BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

EXTENSIBLE RUST LIBRARY FOR THE DEVELOPMENT OF EMBEDDED SENSOR APPLICATIONS ON ESP32 PLATFORM

ROZŠIŘITELNÁ KNIHOVNA V JAZYCE RUST PRO PODPORU VÝVOJE VESTAVĚNÝCH SENZORICKÝCH APLIKACÍ NA PLATFORMĚ ESP32

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **KIRILL MIKHAILOV**
AUTOR PRÁCE

SUPERVISOR **Ing. VÁCLAV ŠIMEK,**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

156827

Institut:	Department of Computer Systems (DCSY)
Student:	**Mikhailov Kirill**
Programme:	Information Technology
Title:	**Extensible Rust Library for the Development of Embedded Sensor Applications on ESP32 Platform**
Category:	Embedded Systems
Academic year:	2023/24

Assignment:

1. Survey the existing means of Rust language support for the ESP32 range of microcontrollers. Focus on approaches to peripherals configuration and usage in common embedded application types.
2. Based on results obtained in point 1) of the assignment, propose a set of frequently used sensors and other peripherals to be supported by the library.
3. Outline a generic scheme for handling the peripherals in a user-convenient way. The aim is to isolate end-users from technical details of controlling the peripheral devices.
4. Specify the Rust library interface and architecture. The user interface should be preserved for any future releases. The architecture must employ modular features.
5. Implement the library based on point 4) of the assignment. Create and deploy appropriate unit tests, and add examples to demonstrate common use cases.
6. Prepare concise documentation and publish your solution on GitHub under an open-source license (Apache License preferred, see http://www.apache.org/licenses/LICENSE-2.0).
7. Evaluate the functionality of the library. Assess the achieved results and try to propose further directions for the development.

Literature:

- According to the instructions of the supervisor.

Requirements for the semestral defence:
Fulfillment of points 1 to 4 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Šimek Václav, Ing.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	30.10.2023

Abstract

This thesis introduces an extensible Rust library designed for embedded sensor applications on the ESP32 platform, addressing the need for simplified development of real embedded systems in a Rust language environment on this platform. A significant contribution of this thesis is the development of a user-friendly interface for sensor management. This interface allows for straightforward sensor installation, activation, and monitoring, catering to applications like smart homes and automation without requiring in-depth technical knowledge of peripherals. The library's architecture is carefully designed for modularity and extensibility, adhering to Rust's safety and efficiency principles. Accompanied by comprehensive documentation, the project is released on GitHub under an open-source Apache License, complete with unit tests and use-case examples. The thesis concludes with an evaluation of the library's functionality and potential future enhancements, demonstrating its practicality for embedded system developers.

Abstrakt

Tato práce představuje rozšiřitelnou knihovnu jazyka Rust určenou pro vestavěné senzorové aplikace na platformě ESP32, která řeší potřebu zjednodušeného vývoje reálných vestavěných systémů v prostředí jazyka Rust na této platformě. Významným přínosem této práce je vývoj uživatelsky přívětivého rozhraní pro správu senzorů. Toto rozhraní umožňuje jednoduchou instalaci, aktivaci a monitorování senzorů, což vyhovuje aplikacím, jako jsou inteligentní domy a automatizace, aniž by vyžadovalo hluboké technické znalosti periferií. Architektura knihovny je pečlivě navržena s ohledem na modularitu a rozšiřitelnost a dodržuje zásady bezpečnosti a efektivity Rustu. Projekt je doprovázen rozsáhlou dokumentací a je zveřejněn na platformě GitHub pod open-source licencí Apache, doplněn unit-testy a příklady použití. V závěru práce je zhodnocena funkčnost knihovny a její případná budoucí vylepšení, která ukazují její praktičnost pro vývojáře vestavných systémů.

Keywords

Rust, ESP32, IoT, SoC, embedded systems, embedded applications, user-friendly development, sensors, microcontroller, embedded Rust, Rust library, Rust crate, Smart Home, MQTT

Klíčová slova

Rust, ESP32, IoT, SoC, vestavěné systémy, vestavěné aplikace, uživatelsky přívětivý vývoj, senzory, mikrokontrolér, vestavěný Rust, knihovna Rust, Rust crate, chytrá domacnost, MQTT

Reference

MIKHAILOV, Kirill. *Extensible Rust Library for the Development of Embedded Sensor Applications on ESP32 Platform.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Václav Šimek,

Rozšířený abstrakt

Tato práce představuje vývoj rozšiřitelné knihovny v jazyce Rust, která zjednodušuje vývoj aplikací pro vestavěné senzory na platformě ESP32. S rostoucím zájmem o vestavné systémy, zejména pro aplikace, jako jsou inteligentní domy a automatizace, roste poptávka po vývojových nástrojích, které zjednodušují proces programování. V této souvislosti se často používají mikrokontroléry od společnosti Espressif Systems, které jsou uznávány pro svůj vysoký výkon, všestrannost a cenovou výhodnost. Programování těchto zařízení pomocí jazyka Rust však navzdory jeho výhodám z hlediska bezpečnosti a efektivity přináší komplikace kvůli strmé křivce učení tohoto jazyka a složitosti ovládání potřebného hardwaru.

Pro řešení těchto problémů tato práce vyvíjí knihovnu založenou na jazyce Rust, která abstrahuje složitosti spojené s ovládáním senzorů a periferií a zpřístupňuje proces vývoje jak zkušeným programátorům, tak i nováčkům. Knihovna poskytuje uživatelsky přívětivé rozhraní pro konfiguraci a správu dat ze senzorů a umožňuje uživatelům soustředit se více na logiku aplikace než na nízkoúrovňovou interakci s hardwarem. Implementace knihovny obsahuje metody pro přímé ovládání a čtení dat z různých typů senzorů, jako jsou senzory teploty, vlhkosti a osvětlení, s možností snadného rozšíření na další typy periferií.

Metodika ovládání senzorů v knihovně zahrnuje použití rysů (angl. traits) jazyka Rust a generického programování pro abstrahování hardwarových funkcí, což umožňuje snadné přizpůsobení knihovny pro různé typy hardwaru bez nutnosti změn v kódu aplikace. Vývojáři tak mohou efektivněji vytvářet aplikace kompatibilní s různými senzory a periferiemi bez hlubokých znalostí detailů konkrétního hardwaru. Kromě senzorů byla knihovna rozšířena o podporu displejů a vstupních zařízení. Umožňuje snadnou integraci s Wi-Fi a MQTT, což rozšiřuje její využití pro aplikace internetu věcí.

Architektura knihovny je modulární, což podporuje rozšiřitelnost a údržbu. Podporuje řadu běžně používaných senzorů a periferií, což zajišťuje širokou použitelnost. Knihovna dodržuje přísné bezpečnostní standardy a standardy efektivity Rust a její konstrukce podporuje bezpečný a spolehlivý vývoj aplikací. Pro zvýšení dostupnosti a možností přispívání komunity je projekt umístěn na serveru GitHub pod open-source licencí Apache.

Knihovna je doplněna rozsáhlou dokumentací, která obsahuje návody, případy použití a osvědčené postupy pro implementaci. Součástí práce jsou příklady a jednotkové testy, které demonstrují možnosti knihovny v reálných scénářích. Vyhodnocení knihovny prostřednictvím teoretické analýzy a praktického testování zdůrazňuje její účinnost při zjednodušování vývoje vestavných aplikací. Zpětná vazba od uživatelů neznalých jazyka Rust i od profesionálů naznačuje, že knihovna výrazně snižuje vstupní bariéru pro vývoj vestavných aplikací na mikrokontrolérech rodiny ESP.

Závěrem lze říci, že tato práce nejen přispívá praktickým nástrojem pro vývoj vestavných aplikací, ale také rozšiřuje soubor znalostí o použití jazyka Rust pro složité úlohy interakce s hardwarem. Budoucí práce by se mohla zabývat hlubší integrací s dalšími platformami internetu věcí, aby se ještě více zjednodušil proces vývoje, nebo začlenit více automatizovaných nástrojů a vzdělávacích zdrojů, které by pomohly novým uživatelům.

Extensible Rust Library for the Development of Embedded Sensor Applications on ESP32 Platform

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Václav Šimek. The supplementary information was provided by esp-rs team of Espressif Systems. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Kirill Mikhailov
May 7, 2024

Acknowledgements

First of all, I would like to thank my supervisor Ing. Václav Šimek for his good mentoring, patience and helpful feedback during this work. I am very happy that we matched each other in terms of workflow.

I would also like to thank Martin Vychodil, an employee of Espressif Systems, who made the contact with Mr. Šimek and this project overall possible.

Of course I want to thank the following Espressif employees and esp-rs team members for their responsive help with the driver and quick response to problems: Scott Mabin, Juraj Sadel, Jesse Robin James Braham, Björn Quentin, Sergio Gasquez Arcos, and Juraj Michálek.

And last, on the most important - I would like to thank my family and loved ones for their support, without which I would not have made it through this journey.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The evolution of embedded systems has paved the way for the creation of increasingly complex and diverse applications – from smart home devices to industrial automation. A significant place in these developments is occupied by the ESP-series microcontrollers, known for their versatility and fairly cheap price, relative to competitors. However, programming these devices, especially in the Rust language, might become a tough challenge. The Rust language, known for its safety and efficiency, currently exhibits a steep learning curve in the field of embedded systems, and the code often seems overly complex or intricate [38]. This complexity is an obstacle not only for experienced programmers, but more importantly for enthusiasts, newcomers and professionals outside of the traditional IT field who want to immerse themselves in the world of embedded systems. But despite this, as we can see from the Yalantis survey on Figure 1.1, the number of people interested in Rust is only growing. This also applies to big companies – Microsoft, Facebook, Discord, Dropbox and many others are starting to use Rust in their codebases [38].

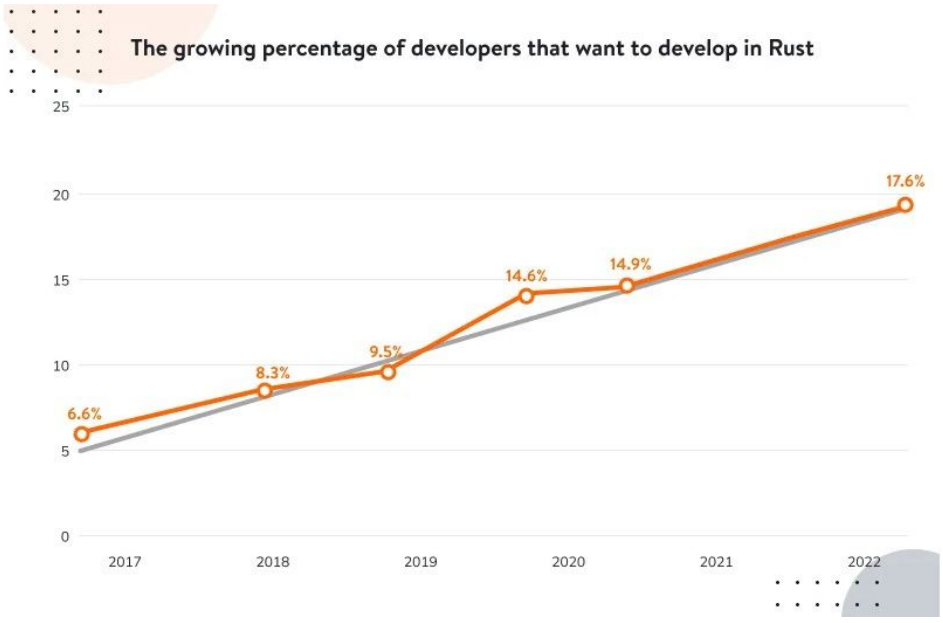


Figure 1.1: A study of the popularity of the Rust language [29].

Another indication of popularity is how many libraries are downloaded daily from **crates.io**[1], the official package registry of the Rust language. As demonstrated on Figure 1.2, at the time of writing this thesis, the average number of library downloads in 7 days is in the hundreds of millions.

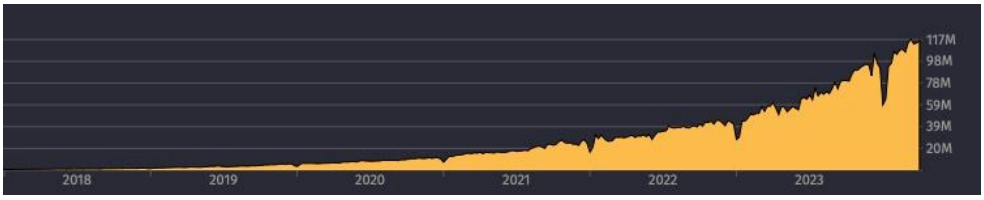


Figure 1.2: Daily crate downloads since Rust 1.0, 7-day average [28].

The primary motivation for this project, therefore, is to simplify the process of writing embedded applications in Rust for the ESP32 platform. The goal is to transform its' current state into a more accessible and convenient format, to approach a state similar to that of Arduino or similar popular solutions in terms of simplicity. This endeavor is aimed not only at programmers already interested in Rust on ESP32, but also at people far from IT and programming. By abstracting away the complexities of the Rust language and ESP32 hardware, the project aims to lower the barrier to entry, allowing a wider range of users to effectively develop and deploy embedded Rust applications. Whether it's a hobbyist looking to automate their home, a teacher incorporating technology into the classroom, or a small business innovating their services, this library is designed to give them the tools to bring their ideas to life without baffling them with the technical depths of Rust programming and microcontroller control.

1.2 Thesis Overview

Each part of this thesis builds on the previous one. Chapter 2 starts with an introduction to the fundamental knowledge of the tools used: the Rust language and its features and peculiarities that must be kept in mind to understand the issues, the ESP32 range of microcontrollers, the support of the aforementioned language on these chips and the infrastructure providing this support. In Chapter 3 the content will develop from theoretical research to practical analysis, review of existing solutions and the process of solving the problem, paying attention to both successful and implemented solutions and experimental ideas that for one reason or another did not make it to the final solution. Chapter 4 will cover the public part of this library, the architecture of the Web site, and the design of the technical documentation for it. It will also focus on suggested ways to test library functionality, providing information on both existing examples for real hardware and automated testing. A guide on how to get started with the library will also be found in this part. The next chapter (5) will provide the reader with the an evaluation of the results of the work by means of pairwise comparison of several programs performing the same functionality but written with and without using the resulting library. After that, the last, 6th Chapter, will summarize the whole work.

[1] crates.io package registry: `https://crates.io`

Chapter 2

Rust Fundamentals and Embedded Systems

2.1 What is Rust?

As the beginning of the analysis, the story told by Clive Thompson [42] of how the creator of Rust came up with the initial idea is an excellent description of why this language is the way it is:

> *„In 2006, Graydon Hoare was a 29-year-old computer programmer working for Mozilla, the open-source browser company. Returning home to his apartment in Vancouver, he found that the elevator was out of order; its software had crashed. This wasn't the first time it had happened, either.*
> *Hoare lived on the 21st floor, and as he climbed the stairs, he got annoyed. "It's ridiculous," he thought, "that we computer people couldn't even make an elevator that works without crashing!" Many such crashes, Hoare knew, are due to problems with how a program uses memory. The software inside devices like elevators is often written in languages like C++ or C, which are famous for allowing programmers to write code that runs very quickly and is quite compact."*
>
> — *Clive Thompson*

From this story reader can see, why Mr. Hoare wanted to create a language that would code that would run fast but have a focus on security.

2.1.1 Rust Language

`Rust` is a general-purpose, multi-paradigm programming language engineered for creating safe, secure and scalable applications. It was initially designed as a systems programming language, however, it has emerged as a more versatile language capable of creating a variety of application types, including systems programming, web services, desktop applications, embedded systems, and more [31]. Developed by the aforementioned Graydon Hoare at Mozilla Research, it was first announced in 2010 and first stable release happened in 2015. Rust has since garnered acclaim for addressing many of the pitfalls of system-level programming [3].

Detailed theoretical aspects:

- **Ownership and Borrowing**: The concepts of Ownership and Borrowing are fundamental to how the language manages memory and ensures safety, all without a garbage collector. Ownership in Rust is based on a few core rules. Each value in Rust has a specific owner – a variable. When the owner goes out of scope, Rust automatically deallocates the memory. This process is automatic and deterministic, contrasting with languages that use garbage collection [26].

  When it comes to assigning values from one variable to another, Rust transfers ownership of the value to the new variable. Example of working with this concept is shown on Listing 2.1.

```
fn main() {
    let s1 = String::from("Hello, Rust!"); // s1 owns the string
    let s2 = s1; // Ownership of the string is moved to s2

    // println!("{}, world!", s1); // Error! s1 no longer owns the string

    let s3 = &s2; // s3 borrows the string
    println!("{}, world!", s3); // OK! s3 only borrows the value in s2

    let len = calculate_length(&s2); // s2 is borrowed by calculate_length
    println!("The length of '{}' is {}.", s2, len);
}

// This function takes a reference to a String and returns its length
fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Listing 2.1: Ownership and Borrowing example [26]

  This unique approach prevents issues like double freeing memory, which is common in other low-level languages.

  The practical implications of these features are significant. They make Rust a language in which memory safety is an essential prerequisite. By solving the problem of compile-time memory management through ownership and borrowing, Rust eliminates entire classes of bugs that other system programming languages suffer from.

- **Type System and Lifetimes**: Rust's type system is central to its guarantee of safe system programming, employing static type checking to validate types at compile time. This preemptive approach eliminates a range of common software errors, enhancing both safety and performance.

  Furthermore, Rust introduces the concept of `Lifetimes`. Ownership and Borrowing (2.1.1) are two of the three pillars in the ownership model, `lifetimes` is the final one [31]. As shown on Listing 2.2, it serves to manage data references, ensuring their validity precisely for their usage duration, thus avoiding issues like dangling references or premature data deallocation.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 2.2: Lifetime usage example [26].

In this function, **'a** is a *lifetime generic parameter* that specifies that the lifetimes of `x`, `y`, and the `return` value are all the same. This ensures that the returned reference will be valid as long as both `x` and `y` are valid, preventing dangling references. Lifetime annotations help the compiler understand how references relate to each other in terms of their lifetimes, ensuring memory safety [26, 31].

- **Pattern Matching and Functional Features**: Pattern matching is one of the many functional features that maybe new to systems programming, but `Rust` has borrowed from functional programming paradigms [6].

  In `Rust`, pattern matching is primarily used with '**match**' statements and in function arguments. This is particularly useful when dealing with `Rust`'s enumerations (`enums`). The '**Option**' type is used to represent an „optional" value: every '**Option**' is either '**Some**' and contains a value, or '**None**' if does not. '**Result**' types, on the other hand, are richer and are used for functions that can return an error. A '**Result**' is either '**Ok**', meaning the operation succeeded, or '**Err**', meaning the operation failed [1]. See example, shown in Listing 2.3.

```
fn divide(numerator: f64, denominator: f64) -> Option<f64> {
    if denominator == 0.0 {
        None
    } else {
        Some(numerator / denominator)
    }
}

fn main() {
    let result = divide(10.0, 2.0);
    match result {
        Some(quotient) => println!("Quotient is: {}", quotient),
        None => println!("Cannot divide by 0"),
    }
    // "read" function returns "Result"
    let file_result = std::fs::read("test.txt");
    match file_result {
        Ok(content) => println!("File content read successfully."),
        Err(error) => println!("Error reading file: {:?}", error),
    }
}
```

Listing 2.3: Pattern matching example [1].

- **Zero-Cost Abstractions**: A key principle in Rust is that abstractions should not impose a runtime cost [41]. This principle, an example of the use of which is shown in Listing 2.4, is evident in its efficient iteration constructs, powerful enum classes, and advanced compile-time generics, all of which contribute to performant, high-level programming constructs. Some examples:

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    let doubled: Vec<i32> = numbers.iter().map(|&x| x * 2).collect();

    println!("{:?}", doubled); // This will print: [2, 4, 6, 8, 10]
}
```

Listing 2.4: Simple iterator usage example [41].

In this example, '`.iter().map(|&x| x * 2).collect()`' looks like it might be creating several intermediate data structures and might therefore be inefficient. However, due to `Rust`'s zero-cost abstractions, the iterator is compiled into efficient loop code that directly creates the doubled vector without any additional runtime overhead.

- **Generics and Traits**: **Traits** and **generics** are two powerful features in Rust that allow for code reuse and type safety in a flexible and efficient manner.

Traits in Rust can be thought of as a collection of methods that specific type has ans can share this behavior with other types. Example of basic **traits** usage is shown on Listing 2.5. When a type **implements** a `trait`, it provides specific behavior for the trait's method signatures [26]. Traits are similar to interfaces in other languages but with some unique Rust-specific properties. To implement a trait, `impl` keyword is used, followed by the trait name for the specific type:

```
trait Summary {
    fn summarize(&self) -> String;
}

struct Article {
    // ..items..
}

impl Summary for Article {
    fn summarize(&self) -> String {
        format!("{}, by {}", self.headline, self.author)
    }
}
```

Listing 2.5: Trait implementation example [26].

Generics are parameters for types, functions, methods, or structs that allow for the operation of the code over different data types while still being type-safe. They allow developers to write flexible, reusable code that works on any data type [26]. Generic types are recognized by their usage of angle brackets enclosing one or more type placeholders, like `Vec<T>`, where **T** can be any type, as it is shown on Listing 2.6.

```
struct Point<T> {
    x: T,
    y: T,
}

let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

Listing 2.6: Generics usage example [26].

- **Systems Programming Capabilities**: Rust is designed with systems programming in mind. It offers a blend of low-level control with high-level safety guarantees, positioning it as a modern alternative to traditional systems languages like C and C++. It's engineered to give developers the ability to manipulate hardware and memory layout directly, enabling the development of operating systems, embedded systems, and other performance-critical applications [38].

  What sets Rust apart is its uncompromising emphasis on safety, despite providing the raw power similar to C and C++. The language's ownership model, along with its type system, ensures that unsafe memory access is prevented at compile time, thereby avoiding a whole class of runtime errors typically associated with systems programming [38].

  However, Rust's approach to system safety does not entirely preclude the use of „unsafe" operations, the use of such a section is demonstrated on a Listing 2.7. The language provides an explicit `unsafe` keyword that allows developers to perform actions that are not checked by the compiler's safety guarantees.

```
fn main() {
    let x: i32 = 10;
    let raw_pointer: *const i32 = &x as *const i32;

    unsafe {
        // Inside an unsafe block, raw pointer might be dereferenced.
        println!("raw_pointer points to: {}", *raw_pointer);
    }

    // Outside the unsafe block, this operation is not allowed
    // println!("raw_pointer points to: {}", *raw_pointer);
    // Compile-time error
}
```

Listing 2.7: „Unsafe" block example [26].

Another example of powerful unsafe tool in Rust's arsenal is **std::mem::transmute** (or core::mem:transmute for `no_std` approach 2.2.2), which allows developers to arbitrarily cast a value between types without changing the bits that make up that value. This function is inherently unsafe and can lead to undefined behavior if not used with extreme caution. However, similar tools are often used in development at the closest levels to the hardware, such as in the **esp-hal** [11].

- **Cargo and Crates**: `Cargo` is a Rust's built-in package manager, in tandem with `crates.io`[1], the official package registry, form the backbone of Rust's package management and distribution system. They work together to greatly simplify several aspects of the Rust development workflow, from code compilation to dependency management.

  Rust programs are made of **crates**. Each of them is a complete, cohesive unit: all the source code for a single library or executable, plus any associated tests, examples, tools and configuration. Such library can be downloaded from `Crates.io`, a git repository or a path in a local machine.

  `Cargo` has a lot of different functions and it would not make sense to describe all the features, so for the purposes of this analysis, the focus is set on the most key ones. It allows the use of different build profiles. Those are customizable and configurable in the **Cargo.toml** file, as shown on Listing 2.8:

```
[profile.dev]
opt-level = 0
[profile.release]
opt-level = 3
# Customizing an optimization level of a profile for a specific package
[profile.dev.package.esp-wifi]
opt-level = 3
```

Listing 2.8: Profile configuration in **Cargo.toml** [26, 15].

  Then, when building, the profile can be selected by passing or not passing the „`--release`“ parameter (see Listing 2.9):

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
$ cargo build --release
    Finished release [optimized] target(s) in 0.0s
```

Listing 2.9: Building with different profiles [26].

  A prime example of `cargo`'s capabilities is its straightforward approach to integrating external libraries (or „crates“) into a Rust project. This integration is facilitated through the **Cargo.toml** file located at the project's root. In this file, developers can concisely list their project's dependencies, making project setup and management both clear and efficient. An example of the part of **Cargo.toml** file with different ways and features of including is shown in Listing 2.10.

```
[dependencies]
# Basic include
fugit = "0.3.7"
# Include from git
esp-wifi = { git = "https://github.com/esp-rs/esp-wifi.git" }
# Indlude with extra settings
esp-alloc = { version = "0.3.0", optional = true }
embedded-hal = { version = "0.2.7", features = ["unproven"] }
```

Listing 2.10: Including dependencies in **Cargo.toml**.

[1] crates.io package registry: https://crates.io

`Cargo` also supports additional features, allowing us to conditionally compile certain parts of code depending on what chosen features. This is especially useful for enabling or disabling certain features in the crate, something massively used in this thesis to create configurations for specific chips or to customize the library for specific user needs. It is also possible to enable and include a dependency that has been marked as `optional` in the build process, which is demonstrated on Listing 2.11.

```
[features]
esp32c6-mqtt = ["esp32c6-wifi", "mqtt"]
alloc = ["esp-alloc"]
async = ["embassy-executor"]
```

Listing 2.11: Feature definition.

- **Asynchronous Programming** Asynchronous programming in Rust is built around the concept of `Futures`, special types that represent a value which may not yet be available. Rust's **async** constructs enable non-blocking execution, allowing other tasks to progress in parallel to awaiting operations [7].

  An asynchronous function, denoted by the `async` keyword, returns a **Future**. This **Future** doesn't perform any operation until it is explicitly awaited with the `await` keyword, making it lazy by nature. The execution of asynchronous tasks is managed by an executor, which is responsible for polling futures to completion [7].

  The advantage of this model is its ability to handle a large number of concurrent operations efficiently. Basic example of **async** program is presented in Listing 2.12. Also, Rust does not include an executor for running these **Futures** by default, necessitating third-party libraries like **tokio**[2] or **embassy-executor**[3], frequently used for embedded applications.

```
use tokio::time::{sleep, Duration};

async fn perform_delayed_task() {
    println!("Task starts");
    sleep(Duration::from_secs(2)).await;
    println!("Task ends after a 2-second delay");
}

#[tokio::main]
async fn main() {
    perform_delayed_task().await;
}
```

Listing 2.12: Basic example of **async** usage. Adapted from concepts explained in [7].

- **Compiler and Error Handling**: The Rust compiler is often recognized for its comprehensive and instructive error messages. It not only identifies errors but often provides suggestions on how to rectify them, enhancing the development process by making debugging more intuitive and educational [22].

[2] tokio executor crate: https://crates.io/crates/tokio
[3] embassy-executor crate: https://crates.io/crates/embassy-executor

- **Community**: The language benefits from an active and engaged community. Rust evolves through an open RFC process, enabling community involvement in its development.

  Observing the **esp-rs** organization reveals that numerous commits are contributed by community members, indicating a robust community involvement. Furthermore, some drivers within the ecosystem are supported exclusively by community efforts, showcasing the collaborative nature of the development [11, 12].

2.1.2 Disadvantages of Rust

But, of course, like any language, Rust also has drawbacks compared to its competitors C and C++ [38]:

- **Complexity**: Compared to more traditional languages like C and C++, Rust's learning curve can be quite steep due to its distinctive features like ownership and borrowing, which require a new understanding of memory management.
- **Build Duration**: The sophistication of Rust's type system and its borrow checker, which provide safety guarantees, can also lead to increased compilation times, a factor that becomes more noticeable in larger codebases.
- **Development Environment**: Despite Rust's growing popularity, its development tooling is not yet as mature as that of languages with a longer history, such as above-mentioned C and C++, which may present a challenge in finding the right tools for specific tasks.
- **Direct Hardware Access**: While Rust prioritizes safety, this can sometimes mean sacrificing a degree of low-level control that languages like C and C++ offer, potentially complicating direct hardware interaction or certain optimizations.
- **Community**: Although the Rust community is active and expanding, it's still smaller than those of longstanding languages like C and C++, which means there may be fewer resources, libraries, and tools available to Rust developers.

2.2 Comparison of Rust Standard and Bare-Metal Environments

In Rust programming, the choice between `std` and `no_std` environments is pivotal, especially when dealing with system-level and embedded applications. This decision impacts how developers will structure their code, manage dependencies, and interact with the underlying system or hardware.

2.2.1 The Standard Library

The Rust standard library, also known as `std`, is a rich set of tools and functionalities that form the foundation for Rust software development [40]. It offers a wide variety of data types, such as vectors (2.1.1) and options (2.1.1), and includes utilities for tasks like I/O operations, threading and other functionalities like `net`, which provides interfaces to

work with network part. `Crate std` [1] page provides comprehensive and well-structured documentation for the standard library in a common „Rust Docs“ format.

As stated in Embedonomicon [5]: „`std` contains functionality that assumes the program will run on the operating system rather than *directly on the metal*. `std` also assumes that the operating system is a general purpose operating system, like those found in servers and desktops. For this reason, std provides a standard API for functionality typically found in such operating systems: threads, files, sockets, file system, processes, and so on.“

2.2.2 Rust for Bare-Metal Systems

`no_std` is an essential approach for creating applications that do not rely on the `std`. It becomes very useful in embedded systems, resource-constrained environments, or scenarios requiring direct hardware interaction. By opting for `no_std`, developers leverage the `core` functionalities of Rust without the overhead of `std`, which includes features dependent on an underlying OS [35].

The `core` library[4], which non-standard environments depend on, provides a subset of the functionalities found in standard library, focusing on features that do not require OS-level support. This includes basic types, arithmetic, iterators, and traits but excludes dynamic memory allocation, I/O operations, and concurrency, which are part of std, which means developers need to manage memory allocation and system initialization manually, and may need to implement platform-specific features, linker scripts and etc. Also there is a `Crate alloc`[5], which provides facilities for dynamic memory allocation, however platform-specific allocators are required, on top of which this the alloc crate will be used [35]. In case of Rust on Espressif chips, there is an `esp-alloc` crate, covered in 2.4.2, which fulfils the role and deals with the functionality of the allocator [10].

2.2.3 Summary

However, despite all the convenience and comfort provided by the standard library, more and more often drivers for microcontrollers and embedded platforms work in `no_std` and here's why:

- **Resource Efficiency**: Minimizes runtime overhead due to the absence of many standard functions and data types, which is very important for devices with limited memory and processing power, despite being more complex to implement.
- **Direct Hardware Access**: Allows closer control and interaction with hardware peripherals, essential in embedded systems.
- **Reduced Binary Size**: bare-metal approach can lead to smaller binary sizes, an important factor for systems with limited storage space.
- **OS Independence**: `no_std` can run in environments without a full-fledged operating system, aligning with the typical setup of embedded systems.

A great way to end this discussion is another quote from the author of the Ebedonomicon [5]:

[4]Core crate documentation: https://doc.rust-lang.org/core/
[5]Alloc crate documentation: https://doc.rust-lang.org/alloc/

„Because of these properties, a `no_std` application can be the first and/or the only code that runs on a system. It can be many things that a standard Rust application can never be, for example: The kernel of an OS, firmware, a bootloader.“

— *Jorge Aparicio*

2.3 Espressif Microcontrollers

Espressif Systems[6], established in 2008, has grown from a small startup in China to a leading figure in the IoT/AIoT and semiconductor industry on a global scale. The company is originally known for its innovative ESP8266 and ESP32 chip series, which, back in a days, made a sensation with how powerful, versatile and simultaneously cost-effective they are, relative to the competitors. Espressif's commitment to open-source technology has enabled developers worldwide to create a number of smart-connected devices, fostering a community of innovation and collaboration.

2.3.1 Chip Range

This section outlines the current suite of microcontrollers from Espressif Systems, detailing their key characteristics. The information is drawn from the company's official website and documentation [20], supplemented by personal insights and knowledge within the field of embedded applications development.

- **ESP32**: This platform can now be confidently referred to as a „classic“. It gave incredible popularity to Espressif Systems company, becoming a revolutionary microcontroller after the equally popular and still used, but no longer supported ESP8266. Two or one CPU core(s) Tensilica `Xtensa`[7] 32-bit microprocessor, 520 KB SRAM (some variations have 448KB ROM) , 2.4GHz Wi-Fi module, dual-mode Bluetooth (classic and BLE), SD card interface, high-speed SPI, UART, I2S and I2C).

- **ESP32-S2**: Following the success of the original ESP32, ESP32S2 chip was released with a high-performance single-core `Xtensa`-driven CPU and focus on ultra-low-power performance with a LP `RISC-V` core. Lacks Bluetooth, which differentiates it significantly from the ESP32.

- **ESP32-S3**: Offers dual-core `Xtensa` and low-power `RISC-V` processors, distinguishes itself with AI capabilities and enhanced security features, supporting both Wi-Fi and Bluetooth 5. Which makes this chip ideal when it comes to AIoT and smart-home devices, for example ESP32-S3-BOX[8] product. Currently, it is the last `Xtensa` chip released from Espressif.

- **ESP32-C3**: Stands out with its `RISC-V`[9] single-core CPU, marking a shift in architecture. It brings Bluetooth 5.0 into the low-cost and low-power domain. Also brings hardware acceleration for cryptographic algorithms. Espressif also released an

[6] Espressif Systems about themselves: https://www.espressif.com/en/company/about-espressif

[7] Xtensa architecture: https://www.cadence.com/en_US/home/tools/silicon-solutions/compute-ip/tensilica-xtensa-controllers-and-extensible-processors/xtensa-lx-processor-platform.html

[8] ESP32-S3-BOX showcase: https://www.youtube.com/watch?v=KGVOi1MrjbO

[9] RISC-V official page: https://riscv.org

`esp-rust-board`[10] that ran on the ESP32-C3 chip. In a way, there was a slight inaccuracy here, as Rust is supported on all of the company's released chips, but because of the naming, some part of the community assumed that it was only supported on this board.

- **ESP32-C6**: A successor of ESP32-C3 board with the same processor, but characterized by its support for IEEE 802.15.4[11] and Wi-Fi 6, first in a whole lineup.
- **ESP32-H2**: 32-bit `RISC-V` core on board, with support of IEEE 802.15.4 and Bluetooth 5, but no Wi-Fi module, designed for low power consumption, considered as a very suitable chip for `Thread`[12] end devices and border router, `Zigbee`[13] protocol communcation and `Matter`[14] brigde by combining it with some ESP SoC with Wi-Fi.
- **ESP32-C2**: New chip from the C-line with `RISC-V` processor and support for Wi-Fi and Bluetooth 5.0. This chip was made with a focus on price efficiency and for the most part is used by big customers (their list is private company data) to create various Smart-Home products, meaning if this chip is procured, it is procured by the thousands and there are not that many feedback or issues from the community on GitHub.

2.4 Rust on Espressif Chips

Following on from previous sections on the powerful Rust type system and its commitment to memory safety, and given the wide range of Espressif chips, the integration of Rust programming with Espressif hardware is a notable step forward in embedded system development and another step towards popularising the Rust language. This combination leverages the strict safety and efficiency principles inherent in Rust, along with the significant computational and connectivity features offered by Espressif chips. This collaboration not only promises to increase the reliability of embedded applications, but also opens up new opportunities for innovation in the IoT ecosystem.

When Espressif did decide to start supporting the Rust language, a separate team was created to do so. A separate organisation was created for it on GitHub – **esp-rs**. Which means Rust drivers are not under the main **Espressif**[15] organization.

2.4.1 Standard Library Support on Espressif Chips

Within the broader context of using Rust with Espressif microcontrollers, the **esp-idf-hal** project warrants a brief mention. Although this thesis focuses on `bare-metal` implementations, as the main focus of the esp-rs team at the moment is also the development of `no_std` implementation, understanding the scope and purpose of `esp-idf-hal`, which is `std` driver, adds depth to the overall understanding of Rust's integration with ESP32 hardware.

[10] Rust-board project: https://github.com/esp-rs/esp-rust-board
[11] IEEE 802.15.4 standard: https://standards.ieee.org/ieee/802.15.4/7029/
[12] Thread protocol: https://www.threadgroup.org/What-is-Thread/Thread-Benefits
[13] Zigbee protocol: https://csa-iot.org/all-solutions/zigbee/
[14] Matter protocol: https://csa-iot.org/all-solutions/matter/
[15] Espressif organization on GitHub: https://github.com/espressif

This project is part of the **esp-rs** initiative, aimed at providing a HAL for Espressif chips, utilizing the ESP-IDF[16]. Basically, this driver is a Rust-centric wrapper over the lower-level details provided by the **ESP-IDF**, offering Rust developers a more familiar and time-tested interface to work with [12].

It is also worth noting an interesting fact that perfectly describes the community of Rust and Espressif – at the current moment this project is entirely maintained by the community, which means that the employees of the **esp-rs** team only rarely make new commits there.

2.4.2 Bare-Metal Programming with Rust on Espressif Chips

This section will explore the primary Rust driver for bare-metal programming on Espressif chips, known as **esp-hal**, which is a central tool in the development of the project discussed. This project and `no_std` functionality is a priority for the esp-rs team – `esp-idf-hal` is a complete and working driver, but it still doesn't reveal the full potential of the Rust language and utilizes **ESP-IDF** functions anf entirely dependant on it, which makes projects rather „heavy".

This subsection will delve into the expansive ecosystem surrounding the **esp-hal** project, discussing the various modules that together enhance the functionality of the main driver, providing a comprehensive toolkit for development with Espressif's chips.

esp-hal

esp-hal is the Hardware Abstraction Layer (HAL) for Espressif chips [11] provided as a crate[17]. Currently supports these chips:

- ESP32 Series: ESP32
- ESP32-C Series: ESP32-C2 (aka ESP8684), ESP32-C3 (aka ESP8685), ESP32-C6
- ESP32-H Series: ESP32-H2
- ESP32-P Series: ESP32-P4 (Support for the ESP32-P4 microcontroller remains relatively limited at this time, which has led to the exclusion of support for this chip within the scope of the current project.)
- ESP32-S Series: ESP32-S2, ESP32-S3

Detailed description of the entire chip range is in Section 2.3.1.

Core features:

- Support for all peripherals available on each chip. Despite the fact that the `README` to the driver says „Drivers are currently implemented for a significant number of peripherals, but they have varying levels of functionality" [11], in practice, the driver is observed to be quite reliable and extensively implemented across a wide range of functionalities.

[16]ESP-IDF project: https://github.com/espressif/esp-idf/tree/master
[17]esp-hal crate: https://crates.io/crates/esp-hal

- This library is designed to be compatible with **embedded-hal** with its collection of traits representing common embedded functionalities, erasing the specific details and unifying access to core functionality. In the Rust community, it's common practice for driver developers, both microcontrollers and sensors, to use common traits from this crate, so for the most part libraries for external peripherals are platform-agnostic [2].

  Also historically speaking, for a long time the most used and stable version was `embedded-hal 0.2.7`. However, not long ago a version `1.0.0` was finally released. Thus, a huge number of drivers still work on `0.2.7`, but `1.0.0` is also gaining momentum. The **esp-hal** library offers compatibility for both current and past versions of a crate. This feature was particularly beneficial during the development of the project, as many of the fundamental sensor drivers relied upon are still utilizing version **0.2.7**.

- Support for LP Cores. **esp-hal** crate provides (yet limited) support for programming low-power `RISC-V` co-processors, found on ESP32-C6, ESP32-S2 and ESP32-S3 [11]. This functionality can be found in the **esp-hal** project[18], as well as the examples for it.

- Availability of informative documentation unique to each chip [14].

- Automation with the `xtask`[19] tool. It significantly simplifies the process of running examples, testing in UI, building documentation. An example of usage is demonstrated on Listing 2.13

```
# Running examples
cargo xtask run-example esp-hal esp32c6 hello_world
# Building examples for ESP32(useful for automated testing)
cargo xtask build-examples esp-hal esp32
# Locally build documentation
cargo xtask build-documentation esp-hal esp32s2
```

Listing 2.13: Xtask usage example [11].

- Pretty informative examples, `The Rust on ESP Book` [17] and Contribution guide, which makes it easier for newcomers to get into this development environment and gives clear guidelines on how to create own contributions.

- Code styling and formatting. The root folder of the repository contains a **rustfmt.toml** file, which is further discussed in a later section (see Section 4.6) and a shell script **pre-commit**, which, when run, shows all places that do not match the declared format for the repository. To make the **cargo fmt** command from this script fix everything itself, user just needs to remove the last pass of the „`--check`“ argument in the **pre-commit** script and execute `./pre-commit` once again.

espflash

The **espflash** tool is a pivotal component within the Rust ecosystem for Espressif chips, designed for compiling and deploying Rust applications to ESP hardware.

[18] esp-lp-hal: https://github.com/esp-rs/esp-hal/tree/main/esp-lp-hal
[19] xtasks documentation: https://docs.rs/xtasks/latest/xtasks/

It's a serial flasher utility for those devices, loosely based on esptool.py[20], which is a ESP-IDF tool also for serial communication with a board [16].

There are two versions of **espflash** available:

- As a command-line tool (Listing 2.14)

```
# Install tool
cargo install espflash
# Flash binary and monitor output
espflash flash binary.elf --monitor
```

Listing 2.14: Install and flash with espflash (CLI tool) [16].

- As a cargo extension (Listing 2.15)

```
# Install tool
cargo install cargo-espflash
# Build and flash the project (executed from the root)
cargo espflash flash
```

Listing 2.15: Install and flash with espflash (cargo extension) [16].

esp-alloc

This crate offers a simple `no_std` heap allocator for the processors from Espressif and supports all currently available [10] (see initialization in Listing 2.16). This unlocks one more component of the Rust language besides the **core** library – the **alloc**[21] crate. It makes it possible not only to use smart pointers and heap, but also to use **std** Rust data types such as `String` or `Vec`.

```
#[global_allocator]
static ALLOCATOR: esp_alloc::EspHeap = esp_alloc::EspHeap::empty();

fn init_heap() {
    const HEAP_SIZE: usize = 32 * 1024;
    static mut HEAP: MaybeUninit<[u8; HEAP_SIZE]> = MaybeUninit::uninit();

    unsafe {
        ALLOCATOR.init(HEAP.as_mut_ptr() as *mut u8, HEAP_SIZE);
    }
}
```

Listing 2.16: Global allocator initialization with esp-alloc [10].

esp-wifi

The **esp-wifi** crate serves as a comprehensive driver for managing Wi-Fi, Bluetooth, and ESP-NOW[22] communication on Espressif chips. Developed under the **esp-rs** project, it

[20] esptool documentation: https://docs.espressif.com/projects/esptool/en/latest/esp32/
[21] Crate alloc: https://doc.rust-lang.org/alloc/
[22] ESP-NOW protocol: https://www.espressif.com/en/solutions/low-power-solutions/esp-now

facilitates the integration of these communication technologies within the Rust ecosystem, making it an essential tool for developers working on communication features with Espressif's microcontrollers in a **no_std** environment. The list of supported methods is also located at the source [15].

The main **esp-wifi** driver also uses the supporting **esp-wifi-sys** directory located at the root of the repository. It provides pre-compiled drivers and generated bindings generated from the C header files from **ESP-IDF**.

Due to the fact that this driver is quite complex, very careful management of `features` is crucial, as activating a specific feature in the user's **Cargo.toml** unlocks particular functionalities. Additionally, passing special parameters to the compiler is necessary for the driver's correct operation. These aspects are examined and detailed further in the section discussing the project's implementation related to Wi-Fi and MQTT (see Section 3.6).

esp-pacs

This library houses Peripheral Access Crates (PACs) for every Espressif SoC available. These PACs are the lowest level of abstraction over raw registers and microcontroller peripherals, which wraps them into accessible data structures, providing driver developers with the necessary interfaces to build higher-level abstractions. They are generated using the **svd2rust**[23] tool from Espressif **SVD**s [13].

The project is structured to allow for adjustments or corrections to **SVD** files through **patch** files in „`.yaml`“ format. When modifications are required, these patches can be applied using the **xtask** tool (see footnote 19), regenerating the entire **SVD** for the specific chip that was altered.

2.4.3 Peripherals Configuration in esp-rs Drivers

In the context of the **esp-rs** ecosystem, peripheral configuration and management are handled through a structured and systematic process. This process begins with the data structures that are generated from SVD files within the **esp-pacs** crate. These data structures form the foundation upon which the **esp-hal** crate operates. Here, a singleton pattern is employed to instantiate a **Peripherals** structure. Access to the hardware peripherals is then obtained by „taking“ this singleton at the start of the program using the „`Peripherals::take();`“ command. This approach is standard in embedded applications, providing a controlled mechanism to ensure that peripheral access is managed safely and efficiently, preventing concurrent access conflicts.

[23] svd2rust tool: https://github.com/rust-embedded/svd2rust/

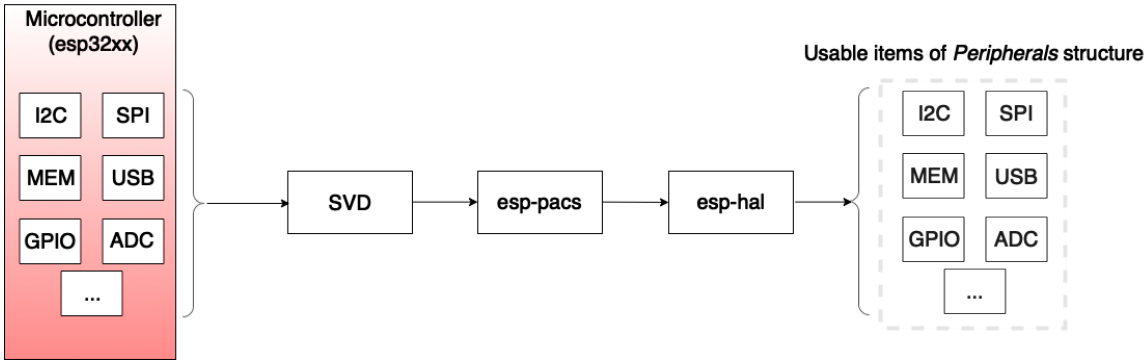


Figure 2.1: Architecture for operating on peripherals (on basis of [11]).

As part of the answer to one of the questions posed in this thesis, it is worth noting that the **esp-hal** driver itself, as well as none of the elements of the **esp-rs** infrastructure, does not support external peripherals and focuses only on the internal peripherals of a particular chip, as shown the the Figure 2.1.

Chapter 3

Design and Implementation

This section will introduce a developed solution aimed at simplifying the task of programming embedded devices with Rust. It will present an analysis of existing projects already in place to address this challenge, describe the structure and reasoning behind the suggested crate, and detail particular implementation strategies. Additionally, the text will guide the reader through the use of the library, cover both successful ideas and those not used because they were ineffective, and discuss updates made to existing drivers for peripheral devices. The section will also cover illustrative examples and the approach to automated testing.

The purpose of this part is to clarify the designs of the project, the difficulties encountered, and the the methods used to overcome these difficulties.

3.1 Existing Solutions

In order to offer a really new and valuable solution to the market, it is essential to conduct a thorough review of existing products that address the problem at hand. Drawing from the strengths and weaknesses of current offerings, an improved vision for tackling this challenge can be proposed.

esp-bsp-rs: This repository on GitHub offers Rust Bare Metal Board Support Packages tailored for ESP32-based boards, with an emphasis on compatibility with the `Embassy Async`[1] runtime for embedded applications in Rust [32].

The main functionality of this library is to provide an interface for initializing displays for specific devkits, including **ESP32-C3-LCDKit**[2], **M5Stack**[3] or **ESP32-S3-Box**[4]. In this crate there are macros that return prepared pins to the user depending on the configuration and used board, as well as macros that return the type of a specific display for specific devkits, which is a nice relief because due to the security of this language, a strict and specific data type is required with all generic parameters provided. Positive and negative sides of this solution are listed in Table 3.1.

[1] Embassy embedded framework: https://github.com/embassy-rs/embassy

[2] ESP32-C3-LCDKit: https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32c3/esp32-c3-lcdkit/user_guide.html

[3] M5Stack product: https://m5stack.com

[4] ESP32-S3-Box(discontinued): https://docs.platformio.org/en/stable/boards/espressif32/esp32s3box.html

Pros	Cons
• High modularity within the code, facilitating easy updates or changes to board configurations. • Supports a range of configurations, promoting scalability across different hardware platforms. • Organized and maintainable code using Rust's enums, structs, and macros effectively.	• Complexity due to extensive use of macros, which may be difficult for new users to understand. However, there is a reason for this and there is no other option in Rust to declare and work with such types, or use pins that are part of a larger peripherals structure(see Section 5.3). • Support for only a narrow range of selected devkits and only for two displays (deduced from the code [32]).

Table 3.1: Pros and Cons of the **esp-bsp-rs** crate.

The **OxidESPark** project on GitLab is a Rust-based initiative for the Rust ESP Board, which has been already mentioned in 2.3.1, based on the ESP32-C3 microcontroller. Unlike crate, which was observed above, this one is based on the `esp-idf-hal` driver (see 2.4.1), which depends on `ESP-IDF` [30]. However, the concept and problem that this library solves corresponds to the topic of this research paper, so there is no reason why it should not be analyzed as well.

It supports both internal microcontroller sensors (**SHTC3**[5] and **ICM-42670-P**[6]) and external devices, potentially covering a range of sensors and peripherals commonly used in IoT applications. At the current stage only **TSL2561**[7] and **WS2812B**[8] LED strips are supported, **BME280**[9] and **MPU9250**[10] are coming soon. Planned features are aimed at extending this support, integrating more functionality and wider device compatibility. For the most part, the project also utilizes already existing libraries for sensors.

This library opens the possibility for the user to customize his project with a separate file in „`.toml`“ format, the specified parameters from which will later be applied to the project [30].

[5]SHTC3 temperature and humidity sensor: https://www.sensirion.com/products/catalog/SHTC3/

[6]ICM-42670-P accelerometer and gyroscope sensor: https://invensense.tdk.com/products/motion-tracking/6-axis/icm-42670-p/

[7]TSL2161 visible and infrared light sensor: https://ams-osram.com/products/sensors/ambient-light-color-spectral-proximity-sensors/ams-tsl2561-ambient-light-sensor

[8]WS2812B RGB LED strips: https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf

[9]BME280 temperature, humidity and pressure sensor: https://www.bosch-sensortec.com/products/environmental-sensors/humidity-sensors-bme280/

[10]MPU9250 accelerometer, gyroscope and magnetometer sensor: https://invensense.tdk.com/products/motion-tracking/9-axis/mpu-9250/

Pros	Cons
• Highly modular structure, allowing for flexible development. • Supports not only internal Rust Board sensors, but also external devices, which expands the range of covered use-cases. • Basic Wi-Fi and MQTT support. • Configuration via external `toml` file. • The example in the repository covers all use-cases that this library simplifies, although there is only one of them	• Focus on one specific chip. • While the library currently supports a range of external peripherals, not all built-in sensors are yet accommodated. • Using the **std** environment, while embedded development tends towards **no_std**. • Having the ability to choose between configuration directly in the code or via `toml` file could be a great feature.

Table 3.2: Pros and Cons of the **OxidESPark** crate.

It's worth noting that the project is still a work in progress and the author makes private commits quite often and has some experiments in branches besides main [30]. Structured pros and cons of this solutions are shown in Table 3.2 above.

Also worth mentioning is a project related to the Espressif Systems **ESP-IDF** core driver that served as the inspiration for **esp-bsp-rs** – the **esp-bsp** project. This is where the original idea of supporting and simplifying the work with different Espressif devkits and their features came from, providing higher level of abstraction for working with peripherals such as cameras, touchscreens, audio, card slots, etc [18].

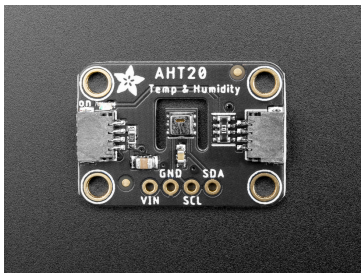
This driver will not be discussed in detail, because it is rather indirectly related to the problem of simplifying the development of embedded applications in Rust on Espressif platforms, which is addressed in this research, being only an example of the fact that such a problem is also addressed in C drivers from Espressif Systems.

3.2 Default Set of Proposed Peripherals

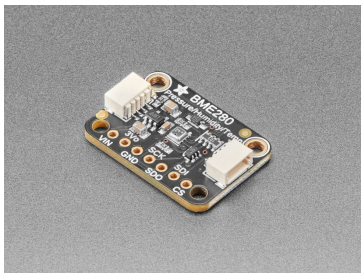
This section will describe the set of sensors that have been selected for default support. The word „default“ is used because the library has been designed and implemented to be easily extensible, and when the library is released it will be open to further extensions from both the developer and the community.

3.2.1 Environmental Sensors

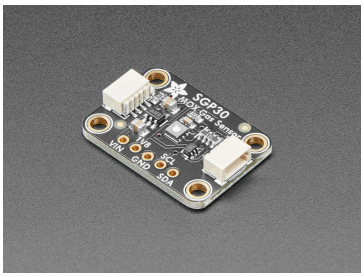
Environmental sensors are an integral part of most embedded applications, so when these sensors were selected, the intention was to cover as many types of measured data as possible: temperature, humidity, lighting, air pollution and so on. Set of chosen sensors is shown on Figure 3.1.



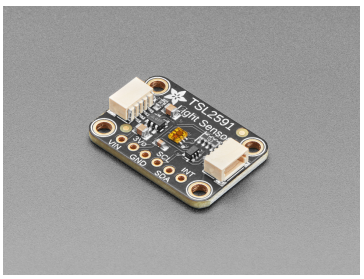
AHT20



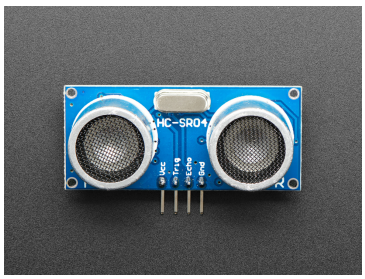
BME280



SGP30



TSL2591



HC-SR04

Figure 3.1: Used environmental sensors [4].

- **AHT20**: Cost-effective I2C temperature (°C) and relative humidity (%RH) sensor [4].
- **BME280**: A sensor from Bosch, belonging more to the medium-high price category. It can operate on both SPI and I2C buses and is capable of measuring temperature (°C), barometric pressure (hPa) and air humidity (%RH) [4].
- **SGP30**: I2C-driven sensor for getting information about TVOC (PPM) and eCO2(PPB) in the air. However, it is important to know that this sensor is not suitable for use in laboratories and other ultra-precise calculations [4].
- **TSL2591**: Ultra-high-range (600,000,000:1) luminosity sensor (lux) also transmitting data via the I2c bus, however, changing the address, unlike all previous sensors will not work, so user should be careful when using with other peripherals so that it doesn't come to address interference [4].
- **HC-SR04**: A sensor that allows to measure distance. The best result is achieved at a distance of 10-250cm from the target. This sensor does not work on any bus, using the simplest connection via GPIO pins. It works on the principle of echolocation – one speaker emits a wave (trigger), the second speaker waits for the reflected wave (echo) and with the help of simple calculations from the difference in time, sound speed and temperature of the surrounding environment the final distance to the object from which the sound wave was reflected is obtained [4].

$$V = 331.3 + (0.606 * T) \tag{3.1}$$

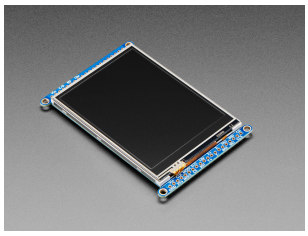
$$S = V * \Delta t \tag{3.2}$$

where V is a speed of sound in the environment, T – temperature in the environment, S – distance to the object and Δt is the time difference between the emission and reception of the reflected wave. Constants: `331.3 m/s` is a base speed of sound in air at 0 degrees Celsius and `0.606` is the increase of speed of sound per degree Celsius [9].

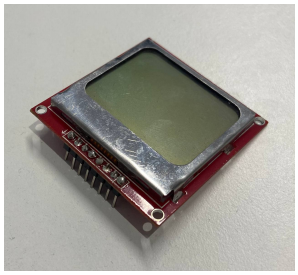
3.2.2 Displays

In some use-cases it is also important to be able to deduce information not only on the terminal of PC connected to the board, but also on some displays, so it was decided to introduce support for displays shown on Figure 3.2.

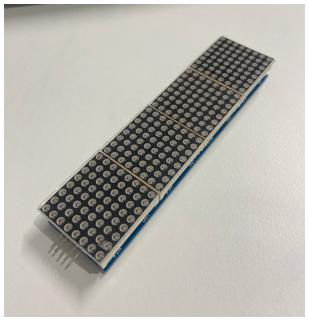
- **ILI9341**: Driver for a 2.8/3.2 inch RGB TFT LCD display with 240x320 pixel resolution, operating via SPI bus [24]. In Espressif Systems products it is used as part of the extension board for ESP32-S2-Kaluga-1 Kit
- **MAX7219**: This driver allows 64 Dot LED 8x8 display modules with only 16 pins through multiplexing [4].
- **PCD8544**: Classic monochrome LCD display used in various Nokia phones. Often used in battery powered devices due to its low power consumption. Be sure to use this display with a 5V power supply output [21].



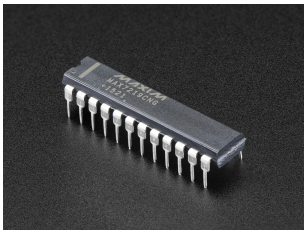
ILI9341 [4]



PCD8544



Chain of 8x8 displays with MAX7219

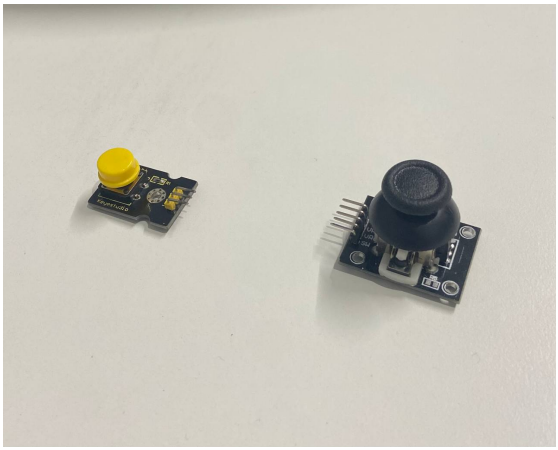


MAX7219 driver [4]

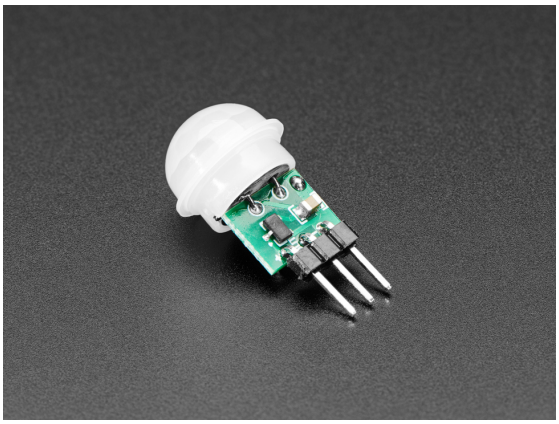
Figure 3.2: Used displays.

3.2.3 Miscellaneous

- **PIR Sensor**: Motion sensor is the simplest, yet quite useful element for smart home systems, security systems or other embedded applications. The simplest sensor is controlled by reading a single data pin.
- **Joystick**: An analog 2-axis joystick is also a common element in such use-cases, so it was decided to include its support in the library as well.
- **Button**: Button support and simplification should be part of any such API, as this crate provides. Meaning a normal button with one data contact.



Common button and joystick



PIR Sensor [4]

Figure 3.3: Used miscellaneous peripherals.

3.3 Conceptual Design for Simplified Peripheral Operations

The design of this library strategically utilizes Rust's advanced features to streamline the interaction with peripherals. By implementing traits, the crate defines a set of common behaviors that various peripherals must adhere to, promoting a uniform interface for peripheral management. This abstraction allows users to interact with different hardware components through a consistent set of high-level functions, abstracting the complexities associated with direct hardware manipulation.

It will also be a task to bypass such peripherals that do not fit any of the generalization traits, requiring more or different input parameters. Potential difficulties also include the appearance of possible generics, which, although a very useful element of the language, can cause a lot of problems when trying to satisfy the compiler.

A key element of all such libraries and frameworks is to provide an intuitive, simple, and memorable API for peripherals, be it a display, sensor, or button. That is what this crate will strive for as well.

3.4 Peripheral Management Architecture

This subsection describes a general approach to peripheral management in the library, both internal on ESP device and external supported peripherals (see Section 3.2) designed to take the complexity out of device management for end users. Users interact with peripherals using high-level functions or macros, eliminating the need to manage complex details such as pin configuration or device-specific protocols.

3.4.1 On-board Peripherals

The root file of the library (*esp-ward/src/lib.rs*, see project structure 3.8) contains most of the functionality related to working with internal peripherals of ESP32 devices. Here there are both primitive simplifications, for example macro „`take_periph`“ and „`take_system`“, which exist in order to reduce the complexity of the syntax of these operations itself, and also macros aimed at providing a shorter and more complex ones, like simplified initialization of `SPI` and `I2C` structures necessary for the development of embedded applications or general chip initializing macro „`init_chip`“.

The above-mentioned „`init_chip`“ macro serves as a utility to initialize the essential hardware components required for operations. It takes identifiers `peripherals` and `system` as inputs, and uses them to initialize the clocks to their default settings, configure GPIO pins, and create a delay object based on the system clocks. It returns a tuple of three initialized components, making them readily available for immediate use in the system's runtime environment. This macro also imports the necessary traits from **embedded_hal** (Section 3.5.1) for the correct operation of the delay into the program of the end user [2].

```
let peripherals = esp_ward::take_periph!();
let system = esp_ward::take_system!(peripherals);
let (clocks, pins, mut delay) = esp_ward::init_chip!(peripherals, system);
```

Listing 3.1: Initial initialization using the library.

As demonstrated on Listing 3.1, it was decided to return the delay source as well due to the fact that it is used quite often in applications. However, if there is no need for it, the user can simply put the „_“ sign instead of assigning it to the „delay“ variable, which will satisfy the Rust compiler.

Two initialization variations, default and custom, were implemented for I2C and SPI peripherals, see Listing 3.2. In the default variation, the program instances of the peripherals are designed to fit most possible configurations of both the chips themselves and the external peripherals used, so as not to create interference with, for example, ADC pins (see Section 3.5.2)

```
let i2c_bus = esp_ward::init_i2c_default!(peripherals, pins, clocks);
let spi_bus = esp_ward::init_spi_default!(peripherals, pins, clocks);
```

Listing 3.2: Default I2C and SPI initialization.

These macro calls will return an I2C instance, where:

- `GPIO6` – SDA pin
- `GPIO7` – SCL pin

and an instance of SPI, where:

- `GPIO0` – SCLK pin
- `GPIO4` – MOSI pin
- `GPIO2` – MISO pin
- `GPIO5` – CS pin

It is also possible to create a completely custom structure on the pins and frequency that the user will require in a way, shown on Listing 3.3.

```
let i2c_bus =
    esp_ward::init_i2c_custom!(peripherals, &clocks,
    pins.gpio21, pins.gpio22, 100u32.kHz()
);

let spi_bus =
    esp_ward::init_spi_custom!(peripherals, clocks,
    pins.gpio18, pins.gpio23, pins.gpio19, pins.gpio5,
    100u32.MHz()
);
```

Listing 3.3: Custom I2C and SPI initialization.

The knowledge to implement these macros was obtained from the **esp-hal** library [11].

Also in the root file of the library is a macro that handles the preinitialization of the allocator, which is required for some applications. This functionality is represented in the macro `prepare_alloc`. Example is demonstrated on Listing 3.4. It should be called once and only once in the program at the very beginning, even before initializing the chip itself.

```
esp_ward::prepare_alloc!();
let peripherals = esp_ward::take_periph!();
let system = esp_ward::take_system!(peripherals);
let (_clocks, pins, delay) = esp_ward::init_chip!(peripherals, system);
```

Listing 3.4: Allocator preparation example.

The concept and knowledge for this functionality was taken from the library **esp-alloc** [10].

3.4.2 External Peripherals

The „*esp-ward/src/peripherals/mod.rs*“ (see project structure 3.8) directory serves as a central module for managing external peripherals. This file structures the integration and management of these devices through Rust traits, which define common interfaces for various types of peripheral interactions. These traits facilitate the implementation of standardized methods for operations like reading from or writing to peripherals, ensuring consistency across different hardware components. The design emphasizes modularity and reusability, allowing developers to extend functionality without altering the core system, thus streamlining the development process for applications involving diverse external devices.

The library emphasizes adhering to conventional naming conventions in order to create an intuitive API. Thus, the following general standards for working with peripherals have been defined:

- `create_on_<bus>` – functions and macros initializing any sensor or other external peripheral adhere to this standard. `<bus>` can be „`i2c`“, „`spi`“ or „`pins`“ in case the periphery does not operate on any bus but shares information directly via pins.

- `get_<data>` – functions that are used to retrieve data from a particular device. `<data>` is the type of the returned metric. This logic also applies to the Wi-Fi functionality (except for the MQTT) provided by this crate, this is covered in section 3.6.1.

For example, the `I2cPeriph` trait defines a method „`create_on_i2c`“, which standardizes the creation of peripherals on the I2C bus, emphasizing ease of use and consistency. This trait definition is shown on Listing 3.5. The same applies to the trait `SpiPeriph` – it performs the same functionality, but for peripherals working on the SPI bus.

```
pub trait I2cPeriph {
    type Returnable;
    fn create_on_i2c(
        bus: I2C<'static, esp_hal::peripherals::I2C0>,
        delay: Delay,
    ) -> Result<Self::Returnable, PeripheralError>;
}
```

Listing 3.5: `I2cPeriph` trait definition.

Here the flexibility that the Rust language allows can be seen. A **Returnable** type has been declared within the trait itself, which ensures that when implementing this trait, the developer will be able to declare that the instantiating method will return some custom data type. This unleashes the unification of methods.

It was decided not to create trait **NoBusPeriph** for peripherals that do not operate on any bus, but communicate with the chip via GPIO communication, because unification of initializing function parameters will not be able to cover most cases, due to the fact that different peripherals may require different amounts and types of input data. In view of this it was decided to keep the name convention and use for this kind of peripherals „`create_on_pins`“, but not to create any strictly defined trait for this.

This module also defines a set of traits that describe the behavior of sensors of specific environment values, such as temperature, humidity, pressure, illumination, CO2 concentration. This will allow flexible implementation of traits for peripherals with different purposes and, as in the case of **BME280** (see 3.2.1), will also allow convenient handling of sensors that combine measurements of different types of data.

```
pub trait TemperatureSensor {
    // Reads the temperature in degrees Celsius
    fn get_temperature(&mut self) -> Result<f32, PeripheralError>;
}
```

Listing 3.6: Example of type-specific measurement trait.

The `UnifiedData` trait in this library is designed to provide a standardized way to get data from all readable peripherals. This trait includes a `read` method that takes a **Delay** object as an argument (which is a type from **esp-hal** and will be retrieved by the `init_chip` function, see 3.4.1) and returns a **Result** containing the serialized data retrieved from the peripheral. This trait allows to declare what type of data will be extracted from the device (see 3.4.2) and return at once all data received from the sensor serialized to the form of tuple.

In the crate, all functions that interact with peripherals return a **Result** type. In the event of an error, when `.unwrap()` is called on the result, it will return a standardized error type from the `PeripheralError` enum. This approach ensures that error handling is consistent across the library, allowing developers to more easily manage and debug issues related to peripheral operations.

These traits encapsulate specific behaviors and operations, making them essential for ensuring that peripheral interactions are both efficient and error-resistant, which is crucial for reliable embedded systems development.

3.5 A Closer Examination of Peripheral Operations

This section offers a thorough examination of peripheral operations within the library, going beyond basic usage to explore the detailed configuration and management processes. It will highlight the techniques and internal mechanisms used to interact with devices such as sensors, buttons, joysticks and displays, providing insights into the library's handling capabilities. This closer look aims to equip reader with a nuanced understanding of peripheral integration and optimization in their embedded system projects. The methods of implementing support for certain peripherals can be divided into several groups and certain devices can be assigned to them. This is discussed in the following sections.

3.5.1 Sensors

The concept of implementing support for the environmental sensors that have been listed in 3.2.1 (with the exception of HC-SR04) is almost identical. It will be explained on the example of BME280 sensor as the most complex of the presented devices in terms of the number of provided metrics.

First step is to find the right crate in the official package registry of the Rust language – **crates.io**, which was covered in 2.1.1. Finding the right library can be a major challenge because many of them are either abandoned or made only for the **std** environment, which is what this project encountered when supporting the `MAX7219` display (see 3.5.3). The best place to start looking for a suitable crate is based on the date of the last update – these drivers are most likely to support current versions of **embedded-hal** (see, through which the ESP32 line chip will communicate using **esp-hal** (see 2.4.2) and my library). The second and most important criterion would be whether the driver for the desired sensor operates in **no_std**. After selecting the right driver – proceed to its study, often many of them have excellent documentation, in a uniform format for all Rust libraries, so that even if the driver author has not prepared examples, after some time spent on studying the driver, the algorithm of working with it can be built. In this case, a suitable driver[11] was found quite quickly, so the next step could be initiated.

First of all, it is necessary to implement the initialization function within the trait for the bus selected for communication. In the case of `BME280`, this sensor can operate on both `I2C` and `SPI`, but at the moment I2C was chosen because after many attempts it was found out that there is a conflict between the structures in the driver for the sensor and esp-hal

[11] BME280 rust crate: https://docs.rs/bme280/0.5.1/bme280/

within SPI, more about it in the Section 5.3 devoted to testing, problems encountered and evaluation .

Wrapper Structure

Next, an additional new structure must be created that wraps the original structure provided by the driver for the sensor. This is done so that:

- Have all the data needed for the sensor in a single data unit (e.g., delay source).
- One of the goals of the project is to unify access to peripherals, so it is important that the interface to all peripherals and sensors can be obtained by importing only one my library into the user's code, rather than all libraries associated with these devices.
- The names of these structures should have the same format.
- Most sensor structures require generics. If wrapper structure is used, the „default" value will be passed to that generic, shielding the user from working with this complex element.

```
use bme280::{i2c::BME280 as ExternalBME280_i2c};
pub struct Bme280Sensor {
    /// The internal BME280 driver from the ‘bme280‘ crate used over I2C.
    pub inner: ExternalBME280_i2c<I2C<’static, esp_hal::peripherals::I2C0>>,
    /// A delay provider for timing-dependent operations.
    pub delay: Delay,
}
```

Listing 3.7: Wrapper structure.

As can be seen, the `BME280` structure from the original driver for this sensor requires having the **I2C** type as a generic parameter – the full type is statically declared in the newly created wrapper.

The `inner` field within the `Bme280Sensor` structure encapsulates the device's interface, specifically using an I2C type statically linked to the **I2C0** peripheral. The choice of **I2C0** as the default peripheral across ESP chips is strategic: while some ESP models feature a second **I2C1** peripheral, all are equipped with basic **I2C0**, making it a universally applicable option [13]. This standardization simplifies the design, ensuring compatibility and reducing complexity in configuration processes. Moreover, the examples in **esp-hal** also always use **I2C0** [11]. Using a „`’static`" reference ensures that the I2C interface remains valid for the lifetime of the sensor object, preventing issues related to dangling references or invalid hardware access in embedded systems programming.

Also, after examining the driver for this sensor, it turns out that a `delay source` will be required to measure the data – this is also included in the wrapper structure on Listing 3.7.

All this makes this structure usable and suitable for the logic of operations with the internal peripherals (see 3.4.1) of the Espressif chip and allows to work on simplification further.

Initialization

The next step is to implement the trait that is responsible for initializing the sensor on this bus, the process is demonstrated in Listing 3.8. In this case, it is `I2cPeriph` (see 3.4.2). Using the knowledge obtained from the driver for this sensor, the output is a similar function for sensor instantiation.

```
impl I2cPeriph for Bme280Sensor {
    type Returnable = Self;

    fn create_on_i2c(
        bus: I2C<'static, esp_hal::peripherals::I2C0>,
        mut delay: Delay,
    ) -> Result<Self::Returnable, PeripheralError> {
        let mut sensor = ExternalBME280_i2c::new_primary(bus);
        match sensor.init(&mut delay) {
            Ok(_) => {}
            Err(_) => return Err(PeripheralError::InitializationFailed),
        }
        Ok(Bme280Sensor {
            inner: sensor,
            delay: delay,
        })
    }
}
```

Listing 3.8: Peripheral initialization trait implementation.

Using pattern matching, an error of type `PeripheralError` is returned (see 2.1.1 and 3.4.2) or, if successful, the function continues its process and returns the above defined structure with the sensor instance inside. Thus, all that remains is to create an API for operations on it.

Measurements

Then the final step remains to provide the API for the selected sensor to the end user – the implementation of metric-related traits for this new structure. It is necessary to select one of the provided traits, which is suitable for this device and write the body of the **getter** function for this metric declared by a trait. In this particular case the implementation only for temperature is demonstrated on Listing 3.9, other implementations can be found in the files dedicated to sensors (see project structure in 3.8).

```
impl TemperatureSensor for Bme280Sensor {
    fn get_temperature(&mut self) -> Result<f32, PeripheralError> {
        match self.inner.measure(&mut self.delay) {
            Ok(measurement) => Ok(measurement.temperature),
            Err(_) => Err(PeripheralError::ReadError),
        }
    }
}
```

Listing 3.9: Metric trait implementation.

This uses the `measure` method from the library for this sensor, which returns `Result`. Instead of the typical `unwrap`, a `match pattern` is used here, so that if the temperature is returned successfully, it returns this value „wrapped“ in `Ok` (so that the data type matches the `Retrurn` type declared in the returned value of this function), and in case of any error (the „_“ sign), it returns `Err` with an error packed inside, standard for the library being created (see 2.1.1 and 3.4.2).

This completes the addition of support for common environmental sensors. All the libraries used to support these sensors can also be found in the **Cargo.toml** file, the library names are unambiguous and unique and can be found by name on **crates.io**.

3.5.2 Input Devices

The following peripherals were allocated to the Input Devices category: PIR sensor (motion sensor), button, and joystick. This subsection will describe how the API for these components, which are important in many embedded applications, was developed.

PIR sensor

It was decided to write an interface for this sensor and not to use ready-made libraries, as this is a rather simple peripheral and this approach will reduce the number of imported libraries, respectively, and the size of the binary. The **PIR** module (*esp-ward/src/peipherals/pir.rs*, see project structure in 3.8) in my library is designed to abstract and simplify the interaction with typical PIR motion sensors that transmit data by a single pin, described in 3.2.

The module defines a `PirSensor` structure that encapsulates the input pin used for motion detection. It was decided to use the **InputPin** trait from the **embedded-hal** library (version 0.2.7, see 3.5.1), since the pin structures from **esp-hal** implement it. This structure is initialized with a specific GPIO pin configured as a pull-up resistor input using the **create_on_pins** function (see 3.4.2). The only operation that must be performed to make it work correctly, make it to not cause critical errors during program run and satisfy the compiler is to override the error type in **InputPin** with `core::convert::Infallible`, according to what is used in the **gpio** module in **esp-hal** [11]. This is demonstrated on Listing 3.10.

```
impl<PIN: InputPin<Error = core::convert::Infallible>> PirSensor<PIN> {
    pub fn create_on_pins(pin: PIN) -> Self {
        PirSensor { inner: pin }
    }
```

Listing 3.10: Redefining the Error type in generic argument.

The same should be done in all peripherals requiring pins from the ESP chip.

Motion detection is done using the `read` method, part of the **UnifiedData** trait (see 3.4.2), which checks the state of the pin and returns `true` if motion is detected (input pin is in `high` state), otherwise `false`.

Button

The **Button** driver in this library (*esp-ward/src/peipherals/button.rs*, see project structure in 3.8) contains functionality to control button inputs with debugging logic.

It defines an `Event` enumeration with states `Pressed`, `Released`, and `Nothing` to represent possible button events. The **Button** structure contains an input pin (see reference in previous **PIR sensor** sub-subsection 3.5.2) and tracks the state of the button in **pressed** item of a structure.

The debouncing mechanism provides a reliable readout of the button state by introducing a fixed delay after detecting a state change, and then checking the state again to confirm the change, which reduces false positives due to noise. This setup enables accurate detection of high-level events through method calls, simplifying user interaction with hardware buttons. The algorithm was implemented by me personally, with inspiration drawn from an article by The Ganssle Group [23]. It was also tested on examples I created on Wokwi.com[12] [33].

To retrieve data from a button in code using this driver, user would typically utilize the `read` method provided by the `UnifiedData` trait implemented for the `Button` structure. This method includes debouncing logic function to ensure the button's state is stable before a read operation is performed.

Joystick

This module (*esp-ward/src/peipherals/joystick.rs*, see project structure in 3.8) is designed to facilitate intuitive control over a 2-axis joystick with a built-in select button described in 3.2 and shown on Figure 3.3, configured specifically for ESP devices.

The **Joystick** structure stores and utilizes dedicated ADC pins for the X and Y axes, carefully selected based on their general availability and uniformity across different Espressif chip models, based on information obtained from the official Espressif Systems documentation [20] and the **esp-hal** driver [11]. ADC pins were also selected for each chip specifically so that there is no interference with other default GPIOs, e.g. in „default“ I2C or SPI configurations, analyzed in 3.4.1. Structure also stores a custom pin connected to `select` pin. This ensures broad compatibility with other parts of driver and simplifies setup.

Due to the specific architecture of the **esp-hal** and its ADC driver, the initialization of the joystick and its ADC settings is implemented using a macro instead of a function. This macro approach avoids the potential „partial move“ error in Rust, which can occur when trying to access parts of the **Peripherals** structure multiple times (see 5.3). The `create_joystick` macro effectively bypasses this limitation by allowing multiple accesses to the **Peripherals** in one scope, thus enabling dynamic configuration of ADC pins during runtime.

In addition to managing the joystick's axes through ADC pins, it also incorporates a **Button** structure (see previous sub-subsection 3.5.2) to handle the select button functionality.

[12] Wokwi.com – IoT simulations in browser: `https://wokwi.com`

This driver setup does not conform to the standard peripheral traits like `UnifiedData`, mainly because these operations require access to the ADC for reading pin voltages, which does not align well with the static trait method signatures designed for more general use cases. Despite this, the API and naming for the functions was made in line with the rest of driver and can be found in the detailed Rust format documentation for this library, presented in 4.2.

However, there is a significant problem with ADC – this peripheral calibration is only available on a limited number of Espressif microcontrollers, which means that the user's joystick will either have to be „equalized“ with an auxiliary resistor or the final application of the potential user will have to work according to how the ADC works without calibration. Despite this, this module works correct and also introduces the `ROUGH_THRESHOLD` constant, which represents the threshold for side-to-side transition on any axis under ideal conditions. This will be used as part of the example on the aforementioned Wokwi.com simulator when analyzing performance results in 5.1.

HC-SR04 (ultrasonic distance sensor)

Last sub-module in the **peripherals** provides an interface for ultrasonic distance sensor HC-SR04. It operates by triggering ultrasonic pulses via an output pin (trigger) and measuring the time until the echo is received via an input pin (echo), stored in the structure **USDistanceSensor**, which also includes a delay source to manage timing.

The initialization method `create_on_pins` prepares the sensor by setting the trigger pin low initially. The `get_distance` method sends a pulse and calculates the distance based on the echo delay, factoring in the ambient temperature to adjust for variations in sound speed, using algorithm and equation covered in 3.2.1.

None of the existing crates were suitable for the chosen environment, so it was decided to implement a driver for this independently. Final driver does not implement the **UnifiedData** trait due to the need for an ambient temperature parameter, which does not conform to the trait's method signature.

During the implementation of this driver, a system timer source was needed to get the Δt (see equation in 3.2.1). Private constants for this are also declared. While all other ESP line microcontrollers have it, it is not available on ESP32. Therefore it was necessary to use the esp-wifi driver and its `current_millis` function, which is an analog of this function. Therefore, in order to use this sensor on the ESP32 chip, the feature „`esp32-wifi`“ (see features configuration in 4.5.1) must be activated. Yes, this will increase the size of the binary, but the sensor will become available on that chip with the simplicity of the API preserved, which I considered to be a fairly equivalent exchange [15, 11].

3.5.3 Displays

The **display** (*esp-ward/src/display* directory, see project strucutre in 3.8) module in the presented software architecture provides interaction with different types of displays using a structured set of traits. It includes a `Display` trait for basic display operations such as setting pixels and writing text, and an `EGDisplay` trait that integrates with the **embedded-graphics** [36] library for advanced and more flexible graphics capabilities. This

customization allows complex drawing operations and text rendering using different fonts.

The **DisplaySegment** enum categorizes the sections of a display where text or graphics can be specifically targeted. This enumeration includes `TopLeft`, `TopRight`, `BottomLeft`, `BottomRight`, and `Center`. Each of these values represents a specific area on the display, allowing for localized control over where content is rendered. This is particularly useful for applications needing to organize information spatially or enhance the user interface by aligning content precisely within designated areas of the screen. In terms of default displays, it is used in **ili9341** module below in text.

The specific display drivers are organized into submodules with peripherally appropriate names, each for a specific display, such as the ILI9341, MAX7219, and PCD8544, covered in 3.2. Each display type supports different initialization parameters, so a universal constructor function is not possible, but specific constructors such as `create_on_pins` or `create_on_spi` (depending on the specific device) are provided for each display type to meet their initialization requirements.

These features and the modular design of the drivers enable flexible and scalable integration of displays. Which enhances the usability and customizability of display functions in embedded systems using this library.

MAX7219

The MAX7219 module facilitates operation of MAX7219 LED matrix displays (see 3.2) by performing tasks such as setting individual pixels on LEDs and displaying scrolling text. The driver is built around **OutputPin** traits from **embedded-hal** [2] for data pins, chip select, and clocking, allowing direct interaction with the hardware. Initializing the display involves setting up these pins and configuring the number of daisy-chained devices in the `create_on_pins` signature function.

Trying to find a suitable crate for this peripheral was unsuccessful: some of them worked only in the **std** environment, others were completely customized for other companies' chips and architecture. Therefore it was decided to take as a basis a driver made by Philipp Schuster [39] for another platform and modify it in such a way that it clearly fits the **esp-rs** infrastructure. Also during the analysis of this driver it turned out that although the author writes in the description that crate supports **no_std**, in reality it turned out that the code is not adapted to it.

I started the driver redesign by making a fork of the author's repository to make it clear that this project was not written by me from scratch, but only significantly improved.
First of all, it was decided to make this crate really **no_std**. Fortunately, it turned out that from the **std** environment the author used only the `sleep` function and the `Duration` data type, which is intended for the aforementioned function [39, 1]. After modifying all the places in the code where sleep was used and replacing it with conditions suitable for ordinary `Delay` from **esp-hal** driver, using traits from **embedded-hal**, the library became fully usable as bare-metal [2]. Then all references and attributes related to **std** were removed.

The next step was to add **esp-alloc**, covered in 2.4.2, to the list of project dependencies, because allocation functions used for dynamic memory management when working with displays requires a custom allocator for a particular platform (see 2.2.2). All functionality related to this has already been implemented by Mr. Philipp Schuster [39].

I have also added a function to represent static text, which is used in my library's API. Writing a text of unlimited length that scrolls in a loop was already implemented in the original version of the library.

Next, it was discovered that the library uses typical bit mappings for all characters that can be displayed on a single 8x8 LED Dot display.

```
pub const CAP_C: SingleDisplayData = [
    0b01111100,
    0b01000000,
    0b01000000,
    0b01000000,
    0b01000000,
    0b01000000,
    0b01000000,
    0b01111100
];
```

Listing 3.11: Example of letter mapping in max-7219-led-matrix-util crate [39].

It was noticed that many letters and symbols were left unrealized and in my revised version of the library I decided to finalize them by analogy with the existing ones (example is demonstrated on Listing 3.11) so that all Latin alphabet, numbers and trivial punctuation marks such as comma, question mark would be covered. Some of the already implemented ones were also redesigned for better and more explicit representation.

As it was said before, the driver requires an allocator, therefore users should make sure that feature „`alloc`“ is activated in their project's **Cargo.toml** file when importing my library if they want to use this module, read more about using this functionality in 3.4.1. Customizing this feature is important to support dynamic content, such as scrolling text, which depends on memory allocation at runtime.

The improved version of the library has been renamed to the descriptive **esp-max7219-nostd**[13]. Under this name, using an import from the GitHub repository, this library is included, defining the chip with which my library will be used (see 2.1.1).

The **max7219** module itself offers several key functions to interface with these LED matrix displays. A **Max7219Display** structure is presented that has an internal instance of the periphery itself, as well as a private **delay source** and a vector (see 2.4.2) in the actual state of the display or chain of displays. The `write_str_looping` function displays scrolling text on the matrix, looping it to infinity. This function locks the entire program because it constantly updates the display in a loop. `set_pixel` sets, correcting the position in a sequential chain of displays. `reset` is part of the **Display** trait defined in **mod.rs** file

[13] Modified library for MAX7219 display: https://github.com/playfulFence/esp-max7219-nostd/tree/alpha/0.2.0

of **display** module, it clears all LEDs in the display, effectively resetting the matrix to an empty state. `write_str` is part of the same trait, it displays static text on the LED matrix. The length of the text is limited only by the number of displays connected in the chain in the user configuration. It is impossible to somehow predict and limit the maximum number of characters, because different characters take up different amounts of display space, so this is left to the library user. The driver does not implement the **EGDisplay** trait because there is no support with the **embedded-graphics** driver.

PCD8544

The PCD8544 display module is designed for the display of the same name. It provides basic functions for initializing and operating the display, including drawing text and setting individual pixels. An existing driver has been used here for a low-level operations.

The **Pcd8544Display** structure encapsulates the hardware-specific customization associated with the various control pins using a function in the `create_on_pins` signature (detailed documentation is described in the 4.2).

All provided functionality is a consequence of the **Display** trait implementation covered at the beginning of the 3.5.3 section and also touched upon in the previous paragraph about MAX7219. The user has access to lighting certain pixels by coordinates, writing lines to the display, and resetting the display The structures and methods are implemented using the **embedded-hal** trait to unify the pins.

ILI9341

The ILI9341 module is a complete solution for interfacing with the ILI9341 LCD using the SPI bus. This module, built using the functionality of the **mipidsi** library[14], extends its functionality by integrating into the **embedded-graphics** ecosystem, offering both basic and advanced graphics capabilities. The driver allows not only basic pixel manipulation, but also sophisticated graphical rendering, including writing text into predefined segments for more informative data display.

The **Ili9341Display** structure is presented, wrapping the internal instantiation of the peripheral itself running in `Rgb565` mode, and taking care that the user does not have to configure all the complex types, generic parameters and initialization manually.

```
pub struct Ili9341Display<
    T: _esp_hal_spi_master_Instance + 'static,
    M: IsFullDuplex,
    RST: OutputPin<Error = core::convert::Infallible>,
    DC: OutputPin<Error = core::convert::Infallible>> {
    pub inner: mipidsi::Display<
       SPIInterfaceNoCS<spi::master::Spi<'static, T, M>, DC>,
        mipidsi::models::ILI9341Rgb565, RST>,
}
```

Listing 3.12: Ili9341Display structure.

[14]mipidsi crate: https://crates.io/crates/mipidsi

The structure from Listing 3.12 is being instanced with `create_on_spi` function in a way demonstrated in Listing 3.13. All types and generic parameters expected in the structure are satisfied in this function and the display itself is set to default settings. All this was created with the expectation that it would be combined with **getter** macros for SPI peripherals (see 3.4.1).

```
pub fn create_on_spi(
    spi: spi::master::Spi<'static, T, M>,
    reset: RST,
    dc: DC,
    mut delay: Delay,
) -> Ili9341Display<T, M, RST, DC> {
    let di = SPIInterfaceNoCS::new(spi, dc);

    let mut display = mipidsi::Builder::ili9341_rgb565(di)
        .with_display_size(240 as u16, 320 as u16)
        .with_orientation(mipidsi::Orientation::Landscape(true))
        .with_color_order(mipidsi::ColorOrder::Rgb)
        .init(&mut delay, Some(reset))
        .unwrap();

    display.clear(Rgb565::WHITE).unwrap();

    Ili9341Display { inner: display }
}
```

Listing 3.13: `create_on_spi` function for ILI9341 Display.

It was decided that the basic fonts included with embedded-graphics are somewhat limited and may not be suitable for most user projects, so this module uses predefined **MonoTextStyle** type fonts from the **profont**[15] package, which range from small to large sizes, allowing text to be displayed in a clear and varied manner. Used **profont** crate is an updated version by Samuel Benko of original package created Wesley Moore. Updated version is adjusted to latest changes in **embedded-graphics** environment. 3 default font styles have been created that a user of my library can use with this display. They are built and available in constants `DEFAULT_STYLE_<SIZE>`, where **<SIZE>** can be „SMALL“ (14pt), „MID“ (18pt) or „LARGE“ (24pt).

To accommodate different display needs, the module includes a **Display** trait for basic functions such as pixel setting and display resetting, as well as an **EGDisplay** trait. The latter extends capabilities to more sophisticated operations such as segment-specific text rendering and advanced graphical layouts that optimize display usage and enhance user interfaces.

The module utilizes features related to the enum **DisplaySegment** covered in 3.5.3 – the user will be able to write out data from their sensors to the display in a more structured way. The functions `write_segment_name`, which writes the name of the segment itself on top of it and `write_to_segment`, which writes the data directly to the center of

[15] updated profont package: https://github.com/sambenko/profont/tree/master

the segment, serve this purpose. Inside these functions, using **match pattern** covered in 2.1.1 and simple math calculations, the point where the string should be written is considered based on the segment passed to the function. The segments are arranged so as not to overlap each other, see Listing 3.4.

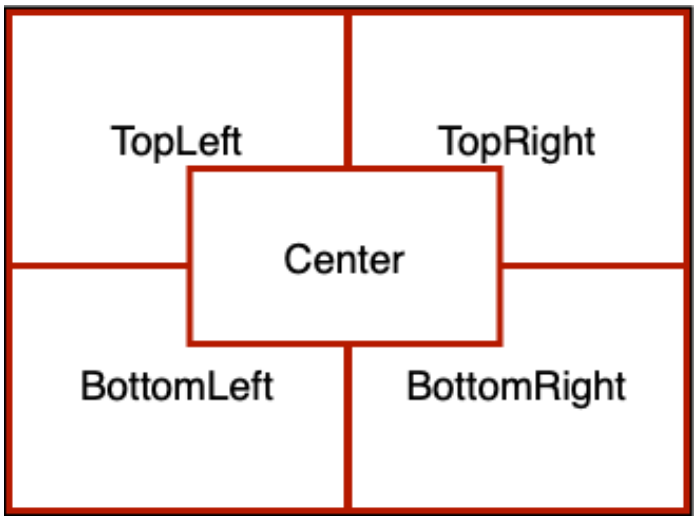


Figure 3.4: Segments layout for ILI9341 display.

In addition, the driver in my library manages display orientation and color ordering directly through **mipidsi**'s default configurations, sparing the user from complex settings. This integrated approach not only simplifies the development process, but also improves the performance and flexibility of applications utilizing the ILI9341 display.

3.6 Connectivity Features

This section will cover the basic Wi-Fi and MQTT related functionality of this library. This was not part of the assignment and can be seen as an extension, but I thought it was worth implementing at least relatively basic connectivity related functions for the sake of completeness.

First of all, in order to make this functionality of my crate available, `<chip_name>-wifi` feature should be activated in library import in in case the user is going to use the Wi-Fi only functionality, or `<chip_name>-mqtt` in case the user is going to use MQTT. Only one of these features must be activated! This approach greatly simplifies the process of importing this functionality from **esp-wifi**, due to the fact that this driver from **esp-rs** is quite complex, so the different modules are connected by configuring certain features, in which an inexperienced user of the Rust language and esp-rs infrastructure will find it quite difficult to get confused [15]. So all the work is done for him.

Also, to simplify code modulation, two so-called „marker features“ were created in the Cargo.toml file of this project – feature „`mqtt`“ and feature „`wifi`“. The point is that **esp-wifi** also requires selecting a microcontroller via features on which to run this library. Thus, it is simply not possible to combine and unify all imports for Wi-Fi and MQTT functionality for each into one or two features. And this is where the situation is saved by

the above-mentioned marker traits, which have no dependencies or only general ones. With their help, as it was said before, it is possible to modulate the code and, provided that the „mqtt“ feature has been activated, to make this or that functionality active.

```
#[cfg(feature = "wifi")]
pub mod connectivity;
```

Listing 3.14: Conditioned code activation by enabled feature.

The module responsible for connectivity features is located in the *esp-ward/src/connectivity* directory. Whole project structure is covered in 3.8.

The root file of this module (connectivity/mod.rs) is the central point for this module and if the „`mqtt`“ feature is activated, includes the corresponding module with the **mqtt** sub-module in a same way that shown in Listing 3.14. Also a macro called „`init_wifi`“ has been implemented in this file. The reason why it is a macro and not a function is the same – access to the **Peripherals** structure, which, as it was explained above in the text, is not possible through the function because of the „partial value move“ error. It has two variations, of which the desired one is also selected based on the `cfg` attribute – for configuration with only feature „`wifi`“ enabled and for configuration using MQTT functionality. For the end user there is no difference in calling this macro, the only difference is the data returned. The point is that a slightly different algorithm is required to work with the MQTT stack. This is discussed further in the subsections devoted to these two communication methods. The creation of this functionality is inspired by the documentation and examples provided in **esp-wifi** [15]. It was necessary to break down this knowledge into specific pieces and compose functions that would greatly simplify and shorten the code for initializing the Wi-Fi module and accessing the network. Which I believe has been accomplished successfully, detailed documentation can be seen through the 4.2 section and the use cases described in the 4.5.1 section.

3.6.1 Wi-Fi

The **wifi** submodule provides the necessary functionality for networking with the **esp_wifi** library and the **embedded_svc**[16] helper crate. Key features include the ability to create and manage sockets, making it easy to send and receive data over network connections.
Key features and capabilities:

- Socket Management: The sub-module allows user to create sockets with specified IP addresses and ports using the `create_socket` function, enabling the sending and receiving of data. This is very important for interfacing with various network services.
- IP address processing: Functions for converting string IP addresses into byte arrays are included, which ensures proper processing of IP data required for working inside the program – function `ip_string_to_parts`.
- Time Synchronization: Includes methods to retrieve the current time from the server. For this purpose, the WorldTimeAPI service[17] has been chosen, which, as part of normal insecure communication, will send a message with the current timestamp as the first response. Conversions of UNIX timestamps into more readable formats

[16] embedded-svc crate: https://crates.io/crates/embedded-svc
[17] WorldTimeAPI: http://worldtimeapi.org

(hours, minutes, seconds, days of the week) are also implemented. The function `get_time`, which returns tuple in the format (hours, minutes, seconds), and the function `get_timestamp`, which returns raw UNIX timestamp. The functions `timestamp_to_hms` and `weekday_from_timestamp` are also implemented to convert it into a readable format.

- Data Transfer: Users can send requests to remote servers using established sockets, with support for response processing that includes utilities to parse timestamps from server responses. Two functions `send_request` and `get_response` serve this purpose.
- Error Handling: Robust error handling of various errors is built into the functions to ensure reliable and stable network operation, including resolving potential transmission and reception problems.

Detailed documentation for all functions, including input parameters and returnables can be found in the Section 4.2.

The inspiration for the time and timestamp functions came from experiments in my own project in the past [34]. General knowledge about Wi-Fi was obtained from the **esp-wifi** project [15].

3.6.2 MQTT

This module in the library provides communication with MQTT brokers, allowing devices to send and receive messages using this protocol. This includes creating network sockets, managing connections, and handling the sending and receiving of data using well-defined interfaces and procedures.

It is worth noting that this module operates entirely in **async** mode, the features of which were discussed in 2.1.1. The reason for this is that at the time of creating the project described in this thesis, no library was found that would provide a working MQTT client in **no_std** environment (see 2.2.2) that would work in blocking (non-async) mode. While creating such a library is theoretically possible, it would be very time consuming and add enormous complexity. Such a client in itself requires a completely different stack knowledge and would be a significant complication of the task, going far beyond its scope. Core Functionalities:

- Initializing Wi-Fi communication: As mentioned in the previous 3.6.1 sub-section on Wi-Fi, and above in the text – this MQTT module uses its own variation of macro `init_wifi`. It will be selected automatically when the user activates the feature **<chip-name>-mqtt**.
- Socket Creation and Management: Establishes sockets for communication, handles IP address configurations, and manages data buffers for sending and receiving information.
- Creating and managing sockets and connection: Creates sockets for communication within features, handles IP address configuration while creating a channel with the MQTT broker. Functions exist in the library to communicate with the HiveMQ Websocket Client[18], which is chosen as the default MQTT broker due to its extraordinary

[18] HiveMQ Websocket Client: https://www.hivemq.com/demos/websocket-client/

ease of use. The function `mqtt_connect_default` is responsible for this functionality, which will return to the user a ready-made MQTT client already connected to HiveMQ. There is also a `mqtt_connect_custom` function by itself, which will join a given broker using DNS Query. The user can pass **username** and **password** to it, if they are needed. If not, user only needs to pass „`None`“ for these parameters when calling the function (see 2.1.1). All connections are made via TCP/IP stack.

- Message handling: Features for sending requests to MQTT brokers, subscribing to topics, and receiving messages. They include robust error handling mechanisms to ensure reliability, such as reconnection strategies in case of network failures. The functions `mqtt_send`, `mqtt_subscribe` and `mqtt_recieve` are responsible for these functions.

- Executor tasks: The library also offers two necessary functions to ensure that the connection to the selected Wi-Fi network is established and the network stack is started. Implemented under the `#[embassy_executor::task]` attribute, these two functions are appropriate functions to let into the task spawner of the **embassy_executor** library, which was briefly discussed in the 2.1.1 section.

- Utility Functions: This module also provides a set of helper macros that greatly enhance the user-friendliness of the entire API. Such as `prepare_buffers`, which returns a tuple of four buffers of different sizes, which are needed for various MQTT functionality. The `wait_wifi` macro will asynchronously wait until the Wi-Fi stack is fully configured and the network is connected, while `get_ip` will make sure that the Espressif chip running the code gets its IP address before starting the MQTT communication itself. `create_stack` makes sure that the wifi interface obtained from the above `init_wifi` is wrapped up to the type that will be used in all of the above MQTT functions.

As with the previous Wi-Fi section 3.6.1, this module was written based on a basis of an analysis of an existing project from an **esp-rs** team member Juraj Sadel, that was shown at Espressif DevCon 2023 [37]. After carefully examining the functionality of this example, as well as looking at side drivers, a plan was made to make it as easy as possible for new users of the **esp-rs** infrastructure to access MQTT functionality.

An example use case can be found through the 4.5.1 section.

3.7 Build Environment

When programming embedded systems, especially when working with different microcontroller architectures, it is very important to carefully customize the build environment. This ensures that the compiled code is a perfect fit for the intended hardware. My project uses the **build.rs** script as well as special settings in the **.cargo/config.toml** file to manage this complex task, ensuring a smooth build process for different microcontrollers.

3.7.1 The Role of Build Script

The **build.rs** file plays a pivotal role in project setup. It is a build script, automatically executed by Cargo before the main build process begins. This script is used to perform

custom build actions that are outside of **Cargo**'s typical build capabilities. In this case, **build.rs** handles the compilation of necessary components, sets up the linker, and configures various flags and options essential for the proper functioning of firmware on different MCUs.

It checks that exactly one chip-specific feature has been activated for the library and in case of an error writes an error to the terminal and ends the build process.

Also, for correct parameter passing to the linker (see next 4.3 subsection), it controls if the marker trait „`wifi`“ is activated, which signals the use of Wi-Fi functionality (covered in 3.6.1) in the project and means that another parameter should be passed to the linker. I got the information about this from a similar build script in **esp-wifi** [15]. General knowledge about build scripts was obtained from „The Cargo Book“ [8].

3.7.2 Configuring Linker Parameters

The **.cargo/config.toml** file provides a centralized configuration for specifying build parameters and options. In my project, this configuration file is used to define custom linker arguments for different target architectures. The `rustflags` parameter specified in the file contains important arguments for that direct the compiler to use our specialized **linkall.x** linker script to determine the memory location for flashing. Necessary information about this configuration was taken from **esp-hal** [11] and **esp-wifi** [15] projects.

3.8 Final Project Structure

The structure of the project was conceived to facilitate the development and testing of embedded systems components, with a clear division of tasks evident through the directory layout.

- **.cargo/**, **.github/**: Configuration for cargo build system (covered in 3.7) and GitHub-specific workflows (like CI/CD processes) (described in 4.5.2).
- **examples/**: Contains various example programs demonstrating the usage of the library's features, which cover all implemented functionality (see 4.5.1).
- **src/**: The source directory, encompassing the core library code.
  - **lib.rs** – The root file of the entire library. This is where the key macro for controlling internal peripherals and the connection logic of other modules are located. Detailed description is present in 3.4.1.
  - **connectivity/**: Modules related to network connectivity features like Wi-Fi and MQTT communication. Covered in 3.6.
  - **display/**: Handles display functionalities, possibly interfacing with different types of screens. Described in 3.5.3.
  - **peripherals/**: Code for interacting with external devices such as sensors and buttons. See 3.5.1 and 3.5.2.
- **resources/**: This directory contains data, required for the documentation website. Detailed description can be found in 4.

- **Cargo.toml**: Defines package information, dependencies, used crates, and provides features for specific configurations, described in multiple sections above.
- **LICENSE-APACHE**, **CONTRIBUTING.md** and **README.md**: Standard documentation files for open-source software explaining licensing, contribution guidelines, and project information. More details are described in Chapter 4.
- **rustfmt.toml**: This file contains settings for formatting Rust code. This helps keep the code style consistent throughout the project. Due to the fact that it is difficult to keep a uniform and beautiful code style, and the library is intended as an open-source project, such a measure is simply necessary. See 4.6 for more details.

Chapter 4

Release of the Project

One of sub-tasks in the assignment is to publish the library on GitHub. Because of this, it was decided to use the git tool as a version control system from the very beginning of the project.

Many times when working with different libraries, a programmer comes across the fact that the creator simply does not provide any tutorial guide, which artificially prolongs the time of working with the required functionality. Indeed, having good documentation, examples of how to use the library's functionality, automated testing, and a well-defined contributing guide are a big part of the success of any library. This will shorten and make it more pleasant for a potential user to learn the API of the crate.

This chapter will describe what and how my project has done to make its public resources user-friendly and understandable.

Final version of library is available on: https://github.com/playfulFence/esp-ward

4.1 Library Identity and Community Engagement

The name of the public library should be simple, memorable and reflect the idea of the project. It was decided to name the crate as „esp-ward“. This name is an amalgamation of the words „ESP“, referring to the Espressif systems on which the library runs, and „ward“, meaning protection or surveillance, a reference to the popular World of Warcraft series of games. Together, „**esp-ward**“ symbolizes the library's role in securing and facilitating the development process on ESP devices. This name reflects the library's core mission to provide reliable and user-friendly tools for developers working with Espressif chips, enhancing their ability to efficiently manage and deploy sensor-based applications.

This section also discusses the **README.md** and **CONTRIBUTING.md** files required for initial guidance of users and potential contributors to the **esp-ward** project.

The **README.md** serves as the initial interface for the project, providing an overview of key features, installation instructions, and a quick start guide. It emphasizes the library's ease of use, modular architecture, and the extensive support for various peripherals and features. Additionally, it directs users to detailed documentation (see Section 4) and offers

troubleshooting tips.

The **CONTRIBUTING.md** guide encourages community involvement by detailing how to contribute to the project. It includes guidelines for submitting issues and pull requests, outlines the code of conduct, and provides instructions for setting up a development environment. This document is crucial for maintaining a healthy and productive community, fostering contributions that help enhance the project. Similar practices have been found in many open-source projects, such as **esp-hal** as well [11]. The general idea and necessary patterns for this guide were found in an article from Mozilla Science Lab [27].

4.2 Website and Documentation Deployment

Such libraries should always be accompanied by rigorous technical documentation, with which you can conveniently find the information you need about a particular crate functionality. This section details the deployment and hosting of the project's website and documentation on GitHub Pages. This method was chosen because of its simplicity and convenience at the same time – both the project and its website are located on the same site, and to activate the ability to deploy the website you only need to configure it in the repository options.

The site's landing page (see Figure 4.1), written in „**index.html**“ file and located in the „**resources**“ directory, serves as an entry point. It has been written to fit all styles into the general style of the Rust documentation (which can be seen in generated documentation). The general concept was taken from a similar page in **esp-hal**, which I wrote for **esp-rs** [11]. The landing page is pretty minimalistic, but informative. It leads to several key links in the project, including the technical documentation itself.

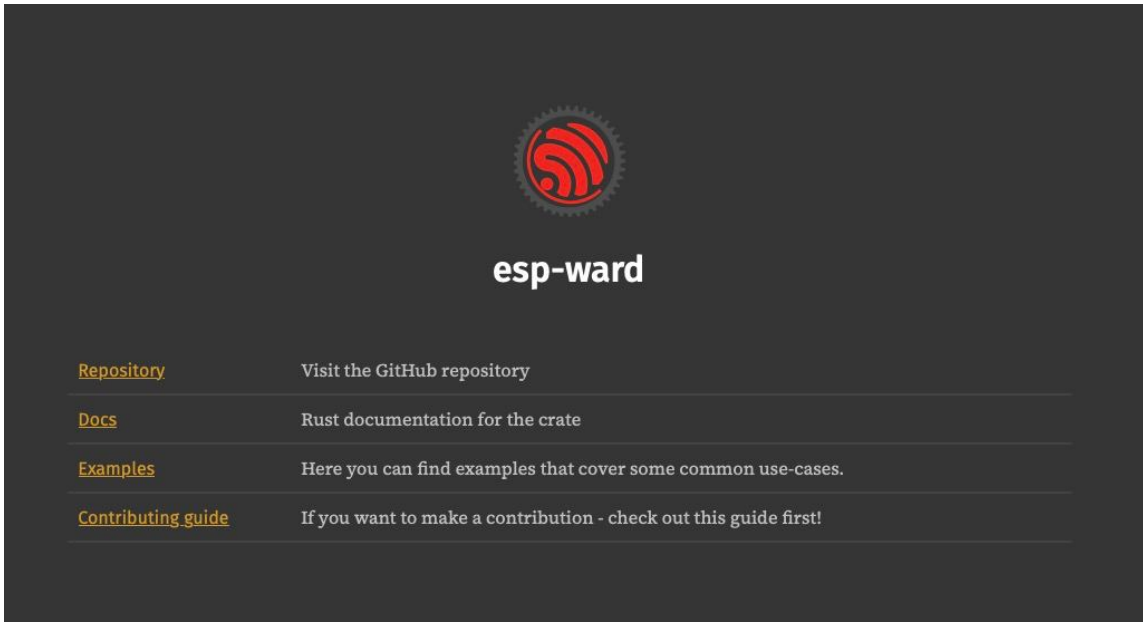


Figure 4.1: Landing page of the crate.

The project documentation (see Figure 4.2) is automatically generated and updated for each commit to the project during the continuous integration process (which is described in the 4.5.2 section) specified in the „**.github/ci.yml**“ file under the „`build-and-deploy-docs`“ task. The documentation is created using the `cargo doc` tool [8], specifically for the ESP32C6 (chosen randomly) chip with „**esp32c6-mqtt**“ enabled, ensuring that all library functionality is fully included in the documentation. Detailed doc strings with „`//!`“ at the beginning of the comment were written to create documentation throughout the code.

The GitHub action „`JamesIves/github-pages-deploy-action@v4.5.0`“[1] is used to deploy project to the GitHub Pages. This action is chosen for its ease of use and efficiency in deploying web content directly to the GitHub repository. Documentation and landing pages are created in a virtual environment and then deployed to a dedicated „**gh-pages**“ branch that hosts the GitHub Pages site. With this setup, the project website is constantly updated with the latest documentation and resources, providing a reliable and informative platform for users and contributors.

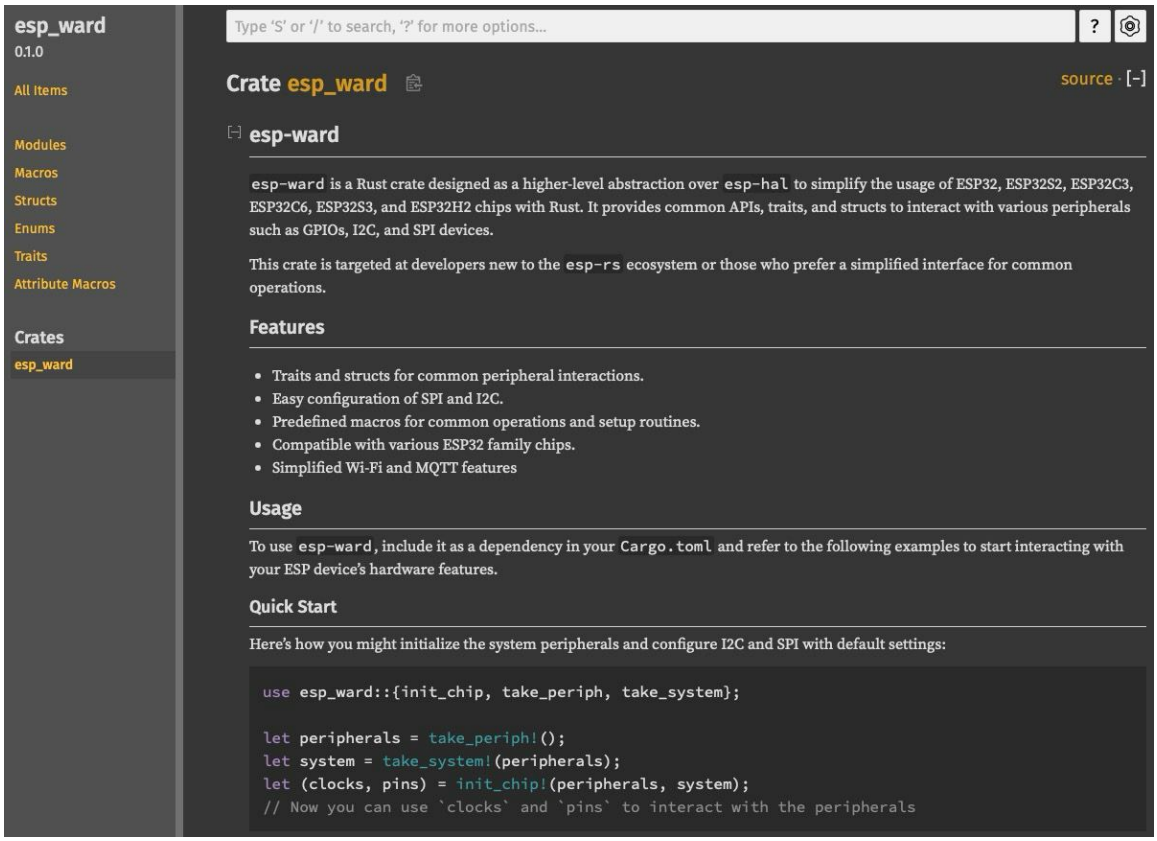


Figure 4.2: Technical documentation for the library.

Documentation is available by clicking on the „`Docs`“ link on the landing page. Detailed documentation for all functionality, modules and macros is available there.

[1] James Ives deploy action: https://github.com/JamesIves/github-pages-deploy-action

The library's website is available on https://playfulfence.github.io/esp-ward/. It can also be built and run locally using the `build_docs.sh` script located in the resources folder. The script must be run from the resources folder. Before executing it, run the `chmod +x build_docs.sh` command or similar, depending on your system. After running the script, it will be enough to open the `index.html` file from the same folder in a browser.

4.3 Getting Started

The first thing to do to start writing any embedded application in Rust is to install all the necessary software environment for it. A similar guide from „**The Rust on ESP Book**" is briefly quoted here [17]:

- Install Rust from the **rustup** website[2] with a command demonstrated in Listing 4.1.

  ```
  $ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
  ```

  Listing 4.1: Rust installation.

  Using `homebrew`, `apt-get` or other default package managers may result in incorrect installation of components and lead to further incompatibilities.

- Then directly via „**cargo**" (see Listing 4.2) user needs to install the official **esp-rs** tool – **espup**[3], which will install the entire environment for using the language with Espressif chips.

  ```
  $ cargo install espup
  $ espup install
  ```

  Listing 4.2: **esp-rs** environment installation.

- Now that all the necessary drivers are installed, user can get started on the new project itself. Then it is possible to deploy a new project according to a pre-prepared template from the **esp-rs** team. Use the `cargo generate` tool (see Listing 4.3) to create a new template project.

  ```
  $ cargo install cargo-generate
  $ cargo generate esp-rs/esp-template
  ```

  Listing 4.3: **esp-rs** environment installation.

  After answering a few clarifying questions, a folder with the project will be created in the terminal where the user will select the chip of interest and other clarifying details. To the question „Configure advanced template options?" it is necessary to choose „false", because it will still require other further customization of the project, which is described below.

- Then in the project folder the user should go to the file **Cargo.toml**, in which he needs to change the versions of libraries of the organization **esp-rs** to those on which my library **esp-ward** operates (more details are described in 5.3).

[2] rustup website: https://rustup.rs
[3] espup tool: https://github.com/esp-rs/espup

```
esp-hal = { version = "0.16.1", features = ["eh1"]}
esp-println = { version = "0.9.1"}
esp-backtrace = { version = "0.11.1", features = [
    "panic-handler",
    "exception-handler",
    "println",
]}
esp-wifi = { version = "0.4.0", features = ["wifi-default"] }
```

Listing 4.4: Versions of **esp-rs** drivers used in library.

In all these crates, as a feature, user needs to pass the name of the chip on which the user's project will operate. This will already be done in the template or check an example of such an operation in the 2.1.1 section.

- Next, the user needs to configure the `.cargo/config.toml` file, which contains the parameters in the linker script. It is already preconfigured by the template, it is only necessary to double-check that everything is set exactly as it should be.

```
[target.riscv32imc-unknown-none-elf]#esp32c3, esp32c2
rustflags = ["-C", "link-arg=-Tlinkall.x", "-C", "force-frame-pointers"]

[target.riscv32imac-unknown-none-elf] #esp32c6, esp32h2
rustflags = ["-C", "link-arg=-Tlinkall.x", "-C", "force-frame-pointers"]

[target.xtensa-esp32-none-elf] #esp32
rustflags = ["-C", "link-arg=-Tlinkall.x"]

[target.xtensa-esp32s3-none-elf] #esp32s3
rustflags = ["-C", "link-arg=-Tlinkall.x"]

[target.xtensa-esp32s2-none-elf] #esp32s2
rustflags = ["-C", "link-arg=-Tlinkall.x", "-C", "force-frame-pointers"]
```

Listing 4.5: **config.toml** linker parameters configuration for different architectures.

If allocator will be used („**alloc**“ feature, see feature layout in 4.5.1), user needs to make sure that the „**alloc**“ parameter is passed to the „**build-std**“ setting in this file. This determines which parts of the standard library are used in a **no_std** application (see section 2.2.2). If Wi-Fi functionality is used, the „**rustflags**“ setting should also include the parameter „`-C`", „`link-arg=-Trom_functions.x`". Unfortunately, library can't do anything about these complexities on its end and these customizations need to be done by the user in their own project.

- Install **espflash** according to description in 2.4.2.
- Import **esp-ward** library with necessary features as it shown on Listing 4.6 (see feature layout in 4.5.1).

```
esp-ward = { git = "https://github.com/playfulFence/esp-ward.git",
    features = ["esp32c3-wifi", "alloc"]}
```

Listing 4.6: Library import.

- Based on the provided examples described in 4.5.1 and documentation covered in 4.2, user can start writing his own project.
- Project could be flashed with command demonstrated in Listing 4.7:

```
$ cargo espflash flash --target=<chip-architecture> --release --monitor
```

Listing 4.7: Flashing with **espflash**.

Where **<chip-architecture>** should be set according to the targets seen in the 4.5.

4.4 Troubleshooting

- If something is wrong with a building process and **CI** in repository is green after latest commit to 'esp-ward' – problem is on users side.
- Check if toolchains and the rest of environment are correctly installed. Use **The Rust on ESP Book** [17] or 4.3 for more info.
- Try to pass correct **toolchain** to **cargo** executing command. (**esp** for Xtensa chips, **nightly** for 'RISC-V' microcontrollers, see 2.3.1):
  `cargo +<toolchain> espflash flash ..`
- On Xtensa targets linker issues might be encountered. User should install and export the **esp-idf** like it is demonstrated in Listing 4.8, according to guide in repository of this project [19].

```
$ git clone --recursive https://github.com/espressif/esp-idf.git
$ cd esp-idf
$ ./install.sh all
$ . ./export.sh
```

Listing 4.8: ESP-IDF installation [19].

4.5 Testing

Testing of the final solution will take place in two paradigms. First of all, we need to provide a set of examples that show not only what can be done with the functionality provided by the library, but also how to specifically use the resulting API in various situations.

The second testing paradigm is mixed with evaluation of results – it will compare several applications in a „before-and-after“ format, where on one side there is a program written without using esp-ward, and on the other side there is a code that performs the same task, but using the API of the resulting library. In this way we will try to find out whether my library has simplified the work with external peripherals and the development of embedded applications in general. Naturally, this type of testing cannot do without interviewing people who have been asked to use the library. Their comments will also be included in the evaluation.

Also, this part will talk about the various experiments and decisions that were made while working on this project, but in the end never made it into the final crate version.

4.5.1 Examples

In the repository with the project, in the folder „**examples**“ there are programs that show how to work with the library and cover most of the functionality implemented in **esp-ward**. The following examples have been implemented:

- `display_button` – this example demonstrates the simplest interaction with the functionality of a button and a display, where when a button is pressed, there is an instantaneous response on the display showing which button was pressed.
- `display_data` – a complex example of displaying several types of data on ILI9341 – temperature and humidity using BME280 and time using Wi-Fi functionality of the library.
- `distance_sensor` – basic example of a full use of distance sensor. Due to its design, this peripheral requires the ambient temperature, which in the example is received from the measurements of the AHT20 sensor 3.5.1.
- `etch_a_sketch` – adaptation of the popular game of the same name, using MAX7219 and joystick. You can use the joystick to control the drawing pointer, and pressing the joystick will erase the drawing and move the pointer to the starting point. In this example, there is a limitation related to ADC calibration, see section 3.5.2 for more details.
- `led_scrolling` – this example also uses the **MAX7219** display (see 3.5.3), which will show the scrolling text specified in the example.
- `motion_detect` – the simplest example, signaling motion detected on the **PIR** sensor shown in 3.5.2 to the terminal.
- `mqtt_client` – an example of an MQTT client that subscribes to a default theme and sends a default message to the websocket client from HiveMQ, which has been selected as the default client, it was covered in 3.6.2. The example demonstrates how to work with MQTT and shows the correct initialization and use of functions within the **async** environment. In real usage, the user must first configure the HiveMQ client itself, then set the generated Client ID and topic name in the example. Also, the user can build additional sensor measurements into the code themselves, for example from **sensor_i2c** and pass the values from it to the `mqtt_send` function.
- `send_request` – this example shows how you can interact with Wi-Fi functionality, open sockets, and send and receive messages from a remote server. It is based on the example idea from the **esp-wifi** repository [15].
- `sensor_i2c` – demonstrates the simplest measurement from sensors and outputting that data to a terminal.

Any example might be flashed with a command from Listing 4.9.

```
$ cargo espflash flash --example=<example-name>
    --features=<chip-feature> --target=<target> --release --monitor
```

Listing 4.9: Flashing examples.

Where **<chip-feature>** – if chosen example utilizes **Wi-Fi**-related features – enable corresponding „**wifi**“ chip feature (example: `esp32s2-wifi`), same works with **MQTT** functionalities (example: `esp32c6-mqtt`). In case allocator is going to be used (**led_scrolling** and **etch_a_sketch**) – make sure to enable **alloc** feature (example: „`esp32,alloc`“). In case just basic functionality is used – just name of required chip to a feature list (example: `esp32s3`). **<target>** is architecture of a chip (see 4.5):

- `xtensa-<chip>-none-elf` – for ESP32, ESP32S2 and ESP32S3
- `riscv32imc-unknown-none-elf` – for ESP32C3 and ESP32C2
- `riscv32imac-unknown-none-elf` – for ESP32C6 and ESP32H2

These examples fulfill several important tasks: they show users on visual examples how to use the provided API, on these examples you can see how and to what extent the resulting library simplifies the developer's process of writing a year when working with supported peripherals, simplifying both the code itself in terms of syntax and introducing a common and understandable architecture for all supported devices. The second important role of these examples is in directly testing the implemented methods, which allowed me, as a developer of the library itself, to understand whether this or that functionality works as expected, whether the proposed design is intuitive and more pleasant for me myself.

4.5.2 Automated Testing

The **esp-ward** library's continuous integration process ensures that every existing example in the „**examples**“ folder (to be described in section 4.5.1) and feature set is tested on all supported Espressif chips (see 2.3.1). This robust testing strategy helps maintain control over the reliability and compatibility of the library with different hardware configurations. In a project with so many moving parts and dependencies, this measure is simply necessary.

CI is configured to automatically build and test the library itself and its examples for different chips, ensuring comprehensive coverage. This is done using GitHub Actions, which organizes testing in a matrix of target environments. The process tests the library on **Xtensa** and **RISC-V** architectures. Each chip has specific target specifications and toolchains to meet their unique architectural requirements (see section 4.3). This includes building with basic functions and, where possible, additional functionality such as „`alloc`“, „`wifi`“ and „`mqtt`“. Such step ensures that the core library and its functional extensions compile without errors in all supported configurations.

Depending on the requirements of the example and the capabilities of the chip, different feature flags are applied during the build process. This ensures that each example is tested under conditions simulating real-world usage. Special conditions are applied to exclude certain examples from chips where they are not supported or relevant, e.g., omit Wi-Fi and MQTT examples for an ESP32-H2 chip that does not carry a Wi-Fi module (see brief chips specifications in section 2.3.1).

The CI configuration is designed to automate the verification of each commit to the repository. By testing all supported chips and configurations, CI ensures that any changes to the library will not break compatibility with existing features or hardware, and if there are any errors, they will be explicitly detected.

The basic knowledge stack for implementing this functionality was obtained from the JetBrains CI/CD guide [25], as well as from the example of similar testing in **esp-hal** [11].

4.6 Code Formatting

In the **esp-ward** project, the formatting of the code directly is controlled by the **rustfmt.toml** file. This configuration file ensures that the codebase adheres to a uniform style, making it easier to read, maintain and contribute to. It sets the rules for formatting the code. Among the key settings are enabling formatting in comments, in function definitions, in function calls, and so on, limiting the length of a line of code so that the code lines up in a readable vertical structure.

By automating code formatting, the project minimizes stylistic inconsistencies and simplifies the review process, facilitating smoother collaboration with the community. This practice is especially useful in open source projects where multiple contributors may have different coding styles.

Formatting can be used with the `cargo fmt` [8] command from the project root folder or enable the format on save feature in the code editor. For this purpose, for example, in the VS Code editor user needs to download the rust-analyzer extension, which will activate this functionality.

Due to the focus on cooperation with the **esp-hal** driver, it was decided to use the same code style as in the aforementioned crate [11].

Chapter 5

Testing and evaluation

This chapter will provide an evaluation of how **esp-ward** simplifies writing embedded applications, makes code clearer, shorter and more readable. The analysis is performed in „before-after“ format by comparing two programs with the same functionality, but one of them will be written without using my library and the other with it. This way it is possible to clearly understand whether the goal of the project has been achieved, i.e. whether writing similar applications in the Rust language on the Espressif chip platform has become easier.

Those pairs of programs are listed on the media attached to this thesis, there is a „**compare**“ folder, it contains Rust format (`.rs`) files with a source code of observed examples, but they are not buildable and are not intended to be used, their only purpose is to serve as a visual comparison. All examples written using **esp-ward** are named in the format `<example_name>_ward`, where **<example_name>** is the name of the original example from Wokwi or other sources.

5.1 Validation

One of the validation methods I chose was the Wokwi[1] simulator. This is a platform where you can create a hardware configuration, flash code to it and see in real time how the same process would run on a real device. Luckily, Wokwi has a collaboration with the **esp-rs** team, there is even a link to existing Bare-metal Rust examples on the title page. These are the ones I'll be using.

1) Comparison will start from one of the examples from the Rust page – „Crispy Click“[2]. As mentioned in 3.5.2 this example and the algorithm to run it was developed by me personally. This program demonstrates the basic initialization of the display and buttons, the interaction between them, and the graphical functionality. The typical initialization of onboard peripherals such as **SPI** and **GPIO**, done in classic **esp-hal** is also shown. There will be no further mention of this, it applies to all programs using the native **esp-hal** driver. Also to work with buttons it is required to implement a complex debounce mechanism, otherwise it will not work in expected way. It is noticeable the need for rather complex technical customization of internal peripherals, importing a large number of libraries and

[1] Wokwi.com browser IoT simulator: https://wokwi.com
[2] Crispy Click example: https://wokwi.com/projects/341706650098336338

modules, which requires a lot of time to study.

As a counterpoint, I implemented a version[3] of it using the **esp-ward** library in the Wokwi. What's immediately noticeable is the significant reduction in the amount of code. The readability of the code has also improved significantly – even a person who doesn't know the Rust language fully understands what's going on in the code. The settings of the internal chip have also been greatly simplified. The programmer uses pre-defined macros that will return all preconfigured peripherals to variables. Initialization of the display has also been greatly simplified to just passing a bus instance, two pins and a delay source to a function, instead of complex configuration of resolution, orientation, color order and so on. All provided methods for comfortable work with buttons also give their results – after initialization, it is enough just to use the method „`read`“, debounce and low-level work is done without user's participation. Also, writing inscriptions to the display has been simplified to literally one line and function, instead of careful customization by coordinates and font. These functions also work completely correctly, first scrubbing the area and then writing new text. In the table **esp-hal** version, this has to be done manually.

2) The second example to be compared is the MAX7219 Dot matrix display, which shows a scrolling text[4]. Here the user has to find how to initialize the allocator on their own, as well as a slightly more complicated display setup.

In the redesigned example[5] using my library, the alligator setup is accomplished with a single call to the appropriate macro `prepare_alloc!`. There is also no need to extra import libraries for this display – everything is already enabled via **esp-ward**. From this example, you can see that simplification is more apparent when the project in which the library is used is more complex. However, the code has definitely become more readable.

3) The last example discussed in Wokwi is an example written in the std environment, but this does not prevent the study from comparing it to a similar implementation using the resulting library. The most complex example is analyzed – **esp-clock**[6]. In the example, the chip connects to Wi-Fi and shows the time and day of the week. It does a huge range of settings. An incremental, complex and deep initialization of Wi-Fi peripherals is done, with the need to get non-obvious peripheral instances to the user beforehand. **SNTP** protocol is used, which is a convenient way to get time. The code uses a lot of wrapped C-functions, which requires explicit use of unsafe sections, which is not a nice practice for this kind of use-case [12]. Usually, unsafe code is used at the lowest levels of hardware communication, but not in the application itself. In general, due to the very high complexity of such a program in embedded programming, the source code is very huge, a lot of low-level and confusing settings are used, which may rather scare off a newcomer. Also the code is unreadable without knowing the context of the program and the resulting output.

On the other side of the comparison is an example[7] I wrote that performs similar actions. It also shows the time and day of the week on the screen, though without the introductory

[3] Crispy Click emulator, remade using esp-ward: https://wokwi.com/projects/396547396717778945
[4] MAX7219 Scrolling Text example: https://wokwi.com/projects/344869867332043347
[5] MAX7219 Scrolling Text example, using esp-ward: https://wokwi.com/projects/396544824213521409
[6] esp-clock project on Wokwi: https://wokwi.com/projects/357451677483992065
[7] esp-clock with esp-ward: https://wokwi.com/projects/396552224931760129

esp-rs logo and graphics for Wi-Fi communication. However, the above functionality has a rather minor effect on the complexity of the original example, so the comparison can be considered fair. The comparison by the number of code lines with almost the same functionality speaks volumes – 400+ lines of code against 75, in the case of the example using **esp-ward**. The readability of the code is incomparably higher, the API is easy to read and use, considering that all documentation is in one place (see 4.2). The downside is that developer have to use a third-party function to write numbers into string format, but this is more a problem of the **no_std** environment (see 2.2.2).

The provided reflection on the introduced functionality and simplification is not only the subjective opinion of the author, but also of the users to whom the library was offered for testing. Read more about the results of the survey in the next section 5.2

5.2 User Survey

This section will describe the results of a survey of users who I asked to test the library and give their opinion on whether esp-ward accomplishes its goal of simplifying the development of embedded applications in Rust on ESP chips. One of the goals of the project is to simplify not only for people who already program in this language, but also for newcomers (the methods of testing the project with them are described in the 5.2.1 subsection).

5.2.1 Feedback from Non-Rust Users

From testing the library on people who do not write in Rust and on those for whom this language is not a profile language, it is impossible to get any deep analysis of the resulting API and functionality. However, another valuable metric can be received from them – how much more readable programs using my crate are than projects written without it. The interviewees were asked to consider several programs that used **esp-ward** and the same or similar programs written using drivers from **esp-rs** as they are (the principle is similar to what is described in the 5.1 section). The questions were posed as follows: „From which of these two codes is it clearer to you what is happening in the program? Do you think you could write a simple program using my project faster than without it?“.

The answers from all of the interviewees were mostly laudatory toward the library. Some of them became interested in developing embedded applications in Rust and expressed a desire to start learning it. Also very important to the newcomers was the clear and descriptive GitHub repository and in particular the technical documentation described in 4.2. It was easy for them to navigate through it. Examples covering most of the functionality were also well received. Although not all specific sensors were involved, by analogy users were able to understand how to work with peripherals that are not involved in the examples themselves.

From the negative feedback – because macros are used instead of functions, the extensions for Rust do not tell what arguments should be in a function, for the same reason, some of the libraries used inside macros are not loaded automatically (as they are in functions), because of this the user has to include some libraries that are not obvious to them in the project. Such problems were expected, and are discussed in the 5.3 section as well.

5.2.2 Feedback from Rust Programmers

From the testing of people competent in Rust and embedded development in this language, it is possible to emphasize a deeper and more professional analysis of the resulting project. We also managed to show the library to members of the esp-rs team.

This testing group went through the whole code and functionality very specifically, thanks to which it was possible to identify several places in the project where improvements were needed. During testing with them it was discovered that the `#[cfg(...)]` attribute, which is responsible for running different code depending on the features configuration, often didn't work correctly inside the macro when the library was imported into the project. The problem has been fixed, more on this in Section 5.3. They also recommended a few changes to the **connectivity** module API (described in 3.6) related to the type of data passed to functions. Prior to this comment, these functions received, for example, string as input. The developers also noted that setting compiler parameters in the **.cargo/config.toml** file on the user's side could be confusing for the user. But with this also at the moment nothing can be done on the Rust language side, the parameter must be passed to the linker. The process is described in more detail in 4.3.

All the positive feedback given by newcomers to the Rust language in the 5.2.1 section was also highlighted by this test group. They also praised the fact that when using all „default“ initializations, no conflicts in pins occur. The structure of the project was also praised in terms of its openness to commits from the community, and the idea of a **CONTRIBUTING** guide (see 4.1) has paid off. Among other things, the idea of unifying access to supported peripherals and the way various difficulties related to the peculiarities of the Rust language were overcome was well appreciated.

5.3 Challenges Encountered

This section will talk about various difficulties encountered during the project, limitations, ideas not realized or canceled for various reasons.

First of all, it is worth mentioning again that some percentage of functionality in the library is in the form of macros rather than functions. This is due to both the ownership and borrowing mechanism in Rust, as described in 2.1.1, and the way the **Peripherals** structure is organized in the **esp-hal** driver [11]. This certainly complicates the API somewhat, because if an invalid parameter is passed, the error message will not be as intuitive as in the case of a function.

Also with the use of macros comes the need to import non-obvious libraries into the **Cargo.toml** file in the user's project. For example, a user uses some macro that uses some functionality from the **embedded-hal** library. In this case, the compiler will generate an error like „`use of undeclared library <crate_name>`“, which makes it absolutely clear which library should be included, and the user can also look in the **Cargo.toml** of the project **esp-ward** to find out the specific version of the driver needed (which is a common practice in Rust development). I have partially reduced this problem by using the full path to a particular module or function inside all macros, which saves the user from having to import in code (`use <crate_name>`, see 2.1.1). However, it is not possible to completely

get rid of this consequence of necessary use of macros.

Initially, there was idea to create a structure similar to the one specified in Listing 5.1.

```
pub struct ChipConfig<'a> {
    // The fields here represent the peripherals that have been initialized
    pub clocks: esp_hal::clock::Clocks<'static>,
    pub gpio: esp_hal::gpio::Pins,
    pub periph: &'a esp_hal::peripherals::Peripherals,
}
```

Listing 5.1: Unsucceeded prototype of ChipConfig structure.

Such an approach could have further simplified on-board initialization, but such an approach crashed with the security and Ownership and Borrowing concept of the Rust language described in 2.1.1. For a long time I tried to get the code working, inventing various tricks, but nothing worked at the moment of testing. So I decided to initialize all internal peripherals simply through macros and just have their instances in the code (see 3.4.1).

I was also trying to create a mechanism that would allow peripherals that support both SPI and I2C buses (like the BME280, see 3.2) to be initialized on one of them with a single function. Such a mechanism would be a great innovation and something new, because before this sort of thing was handled by simply having two structures for two different buses in the driver for that peripheral. The prototype is shown in Listing 5.2. However, after much experimentation, the concept was not successful due to various limitations and the fact that the size of the items in the structure must be known at compile-time [26].

```
pub enum Bme280Interface {
    I2C(ExternalBME280_i2c<I2C<'static, I2cInstance>>),
    SPI(ExternalBME280_spi<Spi<'s, SpiInstance, FullDuplexMode>>),
}

pub struct Bme280Sensor {
    inner: Bme280Interface,
}
```

Listing 5.2: Unsucceeded prototype of adaptable bus concept.

Also, it was not possible to implement SPI mode for **BME280** chip due to incompatibility of this bus driver in **esp-hal** [11] and used sensor library.

One of the most insidious problems was the unexpected inability to use the `#[cfg(...)]` attribute, which filters library code based on the feature passed to the macro. In cases like the one described in Listing 5.3, the last option was always chosen.

```
let mut x_axis = adc1_config.enable_pin(
    #[cfg(any(feature = "esp32"))]
    $pins.gpio32.into_analog(),
    #[cfg(not(feature = "esp32"))]
    $pins.gpio1.into_analog(),
    esp_hal::analog::adc::Attenuation::Attenuation11dB,);
```

Listing 5.3: Unsucceeded pin selection with condition in macro.

In this code, **pin 32** should have been selected for the ESP32 configuration and **pin 1** in all other cases, but always in such constructions inside the macro, regardless of the activated feature, the last option in order was selected. I solved this problem by creating several macros in the way shown in Listing 5.4.

```
#[cfg(not(feature = "esp32"))]
#[macro_export]
macro_rules! get_x_adc_pin {
    ($pins:expr) => {
        $pins.gpio1.into_analog()
    };
}

#[cfg(feature = "esp32")]
#[macro_export]
macro_rules! get_x_adc_pin {
    ($pins:expr) => {
        $pins.gpio32.into_analog()
    };
}
```

Listing 5.4: Solution for conditional attributes in macros issue.

This way, the macro itself is selected depending on the configuration, not any of its macros. In the code that follows, it is the one that is used, which no longer creates such problems.

It should also be noted that since all the **esp-rs** drivers are under active development, they may be unstable in one way or another. Therefore, fixing the library on certain versions of them is a necessity. The specific versions can be found in the **Cargo.toml** file of the project. This restriction is introduced for stability reasons, because often breaking changes can occur that will break the whole project. As stated in 2.4.2, the project is in an active stage of development, so fixing on a version is a necessity. Further, in the lifecycle of the project, there will be constant updates to the most current stable versions of these drivers. Exact versions are listed in Listing 4.4.

Chapter 6

Conclusion

To summarize, it is necessary to mention the goal of the project. The main goal was to offer an open-source library that would simplify the development of embedded sensor applications in the Rust language on microcontrollers from Espressif Systems. An architecture and specific methods were designed that utilize the broad capabilities of this language. An API was developed which, on the one hand, simplifies for the library user the use of sensors and other peripherals when creating his own application, and on the other hand, for the developer who was offered a convenient interface to work directly with the hardware itself. The project affects both internal peripherals of chips (e.g. SPI, I2C, simplified initialization of allocator) and a set of proposed external devices. Access to them was unified and simplified, all work with deep settings is done in the library itself, and the user works with a clear interface. As one of the testers said, the library with its sense and meaning resembles a similar Arduino project, which is a reference and the choice of many users when developing embedded applications. The project went a bit beyond the scope of the task – support was added not only for sensors, but also for some displays and input devices, such as buttons, joysticks and motion sensors. An important aspect and a very useful extension is also the addition of support for network functionality, allowing user systems to utilize Wi-Fi and MQTT communication. All this allows my library to be suitable not only for basic sensor applications, but also for more complex systems.

One of the tasks was to post the project on GitHub under the APACHE license. Also, the library was intended to be an open-source project, implying community involvement in the development. Work was also done in these areas – the structure of the project creates convenience and clearly shows the logic of modules, which makes it easy to navigate in the repository. For those interested in the development of the project, a detailed guide on how to do it has been compiled. A detailed README provides information on what functionality is supported by the library and how to start using it in your projects, including troubleshooting. As part of the public part, a website hosted by GitHub Pages has been created. The landing page is designed in the classic Rust Docs style, which gives the user a sense of unified design. It provides access to the documentation, which the interviewed users praised for how detailed and clear the API is described.

Considering all said above, the project can be called a success. It not only fulfills all the requirements, but also significantly expands the support offered.

6.1 Possible Extensions

Possible improvements to the library functionality include, for example, expanding the range of supported peripherals. Also add support for more networking functionality, including some esp-specific ones like esp-now[1].

[1] esp-now communication protocol: https://www.espressif.com/en/solutions/low-power-solutions/esp-now

Bibliography

[1] *Crate std* [online]. [cit. 2024-04-18]. Available at: https://doc.rust-lang.org/std/index.html.

[2] *Embedded_hal* [online]. [cit. 2024-04-18]. Available at: https://crates.io/crates/embedded-hal.

[3] 0xDEADBEEF(nickname). *Rust (programming language)* [online]. 2024 [cit. 2024-04-18]. Available at: https://en.wikipedia.org/wiki/Rust_(programming_language).

[4] Adafruit-Industries, LLC. *Adafruit Official Website* [online]. [cit. 2024-04-18]. Available at: https://www.adafruit.com.

[5] Aparicio, J. *The Embedonomicon (CC BY-NC-SA 4.0)* [online]. [cit. 2024-04-18]. Available at: https://docs.rust-embedded.org/embedonomicon/preface.html.

[6] Blandy, J., Orendorff, J. and Tindall, L. F. S. *Programming Rust.* 2nd ed. O'Reilly Media, Inc., June 2021 [cit. 2024-04-18]. ISBN 978-1-492-05259-3.

[7] Cramer, T. *Asynchronous Programming in Rust* [online]. [cit. 2024-04-18]. Available at: https://rust-lang.github.io/async-book/.

[8] Crichton, A., Klabnik, S. and Nichols, C. *The Cargo Book* [online]. [cit. 2024-04-18]. Available at: https://doc.rust-lang.org/cargo/.

[9] de Bakker, B. *How to use an HC-SR04 Ultrasonic Distance Sensor with Arduino (CC BY-NC-SA 4.0)* [online]. [cit. 2024-04-18]. Available at: https://www.makerguides.com/hc-sr04-arduino-tutorial/.

[10] esp-rs. *Esp-alloc* [online]. [cit. 2024-04-15]. Available at: https://github.com/esp-rs/esp-alloc.

[11] esp-rs. *Esp-hal* [online]. [cit. 2024-04-15]. Available at: https://github.com/esp-rs/esp-hal.

[12] esp-rs. *Esp-idf-hal* [online]. [cit. 2024-04-15]. Available at: https://github.com/esp-rs/esp-idf-hal.

[13] esp-rs. *Esp-pacs* [online]. [cit. 2024-04-15]. Available at: https://github.com/esp-rs/esp-pacs/tree/main.

[14] esp-rs. *Esp-rs docs* [online]. [cit. 2024-04-15]. Available at: https://docs.esp-rs.org/esp-hal/.

[15] ESP-RS. *Esp-wifi* [online]. [cit. 2024-04-15]. Available at: https://github.com/esp-rs/esp-wifi/tree/main/esp-wifi.

[16] ESP-RS. *Espflash* [online]. [cit. 2024-04-15]. Available at: https://github.com/esp-rs/espflash.

[17] ESP-RS. *The Rust on ESP Book (CC BY-NC-SA 4.0)* [online]. 2018 [cit. 2024-04-15]. Available at: https://docs.esp-rs.org/book/.

[18] ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD.. *Esp-bsp* [online]. [cit. 2024-04-18]. Available at: https://github.com/espressif/esp-bsp.

[19] ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD.. *Esp-idf* [online]. [cit. 2024-04-18]. Available at: https://github.com/espressif/esp-idf/tree/master.

[20] ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD.. *SoCs* [online]. [cit. 2024-04-18]. Available at: https://www.espressif.com/en/products/socs.

[21] ESPRUINO. *PCD8544 LCD driver (Nokia 5110)* [online]. [cit. 2024-04-18]. Available at: https://www.espruino.com/PCD8544.

[22] EVANS, J. *A couple of Rust error messages* [online]. [cit. 2024-04-18]. Available at: https://jvns.ca/blog/2022/12/02/a-couple-of-rust-error-messages/.

[23] GANSSLE, J. *A Guide to Debouncing, or, How to Debounce a Contact in Two Easy Pages* [online]. [cit. 2024-04-18]. Available at: https://www.ganssle.com/debouncing.pdf.

[24] ILI TECHNOLOGY CORP.. *A-Si TFT LCD Single Chip Driver 240RGBx320 Resolution and 262K color* [online]. [cit. 2024-04-18]. Available at: https://cdn-shop.adafruit.com/datasheets/ILI9341.pdf.

[25] JETBRAINS. *TeamCity CI/CD Guide* [online]. [cit. 2024-04-18]. Available at: https://www.jetbrains.com/teamcity/ci-cd-guide/.

[26] KLABNIK, S. and NICHOLS, C. *The Rust Programming Language*. 2nd ed. No Starch Press, February 2023 [cit. 2024-04-18]. ISBN 978-1-7185-0311-3.

[27] LAB, M. S. *Wrangling Web Contributions: How to Build a CONTRIBUTING.md* [online]. [cit. 2024-04-18]. Available at: https://mozillascience.github.io/working-open-workshop/contributing/.

[28] LESIŃSKI, K. *Rust crates ecosystem statistics* [online]. [cit. 2024-04-18]. Available at: https://lib.rs/stats.

[29] MAIDAN, M. and PANASENKO, Y. *Rust market overview: reasons to adopt Rust, Rust use cases, and hiring opportunities* [online]. [cit. 2024-04-18]. Available at: https://yalantis.com/blog/rust-market-overview/.

[30] MARPAUD, C. *OxidESPark* [online]. [cit. 2024-04-18]. Available at: https://gitlab.com/cyril-marpaud/oxide-spark.

[31] MARSHALL, D. *Programming with Rust*. Addison-Wesley Professional, December 2023 [cit. 2024-04-18]. ISBN 978-0-13-788965-5.

[32] Michálek, J. *Esp-bsp-rs* [online]. [cit. 2024-04-18]. Available at: https://github.com/georgik/esp-bsp-rs.

[33] Mikhailov, K. *Crispy Click (Wokwi example)* [online]. [cit. 2024-04-18]. Available at: https://wokwi.com/projects/341706650098336338.

[34] Mikhailov, K. *Esp-clock-nostd* [online]. [cit. 2024-04-18]. Available at: https://github.com/playfulFence/esp-clock-nostd/tree/main.

[35] Munns, J. *The Embedded Rust Book (CC BY-NC-SA 4.0)* [online]. [cit. 2024-04-18]. Available at: https://docs.rust-embedded.org/book/intro/index.html.

[36] Rust Embedded Graphics. *Embedded_graphics* [online]. [cit. 2024-04-18]. Available at: https://crates.io/crates/embedded-graphics.

[37] Sadel, J. *Esp32c3-no-std-async-mqtt-demo* [online]. [cit. 2024-04-18]. Available at: https://github.com/JurajSadel/esp32c3-no-std-async-mqtt-demo.

[38] Sadel, J. Rust + Embedded: A Development Power Duo. *Elektor Magazine* [online]. December, 2023, [cit. 2024-04-18].

[39] Schuster, P. *Max_7219_led_matrix_util* [online]. [cit. 2024-04-18]. Available at: https://crates.io/crates/max-7219-led-matrix-util.

[40] Semenuk, B. *Exploring The Rust Standard Library* [online]. [cit. 2024-04-18]. Available at: https://marketsplash.com/tutorials/all/rust-standard-library/.

[41] Sequeira, R. *Learning Rust: Understanding Zero-Cost Abstraction with Filter and Map* [online]. May 23, 2023 [cit. 2024-04-18]. Available at: https://ranveersequeira.medium.com/learning-rust-understanding-zero-cost-abstraction-with-filter-and-map-e967d09fff79.

[42] Thompson, C. *How Rust went from a side project to the world's most-loved programming language* [online]. February 2023 [cit. 2024-04-18]. Available at: https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/.

Appendix A

Contents of the storage medium

- **/esp-ward** - source code of the library.
- **/compare** - a folder with visual examples of how the library has simplified code writing.
- **/latex** - source code of the bachelor's thesis text
- **xmikha00.pdf** - text of bachelor's thesis.

Appendix B

Project related links

- https://github.com/playfulFence/esp-ward - project repository.
- https://playfulfence.github.io/esp-ward/ - library's website, hosted on GitHub Pages.
- https://playfulfence.github.io/esp-ward/docs/esp_ward/index.html - API documentation.

B.1 Comparison Examples

- https://wokwi.com/projects/341706650098336338 - official `Crispy Click` example.
- https://wokwi.com/projects/396547396717778945 - `Crispy Click` example remade with my library.
- https://wokwi.com/projects/344869867332043347 - official text scrolling on MAX7219 example.
- https://wokwi.com/projects/396544824213521409 - text scrolling on MAX7219 example, remade with my library
- https://wokwi.com/projects/357451677483992065 - official `esp-clock` example.
- https://wokwi.com/projects/396552224931760129 - example, similar to `esp-clock`, remade with my library (may not work due to a bug with internal Wokwi builder, in this case - used as a proof of concept).
- https://wokwi.com/projects/342697409287029332 - official `etch-a-sketch` example.
- https://wokwi.com/projects/396539527833650177 - `etch-a-sketch` example, remade with my library (may not work as expected due to incompatibility of ADC1 and Wokwi, used as a proof of concept).

Appendix C

Feedback from Espressif Systems

In order to get an opinion from the company about my project, I personally asked the team leader of the **esp-rs** to write a short text with his conclusion.

> *„esp-ward is a high-level wrapper for esp-hal and esp-wifi. esp-ward allows for rapid prototyping and experimentation in the Espressif Rust ecosystem by abstracting over common actions many users will need to bring up their application. Rust is notoriously hard to learn, so having esp-ward as an option for new users is a great boon to the ecosystem.*
>
> *esp-ward has a CI workflow to ensure it's always building with the latest esp-rs crates and helpful self-hosted documentation under* `https://playfulfence.github.io/esp-ward/`*. Within the repository, there are several helpful examples that showcases the simplicity of applications when using esp-ward.*
>
> *esp-ward doesn't just abstract over esp-rs crates, it adds valuable integrations into other third-party crates and drivers. The most common feedback we receive as the maintainers of esp-rs is that it's hard to glue all the separate pieces of the ecosystem together; esp-ward can help alleviate this issue by providing these integrations. Along with the integrations, esp-ward adds its own project building blocks such as a PIR sensor, Ultrasonic Distance sensor (HC-SR04) and button abstractions. There are also networking abstractions for WiFi and MQTT.*
>
> *Finally, esp-ward holds the same values as we do when it comes to community contributions, after all, most of the esp-rs team is from the community; it has a thorough contribution guide to ensure that community contributions can be made frictionlessly. "*
>
> — *Scott Mabin*