Journal of Internet Services and Applications, 2024, 15:1, doi: 10.5753/jisa.2024.3634

Hardware-Independent Embedded Firmware Architecture Framework

Mauricio D.O. Farina [**Federal University of Rio Grande do Sul - PPGEE|**
email: ***mauriciofarina@icloud.com***]
Daniel H. Pohren [**Federal University of Rio Grande do Sul - PPGEE|**
email: ***daniel.pohren@ufrgs.br***]
Alexandre dos S. Roque [**Federal University of Rio Grande do Sul - PPGEE|**
email: ***ale.roque@gmail.com***]
Antonio Silva [**Federal University of Rio Grande do Sul - PPGEE|**
email: ***aassilva@inf.ufrgs.br***]
Joao Paulo J. da Costa [**Hamm-Lippstadt University of Applied Sciences - PK NRW|**
email: ***JoaoPaulo.daCosta@hshl.de***]
Lisandra Manzoni Fontoura [**Federal University of Santa Maria - PPGCC|**
email: ***lisandra@inf.ufsm.br***]
Julio C. S. dos Anjos [**Federal University of Ceara Campus Itapajé - PPGETI|**
email: ***jcsanjos@ufc.br***]
Edison Pignaton de Freitas [**Federal University of Rio Grande do Sul - PPGEE|**
email: ***edison.pignaton@ufrgs.br***]

Federal University of Rio Grande do Sul - UFRGS - PPGEE - Av. Osvaldo Aranha, 103 - Bom Fim, Porto Alegre, RS , 90035-190, Brazil

Received: 18 August 2023 • **Accepted:** 19 February 2024 • **Published:** 16 April 2024

Abstract Unlike other forms of development, the way firmware development is designed is somewhat outdated. It is not unusual 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 problem for Internet of Things (IoT) applications that evolve very frequently. To address this problem, this study proposes an embedded firmware development framework 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. An embedded IoT system project was refactored for this framework model to validate this proposal. Finally, a comparison was made between a legacy and framework project to demonstrate that the proposed framework can make a substantial improvement in portability, reuse, modularity, and other firmware factors.

Keywords: Embedded Systems, Firmware Architecture, Development Guidelines

1 Introduction

Even though many embedded devices are designed to last for a short time, they constantly face unexpected situations that may reduce their lifespan. For example, unforeseen circumstances may require devices to adopt extra routines that result in energy overheads and thus shorten their battery lives. 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.

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 *et al.*, 2023]. For example, dunn2021covid discusses the impacts of the COVID-19 pandemic on global supply chains and shows how the shortage of microchips has forced companies to redo consolidated projects to make use of alternative components available on the market.

Internet of Things (IoT) devices are widely used on emerging applications due to their ease of implementation, their flexibility, and their proximity to the data source [Anjos *et al.*, 2021]. However, the heterogeneity of hardware devices provided by various vendors requires continuous firmware updates to overcome software flaws and security issues or to provide functionality improvements. A concrete example of this demand is the growing use of the Arduino platform, where several new components are added daily, making compatibility between different hardware a reality. It is known that this platform brings a chronic problem: the need for compatibility between new and legacy equipment. New platforms are being created to correct these limitations, but they are returning to the initial model of being dedicated to certain families of Microcontroller Unit (MCU) and Microprocessor Unit (MPU).

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 simple example is when development teams refuse to use an available Real-Time Operational System (RTOS) but prefer to develop their own in-house scheduler instead [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 is becoming less reusable and portable. This type of undocumented change is common on open-source platforms, where each programmer changes features at their own pleasure.

Firmware maintenance usually takes up a large stage of a product life cycle, and automated testing may significantly reduce costs. Conducting automated tests for embedded systems is a considerable challenge and, in the case of disorganized firmware, an unsustainable task. Finally, even well-architected firmware may not be reusable or portable if novice software developers make unsupervised changes to the production code.

Frameworks make it possible to abstract low-level details, and thus allow 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 the development of frameworks 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 [Gamma *et al.*, 1995], and defining a robust, scalable, and reusable architecture [Fowler, 2012]. In addition, frameworks that enable reuse can generally dispense with the most tedious tasks related to MCU development [Tremaroli, 2023].

According to Sommerville [2015], a framework is a generic structure designed to create a specific application or subsystem. Schmidt et al. define a framework as an integrated set of software artifacts (such as classes, objects, and components) that collaborate to create a reusable architecture for a family of related applications [Schmidt *et al.*, 2004]. Thus, by forming a pre-defined structure, frameworks can accelerate software development by allowing developers to focus on its specific features. In addition, frameworks can improve software quality by reusing components that have already been implemented, tested, and checked.

Embedded software, particularly for IoT, still requires improvements in the development process and methodology [Fahmideh *et al.*, 2022]. The current world scenario requires a more flexible and robust firmware development process for embedded software, particularly IoT, like other more mature software areas. This paper attempts to fill this gap by introducing an embedded firmware development framework that addresses problems such as the lack of code maintainability, reuse, portability, and the absence of development standardization. In the 1980s, Brooks and Bullet [1987] declared *there is no silver bullet for software development* [Brooks and Bullet, 1987], which is still applicable today.

The embedded world covers an extensive universe that requires different development approaches. The proposed framework aims at a fraction of MCU based systems where the firmware's 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. Finally, systems based on application processors or sophisticated System-on-Modules (SoM) that possess advanced features, for example, Graphics processing unit (GPU), Non-uniform Memory Access (NUMA), or multiple heterogeneous core architectures, are not covered by this approach. For the growth of the IoT universe to be solid, the ease and reliability of the resources used are essential, making it necessary to be able to reuse functionalities, services, and interfaces in an easy and friendly way.

The main research contributions of this study are as follows and seek to:

- Provide an embedded firmware development framework that allows reuse and portability;
- Improve firmware development life cycle phases;
- Reduce the risk of mistakes being made by novice developers while not limiting or restricting the scope of advanced developers; and
- Establish development standards and lay down guidelines for embedded firmware development.

The rest of this article is structured as follows. Section 2 reviews the concepts of reuse, portability, maintainability, and rehosting. Section 3 describes the proposed hardware-independent embedded firmware architecture framework, and Section 4 sets out a practical demonstration of the proposed framework. Finally, Section 5 summarizes the main findings of this work.

2 Related Works

Several works in the literature are seeking to address aspects of firmware design so as 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]. Also, Yuan *et al.* [2021] introduce 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 [Yuan *et al.*, 2021].

Modern programming languages use Object Oriented Programming (OOP) to allow modularity and decoupling. However, the C language does not natively support it. Neser and Shoor address this question by introducing a framework to simulate OOP features in C language called Object Oriented C (OOC) Neser and van Schoor [2009]. The method does not allow a full OOP implementation, but most core features can be imported in a similar way. Quantum Leaps also proposes a similar OOC framework that does not make use of preprocessors for emulating OOP features [Quantum Leaps, LLC, 2020].

Concerning portability, several studies have sought to employ methods to facilitate the implementation of portable firmware. For example, Marcondes *et al.* [2006] adopt an application-oriented and component-based operating system that includes code portability between MCUs with different architectural sizes [Marcondes *et al.*, 2006]. In contrast, Sun *et al.* [2017]. provide a microservices-based framework for IoT where service and physical layers communicate through a common message broker medium [Sun *et al.*, 2017]. At the same time, many legacy codes may be required so that they can be ported in the future. In light of this, Martins and Baunach analyse what the current portable IoT operational systems are like and the quality of the currently available ports [Martins Gomes and Baunach, 2021].

Motogna *et al.* [2023] study the quality attributes that are prioritized in embedded systems and identifies the best practices and activities they involve [Motogna *et al.*, 2023]. Their research revealed that the most important quality attributes in embedded systems development are maintainability, safety, and security, along with performance and energy efficiency. Since maintainability is often mentioned in the literature as an essential feature of a reusable and portable code, several studies offer solutions for it. For this reason, Spray and Sinha integrate the knowledge of software architecture with the experience in designing embedded software from the Tru-Test Group to a) create an abstraction layered architecture and b) create code bases with improved long-term maintainability [Spray and Sinha, 2018]. In addition, Willocx et al. establish a layered IoT architecture to support the development of complex and maintainable IoT applications [Willocx *et al.*, 2018]. By abstracting low-level implementation details, developers can focus on business logic without having to be experts in IoT sensor technology.

Rehosting techniques allow firmware testing without a 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 final aim is to port an existing code to a different host. In light of this, Zaddach et al. recommend Avatar, which is a hardware-in-the-loop design for an event-based arbitration framework that orchestrates the communication between an emulator and a targeted physical device [Zaddach *et al.*, 2014].

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 *et al.* [2020], find 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 [Feng *et al.*, 2020]. Clements *et al.* [2020] adopt another approach, which entails introducing a HALucinator. This high-level emulation method offers simple, analyst-created replacements that carry out the same task from the standpoint of the firmware [Clements *et al.*, 2020]. Afterward, the authors expand HALucinator by adding a re-hosting support layer, which significantly reduces the porting time for devices using VxWorks [Clements *et al.*, 2021].

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 take to find solutions. Hubalovsky and Sedivy [2010] examine the most common OOP mistakes made by both beginners and experienced programmers [Hubalovsky and Sedivy, 2010]. Stewart also analyses the most common mistakes made in Real-Time (RT) software development [Stewart, 1999].

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. Table 1 and Table 2 make a comparison to illustrate the differences between this article and the key related works on reuse and portability.

Table 1. Comparative summary of related works on reusable design

Paper	Modularization	Object Oriented	Hardware Independent	Full Firmware Architecture
Dano [2019]	No	No	No	No
Yuan *et al.* [2021]	Yes	No	Possible	No
Neser and van Schoor [2009]	Yes	Yes	Possible	No
Quantum Leaps, LLC [2020]	Yes	Yes	Possible	No
This Paper	**Yes**	**Yes**	**Yes**	**Yes**

Table 2. Comparative summary of related works on portable design

Paper	Hardware Abstraction	Hardware Decoupling	Firmware Refactoring	Rehosting
Marcondes *et al.* [2006]	Yes	No	No	No
Sun *et al.* [2017]	Yes	No	No	No
Martins Gomes and Baunach [2021]	No	No	Yes	No
Zaddach *et al.* [2014]	No	Yes	No	Yes
Clements *et al.* [2020]	Yes	Yes	No	Yes
Clements *et al.* [2021]	Yes	Yes	No	Yes
This Paper	**Yes**	**Yes**	**Possible**	**Possible**

3 Framework

This work proposes an embedded C development framework that covers modularization, reuse, portability, and standardization to address the gaps in the literature in the context of embedded software development, particularly applied to IoT. The framework organizes the development process into multiple areas and layers to mitigate aspects such as hardware coupling, monolithic development, complex maintainability, and lack of development rules.

Initially, the framework is separated into two main regions (Figure 1), the MCU Project (PROJECT) (the red area in the diagram) is responsible for all the hardware-specific implementations. In contrast, Core Library (CORE) (the blue area) contains all the system implementations in a generic form. The main purpose of this structure is to ensure there is a complete separation between the hardware and system, which can enable CORE to be reused in virtually any other MCU that supports the features required by its specifications.

As well as portability, the framework follows organizational procedures to improve the way the firmware code

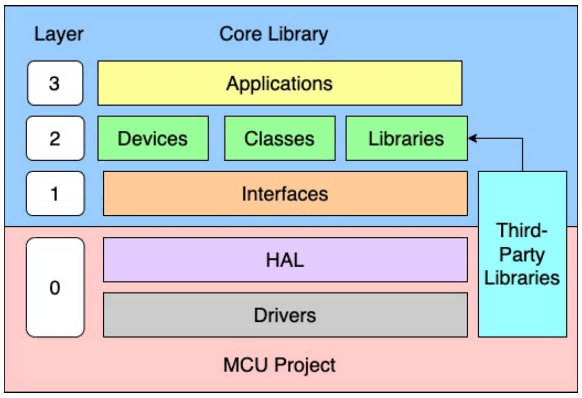

Figure 1. Hierarchy of the Framework

is maintained. In addition, novice developers are often included in projects with little (or even no) supervision by other experienced developers. As a result, these developers may cause problems to the system or violate the design choices of the system architecture. The proposed framework addresses this matter by standardizing and ensuring the accuracy of the development process. It also makes definitions called Novice Framework Choice (NFC), which are only designed to guide these developers and prevent them from making their most common mistakes.

3.1 MCU Project

Embedded projects are usually implemented following how a particular MCU and its peripherals work, which may cause some problems regarding portability and aspects of reuse. Several embedded applications are constrained by limited available MCU resources, which makes this strategy the most efficient. For example, the cooperative development of the hardware and software components can be used to achieve the best performance of an embedded systemZheng *et al.* [2021]. However, a considerable number of modern embedded projects are not restricted by these limits, which means their implementations can devote more resources to other benefits.

In view of this, PROJECT includes a region where all of the MCU-specific implementations can be freely developed without having to interact with any system code. Moreover, developers will given guidance on how to implement standardized functions provided by CORE, which will act as a medium between the two framework regions. The interface region will be discussed in Section 3.3.

3.1.1 MCU Project Specifications (SPECS)

The PROJECT region of framework was designed to support different MCUs. For this reason, developers may need to replace the current PROJECT repository with a different version created for another target MCU. When creating this new version, the system architect must define the required domain where his CORE should exist. Since the PROJECT region is replaceable, this information must be carried out by CORE and should be the starting point for every new PROJECT.

The SPECS documentation must include all the information regarding the portability domain that is foreseen by the system architect. In our experience, the most common domain specifications are as follows:

- MCU Specifications
  - Bit depth (8-bits, 32-bits, etc.)
  - Architecture (ARM, RISC-V, AVR, PIC, etc.)
  - Endianness
  - Minimum memory size
  - Minimum operating frequency
  - Required peripherals
- Bare-Metal or RTOS
- Third-Party library support
- Compiler
- Peripheral configurations
- Driver design path (blocking, Interrupt request (IRQ), Direct Memory Access (DMA), etc.)

However, these requirements are not limited to this list. They are essential and must coexist in every project. Instead, the system architect can define its domain as it sees fit. In this process, the architect may find it necessary to reduce the complexity of development by restricting the domain too much or defining a broad portable domain that may result in a considerable increase in this kind of complexity. Thus, there is a need to balance these two factors to provide a reasonable domain that is realistic for the product under development.

3.1.2 Layer 0 - Drivers and HAL Layer

As well as dealing with portability factors, another goal of the framework is to assist novice developers in improving the firmware under development. The first NFC in this area is to allow developers to choose between developing drivers and Hardware Abstraction Layer (HAL) implementations by starting from scratch or using either the code generators or abstraction frameworks provided by the vendors. In this way, novice programmers can benefit from those tools and manage without a deep understanding of more complex driver implementations. In contrast, experienced programmers can maintain their freedom to design more sophisticated code. As a result, Drivers and HAL Layers may get mixed into single or multiple layers. For this reason, this framework can assume all these implementations are a part of the same layer and thus allow developers more flexibility.

Since Layer 0 is expected to contain hardware-specific implementations, no other layer need directly access or reference the hardware. As a means of providing a medium between PROJECT and CORE, a set of interface functions is operated by Layer 1 (Section 3.3). These functions must be implemented in Layer 0 so that abstracted access can be given to the MCU resources, and ensure CORE is entirely decoupled from the hardware. The framework only establishes a single rule for this layer: Layer 0 can only include headers from Layer 1. In addition, developers can structure the implementation in a way that best fits their needs.

3.2 Core Library

The CORE region was designed to have its hardware entirely decoupled and, thus be, portable to any MCU in the specification domain. In addition, this also allows CORE to be run by the developers' computer or continuous development applications without any hardware. As a result, there can be a reduction in the development time since time-consuming processes, such as programming the MCU, can be avoided. Firmware developers can also rely on tools that were previously only used by software developers. Finally, the whole system can be tested through unit testing without any hardware emulation.

3.3 Layer 1 - Interfaces

The role of interfaces is to provide the necessary prototype functions that may be requiredDouglass [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 the CORE modules do not require hardware-specific knowledge.

In addition accessing peripherals and functionalities, all of the hardware-specific information should be provided to CORE by middleware functions as well. 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, so that it is made independent. Finally, Interfaces should only be displayed in the form of header files.

3.4 Layer 2 - Components

The components layer is designed to allow code modularization and reuse. The primary purpose of components is to separate codes and break down their implementation into small, easily understood units. Components can be divided into two types:

External Components: these are implementations that can be reused in multiple projects. They can be seen as independent modules that can be imported into the project. These components must be self-contained and separate from other project-specific components. In addition, they must have their software versioning of the system, and only released versions should be used by CORE. The decoupling system enables a code to be tested, debugged, and reused more simply since implementations are not interdependent and contain a good deal of unrelated content.

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

As well as types, components are separated into three categories , each of which is responsible for a different modularization and, is thus designed to meet other requirements.

3.4.1 Devices

Devices are components representing Integrated Circuit (IC) peripherals that from a part of the product. These components must be separate from the project so that an IC devices library can be formed. The device implementation process is also expected to be bare-metal and allow multiple instances so that the use domain can be expanded.

```c
#ifndef __LED_DRIVE_H__
#define __LED_DRIVE_H__
#include <stdint.h>

/** @brief Device Pins */
typedef enum {
    LED_DRIVE_PIN_0 = 0, /** Pin 0 */
    LED_DRIVE_PIN_1,     /** Pin 1 */
    LED_DRIVE_PIN_2,     /** Pin 2 */
    LED_DRIVE_MAX_PIN,   /** Max Number of Pins
    */
} led_drive_pin_t;

/** @brief Device Handler */
typedef struct {
    led_drive_pin_t pin; /** Pin */
    uint32_t frequency;  /** Frequency (Hz)*/
    uint8_t duty_cycle;  /** Duty Cycle (%) */
    uint8_t address;     /** Device I2C Address
    */
    void (*i2c_write)(uint8_t address, uint8_t *
    data, uint32_t size); /** I2C Data Write
    Function */
} led_drive_t;

/** @brief Init Device
 * @param handler[in/out] Device Handler
 */
void dev_led_drive_init(led_drive_t *const
    handler);

/** @brief Set Pin Duty Cycle
 * @param handler[in] Device Handler
 * @param duty_cycle[in] Duty Cycle (%)
 */
void dev_led_drive_set_duty_cycle(led_drive_t *
    const handler, uint8_t duty_cycle);

/** @brief Set Pin Frequency
 * @param handler[in] Device Handler
 * @param frequency[in] Frequency (Hz)
 */
void dev_led_drive_set_frequency(led_drive_t *
    const handler, uint32_t frequency);
#endif
```

Listing 1: Device Header

Every device should include a type of handler variable containing all the information regarding a single instance. This handler is expected to be passed on as the first function argument to access device functionalities, and thus dispenses with the need for instance-specific implementations inside the functions. Code listing 1 provides a simple example of this structure.

3.4.2 Classes

One advantage of modern programming languages is their built-in support for OOP. Unfortunately, C is a procedural language that does not include these features. Luckily, it is possible to replicate some of the core features of OOP in C by following design guidelines that are often called OOC in the literature. In this framework, the implementation was inspired by and based on the work of Quantum Leaps, LLC [2020].

OOC classes inherit the following OOP features:

- Encapsulation
- Inheritance
- Interface
- Polymorphism

and can be displayed in two ways:

Interface Classes: these are responsible for enforcing a required signature based on the methods of the classes that implement it.

Classes: these are any other class type. They may inherit interface classes or other classes that can lead to a specialized pattern of behavior.

Every class should always ensure the structure initialization (input argument) and instance (output argument) 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 a 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 abstracted format as arguments of the initialization structure.

Being a NFC, classes should only contain private elements. This means that, only the class can directly access variables from the instance handler. Any other external access should always be done by "getter and setter" methods.

3.4.3 Libraries

Libraries are single instance components that may cluster a group of other components to implement a system functionality. Besides that, libraries can also abstract third-party libraries or provide multi-thread implementations. Library components are presented into three types:

Libraries: these are components that provide higher-level functionalities. To illustrate this, Figure 2 shows a simplified design of a robot arm firmware.

In this example, which is depicted in Figure 2, Inter-Integrated Circuit (I2C) motor driver IC device instances are responsible for controlling the arm motors. Each of the Joint and Claw classes instantiates a single instance of the motor device. The 3 Joint Arm Robot Class inherits the Arm Robot Interface class and instantiates three Joint classes and one Claw class object. Next, the Robot library instantiates one 3 Joint Arm Robot class Object and operates the necessary I2C functions. Finally, the library offers thread-safe access functions so that all the tasks (applications) can control the robot in a participatory way.

Third-Party Libraries: these are libraries 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 framework is code reuse, it is highly recommended that developers make use of good-quality libraries instead of starting everything from scratch.

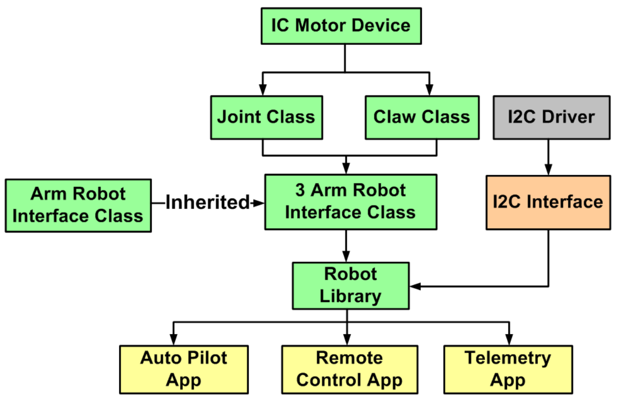

Figure 2. Example of Robot Modules

It is essential that these libraries are not considered to be a part of the component layer. Third-party libraries are often found to be provided with vendor code-generator tools. This means they are an exception layer that can exist in both PROJECT and CORE.

The main disadvantage of third-party libraries is that they may be limited to a particular domain. For example, a Liquid Crystal Display (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 whole revised implementation would be required. For this reason, an Interface library should abstract every third party.

Interface Libraries: these are responsible for combining and standardizing other libraries or third-party libraries into a process flow. If the previous LCD example is followed, every display library could be added to the generic interface display library, which would carry out abstracted display functions (Figure 3). The interface library can select the corresponding library by means of a run-time implementation or compile time through a macro compiler preprocessor.

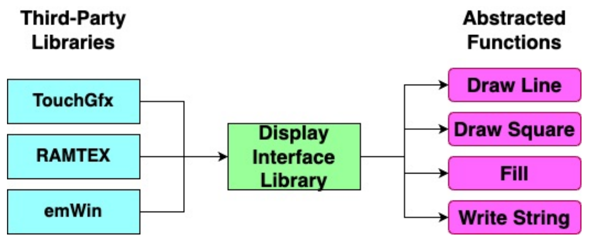

Figure 3. Display Interface Library

The 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 Application Programming Interface (API) called CMSIS-RTOS that already abstracts several RTOSs. Unfortunately, these libraries are usually architecture-specific and to avoid this problem, the RTOS kernel should always be abstracted by an in-house library.

3.5 Layer 3 - Applications

The applications layer is the last framework layer. It is responsible for deploying every system flowchart. The application can be designed in the form of RTOS threads or bare-metal state machines. In both cases, applications should only include modules from Layer 2.

A typical problem that novice developers face is encountering unexpected patterns of behavior which are caused by a failure to see the implications of running a code outside the context of the RTOS. This means that, every CORE should contain a System application (even for bare-metal projects). It is a unique application that should be seen as the "main function" of the project. All the systems initialization process and starting -points for other activities should be carried out inside this task. This application is a NFC that is designed to ensure that every developer executes all the CORE code in an RTOS environment. Finally, starting the System task should be done by PROJECT and is, strictly, the only framework , without exception.

3.6 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 framework establishes a few conventions. It is also advisable for the system architect to define his conventions.

3.6.1 Module Prefixes

Architects are free to choose module prefixes as they see fit. However, the framework recommends the list provided in Table 3. The modules not listed in this table do not need to have prefixes. The prefixes should be used in module files and every public function. Moreover, the framework advises that they should be used by other code elements (`typedef`s and macros, for example), although this is not a essential requirement. The use of prefixes allows developers to locate and understand the dependencies of the code they are working on while also ensuring the project is better organized.

Table 3. Module Prefixes

Module	Prefix
Interfaces	mid
Devices	dev
Classes	cls
Libraries	lib
Applications	app

3.6.2 Module Files

The framework 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. As well as this, a set of standard files (Table 4) is defined to ensure an improved module organization. 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. With the aid of this `typedef`s can be separated from prototype functions, and allow the modules to share types without experiencing conflicts.

3.6.3 Documentation

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

Table 4. Module Files Requirements

File	Interfaces	Devices	Classes	Libraries	Applications
source	No	Yes	Yes	Yes	Yes
public header	Yes	Yes	Yes	Yes	Yes
public types header	Optional	Optional	Optional	Optional	Optional
private header	No	Optional	Optional	Optional	Optional
internal header	No	Optional	Optional	Optional	Optional
override header	No	No	Yes, If Overrides Methods	No	No
override source	No	No	Yes, If Overrides Methods	No	No
readme document	Yes	Yes	Yes	Yes	Yes

3.6.4 Module Implementation

As outlined in previous sections, Table 5 summarizes some key definitions of implementation for the framework modules. Each layer may contain one or more types of modules with restricted access to other modules. Modules should respect the access permissions of each class so that they can maintain the correct level of abstraction. Each module type may require a different path design code. This must also be able to support the standardization framework. Finally, modules may be used for other implementation domains, which means that, multi-threading must always be taken into account if dependencies are not thread-safe.

Table 5. Summary of the Framework

Layer	Module	Implementation	Access Layers	Design
0	Drivers	Bare-Metal	0 and 1	None
0	HAL	Bare-Metal	0 and 1	None
1	Interfaces	Bare-Metal	None	Prototyping
2	Devices	Bare-Metal	None	Instance Handlers
2	Classes	Bare-Metal/RTOS	1 and 2	Object Oriented C
2	Libraries	Bare-Metal/RTOS	1 and 2	Procedural
3	Applications	Bare-Metal/RTOS	2 and 3	Procedural

3.7 Final Considerations

A comprehensive description of the framework would be too extensive for a single paper, and for this reason, only its key features were described in this article. In addition, the framework is under constant development. Readers can access the current state of the framework and the complete documentation in the public framework repository [1].

4 Case Study

To validate the proposed framework, a case study based on an IoT system was used. The system supports an application that collects environmental data and provides fused data to end-users. An important feature of this system is the ability to autonomously calibrate new sensor nodes that are added into the network [Farina *et al.*, 2023]. This self-calibration sensor firmware was refactored into the framework format and compared with the original one, demonstrating the proposed framework's value.

The deployment was implemented using an ESP32-WROOM-32 Module [Systems, 2017]. The official Espressif IoT Development Framework (ESP-IDF) was used to develop the hardware-specific implementations. Each firmware region was implemented in a different repository. First, PROJECT [2] was created, and then the initial project files were added. Next, CORE [3] was built and included as a submodule of PROJECT.

Table 6. Framework of the Dependency Map

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**	

The system was separated into two applications. As expected, System (Figure 4), the first one, was responsible for creating the required initialization and starting the Calibration task (Figure 5). Two interface libraries were created for this project ensure abstraction of Third-Party libraries. Even though these modules are project components, they were structured as external components to demonstrate that they were examples of reusable libraries. In both cases, developers can include other Third-Party libraries in these interfaces without affecting other modules.

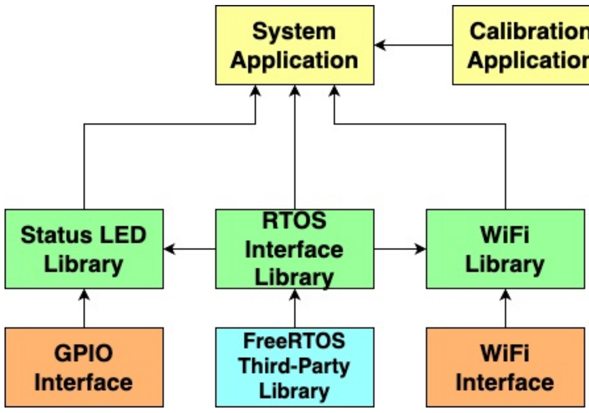

Figure 4. System Application Dependencies

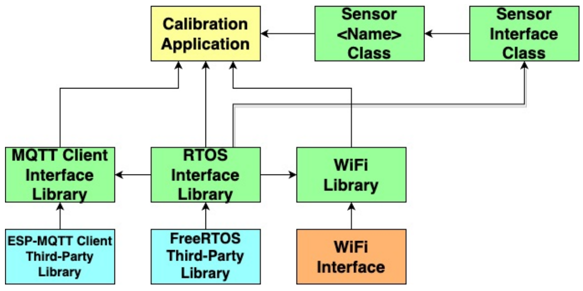

Figure 5. Calibration Application Dependencies

The Status LED and WiFi libraries are responsible for the custom project implementations of the middleware peripheral interfaces and ensuring these resources had thread-safe access to the applications. Different IoT sensors can be implemented through sensor interface class child classes. By doing this, developers can override the implementation of the method responsible for collecting raw sensor sample data. All the other implementations are inherited from the parent class, and do not require modifications.

4.1 Evaluation

A comparison between the framework and the legacy project was made to assess the framework's benefits. First, the statistics for both projects were extracted with the help of code metric analyzer software [Arnold, 2022]. The initial analysis (Table 7) compares the outlines of both projects, and the results show a significant increase in all four items. This was expected since the framework enhances the system modularization; thus, more Lines of Code (LOC) and files are required to encapsulate these modules.

Table 7. Comparative Overview of the Project

	Legacy	**Framework**
Lines Of Code	367	487
Files	9	33
Functions	15	44
Modules	4	10

In a second analysis, the number of Effective Lines of Code (ELOC), represented by the LOC inside functions, were compared. The framework advises developers to restrict functions to 40 LOC. The reason for this is that most Integrated Development Environments (IDE) can display, at least, this number of lines (in their default configuration) without requiring any scroll. As a result, the functions become more readable and less complex.

The ELOC comparison is displayed in Table 8. The results show that the functional size of the framework was significantly reduced while being kept in a better range. McCabe's Cyclomatic Complexity McCabe [1976] was used to evaluate the quality of the functions and improve this analysis.

Table 8. Effective Lines of Code Comparison

	Legacy	**Framework**
Mean	21.33	8.70
Standard Deviation	22.73	5.42
Max	92	28
Min	3	3
Total	320	383

The results of the functional complexity (Table 9) show that when compared with legacy functions, framework functions have an average of 70% less complexity with an improved variation range. These results demonstrate that as well as portability and reuse, the framework can also improve the quality of the developed code, and enable it to be more easily understood, changed, and debugged.

In a final evaluation, a coupling analysis for both projects was

Table 9. Comparison of Cyclomatic Complexity

	Legacy	**Framework**
Mean	3.80	1.05
Standard Deviation	5.60	1.68
Max	21	9
Min	0	0
Total	57	46

conducted. In the legacy project, 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 Framework version, implementations on the PROJECT region do not directly interact with CORE modules. In light of this, the Framework project only takes account of the dependencies that are included inside the CORE region. The dependency map for the legacy and framework projects is shown Table 10 and Table 6 respectively.

Table 10. Legacy Dependency Map

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**	

Since both projects make use of a RTOS, it is natural that multiple modules depend on either `FreeRTOS` (legacy) or `lib_rtos` (framework). This dependency on framework 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 usually designed to be portable, but are not reusable. Their main purpose is to use other modules to implement a particular system flow which means that they will probably rely on multiple dependencies.

Finally, the quality of the module encapsulation of the framework is clear from the results obtained for the component and interface modules. Each of these modules is completely decoupled from the others , in contrast with the legacy version and, as a result, provides several benefits. Moreover, in this way, the modules can be easily changed without causing problems to the others. Other modules can replace them with the same functionalities without requiring a complete system implementation review.

5 Conclusion

This paper proposed an embedded firmware development framework that simplifies IoT device portability. Despite its usefulness for embedded firmware of devices used in other domains, it is particularly useful for IoT devices, such as explored in the validation case study. It also lays down guidelines for developing less complex and reusable modules. The study included a complete descriptive overview of the framework and a practical example of an implementation. Moreover, by evaluating the implemented system, it was possible to demonstrate some of the key benefits of this framework.

In summary, the proposed framework gives guidelines to developers on how to improve their design, and deploy less complex, portable, and reusable firmware architectures that are flexible and simple. Novice software developers can also benefit from NFCs by learning how to increase their collaboration on projects and improve their skills. Finally, discontinued MCUs and other acICs can be easily replaced with minimal effects on the firmware.

There are still many open areas which can benefit from this framework. For example, future studies could include unit testing, continuous integration and deployment (CI/CD), automation tools, and module management.

Acknowledgements

This study was in part funded by the Coordination for the Improvement of Higher Education Personnel - Brazil (CAPES) - Finance Code 001 and in part by National Council for Scientific and Technological Development - Brazil (CNPq) Project 309505/2020-8. We are also grateful for the assistance provided by the CEREIA Project (# 2020/09706-7) São Paulo Research Foundation (FAPESP), as well as FAPESP–MCTIC-CGI.BR in partnership with Hapvida NotreDame Intermédica group.

Declarations

Authors' Contributions

Mauricio D. O. Farina: Designed the solution; Developed the code; Designed the Experiments; Executed the experiments; Analysed the results; and wrote the first draft Daniel H. Pohren: Analysed the data; Revised the text; Rewrote parts of the text. Alexandre dos S. Roque: Analysed the data; Revised the text; Rewrote parts of the text. Joao Paulo J. da Costa: Designed the Experiments; Revised the text; Validate the final results. Lisandra Manzoni Fontoura: Provided theoretical support; Revised the text; Rewrote parts of the text. Julio C. S. dos Anjos: Designed the Experiments; Analysed the data; Revised the text; Rewrote parts of the text. Edison Pignaton de Freitas: Supervised the whole work; Designed the research; Designed the Experiments; Revised the manuscript and Validate the final results.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

Data can be made available upon request.

References

Anjos, J. C. S., Gross, J. L. G., Matteussi, K. J., González, G. V., Leithardt, V. R. Q., and Geyer, C. F. R. (2021). An Algorithm to Minimize Energy Consumption and Elapsed Time for IoT Workloads in a Hybrid Architecture. *Sensors*, 21(9):1–20. DOI: 10.3390/s21092914.

Arnold, S. (2022). Cccc. Available at: `https://github.com/sarnold/cccc` .

Beningo, J. (2017). *Reusable Firmware Development: A Practical Approach to APIs, HALs and Drivers*. Apress. DOI: 10.1007/978-1-4842-3297-2.

Brooks, F. P. and Bullet, N. S. (1987). Essence and accidents of software engineering. *IEEE computer*, 20(4):10–19. Available at: `https://www.researchgate.net/profile/Frederick_Brooks_Jr/publication/30868224_Essence_and_Accidents_of_Software_Engineering/links/0fcfd50d5e8c4aaf8a000000.pdf`.

Clements, A. A., Carpenter, L., Moeglein, W., and Wright, C. M. (2021). Is your firmware real or re-hosted?. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States). Available:`https://www.osti.gov/servlets/purl/1855562`.

Clements, A. A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., and Payer, M. (2020). {HALucinator}: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1201–1218. Available at:`https://www.usenix.org/conference/usenixsecurity20/presentation/clements`.

Connectivity Standards Alliance (2024). Matter. Available at: `https://csa-iot.org/all-solutions/matter/` Acessed in: 2024-02-12.

Dano, E. B. (2019). Importance of reuse and modularity in system architecture. In *2019 International Symposium on Systems Engineering (ISSE)*, pages 1–8. IEEE. DOI: 10.1109/ISSE46696.2019.8984472.

Douglass, B. P. (2010). *Design patterns for embedded systems in C: an embedded software engineering toolkit*. Elsevier. Book.

Fahmideh, M., Ahmad, A., Behnaz, A., Grundy, J., and Susilo, W. (2022). Software engineering for internet of things: The practitioners' perspective. *IEEE Transactions on Software Engineering*, 48(8):2857–2878. DOI: 10.1109/TSE.2021.3070692.

Farina, M. D., dos Anjos, J. C., and de Freitas, E. P. (2023). Real-time auto calibration for heterogeneous wireless sensor networks. *Journal of Internet Services and Applications*, 14(1):1–9. DOI: 10.5753/jisa.2023.2739.

Feng, B., Mera, A., and Lu, L. (2020). {P2IM}: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1237–1254. Available at:`https://www.usenix.org/conference/usenixsecurity20/presentation/feng`.

Fowler, M. (2012). *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley. Book.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH. DOI: 10.1007/3-540-47910-4_21.

Gustafson, E., Muench, M., Spensky, C., Redini, N., Machiry, A., Fratantonio, Y., Balzarotti, D., Francillon, A., Choe, Y. R., Kruegel, C., *et al.* (2019). Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 135–150. Available at:`https://www.usenix.org/conference/raid2019/presentation/gustafson`.

Hubalovsky, S. and Sedivy, J. (2010). Mistakes in object oriented programming. In *2010 2nd International Conference on Information Technology,(2010 ICIT)*, pages 113–116. IEEE. Available at:`https://ieeexplore.ieee.org/abstract/document/5553377`.

Jia, M., Sha, E. H.-M., Zhuge, Q., and Gu, S. (2022). Transient computing for energy harvesting systems: A survey. *Journal of Systems Architecture*, 132:102743. DOI: 10.1016/j.sysarc.2022.102743.

Marcondes, H., Hoeller, A. S., Wanner, L. F., and Frohlich, A. A. M. (2006). Operating systems portability: 8 bits and beyond. In *2006 IEEE conference on emerging technologies and factory automation*, pages 124–130. IEEE. DOI: 10.1109/ETFA.2006.355371.

Martin, R. C. (2017). Clean architecture: A craftsman's guide to. Available at:`https://www.papiro-bookstore.com/wp-content/uploads/2021/12/Clean-Architecture.pdf`.

Martins Gomes, R. and Baunach, M. (2021). A study on the portability of iot operating systems. *Tagungsband des FG-BS Frühjahrstreffens 2021*. DOI: 10.18420/fgbs2021f-01.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320. DOI: 10.1109/TSE.1976.233837.

Motogna, S., Vescan, A., and Şerban, C. (2023). Empirical investigation in embedded systems: Quality attributes in general, maintainability in particular. *Journal of Systems and Software*, 201:111678. DOI: 10.1016/j.jss.2023.111678.

Neser, M. and van Schoor, G. (2009). Object-oriented embedded c. *SAIEE Africa Research Journal*, 100(4):90–96. DOI: 10.23919/SAIEE.2009.8531856.

Quantum Leaps, LLC (2020). Object-oriented programming in c. Available at:`https://github.com/QuantumLeaps/OOP-in-C` Accessed in: 2022-09-26.

Schmidt, D. C., Gokhale, A., and Natarajan, B. (2004). Leveraging application frameworks: why frameworks are important and how to apply them effectively. *Queue*, 2(5):66–75. DOI: 10.1145/1016998.1017005.

Sommerville, I. (2015). Software engineering. 10th. *Book Software Engineering. 10th, Series Software Engineering*, 10. Book.

Spray, J. and Sinha, R. (2018). Abstraction layered architecture: Writing maintainable embedded code. In *European conference on software architecture*, pages 131–146. Springer. Book.

Stewart, D. B. (1999). Twenty-five most common mistakes with real-time software development. In *Proceedings of the 1999 Embedded Systems Conference (ESC'99)*, volume 141. Available at:`https://webpages.charlotte.edu/~jmconrad/ECGR4101-2005-08/Twenty-Five.pdf`.

Sun, L., Li, Y., and Memon, R. A. (2017). An open iot framework based on microservices architecture. *China Communications*, 14(2):154–162. DOI: 10.1109/CC.2017.7868163.

Systems, E. (2017). Esp-wroom-32 datasheet. Available at:`https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf`.

Tremaroli, N. J. (2023). *Adaptive Firmware Framework for Microcontroller Development*. PhD thesis, Virginia Tech. Available at:`https://vtechworks.lib.vt.edu/items/fb7eeaaa-f6c8-4a67-88d2-a20f0375a274`.

Willocx, M., Bohé, I., Vossaert, J., and Naessens, V. (2018). Developing maintainable application-centric iot ecosystems. In *2018 IEEE International Congress on Internet of Things (ICIOT)*, pages 25–32. IEEE. DOI: 10.1109/ICIOT.2018.00011.

Yuan, C., Liu, Z., Wang, X., and Yuan, F. (2021). A component development framework for embedded software. In *2021 IEEE International Conference on Information Communication and Software Engineering (ICICSE)*, pages 71–75. IEEE. DOI: 10.1109/ICICSE52190.2021.9404109.

Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D., *et al.* (2014). Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, volume 14, pages 1–16. Available at:`https://www.researchgate.net/profile/Jonas-Zaddach/publication/269197057_`

`Avatar_A_Framework_to_Support_Dynamic_Security_Analysis_of_Embedded_Systems'_Firmwares/links/5e0b2725299bf10bc3852355/Avatar-A-Framework-to-Support-Dynamic-Security-Analysis-of-Embedded-Systems-Firmwares.pdf`.

Zheng, X., Liang, S., and Xiong, X. (2021). A hardware/software partitioning method based on graph convolution network. *Design Automation for Embedded Systems*, 25(4):325–351. DOI: 10.1007/s10617-021-09255-9.