Dion Dokter

There’s going to be lots of code.

Please sit close, I don’t bite!

If you can read this, you should be alright.

Who am I?

- Dion Dokter
- Tweede golf

- Embedded Rust since 2019
- Fan of async on embedded

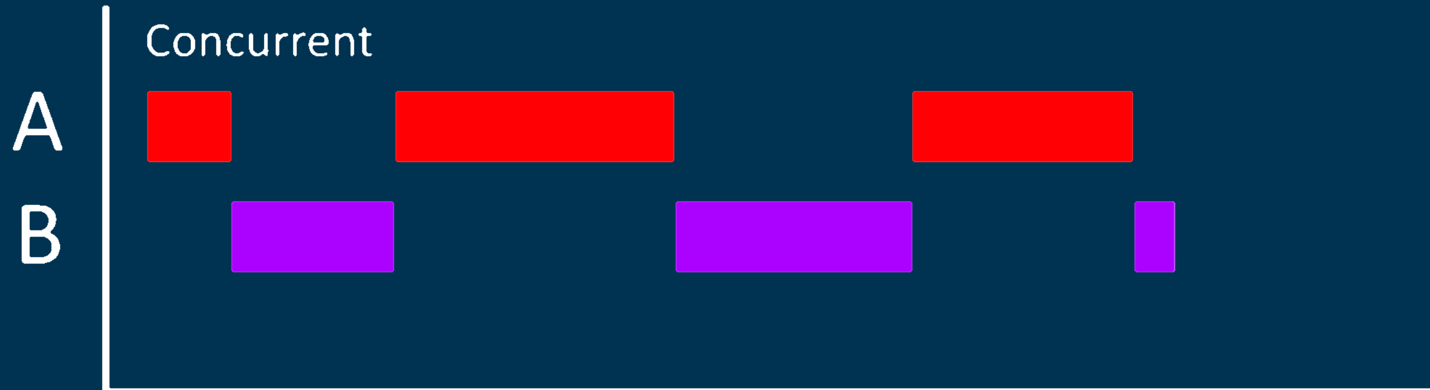
- @geoxion
- @diondokter@fosstodon.org

Agenda

- What is asynchronous coding?
- Polling loop
- Interrupts
- Async Rust
- Futures
- Wakers
- Real embassy code

What is asynchronous coding?

- Doing multiple things 'at the same time'
- Parallel vs concurrent
- Parallel often not possible on embedded
- Two ways to do concurrency



Polling loop

- Easy, just a loop
- Polling events
- Not efficient

```
fn main() {
    let button = Button::new();
    let mut led = Led::new();

    loop {
        let button_state = button.get();
        led.set(button_state);
    }
}
```

Polling loop

- This gets messy very fast
- What happens when one poll takes a long time?

```
fn main() {
    let mut state = State::new();

    loop {
        poll_blink(&mut state.led_blink_state);
        poll_radio(&mut state.radio_state);
    }
}

pub enum LedBlinkState {
    Uninitialized,
    Off { led: Led },
    WaitForOn { led: Led, timestamp: Instant },
    WaitForOff { led: Led, timestamp: Instant },
}

fn poll_blink(state: &mut LedBlinkState) {
    let current_time = Instant::now();

    match core::mem::replace(state, LedBlinkState::Uninitialized) {
        LedBlinkState::WaitOn { mut led, timestamp } if current_time >= timestamp => {
            led.turn_on();
            *state = LedBlinkState::WaitForOff {
                led, timestamp: timestamp + Duration::from_secs(1)
            };
        }
        LedBlinkState::WaitOff { mut led, timestamp } if current_time >= timestamp => {
            led.turn_off();
            *state = LedBlinkState::WaitForOn {
                led, timestamp: timestamp + Duration::from_secs(1)
            };
        }
        _ => return,
    }
}
```

Interrupts

- Very efficient
- A lot of setup
- Not very readable
- RTIC exists though

```
static BUTTON: Mutex<RefCell<Option<Button>>> =
    Mutex::new(RefCell::new(None));
static LED: Mutex<RefCell<Option<Led>>> =
    Mutex::new(RefCell::new(None));

#[interrupt]
fn GPIO_INT() {
    critical_section::with(|cs| {
        let current_level = BUTTON.borrow_ref_mut(cs)
            .unwrap()
            .get();
        LED.borrow_ref_mut(cs).unwrap().set(current_level);
    });
}

fn main() {
    let button = Button::new();
    let mut led = Led::new();

    critical_section::with(|cs| {
        button.enable_any_edge_interrupt();

        unsafe { NVIC::unmask(GPIO_INT_IRQ);}

        *BUTTON.borrow_ref_mut(cs).unwrap() = Some(button);
        *LED.borrow_ref_mut(cs).unwrap() = Some(led);
    });

    loop {
        cortex_m::asm::wfi();
    }
}
```

Async Rust

- Looks easy & readable
- How does it work?
- Efficient?

```rust
#[embassy_executor::main]
async fn main(spawner: Spawner) {
    let p = embassy_nrf::init(
        Default::default()
    );

    let mut button = Input::new(
        p.P0_11, Pull::Up
    );
    let mut led = Output::new(
        p.P0_12,
        Level::Low,
        OutputDrive::Standard
    );

    loop {
        button.wait_for_any_edge().await;
        led.set_level(button.get_level());
    }
}
```

Futures

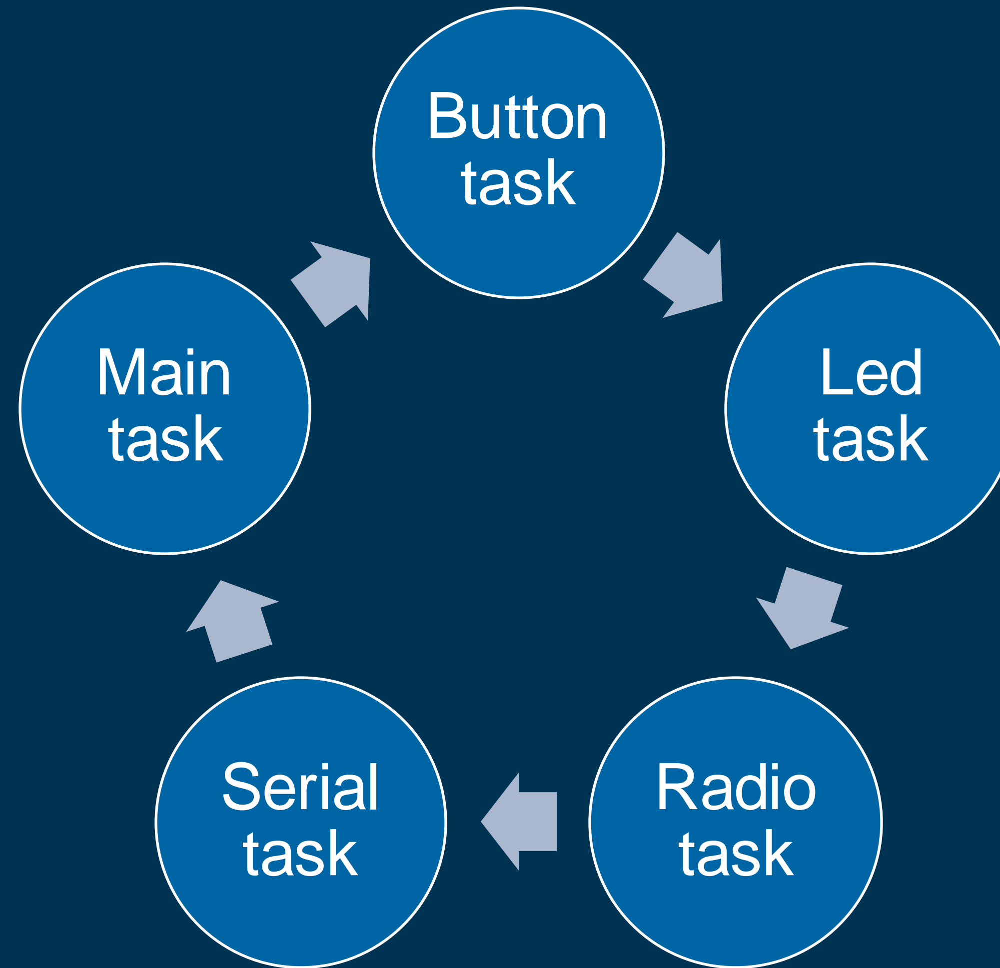
- Async fn's get turned into Futures
- Polling
- Executor
- Tasks

```
pub trait Future {

    type Output;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Self::Output>;
}
```

Futures

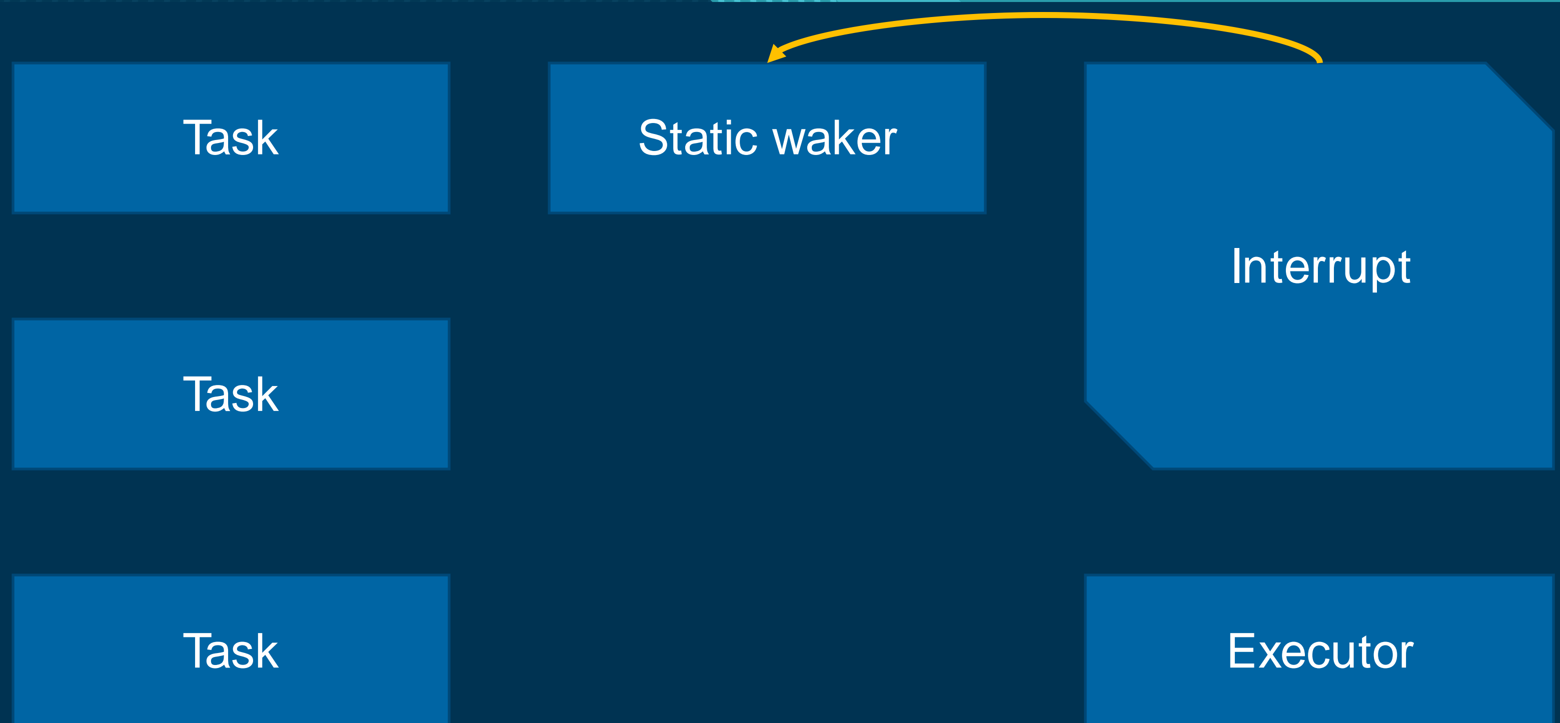
- Async fn's get turned into Futures
- Polling
- Executor
- Tasks

- Make it more efficient?
- How to use interrupts?



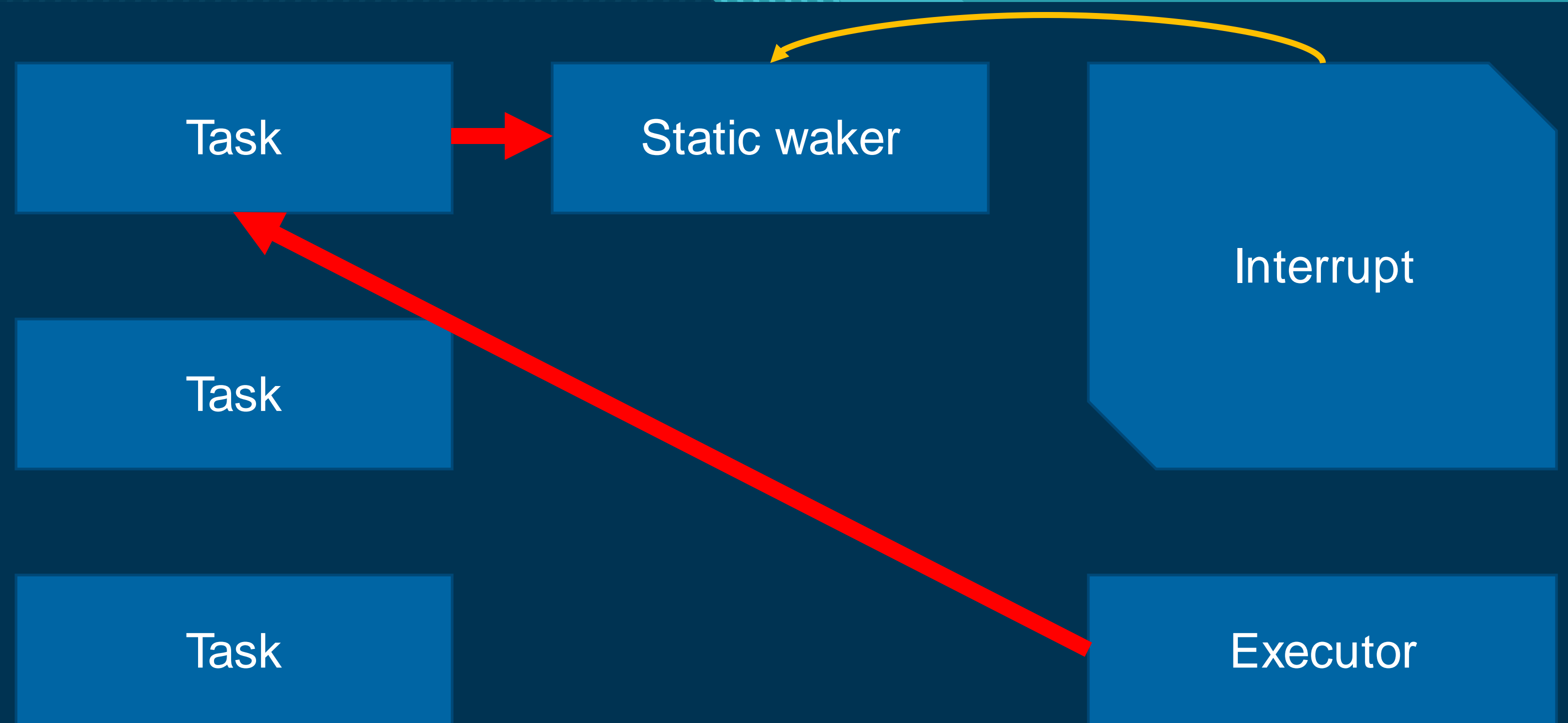
Wakers

- Signal to executor
- Abstract way of setting a flag in the tasks
- Accessed in context parameter of Future



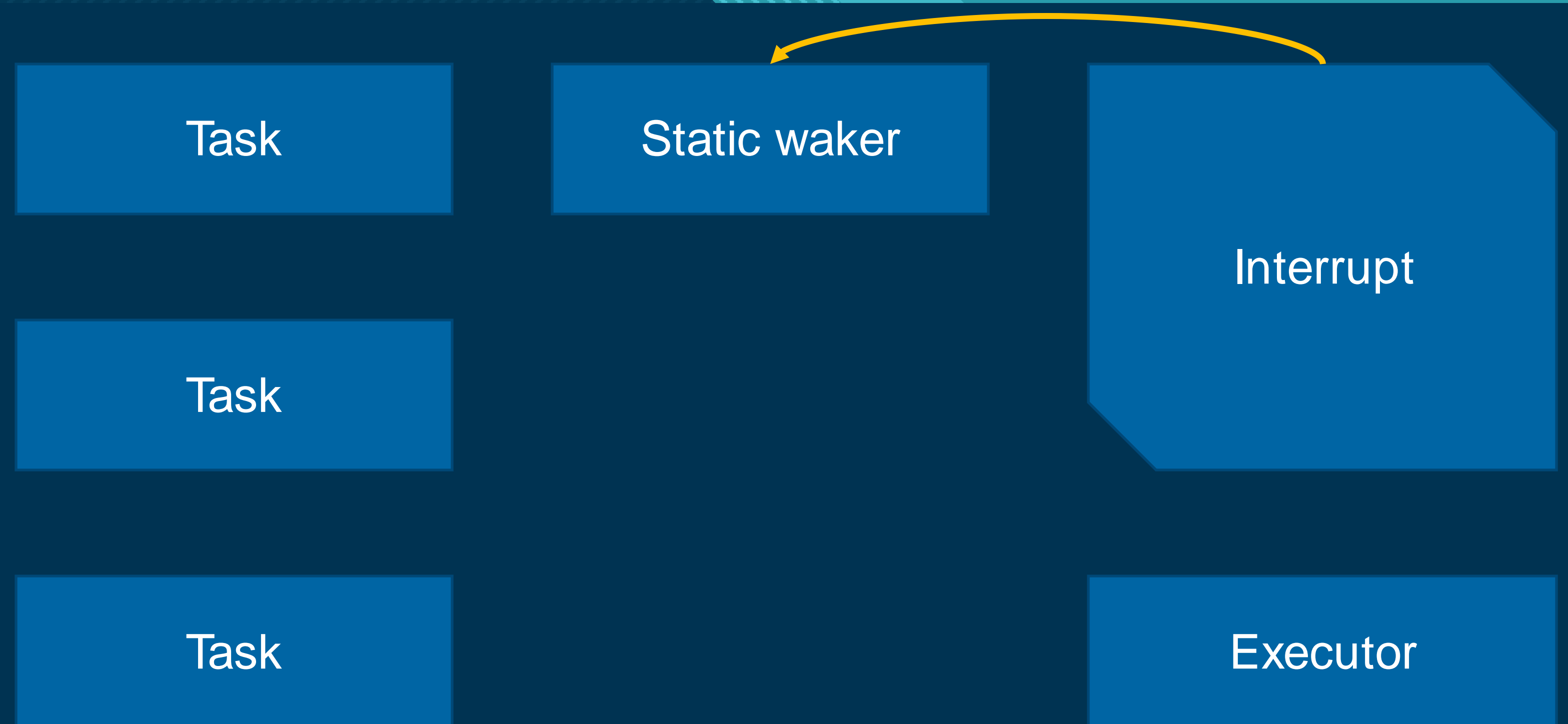
Wakers

- Signal to executor
- Abstract way of setting a flag in the tasks
- Accessed in context parameter of Future



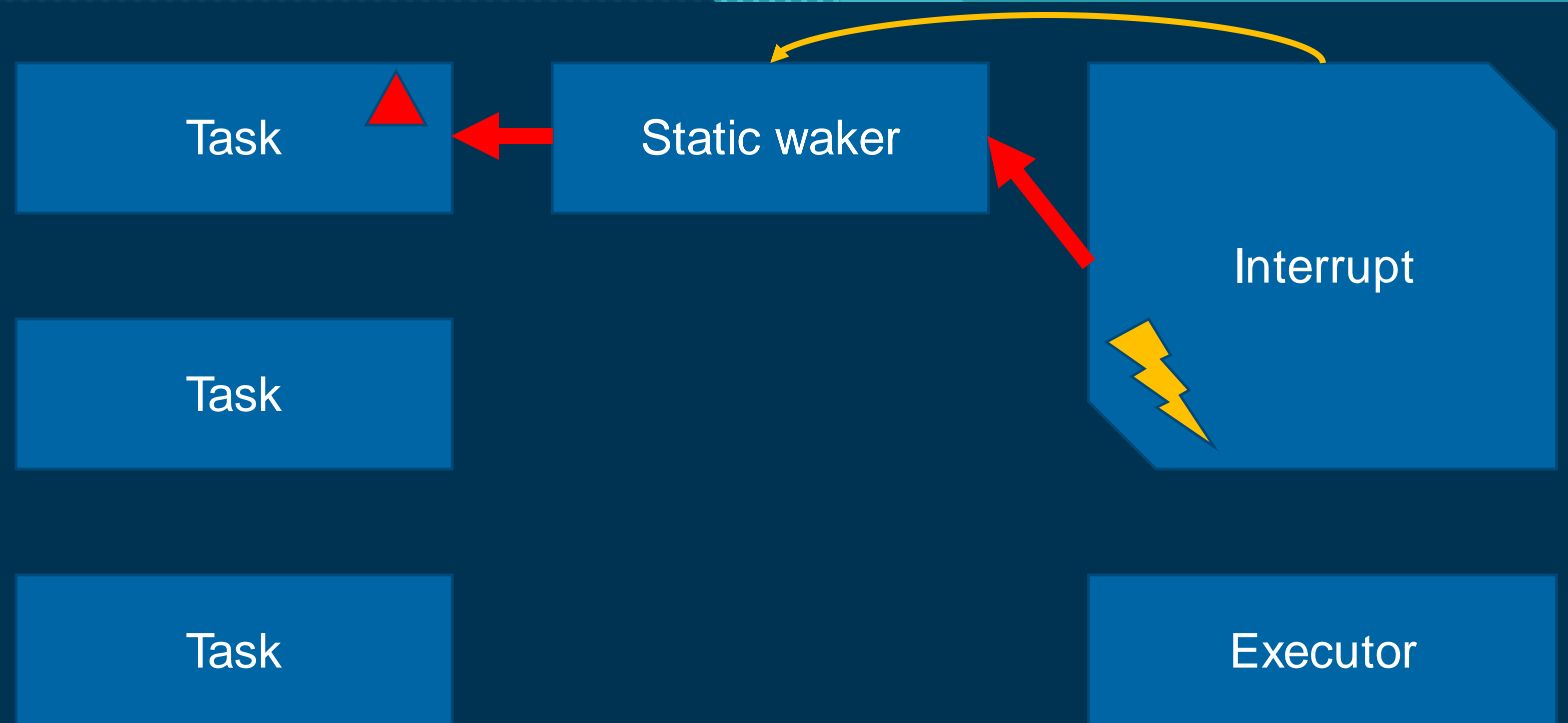
Wakers

- Signal to executor
- Abstract way of setting a flag in the tasks
- Accessed in context parameter of Future



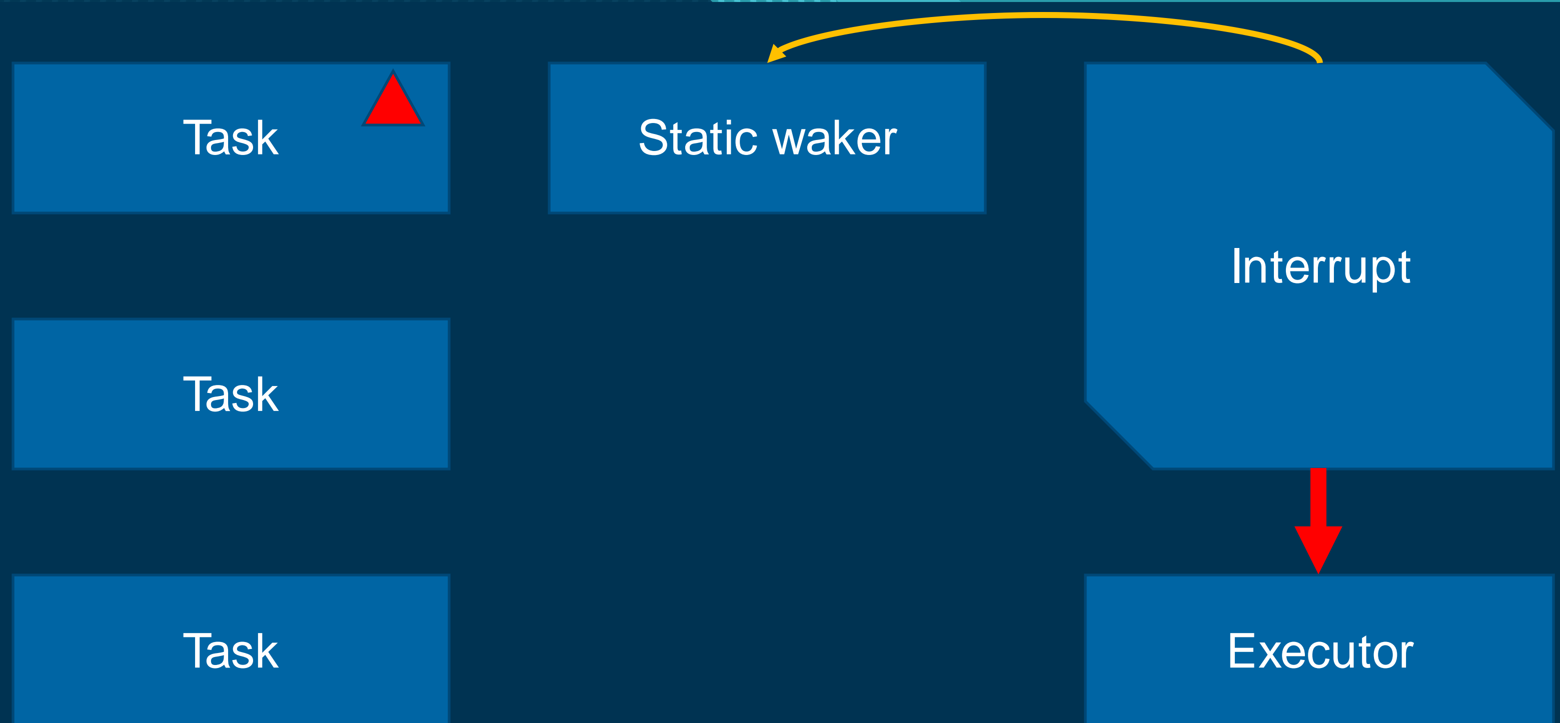
Wakers

- Signal to executor
- Abstract way of setting a flag in the tasks
- Accessed in context parameter of Future



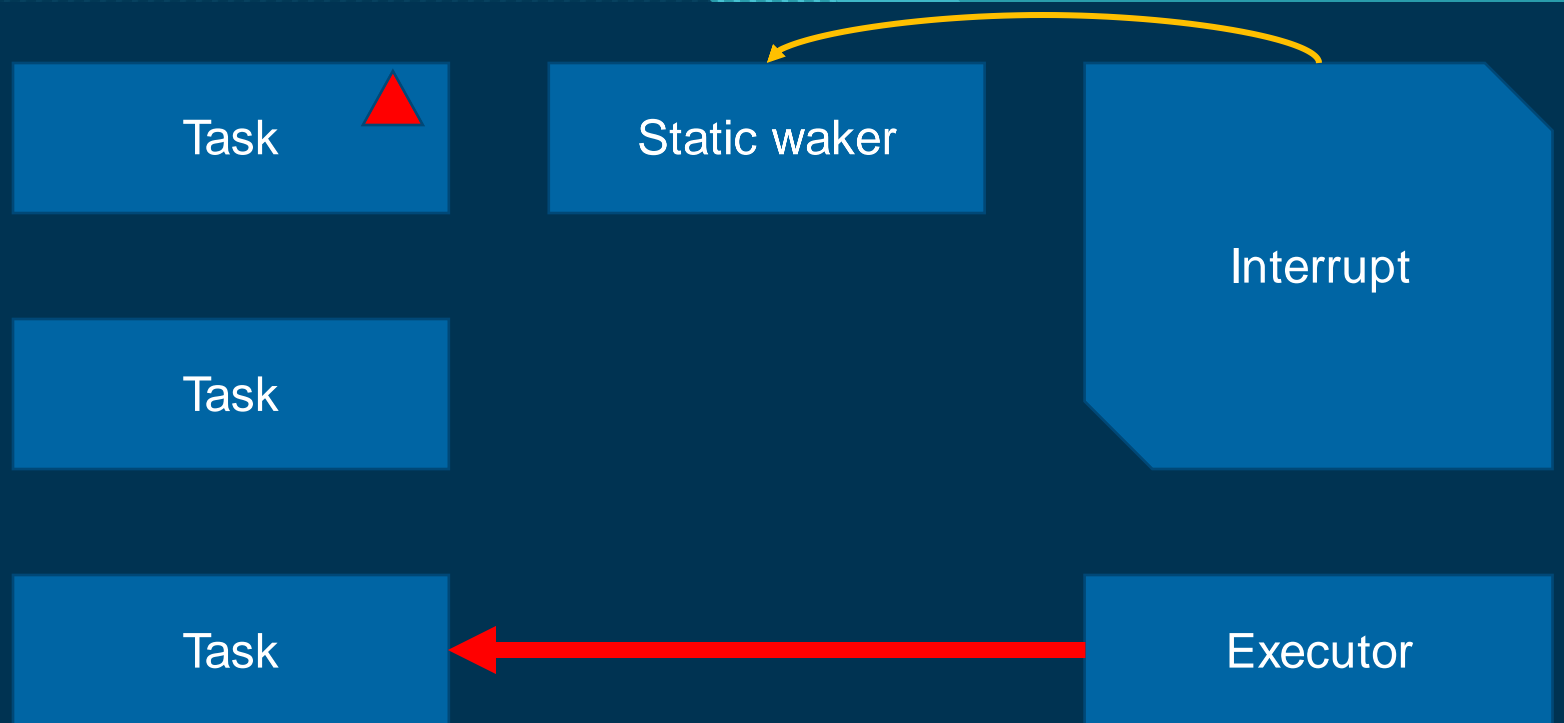
Wakers

- Signal to executor
- Abstract way of setting a flag in the tasks
- Accessed in context parameter of Future



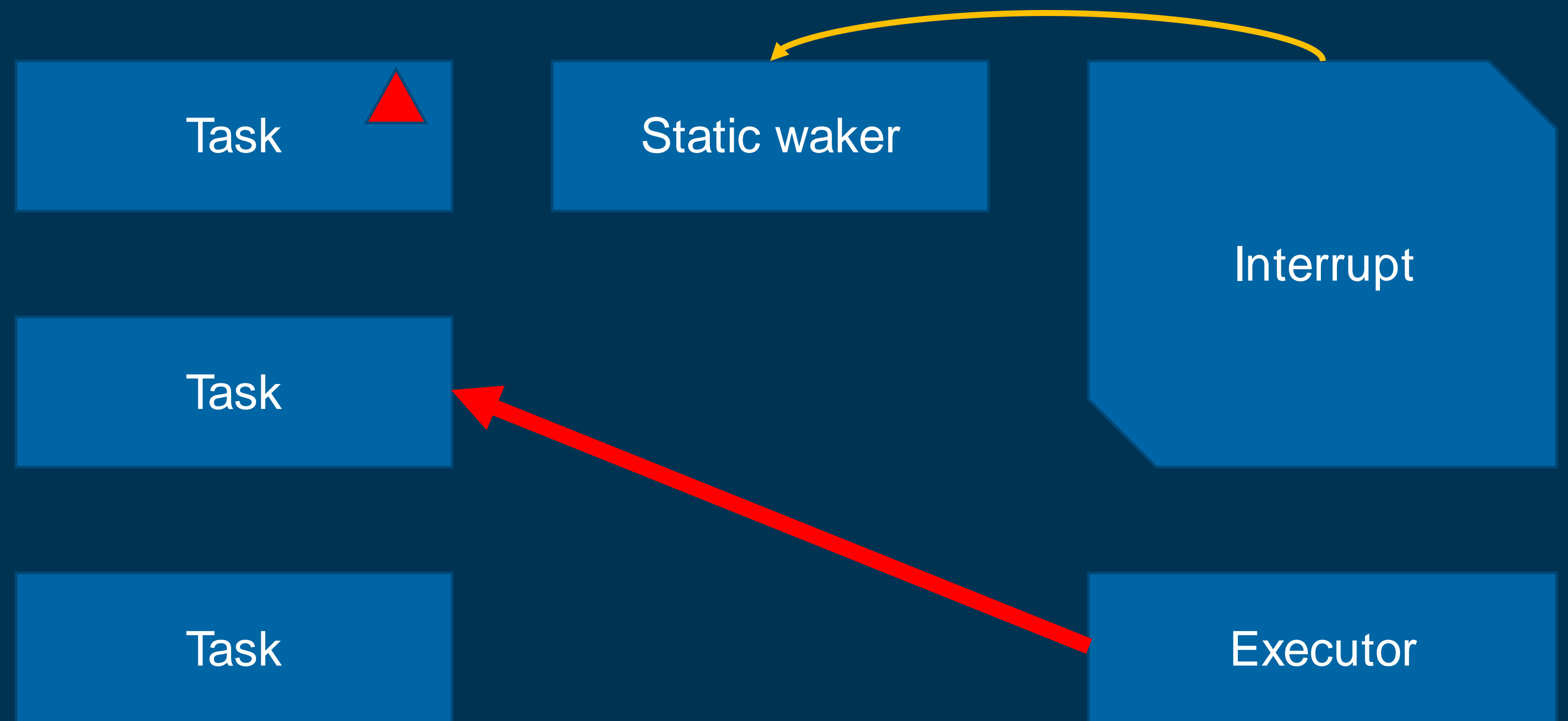
Wakers

- Signal to executor
- Abstract way of setting a flag in the tasks
- Accessed in context parameter of Future



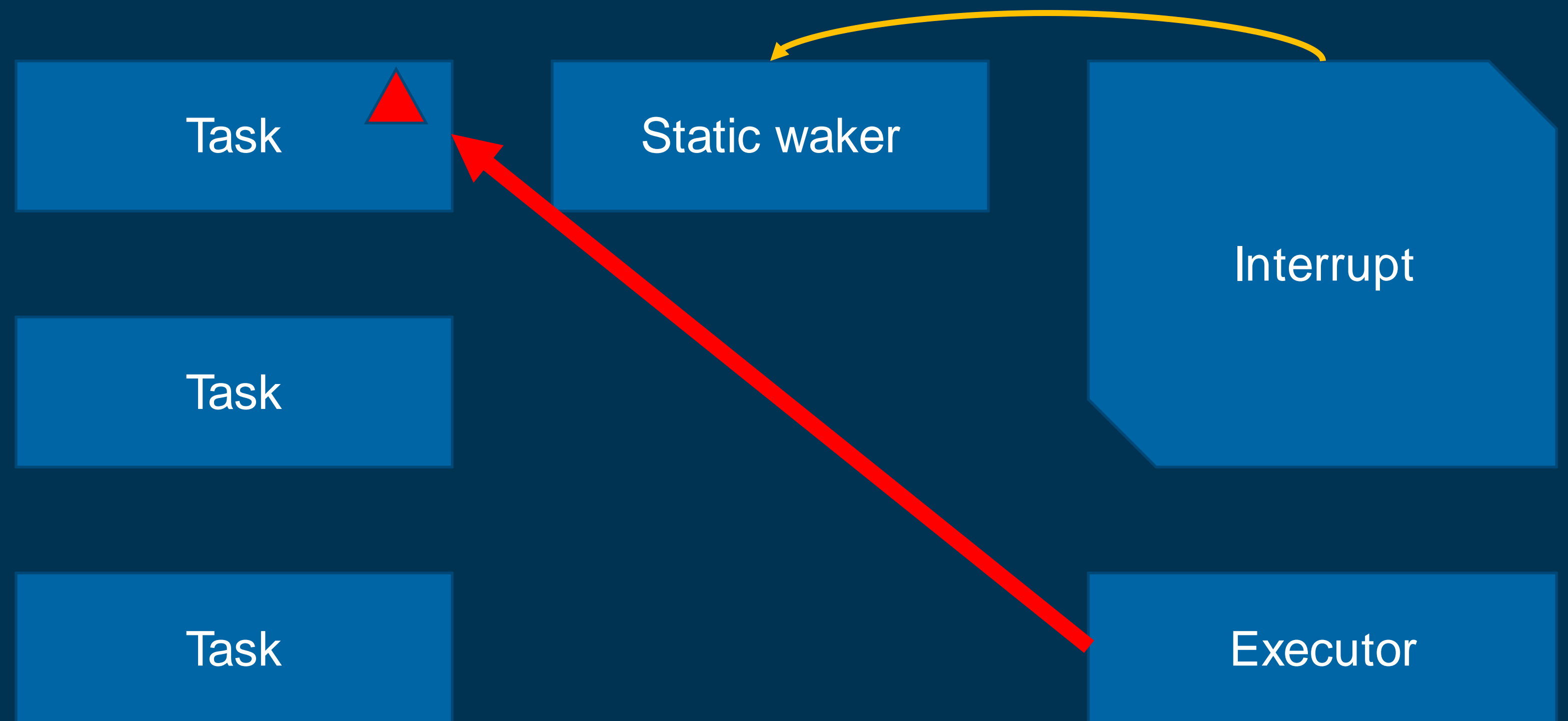
Wakers

- Signal to executor
- Abstract way of setting a flag in the tasks
- Accessed in context parameter of Future



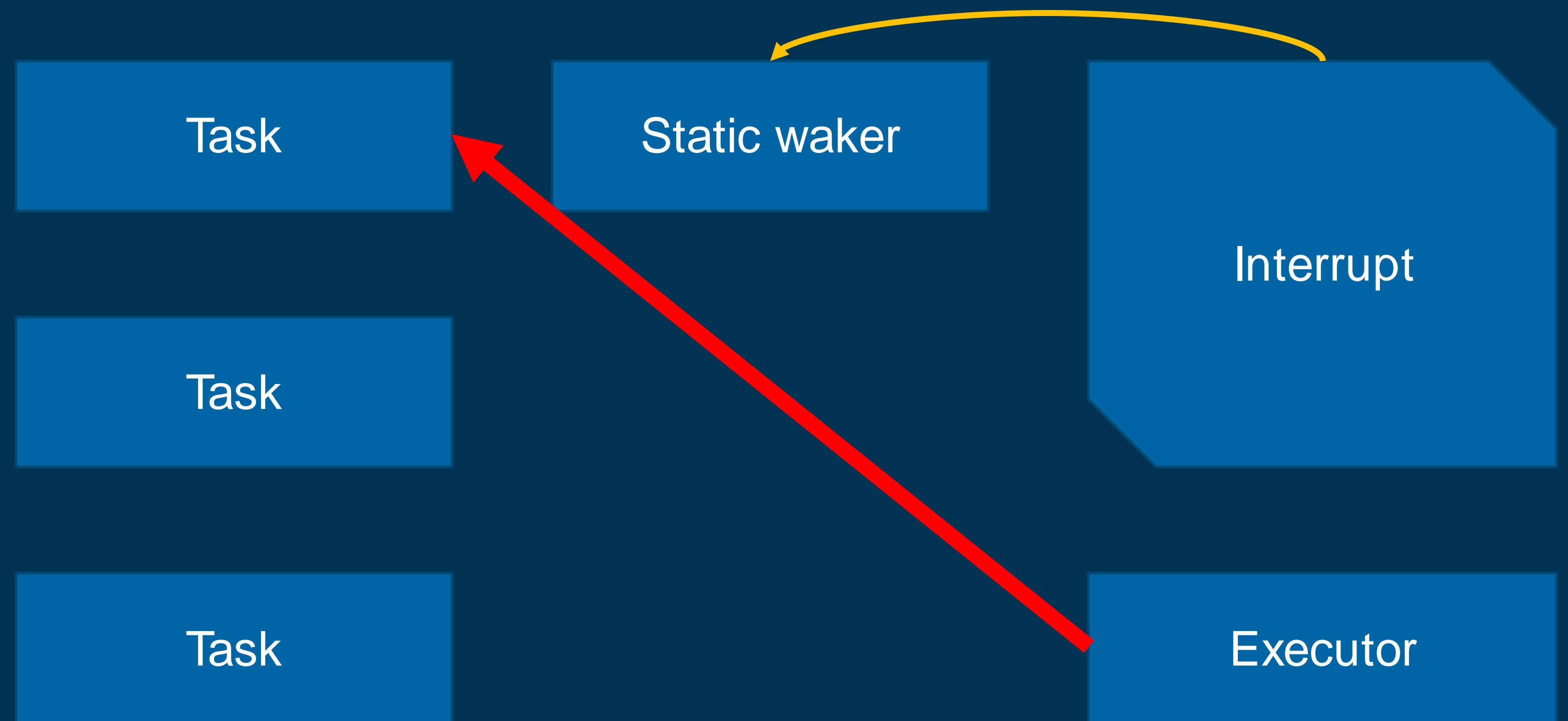
Wakers

- Signal to executor
- Abstract way of setting a flag in the tasks
- Accessed in context parameter of Future



Wakers

- Signal to executor
- Abstract way of setting a flag in the tasks
- Accessed in context parameter of Future



```rust
#[embassy_executor::main]
async fn main(spawner: Spawner) {
    let p = embassy_nrf::init(
        Default::default()
    );

    let mut button = Input::new(
        p.P0_11, Pull::Up
    );
    let mut led = Output::new(
        p.P0_12,
        Level::Low,
        OutputDrive::Standard
    );

    loop {
        button.wait_for_any_edge().await;
        led.set_level(button.get_level());
    }
}
```

Real embassy code

```rust
/// Wait for the pin to undergo any transition, i.e low to high OR high to low.
pub async fn wait_for_any_edge<'self>(&'self mut self) {
    if self.is_high() {
        self.pin.conf().modify(|_, w: &mut W| w.sense().low());
    } else {
        self.pin.conf().modify(|_, w: &mut W| w.sense().high());
    }
    PortInputFuture::new(&mut self.pin).await
}
```

Real embassy code

```rust
#[must_use = "futures do nothing unless you `.await` or poll them"]
1 implementation
pub(crate) struct PortInputFuture<'a> {
    pin: PeripheralRef<'a, AnyPin>,
}

impl<'a> Future for PortInputFuture<'a> {
    type Output = ();

    fn poll<'cx>(self: core::pin::Pin<&mut Self>, cx: &'cx mut Context<'_>) -> Poll<Self::Output> {
        PORT_WAKERS[self.pin.pin_port() as usize].register(cx.waker());

        if self.pin.conf().read().sense().is_disabled() {
            Poll::Ready(())
        } else {
            Poll::Pending
        }
    }
}
```

Real embassy code

```rust
#[cfg(any(feature = "nrf52833", feature = "nrf52840"))]
const PIN_COUNT: usize = 48;
#[cfg(not(any(feature = "nrf52833", feature = "nrf52840")))]
const PIN_COUNT: usize = 32;

#[allow(clippy::declare_interior_mutable_const)]
const NEW_AW: AtomicWaker = AtomicWaker::new();
static CHANNEL_WAKERS: [AtomicWaker; CHANNEL_COUNT] = [NEW_AW; CHANNEL_COUNT];
static PORT_WAKERS: [AtomicWaker; PIN_COUNT] = [NEW_AW; PIN_COUNT];
```

```rust
unsafe fn handle_gpiote_interrupt() {
    let g: &RegisterBlock = regs();

    for i: usize in 0..CHANNEL_COUNT {
        if g.events_in[i].read().bits() != 0 {
            g.intenclr.write(|w: &mut W| w.bits(1 << i));
            CHANNEL_WAKERS[i].wake();
        }
    }

    if g.events_port.read().bits() != 0 {
        g.events_port.write(|w: &mut W| w);

        #[cfg(any(feature = "nrf52833", feature = "nrf52840"))]
        let ports: &[&RegisterBlock; 2] = &[&*pac::P0::ptr(), &*pac::P1::ptr()];
        #[cfg(not(any(feature = "nrf52833", feature = "nrf52840")))]
        let ports = &[&*pac::P0::ptr()];

        for (port: usize, &p: &RegisterBlock) in ports.iter().enumerate() {
            let bits: u32 = p.latch.read().bits();
            for pin: u32 in BitIter(bits) {
                p.pin_cnf[pin as usize].modify(|_, w: &mut W| w.sense().disabled());
                PORT_WAKERS[port * 32 + pin as usize].wake();
            }
            p.latch.write(|w: &mut W| w.bits(bits));
        }
    }
} fn handle_gpiote_interrupt
```

Real embassy code

```rust
/// Wake a task by `TaskRef`.
///
/// You can obtain a `TaskRef` from a `Waker` using [`task_from_waker`].
pub fn wake_task(task: TaskRef) {
    critical_section::with(|cs: CriticalSection| {
        let header: &TaskHeader = task.header();
        let state: u32 = header.state.load(order: Ordering::Relaxed);

        // If already scheduled, or if not started,
        if (state & STATE_RUN_QUEUED != 0) || (state & STATE_SPAWNED == 0) {
            return;
        }

        // Mark it as scheduled
        header.state.store(val: state | STATE_RUN_QUEUED, order: Ordering::Relaxed);

        // We have just marked the task as scheduled, so enqueue it.
        unsafe {
            let executor: &Executor = header.executor.get().unwrap_unchecked();
            executor.enqueue(cs, task);
        }
    })
}
```

Real embassy code

```rust
pub fn run(&'static mut self, init: impl FnOnce(Spawner)) -> ! {
    init(self.inner.spawner());

    loop {
        unsafe {
            self.inner.poll();
            asm!("wfe");
        };
    }
}
```

Real embassy code

```
#[must_use = "futures do nothing unless you `.await` or poll them"]
1 implementation
pub(crate) struct PortInputFuture<'a> {
    pin: PeripheralRef<'a, AnyPin>,
}

impl<'a> Future for PortInputFuture<'a> {
    type Output = ();

    fn poll<'cx>(self: core::pin::Pin<&mut Self>, cx: &'cx mut Context<'_>) -> Poll<Self::Output> {
        PORT_WAKERS[self.pin.pin_port() as usize].register(cx.waker());

        if self.pin.conf().read().sense().is_disabled() {
            Poll::Ready(())
        } else {
            Poll::Pending
        }
    }
}
```

Real embassy code

```rust
/// Wait for the pin to undergo any transition, i.e low to high OR high to low.
pub async fn wait_for_any_edge<'self>(&'self mut self) {
    if self.is_high() {
        self.pin.conf().modify(|_, w: &mut W| w.sense().low());
    } else {
        self.pin.conf().modify(|_, w: &mut W| w.sense().high());
    }
    PortInputFuture::new(&mut self.pin).await
}
```

```rust
#[embassy_executor::main]
async fn main(spawner: Spawner) {
    let p = embassy_nrf::init(
        Default::default()
    );

    let mut button = Input::new(
        p.P0_11, Pull::Up
    );
    let mut led = Output::new(
        p.P0_12,
        Level::Low,
        OutputDrive::Standard
    );

    loop {
        button.wait_for_any_edge().await;
        led.set_level(button.get_level());
    }
}
```

And sorry for the headache