

Rust for Embedded Developers

Day 4

Rust on bare-metal

- Project setup & tooling
- `unsafe` Rust
- Safe abstractions
- Interrupts
- `async` on bare-metal

rustc platform support

Common bare-metal targets are "only" tier 2.

Why? Because GitHub Actions doesn't provide runners for these targets.

The difference to tier 1 is that tests are not run automatically for tier 2.

For context:

Google is shipping lots of Rust code in Android, which is also a tier 2 target.

esp-hal 1.0

released on October 30th 2025

Vendor support is progressing, slowly. 😊

Project setup & tooling

- cross-compilation toolchain
- probe-rs
- build configuration
- basic dependencies

cross-compilation toolchain

```
1 rustup target add thumbv6m-none-eabi
```

probe-rs

<https://probe.rs>

The build tool for embedded Rust development.
Handles debug probe discovery, flashing, integration with RTT and more.

Configure `cargo run` to use probe-rs:

```
1 # .cargo/config.toml
2 [target.thumbv6m-none-eabi]
3 runner = "probe-rs run --chip RP2040 --protocol swd"
```

build configuration

```
1  # .cargo/config.toml
2  [target.thumbv6m-none-eabi]
3  linker = "flip-link" # flips .stack with .bss+.data
4  rustflags = [ # embedded-specific compilation options
5      "-C", "link-arg=--nmagic",
6      "-C", "link-arg=-Tlink.x",
7      "-C", "link-arg=-Tdefmt.x",
8  ]
9
10 [build] # set the default build target
11 target = "thumbv6m-none-eabi"
```

basic dependencies

```
1  # Cargo.toml
2  [dependencies]
3  cortex_m      = "0.7"  # low-level access to Cortex-M processors
4  cortex-m-rt   = "0.7"  # "runtime": NVIC setup, stack painting etc.
5  defmt         = "1"    # efficient logging
6  defmt-rtt     = "1"    # logging over RTT
7  embedded-hal  = "1"    # standard traits for HALs
8  panic-probe   = "1"    # panic handler
9  rp2040-hal    = "0.10" # microcontroller HAL
10 rp-pico       = "0.9"  # BSP, if available
```

memory layout

Define memory layout in the linker script `memory.x`.

```
1 MEMORY {  
2     BOOT2 : ORIGIN = 0x10000000, LENGTH = 0x100  
3     FLASH : ORIGIN = 0x10000100, LENGTH = 2048K - 0x100  
4     RAM   : ORIGIN = 0x20000000, LENGTH = 256K  
5 }
```

Bare-metal hello world

```
1  #![no_std] // no standard library
2  #![no_main] // no libc-based entry-point
3
4  use defmt_rtt as _; // link defmt_rtt
5
6  #[panic_handler] // need custom panic handler
7  fn abort(_: &core::panic::PanicInfo) -> ! {
8      loop {} // in case of panic, do nothing...
9  }
10
11 #[rp_pico::entry] // custom entry-point
12 fn main() -> ! { // return type: "never"
13     defmt::info!("Hello, world!");
14     loop {} // we're not allowed to return
15 }
```

unsafe Rust

Allows us to turn off some safety checks when needed.

Use carefully!

unsafe blocks

```
1 fn main() {
2     // 1. dereference raw pointers
3     let x: i32 = unsafe { *core::ptr::null() };
4     // 2. call unsafe functions
5     unsafe {
6         std::alloc::dealloc(
7             1024 as *mut u8,
8             std::alloc::Layout::new::<String>(),
9         );
10    };
11 }
12
13 // function signature of `dealloc`
14 pub unsafe fn dealloc(ptr: *mut u8, layout: Layout);
```

unsafe functions

```
1  /// # Safety
2  ///
3  /// Do not call with argument 42.
4  unsafe fn print_int(x: i32) {
5      if x == 42 { // trigger undefined behavior
6          unsafe { *core::ptr::null::<u8>() };
7      }
8      println!("int: {x}");
9  }
10
11 fn main() {
12     // SAFETY: The argument is not 42.
13     unsafe { print_int(12) };
14 }
```

Two types of safety comments:

The callee documents its invariants, the caller documents how they're upheld.

unsafe blinky

```
1  const LED_GPIO: u8 = 25;
2  const GPIO_OUT_SET_REG: usize = 0xd000_0014;
3  fn turn_led_on() {
4      unsafe { ptr::write_volatile(GPIO_OUT_SET_REG as *mut usize, 1 << LED_GPIO); }
5  }
6  fn wait() {
7      for _ in 0..10_000_000 {
8          unsafe { asm!("nop", options(nomem, nostack, preserves_flags)); }
9      }
10 }
11 fn main() {
12     loop {
13         turn_led_on(); wait();
14         turn_led_off(); wait();
15     }
16 } // full, working code in lumatrix repo
```

no_std

Bare-metal targets don't have an OS, which `std` relies on.

There are two subsets of `std` we can use: `core` and `alloc`.

`core` is always usable, but `alloc` requires that we define a memory allocator.

It's usually a good idea to avoid using `alloc`.

heapless

`heapless` is a crate that provides fixed-size collections.

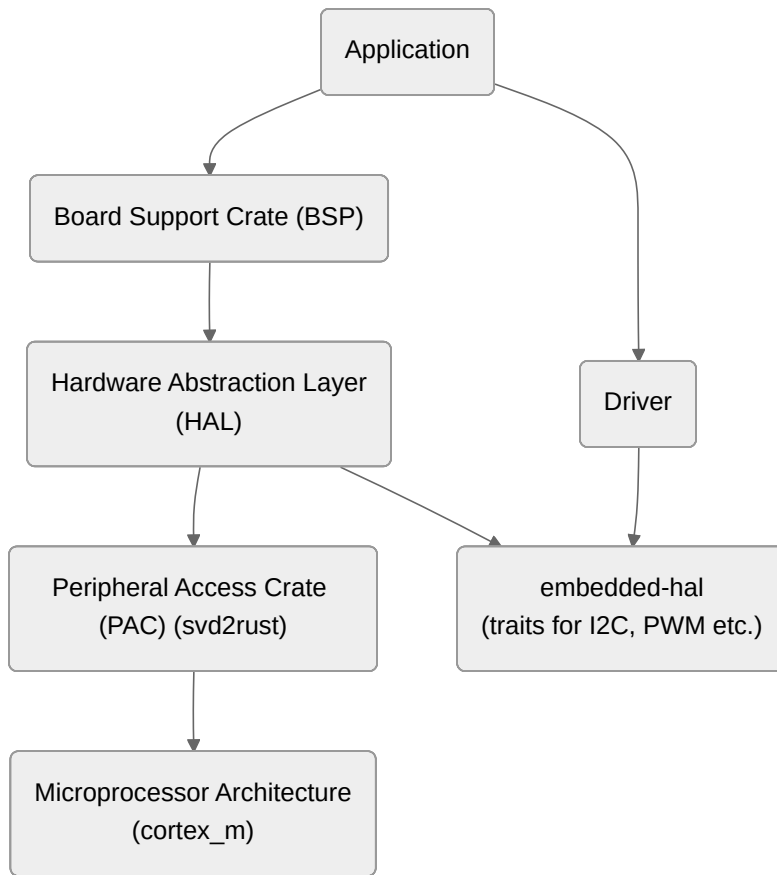
It's often a good substitute for `alloc`.

The only downside is, you have to specify the maximum capacity of a collection when you initialize it.

That way, it can be allocated on the stack.

<https://docs.rs/heapless>

Anatomy of the embedded Rust ecosystem



Safe abstractions

`unsafe` should be used as little as possible.

When we do use it, we should be able to reason about safety **locally**.

Using the **type system** and **encapsulation**, we expose only safe functions outside a module that cannot be abused.

Safely accessing registers

```
1  const LED_GPIO: u8 = 25;
2  const GPIO_OUT_SET_REG: usize = 0xd000_0014;
3  fn turn_led_on() {
4      // SAFETY: It's always OK to write to this register.
5      unsafe { ptr::write_volatile(GPIO_OUT_SET_REG as *mut usize, 1 << LED_GPIO); }
6  }
```

This is a good start.

But what if the GPIO is not correctly configured as output?

It wouldn't be a safety issue, but our code would be incorrect.

Enforcing correct configuration

```
1 pub fn configure_gpio_as_output() -> OutputGpio { /* ... */ }
2
3 pub struct OutputGpio {
4     // zero-sized, no runtime overhead
5     _private: (), // prevent instantiation of struct
6 }
7
8 impl OutputGpio {
9     pub fn turn_led_on(&self) { /* ... */ }
10 }
```

Better: Users must call `configure_gpio_as_output()` before `turn_led_on()`.

Problem: What if another part of our program calls `configure_gpio_as_input()`?

There will be two instances of a GPIO, responsible for the same registers.

Representing hardware as singletons

```
1 struct Gpio;
2
3 impl Gpio {
4     pub fn take() -> Option<Gpio> {
5         static mut ALREADY_TAKEN: bool = false;
6         if unsafe { ALREADY_TAKEN } {
7             None
8         } else {
9             unsafe {
10                 ALREADY_TAKEN = true;
11             }
12             Some(Gpio)
13     } } }
```

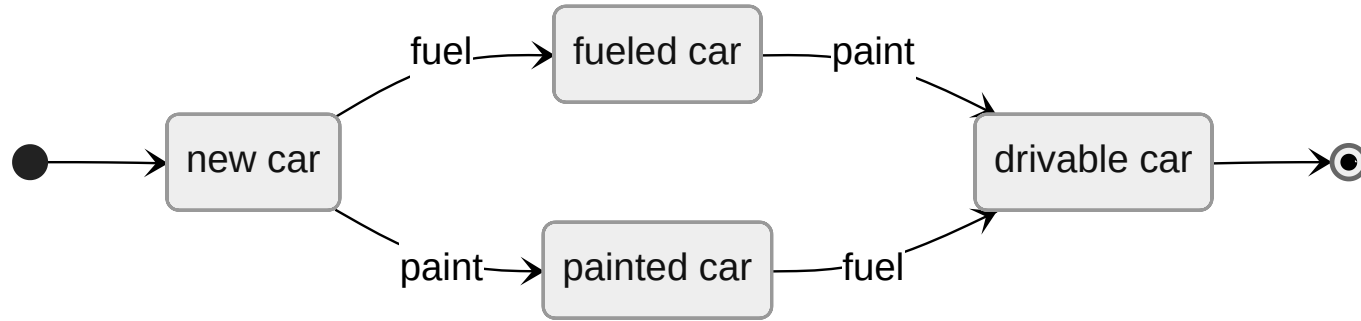
Safe code can only instantiate one `Gpio` instance. ✓

Safe singleton pattern with `cortex_m` macro

```
1 struct Gpio;
2
3 impl Gpio {
4     pub fn take() -> Option<&'static mut Gpio> {
5         cortex_m::singleton!(Gpio = Gpio)
6     }
7 }
```

Note: This returns a (static) mutable reference, not an owned value.
The implementation also uses a critical section for interrupt-safety.

Typestate pattern



Goals:

- Prevent driving before car is fueled and painted.
- Prevent fueling or painting twice.
- Allow fueling and painting in any order.

Limiting functions to specific states

```
1 struct Car<F, P> {
2     fuel: F,
3     paint: P,
4 }
5 impl Car<(), ()> {
6     fn new() -> Self {
7         Car {
8             fuel: (),
9             paint: (),
10        } } }
11 impl Car<u8, String> {
12     fn drive(&self) {}
13 }
14 fn main() {
15     let car = Car::new();
16     car.drive(); // ✗ no method named `drive` found for struct `Car<(), ()>`
17 }
```

State transitions

```
1  impl<P> Car<(), P> {
2      fn fuel(self, amount: u8) -> Car<u8, P> {
3          Car {
4              fuel: amount,
5              paint: self.paint,
6          } } }
7  impl<F> Car<F, ()> {
8      fn paint(self, color: &str) -> Car<F, String> {
9          Car {
10             fuel: self.fuel,
11             paint: color.into(),
12         } } }
13  fn main() {
14      Car::new().fuel(8).paint("blue").drive();
15      Car::new().paint("red").fuel(12).drive();
16  }
```

The downside: complicated generics

```
1  impl<I: PinId, F: func::Function, P: PullType> Pin<I, F, P> {  
2      /// Convert the pin function.  
3      pub fn into_function<F2>(self) -> Pin<I, F2, P>  
4      where  
5          F2: func::Function,  
6          I: func::ValidFunction<F2>,  
7      { /* ... */ }  
8  }
```

example from `rp2040-hal` :
Configuring the function of a GPIO pin.

Practice

<https://github.zhaw.ch/rust-wbk/lumatrix>

Interrupts

Because spinning in a busy loop is inefficient!

Sharing hardware

The main loop needs to setup and configure hardware.

The interrupt handler also needs to access it somehow.

How does this work with Rust's single ownership?

We use global locks:

```
1 static LED_PIN: LockMut<Pin<Gpio25, _, _>> = LockMut::new();  
2 static ALARM: LockMut<Alarm0> = LockMut::new();
```

`LockMut` is a handy type from `critical-section-lock-mut`.

The lock it provides is implemented with a critical section.

Setup code in main

complete code in `lumatrix/examples/blink_interrupt.rs`

```
1 fn main() {
2     // normal setup stuff...
3
4     // configure alarm to trigger interrupt
5     alarm.schedule(MicrosDurationU32::from_ticks(400_000)).unwrap();
6     alarm.enable_interrupt();
7     unsafe { NVIC::unmask(pac::Interrupt::TIMER_IRQ_0); }
8
9     // share hardware with interrupt handler
10    LED_PIN.init(pins.led.into_push_pull_output());
11    ALARM.init(alarm);
12
13    // interrupt handler takes over from here
14    loop { cortex_m::asm::wfi(); }
15 }
```

Writing the interrupt handler

complete code in `lumatrix/examples/blink_interrupt.rs`

```
1  #[interrupt]
2  fn TIMER_IRQ_0() {
3      static mut LED_STATE: bool = false;
4      if *LED_STATE {
5          LED_PIN.with_lock(|pin| { pin.set_low().unwrap(); });
6      } else {
7          LED_PIN.with_lock(|pin| { pin.set_high().unwrap(); });
8      }
9      ALARM.with_lock(|alarm| {
10         alarm.clear_interrupt();
11         alarm.schedule(MicrosDurationU32::from_ticks(400_000))
12             .unwrap();
13     });
14     *LED_STATE = !*LED_STATE;
15 }
```

async on bare-metal

writing interrupt-driven programs conveniently

Embassy

Embassy is currently the most popular `async` runtime for embedded systems.
(RTIC is another popular one.)

- Tasks are allocated statically.
- Interrupts wake the runtime.

async blinky

```
1  #[embassy_executor::main]
2  async fn main(spawner: Spawner) {
3      spawner.spawn(some_task()).unwrap();
4      spawner.spawn(another_task()).unwrap();
5  }
6
7  #[embassy_executor::task]
8  async fn some_task() {
9      do_something().await;
10 }
11
12 #[embassy_executor::task]
13 async fn another_task() {
14     do_something_else().await;
15 }
```

Practice

<https://github.zhaw.ch/rust-wbk/lumatrix>

The async examples are on a separate branch,
because the two different HALs are incompatible.

```
git checkout async
```