Rust como lenguaje de programación unificado para IoT
Rust as a unified programming language for IoT

Trabajo de Fin de Máster
Curso 2022–2023

Autor
Marco Romera Corral

Director
Francisco Daniel Igual Peña
Luis M. Costero Valero

Máster en Internet de las Cosas
Facultad de Informática
Universidad Complutense de Madrid

Rust como lenguaje de programación unificado para IoT
Rust as a unified programming language for IoT

Trabajo de Fin de Máster en Internet de las Cosas
Departamento de Arquitectura de Computadores y Automática

Autor
Marco Romera Corral

Director
Francisco Daniel Igual Peña
Luis M. Costero Valero

Convocatoria: *Septiembre 2023*
Calificación: 9,5 (SOBRESALIENTE)

Máster en Internet de las Cosas
Facultad de Informática
Universidad Complutense de Madrid

1 de septiembre de 2023

Dedicatoria

A mi familia y amigos.

Agradecimientos

Agradezco a mi familia por darme ánimos y todo su apoyo con este proyecto en todo momento.

Gracias a mis directores Francisco y Luis el esfuerzo tiempo que han dedicado a ayudarme y orientarme con este trabajo. También al resto de equipo de profesores del máster por su gran labor.

Resumen

Rust como lenguaje de programación unificado para IoT

El presente trabajo analiza el uso del lenguaje de programación Rust en el ámbito de Internet de las Cosas (IoT). Se plantea el uso del Rust en las áreas de computación en la nube, gateways y nodos edge, y programación embebida de nodos IoT. Se analizan las propiedades de seguridad de memoria y concurrencia de Rust y su adecuación al entorno IoT.

En particular se desarrollan dos ejemplos de aplicación en un nodo IoT basado en placa de desarrollo ESP32-C3-DevKit-RUST-1 con la plataforma RISC-V. El nodo dispone de sensores internos de temperatura, humedad y un acelerómetro (IMU) así como sensores I^2C externos. Los ejemplos comparan modelos de concurrencia distintos. El primer ejemplo se basa en la librería estándar de Rust implementada con el sistema ESP-IDF (Espressif IoT Development Framework) y el uso de threads y servicios de un sistema RTOS (Real Time Operating System). El segundo ejemplo utiliza programación asíncrona, prescindiendo de la librería estándar (`no_std`) y por tanto del sistema operativo, y utiliza la librería Embassy que facilita un ejecutor asíncrono.

Palabras clave

Rust, IoT, concurrencia, seguridad, RISC-V, I2C, WebAssembly, asíncrono, fsm

Abstract

Rust as a unified programming language for IoT

This work analyzes the use of the Rust programming language in the field of IoT (Internet of Things). The use of Rust in the areas of cloud computing, gateways and edge nodes and embedded programming in the IoT node is discussed. The memory safety and concurrency models of Rust and its application to IoT are analyzed.

Two application examples are developed for an IoT node based in the ESP32-C3-DevKit-RUST-1 development board which uses the RISC-V platform. The node has internal sensors for temperature and humidity and accelerometer (IMU) as well as externally connected I^2C sensors. The examples compare different concurrency models. The first example is based on the Rust standard library (`std`) that, in this case, is implemented with the ESP-IDF, and the use of threads and services provided by this RTOS system. The second example uses asynchronous programming, and it does not use the standard library (`no_std` crate), so there is no operating system. It uses the Embassy library that provides an executor for asynchronous tasks.

Keywords

Rust, IoT, concurrency, security, RISC-V, I2C, WebAssembly, asynchronous, fsm

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
  - 1.1.1 Memory safety . . . 2
  - 1.1.2 Concurrency safety . . . 3
  - 1.1.3 Outdated toolchain . . . 3
  - 1.1.4 High level alternatives . . . 3
- 1.2 Objectives . . . 4
- 1.3 Code availability . . . 5
- 1.4 Document structure . . . 5

2 State of the Art **7**
- 2.1 The Rust language . . . 7
  - 2.1.1 Language characteristics . . . 7
  - 2.1.2 WebAssembly . . . 8
  - 2.1.3 Infrastructure and toolchains . . . 8
- 2.2 Concurrency in Rust . . . 10
  - 2.2.1 Asynchronous Rust (cooperative). async/await . . . 11
  - 2.2.2 Preemptive (threads) . . . 11
- 2.3 Rust in IoT deployments . . . 12
  - 2.3.1 Rust in servers . . . 12
  - 2.3.2 Rust in edge nodes . . . 12
  - 2.3.3 Rust in mobile devices . . . 13
  - 2.3.4 Rust in embedded development, IoT nodes . . . 13

3 Hardware setup of an IoT node prototype **15**
- 3.1 Hardware setup . . . 15
  - 3.1.1 ESP32-C3-DevKit-RUST-1 development board . . . 16
  - 3.1.2 Temperature and humidity sensor HTU21D . . . 16
  - 3.1.3 Inertial measurement unit sensor ICM-42670-P . . . 16
  - 3.1.4 Temperature and humidity sensor SHTC3 . . . 17
  - 3.1.5 I^2C (Inter-Integrated Circuit communications bus) bus hardware requirements . . . 17

4 An `std` application with *esp-idf-sys* **19**

4.1 Running the example . . . 20
4.2 Tasks organization and architecture . . . 22
4.3 Finite states machines in Rust . . . 22
4.4 Hardware resources . . . 24
4.5 Passing data between threads using messages . . . 25
4.6 Scheduling sensor reading with timers . . . 26

5 A `no_std` application with Embassy and *esp-hal* 29
5.1 Running the example . . . 29
5.2 `no_std` crate . . . 30
5.3 Hardware access . . . 31
5.4 Hardware and global state . . . 32
5.5 Concurrency with Embassy . . . 34
5.6 Networking and asynchronous wifi handling . . . 35
5.7 MQTT client . . . 36
5.8 I2C bus sharing . . . 37

6 Conclusions and Future Work 41
6.1 Rust as a language for IoT . . . 41
6.1.1 Efficiency . . . 41
6.1.2 Security and reliability . . . 41
6.1.3 Adoption and popularity . . . 42
6.2 Maturity of the ecosystem . . . 42
6.3 Toolchain support. Flashing and debugging workflows . . . 42
6.4 IoT resources . . . 43
6.5 Embedded programming and concurrency . . . 44
6.6 Code organization . . . 45

Bibliography 47

A Console logs for sensor (`std`) example 49

B Console logs for *embsensor* (`no-std`) example 57

Glossary and acronyms 61

List of figures

3.1 IoT node. Breadboard with the components 16
3.2 SGP30 for CO_2, CCS811 for CO_2 and TVOC and HTU21D temperature sensors 17
3.3 I^2C clock signal rise time with included 10 $k\Omega$ resistors 17
3.4 I^2C clock signal rise time with 1 $k\Omega$ resistors 18

4.1 Compilation and flashing of example 21
4.2 MQTT client subscribed to topic /rust/temperature 21
4.3 MPSC channel 26

5.1 MQTT client receiving temperature data from the node 30
5.2 MQTT client publishing command for the node 30

Listings

4.1 Cargo configuration file (`.cargo/config.toml`) . . . 19
4.2 Console log (UART) during Initialization . . . 20
4.3 Erase flash memory (NVS storage) . . . 20
4.4 Compilation and flashing commands . . . 20
4.5 HTTP connection for provisioning . . . 21
4.6 Rust struct representing the finite state machine . . . 23
4.7 Rust enum that holds states . . . 24
4.8 Rust enum that holds events . . . 25
4.9 Hardware resource initialization . . . 25
4.10 Thread and MPSC channel creation . . . 27
5.1 HTTP connection for provisioning . . . 29
5.2 `peripherals` reference to access hardware devices . . . 33
5.3 `StaticCell` declarations in embsens application . . . 34
5.4 `StaticCell` macro . . . 34
5.5 Spawning of Embassy tasks . . . 35
5.6 Polling of MQTT socket withou blocking . . . 36
5.7 Polling of MQTT socket without blocking . . . 36
5.8 Initialisation of I^2C bus with Embassy . . . 38
A.1 Console log of initialisation (unprovisioned) . . . 49
A.2 Console log during provisioning . . . 52
A.3 Console log after provisioning . . . 54
B.1 Console log during execution . . . 57

Chapter 1

Introduction

In the field of IoT, most of the devices that are part of a typical deployment are usually small devices with reduced resources: low power, low CPU, low memory, low bandwidth, etc. At least, this is the case for the IoT nodes. Typically, these devices are not even capable of running a complete operating system. This condition means that the applications that will run in them will need to be embedded applications.

This is the reason why most of these applications are programmed in system-oriented, low-level programming languages like C. There exist alternatives that use higher-level languages to simplify the development and allow to use languages like Python (*MicroPython* [1]) or Javascript (*Espruino* [2]), but always sacrificing performance, memory footprint or efficiency, that in many cases are critical for these devices.

Until recently, the options for low level programming were reduced basically to C or C++. To this day, most of the resources available like drivers and libraries are written in C/C++. Although they are very powerful languages, they have their flaws specially concerning reliability and security. C (and therefore C++) language allows free and direct access to memory, which regarding embedded programming is a big advantage, but also a considerable danger.

The development of Rust [3], a multi-paradigm, general-purpose programming language that emphasizes performance, type safety, and concurrency has recently opened a huge hype on programming in general. Rust was created in 2009 and the first stable release is from 2015, but it has been in recent years that it has taken momentum and wide adoption, especially in the embedded development field.

The reason is that Rust gathers the performance (CPU and memory wise) that C has, but without the usual risks associated with low level programming. Rust features mechanisms that enforce memory safety –ensuring that all references point to valid memory– without requiring the use of a garbage collector or reference counting present in other memory-safe languages like Java.

It is important to note that these checks are performed at compilation time, not at runtime. That is generally expressed with the term "zero cost abstractions", which means that the compiled code is safe but it is not more complex than an equivalent code generated by C, for example. This is a huge advantage, because both the performance, reliability and security are obtained together. Usually one had to choose to give up any of these features.

But not everything is perfect. The Rust language can be hard to learn, at least from the point of view of someone with experience with C and similar languages. Rust rules (its "borrow checker", for example) will enforce the programmer to thing more deeply in the

working of the program from the very beginning. The program will be harder to develop, because most of the time, the compiler will signal a flaw in the design with its compilation errors before we can even run it.

However, this is a good feature as errors and potential problems are detected sooner and not at runtime or, even worse, in the production phase. But there is a steep learning curve until the programmer has the experience to fluently program in Rust.

1.1 Motivation

There are several problems associated with the use of low level programming languages such as C. We will review some of them and comment on how Rust can be helpful to solve them.

There are some estimations about the impact and source of common safety errors. Microsoft estimates that about 70% of all CVEs (Common Vulnerabilities and Exposures) are memory safety issues [4]. A study by Apple [5] found that between 60 and 70% of vulnerabilities in iOS and macOS are memory safety vulnerabilities. Google estimates this number in 90% for Android [6]. For Linux kernel vulnerabilities it is two thirds [7]. And most 0-day vulnerabilities are also memory safety problems [8].

Some of the most famous vulnerabilities like WannaCry [9] or HeartBleed [10] vulnerabilities were memory safety issues.

In IoT, the impact could be even even bigger. The IoT deployments make use of custom developed firmwares for specific hardware devices that don't have so big base of users as the code from operating systems like Android, Linux or macOS. This firmwares are usually not subject to the same level of scrutiny and their long term maintenance is a challenge. They are usually deployed to small devices directly connected to Internet and sometimes in unprotected networks. They represent a big surface of attack and any security flaw can be easily exploited.

The main issues that motivate this work are the search for an alternative language that provides memory safety and concurrency safety by design. A modern toolchain for building, handling dependencies and its repositories is also desirable.

1.1.1 Memory safety

Memory safety issues can be abused by hackers or lead to hard to debug bugs with undefined behavior. This kind of problems are usually the result of one of the following errors:

- Not freeing memory after allocation: it leads to a "memory leak" in which the resources are exhausted because memory cannot be reused.
- Freeing already-freed memory (double-freeing): This is often caused by lack of knowledge of who owns what. It can cause segmentation faults.
- Invalid memory access, either reading or writing. Use of uninitialized, NULL, or already-freed pointers:
  - Buffer overflow: reading in a region of memory not allocated. For example reading further than the end of an array. C makes it possible to access positions in an array that are grater than the maximum size. It is a variant of the previous case.
  - Use after free: it uses a piece of memory even after it has been already freed.

– Use of uninitialized memory: it uses a piece of memory even before there is stored anything on it.

1.1.2 Concurrency safety

In many low level languages, concurrent programming is a hassle. This is because access to memory from different threads can happen at any time, leading to errors (known as "races"). The synchronization to access memory must be handled manually by the programmer using language tools like mutex, semaphores, queues, etc.

This is a common source of error, and these errors are difficult to detect and debug. They are dealt with at run time and can introduce unnoticed security problems.

In Rust, the checks performed at compilation time ensure that some invariants are met (i.e., there is only one owner responsible for each resource, and only the owner has read and write access to it). This allows for a much simpler concurrent programming. References to data can be used for read or read/write access while maintaining thread security.

1.1.3 Outdated toolchain

As applications increase their complexity, including handling of multiple network connections with WiFi, Bluetooth, Lora, etc, AI workloads and so on, it is more important to have access to a greater number of libraries. One problem with C/C++ is that there is not a dependency manager among the tools included in its toolchain. There are tools such as a compiler, a debugger, etc., but there is not an easy way to automatically seek, download, compile and install libraries.

As the number of libraries increases, this management is crucial for the development of a rich ecosystem of applications, drivers and other resources.

In Rust, *Cargo*, the build system and dependency manager, is in charge of this functionality. This, combined with the public repository *crates.io* [11] allows to download, compile and link all the dependencies for a given crate [1] with the only requirement of listing them in a configuration file (*Cargo.toml*).

`Cargo` handles also the compilation process, taking care of the exact releases of every dependency even if the same library is required more than once recursively but with different versions. This makes the compilation process very easy once the toolchain is installed.

This has allowed for the rising of a rich ecosystem of libraries, and a huge number of drivers for many architectures of processors, IoT devices, sensors, actuators, etc. This is very big advantage with respect to what is available in C/C++ and comparable to systems found in other modern higher level languages like Python (`pip`), JavaScript (`NPM`), etc.

The *crates.io* repository is growing very quickly regarding crates that attain to IoT, with projects like *Embassy* [12] or support for IoT protocols from crates like *serde* [13]. Some projects are being developed for direct communication with IoT platforms like Azure or AWS.

1.1.4 High level alternatives

There are platforms that allow programming IoT node applications using high level languages such as Python, JavaScript and others. These are appropriate when performance

[1] A crate is the term used in Rust for a tree of modules that produces a library or executable

requirements are not critical. If the node needs to be efficient energy wise or computationally they are not adequate.

For example, the same program in Rust or C takes much lower number of instructions to execute than in Python for the same task. This means much lower consumption (or much longer battery duration) which is critical in many applications.

Rust can be used for system low level programming, but has very powerful abstractions that combined with the libraries ecosystem make it an option for tasks that were normally addressed with high level languages. This makes it ideal for some IoT tasks in servers or gateways that are not embedded programming.

1.2 Objectives

The goal of this work is to develop an IoT node to showcase the use of Rust to solve the common challenges in embedded development. The chosen hardware for the node is the Espressif ESP32-C3-DevKit-RUST-1 development board. This board includes the ESP32-C3 microcontroller that uses the RISC-V architecture. The reason for choosing this architecture instead of the Xtensa processors is that RISC-V is directly supported by the Rust compiler.

The node will access some sensors through the I^2C data bus and will send the data to a server using the MQTT (MQ Telemetry Transport) protocol. This allows to analyze the way in which hardware is accessed in Rust in a safe way. Some of the sensors are included in the board itself and some are externally connected.

The connection to the network will use the WiFi included in the ESP32-C3 board and various libraries are used to handle this connection.

The node is implemented in two different examples to show the two fundamental types of applications that can be created with Rust depending on the availability of the `std` library: `std` and `no_std` crates.

Different types of concurrency models are compared, and the libraries that are available: `std` threads with ESP-IDF, asynchronous code with Embassy, and minimal RTOS with RTIC (Real-Time Interrupt-driven Concurrency). There are many different options to choose from in the Rust embedded ecosystem. The criteria for choosing between them has been to select the more popular and well established or the ones that seem to have great potential or being innovative.

The node will be complex enough to demonstrate how to solve the usual problems found in IoT. Also, the implementation of each feature is minimal and not intended to be production ready but only to show the way common problems are solved in Rust.

- General organization of a complex node by use of a FSM (Finite State Machine). Use of Rust abstractions to represent the FSM.
- Hardware access: drivers, HAL (Hardware Abstraction Layer) and PAC (Peripheral Access Crate).
- Shared use of resources, specially hardware resources from different tasks or threads.
- Network management, in particular WiFi connections.
- Multitasking and synchronization.
- Provisioning (credentials and initial node configuration).

Other common problems are left aside in this work, although there are many libraries that allow to handle them already in Rust. These are some subjects left:

- Over the air firmware updates (OTA (Over The Air firmware update)).
- Other common network connections like Bluetooth, LoRa (Long Range), mesh networks, etc.
- Power saving management.

1.3 Code availability

All code from this project can be found in the repository *marcormc/rustiot* [14].

1.4 Document structure

The rest of the document is organized as follows:

In chapter 2, the general characteristics of Rust are presented, specially those that are more related with the IoT field. WebAssembly as a target platform of Rust and its role in IoT is discussed. Special attention is paid to concurrency, that is very important for embedded development.

Chapter 3 deals with the hardware setup of the IoT node used to implement the two examples presented in this work. The development board and sensors are connected together in a breadboard. The setup of the I^2C bus and its hardware requirements are analyzed.

Chapters 4 and 5 present the software implementation of both examples. The first one using a `std` crate and the second one with a `no_std`. They differ fundamentally in the way that concurrency is achieved, with an RTOS system and with asynchronous programming.

Chapter 6 presents the conclusions learned from the experience of using Rust to create an IoT node and compares the two solutions tested.

Appendix A and B show the console output of both example nodes during boot and various operations.

Chapter 2

State of the Art

2.1 The Rust language

Rust is one of the safest languages. It is a high-level multi-paradigm language, that can be used for functional and imperative or object oriented programming. Also, it features a modern tooling for compilation, debugging and dependency management system.

We will review some of its main characteristics, focusing on those that make it safer than other languages and specifically those that are more relevant for embedded programming.

2.1.1 Language characteristics

The safety guarantee is one of the most important aspects of Rust; Rust is memory-safe, null-safe, type-safe, and thread-safe by design.

Rust ensures memory safety at compile time by using the concept of ownership and borrow checker that are built into the compiler. The compiler makes it impossible to produce code that is not memory safe. There is no concept of NULL pointer: Rust provides the `Option enum` which is a safe alternative to NULL. NULL pointer exceptions are impossible. This mechanism does not impact performance because the analysis is performed during compilation.

Rust avoids the problems usually associated with manual memory management but at the same time there is no garbage collector at runtime. This allows to achieve execution speeds similar to C/C++ programs while avoiding potential memory-related errors.

Rust is a statically-typed language, as many modern languages. It also allows some degree of dynamic typing with keyword `dyn`, but even in those cases, the compiler ensures type safety. Thread safety is also guaranteed and the language provides channels, mutexes, and ARC (Atomically Reference Counted) pointers.

One very important feature of Rust, specially for embedded programming and for IoT is that, although the compiler ensures that only safe code can be produced, there is an override mechanism when it is necessary. There are times, specially accessing hardware, in which it is necessary to write code that the compiler cannot validate as safe. It does not mean it is not safe: it means that the programmer takes the responsibility to provide the guarantees that are normally given by the compiler.

The `unsafe` keyword allows to produce code that the compiler would not be able to check by itself. This brings all the power of systems programming languages such as C, but limits the delicate parts that are subject to possible memory errors to tiny parts of the

code that are easy to manage. For example, HAL *crates* provide secure high level access to hardware and the `unsafe` code within them is small and localized. It is relatively easy to ensure that this code is safe because its small size.

Rust has also other characteristics that make it unique:

- It is **immutable** by default. Variables are read only unless they are declared as mutable.
- It has **advanced generics**, traits and types. This is similar to C++ templates but it has type inference that makes writing code much less verbose and convenient.
- Rust has very powerful **macros** that allow the creation of declarative and procedural macros. It supports meta-programming, and mini languages for specific purpose can also be implemented.
- It has support for **pattern matching** commonly found in high level languages like Python or Perl.
- Rust code can call other languages and be called from them.
- **Documentation** is produced by the *cargo* build system and extracted directly from code. It can be easily and automatically published to repositories like *crates.io* [11]. Documentation of the language and standard libraries is excellent and well-maintained.
- It has an excellent community and ecosystem. This ecosystem is growing fast.

2.1.2 WebAssembly

Rust code can be compiled for almost any platform. This is because the Rust compiler produces what is called *intermediate representation code* that is platform independent. It is the task of the LLVM compiler to translate this intermediate code to the machine code of each platform. So all platforms supported by LLVM (that are a lot) are possible targets.

One very special target is WASM (WebAssembly), that is supported for modern web browsers to allow execution from web clients. It has very high performance, much higher than JavaScript, allowing for new web application opportunities. The WASM has set up a standard that is also being leveraged for other purposes like cloud computing or embedded execution.

2.1.3 Infrastructure and toolchains

Rust has a very modern infrastructure for building, debugging, managing dependencies, generating documentation, etc. A very important aspect that is critical for any embedded project is the configuration of dependencies.

Rust libraries are evolving very quickly, specially the ones more related with the IoT field. New drivers appear continuously, and many projects are not yet stable. Some APIs, interfaces and traits evolve from version to version. Sometimes it is necessary to use crates that are not even published to *crates.io* repository. This is possible by referencing their Git repositories directly as a dependency. A particular commit or tag can be set as reference.

Many features are already available in Rust but not in the stable release of the language. There are *stable* and *nightly* releases and *cargo* can be configured to employ any of these toolchains. Stable releases have their point releases as a reference, but the nightly release

is a moving target. If used for production, it is possible to reference a nightly release for a particular date. This way the application is guaranteed to compile without errors.

If a fixed date is not specified for the toolchains (or a stable one) and all dependencies are precisely fixed, there is the chance that the application yield errors during compilation due to changes in some of the libraries, the Rust compiler or any component of the toolchain.

In Rust, it is very easy to manage all this versions thanks to files `Cargo.toml`, `Cargo.lock` and `rust-toolchain.toml`. Together they produce a time capsule that describes exactly all that is needed for compilation and guarantees an error-free process. Tools like *rustup* and *cargo* handle the installation of any required version of the toolchain.

2.2 Concurrency in Rust

One of the strongest capabilities of Rust is concurrent programming. Because of its memory safety features (like the borrow checker), concurrent programming is easier than in other languages. The checks that the Rust compiler performs at compile time guarantee the safety of the concurrent memory accesses. This feature avoids a common source of errors difficult to debug, even more in embedded systems programming.

Rust provides useful tools to support concurrency, namely:

- The `std` library provides threads to Rust. They are implemented using the operating system threads.
- Message-passing concurrency, where channels send messages between threads.
- Shared-state concurrency, where multiple threads have access the same data.
- The Sync and Send traits, which extend Rust's concurrency guarantees to user-defined types.

Concurrent programming means that different parts of a program execute independently but not necessarily at the same time. In *parallel programming* different tasks execute at the same time. This is an important distinction because parallel programming is preemptive multitasking, meaning that any task can be interrupted or resumed at any given time to share the processor time. It is the operating system who schedules these interrupts. In embedded devices it is usually an RTOS who switches tasks execution. For a given task, it is not possible to ensure if other operations from a different task will take place in between two instructions or in the middle of one. In the case of Rust, the simplification comes from the fact that the compiler checks at compile time that the program is not performing any dangerous operation.

Other form of concurrency is *asynchronous programming*, in which tasks are executed in turns, and it is the task itself who *yields* the execution to other tasks when there is a piece of work that will take time to execute. This is common when performing input/output operations (writing and reading files or network connections, for example). It is also called co-operative multitasking because the execution of a task is not interrupted until it yields execution to others.

In those cases when the task yields execution there is a small runtime component that decides which other task to resume. This runtime is called an *executor*.

The advantage of this type of concurrency is that the runtime is usually much lighter than the operating system thread. It does not even require an operating system. There is no need for a separate stack for each thread, which saves memory. If the number of threads is small this memory is not a problem. For example if a processor has four kernels, spawning four threads makes it possible to leverage the full performance of this processor.

But many times, the number of tasks is more related to the processing of independent workloads concurrently. For example in the case of a web server, it could be desirable to deploy a big number of tasks to process the requests as soon as possible. In this case, using a thread for each task greatly limits the maximum number of possible tasks. The consumption of memory would be too high and context switching between task is slower than in asynchronous tasks.

There are other forms of concurrency apart from OS threads and asynchronous programming, like event-driven programming: coroutines, the actor model, etc. But the more

popular ones are async and threads. These can also be combined by running more than one async executor, one in its own thread.

In Rust, there are many projects that implement concurrency, specially asynchronous libraries. The language itself brings some basic tools in the form of language keywords like `async`, small libraries like *futures*, threads and some traits, but leave the implementation to user libraries. Rust for example does not integrate a built-in executor for async (there is no runtime).

More information on asynchronous programming can be found in the *Asynchronous Programming in Rust* book [15].

2.2.1 Asynchronous Rust (cooperative). async/await

These are some of the more popular asynchronous libraries for Rust:

- **tokio** [16]: Probably the best known and most widely used async library in Rust. It is for `std` crates as it requires the standard library.
- **async-std** [17]: This is the "official" library but was developed later than *tokio* so is no so popular. It is establishing traits and common interfaces for the development of other asynchronous libraries. It is a good option for new projects that doesn't depend on libraries that require *tokio* themselves. It is also for `std`.
- **embassy** [12]: Async library for `no_std` applications. It is therefore the ideal library for embedded Rust and IoT.
- **smol** [18]: A small, simplified async runtime.

2.2.2 Preemptive (threads)

For preemptive concurrency the main options are:

- **OS threads**: Rust provide a common interface to threads in the standard library. Its implementation leverages the OS thread implementation.
- **RTOS (Real Time Operating System)**: The use of a real time operating system like FreeRTOS or ESP-IDF in the case of embedded programming. In practice, it implies the use of the standard library because if a full operating system is available this means that the `std` library can be implemented too. This is ideal for embedded devices with enough resources to be able to run an OS. Smaller devices like some small microcontrollers cannot use it.
- **RTIC (Real-Time Interrupt-driven Concurrency)** [19]: It is a lightweight operating system that integrates minimum services for concurrency. Many other facilities like queues, channels, event loops, etc. are not included. It is based on interrupts to schedule the tasks. Input and output tasks are awaken by interrupts and the small runtime switch task execution.

  This is ideal for small devices because it is much lighter than a RTOS system and allows to leverage the task communication tools that are already included with Rust. Currently, it is available only for some platforms like STM microcontrollers and ARM processors. One notable small IoT device with ARM processor that is tested to work well with RTIC is Raspberry Pico (based on the RP2040 processor).

Rust and its standard library include support for asynchronous programming:

- Fundamental traits, types and functions, such as the `Future`.
- The async/await syntax that is supported by the Rust compiler.
- Types, macros and functions are provided by the `futures` crate.

Also, there is support with all kind of smart pointers apart from `Box<T>` for passing pointers (references) between tasks or threads.

2.3 Rust in IoT deployments

As a general purpose programming language, Rust can be used in the field of IoT in different parts of a typical deployment, so it can be considered as a common language infrastructure for the complete IoT stack. In this section, we analyze the use of the language at different levels, from the IoT node to server-side development.

2.3.1 Rust in servers

Rust has properties that make it possible to use it for tasks that are normally handled by higher level-languages. The deployment process is not as fast as with dynamic languages but the produced code is very efficient and much safer. Also, in the long term it can be easier to maintain, specially for large code bases.

One of these fields is web servers. Usually, web applications are developed with languages like PHP, Python, Java or JavaScript. But there exists many web frameworks in Rust to produce web applications like Rocket, Warp, Tide, Actix Web or Axum.

Also, in the database field, it is possible to access the more popular database servers such as Postgresql, MariaDB or MongoDB using various crates like Diesel, sqlx, or native drivers for MongoDB, SQLite, PostgreSQL and MySQL. There are also drivers for keystore databases like Redis or services like Azure or AWS.

There are also web assembly (WASM) containers that allow the execution in the cloud of generic applications compiled to WASM. This allows for automatic deployment of the applications and easy management.

2.3.2 Rust in edge nodes

Rust can be used very effectively in edge nodes of IoT. These nodes perform tasks like gateways between different networking technologies, doing intermediate computation that is too heavy for end nodes (edge computing) and summarizing information to send only relevant data to upstream servers.

Usually these devices are more powerful than end nodes but are cost constrained and not so big as cloud servers. It is important that the performance of its software to be high. Rust is ideal for this task because its ecosystem has libraries for all these network technologies and it has the performance. Security of this edge nodes is also very important as they are easy to attack.

The AI/ML field is the one that still lacks libraries as powerful as the existing ones for other languages though.

2.3.3 Rust in mobile devices

For mobile devices, Rust can be used in the browser to develop the front-end of web applications using WebAssembly. The amount of JavaScript can be reduced to a minimum so that most of the application is programmed in Rust. There are several frameworks to integrate WebAssembly code in the browser, like *Yew*, *Seed* or *Percy*.

2.3.4 Rust in embedded development, IoT nodes

In the end node is where Rust is more valuable, thanks to its properties for embedded programming.

The *Awesome embedded Rust* [20] repository contains a directory with a number of links to resources related with embedded programming in Rust. One very interesting list is the PACs and HALs crates.

There are three main layers of abstraction in Rust's drivers:

- PACs (Peripheral Access Crates): they provide a safe direct interface to the peripherals of the chip, allowing to configure every last bit however is needed. Typically, it is not necessary to use this low layer.
- HALs (Hardware Abstraction Layers): This layer is built on top of the chip's PAC. They abstract whole peripherals into single structs. This is the layer more commonly used.
- BSCs (Board Support Crates): this layer abstracts a whole board at once. It provides abstractions to use the microcontroller and the sensors in the board, LEDs, etc.

Currently, there are PACs and HALs for most of the popular hardware architectures, microcontrollers and sensors. The RISC-V platform was chosen for this project because it has direct support in LLVM (so also in Rust) and it is new and promising. Also, Espressif had implemented Rust's standard library for a family of its chips (ESP32-C3). But any other could have been used like ARM, STM, NXP, etc.

Most drivers implement a set of standard traits defined in projects like *embedded-hal*, so they are reusable across different architectures. Drivers for all sensors considered in this project already existed or were trivial to implement with Rust infrastructure for I^2C bus.

There are also crates for technologies like LoRa that are very important to IoT. Many radio transceivers have drivers for Rust. Projects line Embassy and RTIC allow to develop fully concurrent applications.

The community around embedded Rust is very active. Some projects are being developed and changing very quickly which sometimes makes it difficult to configure everything to work together. Fortunately, the nice tooling of Rust allows for very detailed dependencies specification.

Chapter 3

Hardware setup of an IoT node prototype

In this chapter, a prototype implementation of a simple IoT node is shown. The goal is to develop a system complete enough for the common difficulties found in this kind of projects to arise, and to evaluate the potential benefits/capabilities of Rust to overcome them. But at the same time to keep it simple enough so that implementation details do not hide the main ideas and issues to demonstrate.

The IoT node reads data from some sensors and send it periodically to a server through an MQTT connection. This produces a system in which it is necessary to run various tasks concurrently and to synchronize their access to the hardware devices (sensors and WiFi).

The system is comprised of a microcontroller board with WiFi connectivity, one or more external sensors and a small LCD screen. The sensors are all connected to an I^2C bus. This setup allows to show and demonstrate:

- The use of multiple tasks in Rust and their synchronization mechanisms.
- The solutions to access the hardware from Rust in a safe way.
- The sharing of the I^2C bus for concurrent access to read the sensor data.
- The setup and handling of the network stack and network hardware using different libraries.
- The management of the MQTT connection, as well as the publication and subscription mechanisms.
- The use of an OLED LCD screen for showing text or graphics.

3.1 Hardware setup

The prototype node is mounted in a breadboard (figure 3.1), with the following components:

- A microcontroller development board ESP32-C3-DevKit-RUST-1.
- A temperature and humidity sensor (HTU21D).
- A CO_2 and volatile organic compounds (TVOC) sensor SGP30.
- An OLED display panel SSD1306.

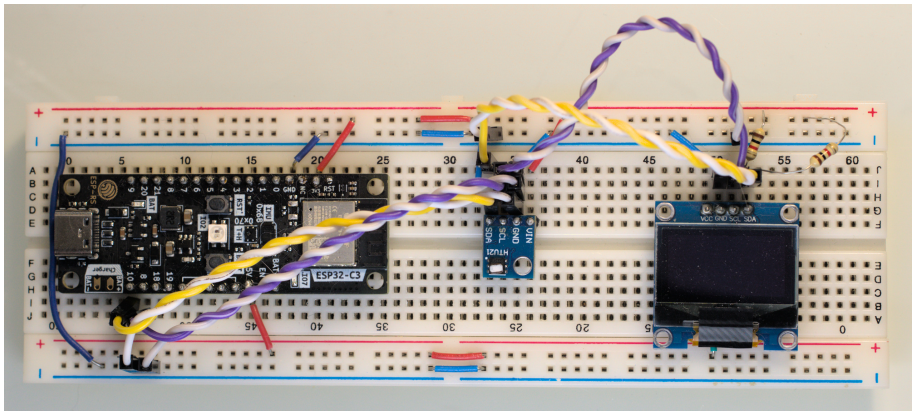

Figure 3.1: IoT node. Breadboard with the components

The external components are connected through an I^2C bus. The pins 8 and 10 are used for the I^2C bus clock (SCL) and data (SDA) to connect the HTU21D sensor and SSD1306 OLED screen. The same I^2C bus is shared between all sensors and the screen.

Regarding the powering of the components, the Espressif development board is fed with 5V through the USB connector. The board has a LDO (Low-dropout regulator) that produces 3.3V that the ESP32-C3 needs and it provides an output for other 3.3V devices.

The OLED screen requires also $3.3V$ but its board includes its own voltage converter so it is feed with $5V$ from the USB directly to not overload the converter in the MCU board.

For testing different sensors, they can be simply swapped in the circuit because all of them are I^2C. Tested external sensors are HTU21D, SGP30 and CCS811.

3.1.1 ESP32-C3-DevKit-RUST-1 development board

ESP32-C3-DevKit-RUST-1 is based on the ESP32-C3, a single-core WiFi and Bluetooth 5 (LE) microcontroller SoC, based on the open-source RISC-V architecture. The board includes the ESP32-C3-MINI-1 module, a 6 degrees of freedom IMU, a temperature and humidity sensor, a Li-Ion battery charger, and a Type-C USB.

3.1.2 Temperature and humidity sensor HTU21D

The HTU21D sensor (figure 3.2) is similar to the temperature and humidity sensor Si7021, the one commonly used in our faculty laboratories. It has the same logic, commands and I^2C signals so it can be replaced in the prototype without needing to modify the software.

3.1.3 Inertial measurement unit sensor ICM-42670-P

The ICM-42670-P is an IMU (Inertial Measurement Unit) included in the ESP32-C3-DevKit-RUST-1 board. It is a 6-axis MEMS Motion Tracking device that combines a 3-axis gyroscope and a 3-axis accelerometer.

In the context of this IoT prototype it is used as a convenient source of additional data to send to the server. It is connected to the same I^2C bus than external sensors.

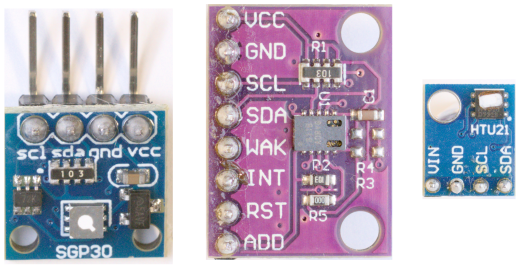

Figure 3.2: SGP30 for CO_2, CCS811 for CO_2 and TVOC and HTU21D temperature sensors

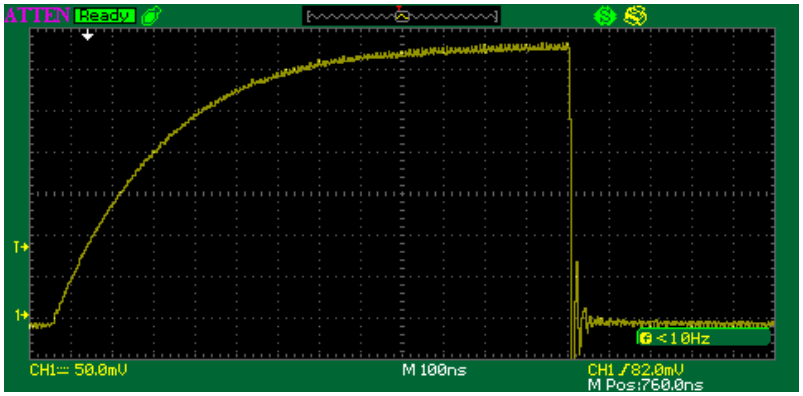

Figure 3.3: I^2C clock signal rise time with included 10 $k\Omega$ resistors

3.1.4 Temperature and humidity sensor SHTC3

The SHTC3 is a temperature and humidity sensor included in the ESP32-C3-DevKit-RUST-1 board. In a real IoT node it wouldn't be very useful as its readings are conditioned by the temperature of the board itself that is hot because of the MCU heat. But it is very convenient for testing and reading from I^2C bus.

3.1.5 I^2C bus hardware requirements

During the mounting of this breadboard prototype several problems where found with the transmission of data between the master (MCU) and slave devices (sensors) using the I^2C bus. The transmission was unreliable at times and even impossible with some sensors or configurations.

Further investigation of this issue revealed that the shape of the supposedly square wave of clock signal was the problem (figure 3.3). The "rise time" of the up flange has a maximum time that has to be met. Otherwise when the clock signal is read it is not still high enough to be considered in high state.

The rise time is influenced by the combination of resistance and capacitance of the bus line. Devices in connected to the I^2C bus use open drain gates to pull down the signal, and when they release it (open the gate) it takes time for the signal to recover de high state. It depends on the pull-up resistor employed. This resistors should match the total capacitance of the line so that the line recover high state fast enough (lower rise time).

The I^2C bus specification states the maximum capacitive impedance in the data and

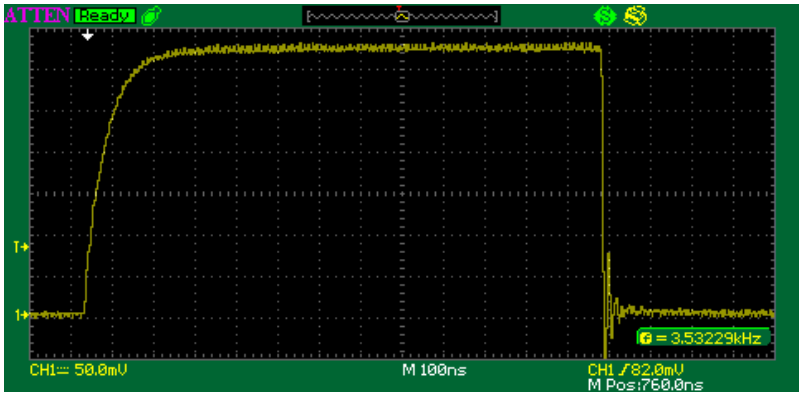

Figure 3.4: I^2C clock signal rise time with 1 $k\Omega$ resistors

clock lines of the bus. This impedance is the result of the addition of the impedances of all sensors connected to the bus plus the own impedance of the cables used.

It is usually recommended to set up 10 kΩ pull up resistors in clock and data lines for I^2C bus. The ESP32-C3-DevKit-RUST-1 includes them already in the own board, so in principle it is not necessary to add them. This board also includes two I^2C sensors, the ICM-42670-P IMU and the SHTC3 temperature sensor. The included pull-up resistors are 10 $k\Omega$, that are a good match for the total capacitance of the bus and the two sensors.

When additional external sensors are added the global capacitive load will increase and it might be necessary to use smaller value resistors for a faster recover of the voltage in the lines.

In the case of this project, for the particular case of ESP32-C3-DevKit-RUST-1 development board, the addition of any sensor leads to the need of bus pull-up resistors. The board is so sensible to this issue that even the addition of a short length of cable to any of the SCL or SDA pins results in an inoperative I^2C bus. Just plugging in the board into a breadboard (without connecting anything else) results in this problem. The added capacitance of the breadboard rails is too much apparently.

It has to be noted that this rise time depends not only on capacitance and resistance of the bus and sensors connected but also on the speed of the bus. At all time in this project it has been kept at 400kbps although lowering it to 100kbps, for example, reduces the problem significantly. However, in a general case the high speed could be necessary depending on the device, like fast drawing in a graphical I^2C lcd, for example. So in this project the 400kbps is kept.

For this project it was necessary to add two 1 $k\Omega$ resistors in both the clock (SCL) and data (SDA) lines. But even with this additions the rise time in the signal was too high. After extensive tests the solution was to create two balances twisted pairs witch each line. SDA line is twisted with ground cable and SCL is also twisted with ground. This way the rise time is kept at a valid amount (figure 3.4).

It should be emphasized that this was the only way to connect external sensors to this board and have an operative I^2C bus.

Chapter 4

An `std` application with *esp-idf-sys*

In this chapter an example IoT node is developed using the Espressif ESP-IDF operating system from Rust.

There are fundamentally two types of crates that can be created in Rust, depending on whether they make use of the standard library (`std` crates) or not (`no_std` crates). In this case we are creating an `std` crate leveraging the full capabilities of a real time operating system (RTOS), ESP-IDF. The Rust standard library is implemented using this system in the ESP32-C3 microcontroller.

The *esp-rs* [21] organization maintain a number of community projects enabling the use of the Rust programming language on various SoC and modules produced by Espressif Systems. In particular, for this example the following crates are used:

- embedded-svc: Abstraction traits for embedded services.
- *esp-idf-svc* [22]: An implementation of embedded-svc using ESP-IDF drivers.
- *esp-idf-hal* [23]: An implementation of the embedded-hal and other traits using the ESP-IDF framework.
- *esp-idf-sys* [24]: Rust bindings to the esp-idf development framework. Gives raw (unsafe) access to drivers, WiFi and more.

Most functions of the `std` Rust standard library are implemented by these crates, which means that this example is a full `std` application.

As esp-idf itself is implemented in C language it is necessary to compile its full code during the compilation of our example application. The cargo build system will take care of this dependency and will download and compile the full esp-idf and its bindings so that `std` is available. This step is very time consuming as esp-idf is large but it is only performed during the first use of esp-idf and its results are cached for subsequent compilations. In addition it is possible to configure cargo so that this cache is shared between different crates by modifying the Cargo configuration file (`config.toml`) as shown in listing 4.1.

Listing 4.1: Cargo configuration file (`.cargo/config.toml`)

```
[env]
ESP_IDF_TOOLS_INSTALL_DIR = { value = "global" }
```

4.1 Running the example

To run the examples of this project the Rust toolchain must be installed. Specifically a nightly toolchain release is required to guaranty an error free compilation. Listing 4.2 shows the commands to install the toolchain from date `2023-06-01`.

Listing 4.2: Console log (UART) during Initialization

```
rustup update
rustup default nightly
rustup component add rust-src --toolchain nightly-2023-06-01-x86_64-unknown
    -linux-gnu
cargo install espflash
cargo install cargo-espflash
```

The process to run the example begins downloading and installing the code from the GitHub repository [14], inside the directory `sensor`.

First change to the directory of *sensor* example. It is recommended to completely erase the firmware so that any previous data stored in NVS (Non Volatile Storage) is wiped (for example previously provisioned credentials or WiFi configuration). For this task the *esptool* program from Espressif must be used. It is required to be installed previously. The instructions show here to install the *esptool* application are specific for the Arch Linux distribution. They must be adapted for other distributions or downloaded directly from Espressif (listing 4.3).

Listing 4.3: Erase flash memory (NVS storage)

```
pacman -S esptool
esptool.py --chip esp32c3 --port /dev/ttyACM0 erase_flash
```

To compile the program and flash it to the board follow command from listing 4.5.

Listing 4.4: Compilation and flashing commands

```
cargo build
cargo run
cargo espflash monitor
```

Figure 4.1 shows the compilation and flashing of the program to the development board.

Listing A.1 shows the console log connected to UART of the development board during initialisation. The messages show the normal ESP-IDF initialization. These messages are logged from the C libraries that are called by the Rust wrappers that implement Rust `std` library (lines 1 to 70).

Next lines show the node initialization. The FSM shows that it starts in the *Initial* state and as no credentials are available in the NVS storage it starts the WiFi in access point mode. Also a HTTP server is started listening in port 80 to receive credentials from a client during configuration.

The node is now awaiting for connection to its WiFi access point from a laptop or mobile phone. For testing purposes a connection is done from a laptop that receives IP `192.168.71.2`. As the IP of the node is `192.168.71.1` a HTTP connection is done to its port 80. When a connection is received for URL "/" it is assumed to receive an HTML form with the fields SSID and password for WiFi (the form is not implemented and data is hard-coded in the program).

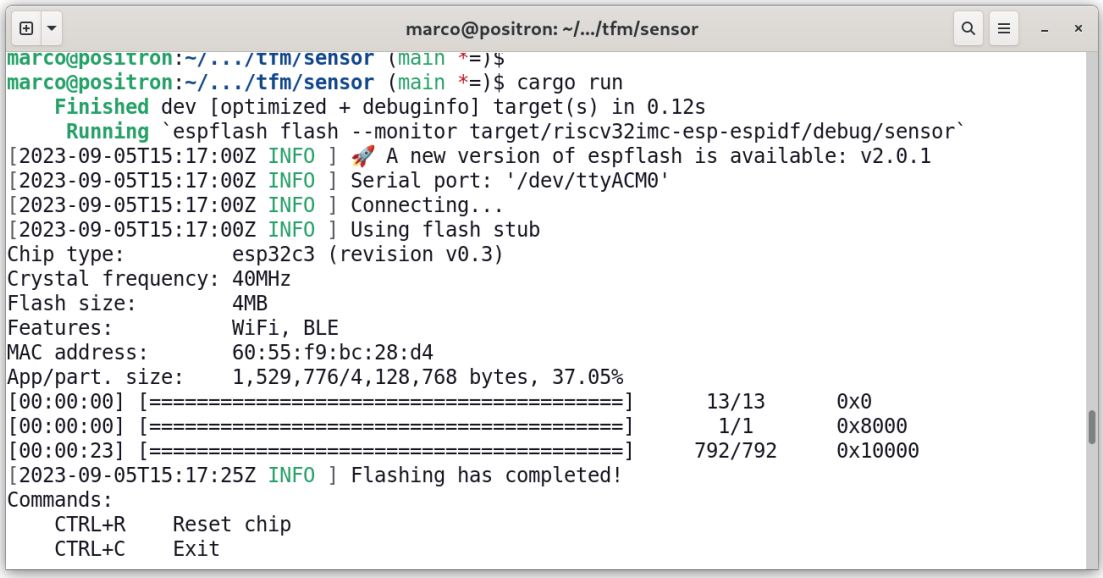

Figure 4.1: Compilation and flashing of example

Listing 4.5: HTTP connection for provisioning

```
curl http://192.168.71.1/
```

The listing A.2 shows the console log when the node receives the HTTP connection. It shows how the FSM transitions from various states:

- `Initial` (unprovisioned)
- `Provisioned` (line 5): credentials received and stored in NVS
- `WifiConnected` (line 59): connected to WiFi as a client
- `ServerConnected` (line 69): when connected to MQTT server

Now the node is reading temperature sensor data and sending over to the MQTT server. Figure 4.2 shows a MQTT client subscribed to the topic `/rust/temperature` receiving the temperature data from the sensor.

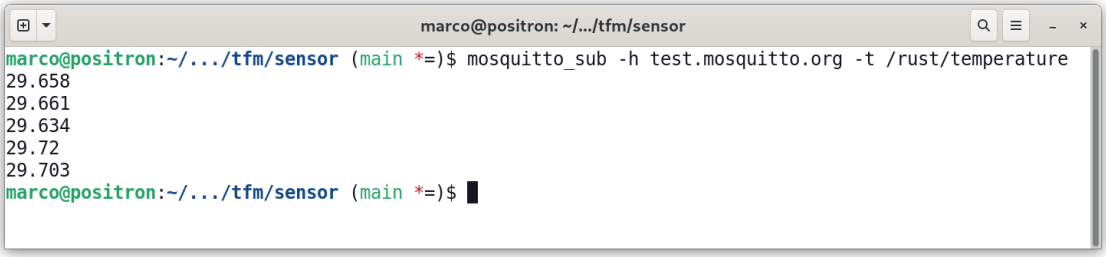

Figure 4.2: MQTT client subscribed to topic /rust/temperature

4.2 Tasks organization and architecture

There are several options to organize different payloads when `std` and a full RTOS systems is available (ESP-IDF):

- One simple loop: We can use the simple solution of one loop that execute every task in sequence and repeats indefinitely.
- RTOS system tasks.
- `std` threads.

While the first option is appealing because its simplicity, it presents several problems when the number of tasks increases or tasks are complex. For example, if the tasks are of asynchronous nature (like networks tasks or long processes that we must wait to complete) it quickly becomes very difficult.

With the second option we can use tasks from the RTOS system. ESP-IDF allows to create tasks and it has its mechanisms to synchronize data and pass signals between them with various tools like queues, synchronization locks, etc.

This is simple (at least from the developer point of view) but increases the resources needed at runtime (memory and processor time). For every task created a new memory space must be allocated for its stack. Also, processor time will be necessary to handle the preemptive multitask and context switch between tasks.

Communication between task must be performed carefully because this context switch can happen at any given time even when not appropriate. The advantage is that the tasks can run isolated between them if that is desired.

The third option, `std` threads is similar because threads are implemented in this case by the crate esp-idf-sys that uses the esp-idf threads underneath. Some fine tuning capabilities that esp-idf offer are lost in this way but the advantage is that the interface to configure the threads is the standard one of Rust, which allows to port the application to other platforms easily. The application won't be so tied to the Espressif infrastructure.

In most cases the optimum is to use a reduced number of threads and split the different tasks between them. This keeps the resources usage reasonable and allow to isolate some tasks from another, while some tasks are run in the same thread by means of a loop or finite state machine.

In this project, one task is spawned for a finite state machine (FSM) that handles most of the tasks. This FSM is implemented in the file `fsm.rs`[1].

4.3 Finite states machines in Rust

For this project, several ideas were tried for the implementation of the finite state machine, being it a very important part of the design and the kernel of the application. The goal was to develop something powerful, simple and elegant.

It was desirable a very scalable design capable of adapt to more complex projects. Rust is a language that provides many abstractions that have zero runtime cost and leveraging one of these abstractions was very appealing. At the same time, many of these constructions allow for a safe type checking at compile time, making it very difficult or even impossible to miss some common bugs.

[1] `https://github.com/marcormc/rustiot/blob/main/sensor/src/fsm.rs`

Listing 4.6: Rust struct representing the finite state machine

```
pub struct Fsm<'a> {
    pub state: State,
    pub tx: mpsc::Sender<Event>,
    pub sysloop: EspSystemEventLoop,
    pub wifi: Box<EspWifi<'a>>,
    pub nvs: EspDefaultNvs,
    pub httpserver: Option<EspHttpServer>,
    pub mqttc: Option<EspMqttClient>,
    mqtt_host: Option<String>,
    mqtt_user: Option<String>,
    mqtt_passwd: Option<String>,
}
```

The ideas shown in references [25], [26] demonstrate various approaches to program finite state machines in Rust. The main idea behind most of them is to use the Rust's structs in a way that the instance running at a given time represents one of the possible states. The transition from one state to another leads to the destruction (de-allocation) of this struct and the instantiation of a new one for the new state.

This approach has many advantages, for example it ensure all the resources (memory, network connections, files, etc) are freed with the destruction of one state and nothing is left behind. Also, it ensures that the system can only be in one state at a time and no part of the program holds a reference or pointer to the old state.

Also, the transition between states can be modeled using Rust `enum`. It makes it possible to ensure correct transitions from one state to the next at compile time, making it impossible to compile a program with unexpected transitions. It makes almost impossible to reach errors at runtime. To represent error states that are the result of logic or runtime conditions (network connections, etc) one solution is to define additional states that represent error conditions.

All this theoretical ideas were tried in this project, but none of then were finally adopted because the destruction of the FSM structs when transitioning from one state to another lead to the destruction and reallocation of many resources that are indeed shared between the states. For example, a state that represents a MQTT connection with a server can be de-allocated and created again for a new connection, but the reference to the WiFi controller should be kept alive for the duration of the program.

In Rust, when a struct holds a mutable reference to a struct it also becomes the one and only owner of this resource. It is necessary to be careful when creating trees of structs containing nested references because to release the innermost reference it will be necessary to release the parent struct that holds the reference to it. Otherwise this would become a dangling reference pointing to a resource that no longer exists.

The easier way to solve this problem without overly complicating the code was finally to create a struct that represents the finite state machine but lives for the duration of the program (see declaration in listing 4.6). The state is stored inside within an `enum` that can take as many values as states are possible (listing 4.7).

In Rust, each option of an `enum` can have additional state. In this case it is used to store data relative to this state. For example, in the state Provisioned the credentials obtained are stored inside the own enum.

Listing 4.7: Rust enum that holds states

```rust
#[derive(Debug, PartialEq)]
pub enum State {
    Initial,
    Provisioned {
        wifi_ssid: String,
        wifi_psk: String,
        mqtt_host: String,
        mqtt_user: Option<String>,
        mqtt_passwd: Option<String>,
    },
    WifiConnected,
    ServerConnected,
    Failure,
}
```

The function `next(&mut self, event: Event) -> Option<State>` in file `fsm.rs` is called to transition from one state to another when a relevant event happens (network command, timer, etc). In this design, the detection of an impossible or erroneous transition is done inside this function at runtime. With the other mentioned designs it is possible to do ensure correct transitions statically.

In this case, the unexpected transitions are ignored by simplicity and the state is kept unmodified. In a real project one should act on these errors as needed.

The change of state can be the result of a direct call from the program or by the reception of an event from a different task. Listing 4.8 shows the possible events in a Rust `enum`. Analog to the states enum, this events can carry information with them.

When the FSM enters a new state, the function `enter_state(&mut self)` is called and perform the necessary actions depending on the state.

The usual progress from booting to a completely connected systems passes by the following states unless an error occur:

- *Initial*: Initial state in which credentials to connect to WiFi and servers are unknown. It credentials can't be read from NVS (Non Volatile Storage) the WiFi is initialized in access point mode and a HTTP server setup to receive them from a client.
- *Provisioned*: Credentials have been obtained from NVS storage or received from HTTP server but the node hasn't connected to the WiFi network as a client yet.
- *WifiConnected*: The node has connected to the WiFi network and has Internet access.
- *ServerConnected*: The node has connected with the MQTT server. It can receive commands by subscription to MQTT topics and can send sensors data by publishing MQTT messages.

4.4 Hardware resources

The entry point to the program is the `main` function in `main.rs`. This function obtains references to the structs that abstract the access to the hardware resources.

Listing 4.8: Rust enum that holds events

```
#[derive(Debug, Clone)]
pub enum Event {
    Credentials {
        wifi_ssid: String,
        wifi_psk: String,
        mqtt_host: String,
        mqtt_user: Option<String>,
        mqtt_passwd: Option<String>,
    },
    WifiConnected,
    WifiDisconnected,
    MqttConnected,
    MqttDisconnected,
    SensorData(f32),
    RemoteCommand {
        command: String,
    },
}
```

Listing 4.9: Hardware resource initialization

```
  let part = EspDefaultNvsPartition::take()?;
  let nvs = EspDefaultNvs::new(part, "storage", true).unwrap();
  let peripherals = Peripherals::take().unwrap();
  let sysloop = EspSystemEventLoop::take()?;
  let wifi = Box::new(EspWifi::new(peripherals.modem, sysloop.clone(), None
    )?);
```

Listing 4.9 shows the acquisition of references to the NVS storage, peripherals from the HAL crates for ESP32-C3, the sysloop from ESP-IDF and WiFi wrapper to the ESP-IDF implementation in esp-idf-svc crate.

These references can be passed to different parts of the program where access to the hardware is necessary, but the obtained references are mutable, which implies ownership. As there can be only one owner for each resource we cannnot pass these references to multiple places in the program.

4.5 Passing data between threads using messages

A `std` thread is created to run the finite state machine (see listing 4.10). This thread will have a stack size of 8000 bytes even when the default recommended size by ESP-IDF is smaller. This is because there is some overhead by some metadata that Rust keeps in each stack for safety.

For the sharing of information between threads, messages are used instead of shared memory. This is a popular approach for safe concurrency. The main tool that Rust provides for passing messages is the `channel`.

Before initiating the thread an MPSC *channel* (`std::mpsc::channel`) is created. This channel is a multi-producer, single-consumer FIFO queue communication primitive. Senders are clone-able (multi-producer) such that many threads can send simultaneously to one receiver (single-consumer). This way Rust provides a standard synchronization mechanism that allow to send a message from one or more threads to another thread in a safe and asynchronous way. This is an unidirectional flow of information (figure 4.3).

In the program we create a sender (`tx`) and a receiver (`rx`). This `tx` half can only be owned by one thread, but it can be cloned to send the reference to other threads. In this case both the `tx` and `rx` are moved to the new thread (ownership is trasferred). There is no need to clone the `tx` end because only one reference is needed here.

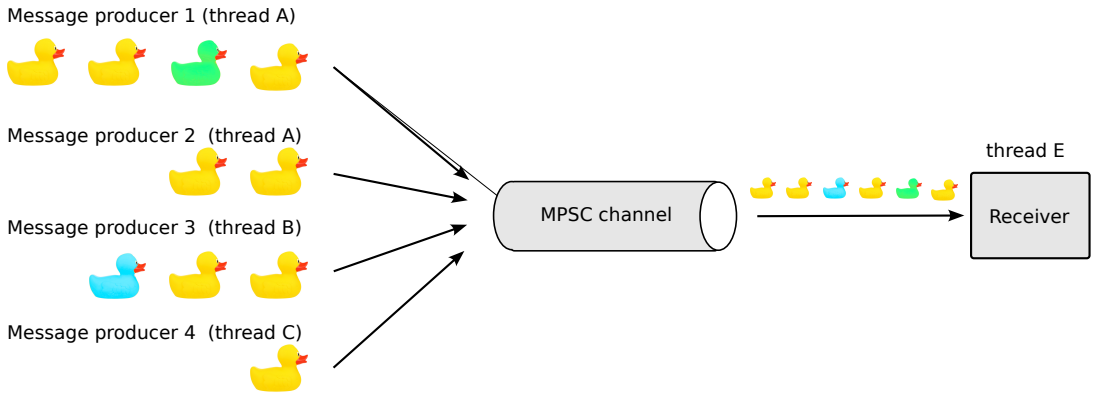

Figure 4.3: MPSC channel

Inside the new thread, the finite state machine is instantiated (FSM). The FSM receives the `tx` and keep it inside to be able to distribute copies (clones) of it to the different parts of the program later. This is the way these other parts will communicate with the FSM, by sending messages through the mpsc channel.

Inside the thread an infinite loop begins. This loop locks execution waiting to receive any message from the channel. This waiting is done using the receiver (`rx.recv()`) that has been moved to the new thread. The lock doesn't waste CPU time. The processor is used for other threads o kept in idle state. This way we avoid pooling with a loop with periodic checks or continuous execution that would waste energy. This is very important for an IoT node.

When a message is received in the channel, the FSM process it calling `process_event` function. Messages passed in the mpsc channel can be any Rust struct. The ownership of the struct is transferred to the receiver side of the channel. The passed struct is an `Event` shown in listing 4.8.

4.6 Scheduling sensor reading with timers

The periodic reading of the temperature sensor is triggered by a timer. The timer is set up in the function `start_sensor` in the `shtc3.rs` crate. The node uses the `EspTimerService` from esp-idf-svc which is a timer implemented by ESP-IDF.

This timer is attached to the main thread. This is the reason we keep the main thread running at the end of the main function with a loop. This loop contains a sleep function to not waste resources unnecessarily.

Listing 4.10: Thread and MPSC channel creation

```
let (tx, rx) = mpsc::channel();
let timer = start_sensor(peripherals.pins, peripherals.i2c0, tx.clone()
)?;

thread::Builder::new()
    .name("threadfsm".to_string())
    .stack_size(8000)
    .spawn(move || {
        info!("Thread␣for␣FSM␣event␣processing␣started.");
        let mut fsm = Fsm::new(tx, sysloop, wifi, nvs);
        loop {
            let event = rx.recv().unwrap();
            info!("Event␣received:␣{:?}", event);
            fsm.process_event(event);
        }
    })?;

// This is to keep this thread alive because it has timers running
loop {
    sleep(Duration::from_secs(10));
    info!("Inactive");
}
```

The timer function gets a clone of the channel mpsc sender (`tx`) to be able to notify the FSM when there is a new temperature value read from the sensor.

Chapter 5

A `no_std` application with Embassy and *esp-hal*

In this chapter we will examine an application example of an IoT node created as a `no_std` Rust crate (without using Rust standard library). The Embassy library for asynchronous embedded environments is used.

5.1 Running the example

The Rust toolchain must be installed as explained in previous chapter. The code for this example can be found in GitHub repository [14], inside the directory `embsens`.

The credentials and WiFi SSID must be provided as environment variables because this example does not integrate provisioning. Before compiling the example issue the commands in listing 5.1.

Listing 5.1: HTTP connection for provisioning

```
export SSID=mywifi
export PASSWORD=my_wifi_password
```

The compilation of the program and flashing procedure are common with those illustrated in the previous chapter.

Listing B.1 shows the console log when the node boots and connects to WiFi and MQTT server. Every Embassy task log messages whit its own prefix. Lines 1 to 22 show booting. Note that ESP-IDF is not executing.

The task `run_i2c` is started for reading the accelerometer *icm42670* included in the board and task `run_htu` for reading external temperature sensor *HTU21D* (similar to *SI7021*) even when there is no network connection yet. Both tasks share access to the i2c bus. The shared access is handled by the Embassy shared bus crate. This sensor tasks send messages to the FSM using `CHANNEL` which is a `embassy_sync::channel::Channel` that allow sending messages between Embassy tasks.

The task `connection` starts attempting to connect to the WiFi access point. Upon connection establishment, the `mqtt_task` task attempts connection with the MQTT server. The task also subscribes to the topic `/embsens/command` to be able to receive remote commands. Once network is connected, all network events are handled by task `net_task`.

When the connection to the MQTT is ready, the FSM start publishing MQTT messages

with the received data from the channel.

The task `mqtt_receiver` poll the TCP socket of the MQTT connection periodically to receive commands published in the `/embsens/command` topic. Received commands are only logged to console. In case some action is to be performed the implementation would consist on sending a message over the `CHANNEL` to the FSM. The in the FSM do the appropriate action.

Figure 5.1 shows the MQTT client receiving data from the node and figure 5.2 shows the MQTT client publishing data to topic `/enbsens/command` for the node to receive.

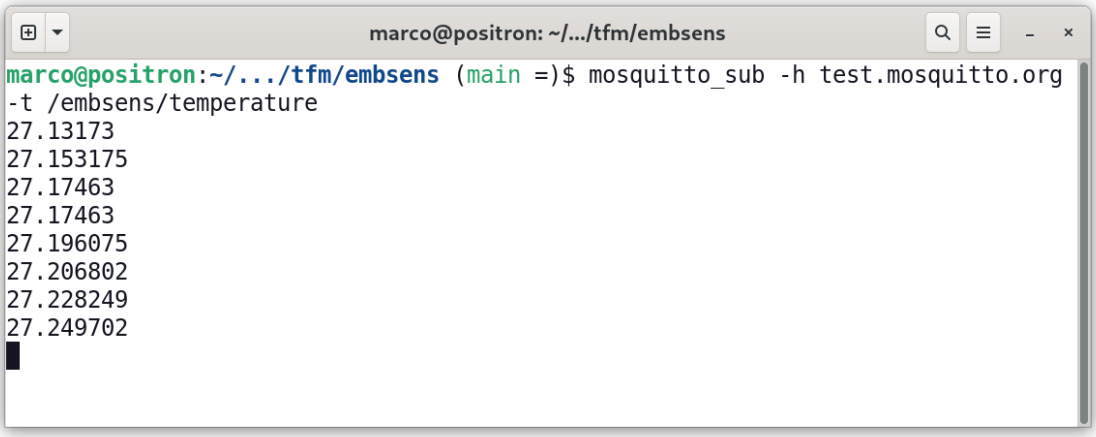

Figure 5.1: MQTT client receiving temperature data from the node

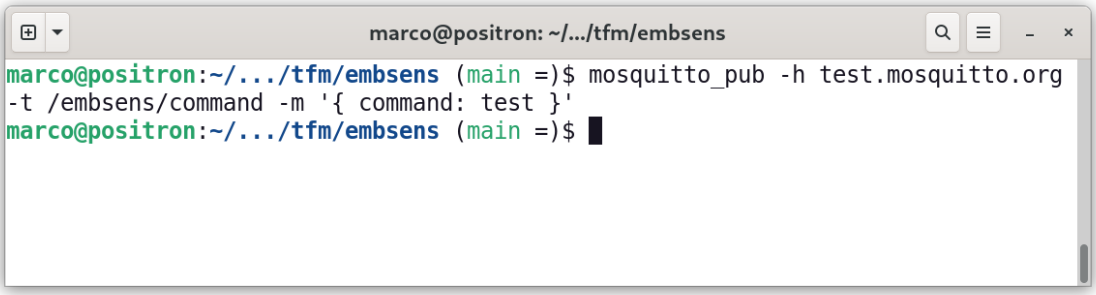

Figure 5.2: MQTT client publishing command for the node

Listing B.1 shows how the command is received in the node with its subscription (line 89).

5.2 `no_std` crate

When a `no_std` application is created, the Rust standard library (`std`) is not available. The reason is that a complete implementation of this library relays in the existence of an actual operating systems to provide some services like threads, IO, etc. To prevent rust from loading the standard library the `no_std` declaration is used. This makes it possible to create software for bare metal environments in which there isn't operating system.

Yet, there are some parts of the standard library that are still available through *libcore*, the platform-agnostic parts of the standard library. But *libcore* excludes parts that might not be desired in an embedded environment leaving only the bare minimum. This imposes some limitations.

One of the main limitation is that there is no heap allocator, which means that the heap memory can't be used and all resources must be statically allocated in the stack. Unless of course that the memory management is done manually. There are some crates that provide allocator for this scenario in case heap is an absolute necessity, but in principle it is recommended to try to limit everything to the stack if possible because of the security and performance reasons in this kind of environment. This limitation is one of the biggest challenges in embedded Rust programming.

Another major limitation is the handling of strings, that in `std` Rust require the use of the heap. For example, usual tasks like for formatting strings and converting numbers to string have to be carried on manually or by using additional libraries like `esp-println` or `core::fmt::Write`.

5.3 Hardware access

For the hardware access, the `esp32c3-hal` [27] is used, which is part of the esp-rs project and in particular implements the hardware abstraction layer, drivers and configurations for the ESP32-C3 chip. Currently there are HAL implementations for many chips of the Espressif family of processors and in particular for RISC-V (`esp32c2`, `esp32c3`, `esp32c6`, `esp32s2`, `esp32s3`, `esp8266`).

Peripherals are all devices included in the microcontroller that are not the RAM, CPU and flash memory. The HAL crates provide an interface to interact with them. The `esp32c3-hal` allows to access the internal hardware devices found in the ESP32-c3 microcontroller. These are high level and safe to use abstractions from Rust.

There are other external devices connected to the microcontroller by different buses like I2C, SPI or UART for example. For them, different crates exists which provide drivers to access from Rust in an analogous way. For example, in the case of this project the `icm42670` crate is used to access the accelerometer data from the IMU (Inertial measurement unit) ICM42670.

All these drivers can define their own API interface to allow communication with the device they serve. But doing so would be a waste of resources in the sense that a different driver would be needed for the same device depending on which platform it is used (same device but different microcontroller architecture, for example). Also, for devices of the same type (two different thermometers, for example) the physical device must be substituted for other model or other brand the client code would need to be rewritten.

The solution is to create a common standard interface that all drivers implementing the same kind of device will follow. The Rust language doesn't define such interfaces but it defines the necessary tools to implement such infrastructure: the traits. In embedded development there are various initiatives to standardise the access to different kind of devices.

One prominent project is `embedded_hal` [28]. With `embedded_hal` it is possible to develop platform agnostic drivers. Many of these drivers can be found in *crates.io* [11] repository by searching for the embedded-hal keyword. The main embedded-hal crate contains only blocking traits but there are related project with similar purpose:

- `embedded-hal-async`: for async/await based traits.

- `embedded-hal-nb`: polling-based, using the `nb` crate.
- `embedded_io`: Replaces Rust's `std::io` because in `no_std` targets they are not available. This crate contains equivalent traits for `no_std`.
- `embedded_svc`: A set of traits for services higher level than embedded-hal and typically found in embedded microcontrollers with WiFi or BLE support. Provide abstraction traits for embedded services like WiFi, Network, Httpd, Logging, etc.

All these projects define traits (interfaces) that provide a separation between client code and drivers rather than implement code themselves. Driver programmers implement the traits so that clients use a common API for all drivers.

The HAL and PAC crates usually include `unsafe` Rust code. Communication between hardware and software is usually performed using memmory mappings. Writing and reading to these memory areas is the way data is transferred from software to hardware and vice-versa. But these read and write operations must be performed in way that the Rust compiler can't guaranty to be safe just from static analysis. This is the reason that normally these code blocks must be declared `unsafe`. Inside this blocks, it is the programmer who must ensure the safety guaranties are maintained.

The good practice is to keep these code blocks as small and simple as possible and to perform only the minimal operations required for memory accesses inside them. This way the rest and majority of the code can benefit from the Rust's compiler verification. HAL and PAC crates wrap all `unsafe` code providing a safe interface to the client application.

But a fundamental problem still remains. The HAL crates provide structs that represent the different hardware devices. For example a variable holds a reference to the I^2C bus to read and write data to the bus or a reference to a thermometer allows reading temperatures. But it is very common that the hardware has to be accessed from various different parts of the application. How can we access the same hardware, at possibly the same time, from different parts of the program, different threads, functions or modules?

5.4 Hardware and global state

In other systems and low level programming languages it is allowed and very common to have pointers or references to hardware resources passed along all the program freely. Indeed, it is typical to have global variables to access the hardware from wherever is necessary.

It is up to the developer to be careful to avoid problems like accessing memory at the same time from several threads. Or avoid accessing to a resource that has already been freed, for example. This is a very common source of bugs in C, for example.

Rust avoids these problems by design. You can have many non mutable references but it is not allowed to have more than one mutable reference for the same struct. There will be only one owner of the resource. This allows to ensure safe concurrent access from different threads and manage the lifetime of resources in a safe way.

If hardware and its state is going to be represented by a struct in Rust, this leads to two possible solutions:

- Creating a global struct for the driver to be able to access it from everywhere. Hardware is in reality global state (the state is stored in a physical device which is unique), so it is only logical to represent it in software with global variables.

- Creating the struct (instantiating the drivers) at runtime in the main function or other function and passing the mutable reference down to the one and only place where it will be used.

The first option is not possible in Rust without using `unsafe` code because there is no way Rust compiler can ensure all static guaranties will be met (no concurrent access, no dangling references when lifetime ends, etc). The only safe global variables allowed are `const`, and only static non mutable references can be obtained.

The second solution has its own problems. For example, it would be possible to instantiate the driver two or more times for the same hardware resource and they have simultaneous accesses resulting in bugs, using different references.

The solution is to use a pattern commonly used in other languages too: the *singleton*. A factory function in the driver is the only exposed way of crating objects (in this case structs) and this function restrict the creation to just one instance.

In Rust, this function returns a mutable reference to the instance that allows access to the hardware. The program is then responsible to pass this reference to the place where it is used. It can be "moved" to another thread (ownership is transferred), passed to other function o stored inside other struct.

It is a common convention in embedded Rust HAL crates to expose a function that allows to obtain unique references for each piece of hardware. In the case of this esp32c3 crate the `Peripherals::take()` call delivers the root reference that contains all the hardware found in the microcontroller. From this reference all other hardware parts are obtained as shows listing 5.2.

Listing 5.2: `peripherals` reference to access hardware devices

```
let peripherals = Peripherals::take();
let system = peripherals.SYSTEM.split();
let mut peripheral_clock_control = system.peripheral_clock_control;
[...]
let mut rtc = Rtc::new(peripherals.RTC_CNTL);
[...]
let timer = hal::systimer::SystemTimer::new(peripherals.SYSTIMER).alarm0;
[...]
let (wifi, _) = peripherals.RADIO.split();
[...]
let io = IO::new(peripherals.GPIO, peripherals.IO_MUX);
let i2c = I2C::new(
    peripherals.I2C0,
    io.pins.gpio10,
    io.pins.gpio8,
    400u32.kHz(),
    &mut peripheral_clock_control,
    &clocks,
    );
[...]
```

If the resource needs to be used from more than one place, from the same or from other thread, it will be necessary to wrap the reference with some kind of smart pointer (Box, Ref, RefCell, etc). This will take care of the management of the lifetime of the resource and the concurrent access if the right pointer is used.

In `no_std` applications there is an additional issue. It is not possible to instantiate the resources (drivers) at runtime because there is no heap memory unless a special library for this purpose is used (an allocator). All resources must be statically allocated in the heap.

If the driver is instantiated in the main function it won't be possible to pass the reference to another thread because Rust will expect *static lifetime* for the reference. This means that Rust needs to know for sure that the memory resource will last for the entire duration of the program so it is safe to call it from the other threads. But to have static lifetime we need a global variable which leads to... `unsafe` code again.

In `no_std` application this is solved using a dedicated smart pointer: `StaticCell`. StaticCell allows the instantiation of a resource in such a way that a reference with static lifetime is obtained. The resource is statically allocated but initialised at runtime. The term cell derives from the fact that it is a wrapper around the original reference as other smart pointers are. `StaticCell` provides a no-std-compatible, no-alloc way to reserve memory at compile time for a value, but initialise it at runtime, and get a `'static` reference to it.

In the developed example node (listing 5.3) some `StaticCell` references are used to globally instantiate the Embassy executor and the I2C bus access Mutex.

Listing 5.3: `StaticCell` declarations in embsens application

```
static EXECUTOR: StaticCell<Executor> = StaticCell::new();
static I2C_BUS: StaticCell<NoopMutex<RefCell<I2C<I2C0>>>> = StaticCell::new
    ();
```

The `EXECUTOR` and `I2C_BUS` references point to structs that will remain alive for the duration of the program lifetime. They also implement the *Send* trait so are safe to pass between threads.

A macro is used to ease the repetitive declaration and initialization and reduce it to one step, see listing 5.4.

Listing 5.4: `StaticCell` macro

```
macro_rules! singleton {
    ($val:expr) => {{
        type T = impl Sized;
        static STATIC_CELL: StaticCell<T> = StaticCell::new();
        let (x,) = STATIC_CELL.init(($val,));
        x
    }};
}
```

This macro is also used to create two buffers for the TCP socket, one for sending and one for receiving, with a static lifetime.

5.5 Concurrency with Embassy

Embassy tasks are spawned in an executor. Only one thread is used but in principle it would be possible to deploy two threads to split tasks and leverage the 2 cores that the esp32c3 microcontroller has. Listing 5.5 illustrates the creation of the tasks.

Each task receives the resources it needs, that are mostly wrapped in RefCell and Mutex smart pointers to allow concurrent access.

The tasks are:

- `fsm` for handling the events that other tasks send when receiving MQTT or sensor data and execute the required action.
- `connection` to stablish WiFi connection.
- `net_task` to handle network events (mostly low level).
- `mqtt_task` to stablish connection with MQTT server and subscribe to topic to receive remote commands.
- `run_i2c` to read accelerometer sensor.
- `run_htu` to read temperature sensor.

The library to manage the hardware WiFi controller and WiFi connection is from *embassy-net* that is part of the *Embassy* [12] project. It is a fully asynchronous TCP/IP implementation that internally uses a C library from Espressif to access the hardware.

Listing 5.5: Spawning of Embassy tasks

```
#[entry]
fn main() -> ! {
    [...]

    let executor = EXECUTOR.init(Executor::new());
    executor.run(|spawner| {
        // General coordination task
        spawner.spawn(fsm(mqtt)).ok();

        // Wifi and network handling tasks
        spawner.spawn(connection(controller)).ok();
        spawner.spawn(net_task(&stack)).ok();

        // Tasks to send and receive MQTT messages
        spawner.spawn(mqtt_task(&stack, mqtt)).ok();
        spawner.spawn(mqtt_receiver(mqtt)).ok();

        // Sensor reading tasks
        spawner.spawn(run_i2c(i2c_dev1)).ok();
        spawner.spawn(run_htu(i2c_dev2)).ok();
    })
}
```

5.6 Networking and asynchronous wifi handling

As it is a `no_std` crate, there is no operating system, and hence all network infrastructure must be supplied by the application itself or by libraries. Input and output services provided by the `std` library are not available.

In this case the *esp-wifi* [29] library is used. This library includes the drivers for the use of WiFi, BLE and ESP-NOW in Espressif microcontrollers. It is compatible with `no_std` crates as it doen's rely on the standard library. It consist of Rust wrappers aroung a previous C library by Espressif.

The network events are handled in the embassy task `net_task` in the example. The call `stack.run().await` takes care of all these events.

5.7 MQTT client

The MQTT protocol is used for sending data from the sensors and receiving commands from the server. This node implements a client that allow publishing messages and subscription to any topic. The implementation is divided in two different layers, one for networking and the other for the MQTT application protocol.

The lower level layer handles the TCP connection with the MQTT server. This layer is in the file `tiny_mqtt.rs` (link to repo) and takes care of handling the TCP socket. The socket is assigned a transmission and reception buffers. The buffer implementation is based on ideas shown in repository *bjoernQ/esp32-rust-nostd-temperature-logger* [30].

The sending is triggered every time the function `publish_with_pid` is called. The receiving buffer is polled periodically from a dedicated Embassy task `mqtt_receiver` (listing 5.6).

Listing 5.6: Polling of MQTT socket withou blocking

```
/// Embassy task to receive data from MQTT server
#[embassy_executor::task]
async fn mqtt_receiver(mqtt: &'static Mutex<NoopRawMutex, RefCell<TinyMqtt
    <'static>>>) {
    loop {
        {
            let shared = mqtt.lock().await;
            let state = shared.borrow_mut().socket.state();
            if state == State::Established {
                println!("Waiting␣for␣MQTT␣packets␣from␣server...");
                if shared.borrow_mut().poll().await.is_err() {
                    println!("Error␣receiving␣data␣from␣mqtt␣server");
                }
            } else {
                println!("Socket␣not␣connected␣yet...");
            }
        }
        Timer::after(Duration::from_millis(1000)).await;
    }
}
```

Before any attempt to read the socket a call to `socket.can_recv()` ensures that there are data waiting to be read. Otherwise the call to `socket.read()` would block the task until something is received. This must be avoided if there are other duties in the same task that would be interrupted. (listing 5.7).

Listing 5.7: Polling of MQTT socket without blocking

```
    async fn receive_internal(&mut self) -> Result<(), TinyMqttError> {
        loop {
            let mut buffer = [0u8; 1024];
            if self.socket.can_recv() {
```

```
                println!("can_recv␣true");
                // socket.read() won't block, there are data waiting to be
read.
            } else {
                println!("can_recv␣false");
                // nothing received in the socket.
                return Ok(());
            }
            let len = self.socket.read(&mut buffer).await.unwrap();
            [...]
```

In this case, the task can be locked because its only purpose is to read the socket. It should be possible to directly call `socket.read()` allowing Embassy to lock on the call and keep the task slept until something is received. Then the received data would be sent through the `CHANNEL` for processing in the required task. This would be more marginally efficient than polling and would avoid the potential problem of overflowing the incoming buffer if the throughput of received commands is high. After many tests it wasn't possible to do so without getting not only the receiving task suspended but also the main mqtt sending task, probably by an interlock in the socket implementation.

The higher level in the MQTT implementation is the application protocol itself. In this case we are using a pure Rust implementation: `mqttrust` [31]. This crate is a `no_std`, `no_alloc` implementation of secure MQTT Client capabilities. This means that everything is internally handled in the stack and can be used in applications that lacks a memory allocator like in this case.

5.8 I2C bus sharing

As it was explained, the usual mechanism in Rust to access hardware resources is to "take" a mutable reference from a factory implemented in the HAL or PAC. This reference is the only one with read and write access and can be passed and stored in the part of the program that need the hardware resource. This reference implies ownership and can't be cloned. The only way to overcome this is to create some kind of smart pointer or "Cell" that allows safe cloning and sharing between threads.

There are times in which it is necessary to access the same resource from many places simultaneously. A typical case is a hardware bus like the I^2C bus, SPI (Serial Peripheral Interface) bus, etc. In these cases one or more devices are connected to the same bus and must be addressed concurrently. This makes necessary to send and receive data through the bus from different tasks concurrently.

Being so common functionality, there are various crates and libraries to handle the sharing of I^2C bus in Rust. Depending on the concurrency model or type of crate being created some options are:

- For `std` applications that use esp-idf-sys implementation, the `esp_idf_hal::i2c` takes care of concurrency using ESP-IDF.

- For `no_std` applications using threads or accessing the bus from many places in the same thread there is the `shared-bus` crate. This crate allows the creation of copies of the reference to the bus. When any read or write operation is performed on the bus, this crate implements the necessary synchronization mechanisms. For example,

if the bus is busy receiving or sending data to a sensor and a call to send data to another sensor it will be delayed so the don't overlap.

- For Embassy applications, the concurrency model is asynchronous. The Embassy project includes the module `embassy_embedded_hal::shared_bus`. It is important that any blocking in a task to wait until the bus is available is coordinated with the Embassy executor. This way the other tasks of the application can continue operating or the CPU can idle without energy waste. This is what is used in this example project.

Embassy provides modules to access both the I^2C and SPI buses and in both cases in a synchronous and asynchronous way.

The module `embassy_embedded_hal::shared_bus::asynch` is used when asynchronous access is preferred. Read and write operations in this crate return a *promise* in which the Embassy task can *await* until the operation is completed and results are available. In the meanwhile, the task can continue with other operations.

Module `embassy_embedded_hal::shared_bus::blocking` is for synchronous access. Calls to read and write will block the calling task until results are ready. In this example, the synchronous task is used because an independent task is created for reading sensor data, so briefly blocking it is not a problem.

The code in listing 5.8 shows the initialization of I^2C bus. The global `I2C_BUS` variable is a `StaticCell` to allocate the resource with `'static` lifetime and be able to initialize it at runtime.

The I^2C driver is initialized with `GPIO8` and `GPIO10` as SCL and SDA lines and a frequency of $400kHz$, taking the hardware resource for I^2C from the *peripherals* reference obtained from HAL. This is stored in the `i2c` variable.

This reference is wrapped in a `RefCell` and a `NoopMutex` to be able to clone it and pass the clones between threads. The wrapped reference is used to initialize the `I2C_BUS` variable that has `'static` lifetime.

So basically, now the `i2c_bus` is cloneable and shareable between threads. The driver for each of the devices connected to the bus will take a clone of this bus reference. All the devices will access the bus using these clones simultaneously but the implementation in `embassy_embedded_hal::shared_bus` will take care of sequencing the access to the hardware real bus.

Listing 5.8: Initialisation of I^2C bus with Embassy

```
[...]
static I2C_BUS: StaticCell<NoopMutex<RefCell<I2C<I2C0>>>> = StaticCell::
  new();
[...]

fn main() -> ! {
  [...]
  let i2c = I2C::new(
    peripherals.I2C0,
    io.pins.gpio10,
    io.pins.gpio8,
    400u32.kHz(),
    &mut peripheral_clock_control,
    &clocks,
```

```
    );
    // Create i2c_bus with static lifetime
    let i2c_bus = NoopMutex::new(RefCell::new(i2c));
    let i2c_bus = I2C_BUS.init(i2c_bus);

    // share the i2c bus between devices in embassy (sync)
    let i2c_dev1 = I2cDevice::new(i2c_bus);
    let i2c_dev2 = I2cDevice::new(i2c_bus);
    hal::interrupt::enable(Interrupt::I2C_EXT0, Priority::Priority1).unwrap
    ();

    let executor = EXECUTOR.init(Executor::new());
    executor.run(|spawner| {
        // Sensor reading tasks
        spawner.spawn(run_i2c(i2c_dev1)).ok();
        spawner.spawn(run_htu(i2c_dev2)).ok();
        [...]
    })
}
```

As the listing 5.8 shows, one reference to the bus can be obtained from the `i2c_bus` variable for each device connected to the bus. In this case, the `i2c_dev1` and `i2c_dev2` variables can be used by the drivers of each device in a transparent way, as if they had exclusive access to the bus.

In this case, two Embassy tasks are spawned to read sensors using I^2C:

- `run_i2c` to read the ICM42670 accelerometer sensor integrated in the own development board.
- `run_htu` to read the external HTU21D temperature and humidity sensor (with hardware and software interface similar to SI7021).

Chapter 6

Conclusions and Future Work

6.1 Rust as a language for IoT

In general, during the development of this project it has been found that Rust is an ideal programming language for IoT in many regards. In the following, we review some of the outcomes and also some of the difficulties found in different areas.

6.1.1 Efficiency

Rust is focused on concurrency, speed, and memory safety, all of them properties vary valuable in all layers of a typical IoT environment.

Rust allows the creation of considerably efficient code in order to fully exploit the IoT device's raw capabilities without degrading its speed. It has the potential for very high performance, specially for embedded development. Due to the availability of powerful abstractions and libraries specially designed for embedded systems (`no_std` applications) it is possible to leverage concurrency without the need of a RTOS real time operating system.

The runtimes of these libraries are tiny and very efficient, replacing more complex systems. Many IoT nodes can benefit from this smaller infrastructure as they do not require a full blown OS.

By design, most Rust constructs are zero-cost abstractions, which means that although from the programming point of view they appear as high level abstractions, the code produced by the compiler factors out the complexity and is lightweight. The Rust compiler can produce code as small o smaller as a comparable C program, with the huge advantage of being checked for safety at compile time.

There are not runtime costs like the case of garbage collectors, for example, but the memory management is guaranteed to be safe without runtime overheads. There are some cases in which a small runtime is needed, as it is the case of executors in asynchronous libraries, but they are tiny. The use of smart pointers like `Box<T>` and its siblings can introduce reference counting that impact runtime performance, but its use can be reduced with good design and is negligible compared to other options.

6.1.2 Security and reliability

Security is a critical aspect of IoT nodes, because these nodes are connected to Internet and can interact with the environment, reading sensible data or controlling sensible infras-

tructure. The attack surface is usually huge so it is critical to keep possible exploits to a minimum. Rust has the properties to reduce security flaws to a minimum. Logical errors still remain, but the number of problems that are simply not possible in Rust is huge, and are also the more common.

Regarding reliability this project showed that although it can be difficult to get a program that compiles without errors, once it compiles and it is flashed to the device it usually works as expected. This is not always the case with other languages as C/C++.

6.1.3 Adoption and popularity

The use of Rust in general has become very popular recently and it is between the most desired languages to learn in recent years. In the field of IoT its adoption is growing quickly thanks of a very active community that is developing many drivers and libraries for all kind of embedded devices. Also, some microcontroller makers are developing their own frameworks to support Rust, as is the case of Espressif.

Although the Rust learning curve is steep, its adoption continues growing. One of the problems found during the development of this project was the unusual way in which some problems are solved in Rust. Additional thought is required in the design phase. For the inexperienced Rust programmer the process is trial and error with multiple iterations required until a design is found that fits well with the way things are done in Rust.

The process of obtaining a program that simply compiles without errors is slow and painful initially. Most of the time the compiler is indicating that the design has flaws that make it impossible to guarantee memory safety and correctness. That does not mean the program has bugs. Instead it means that the compiler cannot assure that everything is safe. This can be exasperating at the beginning, but with time compiler errors are easier to understand.

The other factor that slows down adoption is that currently it can be more difficult to hire Rust experts if needed. C and C++ are still very strong in embedded development.

Not all hardware has Rust drivers yet and hence for some deployments it can be difficult to choose Rust.

6.2 Maturity of the ecosystem

The availability of drivers and libraries for the hardware used in this project was almost complete. Only one sensor required direct read and write to the I^2C bus.

There are drivers, HAL and PAC crates for most of the popular IoT architectures, microcontroller and devices. Many drivers follow the *embedded-hal* abstractions which make it possible its use for any given architecture. The *crates.io* [11] repository makes very easy to quickly find what is available for a given functionality or hardware. The *awesome-embedded-rust* [20] repository contains references to many useful resources for IoT and embedded programming in general with Rust.

The coverage of architectures, microcontroller platforms and makers is huge and improving continuously.

6.3 Toolchain support. Flashing and debugging workflows

Rust integrates a very modern and well designed tool set. It includes Cargo, which handles many of the required tasks automatically.

Generally speaking, in other languages compiling a program requires downloading the dependency libraries, binaries and header files or their source code, and compile everything. Complex applications can require a big number of dependencies and the exact versions are required for successful compilation. Each library has to be fetched from different sources and some might not be easily available. Dependencies can have their own recursive dependencies. Preparing everything for compilation can be cumbersome.

Rust handles all this complexity automatically. *Cargo*, the build and dependency tool automatically installs the architecture toolchain, Rust compiler version, `std` library for the correct release. Then it downloads all required dependencies from *crates.io* or Git repositories if required and compiles everything. The process can be easily reproduced across environments, and works without error.

Then there is the process of flashing it to the embedded devices, which usually is different and requires different software tools depending on the hardware maker. *Cargo* can also handle the flashing to the hardware device. The result is that a simple command like `cargo run` can automatically do all these steps. This has been very useful in this project, specially for quickly running existing examples.

The conclusion in this project is that it is a very powerful and easy to use system. And in the context of IoT, with heterogeneous hardware it is valuable to have a systematic way of compiling, flashing and debugging.

6.4 IoT resources

There are a big number of resources in Rust repositories, and the number of tools that apply to IoT is huge. These are some of the tools, repositories and libraries that have been found very useful for the development of this project.

- Support for ESP ships with esp-idf-sys, esp-idf-hal crates and its esp-idf-hal examples.
- Hardware drivers for board and sensors: crates esp32c3 and Icm42670.
- ivmarkov's Rust on ESP32 STD demo application.
- Embassy project for asynchronous.
- embedded-hal and embedded-hal-async crates.
- shared-bus crate.
- Some documentation books:
  - The Rust Programming Language book
  - The Rust on ESP book
  - The Embedded Rust book
  - Asynchronous Programming in Rust
  - Rust by example
  - The Embedonomicon
  - rust-embedded/discovery: Discover the world of microcontrollers through Rust
- Embedded Rust curated list of resources.

- awesome-esp-rust repository (curated list of resources for ESP32 development in Rust).
- Embedded devices Working Group.
- esp-rs: Libraries, crates and examples for using Rust on Espressif SoC's.
- Embedded Rust Trainings for Espressif.
- rtic book and RTIC repository.
- probe-rs (embedded debug tool).

6.5 Embedded programming and concurrency

Embedded programming is one area in which Rust shines. It is easy to use without all the problems usually associated with it. Also, the language itself does not force any particular way of concurrent programming. The *futures* crate defines traits for other libraries to implement. The standard library `std` define threads and other utilities. There exist many libraries for different forms of concurrency.

In particular, asynchronous programming, a form of cooperative concurrency is very convenient to use. In Rust, there are quite a few libraries with different implementations of asynchronous tasks like *tokio* [16], *embassy* [12], *async-std* [17], etc.

Asynchronous programming also needs all associated libraries for networking and input/output operations. Libraries that read or write data must return the control even if the operation hasn't been completed and don't block the task during the waiting. In Rust, there are many of these libraries that allow to work with various executors.

Other advantage of Rust is that there are many crates dedicated simply to define *traits* that declare generic interfaces for other libraries to implement. There are traits for asynchronous operations, data bus sharing, managing network connections, generic hardware access, generic drivers for each type of device, etc. These traits standardize the way these operations are performed making very easy to port the application that uses them to other hardware or platform or replacing sensors or actuators in a project without the need to rewrite everything or with minimal changes.

Regarding concurrency, the most remarkable conclusion from this project is that Rust provides lightweight solutions as an alternative to the use of threads or operating systems. In embedded development the operating system is typically a RTOS (Real Time Operating System). In other languages like C, the programmer uses directly the services provided by this RTOS like queues, threads, etc. In Rust there are more options:

- Use the standard library `std` that uses underneath the services of the RTOS operating system. Other services can be accessed directly. This is the heavier options and corresponds with the first example in this project ESP-IDF. It also requires an implementation of the `std` library that may exists or not depending on the hardware platform. This produces a `std` crate.
- Use of an asynchronous runtime like `embassy` that is much lighter. Task does not require threads in general and an RTOS system is not necessary. It saves memory and can be faster depending on workload. This produces a `no_std` crate.
- Use of a light form of RTOS that leverage interrupts to provide asynchronous concurrency like RTIC. This also produces a `no_std` crate.

The use of `no_std` is much more appropriate for small IoT devices. Also it does not rely on the existence of a full implementation of the standard library, a requisite that is hard to meet and not available for most of the cases.

6.6 Code organization

For a complex enough project the use of a FSM (Finite State Machine) is still a very good option. Rust provides abstractions that make its implementation very convenient. There are options from more formal, in which all errors can be detected at compile time to more practical, in which some of them are relegated to runtime for a simpler approach. The Rust `Enum` is very powerful in any case because its `Option`'s can contain heterogeneous data. Data types can be different depending on the option's value. The behavior resembles the `Union` structs in C but it is more powerful and definitely safer.

A thorough discussion on alternatives for FSM implementations in Rust can be followed in [25] and [26].

A common problem found during the programming of both examples in this project was finding the best way to store the resources in every part of the program. In Rust, the `struct` replaces the objects of C++ because it can contain variables and implement related code.

Most of the time, the temptations exists to store the references to resources (drivers, hardware, database or network connections, references to other modules, etc) inside the structs. This creates a tree like structure very common in languages like $C++$. In Rust, this can be a problem because this mutable references implicitly give the ownership of the referenced resource to the struct that will hold it. It will be difficult to transfer this ownership even temporarily to another part of the program. For example, when calling a function it won't be possible to pass this resource directly because the reference still exists in the calling struct and there can be only one owner. Only non-mutable references can be created but most of the time this won't be enough.

So it is very important to think carefully where to store these resources. They can be passed (*"moved"* in Rust terms) from function to function but once they are held in a struct that's no longer the case. This produces iterative design cycles until a practical solution is found.

Bibliography

[1] *Micropython*, https://micropython.org/, Accessed: 01-09-2023.

[2] *Espruino. javascript on board*, https://www.espruino.com/, Accessed: 01-09-2023.

[3] *Rust programming language*, https://www.rust-lang.org/, Accessed: 01-09-2023.

[4] *We need a safer systems programming language*, https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/, Accessed: 01-09-2023.

[5] P. Kehrer, *Memory unsafety in apple's operating systems*, https://langui.sh/2019/07/23/apple-memory-safety/, Accessed: 01-09-2023.

[6] *Queue the hardening enhancements*, https://security.googleblog.com/2019/05/queue-hardening-enhancements.html, Accessed: 01-09-2023.

[7] *Linux kernel modules in rust*, https://static.sched.com/hosted_files/lssna19/d6/kernel-modules-in-rust-lssna2019.pdf, Accessed: 01-09-2023.

[8] *0-day attacks identified by google*, https://docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY/view, Accessed: 01-09-2023.

[9] *Wannacry ransomware attack*, https://en.wikipedia.org/wiki/WannaCry_ransomware_attack, Accessed: 01-09-2023.

[10] *Heartbleed*, https://en.wikipedia.org/wiki/Heartbleed, Accessed: 01-09-2023.

[11] C. team, *The rust community's crate registry*, https://crates.io/, Aug. 2023.

[12] *Embassy framework for embedded applications*, https://embassy.dev/, Accessed: 01-09-2023.

[13] *Serde framework for serializing and deserializing rust data structures*, https://serde.rs/, Accessed: 01-09-2023.

[14] M. Romera, *Rust in iot code repository*, https://github.com/marcormc/rustiot, Accessed: 01-09-2023.

[15] *Asynchronous Programming in Rust*, Accessed: 01-09-2023.

[16] *Tokio. asynchronous runtime for rust*, https://tokio.rs/, Accessed: 01-09-2023.

[17] *Async-std. asynchronous library for rust*, https://async.rs/, Accessed: 01-09-2023.

[18] *A small and fast async runtime*, https://github.com/smol-rs/smol, Accessed: 01-09-2023.

[19] *Real-time interrupt-driven concurrency*, https://github.com/rtic-rs, Accessed: 01-09-2023.

[20] R. community, *Awesome embedded rust repository*, https://github.com/rust-embedded/awesome-embedded-rust, Sep. 2023.

[21] esp-rs project, *Esp-rs project. libraries, crates and examples for using rust on espressif soc's*, https://github.com/esp-rs, Aug. 2023.

[22] esp-idf-sys community, *Esp-idf-sys: Raw rust bindings for the esp idf sdk*, https://github.com/esp-rs/esp-idf-sys, Aug. 2023.

[23] esp-rs, *Embedded-hal implementation for rust on esp32 and esp-idf*, https://github.com/esp-rs/esp-idf-hal, Aug. 2023.

[24] esp-idf-sys community, *Esp-idf-sys: Raw rust bindings for the esp idf sdk*, https://github.com/esp-rs/esp-idf-sys, Aug. 2023.

[25] A. Hoverbear, *Pretty state machine patterns in rust*, https://hoverbear.org/blog/rust-state-machine-pattern/, 2016.

[26] R. Laddad, *Modeling finite state machines with rust*, https://www.ramnivas.com/blog/2022/05/09/fsm-model-rust, 2022.

[27] esp-rs, *Hal crates for the esp32, esp32-c2/c3/c6, esp32-h2, and esp32-s2/s3*, https://github.com/esp-rs/esp-hal, Aug. 2023.

[28] H. team, *A hardware abstraction layer (hal) for embedded systems*, https://crates.io/crates/embedded-hal, Aug. 2023.

[29] *Espwifi: Wifi, ble and esp-now driver for espressif microcontrollers*, https://github.com/esp-rs/esp-wifi/, Accessed: 01-09-2023.

[30] *Demo of rust on esp32 with mqtt and adafruit.io for temperature logging*, https://github.com/bjoernQ/esp32-rust-nostd-temperature-logger, Accessed: 01-09-2023.

[31] Factbird, *Mqtt protocol implementation*, https://crates.io/crates/mqttrust, Aug. 2023.

Appendix A

Console logs for sensor (`std`) example

This appendix shows the outputs of *sensor* (`std`) example.

Listing A.1 shows the console log connected to UART of the development board during initialization.

Listing A.1: Console log of initialisation (unprovisioned)

```
I (43) boot: ESP-IDF v5.0-beta1-764-gdbcf640261 2nd stage bootloader
I (43) boot: compile time 11:30:26
I (43) boot: chip revision: V003
I (46) boot.esp32c3: SPI Speed      : 80MHz
I (51) boot.esp32c3: SPI Mode       : DIO
I (56) boot.esp32c3: SPI Flash Size : 4MB
I (61) boot: Enabling RNG early entropy source...
I (66) boot: Partition Table:
I (70) boot: ## Label            Usage          Type ST Offset   Length
I (77) boot:  0 nvs              WiFi data        01 02 00009000 00006000
I (84) boot:  1 phy_init         RF data          01 01 0000f000 00001000
I (92) boot:  2 factory          factory app      00 00 00010000 003f0000
I (99) boot: End of partition table
I (103) boot_comm: chip revision: 3, min. application chip revision: 0
I (111) esp_image: segment 0: paddr=00010020 vaddr=3c110020 size=5b3f0h
    (373744) map
I (179) esp_image: segment 1: paddr=0006b418 vaddr=3fc90800 size=023f0h (
    9200) load
I (181) esp_image: segment 2: paddr=0006d810 vaddr=40380000 size=02808h (
    10248) load
I (186) esp_image: segment 3: paddr=00070020 vaddr=42000020 size=107890h
    (1079440) map
I (365) esp_image: segment 4: paddr=001778b8 vaddr=40382808 size=0dec8h (
    57032) load
I (381) boot: Loaded app from partition at offset 0x10000
I (382) boot: Disabling RNG early entropy source...
I (393) cpu_start: Pro cpu up.
```

```
I (401) cpu_start: Pro cpu start user code
I (401) cpu_start: cpu freq: 160000000
I (402) cpu_start: Application information:
I (404) cpu_start: Project name:     libespidf
I (409) cpu_start: App version:      1
I (414) cpu_start: Compile time:     Sep  5 2023 16:54:39
I (420) cpu_start: ELF file SHA256:  0000000000000000...
I (426) cpu_start: ESP-IDF:          4b40411-dirty
I (431) cpu_start: Min chip rev:     v0.3
I (436) cpu_start: Max chip rev:     v0.99
I (441) cpu_start: Chip rev:         v0.3
I (446) heap_init: Initializing. RAM available for dynamic allocation:
I (453) heap_init: At 3FC97180 len 00045590 (277 KiB): DRAM
I (459) heap_init: At 3FCDC710 len 00002950 (10 KiB): STACK/DRAM
I (466) heap_init: At 50000020 len 00001FE0 (7 KiB): RTCRAM
I (472) spi_flash: detected chip: generic
I (477) spi_flash: flash io: dio
I (481) sleep: Configure to isolate all GPIO pins in sleep state
I (487) sleep: Enable automatic switching of GPIO sleep configuration
I (494) cpu_start: Starting scheduler.
I (514) sensor: Inicializando wifi
I (514) pp: pp rom version: 9387209
I (514) net80211: net80211 rom version: 9387209
I (524) wifi:wifi driver task: 3fc9f6a8, prio:23, stack:6656, core=0
I (524) system_api: Base MAC address is not set
I (524) system_api: read default base MAC address from EFUSE
I (534) wifi:wifi firmware version: 43eca89
I (534) wifi:wifi certification version: v7.0
I (534) wifi:config NVS flash: disabled
I (544) wifi:config nano formating: disabled
I (544) wifi:Init data frame dynamic rx buffer num: 32
I (554) wifi:Init management frame dynamic rx buffer num: 32
I (554) wifi:Init management short buffer num: 32
I (564) wifi:Init dynamic tx buffer num: 32
I (564) wifi:Init static tx FG buffer num: 2
I (564) wifi:Init static rx buffer size: 1600
I (574) wifi:Init static rx buffer num: 10
I (574) wifi:Init dynamic rx buffer num: 32
I (584) wifi_init: rx ba win: 6
I (584) wifi_init: tcpip mbox: 32
I (584) wifi_init: udp mbox: 6
I (594) wifi_init: tcp mbox: 6
I (594) wifi_init: tcp tx win: 5744
I (604) wifi_init: tcp rx win: 5744
I (604) wifi_init: tcp mss: 1440
I (604) wifi_init: WiFi IRAM OP enabled
I (614) wifi_init: WiFi RX IRAM OP enabled
I (614) esp_idf_svc::wifi: Driver initialized
I (624) esp_idf_svc::wifi: Stop requested
```

```
I (624) esp_idf_svc::wifi: Stopping
I (634) sensor: Inicialización del wifi terminada
I (634) sensor::shtc3: Starting sensor shtc3
I (644) sensor::shtc3: Starting measurements every 5 seconds
I (644) sensor: Thread for FSM event processing started.
I (654) sensor::fsm: ******** Entering state Initial
W (664) sensor::fsm: Key wifi_ssid not found in NVS
W (664) sensor::fsm: Key wifi_psk not found in NVS
W (674) sensor::fsm: Key mqtt_host not found in NVS
W (674) sensor::fsm: Key mqtt_user not found in NVS
W (684) sensor::fsm: Key mqtt_passwd not found in NVS
I (684) sensor::fsm: Credentials not found in NVS.
I (694) sensor::fsm: Activating wifi AP.
I (694) esp_idf_svc::wifi: Setting configuration: AccessPoint(
    AccessPointConfiguration { ssid: "aptest", ssid_hidden: false, channel:
     1, secondary_channel: None, protocols: EnumSet(P802D11B | P802D11BG |
    P802D11BGN), auth_method: None, password: "", max_connections: 255 })
I (724) esp_idf_svc::wifi: Disconnect requested
I (724) esp_idf_svc::wifi: Stop requested
I (734) esp_idf_svc::wifi: Stopping
I (734) esp_idf_svc::wifi: Wifi mode AP set
I (744) esp_idf_svc::wifi: Setting AP configuration:
    AccessPointConfiguration { ssid: "aptest", ssid_hidden: false, channel:
     1, secondary_channel: None, protocols: EnumSet(P802D11B | P802D11BG |
    P802D11BGN), auth_method: None, password: "", max_connections: 255 }
I (764) esp_idf_svc::wifi: AP configuration done
I (774) esp_idf_svc::wifi: Configuration set
I (774) esp_idf_svc::wifi: Start requested
I (784) phy_init: phy_version 950,11a46e9,Oct 21 2022,08:56:12
W (784) phy_init: failed to load RF calibration data (0x1102), falling back
     to full calibration
I (834) wifi:mode : softAP (60:55:f9:bc:28:d5)
I (834) wifi:Total power save buffer number: 16
I (834) wifi:Init max length of beacon: 752/752
I (834) wifi:Init max length of beacon: 752/752
I (844) esp_idf_svc::wifi: Starting
I (844) sensor::wifi: Starting wifi...
I (854) esp_idf_svc::wifi: About to wait for duration 20s
I (854) esp_idf_svc::wifi: Waiting done - success
I (864) sensor::fsm: Activating HTTP server
I (864) esp_idf_svc::http::server: Started Httpd server with config
    Configuration { http_port: 80, https_port: 443, max_sessions: 16,
    session_timeout: 1200s, stack_size: 6144, max_open_sockets: 4,
    max_uri_handlers: 32, max_resp_handlers: 8, lru_purge_enable: true,
    uri_match_wildcard: false }
I (894) esp_idf_svc::http::server: Registered Httpd server handler Get for
    URI "/"
[...]
```

Listing A.2: Console log during provisioning

```
[...]
I (143624) sensor::http: http server: recibido request /
I (143624) sensor: Event received: Credentials { wifi_ssid: "harpoland",
    wifi_psk: "alcachofatoxica", mqtt_host: "test.mosquitto.org", mqtt_user
    : None, mqtt_passwd: None }
I (143634) sensor::fsm: Recibido evento de provisionamiento
I (143644) sensor::fsm: ******** Entering state Provisioned { wifi_ssid: "
    harpoland", wifi_psk: "alcachofatoxica", mqtt_host: "test.mosquitto.org
    ", mqtt_user: None, mqtt_passwd: None }
I (143654) sensor::fsm: Trying to connect to wifi station.
I (143664) sensor::fsm: Using credentials harpoland, alcachofatoxica.
I (143674) sensor::fsm: Deactivating HTTP server
I (143674) esp_idf_svc::http::server: Unregistered Httpd server handler 1
    for URI "/"
I (143784) esp_idf_svc::http::server: Httpd server stopped
I (143794) esp_idf_svc::wifi: Setting configuration: Client(
    ClientConfiguration { ssid: "harpoland", bssid: None, auth_method:
    WPA2Personal, password: "alcachofatoxica", channel: None })
I (143804) esp_idf_svc::wifi: Disconnect requested
I (143804) esp_idf_svc::wifi: Stop requested
I (143814) wifi:station: 98:2c:bc:b5:e4:d3 leave, AID = 1, bss_flags is
    134243, bss:0x3fcad0f4
0x3fcad0f4 - _bss_end
     at ??:??
I (143814) wifi:new:<1,0>, old:<1,1>, ap:<1,1>, sta:<255,255>, prof:1
I (143824) wifi:<ba-del>idx
I (144094) wifi:flush txq
I (144094) wifi:stop sw txq
I (144094) wifi:lmac stop hw txq
I (144094) esp_idf_svc::wifi: Stopping
I (144094) esp_idf_svc::wifi: Wifi mode STA set
I (144094) esp_idf_svc::wifi: Setting STA configuration:
    ClientConfiguration { ssid: "harpoland", bssid: None, auth_method:
    WPA2Personal, password: "alcachofatoxica", channel: None }
I (144114) esp_idf_svc::wifi: STA configuration done
I (144114) esp_idf_svc::wifi: Configuration set
I (144124) esp_idf_svc::wifi: Start requested
I (144134) wifi:mode : sta (60:55:f9:bc:28:d4)
I (144134) wifi:enable tsf
I (144134) esp_idf_svc::wifi: Starting
I (144144) sensor::wifi: Starting wifi...
I (144144) esp_idf_svc::wifi: About to wait for duration 20s
I (144154) esp_idf_svc::wifi: Waiting done - success
I (144154) sensor::wifi: Connecting wifi...
I (144164) esp_idf_svc::wifi: Connect requested
I (144164) esp_idf_svc::wifi: Connecting
I (144174) esp_idf_svc::netif::status: About to wait for duration 20s
I (144174) wifi:new:<1,0>, old:<1,0>, ap:<255,255>, sta:<1,0>, prof:1
```

```
39 I (144814) wifi:state: init -> auth (b0)
40 I (144844) wifi:state: auth -> assoc (0)
41 I (144854) wifi:state: assoc -> run (10)
42 I (144934) wifi:connected with harpoland, aid = 5, channel 1, BW20, bssid =
       78:d2:94:a8:21:6b
43 I (144944) wifi:security: WPA2-PSK, phy: bgn, rssi: -71
44 I (144944) wifi:pm start, type: 1
45
46 I (144944) wifi:set rx beacon pti, rx_bcn_pti: 0, bcn_timeout: 0, mt_pti:
       25000, mt_time: 10000
47 I (144954) wifi:AP's beacon interval = 102400 us, DTIM period = 2
48 I (145044) wifi:<ba-add>idx:0 (ifx:0, 78:d2:94:a8:21:6b), tid:6, ssn:2,
       winSize:64
49 I (145664) sensor::shtc3: Temperature reading: 31.856 °C
50 I (145964) esp_netif_handlers: sta ip: 192.168.6.108, mask: 255.255.255.0,
       gw: 192.168.6.1
51 I (145964) esp_idf_svc::netif::status: Got IP event: DhcpIpAssigned(
       DhcpIpAssignment { netif_handle: 0x3fca5d00, ip_settings: IpInfo { ip:
       192.168.6.108, subnet: Subnet { gateway: 192.168.6.1, mask: Mask(24) },
        dns: None, secondary_dns: None }, ip_changed: true })
52 0x3fca5d00 - _bss_end
53     at ??:??
54 I (145984) esp_idf_svc::netif::status: Waiting done - success
55 I (145994) sensor::wifi: Wifi DHCP info: IpInfo { ip: 192.168.6.108, subnet
       : Subnet { gateway: 192.168.6.1, mask: Mask(24) }, dns: Some
       (192.168.6.1), secondary_dns: Some(0.0.0.0) }
56 Wifi sta activado true
57 I (146004) sensor: Event received: SensorData(31.856)
58 I (146014) sensor: Event received: WifiConnected
59 I (146014) sensor::fsm: ******** Entering state WifiConnected
60 I (146024) sensor::fsm: State WifiConnected.
61 I (146034) sensor::mqtt: Sending mqtt welcome message.
62 I (146034) sensor::mqtt: Subscribing to mqtt topic /rust/command
63 W (146044) sensor::mqtt: mqtt debug: received from mqtt client: Ok(
       BeforeConnect)
64 I (146074) wifi:<ba-add>idx:1 (ifx:0, 78:d2:94:a8:21:6b), tid:0, ssn:0,
       winSize:64
65 W (146364) sensor::mqtt: mqtt debug: received from mqtt client: Ok(
       Connected(false))
66 I (146364) sensor::fsm: Connected to MQTT server.
67 I (146364) sensor: Event received: MqttConnected
68 I (146374) sensor::fsm: ******** Entering state ServerConnected
69 I (146374) sensor::fsm: State ServerConnected. Start sending periodic data.
70 W (146404) sensor::mqtt: mqtt debug: received from mqtt client: Ok(
       Subscribed(51578))
71 W (146504) sensor::mqtt: mqtt debug: received from mqtt client: Ok(
       Published(24567))
72 I (150664) sensor::shtc3: Temperature reading: 31.898 °C
73 I (150664) sensor: Event received: SensorData(31.898)
```

```
I (150664) sensor::fsm: Sending temperature sensor data to MQTT: 31.898 °C
I (150674) sensor::mqtt: Sending mqtt data.
W (150714) sensor::mqtt: mqtt debug: received from mqtt client: Ok(
    Published(20908))
I (150904) sensor: Inactive
[...]
```

Listing A.3: Console log after provisioning

```
[...]
I (615) esp_idf_svc::wifi: Driver initialized
I (625) esp_idf_svc::wifi: Stop requested
I (625) esp_idf_svc::wifi: Stopping
I (635) sensor: Inicialización del wifi terminada
I (635) sensor::shtc3: Starting sensor shtc3
I (645) sensor::shtc3: Starting measurements every 5 seconds
I (645) sensor: Thread for FSM event processing started.
I (655) sensor::fsm: ******** Entering state Initial
W (665) sensor::fsm: Key mqtt_user not found in NVS
W (665) sensor::fsm: Key mqtt_passwd not found in NVS
I (675) sensor::fsm: Credentials from NVS: ssid = harpoland, psk =
    alcachofatoxica, mqtt = test.mosquitto.org,None,None
I (685) sensor: Event received: Credentials { wifi_ssid: "harpoland",
    wifi_psk: "alcachofatoxica", mqtt_host: "test.mosquitto.org", mqtt_user
    : None, mqtt_passwd: None }
I (695) sensor::fsm: Recibido evento de provisionamiento
I (705) sensor::fsm: ******** Entering state Provisioned { wifi_ssid: "
    harpoland", wifi_psk: "alcachofatoxica", mqtt_host: "test.mosquitto.org
    ", mqtt_user: None, mqtt_passwd: None }
I (725) sensor::fsm: Trying to connect to wifi station.
I (725) sensor::fsm: Using credentials harpoland, alcachofatoxica.
I (745) esp_idf_svc::wifi: Setting configuration: Client(
    ClientConfiguration { ssid: "harpoland", bssid: None, auth_method:
    WPA2Personal, password: "alcachofatoxica", channel: None })
I (755) esp_idf_svc::wifi: Disconnect requested
I (755) esp_idf_svc::wifi: Stop requested
I (765) esp_idf_svc::wifi: Stopping
I (765) esp_idf_svc::wifi: Wifi mode STA set
I (775) esp_idf_svc::wifi: Setting STA configuration: ClientConfiguration {
     ssid: "harpoland", bssid: None, auth_method: WPA2Personal, password: "
    alcachofatoxica", channel: None }
I (785) esp_idf_svc::wifi: STA configuration done
I (795) esp_idf_svc::wifi: Configuration set
I (795) esp_idf_svc::wifi: Start requested
```

```
I (805) phy_init: phy_version 950,11a46e9,Oct 21 2022,08:56:12
I (835) wifi:mode : sta (60:55:f9:bc:28:d4)
I (835) wifi:enable tsf
I (845) esp_idf_svc::wifi: Starting
I (845) sensor::wifi: Starting wifi...
I (845) esp_idf_svc::wifi: About to wait for duration 20s
I (845) esp_idf_svc::wifi: Waiting done - success
I (855) sensor::wifi: Connecting wifi...
I (855) esp_idf_svc::wifi: Connect requested
I (865) esp_idf_svc::wifi: Connecting
I (865) esp_idf_svc::netif::status: About to wait for duration 20s
I (905) wifi:new:<1,0>, old:<1,0>, ap:<255,255>, sta:<1,0>, prof:1
I (1535) wifi:state: init -> auth (b0)
I (1565) wifi:state: auth -> assoc (0)
I (1575) wifi:state: assoc -> run (10)
I (1625) wifi:connected with harpoland, aid = 5, channel 1, BW20, bssid =
    78:d2:94:a8:21:6b
I (1625) wifi:security: WPA2-PSK, phy: bgn, rssi: -76
I (1625) wifi:pm start, type: 1

I (1625) wifi:set rx beacon pti, rx_bcn_pti: 0, bcn_timeout: 0, mt_pti:
    25000, mt_time: 10000
I (1715) wifi:<ba-add>idx:0 (ifx:0, 78:d2:94:a8:21:6b), tid:6, ssn:2,
    winSize:64
I (1725) wifi:AP's beacon interval = 102400 us, DTIM period = 2
I (2635) esp_netif_handlers: sta ip: 192.168.6.108, mask: 255.255.255.0, gw
    : 192.168.6.1
I (2635) esp_idf_svc::netif::status: Got IP event: DhcpIpAssigned(
    DhcpIpAssignment { netif_handle: 0x3fca5da4, ip_settings: IpInfo { ip:
    192.168.6.108, subnet: Subnet { gateway: 192.168.6.1, mask: Mask(24) },
     dns: None, secondary_dns: None }, ip_changed: true })
0x3fca5da4 - _bss_end
     at ??:??
I (2655) esp_idf_svc::netif::status: Waiting done - success
I (2655) sensor::wifi: Wifi DHCP info: IpInfo { ip: 192.168.6.108, subnet:
    Subnet { gateway: 192.168.6.1, mask: Mask(24) }, dns: Some(192.168.6.1)
    , secondary_dns: Some(0.0.0.0) }
Wifi sta activado true
I (2675) sensor: Event received: WifiConnected
I (2685) sensor::fsm: ******** Entering state WifiConnected
I (2685) sensor::fsm: State WifiConnected.
I (2695) sensor::mqtt: Sending mqtt welcome message.
I (2695) sensor::mqtt: Subscribing to mqtt topic /rust/command
W (2705) sensor::mqtt: mqtt debug: received from mqtt client: Ok(
    BeforeConnect)
I (2725) wifi:<ba-add>idx:1 (ifx:0, 78:d2:94:a8:21:6b), tid:0, ssn:0,
    winSize:64
W (2875) sensor::mqtt: mqtt debug: received from mqtt client: Ok(Connected(
    false))
```

```
64 I (2885) sensor::fsm: Connected to MQTT server.
65 I (2885) sensor: Event received: MqttConnected
66 I (2885) sensor::fsm: ******** Entering state ServerConnected
67 I (2895) sensor::fsm: State ServerConnected. Start sending periodic data.
68 W (2925) sensor::mqtt: mqtt debug: received from mqtt client: Ok(Subscribed
       (25000))
69 W (2975) sensor::mqtt: mqtt debug: received from mqtt client: Ok(Published
       (47203))
70 I (5665) sensor::shtc3: Temperature reading: 29.666 °C
71 I (5665) sensor: Event received: SensorData(29.666)
72 I (5665) sensor::fsm: Sending temperature sensor data to MQTT: 29.666 °C
73 I (5675) sensor::mqtt: Sending mqtt data.
74 W (5745) sensor::mqtt: mqtt debug: received from mqtt client: Ok(Published
       (16489))
75 I (10665) sensor::shtc3: Temperature reading: 29.653 °C
76 I (10665) sensor: Event received: SensorData(29.653)
77 I (10665) sensor::fsm: Sending temperature sensor data to MQTT: 29.653 °C
78 I (10675) sensor::mqtt: Sending mqtt data.
79 W (10705) sensor::mqtt: mqtt debug: received from mqtt client: Ok(Published
       (64319))
80 I (10735) sensor: Inactive
81 [...]
```

Appendix B

Console logs for *embsensor* (`no-std`) example

This appendix shows the outputs of *embsens* (`no_std`) example.

Listing B.1 shows the console log connected to UART of the development board during execution.

Listing B.1: Console log during execution

```
I (43) boot: ESP-IDF v5.1-beta1-378-gea5e0ff298-dirt 2nd stage bootloader
I (43) boot: compile time Jun  7 2023 07:59:10
I (44) boot: chip revision: v0.3
I (48) boot.esp32c3: SPI Speed      : 40MHz
I (53) boot.esp32c3: SPI Mode       : DIO
I (57) boot.esp32c3: SPI Flash Size : 4MB
I (62) boot: Enabling RNG early entropy source...
I (68) boot: Partition Table:
I (71) boot: ## Label            Usage          Type ST Offset   Length
I (78) boot:  0 nvs              WiFi data        01 02 00009000 00006000
I (86) boot:  1 phy_init         RF data          01 01 0000f000 00001000
I (93) boot:  2 factory          factory app      00 00 00010000 003f0000
I (101) boot: End of partition table
I (105) esp_image: segment 0: paddr=00010020 vaddr=3c080020 size=1d9f4h
    (121332) map
I (140) esp_image: segment 1: paddr=0002da1c vaddr=3fc836c0 size=0123ch (
    4668) load
I (142) esp_image: segment 2: paddr=0002ec60 vaddr=3fca7b58 size=00168h (
     360) load
I (146) esp_image: segment 3: paddr=0002edd0 vaddr=40380000 size=01248h (
    4680) load
I (156) esp_image: segment 4: paddr=00030020 vaddr=42000020 size=7836ch
    (492396) map
I (270) esp_image: segment 5: paddr=000a8394 vaddr=40381248 size=02478h (
    9336) load
```

```
I (274) boot: Loaded app from partition at offset 0x10000
I (275) boot: Disabling RNG early entropy source...
Rust in IoT. embsens example
[ACEL] accelerations  =  X: +0.0005 Y: +0.0010 Z: +1.0059       GYRO  =  X:
    -0.3659 Y: -0.1220 Z: -0.5488
[RCV] Socket not connected yet...
[MQTT] Wait until network is connected...
WARN - esp_wifi_internal_tx 12290
WARN - Unknown wifi mode in link_state
[CON] start connection task
[CON] Device capabilities: Ok(EnumSet(Client | AccessPoint))
[CON] Starting wifi
[CON] Wifi started!
[CON] About to connect...
[FSM] signal received: AccelDataData([0.00048828125, 0.0009765625,
   1.0058594, -0.36585367, -0.12195122, -0.5487805])
[RCV] Socket not connected yet...
[RCV] Socket not connected yet...
[CON] Wifi connected!
[FSM] signal received: WifiStaConnected
[RCV] Socket not connected yet...
[MQTT] Waiting to get IP address...
[RCV] Socket not connected yet...
[HTU] buf [104, 8], word: 26632, temperatura: 24.557701
[HTU] buf [128, 174], word: 32942, humedad: 56.83188
[FSM] signal received: TempHumData { temp: 24.557701, hum: 56.83188 }
[FSM] mqtt connection not ready to send
[ACEL] accelerations  =  X: +0.0000 Y: +0.0039 Z: +1.0049       GYRO  =  X:
    -0.3659 Y: +0.0000 Z: -0.4878
[FSM] signal received: AccelDataData([0.0, 0.00390625, 1.0048828,
   -0.36585367, 0.0, -0.4878049])
[RCV] Socket not connected yet...
[RCV] Socket not connected yet...
[RCV] Socket not connected yet...
[RCV] Socket not connected yet...
[HTU] buf [104, 16], word: 26640, temperatura: 24.579155
[HTU] buf [128, 162], word: 32930, humedad: 56.80899
[FSM] signal received: TempHumData { temp: 24.579155, hum: 56.80899 }
[FSM] mqtt connection not ready to send
[RCV] Socket not connected yet...
[ACEL] accelerations  =  X: -0.0020 Y: +0.0015 Z: +1.0059       GYRO  =  X:
    -0.3049 Y: +0.1220 Z: -0.5488
[FSM] signal received: AccelDataData([-0.001953125, 0.0014648438,
   1.0058594, -0.30487806, 0.12195122, -0.5487805])
[RCV] Socket not connected yet...
Got IP: 192.168.209.102/24
[MQTT] connecting socket...
[MQTT] TCP socket connected to MQTT server!
[MQTT] Connected to MQTT broker
```

```
63 [RCV] Waiting for MQTT packets from server...
64 [HTU] buf [104, 12], word: 26636, temperatura: 24.568428
65 [HTU] buf [128, 138], word: 32906, humedad: 56.763214
66 [FSM] signal received: TempHumData { temp: 24.568428, hum: 56.763214 }
67 [FSM] mqtt connection not ready to send
68 [RCV] Waiting for MQTT packets from server...
69 Packet received in /embsens/command: Connack(Connack { session_present:
       false, code: Accepted })
70 [MQTT] Subscribe sent
71 [RCV] Waiting for MQTT packets from server...
72 [RCV] Waiting for MQTT packets from server...
73 Packet received in /embsens/command: Suback(Suback { pid: Pid(1),
       return_codes: [] })
74 [ACEL] accelerations  =  X: +0.0020 Y: +0.0034 Z: +1.0039       GYRO  =  X:
        -0.4878 Y: +0.1220 Z: -0.4878
75 [FSM] signal received: AccelDataData([0.001953125, 0.0034179688, 1.0039063,
        -0.4878049, 0.12195122, -0.4878049])
76 [RCV] Waiting for MQTT packets from server...
77 [HTU] buf [104, 8], word: 26632, temperatura: 24.557701
78 [HTU] buf [128, 146], word: 32914, humedad: 56.778473
79 [FSM] signal received: TempHumData { temp: 24.557701, hum: 56.778473 }
80 [FSM] publishing in /embsens/temperature temperature 24.557701 and humidity
        56.778473
81 [RCV] Waiting for MQTT packets from server...
82 [RCV] Waiting for MQTT packets from server...
83 Packet received in /embsens/command: Puback(Pid(4))
84 [RCV] Waiting for MQTT packets from server...
85 [RCV] Waiting for MQTT packets from server...
86 [ACEL] accelerations  =  X: +0.0024 Y: +0.0029 Z: +1.0073       GYRO  =  X:
        -0.3049 Y: +0.1220 Z: -0.6098
87 [FSM] signal received: AccelDataData([0.0024414063, 0.0029296875,
       1.0073242, -0.30487806, 0.12195122, -0.6097561])
88 [HTU] buf [104, 8], word: 26632, temperatura: 24.557701
89 [RCV] Waiting for MQTT packets from server...
90 Packet received in /embsens/command: Publish(Publish { dup: false, qos:
       AtMostOnce, pid: None, retain: false, topic_name: "/embsens/command",
       payload: [123, 32, 99, 111, 109, 109, 97, 110, 100, 58, 32, 116, 101,
       115, 116, 32, 125] })
91 [HTU] buf [128, 150], word: 32918, humedad: 56.786102
92 [FSM] signal received: TempHumData { temp: 24.557701, hum: 56.786102 }
93 [FSM] publishing in /embsens/temperature temperature 24.557701 and humidity
        56.786102
94 [RCV] Waiting for MQTT packets from server...
95 [RCV] Waiting for MQTT packets from server...
96 Packet received in /embsens/command: Puback(Pid(5))
97 [RCV] Waiting for MQTT packets from server...
```