evaluate this new feature and, if approved, move it into

production. In case of disapproval, the DEV should be reassigned and follow the same steps again.

Figure 3 – Development Workflow Flowchart



Source: The Author

Maintenance Workflow

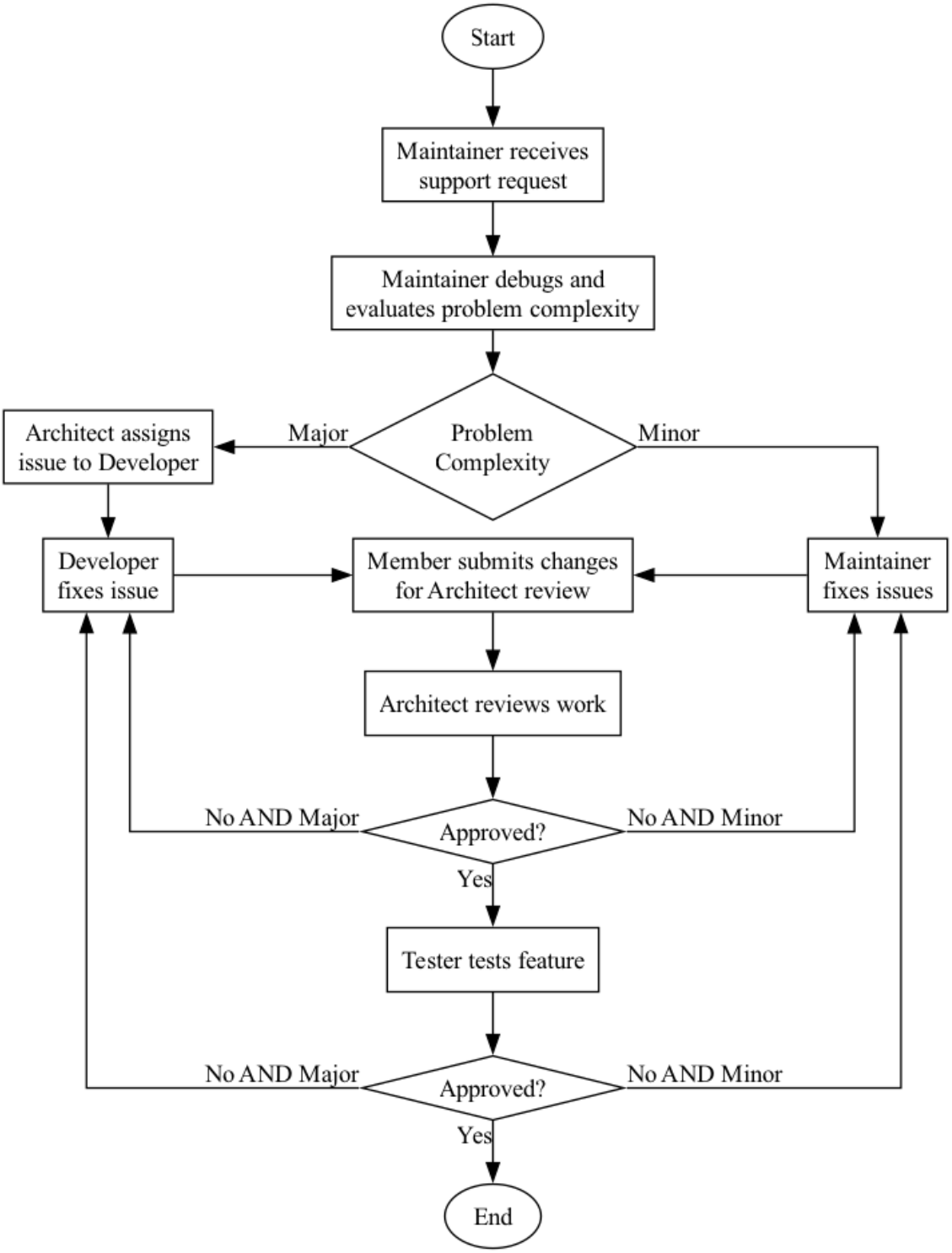
The maintenance workflow (Figure 4) consists of the processes where changes, fixes and corrections need to go through before getting into production.

After receiving a new report or request, the MAIN should evaluate and/or debug it to classify it as a minor or major activity. In this context, minor activities are the ones that do not require extensive changes and, therefore, the MAIN has a clear vision of how to address them. Opposite to that, major activities represent more complex problems or changes that may impact large portions of the code.

Minor tasks should be dealt with by the MAIN who evaluated them, while major ones should be assigned to the ARCH. According to priority and criticality, the ARCH should assign a DEV to this task. After implementation is completed, either minor or major tasks should be submitted to the ARCH for review. On approval, the ARCH should assign a

TEST to evaluate it and, if approved, move it into production. In case of disapproval, the task should be reassigned to the member responsible for its implementation and follow the same steps again.

Figure 4 – Maintenance Workflow Flowchart

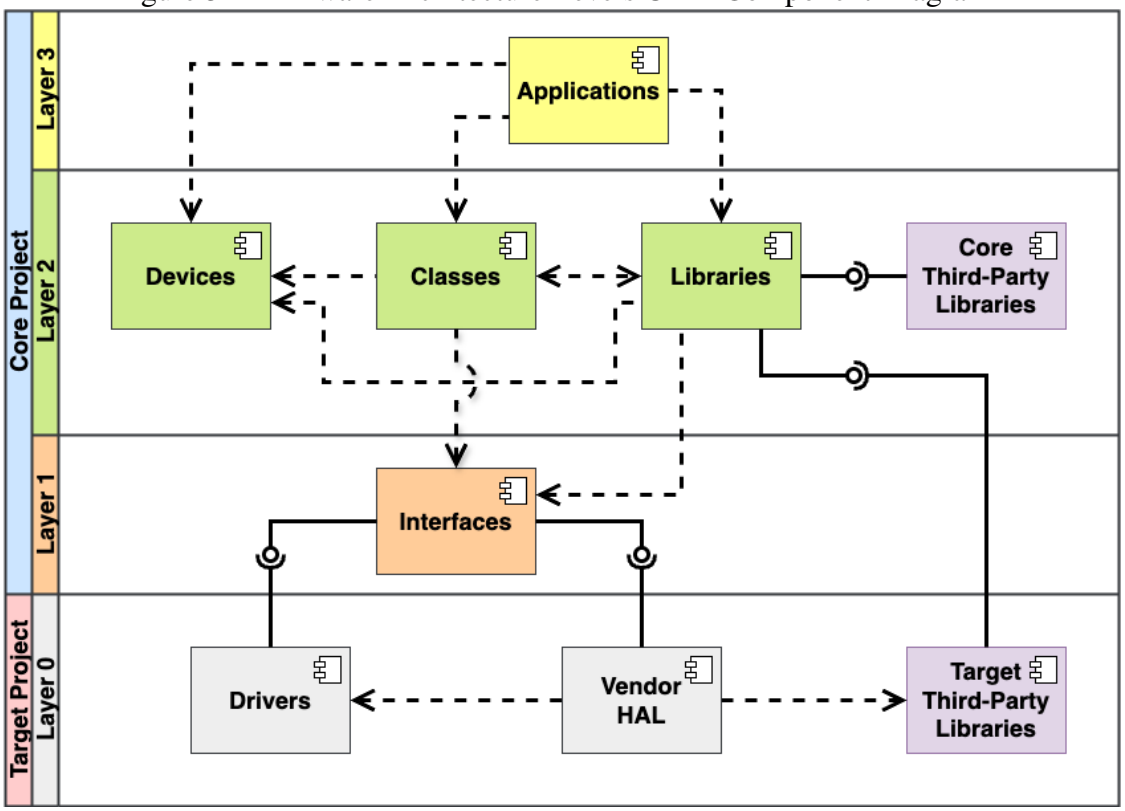


Source: The Author

4.3 Firmware Architecture

One of the goals of this methodology is to allow a complete decoupling between hardware and system implementations. For that, projects are separated into two different repositories. The first repository should contain all system implementations in a generic form and will be called CORE Project (blue region of Figure 5). Next, a second repository should contain all hardware-specific implementations for a given target MCU and will be called TARGET Project (red region of Figure 5). Finally, each TARGET should include CORE as a submodule.

Figure 5 – Firmware Architecture Levels UML Component Diagram



Source: The Author

Besides portability, the methodology provides organizational procedures to improve reuse and how firmware code is maintained. To do so, it enforces the standardization of the development process by providing a set of module types. Each module type has specific development patterns that compartmentalize a part of a given system layer.

Another methodology concern is that novice developers are often included in projects with little or no supervision from other experienced developers. As a result, these developers may introduce problems to the system or violate the design choices made by the system architect. To address this issue, the methodology provides definitions called Novice Firmware Choices (NFC) aimed at guiding these developers and preventing them

from making their most common mistakes.

4.3.1 Target Project

Embedded systems projects are frequently implemented according to how a given MCU and its peripherals were designed to work. The reason is that many embedded applications are conceived at the edge of available MCU resources. Since the cooperative development of the hardware and software components can be used to achieve the best performance of an embedded system (ZHENG; LIANG; XIONG, 2021), this strategy becomes the most efficient.

However, hardware-specific implementation designs may introduce many obstacles regarding the portability and reuse of the code of these systems. In addition, many modern embedded projects do not exist in those limits, allowing their implementations to spend more resources in favor of other benefits. Opposite to hardware-specific implementations, this methodology proposes a system-specific design allowing system-level code to be retargeted by simply replacing TARGET repositories.

Towards that end, TARGET provides a space where all MCU-specific implementations can be freely developed without directly interacting with any system code (Layer 0 in Figure 5). In this context, the methodology does not establish module types or development rules for development done in this layer. Besides being subjected to the rules of interaction between TARGET and CORE (Section 4.6), developers are free to structure and design code as they please.

The flexibility provided in this region allows teams to leverage a wide range of resources and tools provided by vendors and the open-source community. For example, for each different target MCU, developers may choose to use or mix code generation tools, HAL frameworks, driver examples, or even design their own register-level implementations from scratch. Even though the methodology defines no module types, Figure 5 exemplifies one of these scenarios with the help of two different module types ($Drivers$ and $Vendor\ HAL$) that were used to implement hardware-specific according to the requirements provided by CORE.

As a result of not being required to follow the same structure, porting times can be expressively reduced and simplified for different MCUs. This is because no consideration needs to be given to other TARGET projects when developing a new one. Besides that, in many cases, the developers may only be required to develop middleware implementations to fit CORE's functional requirements since most of the MCU-specific code is already provided or generated by a vendor tool. Additionally, this flexible structure does not prevent developers from designing more advanced and optimized implementations.

Portability Domain

Even though conceptually possible, designing a system to be universally portable may result in implementations with excessive or unnecessary complexity or even become impracticable. Also, systems may have intrinsic requirements limiting them to be targeted only to a given domain. For example, a system may require a minimum of 128KB of RAM, a given number of UART peripherals, or a maximum time to produce results of a given algorithm. Besides that, limiting the portability domain may be a simple strategic choice. For example, a company could decide only to use ARM microcontrollers for their products.

Regardless of the reasons, the ARCH is expected to provide a portability specification (SPECS) containing all domain considerations foreseen by the system architecture. In this process, the ARCH may attempt to reduce the development complexity by either restricting the domain too much or defining a broad portable domain, which may result in a substantial increase in development complexity. Therefore, one must balance those two edges and provide a reasonable domain for the reality of the product under development.

Since all TARGET projects will share it, the SPECS documentation should be maintained inside the CORE repository. Examples of common domain specifications are:

- MCU Specifications
  - Bit depth (8-bits, 32-bits, etc.)
  - Architecture (ARM, RISC-V, AVR, PIC, etc.)
  - Endianness (little-endian or big-endian)
  - Minimal memory size
  - Minimal operating frequency
  - Required peripherals
- Bare-Metal or RTOS
- Third-Party library support
- Compiler
- Peripheral configurations
- Driver design features (blocking, IRQ, DMA, etc.)

4.3.2 Third-Party

Third-party libraries are those that are not maintained or developed in-house. Open-source or purchased libraries are examples of this type of library. Since one of the main

objectives of this methodology is code reuse, it is highly recommended that developers make use of good-quality libraries instead of starting everything from scratch.

The main disadvantage of third-party libraries is that they may be limited to a particular domain. For example, an LCD library may apply to only a specific list of display devices. However, for several reasons, replacing the display device with a different unsupported model may be necessary. Having third-party-specific calls inside the product code would cause problems since a revised implementation would be required. For this reason, an Interface Library should abstract every third-party library.

Third-party libraries are often found to be provided with vendor code-generator tools and placed inside TARGET. At the same time, developers may also use third-party libraries obtained elsewhere for system-level functionalities, which should be placed inside CORE. For these reasons, they are an exception layer that can exist in both TARGET and CORE.

4.3.3 Core Project

The CORE project can best be described as the repository containing the complete hardware-decoupled system implementation. Modules are organized into one of three hierarchical layers (Layers 1, 2, and 3, of Figure 5). Each layer comprises a sum of modules describing small parts of the system.

Besides being portable to another MCU, the separation between hardware and system allows developers to run CORE on a host computer or continuous development applications without any hardware. As a result, the development time gets reduced since time-consuming processes, such as programming the MCU, can be avoided.

Firmware developers can also use tools that were previously exclusive to software developers. For example, even though unit testing has proven its effectiveness and advantages in software development, embedded developers tend to avoid it. This is because using microcontrollers, peripherals, sensors, and similar devices may create a hostile environment for unit testing (TOTH; KARLSSON, 2021). Since CORE is hardware-agnostic, developers no longer face that problem, and the whole system can undergo unit and automated testing without hardware emulation.

4.4 Conventions

Another common mistake many firmware projects make is the failure to include developer conventions. However, these play a crucial role in a well-organized environment and make it easy to understand the developed system. For this reason, the methodology establishes a few conventions. It is also advisable for the system architect to define his conventions.

4.4.1 Coding Standard

Coding standards and guidelines are meant to define the rules of how it should be structured and which language features should and should not be used (HOLZMANN, 2006). By following guidelines, developers may be restricted to a subset of the language, thus removing or reducing the opportunity to make mistakes (MISRA, 2012). Besides that, coding standards can also increase the readability and portability of source code (BARR, 2018).

ARCHs are free to create their own coding standards or make use of existing ones. However, this methodology recommends the use of the provided development standard (Appendix A). This standard was designed by combining the work of multiple authors ((BARR, 2018), (MISRA, 2012), (HOLZMANN, 2006) and (QUANTUM LEAPS LLC, 2020)).

Instead of providing a simple standard, the proposed methodology could have defined that a standard already established in the industry should be used. However, one of the main aspects that the methodology seeks to deliver is allowing novice developers to contribute as much as possible to their projects.

The problem with more advanced standards is that they might be overly complicated and discouraging for inexperienced developers. For example, the MISRA C standard introduces safety-critical rules that require a more advanced understanding of the C programming language (MISRA, 2012).

As an example, the MISRA C standard defines rules and restrictions for complex pointer arithmetic. Although those rules prevent critical errors, discussing those concepts may overwhelm beginners still trying to learn basic pointer usage. Besides that, for novice developers, achieving clear and understandable code is more important than defining rules for concepts that these developers might rarely encounter. For those reasons, the provided methodology standard tries to find a balance between a standard that novice developers can easily follow and, at the same time, provides a high level of safety and portability and addresses most of the methodology's characteristics.

4.4.2 Prefixes

Architects are free to choose module naming prefixes as they see fit. However, the methodology recommends a three-letter pattern for CORE modules. The prefixes should be used in the conventioned module files and every public function name. Moreover, the methodology advises that other code elements should use them ($typedef$s and macros, for example), although this is not an essential requirement. Using prefixes allows developers to locate and understand the dependencies of the code they are working on, while also ensuring that the project is better organized.

4.4.3 Standard Files

The methodology stipulates that the modules can only be imported by their respective public headers. This allows developers to implement their modules safely without an extensive documentation overhead of what should or should not be used outside a module. Additionally, a set of standard files (Table 7) is defined to ensure an improved module organization. Also, this was carried out as a NFC to allow developers to get used to the OOP access modifier. Finally, the public type header was added as another NFC as a simple solution to major recursive inclusion problems. These $typedef$s can be separated from prototype functions, allowing the modules to share types without experiencing conflicts.

Table 7 – Methodology Standard Files

File	Filename
source	prefix_name.c
public header	prefix_name.h
public types	prefix_name_types.h
private header	prefix_name_private.h
internal header	prefix_name_internal.h
override header	prefix_name_override.h
override source	prefix_name_override.c
readme document	readme.md

4.4.4 Documentation

Every module must contain a documentation file. Since most version control systems support this, the methodology recommends using $readme.md$ files written in markdown. Developers should use these files to provide essential information regarding module specifications, design choices, how-to-use tutorials, examples, reference links and files, state-flow diagrams, and any other key areas. Architects should also define an in-code documentation standard. Even though this is not a requirement, the methodology recommends using the Doxygen format since it is well-known by the embedded systems community.

4.4.5 Domains

Each project layer may contain one or more types of modules with restricted access to other modules. Moreover, the convention's access permissions should be respected to preserve the correct level of abstraction. Module types are designed to address different requirements. For that reason, developers should respect the design patterns established by the methodology. Finally, modules may be used for other implementation domains, meaning that multi-threading must always be taken into account if dependencies are not thread-safe.

4.5 Robot Example

To better guide the reader into a clear understanding of the different types of modules in CORE, a system draft was designed (ROBOT). To contextualize this example, an imaginary robotics company has successfully implemented the proposed methodology in its development processes. Throughout the years, many products have been developed according to it, and now, this company wants to develop a new arm robot for its portfolio of products. This robot is composed of a three-joint arm and a claw.

For this robot, four identical I2C servo motors were used to control each individual joint and claw. This servo motor model has been used for many other products, and therefore, the company wants to leverage the reusable aspects of the methodology to speed up the development process by reusing previously developed modules. Besides that, it is a requirement that this robot supports a set of common Application modules, therefore requiring it to interact with them the same way other robots do.

After evaluating the requirements of this robot and the available reusable modules, the development team defined the firmware architecture presented in Figure 6. In the diagram, pre-existing reused modules have their names in red. In the following sections, the details of each module presented in this example will be discussed.

4.6 Layer 1 Interfaces

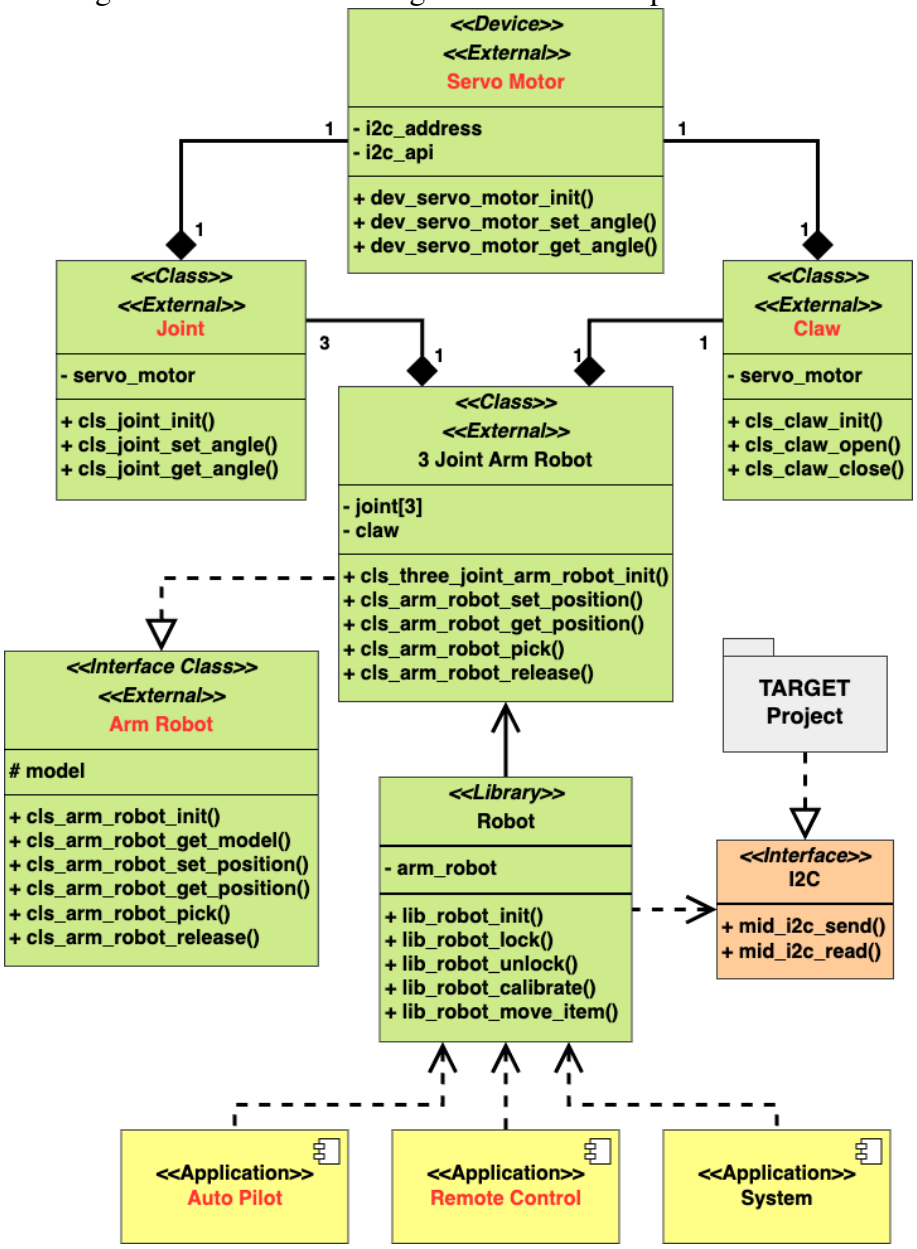
Layer 1 can be defined as the CORE's project hardware abstraction layer. For this layer, the methodology defines a single module type called Interfaces, which should be the only link between CORE and TARGET. It is important to notice that Layer 1, along with Interface modules, is the only HAL that should be used inside of the CORE project. Although HAL modules (not defined by this methodology) may exist in Layer 0, they should only be used inside of it and should never be accessible or used inside the CORE project.

The role of Interfaces is to provide the necessary function prototypes that may be required (DOUGLASS, 2010) by CORE to access all peripheral devices and MCU-specific functionalities and information. Another NFC is that Interface functions must be the only interaction between the two firmware regions, and this border should never be crossed. This ensures that the functions can be independent and that CORE modules do not require hardware-specific knowledge.

In addition to accessing peripherals and functionalities, all hardware-specific information should be provided to CORE by middleware functions. For example, an application jumping address or non-volatile data may be stored in different memory regions for different MCUs. Thus, this information should be abstracted to CORE to make it independent. Finally, Interfaces should only be displayed in the form of header files.

Since Interfaces contain no implementations, these modules should provide a detailed

Figure 6 – UML Class Diagram for the Example of Robot Modules



Source: The Author

implementation and requirements documentation. Every function detail should be described to ensure that new TARGET implementations are consistent with the system architecture.

In many situations, iInterface functions are designed as a one-to-one abstracted representation of the hardware used to develop the initial TARGET. For example, a product may be developed for a given MCU (MCU_A) that contains a UART peripheral that receives serial data and stores it in a 32-byte FIFO buffer. Whenever a new byte is received with a full buffer, the hardware removes the first queued element to make space for the new byte. When creating the Interface module (Listing 1) for this UART, the developers decided that no changes to how MCU_A works were needed. Therefore, the requirements

for this module were set identically to how MCU_A works (Listing 2).

Listing 1 – Example CORE Interface Module for a UART

```
/**
 * @brief Receive next RX byte from UART buffer
 *
 * @details UART Requirements:
 * - Minium buffer size: 32 bytes
 * - Should always return oldest received byte available
 * - Receiving new byte with full buffer should drop oldest byte
 *
 * @param byte[out] Byte to store received byte
 *
 * @return true Byte received
 * @return false Empty buffer
 */
bool mid_uart_receive_byte(uint8_t *byte);
```

Listing 2 – Example of TARGET implementation for MCU A

```
#include "mcu_a.h" // MCU A vendor HAL
#include "mid_uart.h"

bool mid_uart_receive_byte(uint8_t *byte) {
    return mcu_a_vendor_hal_uart3_receive_byte(byte);
}
```

After a while, the product may need to be retargeted to a different MCU (MCU_B) that contains a bufferless UART peripheral. On it, each serial byte is received and stored in a register that can be read inside of an ISR. In this case, the serial reception function was implemented following the design of MCU_A. However, for MCU_B, the developer should provide an implementation that mimics the MCU_A behavior (Listing 3). In this process, the developer should not be required to check the MCU_A implementation code or to review its datasheet. Instead, the developer should only need to read the Interface function documentation to understand the requirements and the expected behavior of this function.

Listing 3 – Example of TARGET implementation for MCU B

```
#include "mcu_b.h" // MCU B vendor library
#include "mid_uart.h"

#define BUFFER_SIZE 32

uint8_t buffer[BUFFER_SIZE] = {0};
uint32_t head = 0;
uint32_t tail = 0;

void UART0_RX_IRQHandler(void) {

    if (UART->STATUS & UART_STATUS_RX_BYTE) {

        uint32_t next_head = ((head + 1) % BUFFER_SIZE);
        if (next_head == tail) {
            tail = (tail + 1) % BUFFER_SIZE; // full, drop oldest byte
```

```
        }
        buffer[head] = UART->RXDATA;
        head = next_head;
    }
}

bool mid_uart_receive_byte(uint8_t *byte) {

    bool has_byte = false;

    if (head != tail) {
        *byte = buffer[tail];
        tail = ((tail + 1) % BUFFER_SIZE);
        has_byte = true;
    }

    return has_byte;
}
```

Finally, the Interface module domain is summarized in Table 8:

Table 8 – Interface Domain

Implementation	Bare-Metal
Access Layers	None
Design	Prototyping

Analysis in the Robot Example

Similarly to the UART example, ROBOT (Figure 6) needs to provide an abstraction for TARGET's I2C peripheral so that other CORE modules can later use it. For such purpose, the I2C Interface module was designed, providing function prototypes along with their respective documentation (Listing 4). Finally, TARGET developers can implement those functions according to the specifications provided in this module.

Listing 4 – Robot Example: I2C Interface Module Implementation Fragments

```
/**
 * @brief Send I2C data
 *
 * @details Should block until the data is sent
 *
 * @param address[in] Device bus address
 * @param data[in] Data buffer
 * @param size[in] Data size (Bytes)
 *
 * @return true Success
 * @return false Fail
 */
bool mid_i2c_send(uint8_t address, uint8_t *data, uint32_t size);

/**
 * @brief Read I2C data
 *
 * @details Should block until the data is read
```

```
 *
 * @param address[in] Device bus address
 * @param data[out] Data buffer
 * @param size[in] Data size (Bytes)
 *
 * @return true Success
 * @return false Fail
 */
bool mid_i2c_read(uint8_t address, uint8_t *data, uint32_t size);
```

4.7 Layer 2 Domains

Layer 2 is designed to allow code modularization and reuse. The primary purpose of this layer is to separate codes and break down their implementation into small, easily understood units. For that, modules from this layer can be divided into two domains:

4.7.1 Project Domain

Modules of this domain are designed for the firmware under development and, currently, only exist in it. They do not have to be reusable and may depend on other modules. This flexibility is intended to reduce the developer's overhead for features that should only exist in a single project. At the same time, they should still be designed as modular parts of the system. As a result, they create an abstraction level between generic and system-specific behavior patterns. Finally, Project modules are the only ones that can directly import Interface modules from Layer 1.

4.7.2 External Domain

Modules of this domain are designed to be reused by multiple projects. They can be seen as generic modules that can be imported into projects. They must be self-contained and decoupled from other project-specific modules. In addition, they must be versioned independently, and only released versions should be used by CORE. This decoupling system enables external modules to be tested, debugged, and reused more simply since implementations are not interdependent or contain lots of unrelated content.

This methodology recommends developers to follow the KISS ("keep it simple, stupid") principle (MILICCHIO, 2007), meaning that developers should restrict themselves to their current task scope. This is an important NFC since novice developers tend to over-complicate their code by trying to predict future requirements and address every possible scenario that may never be required. This is a common mistake that can lead to a significant increase in development time and complexity.

Developers also tend to neglect inconvenient tasks whenever possible (BODEN; NETT; WULF, 2010). Developers may avoid adding new features or changes to external modules to avoid the work overhead of recompiling them. Therefore, unless strictly necessary, external modules should not be provided as compiled libraries. Having direct access to the

source code of the external modules encourages developers to follow the KISS principle since they can easily add future requirements later.

Analysis in the Robot Example

In Figure 6, it is possible to observe that ROBOT contains five external domain modules (Servo Motor, Joint, Claw, Arm Robot, and 3 Joint Arm Robot) and a single project module (Robot). It is interesting to notice that, although the 3 Joint Arm Robot module was developed specifically for this project, it was designed as an external module, enabling it to be reused in future projects that might require the same type of arm robot.

4.8 Layer 2 Module Types

Besides domains, Layer 2 modules are separated into three module types (Devices, Classes, and Libraries), each of which is responsible for a different modularization pattern and is thus designed to meet different requirements. As a result, it becomes possible for developers to leverage from beneficial aspects of both procedural and object-oriented development.

4.8.1 Devices

Devices are modules representing IC peripherals that are external to the target MCU. These modules must be separate from the project (always be external domain) so that a reusable IC devices library can be formed. The Device implementation process is also expected to be bare-metal and allow multiple instances to provide a wider portability domain. Finally, it is important to mention that, in Figure 5, it is possible to notice that, different from other Layer 2 modules, Device modules must not depend on or include any other modules.

Every Device should provide a type of handler variable containing all the information regarding a single instance. This handler is always expected to be passed on as the first function argument to access Device instance functionalities, thus dispensing with the need for instance-specific implementations that may result in undesirable repetitiveness. To illustrate, Listing 5 exemplifies both instance-specific and instance-handler implementations.

Listing 5 – Example of Instance-Specific versus Instance Handler Implementation

```
// Instance-Specific Implementations example:

void led_0_set(bool on) {
    // Implementation for LED 0
}
void led_1_set(bool on) {
    // Implementation for LED 1
}
```

```c
// Instance-Handler Implementation example:

typedef struct {
    uint32_t led_id;
} led_t; // Instance handler Definition

void led_set(led_t *led, bool on) {
    // Generic LED implementation
}
```

Even though this structure may seem similar to the one described in classes (Section 4.8.2), devices do not follow OOP principles. Modules of this type do not provide features such as inheritability, interfaceability, or polymorphism. The reason behind this approach is to extend the use of these implementations to other projects that do not follow this methodology and to facilitate the adaptation of third-party implementation of these devices to the methodology structure.

The Device module domain is presented in Table 9:

Table 9 – Device Domain

Implementation	Bare-Metal
Access Layers	None
Design	Instance Handlers

Analysis in the Robot Example

In the ROBOT example, since the servo motor was already used in a different project, the Device module was already available and, thus, did not require any new development. Consequently, developers do not need to spend time understanding specifics about implementing that particular peripheral (as long as no functionality expansion is required). To better describe this module structure, Listing 6 presents fragments of its implementation.

Listing 6 – Robot Example: Servo Motor Device Module Implementation Fragments

```c
/**
 * @brief Device Instance Handler
 */
typedef struct {
    uint8_t i2c_address; /** I2C Device address */

    /**
     * @brief I2C Send
     * @details Expects blocking behavior
     *
     * @param address[in] Device address
     * @param data[in] Data Buffer
     * @param size[in] Data buffer size (Bytes)
     *
     * @return true Success
     * @return false Communication Failed
     */
```

```
    bool (*i2c_send)(uint8_t address, uint8_t *data, uint32_t size);

    /**
     * @brief I2C Receive
     * @details Expects blocking behavior
     *
     * @param address[in] Device address
     * @param data[out] Data Buffer
     * @param size[in] Data size (Bytes)
     *
     * @return true Success
     * @return false Communication Failed
     */
    bool (*i2c_read)(uint8_t address, uint8_t *data, uint32_t size);
} servo_motor_t;

// ...

/**
 * @brief Initialize Servo Motor
 *
 * @param motor[in/out] Instance Handler
 */
void dev_servo_motor_init(servo_motor_t *motor) {
    // Servo Motor Device Initialization Process
}

/**
 * @brief Set Servo Motor Angle
 *
 * @param motor[in] Instance Handler
 * @param angle[in] Angle (Radians * 100)
 *
 * @return true Success
 * @return false I2C Communication Failed
 */
bool dev_servo_motor_set_angle(servo_motor_t *motor, uint32_t angle) {
    uint16_t data = angle_to_register(angle);
    return motor->i2c_send(motor->i2c_address, (uint8_t *)&data, sizeof(data));
}
```

To be able to use this module, developers only need to provide the required API functions and variable values through the defined instance handler and initialize the module. This process will be demonstrated in Section 4.8.2.

4.8.2 Classes

One advantage of modern programming languages is their built-in support for OOP. Unfortunately, C is a procedural language that does not natively support OOP. Since the proposed methodology does not define any restriction about the use of programming languages and thanks to the large degree of source compatibility between C and C++, Architects may choose to combine those languages to support OOP (POSCH, 2019).

In parallel to that, for several reasons, Architects may also choose to restrict their projects to a single programming language. For that reason, projects that are restricted

to C language require a proper definition and standardization of how Class-type modules should be structured and designed. In that direction, it is possible to replicate some of the core features of OOP in C by following design guidelines, often called Object-Oriented C, in the literature.

For projects that require the development of OOC Class modules, the proposed methodology used the work of QUANTUM LEAPS LLC (2020) as the base to define a structure for Class modules that provides the following OOP features:

- Encapsulation
- Inheritance
- Interface
- Polymorphism

Classes should always provide the initialization (input argument) and instance (output argument) structures used by the class constructor method. If another class is inherited, these structures must contain the respective parent structures, called $super$, as their initial elements. The inherited class structures must be the first element that determines subtyping behavior in the child. Finally, only single inheritance is allowed.

Project Classes can be bare-metal or RTOS, while external Classes must always be bare-metal. If external Classes require RTOS functionalities, they should be provided in an abstract format as arguments of the initialization structure.

Since C language struct elements do not have access modifiers, as an NFC, Class handlers should only contain protected elements. This means that only the class and child classes can directly access variables from the instance handler. Any other external access should always be done through getter and setter methods. Besides making the reviewing process more straightforward, this also avoids the violation of OOC principles and, if necessary, facilitates the inclusion of thread safety in the module.

The Class module also defines a module subtype called Interface Classes. In OOC, the lines between inheritance, interface, and abstraction are blurred because the language does not explicitly enforce those concepts. In terms of implementation, mimicking the OOP interface and abstract classes in OOC has such a slight difference that providing individual descriptions may be redundant. For that reason, the methodology assumes that when developed in OOC, the Interface Class module type can be used for both concepts.

Finally, the Class module domain is presented in Table 10:

Analysis in the Robot Example

In ROBOT, the reuse of multiple previously developed modules was the key factor for the fast development of the project. By observing Figure 6, it is possible to notice that

Table 10 – Class Domain

Implementation	Bare-Metal/RTOS
Access Layers	1 and 2
Design	Object Oriented C

most of the parts of this system were already available, and developers were only required to develop a single Class module to provide the specific characteristics of the robot under development.

In Listing 7, it is possible to observe the structure of the Joint Class. As described in Figure 6, Joint is an External Domain Class Module that does not depend on project-specific modules.

The first aspect to observe in its implementation is that both $joint\_init\_t$ and $joint\_t$ structs are identical. Even though this might not be the case for every Class, it may seem redundant in this case. However, each one of them has a different purpose. While $joint\_t$ purpose is to maintain all the data regarding a single Class instance throughout the execution of the system, $joint\_init\_t$ only exists in the initialization of the instance. After the initialization is executed, there are no guarantees that the data stored on it will still be available or accessible, and, therefore, $joint\_init\_t$ should only be used for initialization purposes.

Another item worth discussing is that, in this simplified scenario, the existence of different motor Devices was not considered. Since one of the Joint Class responsibilities is to abstract specifics about the servo motor, it could be expanded to support multiple different motor Device modules. In this situation, Joint would be required to provide means to select the desirable Device module and standardize all available Devices into a single behavior.

Listing 7 – Robot Example: Joint Class Module Implementation Fragments

```
/**
 * @brief Class Initialization Parameters
 */
typedef struct {
    servo_motor_t servo_motor; /** Servo Motor Initialization */
} joint_init_t;

/**
 * @brief Class Instance Handler
 */
typedef struct {
    servo_motor_t servo_motor; /** Servo Motor */
} joint_t;

// ...

/**
 * @brief Create Class Object
 *
```

```
 * @param handler[out] Instance Handler
 * @param init[in] Initialization Parameters
 */
void cls_joint_init(joint_t *const handler, joint_init_t *const init) {
    handler->servo_motor = init->servo_motor;
    dev_servo_motor_init(&handler->servo_motor);
}

/**
 * @brief Set Joint Angle
 *
 * @param handler[in] Instance Handler
 * @param angle[in] Angle (Radians * 100)
 *
 * @return true Success
 * @return false Communication Failed
 */
bool cls_joint_set_angle(joint_t *const handler, uint32_t *angle) {
    return dev_servo_motor_set_angle(&handler->servo_motor, angle);
}
```

To evaluate the next Class module, Listing 8 presents the implementation of the Claw Class module. Reviewing this module makes it possible to notice a high similarity between Joint and Claw Classes. Because of that, the considerations done by Joint can be replicated in the Claw module. Besides that, although similar, Joint and Claw modules can clearly highlight the OOP encapsulate concept. While the Joint module encapsulates the behavior of positioning the angle of a single robot arm joint, Claw encapsulates control of opening and closing a robotic claw.

Listing 8 – Robot Example: Claw Class Module Implementation Fragments

```
#define OPEN_CLAW_ANGLE  200 /** Open Claw Angle (Radians * 100) */
#define CLOSE_CLAW_ANGLE 400 /** Close Claw Angle (Radians * 100) */

/**
 * @brief Class Initialization Parameters
 */
typedef struct {
    servo_motor_t servo_motor; /** Servo Motor Initialization */
} claw_init_t;

/**
 * @brief Class Instance Handler
 */
typedef struct {
    servo_motor_t servo_motor; /** Servo Motor */
} claw_t;

// ...

/**
 * @brief Create Class Object
 *
 * @param handler[out] Instance Handler
 * @param init[in] Initialization Parameters
 */
void cls_claw_init(claw_t *const handler, claw_init_t *const init) {
```

```
    handler->servo_motor = init->servo_motor;
    dev_servo_motor_init(&handler->servo_motor);
}

/**
 * @brief Set Claw Open
 *
 * @param handler[in] Instance Handler
 *
 * @return true Success
 * @return false Communication Failed
 */
bool cls_claw_open(claw_t *const handler) {
    return dev_servo_motor_set_angle(&handler->servo_motor, OPEN_CLAW_ANGLE);
}
```

In the introduction of ROBOT (Section 4.5), it was discussed that this new robot should be able to interact in the same way other robots do to be integrated with the common Application modules. For that, all the company's arm robots use the same Arm Robot Interface Class (ARIC) module. This module is responsible for defining the behaviors to be implemented by other robot Class modules, along with providing generic implementations. Because of those characteristics, the ARIC module can be better described as an implementation of an OOP abstract class.

In Listing 9, the implementation of the ARIC module is presented and new aspects of OOC can observed. The Overridable structure defined in this module can be used for overriding methods and implementing virtual ones. This design makes implementations and overrides persistent when upcasting the object. Also, to ensure the runtime safety of the code, placeholder functions should always be assigned to virtual or unused methods.

Listing 9 – Robot Example: Interface Arm Robot Class Module Implementation Fragments

```
/**
 * @brief Class Initialization Parameters
 */
typedef struct {
    char *model; /** Robot Model */
} arm_robot_init_t;

/**
 * @brief Class Instance Handler
 */
typedef struct {
    char *model;                        /** Robot Model */
    const struct arm_robot_api *api; /** Overridable API */
} arm_robot_t;

/**
 * @brief Class Overridable API
 */
struct arm_robot_api {
    bool (*set_position)(arm_robot_t *const handler, xyz_t *position); /** @ref
    cls_arm_robot_set_position */
```

```
    bool (*get_position)(arm_robot_t *const handler, xyz_t *position); /** @ref
    cls_arm_robot_get_position */
    bool (*pick)(arm_robot_t *const handler);                          /** @ref
    cls_arm_robot_pick */
    bool (*release)(arm_robot_t *const handler);                       /** @ref
    cls_arm_robot_release */
};

/**
 * @brief Create Class Object
 *
 * @param handler[out] Instance Handler
 * @param init[in] Initialization Parameters
 */
void cls_arm_robot_init(arm_robot_t *const handler, arm_robot_init_t *const init);

/**
 * @brief Get Robot Model
 *
 * @param handler[in] Instance Handler
 *
 * @return Robot Model String
 */
static inline char *cls_arm_robot_get_model(arm_robot_t *const handler) {
    return handler->model;
}

/**
 * @brief Set Robot Position
 *
 * @param handler[in] Instance Handler
 * @param position[in] Position (mm)
 *
 * @return true Success
 * @return false Communication Failed
 */
static inline bool cls_arm_robot_set_position(arm_robot_t *const handler, xyz_t *
    position) {
    return handler->api->set_position(handler, position);
}

// ...

/**
 * @ref cls_arm_robot_set_position
 */
static bool set_position_interface(arm_robot_t *const handler, xyz_t *position) {
    (void)handler;
    (void)position;
    assert(0); // No Implementation
    return false;
}

// ...

void cls_arm_robot_init(arm_robot_t *const handler, arm_robot_init_t *const init) {
    assert(init->model);
    handler->model = init->model;
```

```
    static const struct arm_robot_api api = {
        .set_position = set_position_interface,
        .get_position = get_position_interface,
        .pick = pick_interface,
        .release = release_interface,
    };
    handler->api = &api;
}
```

In order to integrate all previously discussed Layer 2 modules and implement the structure provided by the ARIC module, the new Class module 3 Joint Arm Robot Class (3JAR) module was designed to support the specific implementation of the new robot (Listing 10). This Class instance handler comprises instances of the three joints and claw, along with the parent class, which combines all the system's individual parts. By providing the correct initialization of each one of those objects, the module becomes operational, requiring developers to implement only methods for the interaction between those objects.

Listing 10 demonstrates this by presenting the implementation of two virtual methods defined by the ARIC module. The first one ($three\_joint\_arm\_robot\_set\_position$) positions the arm in a three-dimensional coordinate by calculating the final angle required for each joint and repositioning them. Similar to that, the $three\_joint\_arm\_robot\_pick$ controls the claw to pick an object by closing it. Finally, with 3JAR development completed, an operational version of the completely new robot design could already be tested.

Listing 10 – Robot Example: Three Joint Arm Robot Class Module Implementation Fragments

```
#define JOINTS 3 /** Number of Joints */

/**
 * @brief Class Initialization Parameters
 */
typedef struct {
    arm_robot_init_t super;      /** Inherited Class Initialization */
    servo_motor_t joint[JOINTS]; /** Joints Servo Motor Initialization */
    servo_motor_t claw;          /** Claw Servo Motor Initialization */
} three_joint_arm_robot_init_t;

/**
 * @brief Class Instance Handler
 */
typedef struct {
    arm_robot_t super; /** Inherited Class */
    joint_t joint[3];  /** Robot Joints */
    claw_t claw;       /** Robot Claw */
} three_joint_arm_robot_t;

/**
 * @brief Create Class Object
 *
 * @param handler[out] Instance Handler
 * @param init[in] Initialization Parameters
 */
```

```
void cls_three_joint_arm_robot_init(three_joint_arm_robot_t *const handler,
    three_joint_arm_robot_init_t *const init);

// ...

/**
 * @ref cls_arm_robot_set_position
 */
static bool three_joint_arm_robot_set_position(arm_robot_t *const handler, xyz_t *
    position) {
    bool result = false;

    uint32_t angle[JOINTS] = {0};
    calculate_joint_angles(position, angle);

    three_joint_arm_robot_t *three_joint_arm_handler = (three_joint_arm_robot_t *)
    handler;

    for (uint32_t i = 0; i < JOINTS; ++i) {
        if (!cls_joint_set_angle(&three_joint_arm_handler->joint[i], angle[i])) {
            goto end;
        }
    }
    result = true;

end:
    return result;
}

// ...

/**
 * @ref cls_arm_robot_pick
 */
static bool three_joint_arm_robot_pick(arm_robot_t *const handler) {
    three_joint_arm_robot_t *three_joint_arm_handler = (three_joint_arm_robot_t *)
    handler;
    return cls_claw_close(&three_joint_arm_handler->claw);
}

// ...

void cls_three_joint_arm_robot_init(three_joint_arm_robot_t *const handler,
    three_joint_arm_robot_init_t *const init) {

    cls_arm_robot_init(&handler->super, &init->super); // Initialize Parent Class

    // Initialize Joints
    for (uint32_t i = 0; i < JOINTS; ++i) {
        joint_init_t joint_init = {
            .servo_motor = init->joint[i],
        };
        cls_joint_init(&handler->joint[i], &init->joint[i]);
    }

    // Initialize Claw
    claw_init_t claw_init = {
        .servo_motor = init->claw,
    };
```

```
    cls_claw_init(&handler->claw, &claw_init);

    static const struct arm_robot_api api = {
        .set_position = three_joint_arm_robot_set_position,
        .get_position = three_joint_arm_robot_get_position,
        .pick = three_joint_arm_robot_pick,
        .release = three_joint_arm_robot_release,
    };
    handler->api = &api;
}
```

4.8.3 Libraries

Libraries are single instance modules that may cluster a group of other modules to implement a system functionality. These modules are responsible for providing higher-level functionalities without the need for instances and for reducing the need to share and maintain objects in multiple different parts of the system.

As discussed in Section 4.8.2, Class modules seek to modularize behaviors by breaking system characteristics into smaller, easier-to-maintain structures. In contrast, the Library module's design goes in a different direction by bundling different smaller parts of the system to provide implementations for more advanced and complex functionalities that the system may require. Therefore, by abstracting the interactions between the multiple small elements into a single module, the system's workflows become more compact and focused on the functional aspects of the product.

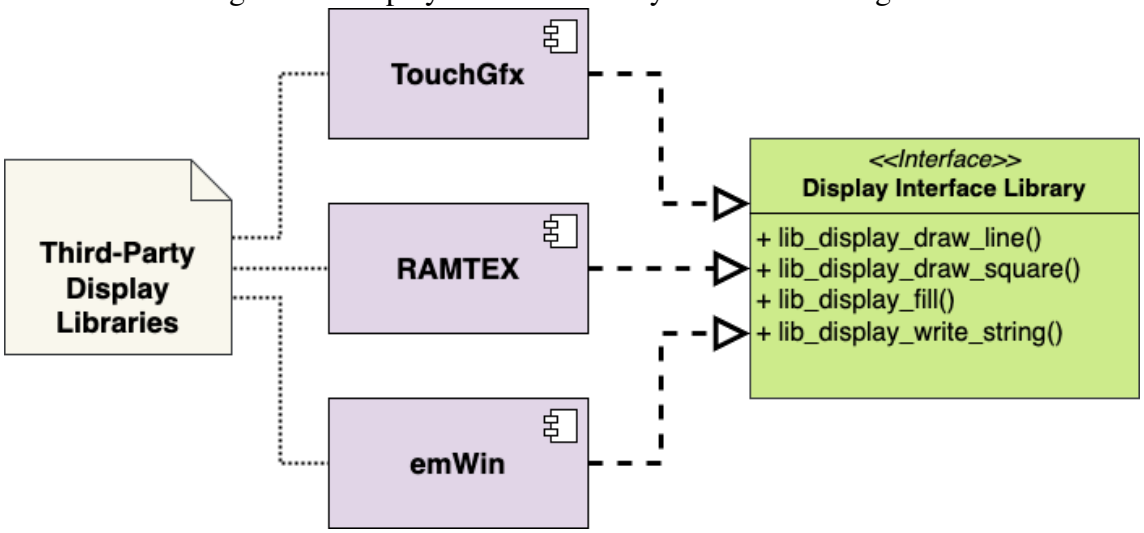
Libraries may also reduce complexity in implementing thread safety for multi-thread projects. For example, by maintaining instances of the small pieces they use inside their private environment, Library modules can have more control over access control and resource sharing of those elements, allowing them to provide external access protection in a grouped way.

The Library module also defines a module subtype called Interface Libraries. These subtype modules abstract and standardize third-party libraries into a single process flow. Following the previous LCD example (Section 4.3.2), every third-party display library could be added to the generic Interface Display Library, carrying out abstracted display functions (Figure 7). Also, the Interface Library could select the corresponding library using a run-time implementation or compile time through a macro compiler preprocessor.

The RTOS kernel itself should be seen as a third-party library for RTOS projects and, thus, be abstracted by an Interface Library. Some vendors already possess kernel abstraction libraries. For example, ARM has an API called CMSIS-RTOS that already abstracts several RTOS (RENAUX, 2014). Unfortunately, these libraries are usually architecture-specific and may not follow the requirements for the portability domain defined by the project Architect. Therefore, an in-house Interface Library should always abstract the RTOS kernel to avoid potential issues.

Finally, the Library module domain is presented in Table 11:

Figure 7 – Display Interface Library UML Class Diagram



Source: The Author

Table 11 – Library Domain

Implementation	Bare-Metal/RTOS
Access Layers	1 and 2
Design	Procedural

Analysis in the Robot Example

In ROBOT, the Robot Library module (Listing 11) has three main responsibilities:

- It is responsible for managing the robot's 3JAR instance.
- It adds thread safety to the robot's access and operation.
- It provides more advanced, project-specific functionalities for the system.

Since 3JAR is an External Domain module, it cannot include project modules such as the I2C Interface module. In order to be able to access the system peripherals required for its operations, 3JAR requires the aid of another Layer 2 Project Domain module so it can be initialized appropriately according to its specifications. The Robot Library module was added to the ROBOT firmware design for that. In its initialization process, it provides 3JAR with the required dependencies and configurations and proceeds with its instance initialization. As a result of this process, the robot finally becomes fully operational.

In order to meet the requirements for supporting the common Application modules, ARIC has already defined the API required for their integration with 3JAR. However, no considerations have been made to address the required multi-thread support. Although the 3JAR module could support multi-thread access, the team decided to address it by

controlling access to the robot through the Robot Library module to easily allow full control of 3JAR for a single application at a time.

Besides managing and controlling the access of 3JAR, the Robot Library module also provides project-specific implementations that could be used throughout the project and avoid repetitiveness. To exemplify this, the Library provides the function *lib_robot_move_item*, which commands the robot into the multiple steps of picking an object and moving it to another location.

Listing 11 – Robot Example: Robot Library Module Implementation Fragments

```
#define JOINT_0_ADDRESS 0x01 /** Joint 0 I2C Address */
#define JOINT_1_ADDRESS 0x02 /** Joint 1 I2C Address */
#define JOINT_2_ADDRESS 0x03 /** Joint 2 I2C Address */
#define CLAW_ADDRESS    0x04 /** Claw I2C Address */

#define DEFAULT_TIMEOUT 1000 /** Default Timeout (ms) */

static three_joint_arm_robot_t robot = {0}; /** Robot Instance */
static bool locked = false;                 /** Robot Lock Status */

/**
 * @brief Initialize Robot Library
 */
void lib_robot_init(void) {

    three_joint_arm_robot_init_t robot_init = {
        .super = {
            .model = "Three Joint Arm Robot",
        },
        .claw = {
            .i2c_address = CLAW_ADDRESS,
            .i2c_send = mid_i2c_send,
            .i2c_read = mid_i2c_read,
        },
    };
    robot_init.joint[0].i2c_address = JOINT_0_ADDRESS;
    robot_init.joint[0].i2c_send = mid_i2c_send;
    robot_init.joint[0].i2c_read = mid_i2c_read;
    robot_init.joint[1].i2c_address = JOINT_1_ADDRESS;
    robot_init.joint[1].i2c_send = mid_i2c_send;
    robot_init.joint[1].i2c_read = mid_i2c_read;
    robot_init.joint[2].i2c_address = JOINT_2_ADDRESS;
    robot_init.joint[2].i2c_send = mid_i2c_send;
    robot_init.joint[2].i2c_read = mid_i2c_read;

    cls_three_joint_arm_robot_init(&robot, &robot_init);
}

// ...

/**
 * @brief Request Robot Lock Access
 *
 * @param timeout[in] Timeout (ms)
 *
 * @return arm_robot_t* Robot Instance
 * @return NULL Timeout
```

```
 */
const arm_robot_t *lib_robot_lock(uint32_t timeout) {
    arm_robot_t *result = NULL;
    if (lock(timeout)) {
        result = (arm_robot_t *)robot;
    }
    return result;
}

// ...

/**
 * @brief Move item from start to end position
 *
 * @param start[in] Start Position (mm)
 * @param end[in] End Position (mm)
 *
 * @return true Success
 * @return false Busy Timeout or Communication Failed
 */
bool lib_robot_move_item(xyz_t *start, xyz_t *end) {

    bool result = false;

    if (!lock(DEFAULT_TIMEOUT)) {
        goto end;
    } else if (!cls_arm_robot_set_position((arm_robot_t *)&robot, start)) {
        goto communication_failed;
    } else if (!cls_arm_robot_pick((arm_robot_t *)&robot)) {
        goto communication_failed;
    } else if (!cls_arm_robot_set_position((arm_robot_t *)&robot, end)) {
        goto communication_failed;
    } else if (!cls_arm_robot_release((arm_robot_t *)&robot)) {
        goto communication_failed;
    } else {
        result = true;
    }

communication_failed:
    unlock();
end:
    return result;
}
```

4.9 Layer 3 Applications

Layer 3 is the last layer defined by the proposed methodology and comprises a single module type called Application Module. These modules are responsible for deploying every system workflow to provide the desired functionalities of a product.

Although the definition of Project Domain and External Domain, defined in Section 4.7, is limited to Layer 2, the proposed methodology does not restrict the reuse of Application modules. Instead, Architects can design the structure of tasks, APIs, and other elements to provide reusable Application modules.

The Application module domain is presented in Table 12:

Table 12 – Application Domain

Implementation	Bare-Metal/RTOS
Access Layers	2 and 3
Design	Procedural

Since the methodology supports both mono and multi-thread approaches, Application modules can be designed as either bare-metal state machines or RTOS threads. In the case of RTOS support, a typical problem that novice developers face is encountering unexpected patterns of behavior caused by their inexperience in the implications of running a code outside the context of the RTOS. For that, the methodology defines another NFC: every CORE should contain a System Application module (even for bare-metal projects).

It is important to iterate that the System Application module is not a module type but, yes, a unique application that should be seen as the project's main function. All the system initialization processes and starting points for other activities should be carried out inside this application, ensuring all the CORE's code is always executed in the RTOS environment.

As previously discussed, the methodology does not allow TARGET to include CORE code. However, each TARGET requires a starting point for running CORE. For that reason, the methodology defines that the System Application module as the only non-Layer 1 module that TARGET can access directly. Also, the only interaction allowed between the System and TARGET should be done through a single function that does not expect arguments or return any data. Finally, this should, strictly, be the only exception defined by the methodology, given that violating this principle may lead to non-portable systems.

Analysis in the Robot Example

As the final step of the ROBOT system development from Figure 6, the company's common Application modules were added to the project. Finally, as required by the methodology, the System Application was also added to provide the CORE's starting point for TARGET projects.

To conclude, the ROBOT example described a scenario capable of applying the complete structure of this methodology. By providing a design that covers all Layers and module types of the proposed methodology, ROBOT demonstrated some of the benefits the defined architecture can provide.

As a final consideration, even though the architecture describes a wide range of definitions that enable the developing high-quality systems, enforcing those definitions is crucial to ensuring those quality standards are met. To be able to extract the fullness of

the benefits provided by the proposed methodology, it is essential for teams to respect the responsibilities of the roles defined in Section 4.1 and follow the development (Figure 3) and maintainment (Figure 4) steps described in Section 4.2 to ensure the correctness of the proposed methodology's architecture implementation is achieved.

5 CASE STUDIES

Two different case studies are presented to evaluate the proposed methodology's effectiveness and benefits. In each case study, a comparison was made between the Legacy Project (LEGP) and the New Project (NEWP) based on the metrics results extracted by the code metric analyzer software Metrix++ (METRIX++, 2024). Finally, each study was selected to address different aspects of the methodology, providing a comprehensive assessment of its capabilities.

The first case study was based on an author's previous work. For that, the source code used to evaluate that work (LEGP) was refactored using the methodology approach into the NEWP (FARINA *et al.*, 2024). This project was chosen because it represents a typical scenario where the proposed methodology could be easily applied, allowing for an initial demonstration of its principles. The primary goal of this case study was to evaluate the methodology's impact on code modularity, portability, and maintainability in a controlled environment with straightforward requirements. This study highlights how the methodology can enhance development, even for smaller-scale projects.

In contrast, the second case study involves a more complex, real-world project with significantly higher levels of complexity and requirements. This study was designed to push the boundaries of the methodology, testing its scalability and adaptability in a more demanding environment. By being actual life employment of the proposed methodology in the industry, the chosen product was never intended to be used as a demonstration or to evaluate this methodology. Instead, the NEWP development team worked on its development without any guidance other than the company's development methodology training received during their orientation, making it an ideal candidate for assessing the benefits of the methodology in a real-world scenario.

Together, these two case studies provide a holistic evaluation of the proposed methodology. The first study serves as a proof of concept, demonstrating the methodology's benefits in a simpler and controlled context. In contrast, the second study validates its applicability and effectiveness in more complex, real-world scenarios. This dual approach ensures that the methodology is not only theoretically sound but also practically viable across different scales and complexities of embedded firmware development.

5.1 Case Study 1

For this first case study, an IoT system developed by the author in a previous work, was used to evaluate the aspects and benefits of the proposed methodology. The system supports an application that collects environmental data from IoT devices and provides fused data to end-users. Besides that, an essential feature of this system is the ability to autonomously calibrate new sensor nodes that are added to the network (FARINA; DOS ANJOS; DE FREITAS, 2023).

In that work, a proof of concept system was implemented to test and evaluate the proposed system. For that, the author developed a monolithic firmware (LEGP) whose only intention was to be used for that. Therefore, no consideration was made to ensure the quality of the developed firmware. Refactoring the firmware for this self-calibration sensor device (NEWP) according to the structure described in Figure 5 made it possible to compare LEGP and NEWP and to demonstrate the value of applying the proposed methodology to firmware development.

The target hardware used for this firmware was an ESP32-WROOM-32 Module from Espressif, and the official vendor framework (ESP-IDF) was used to develop the hardware-specific implementations (ESPRESSIF SYSTEMS, 2017). Following the structure defined by the proposed methodology, each firmware region was implemented in a different repository. First, TARGET [1] was created, and then the initial project files were added. Next, CORE [2] was built and included as a submodule of TARGET.

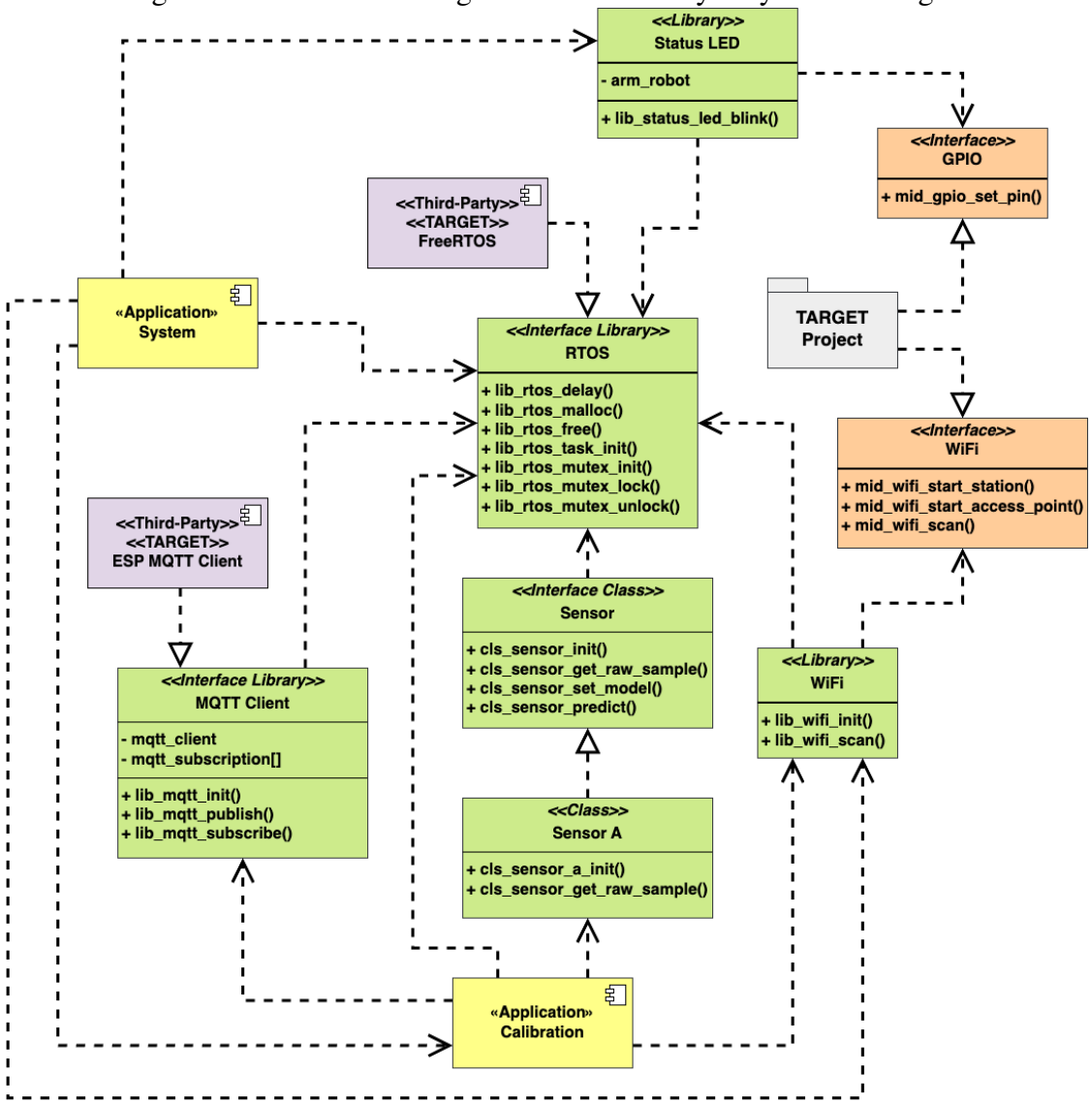
In NEWP, CORE was designed according to Figure 8. For that, a Calibration Application module was designed to provide the system's self-calibration workflow. Besides it, as required by the methodology, the System Application module was also implemented to provide the required initialization and start the calibration task.

Next, Layer 2 modules were implemented to provide the required functionalities for those applications. For this project, similarly to the display example (Figure 7), third-party libraries (FreeRTOS and ESP MQTT Client) were abstracted by two different Interface Library modules (RTOS and MQTT Client, respectively). Besides those, two other Library modules were designed to provide implementations for the Status LED and the WiFi connectivity (Status LED and WiFi Library modules). Finally, the Sensor Interface Class module was used to define and enforce the requirements for the different types of heterogeneous sensors that could be used, and the Sensor A Class module was implemented as an example of one of those sensors.

To conclude the design of CORE, two Layer 1 modules were included to abstract TARGET. First, the GPIO Interface module abstracted the access to the GPIO peripheral required for controlling the status LED. Next, the WiFi Interface module provided access

[1] https://github.com/mauriciofarina/Framework-Example-Project
[2] https://github.com/mauriciofarina/Framework-Core-Example

Figure 8 – UML Class Diagram for Case Study 1 System's Design



Source: The Author

to the wireless communication peripheral.

5.1.1 Code Comparison

The implementation of the calibration process workflow for the two projects was used to evaluate and compare LEGP and NEWP. First, Listing 12 displays the implementation LEGP. On it, it is possible to observe several items that are potential indicators of poor design:

- Excessive number of LOC
- High MNL and MVG

- Bad readability
- Lack of separation between concerns
- Strong dependency coupling

Understanding and maintaining this code can be challenging, even for experienced developers. As a result, the aspects described by (BENINGO, 2022) in tables 1 and 2 about the increased chance of introducing problems along with code changes in complex code become evident. Besides maintainability, the mix of system-level, target-specific, and third-party implementations significantly reduces portability and the chance of reuse of this code in the future.

Listing 12 – Legacy Project Calibration Process Fragments

```
static void mqtt_event_handler(void *handler_args, esp_event_base_t base, int32_t
    event_id, void *event_data) {

    esp_mqtt_event_handle_t event = event_data;
    esp_mqtt_client_handle_t client = event->client;

    switch ((esp_mqtt_event_id_t)event_id) {
        case MQTT_EVENT_CONNECTED:
            ESP_LOGI(TAG, "MQTT_EVENT_CONNECTED");
            esp_mqtt_client_publish(client, last_will_topic, "ONLINE", 0, 2, 1);

            char topic[32] = {0};
            sprintf(topic, "/%d/+", SENSOR_ID);
            esp_mqtt_client_subscribe(client, topic, 1);
            break;
        case MQTT_EVENT_DISCONNECTED:
            ESP_LOGI(TAG, "MQTT_EVENT_DISCONNECTED");
            break;
        case MQTT_EVENT_SUBSCRIBED:
            ESP_LOGI(TAG, "MQTT_EVENT_SUBSCRIBED, msg_id=%d", event->msg_id);
            break;
        case MQTT_EVENT_UNSUBSCRIBED:
            ESP_LOGI(TAG, "MQTT_EVENT_UNSUBSCRIBED, msg_id=%d", event->msg_id);
            break;
        case MQTT_EVENT_PUBLISHED:
            break;
        case MQTT_EVENT_DATA: {
            char topic[32] = {0};
            memcpy(topic, event->topic, event->topic_len);

            switch (event->topic[3]) {

                case '1': // Receive Image
                {
                    if (event->data_len != sizeof(data_sample_t)) {
                        ESP_LOGE(TAG, "SIZE ERROR 1 (GOT %u EXPECT %u)", event->data_len
    , sizeof(data_sample_t));
                    } else if (xSemaphoreTake(image_receive_semaphore, 0) == pdTRUE) {
                        memcpy(&data_sample, event->data, sizeof(data_sample_t));
                        update_vector_clock(&data_sample.timestamp);
                        xSemaphoreGive(image_receive_semaphore);
```

```
                }
            } break;
            case '2': // Request Image
            {
                if (event->data_len != sizeof(vector_clock_t)) {
                    ESP_LOGE(TAG, "SIZE ERROR 2 (GOT %u EXPECT %u)", event->data_len
, sizeof(vector_clock_t));
                } else if (xSemaphoreTake(image_receive_semaphore, 0) == pdTRUE) {
                    update_vector_clock((vector_clock_t *)event->data);
                    char topic[32] = {0};
                    sprintf(topic, "/SENSOR/%d/2", SENSOR_ID);
                    update_vector_clock(NULL);
                    memcpy(&data_sample.timestamp, &sensor_clock, sizeof(
vector_clock_t));
                    esp_mqtt_client_publish(client, topic, (char *)&data_sample,
sizeof(data_sample_t), 0, 0);
                    xSemaphoreGive(image_receive_semaphore);
                }
            } break;
            case '3': // Request Prediction
            {
                if (event->data_len != sizeof(vector_clock_t)) {
                    ESP_LOGE(TAG, "SIZE ERROR 3 (GOT %u EXPECT %u)", event->data_len
, sizeof(vector_clock_t));
                } else {
                    if (xSemaphoreTake(image_receive_semaphore, 0) == pdTRUE) {
                        update_vector_clock((vector_clock_t *)event->data);
                        xSemaphoreGive(image_receive_semaphore);
                    }
                    xSemaphoreGive(predict_semaphore);
                }
            } break;
            case '4': // Scan Devices
            {
                wifi_ap_record_t ap_records[30] = {0};
                uint16_t records = 30;
                wifi_scan(ap_records, &records);

                ap_info_t ap_info[records];
                for (uint32_t i = 0; i < records; ++i) {
                    memcpy(ap_info[i].ssid, ap_records[i].ssid, 33);
                    ap_info[i].rssi = ap_records[i].rssi;
                }

                char topic[32] = {0};
                sprintf(topic, "/SENSOR/%d/4", SENSOR_ID);
                esp_mqtt_client_publish(client, topic, (char *)&ap_info, (sizeof(
ap_info_t) * records), 2, 0);
            } break;
        }
    } break;
    case MQTT_EVENT_ERROR:
        ESP_LOGI(TAG, "MQTT_EVENT_ERROR");
        esp_restart();
        break;
    default:
        ESP_LOGI(TAG, "Other event id:%d", event->event_id);
        break;
}
```

```
}
```

In contrast, in NEWP, the implementation of the same workflow is done by the Calibration Application module (Listing 13), and it is possible to observe a visually more organized design on it. In further analysis, it is possible to notice that the different parts of this workflow are clearly separated according to the system's different modules.

Opposite to LEGP, which mixes MQTT protocol aspects, system API structure, and calibration workflow, NEWP separates those concerns into different modules. As a result, it becomes easier to focus and more straightforward to understand the specifics of the Calibration Application module workflow since large parts of the development required for it are encapsulated in the other specialized modules. Consequently, novice developers can easily collaborate in NEWP since introducing code changes does not require extensive development experience to mitigate the high risk of adding problems to the system anymore. Along with that, advanced C language patterns, such as callbacks, could be used without compromising the comprehension of novice developers.

Listing 13 – New Project Calibration Application Module Fragments

```
static void get_raw_sample_callback(char *topic, uint8_t *data, uint32_t length) {
    uint8_t buffer[128] = {0};
    uint32_t size = cls_sensor_get_raw_sample(sensor, buffer);
    lib_mqtt_client_publish(response_topic, buffer, size, 0);
}
static void predict_callback(char *topic, uint8_t *data, uint32_t length) {
    uint32_t label = cls_sensor_predict(sensor);
    lib_mqtt_client_publish(response_topic, (uint8_t *)&label, sizeof(uint32_t), 0);
}
static void set_model_callback(char *topic, uint8_t *data, uint32_t length) {
    cls_sensor_set_model(sensor, data, length);
}
static void scan_devices_callback(char *topic, uint8_t *data, uint32_t length) {
    char buffer[512] = {0};
    lib_wifi_scan(buffer);
    lib_mqtt_client_publish(response_topic, (uint8_t *)buffer, strlen(buffer), 0);
}

static void subscribe_topics(void) {

    mqtt_subscription_t subscribe = {
        .qos = 0,
    };

    sprintf(subscribe.topic, "/%d/raw", SENSOR_ID);
    subscribe.callback = get_raw_sample_callback;
    lib_mqtt_client_subscribe(&subscribe);

    sprintf(subscribe.topic, "/%d/predict", SENSOR_ID);
    subscribe.callback = predict_callback;
    lib_mqtt_client_subscribe(&subscribe);

    sprintf(subscribe.topic, "/%d/model", SENSOR_ID);
    subscribe.callback = set_model_callback;
    lib_mqtt_client_subscribe(&subscribe);
```

```
    sprintf(subscribe.topic, "/%d/scan", SENSOR_ID);
    subscribe.callback = scan_devices_callback;
    lib_mqtt_client_subscribe(&subscribe);
}

static void app_calibration_task(void *context) {

    (void)context;

    subscribe_topics();

    while (1) {
        lib_mqtt_client_loop();
    }
}
```

The improvements in the reuse aspects can also be observed by comparing Listing 12 and Listing 14. In their corresponding projects, both of those implementations were placed into an MQTT Client context. However, in LEGP, other non-related implementations were added, resulting in a particular implementation that cannot be applied for a different purpose. In contrast, NEWP provides a generic implementation of the MQTT Client that does not contain any project-specific implementations, thus making it possible to reuse it in several other projects.

Listing 14 – New Project MQTT Client Interface Library Module Fragments

```
static void mqtt_data_event_handler(esp_mqtt_event_handle_t *event) {

    for (uint32_t i = 0; i < MQTT_MAX_SUBSCRIPTIONS; ++i) {

        if (strlen(mqtt_subscription[i].topic) == event->topic_len) {
            if (0 == memcmp(event->topic, mqtt_subscription[i].topic, event->topic_len))
     {
                mqtt_subscription[i].callback(event->topic, (uint8_t *)event->data, (
    uint32_t)event->data_len);
                break;
            }
        }
    }
}

static void mqtt_event_handler(void *handler_args, esp_event_base_t base, int32_t
    event_id, void *event_data) {

    esp_mqtt_event_handle_t event = event_data;

    switch ((esp_mqtt_event_id_t)event_id) {
        case MQTT_EVENT_DATA:
            ESP_LOGI("MQTT_CLIENT", "GOT DATA %s %d", event->topic, event->topic_len);
            mqtt_data_event_handler(&event);
            break;
        case MQTT_EVENT_ERROR:
            ESP_LOGE("MQTT_CLIENT", "MQTT_EVENT_ERROR");
            assert(0);
            break;
        default:
            // Ignore
```

```
            break;
    }
}

// ...

void lib_mqtt_client_subscribe(mqtt_subscription_t *subscription) {

    for (uint32_t i = 0; i < MQTT_MAX_SUBSCRIPTIONS; ++i) {

        if (NULL == mqtt_subscription[i].callback) {
            memcpy(&mqtt_subscription[i], subscription, sizeof(mqtt_subscription_t));
            break;
        }
        assert(i != (MQTT_MAX_SUBSCRIPTIONS - 1)); // Check if Buffer is full
    }

    esp_mqtt_client_subscribe(mqtt_client, subscription->topic, subscription->qos);
}
```

5.1.2 Region Analysis

The initial analysis (Table 13) compares the outlines of both projects, and the results show a significant increase in all three regions. This was expected since the methodology enhances the system modularization; thus, more files are required to encapsulate these modules.

Also, to improve readability and maintainability, the methodology advises that the LOC in a function should not exceed 40. The reason for this is that most IDEs can display at least this number of lines (in their default configuration) without requiring any scrolling. As a consequence, functions become less complex and more specialized.

Modularization of systems promotes reuse, and therefore, it would be expected that NEWP presented fewer functions in comparison with LEGP (as demonstrated in Section 5.2). However, in a project with such a small scope, it becomes unavoidable for NEWP to require more functions to modularize the few monolithic functions of LEGP. Because of that, further analysis will be provided in Section 5.1.3 to evaluate if the expressive changes in the number of functions resulted in system overheads.

Table 13 – Case Study 1 Region Overview

	Legacy	**New**
Modules	4	10
Files	10	33
Functions	17	45

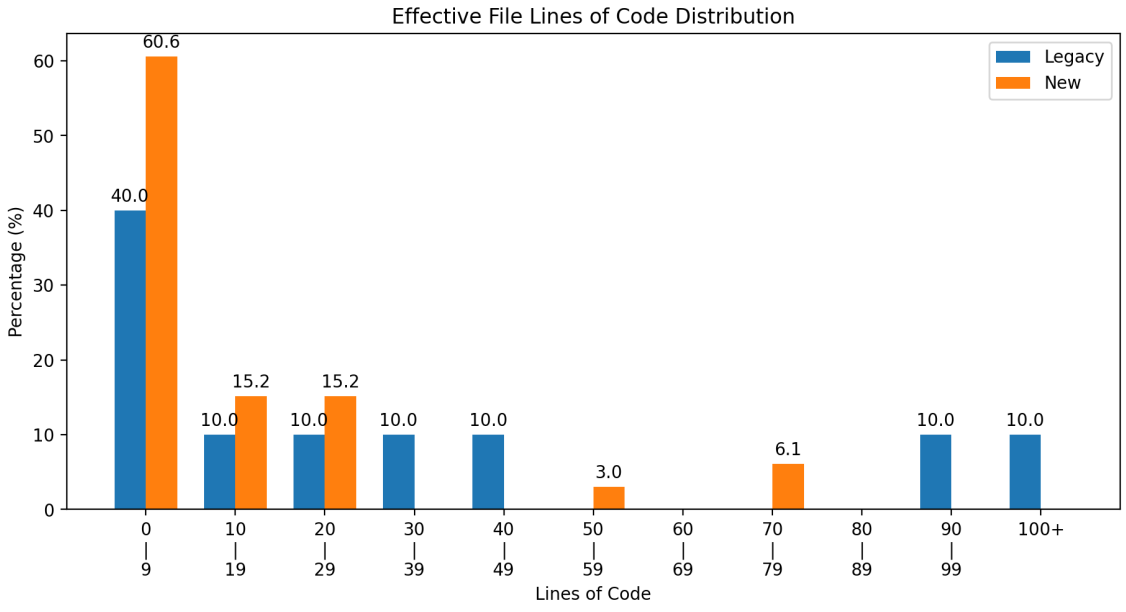
5.1.3 Lines of Code

Projects with less LOC tend to be more readable, maintainable, and less error-prone. Since both projects execute the same system flow, a smaller LOC could indicate improved quality of the developed code. Since LEGP was developed in a certain monolithic way and it was not expected to be used after the experiment it was designed for, no efforts were made to provide documentation comments, configuration preprocessors, and other similar elements. For that reason, those metrics will be disregarded for this analysis.

As the first comparison, Table 14 shows that the TLOC and, for the same reasons of Section 5.1.2, TELOC for NEWP have increased more than double that of the previous version. Although this may seem like a negative result for maintainability, it is important to remember that smaller and simple functions have a linear impact on maintainability, while large and complex exponential (Section 2.6.4). Therefore, TLOC should not be considered for small projects since it may result in misleading conclusions.

In contrast to TLOC and TELOC, LOCFI and ELOCFI have significantly decreased. By increasing the number of modules and functions in the project, files and functions have become more specialized and, therefore, easier to reuse and maintain. Finally, Figure 9 shows that files are now distributed in a better ELOC range in a more specialized way.

Figure 9 – Effective File Lines of Code Distribution for Case Study 1



Source: The Author

Next, the ELOCFU results are shown in Table 15. The results show that NEWP functions are smaller and have fewer size variations than the LEGP functions, confirming that functions are now more specialized and easier to understand and maintain. Moreover, the lower standard deviation indicates that developers got more consistent in their function implementation scopes.

Table 14 – File Lines of Code results for Case Study 1

	Legacy	**New**	**Difference**	**Change**
Code				
Mean	38.6	14.1	-24.5	-63.4%
Standard Deviation	48.3	18.4	-29.9	-62.0%
Min	2	0	-2	-100.0%
Max	160	76	-84	-52.5%
Total	386	466	80	20.7%
Comments				
Mean	1.0	15.8	14.8	1484.8%
Standard Deviation	1.9	13.1	11.2	590.4%
Min	0	5	5	100.0%
Max	6	51	45	750.0%
Total	10	523	513	5130.0%
Preprocessor				
Mean	9.2	7.3	-1.9	-20.3%
Standard Deviation	4.9	7.1	2.2	43.8%
Min	3	0	-3	-100.0%
Max	21	32	11	52.4%
Total	92	242	150	163.0%
Total				
Mean	48.4	35.7	-12.7	-26.2%
Standard Deviation	46.9	25.5	-21.4	-45.7%
Min	10	6	-4	-40.0%
Max	167	98	-69	-41.3%
Total	484	1179	695	143.6%

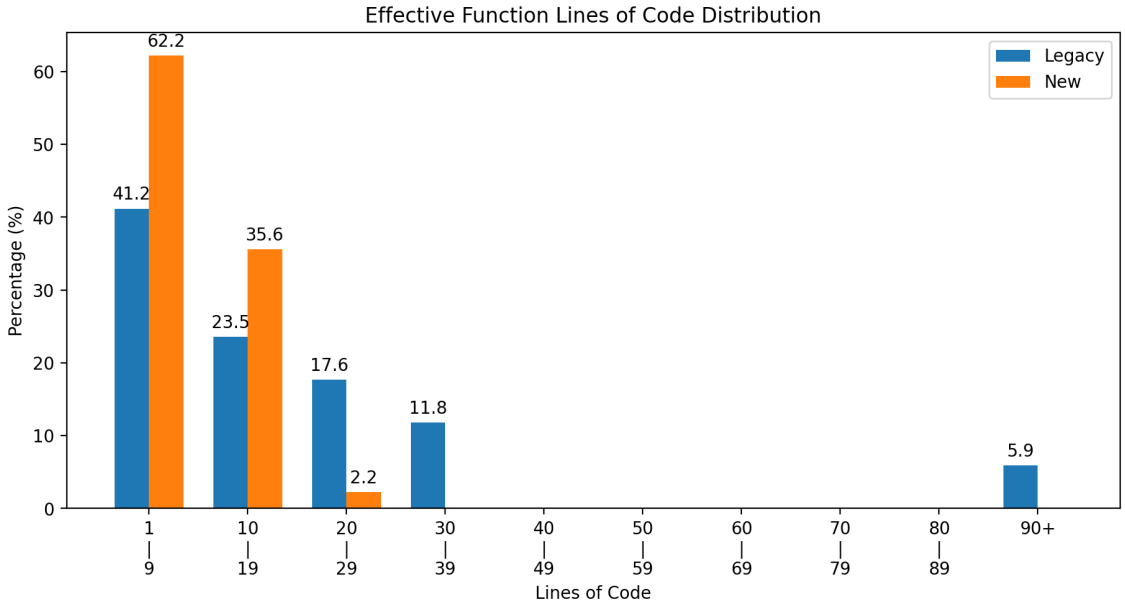
Table 15 – Effective Function Lines of Code results for Case Study 1

	Legacy	**New**	**Difference**	**Change**
Mean	19.1	7.8	-11.3	-59.3%
Standard Deviation	20.9	4.4	-16.6	-79.1%
Min	3	3	0	0.0%
Max	91	20	-71	-78.0%
Total	325	350	25	7.7%

Even though most of the LEGP functions are in a good ELOC range, many present an expressive size compared to NEWP (Figure 10). This hints that the functions may be more complex than necessary. However, this can only be confirmed by the analysis of

their complexity.

Figure 10 – Effective Function Lines of Code Distribution for Case Study 1



Source: The Author

Returning to the Section 5.1.2 discussion about NEWP's expressive increase in the number of functions, further conclusions can be obtained with the ELOCFU metric. Even though the number of functions increased, Table 15 displays that the total number of ELOCFU has increased by less than 10%. This indicates that, although more functions were added, no significant overhead in instructions was observed, and consequently, no considerable system overhead should be expected.

5.1.4 Maximum Nesting Level

The MNL comparison between LEGP and NEWP is shown in Table 16, and the results show that both projects present good results in terms of readability. However, by analyzing the distribution of functions MNL (Figure 11), compared with LEGP, it is possible to see that a more significant portion of NEWP functions is distributed in a lower MNL range. Those results indicate that the NEWP functions are easier to read and understand and, therefore, to maintain and reuse.

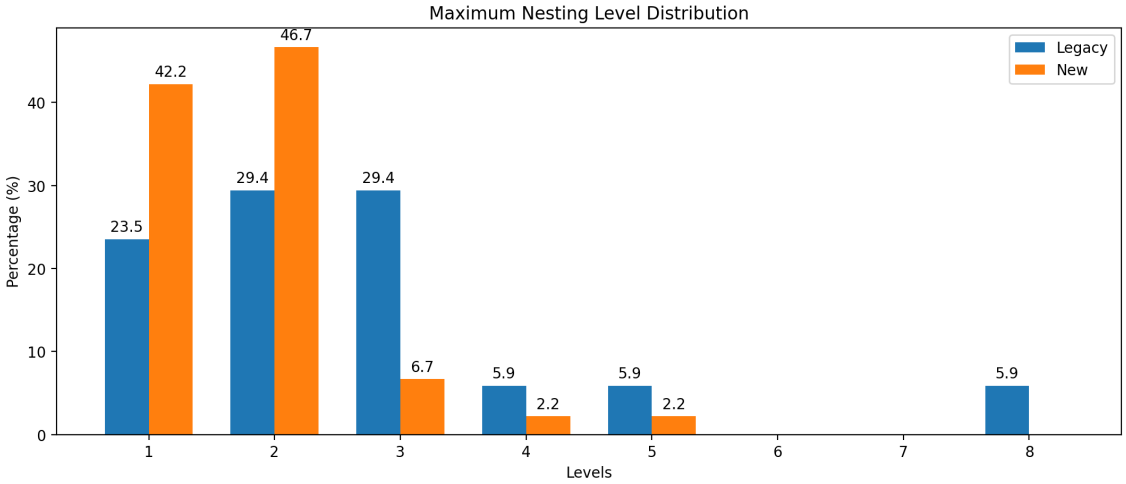
5.1.5 McCabe's Cyclomatic Complexity

The MVG results provided by Table 17 show that NEWP significantly decreases 64.3% in the average complexity of functions. Besides that, NEWP stays only in a low-risk range, while LEGP contains functions in a medium-risk range (Figure 12). Also, a significant increase in complexity 1 functions in NEWP confirms that functions are indeed more specialized.

Table 16 – Maximum Nesting Level results for Case Study 1

	Legacy	**New**	**Difference**	**Change**
Mean	2.7	1.8	-1.0	-35.1%
Standard Deviation	1.7	0.8	-0.9	-50.4%
Min	1	1	0	0.0%
Max	8	5	-3	-37.5%
Total	46	79	33	71.7%

Figure 11 – Maximum Nesting Level Distribution for Case Study 1



Source: The Author

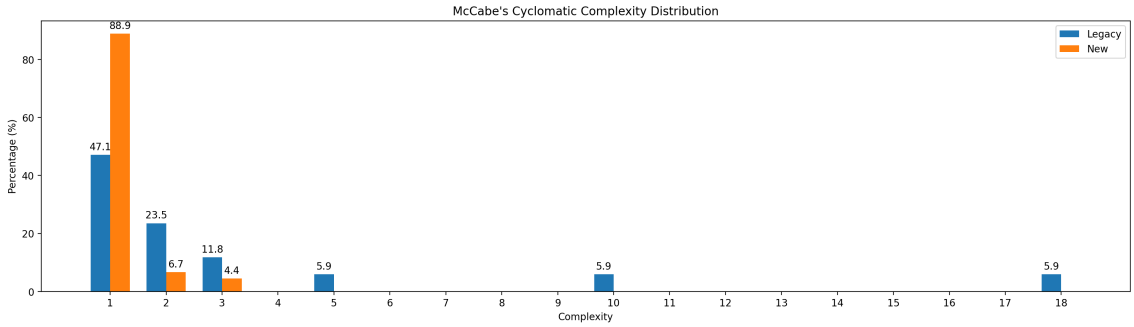
Table 17 – McCabe's Cyclomatic Complexity results for Case Study 1

	Legacy	**New**	**Difference**	**Change**
Mean	3.2	1.2	-2.1	-64.3%
Standard Deviation	4.3	0.5	-3.8	-89.1%
Min	1	1	0	0.0%
Max	18	3	-15	-83.3%
Total	55	52	-3	-5.5%

Next, LEGP and NEWP bug risks are compared in Table 18. The results show that the chance of changes introducing bugs in NEWP is around 15% lower than in LEGP. Even though this is already a significant improvement for projects of this size, it becomes more expressive when we consider that the NEWP has almost 8% more ELOC than LEGP.

Finally, the MVG results were used by a script to calculate the system's flow with the maximum number of paths (Table 19). The results show a significant decrease in possible paths within a single function. These results demonstrate that as well as portability and reuse, the methodology can improve the quality of the developed code, enabling it to be more easily understood, changed, and debugged.

Figure 12 – McCabe's Cyclomatic Complexity Distribution for Case Study 1



Source: The Author

Table 18 – Risk of Bugs and Changes of Bug Injection results for Case Study 1

	Low (5%)	**Medium (20%)**	**High (40%)**	**Very High (60%)**	**Bug Injection Risk**
Legacy	16 (94.1%)	1 (5.9%)	0 (0.0%)	0 (0.0%)	5.9%
New	45 (100.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	5.0%

Table 19 – Code Paths for Case Study 1

	Paths
Legacy	129600
New	72

It is essential to mention that this metric does not evaluate paths for functions that are not system-level implementation. This means that for LEGP, all drivers, HAL, third-party, and auto-generated functions were disregarded. Additionally, only CORE functions were considered for NEWP (except third-party functions). Since this value does not introduce new paths, all non-system-level functions were set as complexity 1 functions. Finally, to simplify the analysis, all loops were disregarded. This means that functions inside a loop were considered to be called only once.

5.1.6 Simple Manintaibility Index

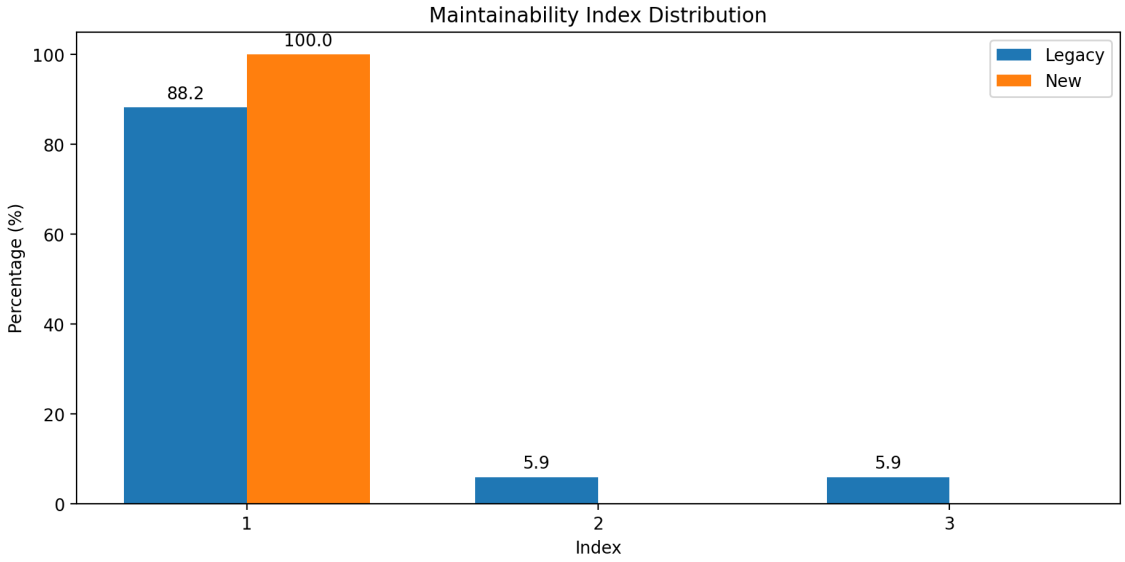
Table 20 provides the results for the SMI metric. Even though most LEGP functions have SMI 1 values (Figure 13) and a good ASMI, all NEWP functions have SMI 1 values, which provides the best ASMI possible.

This metric may not be very expressive in terms of improvement for small projects. However, in this case study, an important aspect to observe is how much a few bad implementations can affect the result of the project's overall maintainability. Even though LEGP has only two functions with SMI greater than 1, that resulted in an ASMI 78.8% worse than NEWP.

Table 20 – Maintainability Index results for Case Study 1

	Legacy	**New**	**Difference**	**Change**
Mean	1.2	1.0	-0.2	-15.0%
Standard Deviation	0.5	0.0	-0.5	-100.0%
Min	1	1	0	0.0%
Max	3	1	-2	-66.7%
Total	20	45	25	125.0%
Overall Index	4.6	1.0	-3.6	-78.8%

Figure 13 – Maintainability Index Distribution for Case Study 1



Source: The Author

5.1.7 Module Coupling

In LEGP, no abstraction was made to separate ESP-IDF from the system functions. For this reason, the required dependencies from the API vendor were added to this analysis. In the case of the NEWP version, implementations on the TARGET do not directly interact with CORE modules. Therefore, NEWP only takes into account the dependencies that are included inside of CORE. The LEGP and NEWP dependency maps are shown in Table 21 and Table 22, respectively.

Since both projects make use of an RTOS, it is natural that multiple modules depend on either $FreeRTOS$ (LEGP) or lib_rtos (NEWP). These dependencies on methodology modules could be reduced by including kernel functions within the module initialization parameters. However, the RTOS can be regarded as a global dependency for project modules.

Application modules are designed to use different other modules to implement the workflow of system processes. As discussed in Section 4.9, although Application modules

Table 21 – Case Study 1 Dependency Map for Legacy Project

Dependency	**Module**				**Dependency of Total**
	mqtt	wifi	model	main	
mqtt		X		X	**2**
wifi	X			X	**2**
model	X			X	**2**
main					**0**
esp-event	X				**1**
esp-wifi		X			**1**
esp-nvs				X	**1**
esp-gpio				X	**1**
esp-mqtt	X				**1**
FreeRTOS	X	X	X	X	**4**
Total Dependencies	**5**	**3**	**1**	**6**	

Table 22 – Case Study 1 Dependency Map for New Project

Dependency	**Module**										**Dependency of Total**
	mid_gpio	mid_wifi	cls_sensor	cls_sensor_demo	lib_mqtt	lib_rtos	lib_status_led	lib_wifi	app_calibration	app_system	
mid_gpio							X				**1**
mid_wifi								X			**1**
cls_sensor				X							**1**
cls_sensor_demo									X		**1**
lib_mqtt									X		**1**
lib_rtos			X		X		X	X	X	X	**6**
lib_status_led										X	**1**
lib_wifi									X	X	**2**
app_calibration										X	**1**
app_system											**0**
esp-mqtt					X						**1**
FreeRTOS						X					**1**
Total Dependencies	**0**	**0**	**1**	**1**	**2**	**1**	**2**	**2**	**4**	**4**	

can be reusable, the proposed methodology does not define rules to enforce it. Therefore, it is acceptable that Application modules may rely on multiple dependencies.

Finally, the quality of the module encapsulation of NEWP is clear from the results obtained for the other modules. In contrast with the LEGP, each NEWP module is completely decoupled from the others, evidencing high reusability characteristics. Moreover, these modules can be replaced by different module versions without implicating changes in the other modules.

5.2 Case Study 2

Even though Case Study 1 (CS1) was a great starting point to evaluate the benefits of this methodology, the results obtained from it may be shallow and insufficient to provide

a clear picture of the methodology's potential. Therefore, a second case study (CS2) was conducted to evaluate the methodology further. To provide a more realistic scenario, a comparison was made between two firmware versions of the same product from Luminator Technology Group (LTG). The first version (LEGP) was developed using traditional software development practices, while the second version (NEWP) was developed using the methodology proposed in this work.

To contextualize, LTG is a company that develops and manufactures electronic equipment for the public transportation market. During the 2020 pandemic and the consequent global microchip supply chain shortage, LTG faced a challenge to keep up with the demand for their products. One of their products, an LED display controller, was particularly affected by the shortage, as it relied on a microcontroller that was no longer available. To overcome this challenge, the LTG firmware department was tasked with developing a new version of the controller using a different microcontroller that was available in the market.

Since both products were expected to provide the exact same functionalities, the initial idea was to port LEGP to the new microcontroller. However, after evaluating the LEGP code, the assigned development team concluded that it was developed with no considerations for portability, making porting it a very time-consuming task. Considering this issue, along with the fact that the proposed methodology was already being applied to all new projects, the team decided to pursue another approach and developed NEWP from scratch. The NEWP was then able to take advantage of the methodology's benefits. For example, the developers were already familiar with the methodology's structure and could easily identify previously developed modules that could be reused in the new project. This allowed the NEWP to be developed much faster since the developers could focus primarily on developing product-specific features and functionalities.

Even though the selected microcontroller was available in the market at the time, there were no guarantees that LTG would again face long lead times or even shortages of this component in the near future. For this, the methodology's portability aspect provided an essential benefit to the company. Due to its portability, NEWP could be easily ported to a different microcontroller if needed, reducing the risks of future supply chain issues. Moreover, NEWP could also be ported to the previous microcontroller used in LEGP, allowing the same system to be used for both products and reducing maintenance costs.

Since this is an LTG proprietary project, no sensitive or project-specific data can be shared. However, similar to CS1, the required metric data was collected to evaluate the methodology's benefits. The following sections present the results obtained from comparing LEGP and NEWP.

5.2.1 Performance Analysis

In order to ensure the quality of the images displayed by the LTG LED displays, internal tests proved that a minimal 120 frames-per-second rate is required. The high

frame rate is required because those LED displays are designed with eight channels of multiplexation, which, in comparison with other types of display, requires a higher refresh rate to provide a stable image to the user.

The processing times for frames were measured to evaluate that the multiple levels of abstraction and encapsulation, defined by the proposed methodology, did not introduce performance overhead that prevent the system from achieving the expected performance requirements. For that, the devices were configured using the same configuration and image files. A GPIO pin was used to generate a square wave, which was measured with the support of a logic analyzer to observe the processing time of single frames.

For each firmware version, the time for 7200 frames was measured, and the results are presented in Table 23:

Table 23 – Frame Processing Time Performance Evaluation

	Legacy	**New**
Mean	6.72 ms	5.40 ms
Standard Deviation	1.37 ms	0.27 ms
Min	4.70 ms	4.93 ms
Max	12.74 ms	5.88 ms
Missed Deadlines Count	327	0

The results show that although LEGP can reach faster processing times, it does not ensure the RT constraints to provide the quality requirements of this system. Besides this, LEGP's processing time for different samples does not present consistency, resulting in the unpredictability of the system behavior.

In contrast, NEWP provides expressive improvements in the RT aspects of the system. By maintaining a more consistent processing time and a minor standard deviation, NEWP can successfully ensure that the system's RT requirements are met.

In summary, the results demonstrate that, in contrast with LEGP, which exhibits inconsistencies in frame processing times and numerous missed deadlines, the NEWP version consistently meets the real-time constraints required by the system, ensuring greater predictability and stability. This confirms that the proposed methodology does not compromise the system's performance but significantly enhances its real-time characteristics.

5.2.2 Region Analysis

Similar to CS1, the initial overview analysis (Table 24) shows that NEWP has more modules and files than LEGP. Since this project is of a much larger scale than CS1, the reuse aspects of modularization discussed in Section 5.1.2 can be better observed in this study.

In contrast to CS1, the number of functions in CS2 was reduced by almost 38% from

LEGP, demonstrating that a modular approach with well-architected modules can have a more comprehensive use domain of functions. Therefore, the reduction in the number of functions results from reusing functions in other parts of the project.

Table 24 – Case Study 2 Region Overview

	Legacy	**New**
Modules	22	50
Files	152	217
Functions	915	569

5.2.3 Lines of Code

For the LOC analysis, Table 25 presents the results for project and file LOC levels. The results show a drastic reduction of 83.7% and 90.3%, respectively, in the TLOC and TELOC of NEWP compared to LEGP. Based on this reduction, a few conclusions can be drawn.

Since both projects were designed to provide the exact same functionalities, the results demonstrate that using the methodology allowed developers to achieve the same effective results with a much smaller codebase. Even though a LOC metric does not directly measure the required effort to develop a project, it can provide a rough estimate that NEWP demanded just about 10% of the work required for LEGP. Additionally, it is important to consider that the NEWP uses previously developed modules, which were also included in this codebase size.

Besides reducing the development work required, a smaller codebase also provides other benefits. Smaller project codebases can be easier to understand since less information needs to be processed. Also, they are less prone to bugs, as there are fewer lines of code to contain bugs or to find them. The binary size is also reduced, which can be important for embedded systems with limited memory.

Since such a drastic reduction in codebase size was achieved, the number of comments and preprocessors is expected also to be reduced. Despite the reduction, for every 100 ELOC in LEGP, there are about 17 lines of comment, while in NEWP, there are approximately 91 lines of comment. This difference may indicate that NEWP has better code documentation. Moreover, a higher density of preprocessors in NEWP may indicate fewer magic numbers, thus improving code readability and maintainability.

For the LOC analysis, Table 25 presents the results for project and file LOC levels. The results show a drastic reduction of 83.7% and 90.3%, respectively, in the TLOC and TELOC of NEWP compared to LEGP. Based on this reduction, a few conclusions can be drawn.

Since both projects were designed to provide the exact same functionalities, the results

Table 25 – File Lines of Code results for Case Study 2

	Legacy	**New**	**Difference**	**Change**
Code				
Mean	573.6	39.0	-534.6	-93.2%
Standard Deviation	1702.2	86.7	-1615.5	-94.9%
Min	0	0	0	0.0%
Max	9181	1141	-8040	-87.6%
Total	87192	8464	-78728	-90.3%
Comments				
Mean	99.7	35.6	-64.1	-64.3%
Standard Deviation	166.2	27.7	-138.5	-83.3%
Min	0	12	12	100.0%
Max	921	164	-757	-82.2%
Total	15155	7718	-7437	-49.1%
Preprocessor				
Mean	22.9	7.4	-15.5	-67.6%
Standard Deviation	37.1	9.4	-27.7	-74.6%
Min	0	1	1	100.0%
Max	279	88	-191	-68.5%
Total	3487	1612	-1875	-53.8%
Total				
Mean	683.6	77.9	-605.6	-88.6%
Standard Deviation	1724.0	91.2	-1632.7	-94.7%
Min	4	15	11	275.0%
Max	9199	1160	-8039	-87.4%
Total	103904	16914	-86990	-83.7%

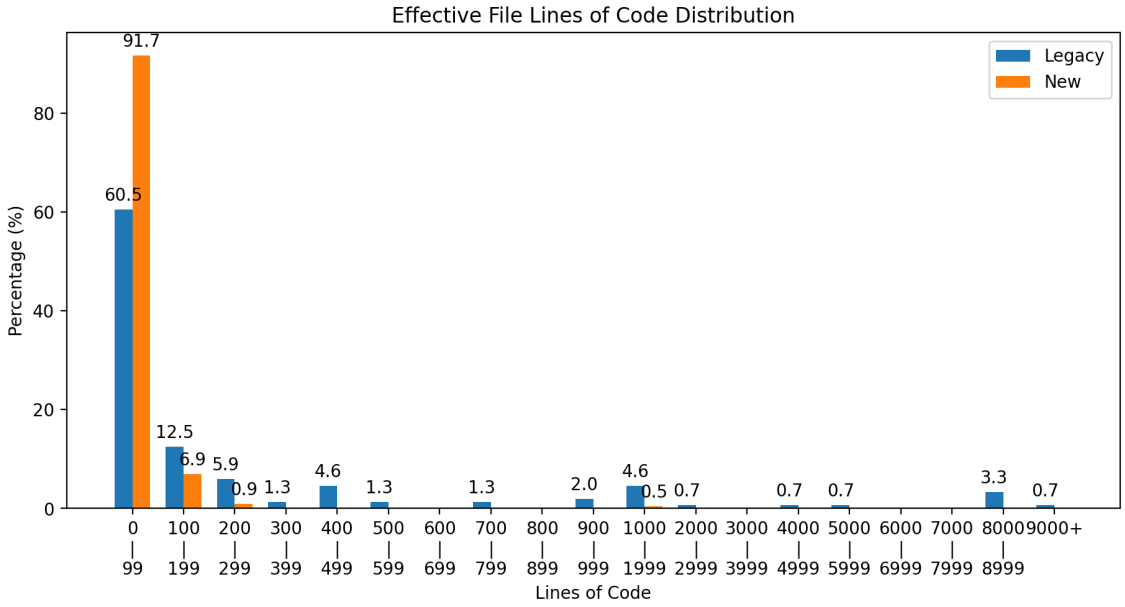
demonstrate that using the methodology allowed developers to achieve the same effective results with a much smaller codebase. Even though a LOC metric does not directly measure the required effort to develop a project, it can provide a rough estimate that NEWP demanded just about 10% of the work required for LEGP. Additionally, it is important to consider that the NEWP uses previously developed modules, which were also included in this codebase size.

Besides reducing the development work required, a smaller codebase also provides other benefits. Smaller project codebases can be easier to understand and maintain since less information needs to be processed. Also, they are less prone to bugs, as there are fewer LOC to find and contain bugs. Finally, the binary size is also reduced, which can be important for embedded systems with limited memory.

Since such a drastic reduction in codebase size was achieved, the number of comments and preprocessors is expected also to be reduced. Despite the reduction, for every 100 ELOC in LEGP, there are about 17 lines of comment, while in NEWP, there are approximately 91 lines of comment. This difference may indicate that NEWP has better code documentation and, therefore, is easier for other developers to reuse and maintain. Moreover, a higher density of preprocessors in NEWP may indicate fewer magic numbers, thus improving code readability and maintainability.

By evaluating the ELOCFI distribution in Figure 14, it is possible to observe that around 20% of LEGP files have more than 300 ELOC. Although this may not be a problem, large files may be more complex to maintain and understand. On the other hand, NEWP has a more balanced distribution, with most files having fewer than 100 ELOC. This distribution indicates that NEWP files are more focused and specialized, providing a better separation of concerns and better maintainability.

Figure 14 – Effective File Lines of Code Distribution for Case Study 2



Source: The Author

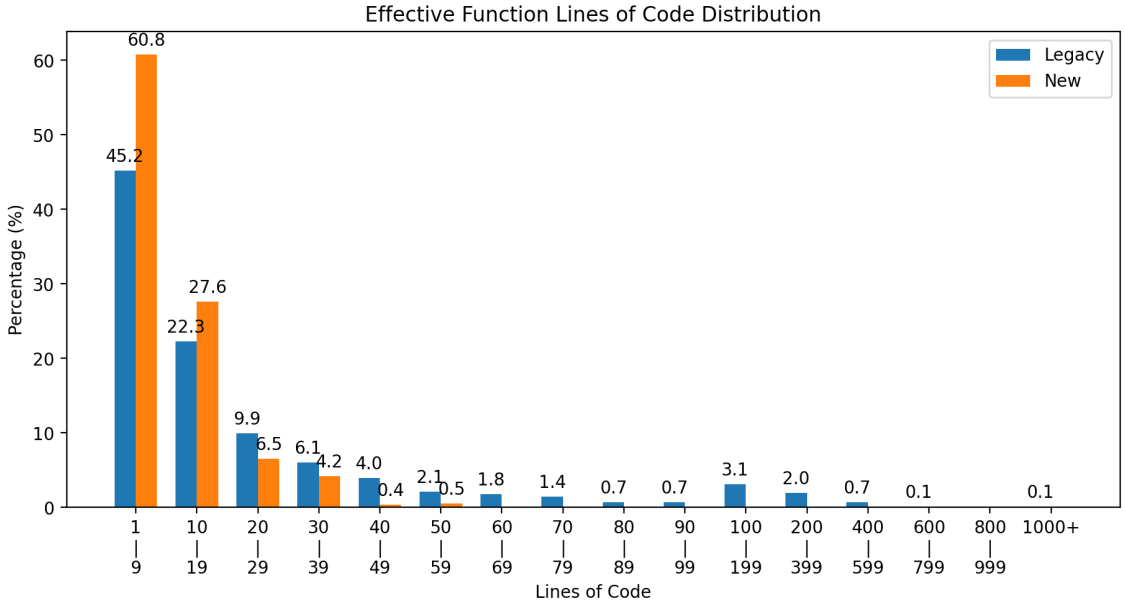
Going deeper into the LOC analysis, Table 26 presents the ELOCFU results, which show that NEWP functions are well distributed with a maximum of 59 ELOCFU. By reviewing the distribution in Figure 15, it is possible to observe that less than 1% of NEWP functions have more than the ideal 40 LOCFU. In comparison, LEGP contains functions going up to 1160 ELOCFU, with more than 10% of the functions falling into a range greater than NEWP.

With fewer functions and an almost ideal ELOCFU distribution, NEWP presents less complexity regarding readability and, thus, better maintainability. By being specialized and easy to understand, NEWP functions require less effort to introduce changes and have less chance to introduce bugs.

Table 26 – Effective Function Lines of Code results for Case Study 2

	Legacy	**New**	**Difference**	**Change**
Mean	31.2	10.2	-20.9	-67.1%
Standard Deviation	72.8	8.6	-64.3	-88.2%
Min	3	3	0	0.0%
Max	1160	59	-1101	-94.9%
Total	28269	5831	-22438	-79.4%

Figure 15 – Effective Function Lines of Code Distribution for Case Study 2



Source: The Author

5.2.4 Maximum Nesting Level

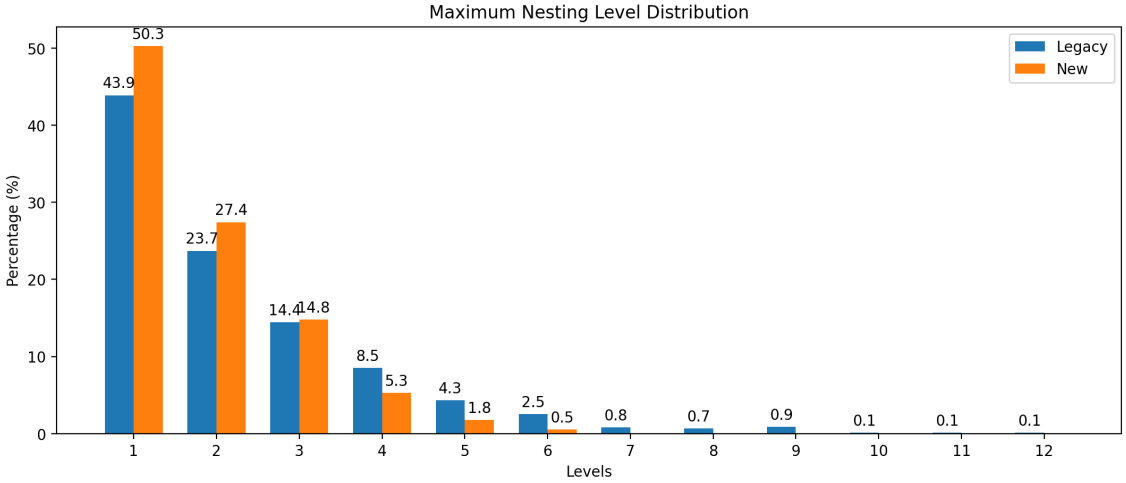
To evaluate complexity and readability aspects, Table 27 presents the MNL results. Unlike the other metric results, the MNL does not display drastic improvements for this metric. However, in contrast with LEGP, NEWP has reduced the average MNL by almost 20%, indicating that NEWP functions have reduced complexity and, therefore, increased maintainability.

Table 27 – Maximum Nesting Level results for Case Study 2

	Legacy	**New**	**Difference**	**Change**
Mean	2.3	1.8	-0.5	-19.8%
Standard Deviation	1.7	1.0	-0.6	-37.8%
Min	1	1	0	0.0%
Max	12	6	-6	-50.0%
Total	2064	1038	-1026	-49.7%

The MNL distribution (Figure 16) was analyzed to better understand why this metric did not present similar improvements to other metrics. This methodology recommends avoiding functions that exceed more than three levels of indentation. Therefore, LEGP has about 82% for that range while NEWP has about 92.5%. Additionally, LEGP presents functions with up to 12 levels, while NEWP functions have a maximum of 6 levels. Even though expressive results could not be observed in this metric, it is possible to notice that, compared with LEGP, NEWP displayed an overall improvement in complexity.

Figure 16 – Maximum Nesting Level Distribution for Case Study 2



Source: The Author

Going deeper into the analysis, unlike MVG, which measures the total of paths within a function, MNL only measures the maximum level of nesting on it. Although LEGP does not have expressive MNL values, its functions nested levels may vary widely. For example, according to Table 26, LEGP has functions with up to 1160 ELOC and, by combining it with high MNL values, indicates that LEGP functions have poor maintainability when compared to NEWP.

5.2.5 McCabe's Cyclomatic Complexity

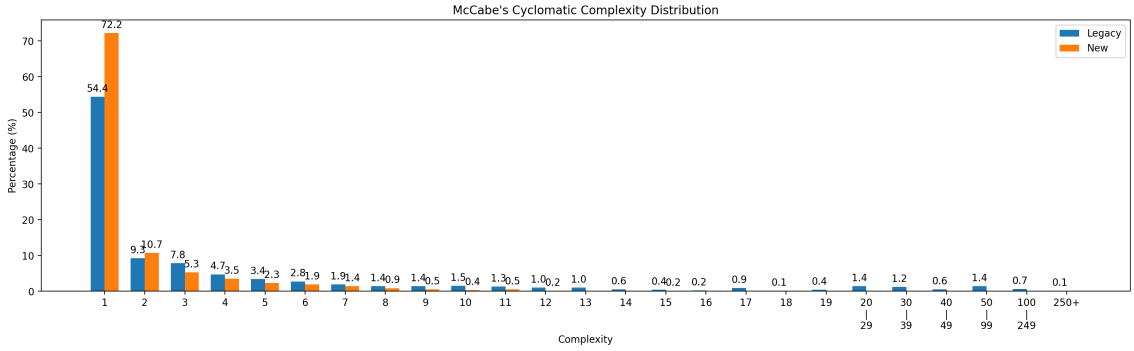
To evaluate the complexity of the projects, Table 28 presents the MVG results, indicating that NEWP has a drastic reduction of about 70% in complexity compared with LEGP. Furthermore, the standard deviation of MVG in NEWP has also been reduced by almost 90%, meaning that most of its functions exhibit a narrow range of complexity. This is illustrated in Figure 17, where about 72% of NEWP functions have an MVG value of 1, while in LEGP, only about 54%. This means that NEWP presents a larger set of small and single-pathed functions.

A large number of complexity 1 functions may indicate two opposite scenarios. The first is that functions are over-simplified or over-specialized, resulting in a poor project design. In the second scenario, functions are well-designed and focused, providing a

Table 28 – McCabe's Cyclomatic Complexity results for Case Study 2

	Legacy	**New**	**Difference**	**Change**
Mean	6.1	1.8	-4.3	-70.1%
Standard Deviation	17.5	1.9	-15.7	-89.4%
Min	1	1	0	0.0%
Max	310	15	-295	-95.2%
Total	5522	1037	-4485	-81.2%

Figure 17 – McCabe's Cyclomatic Complexity Distribution for Case Study 2



Source: The Author

better separation of concerns and readability.

Scenarios of the first type are usually observed in projects with a high number of functions and LOCFI, along with the inexistence of higher complexity functions. In such cases, complexity is distributed among them, and more LOC are required to encapsulate all those functions. Besides that, another possible characteristic of this scenario is the presence of functions with very high MVG values since they are needed to encapsulate a large number of smaller functions.

Since NEWP not only expressively reduces the number of functions and LOCFI compared to LEGP but also presents a good complexity range distribution, the first scenario is unlikely. Therefore, NEWP functions can be considered more consistent and well-designed to encapsulate the procedures while reducing complexity.

To support that, the system's flow with the maximum number of paths (Table 29) was calculated using the same methodology as CS1. In comparison with LEGP, the results show that NEWP has decreased the exponential level by about 74%. This means that NEWP has indeed improved the system's architecture and design, achieving the same results with a narrow number of path ramifications.

To assess the risk of the projects, Table 30 presents the MVG risk results. The data shows that 5.3% of LEGP functions have a high and very high risk of bugs, while NEWP has none. Furthermore, 6.1% of LEGP functions have medium risk, while NEWP has only 0.9%. Supporting the conclusion for CS1, these results also demonstrate the method-

Table 29 – Code Paths for Case Study 2

	Paths
Legacy	3e322
New	7e83

ology's benefits in reducing bugs and improving the quality and maintainability of the projects.

Table 30 – Risk of Bugs and Changes of Bug Injection results for Case Study 2

	Low (5%)	**Medium (20%)**	**High (40%)**	**Very High (60%)**	**Bug Injection Risk**
Legacy	804 (88.6%)	55 (6.1%)	29 (3.2%)	19 (2.1%)	8.2%
New	564 (99.1%)	5 (0.9%)	0 (0.0%)	0 (0.0%)	5.1%

In addition to these considerations, the MVG risk analysis provides good insight into how to improve the project. However, poorly architectured and designed projects may be too entangled to allow those types of improvement. Therefore, although this may be true for projects that follow the methodology, it may not always be true for those that do not.

By following the workflows provided by the methodology in Section 4.2, OPS can provide solutions to automate this metric and identify high and very high-risk functions in the review process. As a result, developers can focus on improving those functions, reducing the risk of bugs, and enhancing the project's overall quality.

5.2.6 Simple Manintaibility Index

Table 31 presents the SMI results, indicating substantial maintainability improvements in NEWP compared to LEGP. Even though NEWP does not reach the ideal SMI value for all functions, it only presents SMI values of up to 3, while LEGP reaches the maximum SMI of 25. Furthermore, approximately 97.4% of NEWP functions have ideal SMI, while LEGP has only about 84.2% (Figure 18).

It is important to note that any function with an SMI value above 1 is considered poorly designed. Therefore, with a mean value of 1.7 and a standard deviation of 2.4, LEGP is considered highly difficult to maintain. On the other hand, even though NEWP has room for improvement, it can be regarded as an easy-to-maintain project. Overall, NEWP improved the project's maintainability by more than 61% compared to LEGP.

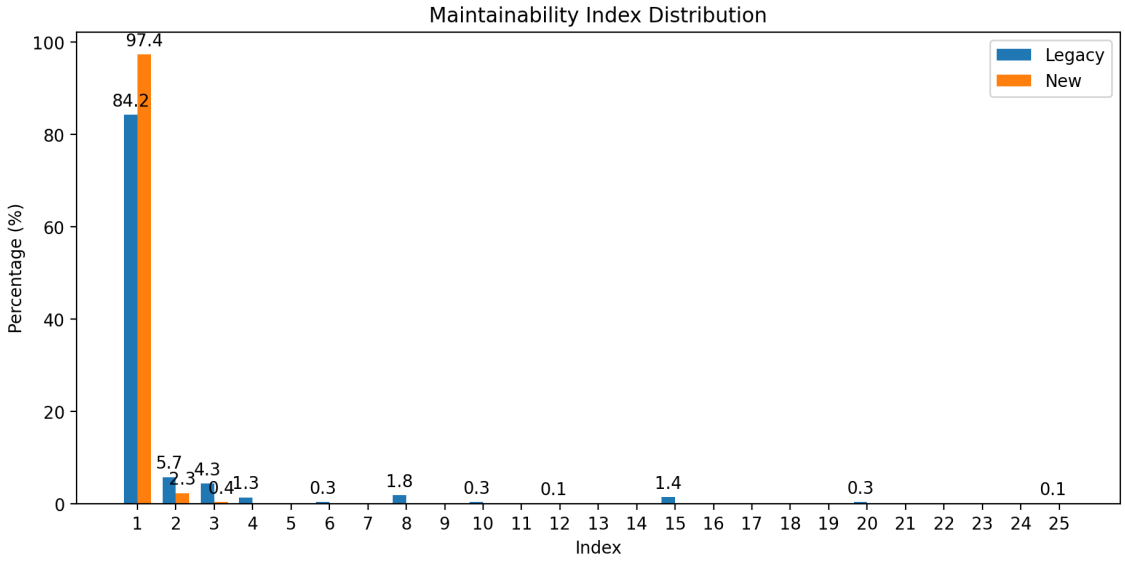
5.2.7 Module Coupling

Comparing the COUP metric for CS2 was not possible due to how LEGP was implemented. By analyzing the project, it was possible to identify that LEGP has a single

Table 31 – Maintainability Index results for Case Study 2

	Legacy	**New**	**Difference**	**Change**
Mean	1.7	1.0	-0.6	-37.8%
Standard Deviation	2.4	0.2	-2.2	-92.2%
Min	1	1	0	0.0%
Max	25	3	-22	-88.0%
Total	1501	586	-915	-61.0%
Overall Index	1501.0	581.4	-919.6	-61.3%

Figure 18 – Maintainability Index Distribution for Case Study 2



Source: The Author

global header that contains all project dependencies. This header is included in every file in the project, making it impossible to identify the actual dependencies of each module since every module depends on all others.

6 CONCLUSIONS

This work presents a development methodology for hardware-independent firmware development. By establishing a set of rules and architecture, this methodology provides numerous benefits besides facilitating project portability across different hardware platforms. By providing a common ground of system comprehension to all team members, it enables them to easily get up-to-speed with new projects or even different modules that they may not have previously participated in. This structure also allows novice developers to improve their development skills and start contributing to the project quickly without limiting the more experienced developers.

By enabling module reuse, the methodology also improves the efficiency of developing new projects or features since developers can leverage previously developed modules. Additionally, modularization facilitates the introduction of new features, corrections, or improvements without affecting the rest of the system. This is a significant advantage when working with large codebases, where changes can have unexpected side effects. Furthermore, automated testing becomes more straightforward, as modules can be tested independently.

In this work, many important aspects of the development process were addressed and defined to ensure a simple, yet effective, development process. Firstly, the team roles were presented and defined. This section outlines the responsibilities and requirements of each role to ensure that the team members understand their roles and the importance of their work.

Next, the team's development workflow was introduced. This workflow distinguishes between development and maintenance tasks to ensure that these activities do not interfere with each other. This separation is essential to ensure that the team can focus on their current task without interruptions from other activities.

Finally, the methodology's architecture was presented. The architecture defines a structure where system and hardware implementations are separated into different domains that only communicate through a well-defined process. Concepts such as portability domain, conventions, standards and patterns were introduced. Next, accompanied by the robot example system, layers and their modules were described to provide a clear

understanding of where each module fits within the system, their interactions and dependencies, and how they should be implemented.

To evaluate the proposed work, the development methodology was demonstrated through the implementation of the first case study. For this purpose, a simple IoT system developed from previous work was refactored according to the methodology. Although the system was simple, the refactoring process was able to demonstrate the benefits of the methodology. The refactored system was more organized, modular, and easier to understand and maintain.

The second case study was a real-world project of a much larger scale and complexity. Besides the benefits previously observed in the first case study, the comparison between new and legacy versions of this system, provided more realistic insights into the benefits of the methodology. Finally, these results demonstrated that the use of the methodology can significantly improve the development process and the quality of the final product.

In conclusion, the methodology presented in this work provides a solid foundation for hardware-independent and modular firmware development. By providing a set of rules, conventions, and architecture, the methodology enables the development of more organized, modular, and maintainable systems. The benefits of the methodology were demonstrated through two case studies, showing that the methodology can significantly improve the development process and the quality of the final product. The methodology is a valuable tool for firmware development teams that want to improve their development process and the quality of their products.

Some possible future work directions that can be followed based on the results obtained in this work are:

- The development of a set of tools to automate the generation of the methodology's structure, modules, and configurations;
- The development and application of tools to automate the testing of the system;
- The development of tools to automate the enforcement and review of the methodology's rules, conventions, and structure;
- The design and implementation of a package management system to facilitate the reuse of modules.

REFERENCES

ALMOGAHED, A. *et al.* A Refactoring Classification Framework for Efficient Software Maintenance. **IEEE Access**, [S.l.], v. 11, p. 78904–78917, 2023.

AMAZON WEB SERVICES. **FreeRTOS**. Avaliable in: https://www.freertos.org/FAQ_Amazon.html#why_has_amazon_taken_stewardship_of_freertos. Accessed in: 01-03-2024.

AMINI, K. **Extreme C**: taking you to the limit in concurrency, oop, and the most advanced capabilities of c. [S.l.]: Packt Publishing Ltd, 2019.

ANDERSON, P. Coding standards for high-confidence embedded systems. *In*: MILCOM 2008-2008 IEEE MILITARY COMMUNICATIONS CONFERENCE, 2008. **Proceedings [...]** [S.l.: s.n.], 2008. p. 1–7.

ASPENCORE. **The Current State of Embedded Development**. Avaliable in: https://www.embedded.com/wp-content/uploads/2023/05/Embedded-Market-Study-For-Webinar-Recording-April-2023.pdf. Accessed in: 23-02-2024.

BANIK, S.; ZIMMER, V. **Firmware Development**. [S.l.]: Apress, 2022.

BARKMANN, H.; LINCKE, R.; LÖWE, W. Quantitative evaluation of software quality metrics in open-source projects. *In*: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION NETWORKING AND APPLICATIONS WORKSHOPS, 2009. **Proceedings [...]** [S.l.: s.n.], 2009. p. 1067–1072.

BARR, M. **Embedded C Coding Standard**. [S.l.]: Barr Group, 2018.

BARR, M.; MASSA, A. **Programming embedded systems**: with c and gnu development tools. [S.l.]: " O'Reilly Media, Inc.", 2006.

BEN SGHAIER, O.; SAHRAOUI, H. Improving the learning of code review successive tasks with cross-task knowledge distillation. **Proceedings of the ACM on Software Engineering**, [S.l.], v. 1, n. FSE, p. 1086–1106, 2024.

BENINGO, J. **Reusable Firmware Development**: a practical approach to apis, hals and drivers. [S.l.]: Springer, 2017.

BENINGO, J. **Embedded Software Design**: a practical approach to architecture, processes, and coding techniques. [S.l.]: Springer, 2022.

BODEN, A.; NETT, B.; WULF, V. Operational and strategic learning in global software development. **IEEE Software**, [S.l.], v. 27, 2010.

BROOKS, F. P. Essence and accidents of software engineering. **IEEE computer**, [S.l.], v. 20, n. 4, p. 10–19, 1987.

CLEMENTS, A. A. *et al.* {HALucinator}: firmware re-hosting through abstraction layer emulation. *In*: USENIX SECURITY SYMPOSIUM, 29., 2020. **Proceedings [...]** [S.l.: s.n.], 2020. p. 1201–1218.

CLEMENTS, A. A. *et al.* Is your firmware real or re-hosted?. *In*: WORKSHOP ON BINARY ANALYSIS RESEARCH, 2021. **Proceedings [...]** [S.l.: s.n.], 2021.

CONNECTIVITY STANDARDS ALLIANCE. **Matter**: the foundation for connected things. Avaliable in: https://csa-iot.org/all-solutions/matter/. Accessed in: 22-04-2024.

COOLING, J. **Software Engineering for Real-Time Systems (The Complete Edition)**. [S.l.]: Packt Publishing Ltd, 2019.

CROSSLEY, C. **Software Supply Chain Security**: securing the end-to-end supply chain for software, firmware, and hardware. 1. ed. [S.l.]: O'Reilly Media, 2024.

DANO, E. B. Importance of Reuse and Modularity in System Architecture. *In*: INTERNATIONAL SYMPOSIUM ON SYSTEMS ENGINEERING, 2019. **Proceedings [...]** [S.l.: s.n.], 2019. p. 1–8.

DE BASSI, P. R. *et al.* Measuring developers' contribution in source code using quality metrics. *In*: IEEE 22ND INTERNATIONAL CONFERENCE ON COMPUTER SUPPORTED COOPERATIVE WORK IN DESIGN, 2018. **Proceedings [...]** [S.l.: s.n.], 2018. p. 39–44.

DEEPA, M. *et al.* Leveraging Agile Framework for a Project Based Learning Environment in Embedded Systems Design Course. **Journal of Engineering Education Transformations**, [S.l.], v. 37, 2024.

DOUGLASS, B. P. **Design patterns for embedded systems in C**: an embedded software engineering toolkit. [S.l.]: Elsevier, 2010.

DUNN, J. COVID-19 and Supply Chains: a year of evolving disruption. **Cleveland Fed District Data Briefs**, [S.l.], n. cfddb 20210226, p. 1–8, 2021.

ECKER, W.; MüLLER, W.; DöMER, R. **Hardware-dependent software**: principles and practice. [S.l.: s.n.], 2009.

ECOS. **The eCos Hardware Abstraction Layer (HAL)**. Avaliable in: http://www.ecos.sourceware.org/docs-1.3.1/ref/ecos-ref.b.html. Accessed in: 12-01-2024.

ELDH, S. Code Review Evolution. **IEEE Software**, [S.l.], v. 41, n. 5, p. 4–8, 2024.

ESPRESSIF SYSTEMS. **ESP-WROOM-32 Datasheet**. Avaliable in: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf. Accessed in: 17-01-2024.

FAHMIDEH, M. *et al.* Software Engineering for Internet of Things: the practitioners' perspective. **IEEE Transactions on Software Engineering**, [S.l.], v. 48, n. 8, p. 2857–2878, 2022.

FAN, X. **Real-Time Embedded Systems**: design principles and engineering practices. [S.l.: s.n.], 2015.

FARINA, M. D.; DOS ANJOS, J. C.; DE FREITAS, E. P. Real-Time Auto Calibration for Heterogeneous Wireless Sensor Networks. **Journal of Internet Services and Applications**, [S.l.], v. 14, n. 1, p. 1–9, 2023.

FARINA, M. D. *et al.* Hardware-Independent Embedded Firmware Architecture Framework. **Journal of Internet Services and Applications**, [S.l.], v. 15, n. 1, p. 14–24, 2024.

FENG, B.; MERA, A.; LU, L. {P2IM}: scalable and hardware-independent firmware testing via automatic peripheral interface modeling. *In*: USENIX SECURITY SYMPOSIUM, 29., 2020. **Proceedings [...]** [S.l.: s.n.], 2020. p. 1237–1254.

FENTON, N.; BIEMAN, J. **Software metrics**: a rigorous and practical approach. [S.l.]: CRC press, 2014.

FERREIRA, L. C. B. *et al.* The three-phase methodology for iot project development. **Internet of Things**, [S.l.], v. 20, p. 100624, 2022.

FOROUZANI, S.; CHIAM, Y. K.; FOROUZANI, S. Method for assessing software quality using source code analysis. *In*: FIFTH INTERNATIONAL CONFERENCE ON NETWORK, COMMUNICATION AND COMPUTING, 2016. **Proceedings [...]** [S.l.: s.n.], 2016. p. 166–170.

FOWLER, M. **Patterns of Enterprise Application Architecture**: pattern enterpr applica arch. [S.l.]: Addison-Wesley, 2012.

FRÖHLICH, A. A.; WANNER, L. F. Operating system support for wireless sensor networks. **Journal of Computer Science**, [S.l.], v. 4, n. 4, p. 272, 2008.

GANSSLE, J. G. A firmware development standard. **Embedded Systems Programming**, [S.l.], 2004.

GARCÍA TUDELA, P. A.; MARÍN MARÍN, J. A. Use of Arduino in Primary Education: a systematic review. **Education Sciences**, [S.l.], v. 13, n. 2, p. 134, 2023.

GOMES, E. *et al.* A survey from real-time to near real-time applications in fog computing environments. *In*: TELECOM, 2021. **Proceedings [...]** [S.l.: s.n.], 2021. v. 2, n. 4, p. 489–517.

GRACIOLI, G. *et al.* Implementation and evaluation of global and partitioned scheduling in a real-time OS. **Real-Time Systems**, [S.l.], v. 49, p. 669–714, 2013.

GRADY BOOCH ROBERT A. MAKSIMCHUK, M. W. E. **Object-oriented analysis and design with applications**. 3rd ed. ed. [S.l.]: Addison-Wesley, 2007. (The Addison-Wesley object technology series).

GRAMS, C. How Much Time Do Developers Spend Actually Writing Code. **The New Stack**, [S.l.], 2019.

GUERRERO-ULLOA, G.; RODRÍGUEZ-DOMÍNGUEZ, C.; HORNOS, M. J. Agile methodologies applied to the development of Internet of Things (IoT)-based systems: a review. **Sensors**, [S.l.], v. 23, n. 2, p. 790, 2023.

GUSTAFSON, E. *et al.* Toward the analysis of embedded firmware through automated re-hosting. *In*: INTERNATIONAL SYMPOSIUM ON RESEARCH IN ATTACKS, INTRUSIONS AND DEFENSES, 22., 2019. **Proceedings [...]** [S.l.: s.n.], 2019. p. 135–150.

HÄNISCH, T. A Case Study on using Microservice Patterns in an Embedded System. *In*: ATINER'S CONFERENCE PAPER PROCEEDINGS SERIES, 2023. **Proceedings [...]** [S.l.: s.n.], 2023.

HOBBS, C. **Embedded Software Development for Safety-Critical Systems**. [S.l.]: CRC Press, 2019.

HOLZMANN, G. J. The Power of 10: rules for developing safety-critical code. **Computer**, [S.l.], v. 39, 2006.

HOMÈS, B. **Fundamentals of software testing**. [S.l.]: John Wiley & Sons, 2024.

HORSTMANNM, L. P. *et al.* Handling WSN Communication Faults at the Edge with Confidence Attribution for Data Imputation. *In*: IEEE 9TH WORLD FORUM ON INTERNET OF THINGS, 2023. **Proceedings [...]** [S.l.: s.n.], 2023. p. 1–6.

HUBALOVSKY, S.; SEDIVY, J. Mistakes in object oriented programming. *In*: INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY, 2., 2010. **Proceedings [...]** [S.l.: s.n.], 2010. p. 113–116.

IFTIKHAR, U. *et al.* A tertiary study on links between source code metrics and external quality attributes. **Information and Software Technology**, [S.l.], p. 107348, 2023.

INTERVALZERO. **Understanding Hard Real-time Determinism**. Avaliable in: https://www.intervalzero.com/understanding-hard-real-time-determinism/. Accessed in: 04-05-2024.

JAMIL, M. A. *et al.* Software testing techniques: a literature review. *In*: INTERNATIONAL CONFERENCE ON INFORMATION AND COMMUNICATION TECHNOLOGY FOR THE MUSLIM WORLD, 6., 2017. **Proceedings [...]** [S.l.: s.n.], 2017.

JIA, M. *et al.* Transient computing for energy harvesting systems: a survey. **Journal of Systems Architecture**, [S.l.], v. 132, p. 102743, 2022.

JOHNSON, J. *et al.* An empirical study assessing source code readability in comprehension. *In*: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, 2019. **Proceedings [...]** [S.l.: s.n.], 2019. p. 513–523.

KALAYCI, M. **Fortifying Automotive Cybersecurity**: the imperative of compliance to misra c and cert c. Avaliable in: https://medium.com/@mkklyci/fortifying-automotive-cybersecurity-the-imperative-of-compliance-to-misra-c-and-cert-c-5b6f968f9c04. Accessed in: 21-04-2024.

KAUR, H. *et al.* Optimizing for Happiness and Productivity: modeling opportune moments for transitions and breaks at work. *In*: HUMAN FACTORS IN COMPUTING SYSTEMS, 2020. **Conference [...]** [S.l.: s.n.], 2020.

KOPETZ, H.; STEINER, W. **Real-time systems**: design principles for distributed embedded applications. [S.l.]: Springer Nature, 2022.

KRÖNING, M. W. **Concurrency Techniques and Hardware Abstraction Layer Concepts for Embedded Systems in Rust**. 2023. Tese (Doutorado em Engenharia Elétrica) — Universitätsbibliothek der RWTH Aachen, 2023.

LEE, Y. *et al.* Embedded Firmware Rehosting System through Automatic Peripheral Modeling. **IEEE Access**, [S.l.], 2023.

LOUBSER, N. **Software Engineering for Absolute Beginners**. [S.l.]: Springer, 2021.

LWAKATARE, L. E. *et al.* Towards DevOps in the embedded systems domain: why is it so hard? *In*: ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 2016. **Proceedings [...]** [S.l.: s.n.], 2016. v. 2016-March.

M. GOMES, R.; BAUNACH, M. A Study on the Portability of IoT Operating Systems. **Tagungsband des FG-BS Frühjahrstreffens 2021**, [S.l.], 2021.

MAKHSHARI, A.; MESBAH, A. IoT bugs and development challenges. *In*: IEEE/ACM 43RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2021. **Proceedings [...]** [S.l.: s.n.], 2021. p. 460–472.

MARCONDES, H. *et al.* Operating Systems Portability: 8 bits and beyond. *In*: IEEE CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION, 2006. **Proceedings [...]** [S.l.: s.n.], 2006. p. 124–130.

MARTIN, R. **Clean Architecture**: a craftsman's guide to software structure and design. [S.l.]: Pearson Education, 2017.

METRIX++. **Extendable Tool for Code Metrics Collection and Analysis**. Avaliable in: https://github.com/metrixplusplus/metrixplusplus. Accessed in: 23-01-2024.

MICROSOFT. **Microsoft acquires Express Logic, accelerating IoT development for billions of devices at scale**. Avaliable in: https://blogs.microsoft.com/blog/2019/04/18/microsoft-acquires-express-logic-accelerating-iot-development-for-billions-of-devices-at-scale/. Accessed in: 10-01-2024.

MILICCHIO, F. The Unix KISS: a case study. **arXiv preprint cs/0701021**, [S.l.], 2007.

MISRA. **MISRA C 2012**: guidelines for the use of the c language in critical systems. [S.l.]: Motor Industry Software Reliability Association, 2012.

MOTOGNA, S.; VESCAN, A.; ŞERBAN, C. Empirical investigation in embedded systems: quality attributes in general, maintainability in particular. **Journal of Systems and Software**, [S.l.], v. 201, p. 111678, 2023.

MUDE, R.; JOSHI, R. D. Modular Design Implementation in the Firmware of Complex Micro-Controller Based Products. *In*: ANNUAL INTERNATIONAL CONFERENCE ON EMERGING RESEARCH AREAS: INTERNATIONAL CONFERENCE ON INTELLIGENT SYSTEMS, 2023. **Proceedings [...]** [S.l.: s.n.], 2023. p. 1–6.

NESER, M.; VAN SCHOOR, G. Object-oriented embedded C. **SAIEE Africa Research Journal**, [S.l.], v. 100, n. 4, p. 90–96, 2009.

OUALLINE, S. **Bare Metal C**: embedded programming for the real world. [S.l.]: No Starch Press, 2022.

PARGAONKAR, S. Enhancing Software Quality in Architecture Design: a survey-based approach. **International Journal of Scientific and Research Publications (IJSRP)**, [S.l.], v. 13, n. 08, 2023.

PASSIG HORSTMANN, L.; CONRADI HOFFMANN, J. L.; FRÖHLICH, A. A. Monitoring the performance of multicore embedded systems without disrupting its timing requirements. **Design Automation for Embedded Systems**, [S.l.], v. 27, n. 4, p. 217–239, 2023.

POSCH, M. **Hands-On Embedded Programming with C++ 17**: create versatile and robust embedded solutions for mcus and rtoses with modern c++. [S.l.]: Packt Publishing Ltd, 2019.

QUANTUM LEAPS LLC. **Object-Oriented Programming in C**. Avaliable in: https://github.com/QuantumLeaps/OOP-in-C. Accessed in: 22-02-2024.

RENAUX, D. P. Comparative performance evaluation of CMSIS-RTOS. *In*: EUROPEAN SIGNAL PROCESSING CONFERENCE, 2014. **Proceedings [...]** [S.l.: s.n.], 2014. v. 1998-January.

ROGER, S. P.; BRUCE, R. M. **Software engineering**: a practitioner's approach. [S.l.]: McGraw-Hill Education, 2019.

SCHMIDT, M. **Implementing the IEEE software engineering standards**. [S.l.]: Sams, 2000.

SEACORD, R. C. **SEI CERT C Coding Standard**. [S.l.]: Carnegie Mellon University, 2016.

SHIN, K. G.; RAMANATHAN, P. Real-time computing: a new discipline of computer science and engineering. **Proceedings of the IEEE**, [S.l.], v. 82, n. 1, p. 6–24, 1994.

SIMMANN, G.; VEERANNA, V.; KRIESTEN, R. **Design of an Alternative Hardware Abstraction Layer for Embedded Systems with Time-Controlled Hardware Access**. [S.l.]: SAE Technical Paper, 2024.

SOLOVEV, A.; YULDASHEV, T. Embedded System Design: challenges of hardware and software development. **Integra Sources**, [S.l.], 2023.

SOMMERVILLE, I. Software engineering. 10th. **Book Software Engineering. 10th, Series Software Engineering**, [S.l.], v. 10, 2016.

SPRAY, J.; SINHA, R. Abstraction layered architecture: writing maintainable embedded code. *In*: EUROPEAN CONFERENCE ON SOFTWARE ARCHITECTURE, 2018. **Proceedings [...]** [S.l.: s.n.], 2018. p. 131–146.

STANKOVIC, J. A. Real-time and embedded systems. **ACM Computing Surveys (CSUR)**, [S.l.], v. 28, n. 1, p. 205–208, 1996.

STEWART, D. B. Twenty-five most common mistakes with real-time software development. *In*: EMBEDDED SYSTEMS CONFERENCE, 1999., 1999. **Proceedings [...]** [S.l.: s.n.], 1999. v. 141.

SUN, L.; LI, Y.; MEMON, R. A. An open IoT framework based on microservices architecture. **China Communications**, [S.l.], v. 14, n. 2, p. 154–162, 2017.

THIRUMALAI, G. K. **A Beginner's Guide to SSD Firmware**: designing, optimizing, and maintaining ssd firmware. [S.l.]: Springer, 2023.

TOTH, J.; KARLSSON, F. **Selecting unit testing framework for embedded microcontroller development**. 2021. Dissertação (Mestrado em Engenharia Elétrica) — Jonkoping Univeristy, 2021.

TREMAROLI, N. J. **Adaptive Firmware Framework for Microcontroller Development**. 2023. Tese (Doutorado em Engenharia Elétrica) — Virginia Tech, 2023.

TYRKKÖ, M. *et al.* **Organizing Software Maintenance in a Small Aviation Software Company**: a case study. 2019. Dissertação (Mestrado em Engenharia Elétrica) — , 2019.

WANG, J. **Real-Time Embedded Systems**. [S.l.]: John Wiley & Sons, Inc., 2017.

WANG, J. *et al.* Software testing with large language models: survey, landscape, and vision. **IEEE Transactions on Software Engineering**, [S.l.], 2024.

WANG, K. C. **Embedded and real-time operating systems**. [S.l.: s.n.], 2017.

WHITE, E. **Making embedded systems**. [S.l.]: " O'Reilly Media, Inc.", 2024.

WILLENBRING, J. M.; WALIA, G. S. Using Complexity Metrics with Hotspot Analysis to Support Software Sustainability. *In*: IEEE INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING WORKSHOPS, 2022. **Proceedings [...]** [S.l.: s.n.], 2022. p. 37–42.

WILLENBRING, J. M.; WALIA, G. S. The utility of complexity metrics during code reviews for CSE software projects. **Future Generation Computer Systems**, [S.l.], v. 160, p. 65–75, 2024.

WILLENBRING, J.; SINGH WALIA, G. Evaluating the Sustainability of Computational Science and Engineering Software: empirical observations. *In*: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, 2021. **Proceedings [...]** [S.l.: s.n.], 2021. p. 453–456.

WILLOCX, M. *et al.* Developing maintainable application-centric iot ecosystems. *In*: IEEE INTERNATIONAL CONGRESS ON INTERNET OF THINGS, 2018. **Proceedings [...]** [S.l.: s.n.], 2018. p. 25–32.

YUAN, C. *et al.* A Component Development Framework for Embedded Software. *In*: IEEE INTERNATIONAL CONFERENCE ON INFORMATION COMMUNICATION AND SOFTWARE ENGINEERING, 2021. **Proceedings [...]** [S.l.: s.n.], 2021. p. 71–75.

ZADDACH, J. *et al.* AVATAR: a framework to support dynamic security analysis of embedded systems' firmwares. **NDSS**, [S.l.], v. 14, p. 1–16, 2014.

ZHENG, X.; LIANG, S.; XIONG, X. A hardware/software partitioning method based on graph convolution network. **Design Automation for Embedded Systems**, [S.l.], v. 25, 2021.

APPENDIX A DEVELOPMENT STANDARD

A.1 Standard Rules

1. Non-compliance modules should have a note on each file header comment.
2. In case of a code standard revision or update, legacy files should only be updated to the new version when code changes are required. In this case, the whole module should be updated to the new standard version.
3. Always check if third-party licenses allow their use in closed, commercial and proprietary code.
4. Any code, comment, documentation and filenames should be written in proper English language.
5. All programs should be written to comply with the C99 version of the ISO C Programming Language Standard.

A.2 Formatting and Indentation Rules

1. Braces should always surround the blocks of code, following if, $else$, $switch$, $while$, do, and for statements. Single statements and empty statements following these keywords should also always be surrounded by braces.
2. Each indentation level should align at a multiple of 4 characters from the start of the line.
3. The tab character (ASCII $0x09$) should never appear within any source code file. Use 4 spaces instead (Automate your code editor!).
4. There is no characters per line limit, however, whenever possible try to stay below 100 characters.
5. Whenever a line of code is too long to fit within the maximum line width, indent the second and any subsequent lines in the most readable manner possible.

6. All source code lines should end only with the single character LF (ASCII $0x0A$), not with the pair $CR - LF$ ($0x0D$ $0x0A$).

7. Left Braces ({) should always be on the same line of its statement:

```
// Do This:
if(x > 0) {
    // Some Code..
}
// Not This:
if(x > 0)
{
    // Some Code..
}
```

8. Within a $switch$ statement, the case labels should be aligned; the contents of each case block should be indented once from there.

A.2.1 White Space Rules

1. The left brace ({) on each if, $while$, for and $switch$ should be separated by one space:

```
if(x > 0) {
    while(1) {
    }
}
```

2. Each of the assignment operators $=$, $+=$, $-=$, $*=$, $/=$, $\%=$, $\&=$, $|=$, $^=$, $=$, and $!=$ should always be preceded and followed by one space.

```
x += 1;
```

3. Each of the binary operators $+$, $-$, $*$, $/$, $\%$, $<$, $<=$, $>$, $>=$, $==$,$!=$, $<<$, $>>$, $\&$, $|$, $\wedge$, $\&\&$, and $||$ should always be preceded and followed by one space.

```
if(x != 0)
```

4. Each of the unary operators $+$, $-$, $++$, $--$, !, and $\sim$, should be written without a space on the operand side.

```
x++;
```

5. The pointer operators $*$ and $\&$ should be written without a space on the operand side.

```
uint32_t *ptr = &var:
```

6. The structure pointer and structure member operators ($->$ and ., respectively) should always be without surrounding spaces.

```
x = the_pointer->x;
y = the_struct.y;
```

7. The left and right brackets of the array subscript operator ([and]) should be without surrounding spaces, except as required by another white space rule.

```
uint32_t x[10];
```

8. Expressions within parentheses should always have no spaces adjacent to the left and right parenthesis characters.

```
x = (a + b);
```

9. The left and right parentheses of the function call operator should always be without surrounding spaces. Except when at the end of a line, each comma separating function parameters should always be followed by one space.

```
void foo(uint32_t a, uint32_t b);
```

10. Each semicolon separating the elements of a for statement should always be followed by one space.

```
for(i = 0 ; i < 10 ; ++i) {
}
```

11. Each semicolon should follow the statement it terminates without a preceding space.

```
uint32_t x = 1;  // Good
uint32_t x = 1 ; // Bad
```

A.3 Comments and Code Documentation Rules

1. Each Module and function should be documented in Doxygen comment format.

   (a) The Module Main Header file should contain at least the following information:

   - $@file$ Header Filename
   - $@author$ Code Author Name and Email
   - $@brief$ Module Description

```
/**
* @file app_template.h
* @author Author Name (Developer Email)
* @brief Application Template
*/
```

   (b) Function prototypes should contain at least the following information:

   - $@brief$ Function description

- $@param$ Function argument description
- $[in]$ and $[out]$ tags should be used for input and output arguments respectively
- $@return$ Function return value

```
/**
* @brief Copy String with Offset
*
* @param str_0[in] Input String
* @param str_1[out] Output String
* @param offset[in] Input String Offset
*
* @return bool true Success
* @return bool false Failed
*/
bool copy_string(const char *str_0, char *str_1, size_t offset) {

    bool error = true;

    size_t in_length = strlen(str_0);

    if (in_length < offset) {
        strcpy(str_1, &str_0[offset]);
        error = false;
    }

    return error;
}
```

(c) Global Variables and Definitions should contain an inline brief description:

```
#define STRING_LENGTH 10 /** String Length */
const uint32_t str[STRING_LENGTH] = "Hello World"; /** Hello World String */
```

(d) Structures and Enumerators should contain a header $@brief$ description and inline item descriptions:

```
/**
* @brief Foo Struct
*/
typedef struct {
    uint32_t item_0; /** Item 0 Description */ uint32_t item_1; /** Item 1
    Description */
    } foo_struct_t;

/**
* @brief Foo Enum
*/
typedef enum {
        ITEM_0, /** Item 0 Description */ ITEM_1, /** Item 1 Description */
    } foo_enum_t;
```

(e) Macros should contain at least the following information:

- $@brief$ Macro description

- $@param$ Macro argument description
- $[in]$ and $[out]$ tags should be used for input and output arguments respectively

```
/**
* @brief Copy String with Offset
*
* @param result[out] Sum Result
* @param a[in] value
* @param b[in] value
*/
#define SUM_TWO_NUMBERS(result, a, b) { \
    result = a + b;                     \
    }
```

(f) Always document all measurement units:

```
#define TIMEOUT 100 /** Timeout Time (ms) */

/**
* @brief Cartesian Coordinate
*/
typedef struct {
    uint32_t x; /** X Axis (cm) */
    uint32_t y; /** Y Axis (cm) */
} coordinate_t;
```

2. Do not use the traditional C style comments ($/*...*/$), use C++ style instead ($//...$)

3. Inline comments should always describe a single line of code. Comments on top of a block of code should describe the block behavior.

```
// Create Buffer
uint32_t *buffer = NULL;        // Declare Buffer Pointer
size_t buffer_size = 10;        // Declare Buffer Size
buffer = malloc(buffer_size);   // Allocate Buffer on Heap
```

4. Comments should never contain the preprocessor tokens $/*$, $//$, or
.

5. Committed code should never be commented out, even temporarily. Delete it before committing or use the preprocessor's conditional compilation feature instead. For example:

```
// Some Code...
#ifdef DEBUG
printf("Debug Log\n");
#endif
// Some Code..
```

6. Whenever an algorithm or technical detail is defined in an external reference (e.g., a design specification, patent, or textbook), a comment should include a sufficient reference to the original source to allow a reader of the code to locate the document.

7. All assumptions should be spelled out in comments.
8. All comments should be written in clear and complete sentences, with proper spelling and grammar and appropriate punctuation.
9. Avoid explaining the obvious. Assume the reader knows the C programming language.
10. Use the following capitalized comment markers to highlight important issues:
    - $WARNING$: alerts a maintainer there is risk in changing this code.
    - $NOTE$: provides descriptive comments about the "why" of a chunk of code.
    - $TODO$: indicates an area of the code is still under construction and explains what remains to be done.
    - $FIXME$: alerts about a non-fixed problem.
    - $FUTURE$: informs something to be done in the future.
11. Each project module should provide a $readme.md$ file (written in Markdown) containing the complete module's description and documentation.
12. All external documentation files should be placed inside a $docs$ directory inside the respective module directory.

A.4 General Rules

1. **(NFC)** Do not use $auto$, $register$, $restrict$ and $continue$ keywords.
2. **(NFC)** C Standard Library functions $abort()$, $exit()$, $setjmp()$, and $longjmp()$ should not be used.
3. Do not use ternary expressions ($(a > b)?10 : 20$).
4. No line of code should contain more than one statement.
5. Unintended behavior should be protected with $assert()$.
6. Avoid repeating blocks of code (copy and paste code). Code blocks should never be repeated more than once (create a new function instead).
7. Jumps ($goto$) should always respect code flow:

```
// Good
bool good(void) {

    bool result = false;
```

```
    if(do_something_0()) {
        goto end;
    }

    if(do_something_1()) {
        goto end;
    }

    result = true;

end:
    return result;
}

// Bad
void bad(void) {

loop_back:

    uint32_t value = do_something();

    if(value > 10){
        goto loop_back;
    }
}
```

A.5 Naming Rules

A.5.1 Files

1. All file names should consist entirely of lowercase letters, numbers, and underscores. No spaces should appear within the header and source file names.

2. No header file name should share the name of a header file from the C Standard Library or C++ Standard Library. For example, modules should not be named $stdio.h$ or $math.h$.

3. Any module containing a $main()$ function should have the word $main$ as part of its source file name.

4. Whenever possible, name module files as follows:

A.5.2 Data Types

1. The names of all new data types, including structures, unions, and enumerations should consist only of lowercase characters and internal underscores and end with $\_t$.

2. All new structures, unions, and enumerations should be named via a $typedef$.

3. The name of all public data types should be prefixed with their module name and an underscore.

Filename	Use
name.c	Main Source
name.h	Public Header
name_private.h	Private Header
name_types.h	Type Declarations
name_internal.h	Restricted Header
name_override.c	Class Override Methods Source Code
name_override.h	Class Override Methods Header

4. Enumerates items should always be uppercase characters and internal underscores and end with ,.

```
typedef enum {
    ITEM_0,
    ITEM_1,
    ITEM_2, // Do not forget the "," on this one!
} foo_t;
```

A.5.3 Variables

1. No variable should have a name that is a keyword of C, C++, or any other well-known extension of the C programming language, including specifically K&R C and C99. Restricted names include *interrupt*, *inline*, *restrict*, *class*, *true*, *false*, *public*, *private*, *friend*, and *protected*.

2. No variable should have a name that overlaps with a variable name from the C Standard Library (e.g., *errno*).

3. No variable should have a name that begins with an underscore.

4. No variable name should be shorter than 3 characters (Unless extremely meaningful).

```
// Bad
uint32_t e = 0; // Where e stands for event
uint32_t ch = 0; // Where ch stands for channel

// Acceptable
// Where x and y stand for cartesian coordinates
uint32_t x = 0;
uint32_t y = 0;

// Acceptable
for(uint32_t i = 0 ; i < 10 ; ++i) {}
```

5. No variable name should contain any uppercase letters.

6. No variable name should contain any numeric value that is called out elsewhere, such as the number of elements in an array.

```
// Bad
uint8_t buffer_10[10];
```

7. Underscores should be used to separate words in variable names.

8. Each variable's name should be descriptive of its purpose.

9. Variable names should never be prefixed with their data types.

10. The names of all Boolean variables or Integers containing Boolean information should be phrased as the question they answer. For example, $done\_yet$ or is_buffer_full.

A.5.4 Functions, Macros and ISR

1. All functions that implement ISRs or are ISR-Safe should be given names ending with $\_isr$.

2. All functions that encapsulate threads of execution (a.k.a., tasks, processes) should be given names ending with $\_task$ (or $\_thread$, $\_process$).

3. No procedure should have a name that is a keyword of any standard version of the C or C++ programming language. Restricted names include $interrupt$, $inline$, $class$, $true$, $false$, $public$, $private$, $friend$, $protected$, and many others.

4. No procedure should have a name that overlaps a function in the C Standard Library. Examples of such names include $strlen$, $atoi$, and $memset$.

5. No procedure should have a name that begins with an underscore or numbers.

6. No function name should contain any uppercase letters.

7. No macro name should contain any lowercase letters.

8. Underscores should be used to separate words in procedure names.

9. Each procedure's name should be descriptive of its purpose. Note that procedures encapsulate the "actions" of a program and thus benefit from the use of verbs in their names (e.g., $adc\_read()$); this "noun-verb" word ordering is recommended. Alternatively, procedures may be named according to the question they answer (e.g., $is\_led\_on()$).

10. The names of all public functions should be prefixed with their module name and an underscore (e.g., $module\_name\_read()$).

Postfix	Use	Access
$.c$	Main Source	Private
$.h$	Public Header	Public
$\_private.h$	Private Header	Private
$\_types.h$	Type Declarations	Public
$\_internal.h$	Restricted Header	Restricted
$\_override.c$	Class Override Methods Source Code	Restricted
$\_override.h$	Class Override Methods Header	Restricted

A.6 Module Rules

1. Module files should respect the following access level:
2. Each header file should contain a preprocessor guard against multiple inclusion:

```
#ifndef __FILENAME_H__
#define __FILENAME_H__
...
#endif
```

3. The header file should identify only the procedures, constants, and data types (via prototypes or macros, $\#define$, and $typedef$) about which it is strictly necessary for other modules to be informed.
   - Do not declare variables on header files (Headers containing $const$ variable data only are exceptions)
   - Do not $extern$ variables
   - No storage for any variable should be allocated in a header file
   - Do not include any executable lines of code in a header file (Macros and $inline$ Functions are exceptions)
4. No public header file should contain a $\#include$ of any private/restricted header file.
5. Each source file should $\#include$ only the behaviors appropriate to control one entity. Examples of entities include encapsulated data types, active objects, peripheral drivers (e.g., for a UART), and communication protocols or layers (e.g., ARP).
6. Each source file should always $\#include$ the header file of the same name (e.g., file $adc.c$ should $\#include$ "$adc.h$").
7. Absolute paths should not be used in $\#include$ file names.
8. Each source file should be free of unused $\#include$ files.
9. No source file should $\#include$ another source file.

A.7 Preprocessors Rules

1. Preprocessor directive $\#define$ should not be used to alter or rename any keyword or other aspect of the programming language. For example:

```
// Do not do this...
#define begin {
#define end }

for (int row = 0; row < MAX_ROWS; row++)
begin
// Loop Code...
end
```

2. **(NFC)** Parameterized macros should not be used if a function can be written to accomplish the same behavior. Use $inline$ functions instead.

3. If parameterized macros are used for some reason, these rules apply:

   - Surround the entire macro body with parentheses.
   - Surround each use of a parameter with parentheses.
   - Never include a transfer of control (e.g., $return$ keyword).

```
#define MAX(A, B) ((A) > (B) ? (A) : (B)) // You still should not use ternary. It
    was used here just for simplicity.
```

A.8 Variable Rules

1. **(NFC)** All variables should be initialized before use. This will avoid issues such as:

```
// Will return unknown/trash value if (a <= b)
uint32_t foo(uint32_t a, uint32_t b) {

    uint32_t value;

    if (a > b) {
        value = (a + b);
    }

    return value;
}
```

2. **(NFC)** It is always preferable to initialize variables on declaration.

3. It is preferable to define local variables as you need them, rather than all at the top of a function.

4. If global variables are used, their definitions should be grouped together and placed at the top of a source code file.

5. **(NFC)** Any pointer variable lacking an initial address should be initialized to $NULL$.
6. The comma operator (,) should not be used within variable declarations.
7. The $static$ keyword should be used to declare all global variables that do not need to be visible outside the file where were declared.
8. The $const$ keyword should be used whenever appropriate. Examples include:
   (a) To declare variables that should not be changed after initialization;
   (b) To define call-by-reference function parameters that should not be modified (e.g., $constchar * param$);
   (c) To define fields in a $struct$ or $union$ that should not be modified (e.g., in a $struct$ overlay for memory-mapped I/O peripheral registers);
   (d) As a strongly typed alternative to $\#define$ for numerical constants.
9. The $volatile$ keyword should be used whenever appropriate. Examples include:
   (a) To declare a global variable accessible (by current use or scope) by any interrupt service routine;
   (b) To declare a global variable accessible (by current use or scope) by two or more threads;
   (c) To declare a pointer to a memory-mapped I/O peripheral register set (e.g., $constvolatiletimer\_t * timer$);
   (d) To declare a delay loop counter/spin-lock.
10. The order: $static$, $volatile$, $const$ and data-type, should be respected.
11. Each unsafe cast should feature an associated comment describing how the code ensures proper behavior across the range of possible values on the right side.

```
uint32_t byte_value = 128;
uint8_t byte = (uint8_t)byte_value; // Limits value range to 8 bit values
```

A.8.1 Structures, Unions, Enumerates

1. **(NFC)** Appropriate care should be taken to prevent the compiler from inserting padding bytes within $struct$ or $union$ types. Special attention should be given to these types when used to (but not restricted to):
   - communicate with peripherals
   - communicate over a bus or network
   - communicate with another processor

- casting data arrays to a given type pointer

In such cases, unused spaces should be explicitly declared by the developer.

```
// For a 32 bit processor, this struct will consume 12 Bytes of memory.
typedef struct {
    uint8_t  b;
    uint16_t a;
    uint32_t c;
} foo_t;

// Whenever relevant, padding should be avoided as in the example below
typedef struct {
    uint8_t  b;
    uint8_t not_used_0[3];
    uint32_t c;
    uint16_t a;
    uint8_t not_used_1[2];
} foo_t;
```

Also, compiler modifiers should not be used to solve this problem.

2. Appropriate care should be taken to prevent the compiler from altering the intended order of the bits within bit-fields (even if unused, declare variables that sum the total number of bits of the used data type).

3. Bit-fields should not be defined within $signed$ integer types.

A.8.2 Fixed-Width Integers

1. Do not use $char$, $short$, int, $long$, or $longlong$ for integer values. Use the $stdint.h$ data types $int8\_t$, $uint8\_t$, $int16\_t$, $uint16\_t$, $int32\_t$, $uint32\_t$, $int64\_t$ or $uint64\_t$ instead.

2. The $char$ type should be restricted to the declaration of an operations concerning strings.

3. None of the bitwise operators (i.e., $\&$, $|$, $\sim$, $\wedge$, $<<$, and $>>$) should be used to manipulate $signed$ integer data.

4. $signed$ integers should not be combined with $unsigned$ integers in comparisons or expressions.

5. Always prefer using data types with the size of your processor architecture.

6. Whenever a third-party library expects the data types $char$, $short$, int, $long$, or $longlong$ as fixed-width integers, all developed code should comply with $stdint.h$ data types and, on library function call, be explicitly cast to the respective data type. Variable checks should be made to ensure that variables are equivalent (e.g., use $sizeof(int)$).

A.8.3 Booleans

1. Boolean variables should be declared as type $bool$ defined in $stdbool.h$.

A.8.4 Floating Point

1. **(NFC)** Avoid the use of floating point constants and variables whenever possible. Fixed-point math may be an alternative.

2. When floating point calculations are necessary:

   - Do not use the $float$ and $double$ types. Use instead the C99 type names $float32\_t$, $float64\_t$, and $float128\_t$.
   - Append an f to all single-precision constants (e.g., $pi = 3.141592f$).
   - Ensure that the compiler supports double precision if your math depends on it.
   - Never test for equality ($==$) or inequality ($!=$) of floating point values.
   - Always invoke the $isfinite()$ macro to check that prior calculations have resulted in neither $INFINITY$ nor NaN.

A.9 Operations Rules

1. **(NFC)** Do not rely on C's operator precedence rules, use parentheses to ensure proper execution order within a sequence of operations. Statements with only addition and subtraction operations are an exception.

```
x = a/b/c; // Bad
x = (a/b)/c; // Good

x = a*b + c; // Bad x = (a*b) + c; // Good

x = a + b + c - d; // This is acceptable
```

2. Unless it is a single identifier or constant, each operand of the logical AND (&&) and logical OR (||) operators should be surrounded by parentheses.

```
// Do this:
if ((x >= 0) && (x <= 100)) {
    // Some Code..
}

// Do not do this:
if (x >= 0 && x <= 100) {
    // Some Code..
}
```

3. **(NFC)** Every non-constant division should be protected to avoid zero divisions.

```
// Unsafe
uint32_t foo(uint32_t a, uint32_t b) {
    return (a / b);
}

// Safe
uint32_t foo(uint32_t a, uint32_t b) {
    assert(b != 0);
    return (a / b);
}

// Safe
uint32_t foo(uint32_t a, uint32_t b) {

    uint32_t value = 0;

    if(b != 0) {
        value = (a / b);
    }

    return value;
}
```

A.10 Statement Rules

1. Nested $if/else$ statements should not be deeper than two levels. Use function calls or $switch$ statements to reduce complexity and aid understanding.

2. Assignments should not be made within an if or $elseif$ test.

3. **(NFC)** When evaluating the equality of a variable against a constant, the constant should always be placed to the left of the equal-to operator (==). This avoids problems where a missing = may still result in a valid operation. For Example:

```
uint32_t x = 10;

// The developer missed an = in this comparison
// However, the operation is still valid but the
// comparison will be incorrect.
if(x = 100) {
    printf("Success\n");
}else {
    printf("Fail\n");
}
```

For that reason, always prefer:

```
uint32_t x = 10;

if(100 == x) {
    printf("Success\n");
}else {
    printf("Fail\n");
}
```

If the developer misses an $=$, the expression will become invalid and the compiler will accuse an error.

4. All $switch$ statements should contain a $default$ block.

5. Any $case$ designed to fall through to the next should be commented to clearly explain the absence of the corresponding $break$.

6. Magic numbers should not be used as the initial value or in the endpoint test of a $while$, $do...while$, or for loop.

7. Except for the initialization of a loop counter in the first clause of a for statement and the change to the same variable in the third, no assignment should be made in any loop's controlling expression.

8. Infinite loops should be implemented via controlling expression $for(;;)$.

9. **(NFC)** Each loop with an empty body should feature a set of braces enclosing a comment to explain why nothing needs to be done until after the loop terminates.

10. Decremental to 0 for loops are usually more efficient than Incremental. Also, prefix operations ($--$ and $++$) are more efficient than postfix.

```
for(uint32_t i = 0 ; i < 100 ; i++);     // Slowest
for(uint32_t i = 0 ; i < 100 ; ++i);     // Slow
for(uint32_t i = 100 ; i > 0 ; i--);     // Fast
for(uint32_t i = 100 ; i > 0 ; --i);     // Faster
for(uint32_t i = 100 ; i != 0 ; --i);    // Fastest
```

A.11 Function Rules

1. The $static$ keyword should be used to declare all functions that do not need to be visible outside the file where they were declared.

2. **(NFC)** Unless Single-Lined, the returning value should always be declared as the function's first line of code. Also, every function should only contain a single exit point ($return$) at the bottom of the function.

```
uint32_t foo(void) {

    uint32_t value = 0; // Always first line of code

    if (b != 0) {
        value = (a / b);
    }

    return value; // Always last line of code
}
```

This will avoid issues where returning values may be undefined or not present. For example:

```
// Will Never return if (a == b)
uint32_t foo(uint32_t a, uint32_t b) {

    if (a < b) {
        return 10;
    }else if (a > b) {
        return 20;
    }

}

// Will Never return if SOME_TAG is not defined
uint32_t foo(uint32_t a, uint32_t b) {
#ifdef SOME_TAG
    return (a + b);
#endif
}
```

3. **(NFC)** It is a preferred practice that all returning variables be initiated with fail/$false$/0 values. This allows functions to only result in success after conditions are met. For example:

```
bool foo(uint32_t a, uint32_t b) {

    bool result = false;

    if (a < 10) {
        goto end;
    }

    if (b > 200) {
        goto end;
    }

    if (a != b) {
        result = (a < b); // Will only process the result here
    }

end:
    return result;
}
```

4. A prototype should be declared for each public function in their respective header file.

5. A prototype should be declared for each private function in their respective private header file.

6. Whenever possible, all private functions should be declared $static$.

7. Each parameter should be explicitly declared and meaningfully named.

8. Always prefer passing $struct$ variables by reference.

9. Functions that $return\ float$ or $double$ types should always have a prototype (The compiler will consider an integer $return$ type otherwise).

10. To ensure that ISRs are not inadvertently called from other parts of the software (they may corrupt the CPU and call stack if this happens), each ISR function should be declared $static$.

11. A stub or default ISR should be installed in the vector table at the location of all unexpected or otherwise unhandled interrupt sources. Each such stub could attempt to disable future interrupts of the same type, say at the interrupt controller, and $assert$.