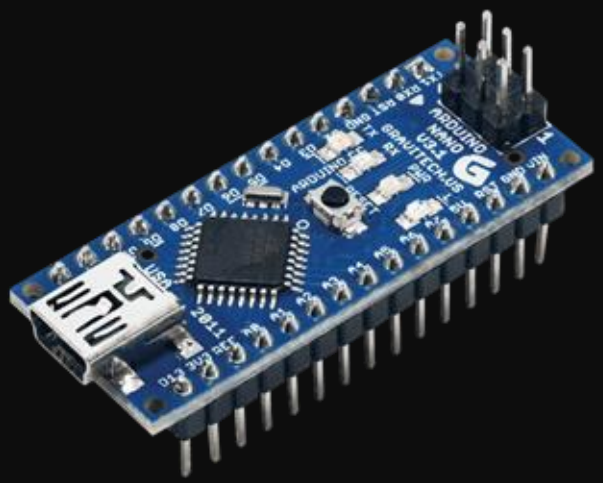
Bare Metal Meets Rust



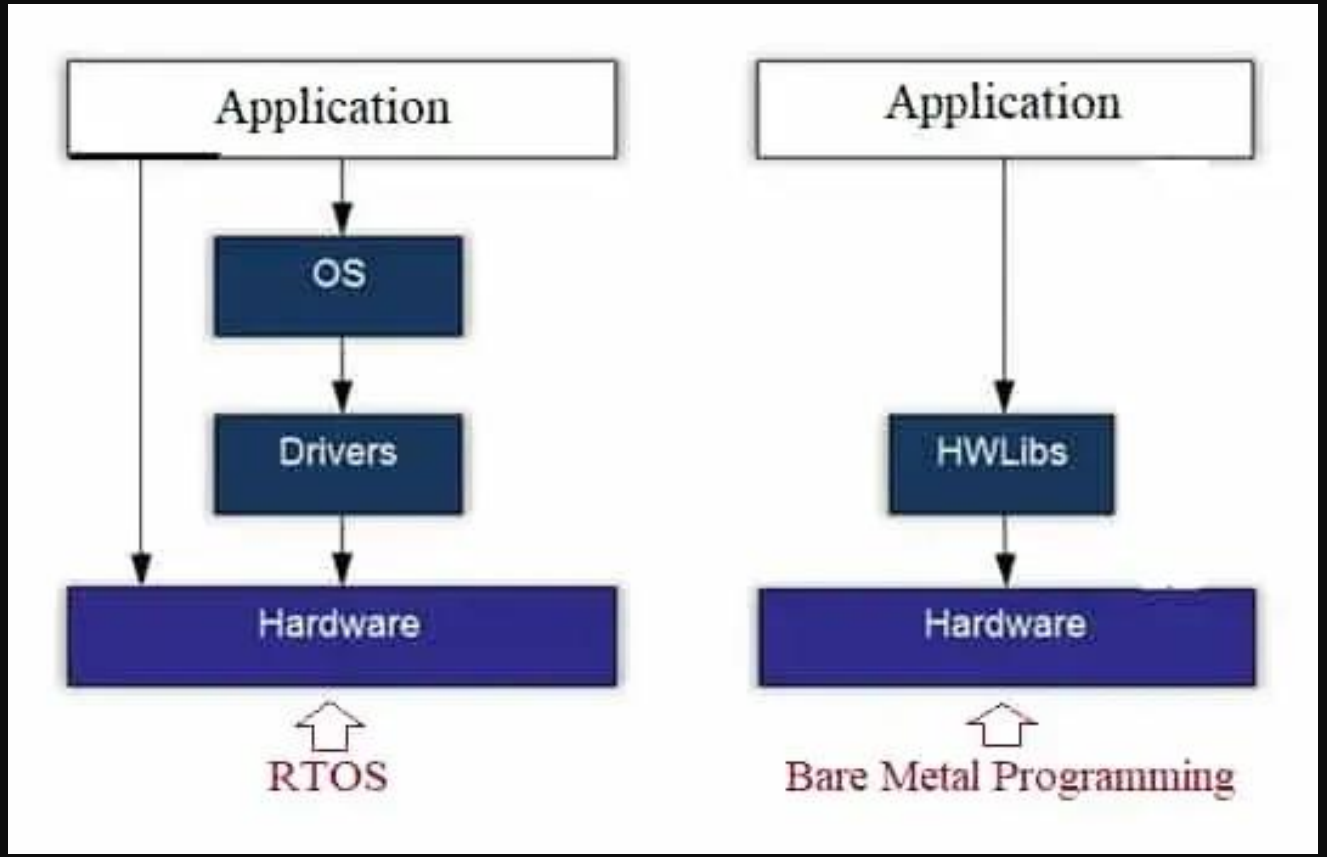
Overview

- Brief introduction to bare metal environments
- #![no_std], the core crate, and #![no_main]
- Rust features that are beneficial to embedded programming
- Embedded Rust Ecosystem: Runtime, PAC, and HAL crates
- Advantages and disadvantages of Rust in embedded systems
- Verdict: Rust or C?

Bare Metal Environments

- No OS
- Single statically linked firmware image
- No dynamic linking
- No processes
- Direct access to hardware registers

Bare Metal Environments



#![no_std]

- Every Rust binary links with the std crate by default.
- std assumes a hosted environment with OS services: file I/O, processes, threads, sockets, heap allocation, etc.
- #![no_std] prevents linking with std and uses core instead.

The core crate

core makes no assumptions about the underlying system, This means...

- No I/O abstraction, no filesystem, no networking
- No heap allocation (and therefore no heap-backed collections)
- No OS-backed threads or synchronization primitives
- No program entry point or runtime initialization

...But by disabling std, we gain the ability to write firmware, bootloaders, and kernel code.

#![no_main]

- Not strictly required but commonly used in conjunction with #![no_std].
- Indicates that the standard Rust entry point (fn main) will not be used.
- Requires the program to define a custom entry point (e.g., reset handler).

Minimal `#![no_std]` Program

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

use core::panic::PanicInfo;

#[panic_handler] // called on panic
fn panic(_: &PanicInfo) -> ! {
        loop {}
}

#[unsafe(no_mangle)] // prevent name mangling
pub extern “C” fn _start() -> ! { // use C ABI
        loop {}
}
```

`#![no_std]` Portability

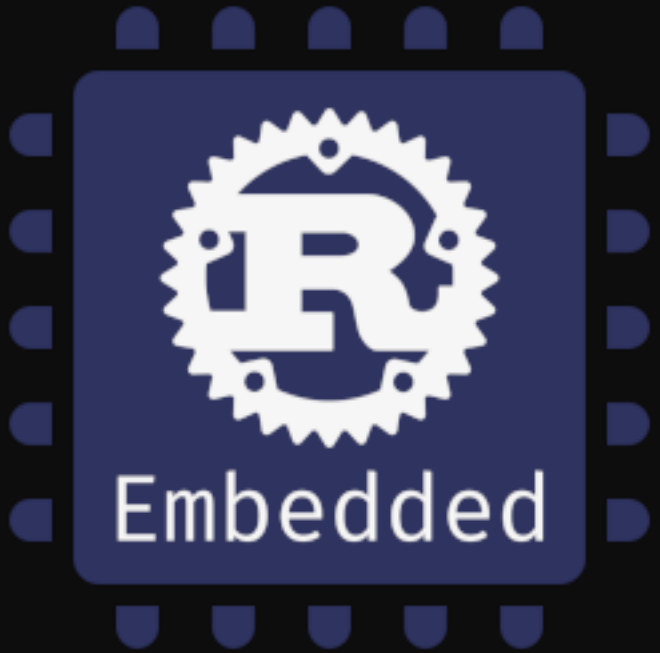
- In reality, std is just core + OS abstractions.
- Explicitly disables standard library features, unlike C, where nothing enforces the stdlib boundary.
- A driver crate written for an STM32 can be compiled and unit tested on PC without any hardware.
- Many crates in the embedded ecosystem are also useable in a hosted context!
- Even the Linux kernel uses #![no_std]!

Cross-Compilation

- In C, a separate cross-compiler is needed per target.
  - Separate install for each
  - Can be PM or vendor-specific
  - Build system configuration is manual
  - 😵‍💫
- In Rust, just do rustup target add <target>
  - Get the official toolchain
  - Just declare target-triple in .cargo/config.toml, and
  - cargo build automatically cross-compiles

Awesome Rust Features for ES

- Ownership & Borrowing
- Type safety
- Typestate patterns
- Rust “OOP”
- Traits



Ownership & Borrowing

- In C, a pointer can be accessed by many functions concurrently.
- In Rust, every value has **exactly one** owner.
- Can have multiple borrows OR one mutable borrow, never both.
- Borrow checker automatically blocks use-after-free.
- References cannot outlive the values they point to.

A Cry For Help

…

//! NOTE: call spi_init() before spi_transfer()

//! NOTE: check DMA completes before freeing buf

//! NOTE: must have only ONE driver hold the bus at a time

//! NOTE: if you're reading this at 2am, I'm sorry

…

Case Study 1: Use-After-Free

```c
// C

uint8_t *buf = malloc(64);
fill_buffer(buf, 64);
DMA_start_transfer(buf, 64); free(buf);

// no error – silent memory  // corruption.
```

```rust
// Rust

let buf = vec![0u8; 64];
dma.start_transfer(&buf);
drop(buf);

// compile error – can't drop // borrowed values
```

Case Study 2: Multiple Ownership

```c
// C

spi_handle_t spi = spi_init(SPI1);
flash_init(&flash, spi);
display_init(&display, spi);

flash_read(&flash, addr, buf, len);
display_write(&display, framebuf);

// silent failure – corrupted      // transaction at
runtime
```

```rust
// Rust

let spi = Spi::new(…)
let mut flash = Flash::new(spi);
let mut display = Display::new(spi);

// compile error – use of moved value
```

Type Safety

- C's type system is simply *too shallow* – nearly anything can be interpreted as an int, conversions are silent.
- Nothing prevents using one integral type where another was expected.
- By contrast, Rust's newtype pattern makes declared types distinct.
- Type distinction only exists at compile-time, zero runtime cost!

Case Study: Unit Mismatch

```c
// C

void sleep(uint32_t duration);

sleep(1000); // ms? µs?

sleep(some_reg_addr);
sleep(some_adc_reading);

// both are valid, compiler doesn't // care!
```

```rust
// Rust

struct Milliseconds(u32);
struct Microseconds(u32);

fn sleep(duration: Milliseconds);

sleep(Milliseconds(1000)); // OK

sleep(Microseconds(1000)); // error
sleep(1000);               // error
```

Typestate Patterns

- In C, object state is runtime data – compiler sees black box
- In Rust, object state can be made *part of* its type using generic parameters.
- Method set changes with state. Calling an invalid method fails because it *doesn't exist* for that state.
- State transitions are explicit; transition consumes previous state and returns the next.

Case Study: GPIO Pin Configuration

```c
// C

GPIO_InitTypeDef PA5 = {0};

// init PA5 as input
PA5.Pin = GPIO_PIN_5;
PA5.Mode = GPIO_MODE_INPUT;
PA5.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOA, &PA5);

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5,
GPIO_PIN_SET); // silently fails
```

```rust
// Rust

let gpioa = dp.GPIOA.split();
let pa5 = gpioa.pa5;

pa5.set_high(); // invalid

let mut led = pa5.into_push_pull_output();

led.set_high(); // OK
```

Rust “OOP”

- Embedded C is polluted with free functions and macros with little to no parameter checking.
- Classes in C++ give encapsulation at the cost of binary overhead and unpredictability.
- In Rust, impl attaches methods directly to structs. Purely compile-time.
- Structs are distinct types that fully encapsulate the logic of objects they represent.

Case Study 1: Timer Initialization

```c
// C

TIM_HandleTypeDef htim;

htim.Instance = TIM2;
htim.Init.Prescaler = 8399;
htim.Init.CounterMode = TIM_COUNTERMODE_UP;
htim.Init.Period = 9999;
htim.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim.Init.AutoReloadPreload =
                TIM_AUTORELOAD_PRELOAD_DISABLE;

HAL_TIM_Base_Init(&htim);
HAL_TIM_Base_Start(&htim);
```

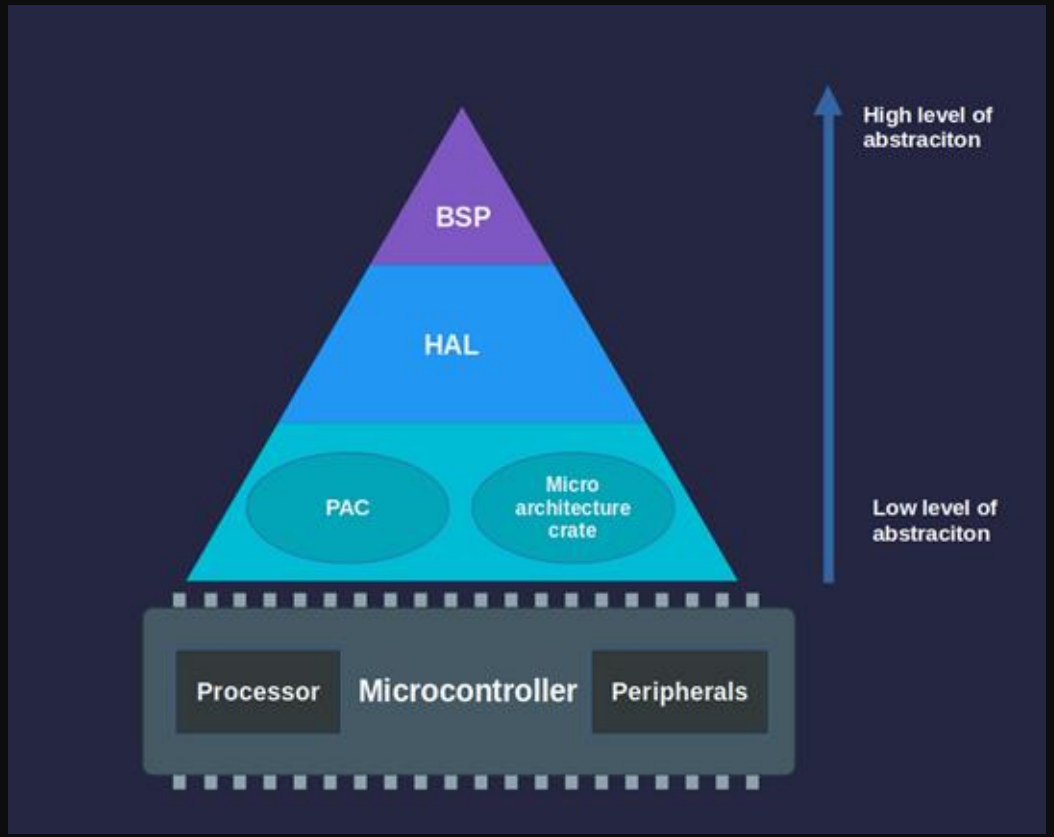
```rust
// Rust

let mut timer =          dp.TIM2.counter_hz(&clocks);

timer.start(1.Hz()).unwrap();
```

Case Study 2: SPI Baud Rate

```
// C

SPI1->CR1 &= ~SPI_CR1_SPE;
SPI1->CR1 = (SPI1->CR1 & ~(0x07 << 3)) | (prescaler_value << 3);
SPI1->CR1 |= SPI_CR1_SPE;

// Rust

spi.cr1.modify(|_r, w| w.spe().clear_bit());
spi.cr1.modify(|_r, w| w.br().bits(prescaler_value));
spi.cr1.modify(|_r, w| w.spe().set_bit());
```

Rust Embedded Ecosystem

Abstraction Layers



Low-Level Hardware Access

Peripheral Access Crates (PACs)

- Provide typed access to memory-mapped registers

- Essentially a Rust representation of the device reference manual

Microarchitecture Crates

- Provide CPU-specific primitives

- Examples include functions for enabling/disabling interrupts or accessing special registers

Runtime crates

Runtime crates are typically provided by architecture ecosystems or hardware vendors (e.g. cortex-m-rt, esp-rt).

These crates:

- Provide startup code and memory initialization

- Set up vector tables, stacks, and reset handlers

- Provide #[entry] and #[interrupt] macros

I.O.W., take care of runtime initialization and low-level boilerplate, allowing developers to focus on application logic.

Hardware Abstraction Layer (HAL)

Hardware vendors and the Rust community also provide HAL crates for specific MCUs or boards (e.g. esp-hal, stm32f4xx-hal), which are built on top of the PACs of those platforms.

These crates:

- Wrap memory-mapped registers in safe Rust interfaces

- Provide APIs for GPIO, SPI, I2C, UART, timers, etc.

Rust's type safety and traits ensures correct peripheral usage and prevents common errors like writing to the wrong bits or registers.

The embedded-hal crate

- Defines a set of traits for common peripherals such as SPI, I2C, GPIO pins with only the interface contract, no implementation.
- A driver module/crate written against embedded-hal traits works on **any** MCU that implements them.
- Completely platform agnostic. Specific hardware doesn't matter, only its capabilities.
- Compared to C/C++, where driver libraries are tightly coupled with a specific HAL.

The Ecosystem

Traditionally in embedded C, tools have been largely vendor-driven.

- E.g. STM32CubeIDE and HAL, ESP-IDF, Nordic nRF5 SDK

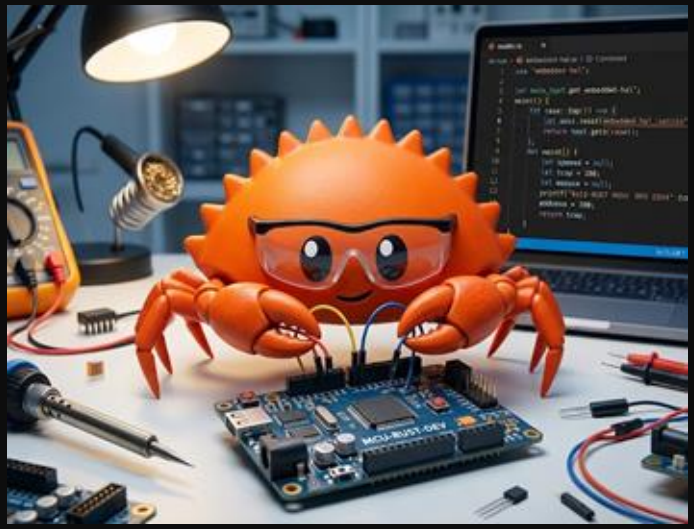
Rust sees a lot more community involvement in the development of core embedded tooling.

- Greater incentive to be platform or vendor-agnostic
- More transparency to users
- E.g. embedded-hal, embassy, probe-rs

Comes with a slight tradeoff.

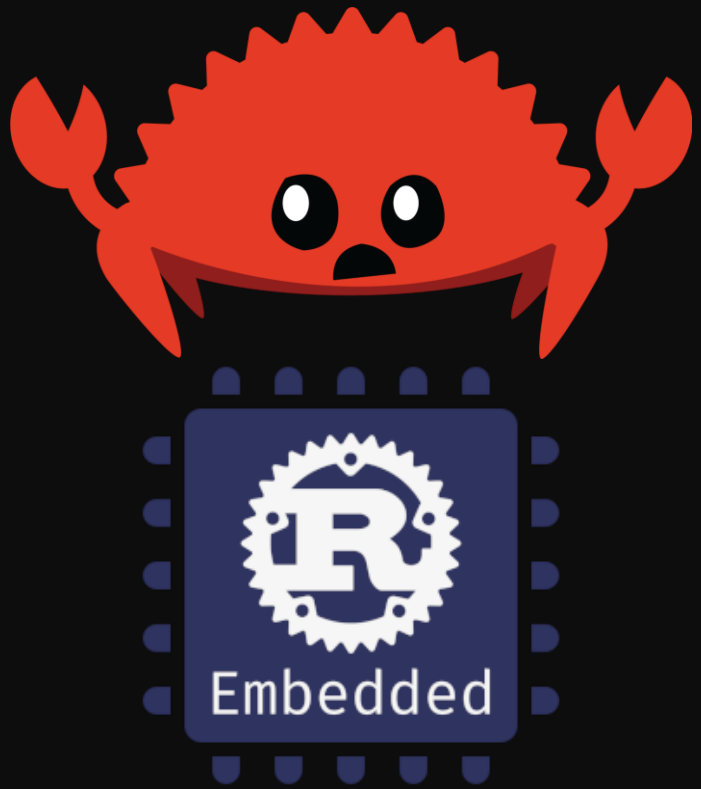
Why Rust is Good for ES

- Zero-cost abstractions
- Safe Rust *is* idiomatic bare metal programming.
- no_std enables truly portable embedded code.
- Compiler helps to enforce the reference manual.



Why Rust is Not So Good for ES

- Nascent ecosystem – Many crates and features still marked unstable.
- Yet to formulate best practices or coding standards.
- Docs can be uninformative.
- Slow to prototype with due to stricter language rules.



Conclusion & Closing Thoughts

- Rust’s strong type system, ownership model, and zero-cost abstractions provide memory safety without runtime overhead, making it a first-class language choice for programming embedded systems.
- Even without the standard library, Rust’s core features are sufficiently powerful to support robust RTOS implementations and async executors.
- Current limitations stem mostly from ecosystem maturity, which is improving rapidly as the embedded Rust community grows.

?

Any Questions?

About Me

- Third year Mechatronic Engineering & Computer Science student
- Interested in robotics, electronics, low-level systems, physics
- Seeking opportunities in embedded systems / robotics!