

17-350 / 17-650

Lecture 14: Interrupts and Concurrency on Bare Metal

Jonathan Aldrich

Oct 22, 2026

Embedded Systems in Rust

Date	Lecture	HW5
Oct 8 (remote)	Bare-metal Rust, hardware abstraction, typestate	Part 1 foundations
Oct 20	Drivers, <code>embedded-hal</code> , DMA, live on the board	Parts 0–1
Oct 22 (today)	Interrupts and concurrency on bare metal	Part 2, and Part 3's measurements

- HW5 is due **Nov 3**, with a live checkoff Nov 3–6
- After today you have everything for Part 2b: interrupt handlers, shared state, the watchdog

Lecture Outline

- Concurrency with one core
- Interrupts
- Sharing state the wrong way
- Sharing state the right way
- Frameworks: RTIC and Embassy
- Failing safely
- Measuring

micro:bit Has One Core & No Threads. No Data Races?

The micro:bit has **one** CPU core, and our programs have no threads.

Question: can two pieces of our code still race on the same memory?

Think about the button handler from Tuesday's HW5 tour, and the main loop that sends the log.

Lecture Outline

- Concurrency with one core
- **Interrupts**
 - what happens when one fires
 - priorities and preemption
 - handlers take no arguments
 - clear the event; buttons bounce
- Sharing state the wrong way
- Sharing state the right way
- Frameworks: RTIC and Embassy
- Failing safely
- Measuring

An Interrupt Is a Thread the Hardware Schedules

1. A peripheral raises an **event** whose interrupt is enabled (Tuesday: `INTENSET`)
2. The **NVIC** (the interrupt controller, ARM's core peripheral) decides it should run
3. The CPU **stops between two instructions**, saves registers to the stack, and jumps to the handler named in the **vector table** (Lecture 12)
4. The handler returns; the CPU restores registers and carries on as if nothing happened

So: the main loop can be preempted between *any* two instructions. You can think of an interrupt as a thread that the hardware schedules---one that has priority over normal execution.

The NVIC: Enable, Pend, Prioritize

```
unsafe { NVIC::unmask(Interrupt::TIMER0) };    // allow it (unsafe)
NVIC::pend(Interrupt::TIMER0);                 // make it pending from software
NVIC::mask(Interrupt::TIMER0);                 // block it
```

- unmasking an interrupt is **unsafe** because it introduces new concurrency
- **Priorities**: each interrupt has one; a **lower number is more urgent**. The nRF52833 has 8 levels
- **Preemption**: a higher-priority handler can interrupt a lower one. Handlers at the **same priority never preempt each other**; they queue
- Remember that last rule: it's what RTIC builds on

Handlers Take No Arguments

```
#[interrupt]
fn TIMER0() {                // called by hardware: no parameters, no return value
    // whatever this touches must be reachable from a `static`
}
```

- `#[interrupt]` (from the PAC) puts the function in the vector table under its name
- No caller means **no arguments**: the handler can't borrow anything from `main`
- So shared state lives in **statics**, and the rest of today is about how to make that safe

Clear the Event, or It Fires Forever

```
#[interrupt]
fn TIMER0() {
    timer.reset_event();    // EVENTS_COMPARE[0] = 0
    // ... the work ...
}
```

- An interrupt fires while its event is set and enabled. Return without clearing it, and the handler runs again **immediately**, forever
- Tuesday's RNG demo, where the uncleared event made the "wait" return at once, is the same bug. As an interrupt, it locks up the main loop
- In HW5 this is `Timer::take_tick` and `Gpiote::take_event`

Buttons Bounce

```
#[interrupt]
fn GPIOTE() {
    let now = cycles(); // the DWT cycle counter, 64 MHz
    /* reset the event, not shown... */
    EDGES.fetch_add(1, Relaxed);
    if now.wrapping_sub(LAST_ACCEPTED.load(Relaxed)) >= DEBOUNCE_CYCLES { // 20 ms
        LAST_ACCEPTED.store(now, Relaxed);
        PRESSES.fetch_add(1, Relaxed);
    }
}
```

```
edges    2    presses    2
edges    3    presses    2    <- the same press, bouncing
...
edges   14    presses   10    <- ten presses: 4 of them bounced
```

- One press can close and reopen the contacts several times in a few milliseconds
- **Debounce by time:** ignore edges too soon after the last accepted one. Decide using a clock, not with a busy-wait in the handler

[Live demo] demos/board: cargo run --release --bin button_bounce

Lecture Outline

- Concurrency with one core
- Interrupts
- **Sharing state the wrong way**
 - `static mut`, and the torn read-modify-write
 - what edition 2024 says
- Sharing state the right way
- Frameworks: RTIC and Embassy
- Failing safely
- Measuring

A Classic Interrupt Bug in C

```
volatile uint32_t count;           /* shared with the interrupt */  
  
void TIMER0_IRQHandler(void) { count++; }  
  
int main(void) {  
    for (int i = 0; i < 1000000; i++) count++;  
}
```

- `volatile` ensures that every load and store happens (Lecture 12)
- It does **not** make `count++` one step: it's a load, an add, and a store
- An interrupt between the load and the store increments `count`. Then main stores its stale value + 1, and the interrupt's increment is gone

The Same Bug in Rust: `static mut`

```
static mut COUNT: u32 = 0;
// main, 1,000,000 times; the TIMER0 handler, 20,000 times a second:
unsafe { let p = addr_of_mut!(COUNT); write_volatile(p, read_volatile(p) + 1); }
```

```
3c2:  ldr    r5, [r2]          @ load COUNT          <- an interrupt here ...
3c4:  subs   r3, #1             @ (the loop counter)
3c6:  add.w  r5, r5, #1         @ COUNT + 1
3ca:  str    r5, [r2]          @ store COUNT          <- ... is overwritten here
```

```
main incremented 1000000 times, the handler 3539 times
COUNT = 1001770, expected 1003539: 1769 increments lost
```

Half of the handler's increments are gone.

[Live demo] `demos/board: cargo run --release --bin counter_racy`. The listing is main's loop, from `llvm-objdump`.

Rust Edition 2024 Warns on Reference Creation

```
let r = unsafe { &mut COUNT };
```

```
error: creating a mutable reference to mutable static
```

```
|         let r = unsafe { &mut COUNT };
```

```
|         ^^^^^^^^^^^^^ mutable reference to mutable static
```

```
= note: mutable references to mutable statics are dangerous; it's undefined behavior  
       if any other pointer to the static is used or if any other reference is  
       created for the static while the mutable reference lives
```

```
= note: `#[deny(static_mut_refs)]` (part of `#[deny(rust_2024_compatibility)]`) on by default
```

- Why references? A live `&mut COUNT` claims *exclusive* access. If anything else touches `COUNT` while it lives, even a call on the same thread, that's UB
- Plain reads and writes (`COUNT += 1`) and `&raw mut` make no such claim, so they still compile in `unsafe`
- But from two contexts (main and a handler), they're a **data race**: undefined behavior, not just a lost update

demos/host: cargo build -p static-mut --features misuse-mut-ref (rustc 1.93)

The Question From Lecture 7

What must be true of a `static` for two contexts to share it?

- A `static` is reachable from everywhere, so `static X: T` requires `T: Sync`
 - `&T` must be safe to use from another thread, or an interrupt handler
- `static mut` is an escape hatch that **skips that requirement**. That's why every use is `unsafe`, and why edition 2024 restricts reference creation
- So the fix is a `sync` type that makes sharing safe. Let's look at two approaches

Lecture Outline

- Concurrency with one core
- Interrupts
- Sharing state the wrong way
- **Sharing state the right way**
 - atomics
 - critical sections: `Mutex<RefCell<Option<T>>>`
 - what it costs
- Frameworks: RTIC and Embassy
- Failing safely
- Measuring

Atomics

```
static COUNT: AtomicU32 = AtomicU32::new(0);  
COUNT.fetch_add(1, Ordering::Relaxed);           // in main, and in the handler: no unsafe
```

```
main incremented 1000000 times, the handler 5813 times  
COUNT = 1005813, expected 1005813: 0 increments lost
```

- On the Cortex-M4, `fetch_add` is a **load-exclusive / store-exclusive** loop
 - `LDREX` loads and marks the address; any exception clears the mark
 - `STREX` fails if the mark is gone, and the loop retries.
 - Thus the handler's increment can't be overwritten.
- The Cortex-M0 (micro:bit V1, Raspberry Pi RP2040) has no `LDREX` / `STREX`.
There, the `portable-atomic` crate falls back to critical sections

[Live demo] demos/board: cargo run --release --bin counter_atomic

Discussion: Are Atomics Enough?

```
static PRESSES: AtomicU32 = AtomicU32::new(0);
static LAST_PRESS_MS: AtomicU32 = AtomicU32::new(0);

#[interrupt] fn GPIOTE() {                                // on each button press
    LAST_PRESS_MS.store(now_ms(), Relaxed);
    PRESSES.fetch_add(1, Relaxed);
}
// in the main loop:
let n = PRESSES.load(Relaxed);
let t = LAST_PRESS_MS.load(Relaxed);
rprintln!("{}", presses; the last at {} ms", n, t);
```

- Every access is atomic. Is there a data race? Can main print something wrong?
- How would you fix it?

Discuss in small groups for 3 minutes, share 2 minutes.

Critical Sections

```
critical_section::with(|cs| {           // single core: interrupts off for the closure
    // nothing can preempt this code
});
```

- `cs: CriticalSection<'_>` is a **token** proving you're inside one. Lecture 8's `Claim` pattern again
- `critical_section::Mutex<T>` hands out `&T` **only** to someone holding the token:

```
static COUNT: Mutex<Cell<u32>> = Mutex::new(Cell::new(0));
critical_section::with(|cs| {
    let count = COUNT.borrow(cs);      // needs the token
    count.set(count.get() + 1);
});
```

- Every field changes together, which fixes the Two Counters problem above

Moving a Peripheral Into a Handler

```
static TICKER: Mutex<RefCell<Option<Timer<TIMER0, Periodic>>>> =  
    Mutex::new(RefCell::new(None));  
  
// main, once the timer is configured:  
critical_section::with(|cs| TICKER.borrow(cs).replace(Some(timer)));  
  
#[interrupt]  
fn TIMER0() {  
    critical_section::with(|cs| {  
        if let Some(timer) = TICKER.borrow_ref(cs).as_ref() { timer.reset_event(); }  
    });  
}
```

- **Option** : a static is initialized at compile time; the timer is configured at run time
- **RefCell** : **Mutex** gives **&T** ; **RefCell** gives **&mut** with a run-time borrow check
- **Mutex** : the critical-section token proves no one else is in there

The Counter, the Right Way

```
main incremented 1000000 times, the handler 4219 times  
COUNT = 1004219, expected 1004219: 0 increments lost
```

- The handler also owns the LED pin now, and blinks it
- **Can the `RefCell` borrow fail?** Not here: inside one critical section, on one core, nothing else can be borrowing. A no-panic codebase can still use `try_borrow_mut` and handle the `Err`

[Live demo] demos/board: cargo run --release --bin counter_mutex

What It Costs

```
cycles per increment, averaged over 100000 (loop overhead included):  
volatile read + write (static mut)    9.00 cycles  
AtomicU32::fetch_add                  15.99 cycles  
critical_section::with + Cell         12.99 cycles
```

- On one core, a critical section is just "interrupts off, on": **cheaper than the atomic's LDREX / STREX loop**. (On a multi-core chip it also takes a spinlock)
- Its real cost: it **delays every interrupt** while it runs
- So: keep critical sections **short**. Do the slow work, like waiting for a DMA transfer, outside them
- In HW5, your longest critical section is your worst-case emergency-stop delay. Part 3 asks you to measure it

[Live demo] demos/board: cargo run --release --bin costs. Averages include loop overhead.

Lecture Outline

- Concurrency with one core
- Interrupts
- Sharing state the wrong way
- Sharing state the right way
- **Frameworks: RTIC and Embassy**
 - RTIC = Real-Time Interrupt-driven Concurrency
- Failing safely
- Measuring

RTIC: Sharing Checked at Compile Time

```
#[rtic::app(device = microbit::pac, dispatchers = [SWIO_EGU0])]
mod app {
    #[shared] struct Shared { count: u32 }           // used by two tasks
    #[local] struct Local { gpiote: Gpiote, led: ROW1 } // each used by one

    #[task(binds = GPIOTE, priority = 2, shared = [count], local = [gpiote])]
    fn button(mut cx: button::Context) {             // the button's interrupt
        cx.local.gpiote.channel0().reset_events();
        cx.shared.count.lock(|count| *count += 1);
    }

    #[task(priority = 1, shared = [count], local = [led])]
    async fn blink(mut cx: blink::Context) {
        loop {
            let _ = cx.local.led.toggle();
            let now = cx.shared.count.lock(|count| *count); // holds off `button`
            /* ... print `now` if it changed ... */
            Mono::delay(250.millis()).await;
        }
    }
}
```

- **Resources** are declared per task; `lock` raises the priority to the resource's **ceiling** (the Stack Resource Policy). No deadlocks, by construction
- `Mutex<RefCell<_>>`, scaled up and checked at compile time by a macro

Embassy: Async Without a Heap

```
static COUNT: AtomicU32 = AtomicU32::new(0);

#[embassy_executor::task]
async fn button(mut button_a: Input<'static>) {
    loop {
        button_a.wait_for_falling_edge().await; // sleeps until GPIOTE wakes it
        COUNT.fetch_add(1, Ordering::Relaxed);
    }
}

#[embassy_executor::task]
async fn blink(mut row1: Output<'static>, _col1: Output<'static>) {
    loop {
        row1.toggle();
        /* ... print COUNT if it changed ... */
        Timer::after_millis(250).await;
    }
}
```

- Tasks are `async fn`s whose futures are **statically allocated**: no heap
- An interrupt **wakes a waker** (Lecture 9). "Don't block the executor" applies here too
- Not required for HW5, which builds the layer underneath, so the sharing is explicit

demos/frameworks/embassy-counter (embassy-executor 0.10.0, embassy-nrf 0.11.0, embassy-time 0.5.1). Runs on the micro:bit. Why ``_col1``: dropping an embassy Output disconnects the pin.

RTIC or Embassy?

	RTIC	Embassy
Scheduling	Preemptive , by hardware priority	Cooperative : a task runs until it <code>.await S</code>
Sharing	<code>lock</code> , with ceilings checked at compile time	Your choice: atomics, mutexes, channels
Worst-case response	Bounded by priorities and lock lengths	Depends on every task's time between <code>.await S</code>
Drivers	Bring your own HAL	Async drivers for most peripherals
Best for	Hard real-time control	I/O-heavy firmware: radios, USB, networking
Our counter	4.6 KB of code	6.2 KB of code

- Both allocate everything statically, and they mix: Embassy has interrupt executors for priorities, and RTIC 2's async tasks can `.await` Embassy's drivers

Code sizes: .text of the two demos/frameworks counters, release builds (llvm-size).

Lecture Outline

- Concurrency with one core
- Interrupts
- Sharing state the wrong way
- Sharing state the right way
- Frameworks: RTIC and Embassy
- **Failing safely**
 - no panics; overflow
 - safe defaults, latching, hysteresis, missing data
 - test hooks; hardware vs. software stops; watchdogs
- Measuring

Panics in Bare-Metal Rust

- There's no process to exit, and no one to read a message. With `panic-halt`, the device **stops responding**
- For an actuator, "stopped responding **while on**" is the hazard
- So: a **no-panic discipline** for code the device runs

```
#![deny(clippy::unwrap_used, clippy::expect_used, clippy::panic, clippy::indexing_slicing)]
```

- `Result` everywhere, and handle the `Err`
- `.get(i)` instead of `[i]`
- `panic-halt` only as the last resort

Integer Overflow

- In a **debug** build, `u32::MAX + 1` **panics**. In release, it wraps
- So say what you mean:

Value	Use	Why
a temperature, a count, an offset	<code>saturating_add</code> / <code>saturating_sub</code>	must not wrap around to a nonsense value
a tick counter, a timestamp	<code>wrapping_add</code> , and compare with <code>wrapping_sub</code>	wraps by design

- `now.wrapping_sub(last) >= DEBOUNCE` is correct even when the counter wraps (the button demo)

Safe Defaults and Latching

- The safe state for the heater is "off" (de-energized). Make it the default, the state after reset, and the state every fault falls into
- **Latching:** after a trip, stay tripped until someone **deliberately** resets.
Never clear yourself because the cause seems to have gone away
- **Missing data is a fault:** a sensor that stops answering must cause a trip, never "keep using the last value"

Hysteresis: Two Thresholds, Not One

A pump keeps a tank at about 50 cm; the level sensor is noisy (± 1.5 cm):

```
pump state, first 72 steps (# = on):  
one threshold (50 cm):      .#.#..#.#.#.#..#..#..#..#.#.##...##....#.###.#.###.#.....##..##.#..#.....  
hysteresis (45 / 55 cm):    .....#####.....  
switches in 600 steps: one threshold 315, hysteresis 16
```

- At one threshold, **noise** decides: the pump chatters on and off
- With two, the controller holds its state **between** them

demos/host: cargo run -p hysteresis (deterministic noise)

Test Hooks Can Only Move Toward Safety

A pressure monitor shuts a valve above 10 bar. For testing, a serial command adds a **calibration offset** to the reading.

- An offset that can be **negative** makes the monitor believe the pressure is lower than it is: a **backdoor around the safety function**
- So: a test hook like the calibration offset mentioned above may only make the system safer. A production build should remove it, or at least authenticate it
- Your HW5 write-up asks you to think about test hooks too

Hardware Versus Software Stops

- A real emergency stop is a **hardwired, normally-closed** circuit: pressing it cuts power, whatever the firmware is doing
- A **software** stop can be **delayed** by a long critical section, or **defeated** by a hang with interrupts off
- If the stop is software:
 - do the kill **in the handler**, not "set a flag for the main loop"
 - keep critical sections short (their length *is* the delay)
 - use a watchdog as the backstop
- **Therac-25** (1985–87): software replaced the hardware interlocks; a race condition then overdosed patients

Watchdogs

- A hardware timer that **resets the chip** unless the program reloads ("pets") it in time
- On the nRF52, once started **it can't be stopped**, and it **survives** flashing and the reset button. Only its own reset, or a power cycle, stops it (measured on a micro:bit)
- **RESETREAS** tells the next boot why it reset: power-on, the reset pin, a software request, the watchdog, a lockup

Question: what should a device do when it boots after a watchdog reset?

Lecture Outline

- Concurrency with one core
- Interrupts
- Sharing state the wrong way
- Sharing state the right way
- Frameworks: RTIC and Embassy
- Failing safely
- **Measuring**

Measuring With the Cycle Counter

```
cp.DCB.enable_trace();
cp.DWT.enable_cycle_counter();           // the DWT: an ARM core peripheral
let start = DWT::cycle_count();          // 64 MHz: 15.6 ns per cycle
/* ... */
let took = DWT::cycle_count().wrapping_sub(start);
```

- **Interrupt latency:** read the counter, `NVIC::pend` an interrupt, read the counter first thing in the handler. Report the **distribution**, not one number
- **CPU cost:** two ways of doing the same work can take the same **wall time** and very different **CPU time**. In HW5 Part 3 you will measure CPU time for `write_blocking` versus `write_dma`
- The counter stops while the CPU sleeps in `wfi`

Summary

- **One core is still concurrent:** an interrupt is a thread the hardware schedules
- `static mut` doesn't check for `Sync`. Use atomics for single values, and a critical-section `Mutex` for anything compound
- **Keep critical sections short:** they delay every interrupt
- **Fail safely:** no panics, safe defaults, latching, hysteresis, missing data as a fault, test hooks only move toward safety, and a watchdog
- **Measure** with the cycle counter

HW5 Part 2 is unlocked. Next: **Verus**. The safety core you're writing is something we'll *prove* things about, not just test