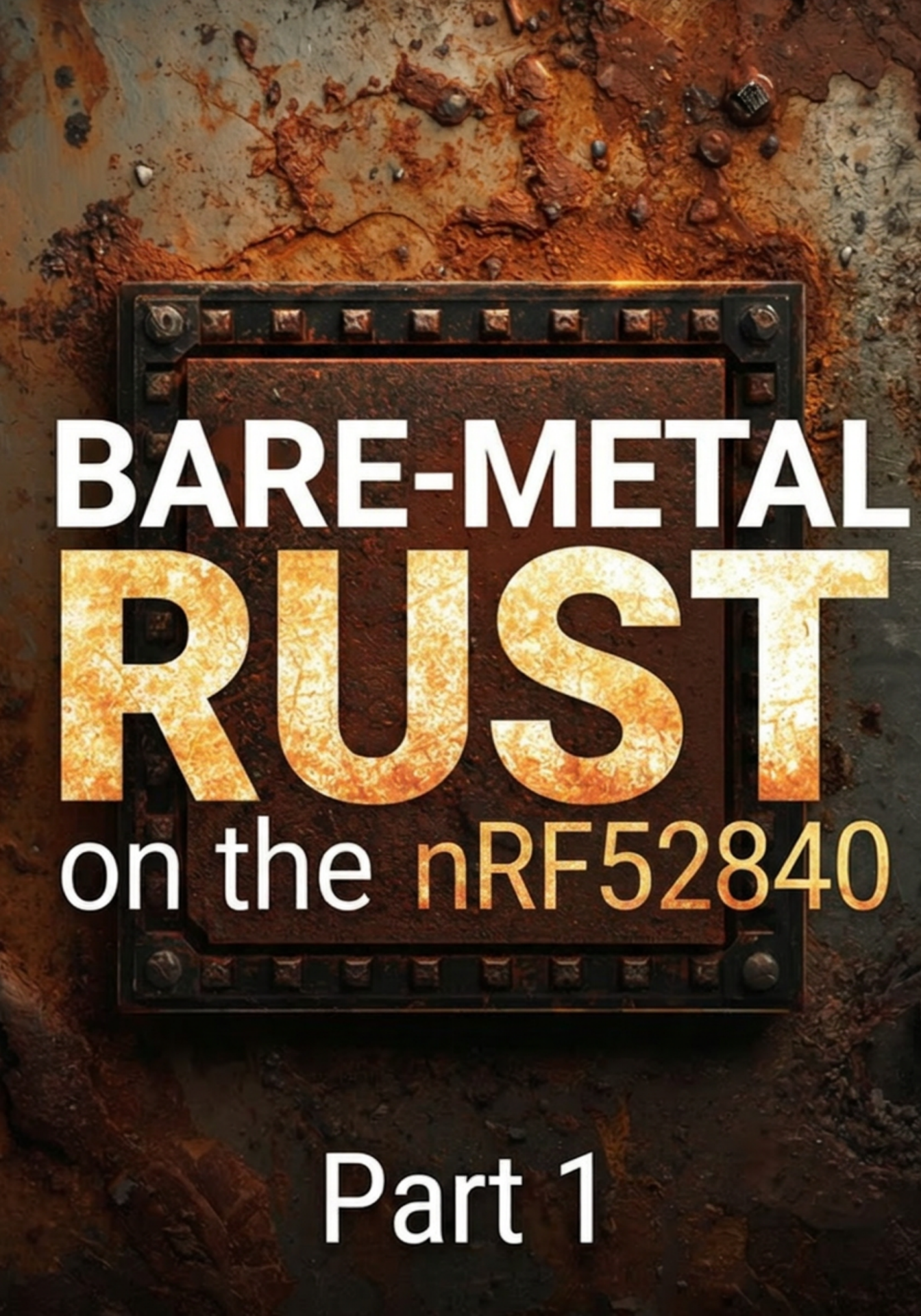
BARE-METAL
RUST
on the nRF52840
Part 1

# Bare-Metal Rust on the nRF52840

## A 10-Day Hands-On Guide to Embedded Systems Without an OS

Jovin Basil

This book is available at https://leanpub.com/bare-metalrustonthenrf52840

This version was published on 2026-06-27


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

# Contents

# Bare-Metal Rust on the nRF52840

## A 10-Day Hands-On Guide to Embedded Systems Without an OS

**Author:** Jovin

**Edition:** First Edition — 2026

**Target hardware:** Nordic nRF52840 DK (`nrf52840dk/nrf52840`) **Rust toolchain:** stable (thumbv7em-none-eabihf) **Key crates:** cortex-m-rt 0.7, nrf52840-hal 0.16, defmt 0.3, RTIC 1.1

* * *

*For every engineer who has wondered what the CPU is actually doing.*

## Copyright



**Disclaimer:** The information in this book is provided "as is." The author makes no warranties regarding accuracy or fitness for a particular purpose. Always verify hardware connections before applying power.

# Preface

I started this book with one question: *what does a microcontroller actually run when there is no operating system?*

RTOSes are excellent tools — but they abstract away the reset vector, the linker script, the interrupt table, the clock tree. You call a delay function and trust that it works. That trust is well-placed — but it is not understanding.

Rust changes the calculus for bare-metal development. Not because it is faster than C (it is about the same), and not because the API is nicer (some days it is, some days it isn't). It changes it because the compiler actively helps you avoid the class of bugs that kill embedded projects: uninitialized memory, race conditions in interrupt handlers, calling a function on the wrong peripheral type. These are not theoretical concerns — they are the kind of bugs that eat three-day debugging sessions.

This book covers ten days of hands-on work on the Nordic nRF52840 development kit. Each day introduces one concept, one buildable example, and four practice tasks. You start with the absolute minimum — a loop that does nothing — and finish with a multi-task concurrent application using the RTIC framework.

After these ten days you will understand:

- Why `#![no_std]` and `#![no_main]` are required
- How registers are accessed safely through the PAC and HAL layers
- How Rust's ownership model prevents data races in interrupt handlers
- How to log, debug, and profile without a UART or a display

You will not need to take my word for any of it — every claim is demonstrated by code that you can flash, run, and modify yourself.

**Jovin** June 2026

# How to Use This Book

## Hardware required

- Nordic nRF52840 DK (`PCA10056`)
- USB cable — data-capable, not charge-only
- A second USB cable or J-Link OB connection for UART (optional, Day 8)

The DK contains the nRF52840 SoC, an onboard J-Link OB programmer, four LEDs, four buttons, and a USB connector wired directly to the SoC's USB peripheral. Everything you need for Days 1–10 is already on the board.

## Software required

Install these before Day 1:

```
# Rust toolchain
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env
rustup target add thumbv7em-none-eabihf

# probe-rs — flash and debug tool
cargo install probe-rs-tools --locked

# Optional: cargo-size and cargo-objdump for binary inspection
cargo install cargo-binutils
rustup component add llvm-tools-preview
```

## Building and flashing

Each day's source lives in `rust_src/dayXX_name/`. Flash any day with:

```
cd rust_src/day01_toolchain_setup
cargo run --release
```

`cargo run` triggers the `runner` defined in `.cargo/config.toml`, which calls `probe-rs run` to flash and start the firmware. Logs (from Day 5 onward) stream to your terminal automatically.

## Reading the code

Each `src/main.rs` file is intentionally short — under 100 lines for most days. Read it alongside the chapter. The chapter explains *why*; the code shows *how*.

## Conventions

| Notation | Meaning |
|---|---|
| `Code font` | Rust identifier, crate name, or shell command |
| **Bold** | Term being defined for the first time |
| `// comment` | Explanation added to code in the chapter (not always in the file) |

**Tip** boxes contain shortcuts and productivity advice.

**Warning** boxes flag common mistakes that are hard to debug.

# Part 0 — Rust Language Foundations

*A compact Rust primer for engineers coming from C or another systems language. Skip this part if you are already comfortable with ownership, borrowing, and enums.*

The bare-metal chapters assume you can read Rust — not write it from scratch, but understand what `let mut`, `&`, `->`, `match`, and `Option` mean when you see them in firmware code. This part covers exactly those ten concepts, focused on the patterns that appear most often in embedded Rust.

Each topic also has a full daily write-up in the companion Rust learning blog (see the companion repository) if you want deeper explanation and exercises.

## Foundation 1 — Variables, Mutability & Basic Types

Rust variables are **immutable by default**. You must opt in to mutation with `mut`:

```
let x = 5;        // immutable — cannot be changed
let mut y = 5;    // mutable — can be reassigned
y += 1;           // ✅ works
// x += 1;        // ❌ compile error
```

**Shadowing** lets you reuse a name and even change its type — useful in firmware when converting a raw register value to a typed struct:

```
let pin = 13_u32;          // raw pin number
let pin = pin as usize;    // shadow with a different type — no mut needed
```

**Integer types** used most in embedded Rust:

| Type | Size | Use case |
|---|---|---|
| `u8` | 8-bit | SPI/I2C byte buffers |
| `u16` | 16-bit | ADC readings, timer counts |
| `u32` | 32-bit | Register values, bitmasks |
| `usize` | arch | Array indices |
| `i32` | 32-bit | Signed calculations, default |
| `bool` | 1-bit | Flags, pin states |

`const` values are inlined at compile time — prefer them over `#define`-style magic numbers in firmware:

```
const LED_PIN: usize = 13;
const CPU_FREQ_HZ: u32 = 64_000_000;
```

## Foundation 2 — Functions, Return Values & Expressions

Rust functions use **implicit return** — the last expression (no semicolon) is the return value. Explicit `return` is reserved for early exits.

```rust
fn cycles_per_ms(freq_hz: u32) -> u32 {
    freq_hz / 1_000          // implicit return — no semicolon
}

fn clamp(val: u32, max: u32) -> u32 {
    if val > max {
        return max;          // early return
    }
    val                      // implicit return
}
```

**Statements vs expressions** — adding a semicolon turns an expression into a statement (void). Forgetting this causes a type mismatch compile error:

```rust
fn broken() -> u32 {
    42;    // ✗ semicolon makes this a statement — returns () not u32
}

fn correct() -> u32 {
    42     // ✓ expression — returns 42
}
```

In firmware you will write many small helper functions of this shape: `fn <name>(<peripheral-type>) -> <result-type>`.

## Foundation 3 — Control Flow

Rust's `if` is an expression — it returns a value:

```
let level = if voltage_mv > 3300 { "high" } else { "low" };
```

**`loop`** is the primary bare-metal loop. It never exits unless you `break`:

```
#[entry]
fn main() -> ! {
    loop {
        // firmware main loop — runs forever
    }
}
```

**`while`** and **`for`** work as expected:

```
while !uart.is_ready() {}          // spin-wait

for byte in &rx_buffer[..len] {    // iterate a slice
    process(*byte);
}
```

`loop` with `break value` lets you return a value from a loop:

```
let result = loop {
    if let Some(b) = uart.read() {
        break b;
    }
};
```

Avoid unbounded `loop` in interrupt handlers — keep ISRs short and return quickly.

## Foundation 4 — Ownership: The Core Mental Model

Ownership is Rust's memory model. Three rules, no exceptions:

1. Every value has exactly one owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value is dropped (freed).

**Move semantics** — assigning a heap value transfers ownership:

```
let s1 = String::from("hello");
let s2 = s1;           // s1 is MOVED into s2 — s1 is gone
// println!("{}", s1); //  compile error: value used after move
```

**Copy types** (integers, booleans, fixed-size arrays) are copied, not moved — their entire value lives on the stack:

```
let a: u32 = 42;
let b = a;             // copy — both a and b are valid
```

In embedded Rust, most peripheral types are moved into the function or struct that "owns" the peripheral. This is intentional — it ensures only one part of the code can drive the peripheral at a time, eliminating races by construction.

```
fn init_led(pin: Pin<Output<PushPull>>) {
    // pin is moved here — caller can no longer use it
}
```

## Foundation 5 — References & Borrowing

A **reference** lets you use a value without taking ownership. The compiler guarantees references never outlive the value they point to (no dangling pointers).

```
fn print_id(id: &u32) {          // borrow — does not take ownership
    println!("ID: {}", id);
}

let device_id: u32 = 0xDEAD_BEEF;
print_id(&device_id);            // pass a reference
// device_id is still valid here
```

**Mutable references** let you modify the borrowed value, but only one mutable reference may exist at a time (prevents data races):

```
fn zero_buffer(buf: &mut [u8]) {
    for b in buf.iter_mut() {
        *b = 0;
    }
}
```

The borrow checker enforces this rule statically — shared state between an interrupt handler and `main` requires special constructs (see Day 7: GPIOTE Interrupts).

| | **Immutable ref `&T`** | **Mutable ref `&mut T`** |
|---|---|---|
| How many at once | Any number | Exactly one |
| Can modify value | No | Yes |
| Alias allowed | Yes | No |

## Foundation 6 — The Slice Type

A **slice** is a view into a contiguous sequence — no ownership, no allocation. In embedded Rust, slices are everywhere: DMA buffers, UART RX windows, flash pages.

```
let buffer: [u8; 64] = [0u8; 64];

let first_four: &[u8] = &buffer[..4];  // slice of first 4 bytes
let all: &[u8] = &buffer[..];          // slice of entire array
```

String slices (`&str`) work the same way:

```
let greeting = "Hello";         // &str — a slice of UTF-8 bytes in flash
```

Functions that accept slices work with any length, making them reusable across different buffer sizes — important when your DMA buffer size varies by peripheral:

```
fn send_over_uart(data: &[u8]) {
    for &byte in data {
        uart_write_byte(byte);
    }
}
```

Prefer `&[u8]` over `&[u8; N]` in function signatures — it accepts arrays of any size, not just `N`.

## Foundation 7 — Structs

Structs group related data. In firmware they model hardware state, configuration blocks, and driver handles:

```
struct DriverConfig {
    baud_rate:   u32,
    rx_pin:      u8,
    tx_pin:      u8,
    buffer_size: usize,
}

let config = DriverConfig {
    baud_rate:   115_200,
    rx_pin:      8,
    tx_pin:      6,
    buffer_size: 64,
};
```

**Tuple structs** — unnamed fields, useful for wrapping raw values with a type:

```
struct Millivolts(u32);
struct Milliseconds(u32);

let vcc = Millivolts(3_300);
let delay = Milliseconds(500);
// vcc + delay would be a compile error — different types even though both u32
```

**Update syntax** — create a new struct from an existing one, changing only some fields:

```
let fast_config = DriverConfig { baud_rate: 921_600, ..config };
```

## Foundation 8 — Methods on Structs

Methods are functions defined inside an `impl` block. `&self` borrows the struct (read-only), `&mut self` borrows it mutably, and `self` consumes it:

```
struct Led {
    pin: u8,
    state: bool,
}

impl Led {
    fn new(pin: u8) -> Self {          // constructor (associated function)
        Led { pin, state: false }
    }

    fn is_on(&self) -> bool {          // read-only method
        self.state
    }

    fn toggle(&mut self) {             // mutating method
        self.state = !self.state;
    }
}

let mut led = Led::new(13);
led.toggle();
```

**Associated functions** (no `self`) work like static constructors — `Led::new()` is the Rust idiom for `led_init()` in C.

## Foundation 9 — Enums & Pattern Matching

Enums in Rust can carry data – they are algebraic data types, far more powerful than C enums:

```
enum UartError {
    BufferFull,
    ParityError(u8),    // carries the offending byte
    Timeout,
}
```

`match` is exhaustive – the compiler forces you to handle every variant:

```
fn handle_error(err: UartError) {
    match err {
        UartError::BufferFull      => flush_and_retry(),
        UartError::ParityError(b)  => log_bad_byte(b),
        UartError::Timeout         => reset_peripheral(),
    }
}
```

`match` on integers is common for decoding protocol bytes:

```
match command_byte {
    0x01 => start_measurement(),
    0x02 => send_reading(),
    0xFF => reset(),
    _    => log_unknown(),    // wildcard – catches everything else
}
```

Forgetting a variant is a **compile error**, not a runtime bug. This is one of Rust's most valuable guarantees in firmware where unhandled cases cause silent failures.

## Foundation 10 — Option<T>

Rust has no `null`. Instead, values that may be absent are wrapped in `Option<T>`:

```
enum Option<T> {
    Some(T),   // value is present
    None,      // value is absent
}
```

In embedded Rust you see `Option` everywhere peripherals can be in an uninitialised or consumed state:

```
fn read_byte(uart: &mut Uart) -> Option<u8> {
    if uart.rx_ready() {
        Some(uart.read())
    } else {
        None
    }
}
```

Common ways to handle `Option`:

```
// unwrap — panics on None (use only when None is impossible)
let b = read_byte(&mut uart).unwrap();

// if let — handle Some, ignore None
if let Some(b) = read_byte(&mut uart) {
    process(b);
}

// unwrap_or — provide a default
let b = read_byte(&mut uart).unwrap_or(0x00);

// match — handle both cases explicitly
match read_byte(&mut uart) {
    Some(b) => process(b),
    None    => handle_empty(),
}
```

Prefer `if let` over `unwrap()` in interrupt handlers — a panic in an ISR is very hard to debug without defmt logging.

# Part I — Foundation

*Days 1–5: toolchain, hardware access layers, and logging.*

# Day 1 — Toolchain Setup and First Flash

## Goal

Install the Rust embedded toolchain, understand why bare-metal Rust looks different from normal Rust, and flash a minimal program onto the nRF52840 DK.

## What you will learn

- Why bare-metal Rust requires `#![no_std]` and `#![no_main]`
- The role of `cortex-m-rt`, the linker script, and `memory.x`
- How to install `probe-rs` and flash a target with `cargo run`
- The full build pipeline from `cargo build` to running firmware on silicon

* * *

## Why Rust for bare-metal?

Rust brings three things to embedded development that C cannot guarantee:

| Property | C | Rust |
|---|---|---|
| Memory safety | By convention | Enforced by compiler |
| Data race freedom | By discipline | Statically impossible (ownership) |
| Undefined behaviour | Common trap | Caught at compile time |

You still have full control over registers, interrupts, and timing – but the compiler catches the class of bugs that cause silent corruption in embedded C.

* * *

## The embedded Rust mental model

On a hosted system (Linux, macOS) your program starts in `main()` because the OS sets up a stack, heap, and standard library before calling `main`. On bare-metal there is no OS. Your program runs directly on reset.

Two attributes mark this change:

```
#![no_std]   // don't link the standard library — it needs an OS
#![no_main]  // don't use the default entry point — we define our own reset
↪  handler
```

Without `std`, you lose `Vec`, `String`, `println!`, threads, and file I/O. You keep `core` (iterators, slices, basic types).

* * *

## Toolchain installation

### 1. Install Rust

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env
```

### 2. Add the ARM Cortex-M4F target

The nRF52840 has a Cortex-M4F core. Its target triple is `thumbv7em-none-eabihf`:

```
rustup target add thumbv7em-none-eabihf
```

- `thumbv7em` — Thumb-2, ARMv7-M with DSP extensions
- `none` — no operating system
- `eabihf` — embedded ABI, hardware floating-point

### 3. Install probe-rs

```
cargo install probe-rs-tools --locked
```

Verify with `probe-rs info --chip nRF52840_xxAA`.

* * *

## Project structure

Every bare-metal Rust project for Cortex-M needs these files:

```
day01_toolchain_setup/
├── .cargo/config.toml   ← default target + cargo run → probe-rs
├── src/main.rs          ← your firmware
├── Cargo.toml           ← dependencies
├── build.rs             ← passes linker flags to cargo
└── memory.x             ← tells the linker where FLASH and RAM are
```

### memory.x

```
MEMORY
{
  FLASH : ORIGIN = 0x00000000, LENGTH = 1M
  RAM   : ORIGIN = 0x20000000, LENGTH = 256K
}
```

### build.rs

```rust
fn main() {
    println!("cargo:rustc-link-arg=-Tlink.x");
    println!("cargo:rerun-if-changed=memory.x");
}
```

### .cargo/config.toml

```
[target.thumbv7em-none-eabihf]
runner = "probe-rs run --chip nRF52840_xxAA"

[build]
target = "thumbv7em-none-eabihf"
```

### Cargo.toml

```
[dependencies]
cortex-m     = { version = "0.7", features = ["critical-section-single-core"] }
cortex-m-rt  = "0.7"
panic-halt   = "0.2"
```

- **cortex-m** – Cortex-M core intrinsics (`asm::nop()`, `asm::wfi()`)
- **cortex-m-rt** – reset handler, `#[entry]` macro, interrupt table
- **panic-halt** – on panic, spin forever

* * *

## The firmware

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

use cortex_m_rt::entry;
use panic_halt as _;

#[entry]
fn main() -> ! {
    loop {
        cortex_m::asm::nop();
    }
}
```

`-> !` means the function never returns (the `!` type, "never"). Bare-metal `main` must never return – if it did, there is nowhere to go.

`#[entry]` marks this function as the reset handler. `cortex-m-rt` generates the vector table with this address in the reset vector slot.

* * *

## Build and flash

```
cd rust_src/day01_toolchain_setup
cargo run --release
```

You should see probe-rs output:

```
  Erasing sectors ✔
  Programming pages ✔
  Finished in 0.42s
```

The CPU is now running your firmware — looping forever, doing nothing. That is exactly what we asked.

* * *

## Practice tasks

1. Change `asm::nop()` to `asm::wfi()` (Wait For Interrupt). What is the difference in power consumption between the two?
2. Add `cortex_m::asm::delay(1_000_000)` inside the loop. What does this call do at the hardware level?
3. Run `cargo size --release` (after `cargo install cargo-binutils`). How large is the binary? Why is it so small?
4. Open the generated `target/thumbv7em-none-eabihf/release/` folder. Which file is the ELF? Which is the raw binary?

## Source code

rust_src/day01_toolchain_setup

# Day 2 — Blinky with the PAC

## Goal

Blink an LED by writing directly to GPIO registers via the Peripheral Access Crate (PAC) – the same way you would in C, but from safe-by-default Rust.

## What you will learn

- What a PAC is and how it is generated from SVD files
- The nRF52840 GPIO register map: PIN_CNF, OUTSET, OUTCLR
- Why PAC register writes require `unsafe`
- nRF52840 DK LED and button pin assignments

* * *

## What is a PAC?

A **Peripheral Access Crate** is an auto-generated Rust crate that maps every register in a microcontroller to a typed accessor. It is generated from an SVD (System View Description) file – the same XML used by Keil and other IDEs to show register views in a debugger.

For the nRF52840 the PAC is `nrf52840-pac`. Every peripheral has a struct, and every register has a method returning a typed accessor:

```
let p = nrf52840_pac::Peripherals::take().unwrap();
let p0 = &p.P0;

// Corresponds directly to the PIN_CNF[13] hardware register
p0.pin_cnf[13].write(|w| w.dir().output());
```

* * *

## nRF52840 DK pin assignments

| Signal | Pin | Notes |
|---|---|---|
| LED1 | P0.13 | Active-low |
| LED2 | P0.14 | Active-low |
| LED3 | P0.15 | Active-low |
| LED4 | P0.16 | Active-low |
| BTN1 | P0.11 | Active-low, needs pull-up |
| BTN2 | P0.12 | Active-low, needs pull-up |
| BTN3 | P0.24 | Active-low, needs pull-up |
| BTN4 | P0.25 | Active-low, needs pull-up |

**Active-low** means: drive LOW to turn the LED on, HIGH to turn it off.

* * *

## Key GPIO registers

| Register | Purpose |
|---|---|
| `PIN_CNF[n]` | Configure direction, pull, drive strength for pin n |
| `OUTSET` | Set specified pins HIGH (atomic — only bits written to 1 change) |
| `OUTCLR` | Set specified pins LOW (atomic) |
| `IN` | Read input state of all 32 pins |

* * *

## The firmware

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

use cortex_m_rt::entry;
use nrf52840_pac as pac;
use panic_halt as _;

const LED1_PIN: u32 = 13;

#[entry]
fn main() -> ! {
    let p = pac::Peripherals::take().unwrap();
    let p0 = &p.P0;

    // Configure P0.13 as push-pull output
    p0.pin_cnf[LED1_PIN as usize].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled()
    });

    // Start with LED off (pin high)
    p0.outset.write(|w| unsafe { w.bits(1 << LED1_PIN) });

    loop {
        p0.outclr.write(|w| unsafe { w.bits(1 << LED1_PIN) }); // LED on
        cortex_m::asm::delay(8_000_000); // ~1 s at 64 MHz

        p0.outset.write(|w| unsafe { w.bits(1 << LED1_PIN) }); // LED off
        cortex_m::asm::delay(8_000_000);
    }
}
```

**Why `unsafe`?** The `bits()` method writes a raw `u32` value. The PAC cannot rule out reserved bit patterns that could damage the peripheral state, so it requires `unsafe`. Day 3 removes this entirely.

* * *

## Practice tasks

1. Modify the program to blink all four LEDs in sequence.

2. Calculate the correct delay count for a 2 Hz blink (250 ms on, 250 ms off) at 64 MHz.
3. Read the `IN` register for a button pin configured as input. What does the idle value tell you about the pull-up state?
4. Try `p0.out.read().pin13().is_high()`. What does it return and why?

## Source code

rust_src/day02_blinky_pac

# Day 3 — Blinky with the HAL

## Goal

Replace raw PAC register writes with the `nrf52840-hal` Hardware Abstraction Layer. Understand the type-state pattern that makes illegal GPIO configurations a compile error.

## What you will learn

- What a HAL is and how it builds on the PAC
- The `embedded-hal` trait ecosystem
- The type-state pattern for GPIO pins
- Why HAL code has no `unsafe` for GPIO

* * *

## What is a HAL?

A **Hardware Abstraction Layer** wraps PAC register access in ergonomic, safe Rust types. `nrf52840-hal` provides:

- GPIO pins as typed structs — a `Pin<Output<PushPull>>` cannot be read like an input
- Drivers for UART, SPI, I2C, Timer, PWM, ADC, and more
- Implementations of `embedded-hal` traits so drivers are portable across chips

The `embedded-hal` project defines common traits (`OutputPin`, `InputPin`, `DelayMs`, `SpiDevice`). Any HAL that implements these traits can use any driver crate that depends on them.

* * *

## The type-state pattern

In Day 2 you could accidentally write to OUTSET for a pin configured as input. The HAL prevents this at the type level:

```
// Type is now P0_13<Output<PushPull>> — not a generic pin
let mut led1 = port0.p0_13.into_push_pull_output(Level::High);

led1.set_low().unwrap();   // ✅ compiles — Output pins have set_low
// led1.is_low().unwrap(); // ❌ compile error — is_low only on Input pins
```

`into_push_pull_output()` consumes the pin and returns a new type. The original `p0_13` binding is gone. This is zero-cost — the types only exist at compile time.

* * *

## GPIO API

```
use nrf52840_hal::{gpio::{p0::Parts, Level}, pac, prelude::*};

let p = pac::Peripherals::take().unwrap();
let port0 = Parts::new(p.P0);

let mut led1 = port0.p0_13.into_push_pull_output(Level::High);
let btn1     = port0.p0_11.into_pullup_input();

led1.set_low().unwrap();        // LED on
led1.set_high().unwrap();       // LED off
btn1.is_low().unwrap();         // true when pressed
```

* * *

## HAL vs PAC: the same binary, better code

| Task | PAC (Day 2) | HAL (Day 3) |
|---|---|---|
| Configure output | `pin_cnf[13].write(...)` | `p0_13.into_push_pull_output(...)` |
| Drive low | ‘outclr.write( | w |
| Wrong direction | Silent bug | Compile error |

| Task | PAC (Day 2) | HAL (Day 3) |
|---|---|---|
| `unsafe` needed | Yes | No |

Both compile to identical machine code. The difference is entirely in what the compiler will and will not allow you to write.

* * *

## The firmware

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

use cortex_m_rt::entry;
use nrf52840_hal::{gpio::{p0::Parts, Level}, pac, prelude::*};
use panic_halt as _;

#[entry]
fn main() -> ! {
    let p = pac::Peripherals::take().unwrap();
    let port0 = Parts::new(p.P0);

    let mut led1 = port0.p0_13.into_push_pull_output(Level::High);

    loop {
        led1.set_low().unwrap();
        cortex_m::asm::delay(8_000_000);

        led1.set_high().unwrap();
        cortex_m::asm::delay(8_000_000);
    }
}
```

* * *

## Erased pins

To store multiple pins of different numbers in an array, use `.degrade()` to erase the pin number from the type:

```
let led1 = port0.p0_13.into_push_pull_output(Level::High).degrade();
let led2 = port0.p0_14.into_push_pull_output(Level::High).degrade();

// Now both are Pin<Output<PushPull>> — can be stored together
let mut leds = [led1, led2];
```

You will need `.degrade()` in Day 7 when sharing a pin with an interrupt handler.

* * *

## Practice tasks

1. Store all four LEDs as `[Pin<Output<PushPull>>; 4]` and iterate through them in a chase pattern.
2. Attempt to call `led1.is_low()` on an output pin. Read the compiler error.
3. Add a `into_pullup_input()` button. Toggle LED2 only while BTN1 is held.
4. Compare the disassembly of Day 2 and Day 3 with `cargo objdump`. Are they identical?

## Source code

rust_src/day03_blinky_hal

# Day 4 — SysTick and Real Delays

## Goal

Replace `asm::delay()` with the SysTick timer and produce accurate millisecond delays. Understand nRF52840 clock sources and why you must start the HFXO for precise timing.

## What you will learn

- nRF52840 clock architecture: HFCLK, LFCLK, HFINT (RC), HFXO (crystal)
- How the Cortex-M SysTick timer works
- Using `nrf52840-hal`'s `Delay` driver
- The `embedded-hal` `DelayMs` trait

* * *

## nRF52840 clock sources

The CPU and most peripherals run from HFCLK (64 MHz). It can come from two sources:

| Source | Accuracy | Default? | Notes |
|---|---|---|---|
| HFINT (RC osc.) | ±2% | Yes | Active immediately at reset |
| HFXO (32 MHz crystal) | ±20 ppm | No | Start with `enable_ext_hfosc()` |

The DK has an onboard 32 MHz crystal. For timing-critical work, start HFXO:

```rust
use nrf52840_hal::clocks::Clocks;

// Blocks ~360 µs until the crystal is stable
let _clocks = Clocks::new(p.CLOCK).enable_ext_hfosc();
```

Keep `_clocks` alive for the entire program – dropping it stops the oscillator.

* * *

## SysTick timer

SysTick is a 24-bit down-counter in every Cortex-M core. At 64 MHz with a reload value of 64,000, it fires every 1 ms (1 kHz tick).

`nrf52840-hal`'s `Delay` driver programs SysTick and busy-waits on the count flag for you:

```rust
use nrf52840_hal::{delay::Delay, pac};
use embedded_hal::blocking::delay::DelayMs; // embedded-hal 0.2 API

let core = pac::CorePeripherals::take().unwrap();
let mut delay = Delay::new(core.SYST);

delay.delay_ms(500u32);  // block for exactly 500 ms
delay.delay_us(100u32);  // block for exactly 100 µs
```

`CorePeripherals::take()` returns Cortex-M core peripherals (SYST, NVIC, SCB). This is separate from `Peripherals::take()` which returns nRF52840 vendor peripherals.

* * *

## Two-LED timing example

```rust
let mut tick = 0u32;

loop {
    if tick % 2 == 0 { led1.set_low().unwrap(); }
    else             { led1.set_high().unwrap(); }

    if (tick / 5) % 2 == 0 { led2.set_low().unwrap(); }
    else                   { led2.set_high().unwrap(); }

    delay.delay_ms(200u32);
    tick = tick.wrapping_add(1);
}
```

LED1 toggles every 200 ms; LED2 toggles every 1000 ms (every 5th 200 ms tick).

* * *

## Practice tasks

1. Measure the actual delay with a logic analyser. Compare HFINT vs HFXO.
2. What happens if you `let _ = Clocks::new(p.CLOCK).enable_ext_hfosc()` (dropping immediately instead of binding to `_clocks`)?
3. Create a heartbeat pattern: 100 ms on, 100 ms off, 100 ms on, 700 ms off.
4. Read `CLOCK.HFCLKSTAT` via the PAC after `enable_ext_hfosc()`. Confirm the `SRC` field reads `Xtal`.

## Source code

rust_src/day04_systick_delays

# Day 5 — Logging with defmt and RTT

## Goal

Add logging to your firmware using `defmt` over RTT. Replace `panic-halt` with `panic-probe` to get useful backtraces on panics.

## What you will learn

- Why `println!` does not exist in `no_std` and how `defmt` replaces it
- How RTT transmits log data through the debug probe with near-zero overhead
- Log levels: `trace!`, `debug!`, `info!`, `warn!`, `error!`
- How `panic-probe` provides a stack trace on panic

* * *

## How defmt works

`defmt` (deferred formatting) keeps formatting logic on the host PC, not the target chip:

1. **Target side:** `info!("count = {}", count)` stores a *string index* and the raw bytes of `count`. No string is constructed on the chip.
2. **Transport:** Raw bytes travel over RTT (a shared memory ring buffer) through the debug probe.
3. **Host side:** `probe-rs` reads the bytes, looks up the string in the ELF's `.defmt` section, and formats the final message on your PC.

This makes logging zero-allocation, fast (3–10 instructions per call), and rich (full format strings, enums, custom types).

* * *

## RTT channel

RTT uses a small shared memory buffer (default 1 KB) between the MCU and the probe. The probe polls it continuously — no CPU interrupts, no UART pins.

```
MCU RAM                Debug Probe (J-Link / CMSIS-DAP)
┌────────────┐         ┌──────────────────────────────────┐
│ RTT buffer │ ←─────→ │ probe-rs reads and formats output │──→ terminal
└────────────┘         └──────────────────────────────────┘
```

* * *

## Setup

Add to `Cargo.toml`:

```
defmt        = "0.3"
defmt-rtt    = "0.4"
panic-probe  = { version = "0.3", features = ["print-defmt"] }
```

Remove `panic-halt`. Then in `main.rs`:

```
use defmt::{debug, error, info, warn};
use defmt_rtt as _;   // registers the RTT transport
use panic_probe as _;  // registers the panic handler
```

* * *

## Log levels

```
defmt::trace!("very verbose");          // only visible at DEFMT_LOG=trace
defmt::debug!("debug: value={}", v);
defmt::info!("system started");
defmt::warn!("unexpected state");
defmt::error!("fatal error code={}", e);
```

Filter with the environment variable (set in `.cargo/config.toml`):

```
[env]
DEFMT_LOG = "debug"
```

* * *

## Running

```
cargo run --release
```

Output streams to your terminal:

```
INFO  Day 5: defmt + RTT logging demo
INFO  LED on  — blink #0
DEBUG LED off — blink #0
WARN  Reached 5 blinks — still going
```

* * *

## Practice tasks

1. Add `panic!("test panic")` after 3 blinks. Does probe-probe print a file and line number? (Note: `defmt::panic!` is also valid — it sends the message via defmt before panicking, which is slightly more informative over RTT.)
2. Derive `defmt::Format` for a custom struct and log it:

```
#[derive(defmt::Format)]
struct Reading { x: i16, y: i16, z: i16 }
info!("{:?}", Reading { x: 100, y: -50, z: 980 });
```

3. Set `DEFMT_LOG=error` and rebuild. Verify that `info!` calls disappear from the binary (`cargo size` should report a smaller `.text` section).
4. Compare binary size with `panic-halt` versus `panic-probe`.

## Source code

rust_src/day05_defmt_logging

# Part II — Hardware Interaction

*Days 6–9: buttons, interrupts, UART, and timers.*

# Day 6 — GPIO Input and Buttons

## Goal

Read the four buttons on the nRF52840 DK using the `InputPin` trait, with proper pull-up configuration, active-low logic, and software debouncing.

## What you will learn

- `InputPin` trait from `embedded-hal`
- How internal pull-up resistors work and when to enable them
- Active-low button logic
- Software debouncing
- Why polling has a latency floor

* * *

## Active-low button wiring

Each button on the DK is wired between the GPIO pin and GND, with no external resistor. The internal pull-up must be enabled:

```
VDD ── [~13 kΩ internal pull-up] ── P0.11 ── [BTN1] ── GND
```

- **Released:** pin pulled HIGH through pull-up → `is_low()` = false
- **Pressed:** pin shorted to GND → `is_low()` = true

* * *

## Reading button state

```rust
use embedded_hal::digital::v2::InputPin;

let btn1 = port0.p0_11.into_pullup_input();

if btn1.is_low().unwrap() {
    // button is pressed
}
```

* * *

## Polling with debounce

Mechanical buttons bounce (rapid make/break contact) for 1–10 ms when pressed. A small delay between samples eliminates most bounce artifacts:

```rust
loop {
    if btn1.is_low().unwrap() { led1.set_low().unwrap(); }
    else                      { led1.set_high().unwrap(); }

    cortex_m::asm::delay(100_000); // ~1.5 ms gap
}
```

* * *

## Polling vs interrupts

| Property | Polling (Day 6) | Interrupt (Day 7) |
|---|---|---|
| Latency | Up to 1 loop period | < 1 µs |
| CPU usage | Runs continuously | Only on event |
| Code complexity | Simple | Shared state, critical sections |

* * *

## Practice tasks

1. Implement edge detection: log “pressed” only on HIGH→LOW transitions, not continuously while held.
2. Count button presses and display the count on the four LEDs as a 4-bit binary number (LED1 = bit 0).
3. Implement long-press detection: if BTN1 is held > 2 s, trigger a different action than a short press.
4. Use `embedded_hal::digital::v2::ToggleableOutputPin` (path in embedded-hal 0.2, not a sub-module) to toggle LED state without manually tracking it.

## Source code

rust_src/day06_gpio_input

# Day 7 — GPIOTE Interrupts

## Goal

Replace the polling loop with a true hardware interrupt using the GPIOTE peripheral. Learn the `Mutex<RefCell<Option<T>>>` pattern for sharing state between main and an ISR.

## What you will learn

- The GPIOTE peripheral and its 8-channel model
- Cortex-M interrupt handlers in Rust with `cortex-m-rt`
- Sharing data between main and an ISR safely
- `Mutex`, `RefCell`, and critical sections in `no_std`

* * *

## GPIOTE peripheral

GPIOTE (GPIO Tasks and Events) routes GPIO signal edges to the NVIC. Each of its 8 channels can:

- Detect rising edge, falling edge, or any change on a GPIO pin
- Fire an interrupt to the CPU
- Drive an output pin via PPI without CPU involvement

```
BTN1 (P0.11) ── falling edge ──→ GPIOTE ch.0 ──→ NVIC ──→ GPIOTE ISR
```

* * *

## Configuring a channel

```
use nrf52840_hal::gpiote::Gpiote;

let gpiote = Gpiote::new(p.GPIOTE);
let btn1 = port0.p0_11.into_pullup_input().degrade();

gpiote
    .channel0()
    .input_pin(&btn1)    // watch this pin (requires degraded pin)
    .hi_to_lo()          // trigger on falling edge
    .enable_interrupt();
```

* * *

## The shared state problem

Interrupt handlers are top-level functions that cannot borrow from `main`. Shared state must be in a `static`. The safe pattern:

```
use core::cell::RefCell;
use cortex_m::interrupt::Mutex;

static LED: Mutex<RefCell<Option<Pin<Output<PushPull>>>>> =
    Mutex::new(RefCell::new(None));
```

| Layer | Role |
|---|---|
| `Option<T>` | Starts as `None`; filled before enabling interrupt |
| `RefCell<T>` | Interior mutability — `&mut T` from `&T` |
| `Mutex<T>` | Cortex-M critical section (disables interrupts while accessed) |

* * *

## Filling the static and enabling the interrupt

```
cortex_m::interrupt::free(|cs| {
    GPIOTE.borrow(cs).replace(Some(gpiote));
    LED.borrow(cs).replace(Some(led1));
});

unsafe { pac::NVIC::unmask(pac::Interrupt::GPIOTE); }
```

The critical section token `cs` is proof that interrupts are disabled – you cannot call `borrow(cs)` without it.

* * *

## The interrupt handler

```
#[interrupt]
fn GPIOTE() {
    cortex_m::interrupt::free(|cs| {
        if let Some(gpiote) = GPIOTE.borrow(cs).borrow().as_ref() {
            if gpiote.channel0().is_event_triggered() {
                gpiote.channel0().reset_events(); // ALWAYS clear the event flag

                if let Some(led) = LED.borrow(cs).borrow_mut().as_mut() {
                    // toggle LED
                }
            }
        }
    });
}
```

**Always call `reset_events()` inside the ISR.** If you don't, the interrupt fires again immediately after returning, looping forever.

* * *

## main after setup

```
loop {
    cortex_m::asm::wfi(); // sleep until next interrupt
}
```

`wfi` (Wait For Interrupt) puts the CPU into a low-power sleep. It wakes on any unmasked interrupt.

* * *

## Practice tasks

1. Add GPIOTE channels for all four buttons, each toggling its LED.
2. Measure interrupt latency with an oscilloscope (toggle a second pin at the start of the ISR; measure time from button press edge to pin change).
3. Count ISR invocations and log the count every 10th press.
4. Deliberately forget `reset_events()`. Observe what happens.

## Source code

rust_src/day07_gpiote_interrupts

# Day 8 — UART Serial Communication

## Goal

Configure UARTE0 to send and receive bytes over the USB-to-UART bridge on the nRF52840 DK. Implement a serial echo loop.

## What you will learn

- The difference between UART (legacy) and UARTE (DMA) on nRF52840
- TX/RX pin assignments on the DK's J-Link OB virtual COM port
- Using `nrf52840-hal`'s `Uarte` driver
- Why DMA buffers must live in static RAM

* * *

## UART vs UARTE

| Peripheral | DMA | Notes |
|---|---|---|
| UART (legacy) | No | Deprecated; byte-at-a-time |
| UARTE0, UARTE1 | Yes (EasyDMA) | Preferred; block transfers |

UARTE uses EasyDMA — the peripheral accesses RAM directly without the CPU. **DMA buffers must be in static RAM, not on the stack.**

* * *

## DK UART pin assignments

| Signal | Pin | Notes |
|---|---|---|
| TXD (MCU → host) | P0.06 | |
| RXD (host → MCU) | P0.08 | |

Connect with any terminal at **115200 baud, 8N1**. On Linux, the port appears as `/dev/ttyACM0` or `/dev/ttyUSB0`.

* * *

## Driver setup

```
use nrf52840_hal::uarte::{self, Baudrate, Parity, Uarte};

let txd = port0.p0_06.into_push_pull_output(Level::High).degrade();
let rxd = port0.p0_08.into_floating_input().degrade();

let mut serial = Uarte::new(
    p.UARTE0,
    uarte::Pins { txd, rxd, cts: None, rts: None },
    Parity::EXCLUDED,
    Baudrate::BAUD115200,
);
```

* * *

## Static buffers

```rust
static mut RX_BUF: [u8; 1]  = [0u8; 1];
static mut TX_BUF: [u8; 64] = [0u8; 64];

// In main:
serial.write(b"Ready.\r\n> ").unwrap();
loop {
    let buf = unsafe { &mut RX_BUF };
    serial.read(buf).unwrap();          // blocks until 1 byte received

    let out = unsafe { &mut TX_BUF };
    out[0] = buf[0];
    serial.write(&out[..1]).unwrap();   // echo back

    if buf[0] == b'\r' {
        serial.write(b"\n> ").unwrap();
    }
}
```

* * *

## Practice tasks

1. Convert received lowercase letters to uppercase before echoing.
2. Buffer a full line (terminated by `\r`) and print it all at once.
3. Parse simple commands: `"on"` □ LED on, `"off"` □ LED off.
4. Write a `write_decimal(serial, n: u32)` function that sends digits without using `format!`.

## Source code

rust_src/day08_uart_serial

# Day 9 — Hardware Timers

## Goal

Use the nRF52840's Timer peripheral to drive precise periodic events without blocking the CPU with `delay_ms`.

## What you will learn

- nRF52840 Timer0–Timer4: 1 MHz base clock, 32-bit counter
- The `CountDown` trait from `embedded-hal`
- The `nb::block!` macro for blocking on non-blocking operations
- Running two independent timer periods

* * *

## Timer peripheral

The nRF52840 has five independent Timer instances. In `nrf52840-hal`, the driver sets the prescaler to divide 16 MHz down to 1 MHz:

```
16 MHz HFCLK → [÷16 prescaler] → 1 MHz → 32-bit counter (max ~71 min)
```

1 count = 1 µs.

* * *

## CountDown trait

```rust
// Start a 500 ms one-shot (500,000 µs)
timer0.start(500_000u32);

// Spin until expired — nb::block! converts WouldBlock into a spin loop
nb::block!(timer0.wait()).unwrap();

// Must call start() again for the next period (no auto-reload)
```

* * *

## Two-LED example

```rust
let mut timer0 = Timer::new(p.TIMER0);
let mut led2_tick = 0u32;

loop {
    timer0.start(250_000u32);
    nb::block!(timer0.wait()).unwrap();

    // LED1 every 250 ms tick
    led1_state = !led1_state;
    if led1_state { led1.set_low().unwrap(); } else { led1.set_high().unwrap();
↪   }

    // LED2 every 750 ms (3 × 250 ms)
    led2_tick += 1;
    if led2_tick >= 3 {
        led2_tick = 0;
        led2_state = !led2_state;
        if led2_state { led2.set_low().unwrap(); } else {
↪   led2.set_high().unwrap(); }
    }
}
```

`nb::block!` spins the CPU. For truly independent timers that do not block each other, use timer interrupts or the RTIC framework (Day 10).

* * *

## Timer vs Delay

| Property | `Delay` (SysTick) | `Timer` (TIMER0–4) |
|---|---|---|
| Resolution | 1 CPU cycle | 1 µs |
| Maximum period | ~262 ms single tick | ~71 minutes |
| Instances | 1 | 5 |
| Interrupt capable | Yes | Yes |

* * *

## Practice tasks

1. Implement a 1 Hz heartbeat on LED1 using TIMER0 and a 3 Hz pulse on LED2 using TIMER1 with interrupt handlers.
2. Stopwatch: count 10 ms ticks between BTN1 press and release. Log elapsed ms.
3. Use TIMER0's capture feature to measure the duration of a button press.
4. What is the maximum single delay from SysTick's 24-bit counter at 64 MHz?

## Source code

rust_src/day09_hardware_timers

# Part III — Concurrency

*Day 10: the RTIC framework.*

# Day 10 — RTIC: Safe Concurrency

## Goal

Replace the `Mutex<RefCell<Option<T>>>` pattern from Day 7 with RTIC (Real-Time Interrupt-driven Concurrency). Understand tasks, resources, priorities, and compile-time data-race freedom.

## What you will learn

- What RTIC is and how it differs from an RTOS
- Hardware tasks, software tasks, and `idle`
- Shared and local resources
- Priority-ceiling protocol via `.lock()`
- How RTIC verifies resource safety at compile time

* * *

## What is RTIC?

RTIC is a **task framework** for Cortex-M, not a full RTOS. It provides:

- **Hardware tasks** — ISRs with typed, owned resources
- **Software tasks** — dispatched via software-pending interrupts, with priorities
- **Static resource analysis** — the compiler proves no data races exist

If your RTIC program compiles, there are no data races. Full stop.

* * *

## App structure

```rust
#[rtic::app(device = nrf52840_hal::pac, peripherals = true,
            dispatchers = [SWI0_EGU0])]
mod app {
    #[shared]
    struct Shared { press_count: u32 }

    #[local]
    struct Local { gpiote: Gpiote, led1: Pin<Output<PushPull>>,
                   led1_state: bool, led2: Pin<Output<PushPull>>,
                   _btn1: Pin<Input<PullUp>> }

    #[init]
    fn init(cx: init::Context) -> (Shared, Local, init::Monotonics) {
        // set up hardware, return initial resource values
    }

    #[idle(local = [led2])]
    fn idle(cx: idle::Context) -> ! {
        loop { cortex_m::asm::wfi(); }
    }

    #[task(binds = GPIOTE, local = [gpiote, led1, led1_state: bool = false],
           shared = [press_count])]
    fn on_button(mut cx: on_button::Context) {
        cx.local.gpiote.channel0().reset_events();

        *cx.local.led1_state = !*cx.local.led1_state;
        if *cx.local.led1_state { cx.local.led1.set_low().unwrap(); }
        else                    { cx.local.led1.set_high().unwrap(); }

        cx.shared.press_count.lock(|count| {
            *count += 1;
            defmt::info!("Press #{}", *count);
        });
    }
}
```

* * *

## Resource types

| Annotation | Access | Sharing |
|---|---|---|
| `#[local]` | Only this task | No – exclusive ownership |
| `#[shared]` | Multiple tasks | Yes – `.lock()` required by lower-priority task |

`#[local]` resources with `= value` initializers are stored in a `static` generated by RTIC – no manual `static mut` needed.

* * *

## The `.lock()` API

When tasks at different priorities share a resource, the lower-priority task calls `.lock()`:

```
cx.shared.press_count.lock(|count| {
    *count += 1; // `count` is &mut u32 here
});
```

`.lock()` raises `BASEPRI` to the ceiling priority – the highest priority of any task sharing this resource – preventing preemption for the duration. This is the **Priority Ceiling Protocol**, proven correct by the RTIC analysis pass.

* * *

## RTIC vs Day 7 manual pattern

| Aspect | Day 7 | Day 10 RTIC |
|---|---|---|
| Shared state declaration | `static Mutex<RefCell<Option<T>>>` | `#[shared] struct Shared { field: T }` |
| Critical section | `interrupt::free(\|cs\| { ... })` | `lock(\|v\| { ... })` |

| Aspect | Day 7 | Day 10 RTIC |
|---|---|---|
| Init order | Manual (move into static, unmask) | `#[init]` return, RTIC handles the rest |
| Priority analysis | None | Compile-time verified |

* * *

## Practice tasks

1. Add a TIMER0 hardware task at priority 2. Share a `blink_count` with the GPIOTE task. Verify TIMER0 preempts the idle blink loop.
2. Add a software task dispatched from the GPIOTE task every 5th press.
3. Try to access a `#[local]` resource from two different tasks. Study the compile error.
4. Change GPIOTE to priority 2 and add a priority 1 software task. With a logic analyser, verify GPIOTE preempts it.

## Source code

rust_src/day10_rtic_concurrency

# Appendix A — Quick Reference

## nRF52840 DK pin map (key signals)

| Pin | Signal | Direction | Notes |
|---|---|---|---|
| P0.13 | LED1 | Output | Active-low |
| P0.14 | LED2 | Output | Active-low |
| P0.15 | LED3 | Output | Active-low |
| P0.16 | LED4 | Output | Active-low |
| P0.11 | BTN1 | Input | Active-low, needs pull-up |
| P0.12 | BTN2 | Input | Active-low, needs pull-up |
| P0.24 | BTN3 | Input | Active-low, needs pull-up |
| P0.25 | BTN4 | Input | Active-low, needs pull-up |
| P0.06 | UART TXD | Output | J-Link OB virtual COM |
| P0.08 | UART RXD | Input | J-Link OB virtual COM |
| P0.26 | I2C SDA | I/O | nRF52840 DK default |
| P0.27 | I2C SCL | Output | nRF52840 DK default |

## memory.x template

```
MEMORY
{
  FLASH : ORIGIN = 0x00000000, LENGTH = 1M
  RAM   : ORIGIN = 0x20000000, LENGTH = 256K
}
```

## .cargo/config.toml template

```
[target.thumbv7em-none-eabihf]
runner = "probe-rs run --chip nRF52840_xxAA"

[build]
target = "thumbv7em-none-eabihf"

[env]
DEFMT_LOG = "debug"
```

## Crate version summary

| Crate | Version | Purpose |
|---|---|---|
| `cortex-m` | 0.7 | Core intrinsics (nop, wfi, delay) |
| `cortex-m-rt` | 0.7 | Reset handler, `#[entry]`, interrupt table |
| `nrf52840-pac` | 0.12 | PAC — direct register access |
| `nrf52840-hal` | 0.16 | HAL — safe GPIO, UART, I2C, SPI, Timer |
| `embedded-hal` | 0.2 | Trait definitions (used by nrf52840-hal 0.16) |
| `defmt` | 0.3 | Efficient target-side logging |
| `defmt-rtt` | 0.4 | RTT transport for defmt |
| `panic-probe` | 0.3 | Panic handler with backtrace over RTT |
| `panic-halt` | 0.2 | Simple panic handler (spin forever) |
| `nb` | 1.0 | Non-blocking result type, `block!` macro |
| `rtic` | 1.1 | Real-Time Interrupt-driven Concurrency framework |

## DEFMT_LOG levels

| Value | Shows |
|---|---|
| `trace` | Everything |
| `debug` | debug, info, warn, error |
| `info` | info, warn, error (recommended default) |
| `warn` | warn, error |
| `error` | Errors only (smallest binary) |

## Common Cortex-M intrinsics

```
cortex_m::asm::nop();           // no-operation (1 cycle)
cortex_m::asm::wfi();           // sleep until interrupt
cortex_m::asm::wfe();           // sleep until event
cortex_m::asm::delay(n);        // busy-loop n cycles
cortex_m::asm::dsb();           // data synchronisation barrier
cortex_m::interrupt::free(|_| { /* critical section */ });
```

# Appendix B — Glossary

**Active-low:** A signal that is asserted (active) when the electrical level is LOW (0 V). LEDs and buttons on the nRF52840 DK are active-low.

**BASEPRI:** A Cortex-M register that masks (blocks) all interrupts at or below a given priority number. RTIC uses it for priority-ceiling locking without disabling all interrupts.

**cortex-m-rt:** A Rust crate that provides the Cortex-M runtime: reset handler, vector table, stack pointer initialization, and the `#[entry]` macro.

**EasyDMA:** Nordic's name for their peripheral DMA engine. Used by UARTE, SPI, and TWI (I2C). Transfers happen directly between peripheral and RAM without CPU instruction cycles.

**GPIOTE:** GPIO Tasks and Events. An nRF52840 peripheral that routes GPIO edge events (rising, falling, any change) to the NVIC interrupt controller and to PPI.

**HAL:** Hardware Abstraction Layer. A crate that wraps PAC register access in safe, ergonomic Rust types. `nrf52840-hal` is the HAL for the nRF52840.

**HFXO:** High-Frequency Crystal Oscillator. The external 32 MHz crystal on the nRF52840 DK. Provides much more accurate timing than the internal RC oscillator (HFINT).

**NVIC:** Nested Vectored Interrupt Controller. The Cortex-M hardware block that manages interrupt priorities, enables/disables interrupts, and dispatches to interrupt handlers.

**PAC:** Peripheral Access Crate. An auto-generated Rust crate (from SVD files) that provides typed access to every hardware register in a microcontroller.

**PPI:** Programmable Peripheral Interconnect. An nRF52840 hardware bus that connects peripheral events to peripheral tasks without CPU involvement.

**RTIC:** Real-Time Interrupt-driven Concurrency. A Rust task framework that models firmware as interrupt-driven tasks with statically-verified shared resource access.

**RTT:** Real-Time Transfer. A Segger protocol that uses a shared-memory ring buffer to stream debug data between the target MCU and a host PC through the debug probe connection.

**SysTick:** A 24-bit down-counter built into every Cortex-M processor. Used by `nrf52840-hal`'s `Delay` driver for millisecond delays.

**Type-state pattern:** A Rust design pattern where the type system encodes the state of a value. A `Pin<Output<PushPull>>` and a `Pin<Input<PullUp>>` are different types — operations only valid for one are compile errors for the other.

**`#![no_main]`:** Attribute that disables Rust's default `main` function entry point. Required for bare-metal firmware where a custom reset handler (from `cortex-m-rt`) is the real entry point.

**`#![no_std]`:** Attribute that disables linking the Rust standard library (`std`). Required for bare-metal targets where no OS exists to provide heap allocation, file I/O, or threads.

## What's Next

You have covered the ten foundational topics of bare-metal Rust on the nRF52840. Here is where the journey continues:

### Days 11–20 (planned)

| Day | Topic |
|---|---|
| 11 | I2C – reading a KXTJ3 accelerometer |
| 12 | SPI – external flash memory |
| 13 | ADC – battery monitoring and analog sensors |
| 14 | PWM – motor control and LED brightness |
| 15 | DMA – background transfers without CPU |
| 16 | BLE advertising with `nrf-softdevice` |
| 17 | BLE GATT service – exposing sensor data |
| 18 | NVS flash storage – persisting state across resets |
| 19 | Low-power modes – System ON sleep and System OFF |
| 20 | Bootloader and OTA updates with MCUboot |

### Capstone: Wireless Sensor Node

Combine I2C sensor reading, BLE GATT notifications, NVS storage, and System OFF sleep into a battery-powered wireless sensor node – the same architecture used in commercial IoT products.

### Further reading

- *The Embedded Rust Book* – official guide at docs.rust-embedded.org
- *The Discovery Book* – hands-on intro to embedded Rust
- *RTIC Book* – rtic.rs/stable/book
- Nordic Infocenter – infocenter.nordicsemi.com (nRF52840 Product Specification)
- Knurling Sessions – knurling.io (defmt and probe-rs tutorials)

* * *

*Thank you for reading. If this book helped you, consider leaving a review on LeanPub — it helps other engineers find it.*

## About the Author

**Jovin** is an embedded systems engineer with experience building firmware for IoT devices, wearables, and industrial sensors. He began writing this book to document the learning path he wished had existed when he first picked up Rust for embedded development.

Questions and corrections are welcome. The source code for all examples is available in the companion repository.