Bern University
of Applied Sciences

Embedded Development with Rust

Chaostreff Bern

2.10.2025

Pascal Mainini

Outline

Introduction

Pascal Mainini

e-mail : pascal.mainini@bfh.ch

- Computer scientist
- Focus on security, cryptography and hardware
- Tenure track at BFH
- Member of the institute ICE (cybersecurity and engineering)
- Long and broad industry experience
- *Teaching embedded Rust since 2021*

I do not like being recorded! Please ask me before making any recordings or taking pictures...

Goals Of This Talk

- Give an overview of the embedded Rust ecosystem.
- Show helpful abstractions: getting from bare metal to realtime OS.
- Provide a starting point and resources for embedded Rust.

> *...Clearly, this will be a fast and bumpy ride – fasten your seat-belts!*

Embedded Systems

- In the context of this talk, we speak of using Rust on **embedded systems**.
- Typically, this encompasses all sorts of **Micro-Controller Units (MCUs)**.
- These are generally small, integrated systems consisting of a CPU, RAM and peripherals.
- This imposes additional difficulties: **limited resources**, (specifically CPU and RAM).
- Examples presented are destined for **ARM Cortex-M4 MCUs**, specifically Nordic's **nRF52840**. [1]

  Single core, 64 MHz, 1 MiB flash / 256 KiB RAM

Cross-Compilation

Typically, development cannot be done on the MCU itself, we need to **cross-compile**. This requires a toolchain (compiler, linker, ...) for the **target architecture**, e.g. **Armv7E-M** for the nRF52840.

For supported architectures, in Rust simply the appropriate **target** ("`thumbv7em-none-eabihf`") can be installed using `rustup`:[1]

```
rustup target install thumbv7em-none-eabihf
```

It is also recommended to install "`cargo-binutils`", which provides useful shortcuts to the correct per-architecture binutils version:

```
rustup component add llvm-tools-preview # required by cargo-binutils
cargo install cargo-binutils
```

⚠ Targets and components are installed per toolchain (e.g. nightly)!
Use "`--toolchain`" to install for a different toolchain, e.g.

```
rustup target install --toolchain nightly thumbv7em-none-eabihf
```

[1] See [2] for a list.

Bare Metal

no_std Rust

Due to the constraints of embedded systems, they normally run in a **no_std** environment:

- Linked against the **core** crate instead of **std**
- Typically used for OS kernels, bootloaders and firmware.
- No OS support for things like memory allocation, multi-threading, CLI arguments etc.
  In particular, no support for dynamic data structures like **vec**.[2]
- The executable runtime must be set up by the executable itself.
- *Many crates do not work.*

At the very basic level, this requires doing two things:

1. Using the appropriate attributes to designate code as **no_std**:

```
#![no_main] // no standard main requiring CLI arguments etc.
#![no_std] // only link against core
```

2. Provide a panic handler (called on any panic, e.g. using the **panic!()** macro etc.):

```
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

[2] See [3] for a potential solution.

Cortex-M4 Initialization

> When programming **bare metal**, no OS and no runtime / standard library is available; we need to set up everything ourselves.

For the Cortex-M4, the following steps are required:

1. Set up the **vector table** (at address `0x00000000`):
   - Set initial address of the **stack pointer**.
   - Set the address of the **reset handler**.
   - Set addresses for other handlers (NMI, hard fault, ...).
2. Initialize memory (static and global variables):
   - Set the `.bss` section to zero.
   - Initialize values in the `.data` section.
3. Jump to application code, e.g. “`main()`”!

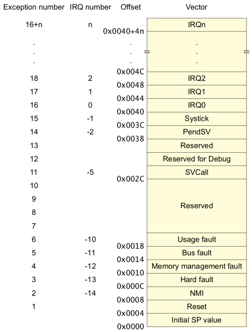

Figure: Cortex-M4 Vector Table [4]

Example: `bare-metal-m4`

Rust Embedded

Rust Embedded Crates

Embedded crates provide library support for different MCUs. We distinguish the following kinds of crates:

- **Micro-architecture crates:** Access to CPU functionality, registers and common peripherals. Example: `cortex-m`, [5].
- **Peripheral access crates (PAC):** Wrappers for register names etc. of a specific MCU. E.g. `nrf52840-pac`, [6].
- **HAL crates:** APIs for generic peripherals like timers, serial ports, GPIOs, etc. Typically implementing traits from `embedded_hal` ([7]). Example: `nrf52840-hal`, [8].
- **Board support crates (BSP):** Crates for a specific board with pre-configured devices (e.g. LEDs/buttons/sensors). Example: `stm32f3-discovery`, [9].

Rust Embedded Crates

The following figure shows the different kinds of embedded crates and their level of abstraction in relation to MCU functionality:

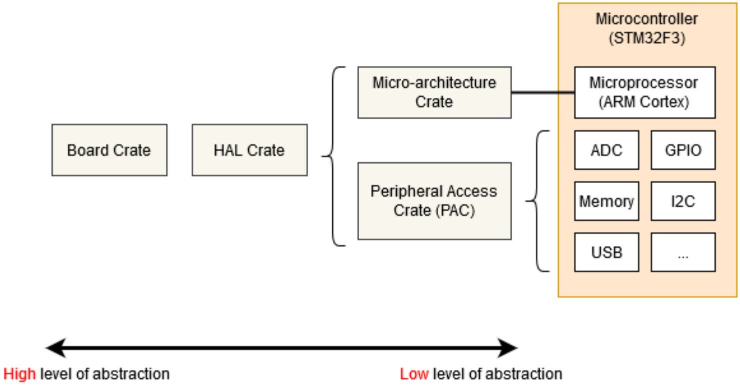

Figure: Types of Rust Embedded Crates [10]

Runtime: The `cortex-m-rt` Crate

The crate `cortex-m-rt` ([11]) provides code for startup and initialization of the runtime, as we did before. Most importantly:

- Initialization of the vector table (and stack pointer)
- Initialization of static variables
- Hooks for exception- and interrupt handlers

It drastically reduces boiler plate required:

```rust
#![no_std]
#![no_main]

use panic_halt as _;      // a second crate providing #[panic_handler]

#[cortex_m_rt::entry]     // all the magic happens here
fn main() -> ! {
    loop {} // ..
}
```

Micro-architecture Crate: `cortex-m`

The crate `cortex-m` ([5]) provides access to common peripherals of all ARM Cortex-M CPUs, like e.g. the system timer (SysTick):

4.4 System timer, SysTick

The processor has a 24-bit system timer, SysTick, that counts down from the reload value to zero, reloads, that is wraps to, the value in the SYST_RVR register on the next clock edge, then counts down on subsequent clocks.

Note

When the processor is halted for debugging the counter does not decrement.

The system timer registers are:

Table 4-32 System timer registers summary

Address	Name	Type	Required privilege	Reset value	Description
`0xE000E010`	SYST_CSR	RW	Privileged	a	*SysTick Control and Status Register*
`0xE000E014`	SYST_RVR	RW	Privileged	Unknown	*SysTick Reload Value Register* on page 4-34
`0xE000E018`	SYST_CVR	RW	Privileged	Unknown	*SysTick Current Value Register* on page 4-35
`0xE000E01C`	SYST_CALIB	RO	Privileged	-a	*SysTick Calibration Value Register* on page 4-35

Figure: Description of System Timer Registers [4]

Micro-architecture Crate: SysTick Peripheral

Example: Using the common Cortex-M system timer peripheral.

```rust
use cortex_m::peripheral::{syst::SystClkSource, Peripherals};

// Singleton
let p = Peripherals::take().unwrap();

// Get SysTick and have it use the system/core clock
let mut systick = p.SYST;
systick.set_clock_source(SystClkSource::Core);

// Divide the clock by 1 million and start counting
systick.set_reload(1_000_000);
systick.clear_current();
systick.enable_counter();

// ...

// Use it as delay
while !systick.has_wrapped() {}
```

PAC: Accessing Peripherals

PAC crates are often automatically generated from SVD files using **svd2rust** [12].[3] Such PACs contain a struct `Peripherals` with a field for every peripheral available on the specific MCU, like e.g. `GPIO`, `PWM`, `TIMER` etc.

Every peripheral in turn is also a struct, providing access to a `RegisterBlock` struct containing the registers of that peripheral.

For example, the nRF52840 GPIO P1 peripheral:

```
let p = nrf52840_pac::Peripherals::take().unwrap();

let gpio = p.P1;    // nrf52840_pac::p1::RegisterBlock
gpio.dir.// ... -> DIR register (configuration)
gpio.out.// ... -> OUT register (writing)
```

[3] CMSIS-SVD is an XML format used by many vendors to describe peripherals of their MCUs.

Example: `temperature`

The Embedded HAL

The main objective of a **Hardware Abstraction Layer (HAL)** is, as the name implies, a uniform way to access hardware.[4]

For embedded Rust, such a HAL is provided by the `embedded-hal` crate, [7]. It provides *traits* to access **common peripherals**:

- GPIOs: `embedded_hal::digital::InputPin`, `embedded_hal::digital::OutputPin`
- Delays, I^2C, SPI, PWM

Additional *companion crates* provide more HW support or other execution models. Examples: `embedded-io` (Serial/UART, [13]), `embedded-hal-async` (non-blocking, [14]).

> Device-specific **HAL crates** consume peripherals from PAC crates and return `embedded-hal` implementations.
>
> Example: Crate `nrf52840-hal` provides the HAL implementation for the nRF52840.

[4] On the downside, HW-specific / potentially more efficient access is not possible.

Type Safe Peripherals

Thanks to Rust's type system and ownership concept, HW can be treated as data and misconfiguration is prevented at compile time!

```
let mut led = port1.p1_01;
// led is type P1_01<Disconnected>

led.set_high().unwrap();
// error: method cannot be called on `P1_01<Disconnected>`
//        due to unsatisfied trait bounds

let mut led = port1.p1_01.into_push_pull_output(Level::Low);
// now led is type P1_01<Output<PushPull>>

led.set_high().unwrap();
// ok!
```

Implementation of `embedded_hal::digital::v2::OutputPin` in [15]:

```
impl<MODE> OutputPin for Pin<Output<MODE>> {
    fn set_high(&mut self) -> Result<(), Self::Error> {
        // ... set pin high. note: requires mutable ref!
    }
}
```

Example: `systick-hal`

RTOS

RTOS (Increasing Abstraction)

For simple applications, bare-metal programming might be enough, but embedded systems become increasingly complex. This applies in particular to IoT devices, but is also true for other systems.

With *advanced peripherals* and *software stacks*, managing resources and concurrency, while meeting *strict timing requirements* is a difficult task. Examples include:

- Networking, e.g. Ethernet/WiFi, BLE (Bluetooth), IEEE 802.15.4, LoRa, ...
- USB device or host stack
- Network stacks and protocols, e.g. TCP/IP, ZigBee, LoRaWAN, ...

In general, applications with such requirements will use an embedded-, or more specifically a **Real-Time Operating System (RTOS)**, which provides the required functionality and eases development.

RTOS Examples

There is a large number of open source and proprietary RTOS available today (see e.g. [16]). Some examples include:

- **Apache Mynewt:** ASF, *NimBLE stack*, Apache license
- **Contiki:** BSD license
- **FreeRTOS:** Amazon, *large user base (e.g. used in ESP-IDF)*, MIT license
- **Mbed OS:** developed by ARM, Apache license
- **QNX:** Blackberry, proprietary
- **RIOT:** Academia, *RUST friendly*, LGPL License
- **ThreadX:** Eclipse Foundation (formerly Microsoft), MIT license
- **VxWorks:** Wind River (Intel), e.g. used in Mars rovers, proprietary
- **Zephyr:** Hosted by the Linux Foundation, *large board support*, Apache license

Due to the increasing number of IoT appliations, interest of cloud providers in RTOS has grown: FreeRTOS (AWS), ThreadX (Azure) and Zephyr (Google, Nordic and others).

Rust RTOS

Nowadays, the majority of RTOS is written in C and assembly language, only few are written in Rust so far. This might change in the future due to interesting language properties, like e.g. memory safety and safe concurrency.

Rust RTOS examples:[5]

- **Ariel:** Built on top of Embassy and other projects, [19]
- **Bern:** Master's thesis at BFH, [20]
- **Drone:** Targets hard real-time applications, [21, 22]
- **Hubris:** By Oxide computers, for their HW, [23]
- **Tock:** For running concurrent, distrustful applications, [24, 25]
- **Xous:** OS for the Betrusted project, [26, 27, 28]

Besides those, there are also the embedded frameworks **RTIC** and **Embassy**, [29, 30]. See also [31] for more Rust RTOS.

---

[5] ☞ For a general-purpose OS in Rust, have a look at Redox [17]. A comprehensive introduction to OS development with Rust is given in [18].

Hardware Support and Abstraction

Supported hardware *depends on the RTOS*. Some may support only a few architectures or even just a single one, while others support many. In general, Cortex-M MCUs, and increasingly also the RISC-V architecture, are well supported.

Depending on the way an RTOS abstracts hardware, support for different kinds of peripherals varies:

- **No abstraction:** The RTOS relies on Rust Embedded abstractions, i.e. on `embedded-hal`, PACs etc.
- **Vendor abstraction:** Abstractions provided by the vendor (e.g. CMSIS-SVD) are used for accessing HW. Example: Drone.
- **Custom HAL:** OS-specific implementation of HW drivers. Example: Tock.

Rust on Non-Rust RTOS

Besides bare metal Rust and pure Rust RTOS, it is also possible to write (parts of) embedded applications for non-Rust RTOS. Some examples:

- **esp-idf-hal:** Espressif MCUs have good Rust support by vendor and community. A HAL interfacing with ESP-IDF (FreeRTOS) is available, also providing `std` support. See [32, 33] for details.
- **FreeRTOS-rust:** The `main()` function can be written in Rust and the global memory allocator can be used. [34]
- **RIOT:** For certain platforms, RIOT provides direct support for Rust. [35]
- **Rust on Zephyr RTOS:** Provides bindings for all syscalls, safe wrappers for some Zephyr APIs and also allocator support. [36]

Writing Rust applications for non-rust RTOS may be an interesting path for integration in existing environments and benefitting from additional guarantees in the business logic.

Async Rust

Runtimes

While futures and the `async`/`await` keywords are part of the Rust language, the **executor** is not. It handles running and awaiting tasks and is responsible for the *in-process scheduling*.

To run async Rust programs, an **async runtime** with an executor is required. By separating it from the language, different implementations for different targets are possible. Well-known examples are:

- **Tokio:** De-facto standard runtime. [37]
- **Smol:** Minimalistic runtime, composed of various other crates. [38]
- **RTIC:** Small, HW-based executor. [39]
- **Embassy:** Runtime for embedded systems. [30]

Tokio and Embassy are both small ecosystems on their own. Tokio e.g. brings a whole stack of client/server libraries and Embassy has its own HAL implementations.

On the potential downside, this leads to a large number of crates depending on a specific runtime – replacing it with a different one is difficult.

RTIC: Real-Time Interrupt-driven Concurrency

RTIC (Real-Time Interrupt-driven Concurrency, [39]) is a framework for **resource sharing** and **interrupt handling**:

- Macro-based framework, not a full-fledged RTOS
- Supports all Cortex-M MCUs
- Guarantees **deadlock-free execution** at compile time
- Most of the **scheduling is done in hardware** using interrupts and interrupt priorities
- Supports **preemptive multitasking**
- Has a low time- and memory overhead

"From RTIC's developers point of view; RTIC is a hardware accelerated RTOS that utilizes the hardware such as the NVIC on Cortex-M MCUs, CLIC on RISC-V etc. to perform scheduling, rather than the more classical software kernel."

☞ Refer to the RTIC book ([29]) for extensive documentation!

Example: `rtic-monotonic`

Embassy

Embassy is a runtime for embedded systems:

- `no_std` / no allocator needed
- Integrated timer for sleeping/delays
- No busy loop, sleeping with `WFI`

Besides the executor, different features and crates are available:

- HALs for ESP32, Nordic, RP2040/RP2350 and STM32 MCUs (other HALs possible)
  Some implement blocking traits from embedded HAL ([7]) also!
- Integration with Nordic SoftDevice for Bluetooth
- Networking- and USB-stack, bootloader

Comparison RTIC vs. Embassy:

- Both provide an async Rust executor
- RTIC does scheduling in HW using interrupts
- RTIC does not provide any HALs
- Parts of both projects may be combined!

Example: `embassy-blink`

Further Resources

Further Resources

Due to limited time, we have only scratched the surface and important topics could not have been presented. The following is a small selection of resources to support your potential further exploration of Embedded Rust:

- **Rust on Embedded Devices Working Group [40]:** Home of the efforts around Rust on embedded devices.
- **The Embedded Rust Book [41]:** "Classic" Rust book for embedded Rust – a good introduction.
- **Awesome Embedded Rust [42]:** Curated list of resources for embedded and low-level development in the Rust programming language.

Feel free to chat with me afterwards or contact me by mail: `pascal.mainini@bfh.ch`!

Discussion, Q&A

Bibliography

Bibliography I

[1] "Nordic Semiconductor Homepage, nRF52840."
`https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52840`.

[2] "The rustc Book, Platform Support."
`https://doc.rust-lang.org/stable/rustc/platform-support.html`.

[3] "crates.io, heapless."
`https://crates.io/crates/heapless`.

[4] "Arm DUI 0553, Cortex-M4 Devices, Generic User Guide."
`https://developer.arm.com/documentation/dui0553/latest/`.

[5] "crates.io, cortex-m."
`https://crates.io/crates/cortex-m`.

[6] "crates.io, nrf52840-pac."
`https://crates.io/crates/nrf52840-pac`.

[7] "crates.io, embedded-hal."
`https://crates.io/crates/embedded-hal`.

[8] "crates.io, nrf52840-hal."
`https://crates.io/crates/nrf52840-hal`.

[9] "crates.io, stm32f3-discovery."
`https://crates.io/crates/stm32f3-discovery`.

[10] "The Embedded Rust Book, Memory Mapped Registers."
`https://docs.rust-embedded.org/book/start/registers.html`.

[11] "crates.io, cortex-m-rt."
`https://crates.io/crates/cortex-m-rt`.

Bibliography II

[12] "crates.io, svd2rust."
https://crates.io/crates/svd2rust.

[13] "crates.io, embedded-io."
https://crates.io/crates/embedded-io.

[14] "crates.io, embedded-hal-async."
https://crates.io/crates/embedded-hal-async.

[15] "GitHub.com, nrf-rs/nrf-hal: nrf-hal-common/src/gpio.rs (v0.15.0)."
https://github.com/nrf-rs/nrf-hal/blob/v0.15.0/nrf-hal-common/src/gpio.rs#L293.

[16] "Wikipedia, Comparison of real-time operating systems."
https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems.

[17] "Redox – Your Next(Gen) OS."
https://www.redox-os.org/.

[18] P. Oppermann, "Writing an OS in Rust."
https://os.phil-opp.com/.

[19] "GitHub.com, ariel-os/ariel-os: Ariel OS is a library operating system for secure, memory-safe, low-power Internet of Things, written in Rust."
https://github.com/ariel-os/ariel-os.

[20] "Bern RTOS: A real-time operating system for microcontrollers written in Rust."
https://bern-rtos.org/.

[21] "Drone – An Embedded Operating System for writing real-time applications in Rust."
https://www.drone-os.com/.

[22] "The Drone Embedded Operating System."
https://book.drone-os.com/.

Bibliography III

[23] "Hubris, A small open-source operating system for deeply-embedded computer systems."
https://oxidecomputer.github.io/hubris/.

[24] "Tock Embedded Operating System."
https://www.tockos.org/.

[25] "Tock Tutorial."
https://book.tockos.org/.

[26] "betrusted.io | A security enclave for humans."
https://betrusted.io/.

[27] "Announcing Xous: the Betrusted Operating System."
https://xobs.io/announcing-xous-the-betrusted-operating-system/.

[28] "GitHub.com, betrusted-io/xous-core: The Xous microkernel."
https://github.com/betrusted-io/xous-core/.

[29] "RTIC Book."
https://rtic.rs/.

[30] "Embassy."
https://embassy.dev/.

[31] "Are We RTOS Yet?."
https://arewertosyet.com/.

[32] "GitHub.com, esp-rs: Rust on Espressif microcontrollers."
https://github.com/esp-rs.

[33] "GitHub.com, awesome-esp-rust: Awesome ESP Rust."
https://github.com/esp-rs/awesome-esp-rust.

Bibliography IV

[34] "GitHub.com, lobaro/FreeRTOS-rust: Rust crate for FreeRTOS."
`https://github.com/lobaro/FreeRTOS-rust`.

[35] "Using Rust in RIOT."
`https://doc.riot-os.org/using-rust.html`.

[36] "GitHub.com, tylerwhall/zephyr-rust: API bindings, libstd, and Cargo integration for running Rust applications on a Zephyr kernel."
`https://github.com/tylerwhall/zephyr-rust`.

[37] "Tokio - An asynchronous Rust runtime."
`https://tokio.rs/`.

[38] "GitHub.com, smol-rs/smol: A small and fast async runtime for Rust."
`https://github.com/smol-rs/smol`.

[39] "crates.io, rtic."
`https://crates.io/crates/rtic`.

[40] "Rust Embedded: Resources for Rust programming on embedded devices."
`https://rust-embedded.org/`.

[41] "The Embedded Rust Book."
`https://docs.rust-embedded.org/book/`.

[42] "GitHub.com, rust-embedded/awesome-embedded-rust: Curated list of resources for Embedded and Low-level development in the Rust programming language."
`https://github.com/rust-embedded/awesome-embedded-rust/`.