Part 2

Embedded Rust Workshop

tweede golf

web / security / embedded

Recap

- Rust’s embedded ecosystem
  - HALs, PACs, embedded-hal
- Portable drivers in Rust
  - Traits
  - Generics

Questions on reading material?

Our day

- **Ask questions anytime!**
- Interrupt me when needed
- Help each other out

We’ll see how far we get

?

Our day

- The RTIC runtime
- Exercise 2A: RTIC basics

Bonus material:

- Rust in IoT
- Exercise 2B: Device-host communication

Part 2A

RTIC

tweede golf

web / security / embedded

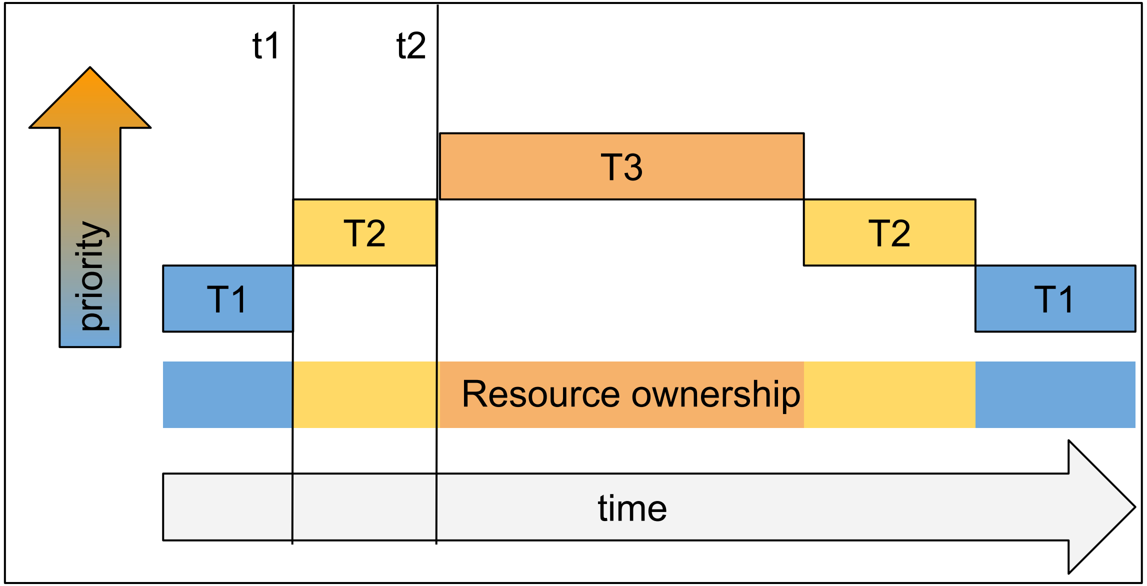
Contents

- **The concurrency problem**
- RTIC introduction
- App structure
- Exercise 2A

Sharing data between tasks

- Multiple tasks
- Shared resource
- Task preemption
- Data corruption

Bad things happen!

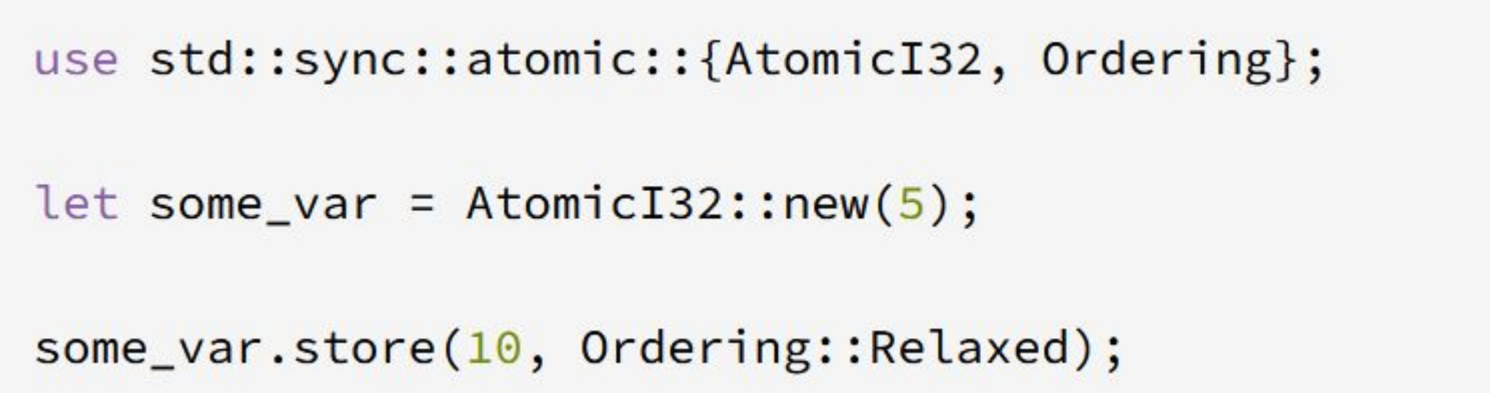


Sharing data

- Rust is pedantic about sharing globals
- Dereferencing mutable globals is `unsafe`

How to safely share mutable data between app code and interrupts normally?

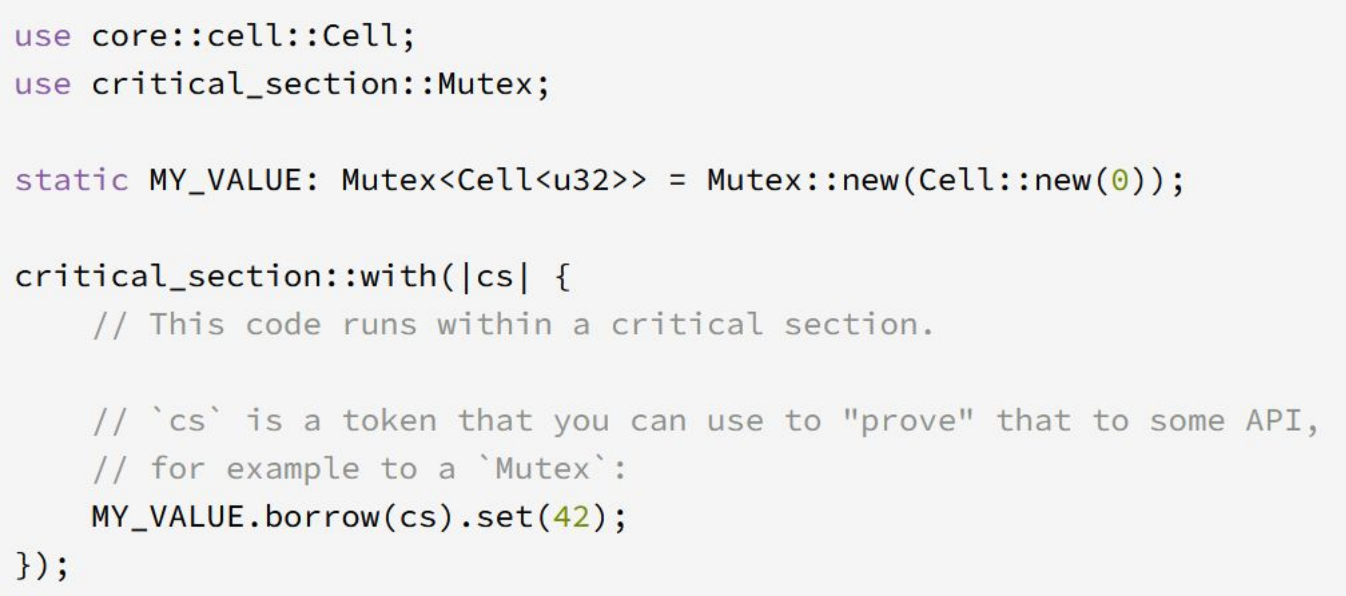
Atomics



```
use std::sync::atomic::{AtomicI32, Ordering};

let some_var = AtomicI32::new(5);

some_var.store(10, Ordering::Relaxed);
```

- Atomic numbers can be shared and mutated
- Good for flags and global counters

Critical sections and mutexes



```rust
use core::cell::Cell;
use critical_section::Mutex;

static MY_VALUE: Mutex<Cell<u32>> = Mutex::new(Cell::new(0));

critical_section::with(|cs| {
    // This code runs within a critical section.

    // `cs` is a token that you can use to "prove" that to some API,
    // for example to a `Mutex`:
    MY_VALUE.borrow(cs).set(42);
});
```

- `critical_seciotn::with` needed to mutate data in CS
- Disables **all** interrupts

Questions so far?

?

Contents

- The concurrency problem
- **RTIC introduction**
- App structure
- Exercise 2A

Real-Time Interrupt-driven Concurrency

- Divide application into tasks
- Heavily uses interrupts to schedule tasks
- Handles passing global resources

Lock only when pre-emption might cause trouble

RTIC features

- Message passing
- Task scheduling
- Deadlock-free execution
- Works on all cortex-m devices
- Lots of control

RTIC trade offs

Heavy on the macros

- Rust analyzer doesn’t like macros
- Compiler still helpful though

RTIC internals

- All tasks are interrupts
- Interrupt priority managed by NVIC
- Priority ceiling

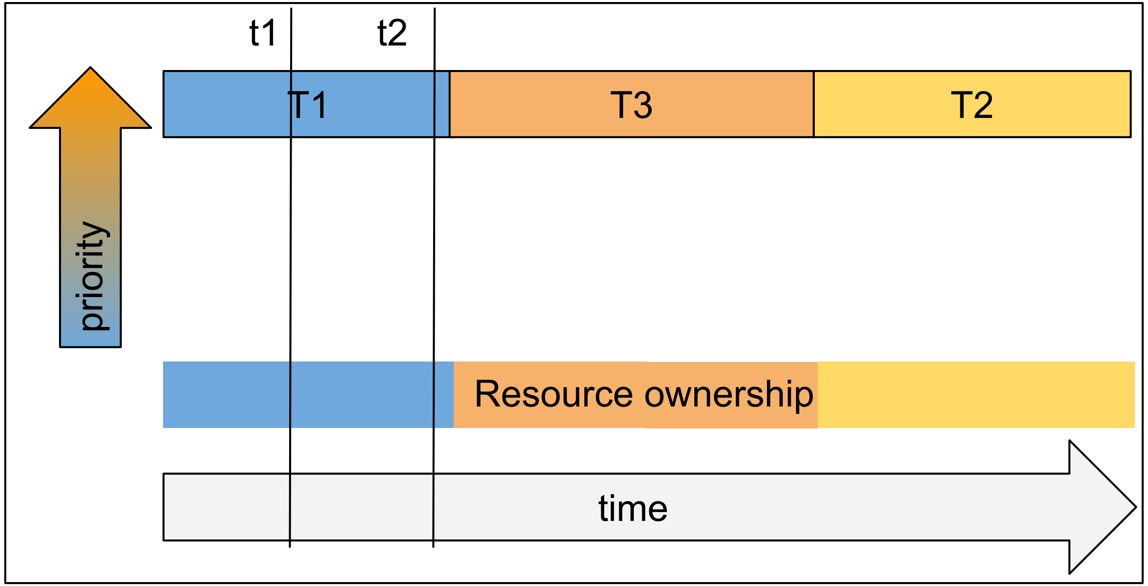
When locking a shared resource, temporarily increase current tasks priority so that no task that uses the resource can preempt the current task.

Other tasks may still preempt

[Idea by T.P. Baker (1990)]

RTIC priority ceiling

- T1 finishes first
- T2 has to wait
- No preemption



Questions so far?

?

Contents

- The concurrency problem
- RTIC introduction
- **App structure**
- Exercise 2A

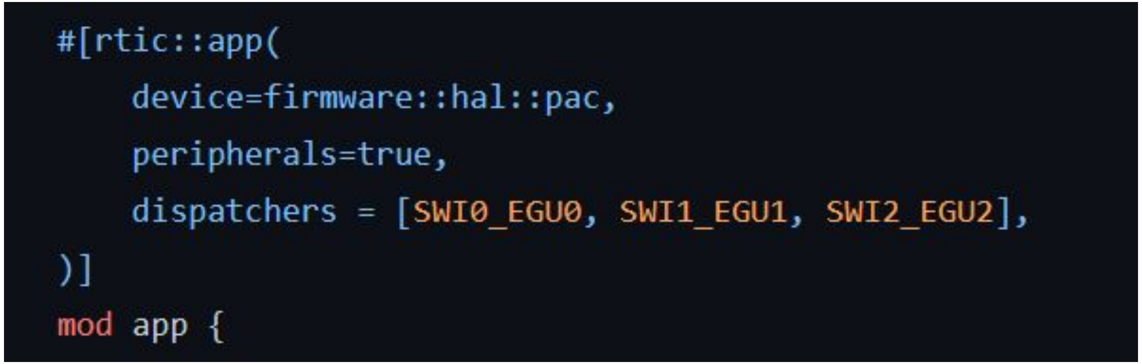
RTIC app outline

- App attribute
- Resources
- Monotonic counter
- Init task
- Idle task
- Hardware tasks
- Software tasks

Example

RTIC app attribute

- Point RTIC to PAC
- Procedural macro
- Analyzes task priorities
- Dispatchers

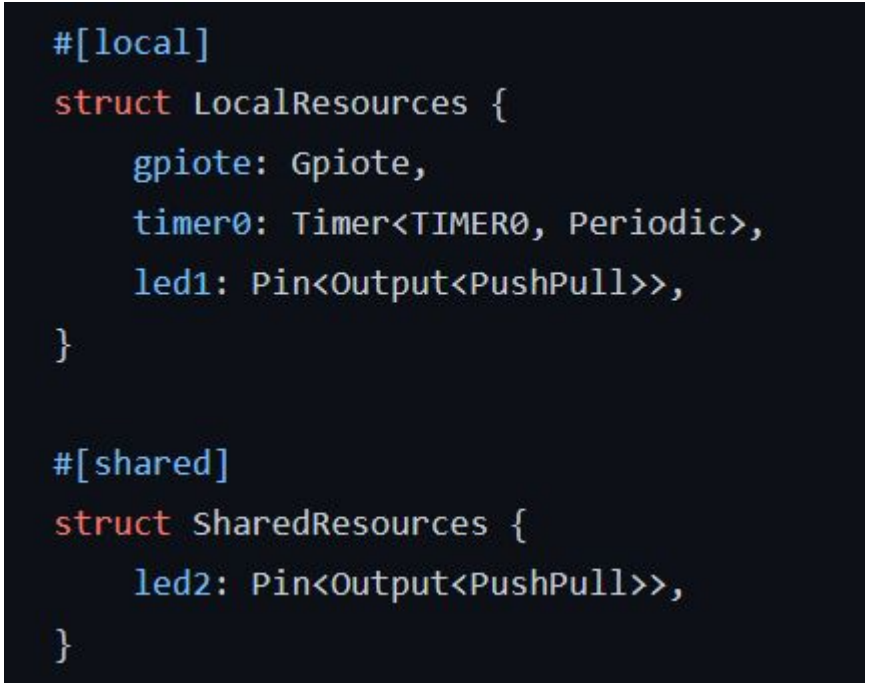


```
#[rtic::app(
    device=firmware::hal::pac,
    peripherals=true,
    dispatchers = [SWI0_EGU0, SWI1_EGU1, SWI2_EGU2],
)]
mod app {
```

Code gets adapted at build!

RTIC resources

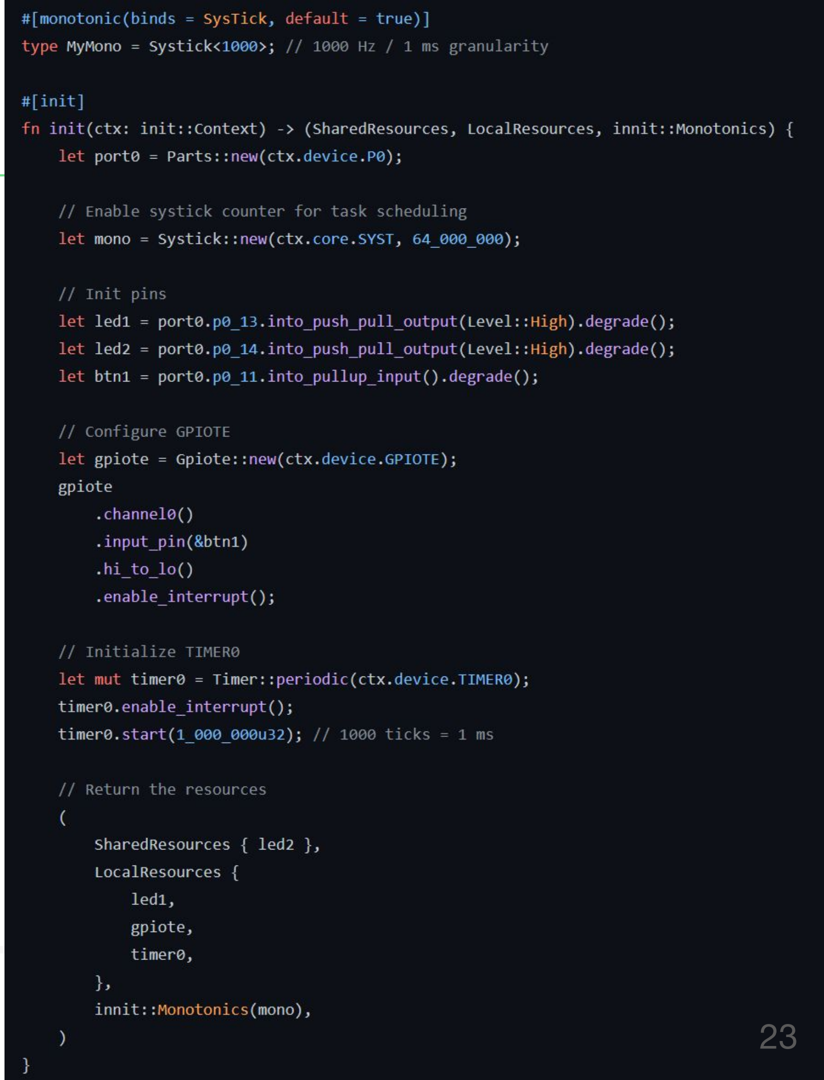
- Shared between tasks
- Initialized in init task
- Some shared and some local resources



```
#[local]
struct LocalResources {
    gpiote: Gpiote,
    timer0: Timer<TIMER0, Periodic>,
    led1: Pin<Output<PushPull>>,
}

#[shared]
struct SharedResources {
    led2: Pin<Output<PushPull>>,
}
```

RTIC init

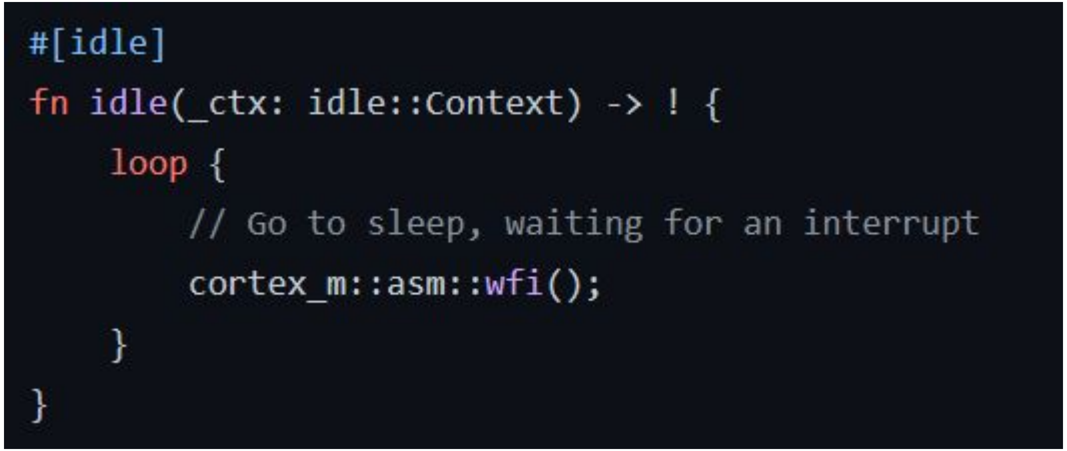
- Initialize all resources
- Peripherals in `ctx.device`
- Use PAC and HAL
- Interrupts disabled
- Also initialize the monotonic timer
  - Used to schedule tasks



```rust
#[monotonic(binds = SysTick, default = true)]
type MyMono = Systick<1000>; // 1000 Hz / 1 ms granularity

#[init]
fn init(ctx: init::Context) -> (SharedResources, LocalResources, innit::Monotonics) {
    let port0 = Parts::new(ctx.device.P0);

    // Enable systick counter for task scheduling
    let mono = Systick::new(ctx.core.SYST, 64_000_000);

    // Init pins
    let led1 = port0.p0_13.into_push_pull_output(Level::High).degrade();
    let led2 = port0.p0_14.into_push_pull_output(Level::High).degrade();
    let btn1 = port0.p0_11.into_pullup_input().degrade();

    // Configure GPIOTE
    let gpiote = Gpiote::new(ctx.device.GPIOTE);
    gpiote
        .channel0()
        .input_pin(&btn1)
        .hi_to_lo()
        .enable_interrupt();

    // Initialize TIMER0
    let mut timer0 = Timer::periodic(ctx.device.TIMER0);
    timer0.enable_interrupt();
    timer0.start(1_000_000u32); // 1000 ticks = 1 ms

    // Return the resources
    (
        SharedResources { led2 },
        LocalResources {
            led1,
            gpiote,
            timer0,
        },
        innit::Monotonics(mono),
    )
}
```

RTIC idle task

- Default task
- Sleep, mostly (default)
- Pre-empted by other tasks
- Can be left out

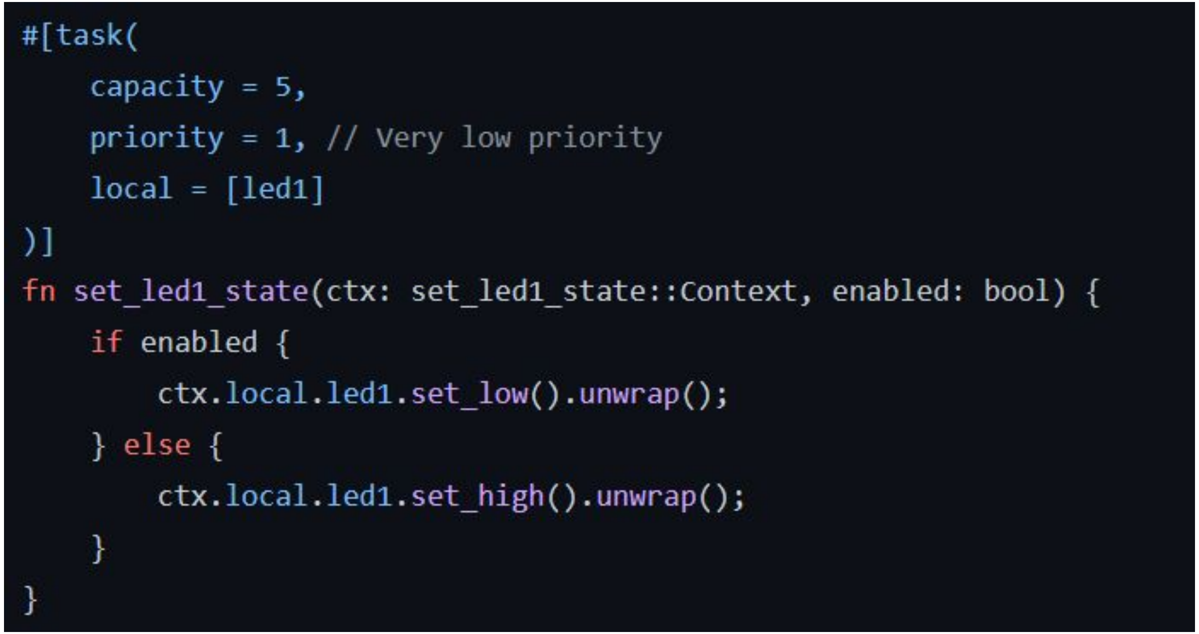


```rust
#[idle]
fn idle(_ctx: idle::Context) -> ! {
    loop {
        // Go to sleep, waiting for an interrupt
        cortex_m::asm::wfi();
    }
}
```

RTIC software task

- Capacity
- Priority
- Resources declared
- Message passing
- Task context

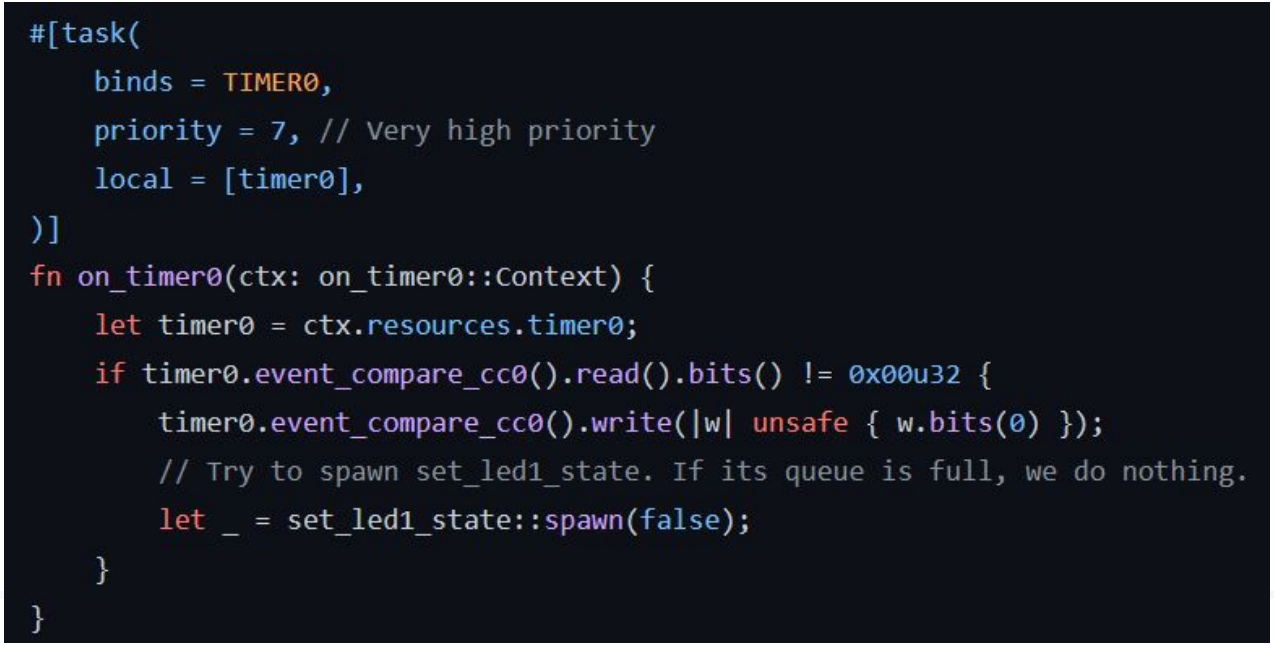
Resources are `&mut`!



```
#[task(
    capacity = 5,
    priority = 1, // Very low priority
    local = [led1]
)]
fn set_led1_state(ctx: set_led1_state::Context, enabled: bool) {
    if enabled {
        ctx.local.led1.set_low().unwrap();
    } else {
        ctx.local.led1.set_high().unwrap();
    }
}
```

RTIC hardware task

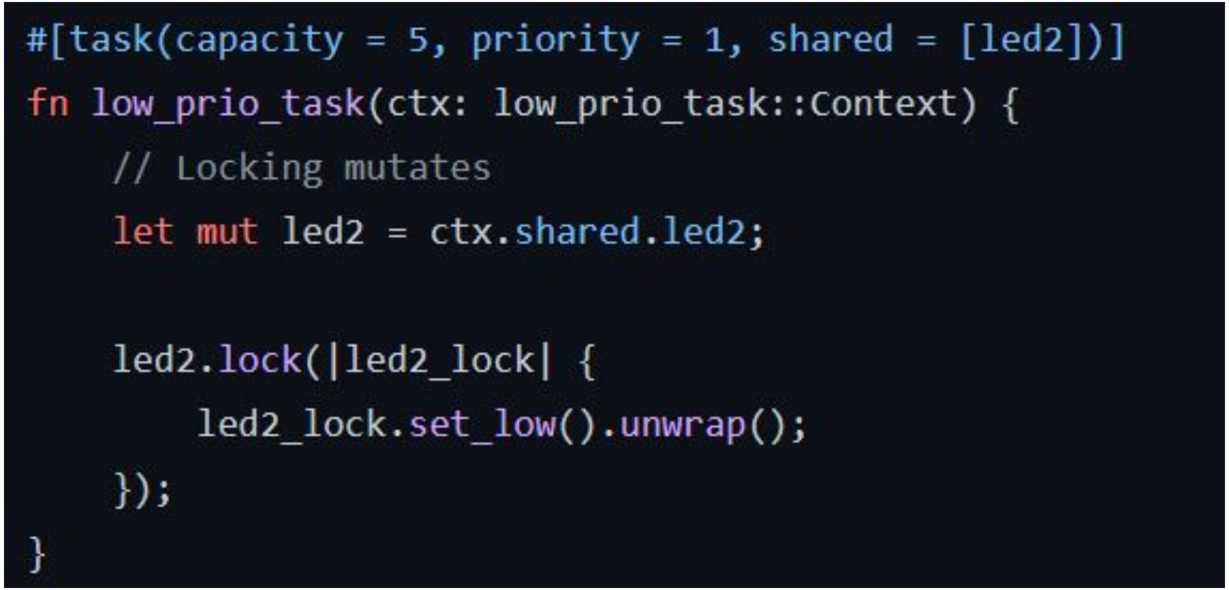
- Binds hardware interrupt
- High priority
- Resources
- Spawned SW tasks
- Task context



```rust
#[task(
    binds = TIMER0,
    priority = 7, // Very high priority
    local = [timer0],
)]
fn on_timer0(ctx: on_timer0::Context) {
    let timer0 = ctx.resources.timer0;
    if timer0.event_compare_cc0().read().bits() != 0x00u32 {
        timer0.event_compare_cc0().write(|w| unsafe { w.bits(0) });
        // Try to spawn set_led1_state. If its queue is full, we do nothing.
        let _ = set_led1_state::spawn(false);
    }
}
```

RTIC shared resources

- Shared resources need to be locked



```rust
#[task(capacity = 5, priority = 1, shared = [led2])]
fn low_prio_task(ctx: low_prio_task::Context) {
    // Locking mutates
    let mut led2 = ctx.shared.led2;

    led2.lock(|led2_lock| {
        led2_lock.set_low().unwrap();
    });
}
```

Questions so far?

?

Exercise 2A

Instructions in the git repo.

Exercise project located in ‘assignments/part_2a’.

Exercise description located in ‘material/part_2/assignment_a.md’

Exercice 2A round up

- Show your code!
- Any questions?

?

Part 2B

Rust in IoT

tweede golf

web / security / embedded

Contents

- **Project intro and demo**
- Exercise 2B

Demo: A bigger Project in Rust

- Intro
    - Device-host communication
    - Abstract over channel
- Project structure
    - Workspace with crates
    - Shared format crate
    - Firmware
    - Command-line application
- Serde and postcard
- CLI REPL

Questions so far?

?

Exercise 2B

Instructions in the git repo.

Exercise project located in ‘assignments/part_2b’.

Exercise description located in ‘material/part_2/assignment_b.md’

Exercice 2B round up

- Show your code!
- Any questions?

AMA

?

tweede golf

web / security / embedded

Castellastraat 26, 6512 EX Nijmegen
info@tweedegolf.com
024 3010 484