Async Rust in embedded systems with Embassy

Dario Nieuwenhuis
@dirbaio

Who is this guy?

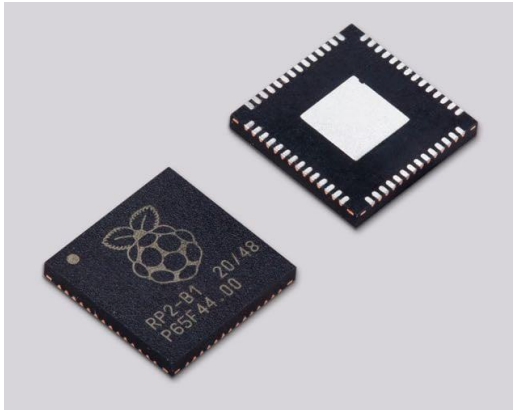
Dario Nieuwenhuis

@dirbaio

- CTO at Akiles
- Maintainer of the Embassy project

Embedded systems

- Microcontrollers
- ARM (M-profile), RiscV, others.
- No OS (unless you BYO)
- 4kB - 256kB of RAM
- 16kB - 1024kB of FLASH
- No `alloc`, ideally



In C you'd usually use an RTOS

RTOS = Real Time Operating System

- Zephyr
- FreeRTOS
- Apache MyNewt
- ChibiOS
- Contiki
- RIOT

In C you'd usually use an RTOS

RTOS = Real Time Operating System

- Zephyr
- FreeRTOS
- Apache MyNewt
- ChibiOS
- Contiki
- RIOT

They give you

- Threads (called "tasks"!)
- Semaphores, mutexes, channels..
- Drivers, I/O
- Networking
- Files

In C you'd usually use an RTOS

RTOS = Real Time Operating System

- Zephyr
- FreeRTOS
- Apache MyNewt
- ChibiOS
- Contiki
- RIOT

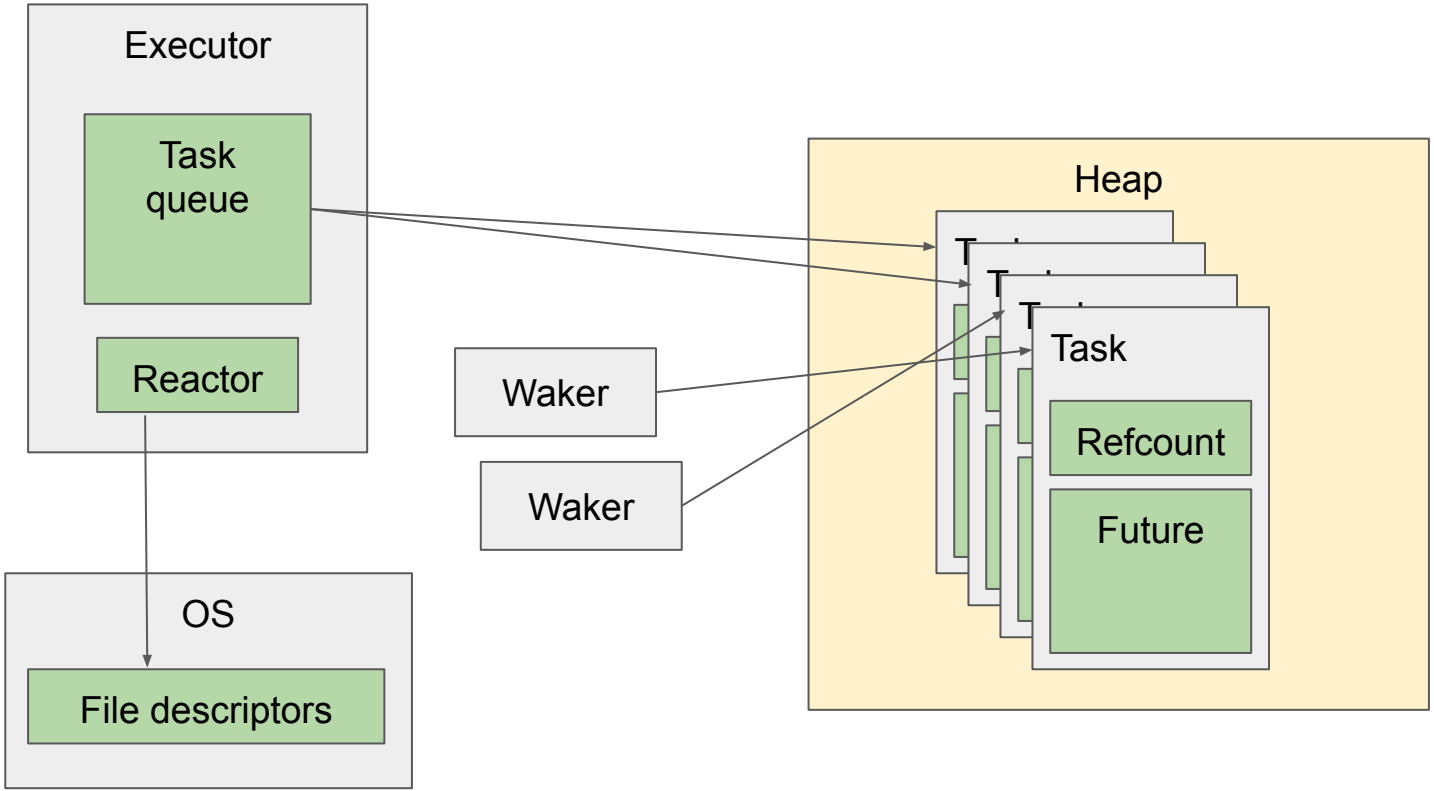
They give you

- Threads (called "tasks"!)
- Semaphores, mutexes, channels..
- Drivers, I/O
- Networking
- Files

Problem: no mature RTOS in Rust

Wouldn’t it be cool to use async instead?

Standard Linux async



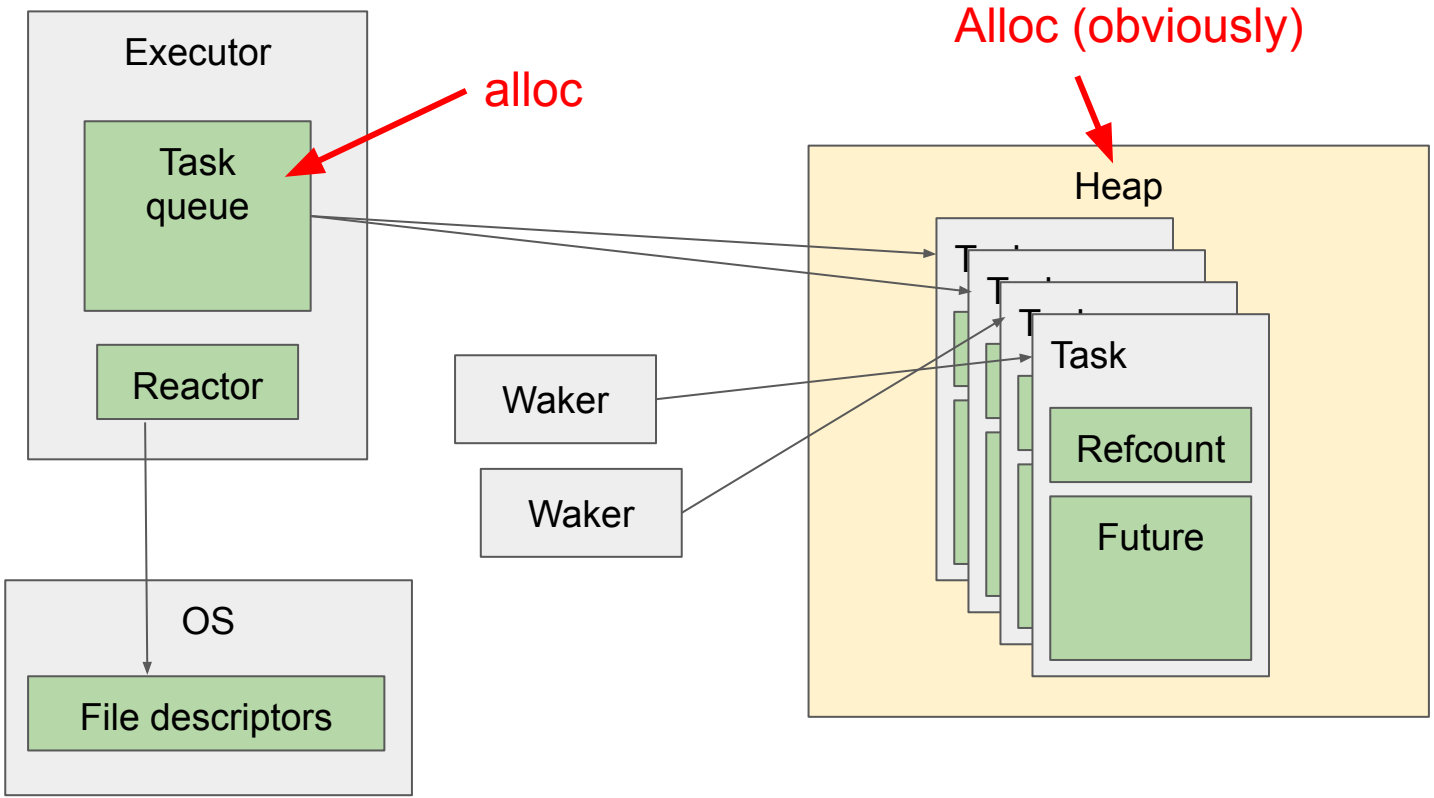
Standard Linux async

```
loop {
    Poll woken tasks
    Wait on epoll() with FDs from reactor
    Wake tasks with FDs that became ready
}
```

Challenges for embedded async

1. No `alloc`!
2. No OS, no file descriptors, no epoll!

Challenge #1: no alloc

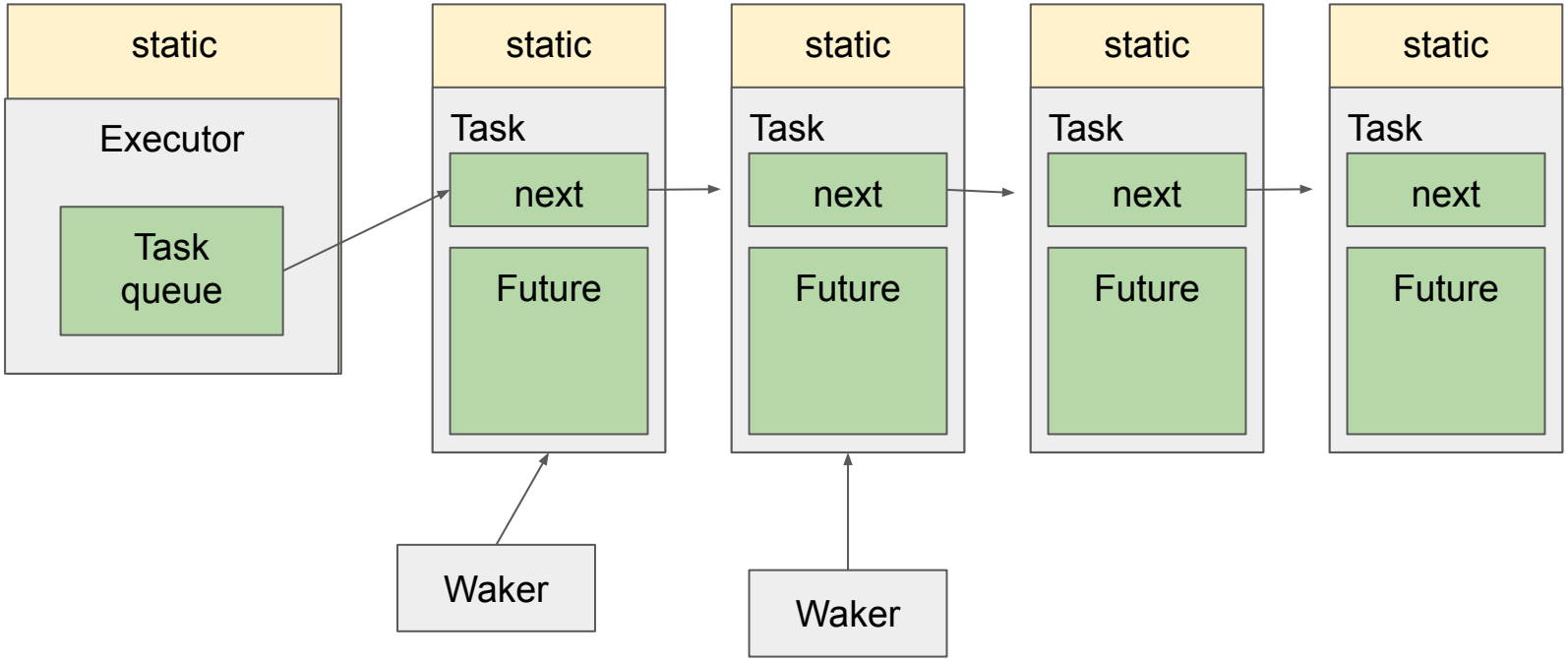


Challenge #1: no alloc

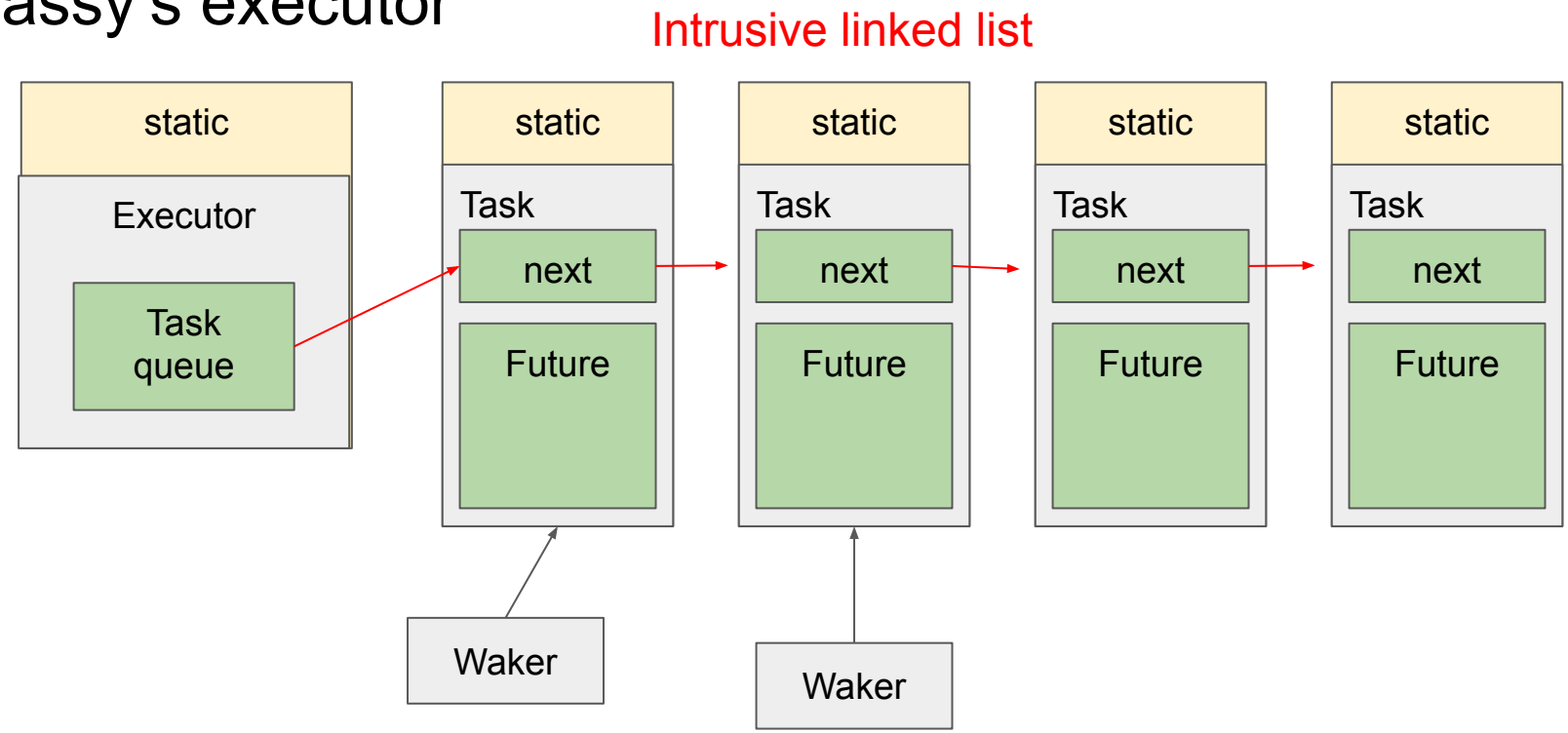
- Can’t heap-allocate tasks
- Must avoid dangling pointers from wakers -> can’t stack-allocate either.

=> Statically allocate tasks

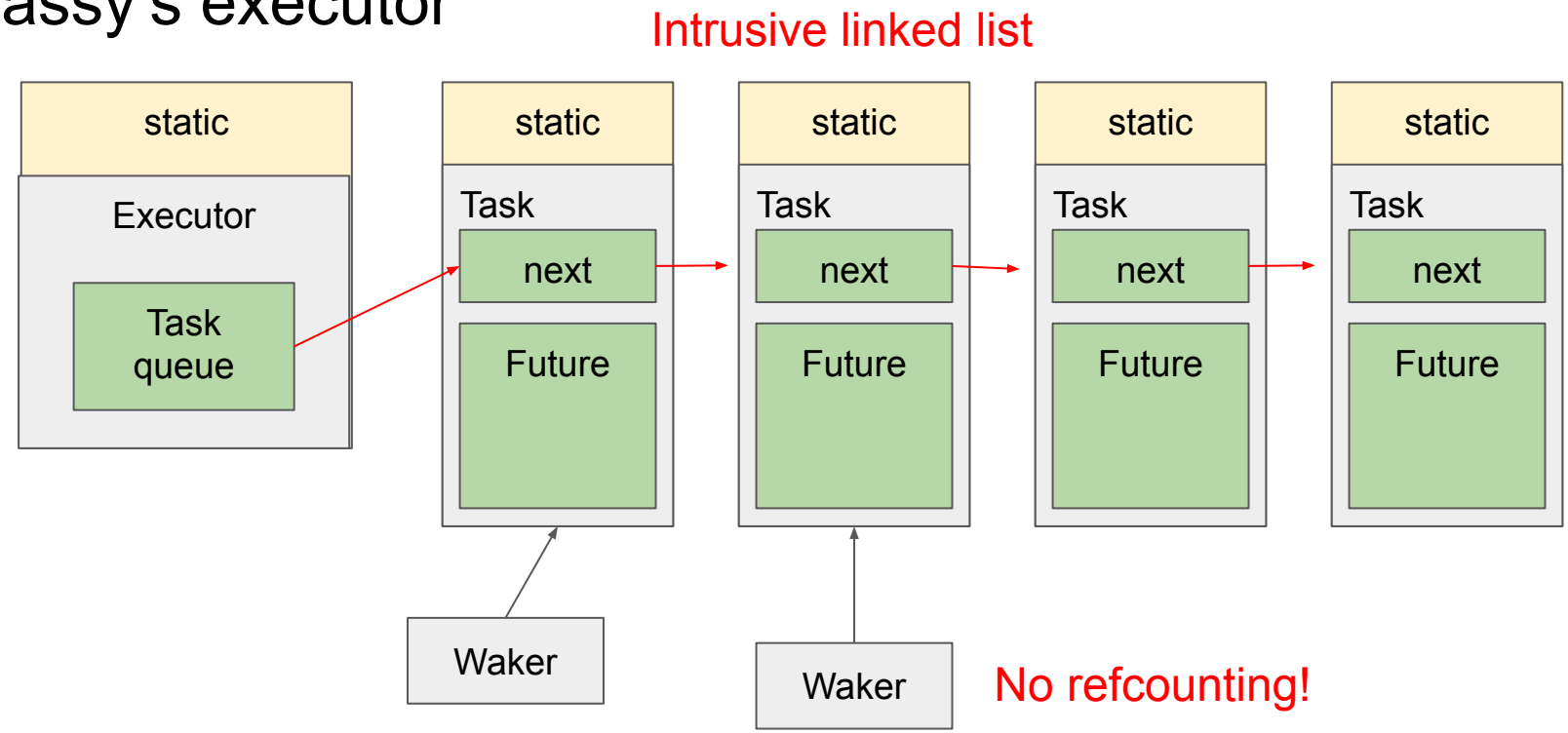
Embassy's executor



Embassy's executor



Embassy's executor



Statically allocating tasks

```
#[embassy_executor::task]
async fn my_task() {
    loop {
        info!("tick");
        Timer::after_secs(1).await;
    }
}

executor.spawn(my_task())
```

Statically allocating tasks

```
#[embassy_executor::task]
async fn my_task() {
    loop {
        info!("tick");
        Timer::after_secs(1).await;
    }
}

executor.spawn(my_task())
```

Statically allocating tasks

```rust
#[embassy_executor::task]
async fn my_task() {
    loop {
        info!("tick");
        Timer::after_secs(1).await;
    }
}

executor.spawn(my_task())
```

Expands to

```rust
static MY_TASK: Task<F> = Task::new();

async fn my_task_inner() {
    loop {
        info!("tick");
        Timer::after_ticks(13000).await;
    }
}

fn my_task() -> SpawnToken {
    MY_TASK.init(my_task_inner())
}
```

Type-alias impl Trait

```
static MY_TASK: Task<F> = Task::new();

async fn my_task_inner() {
    loop {
        info!("tick");
        Timer::after_ticks(13000).await;
    }
}

fn my_task() -> SpawnToken {
    MY_TASK.init(my_task_inner())
}
```

What's this?

```
let fut: ????? = my_task_inner();
```

Type-alias `impl Trait`

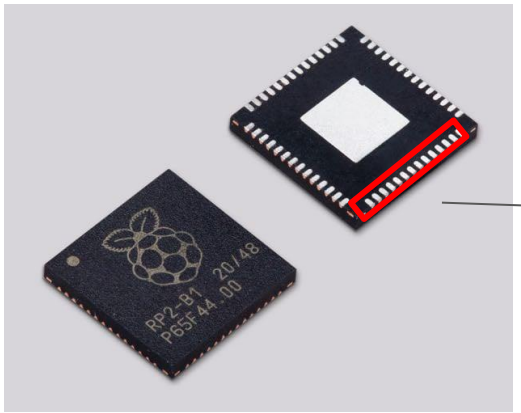
```
#![feature(type_alias_impl_trait)]
type MyTaskFuture = impl Future;

static MY_TASK: Task<MyTaskFuture> = Task::new();

fn my_task() -> SpawnToken {
    MY_TASK.init(my_task_inner())
}
```

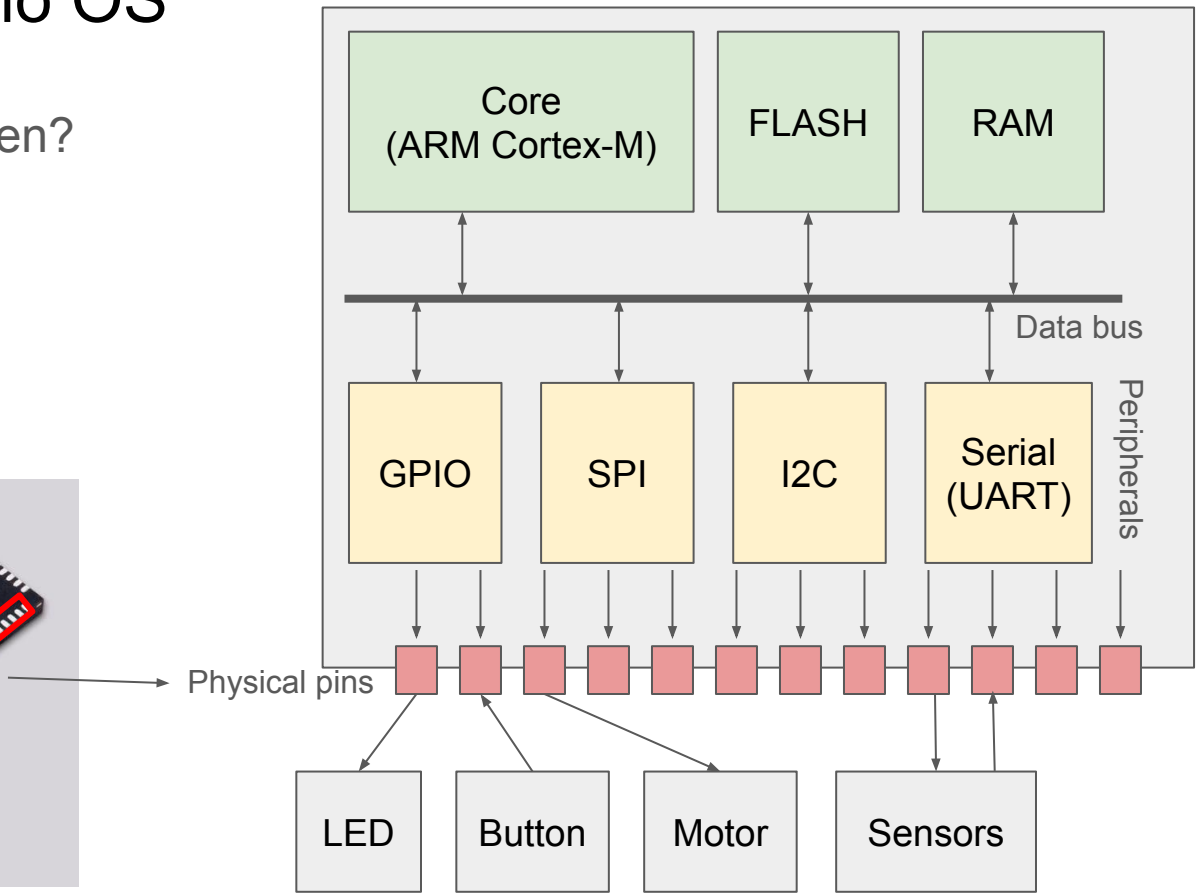
Nightly-only :(

Compiler infers the type of MyTaskFuture here

Challenge #2: no OS

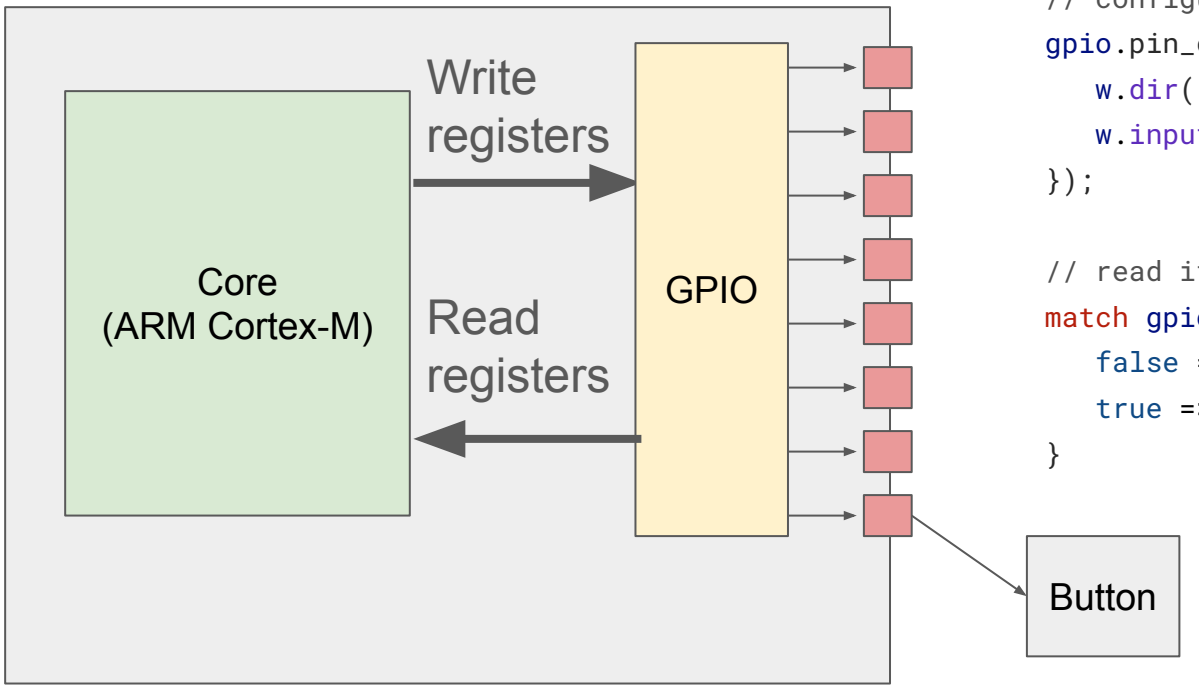
How do you do I/O then?





Challenge #2: no OS

A peripheral in detail: MMIO

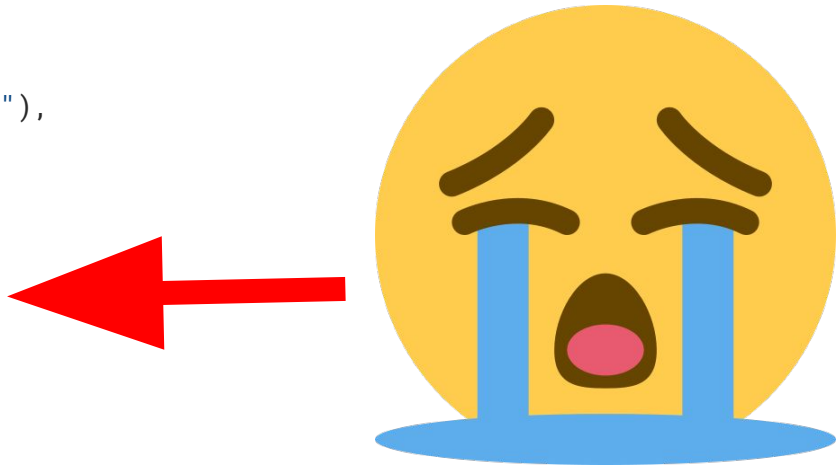



```
// configure the pin as input
gpio.pin_cnf[5].write(|w| {
   w.dir().input();
   w.input().connect()
});

// read it
match gpio.in_.read().pin5().bit() {
   false => info!("button is not pressed"),
   true => info!("button is pressed"),
}
```

Disclaimer: lots of important stuff omitted from code snippets for clarity, they won't work as-is!

So far so good, but….

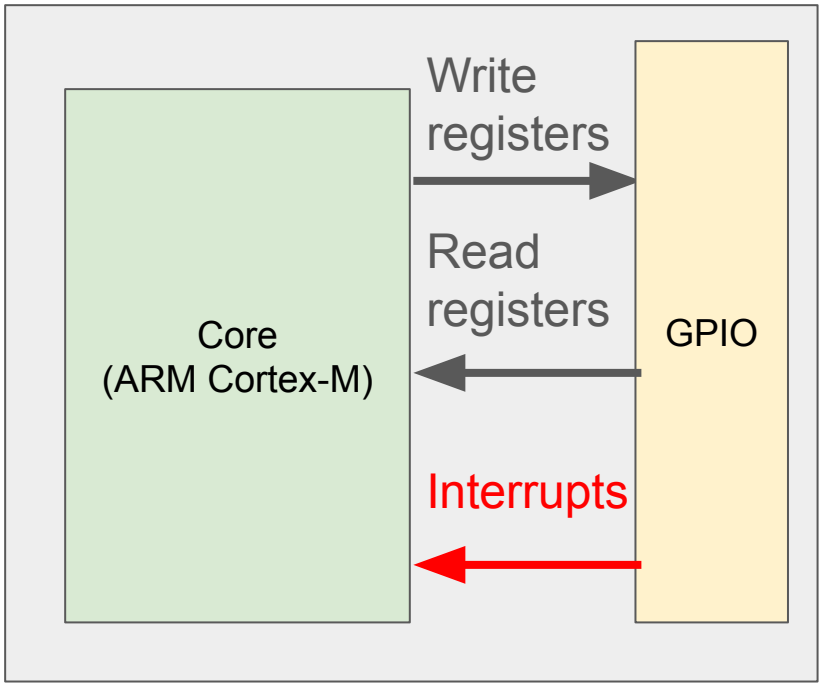
```
// configure the pin as input
gpio.pin_cnf[5].write(|w| {
   w.dir().input();
   w.input().connect()
});

// read it
match gpio.in_.read().pin5().bit() {
   false => info!("button is not pressed"),
   true => info!("button is pressed"),
}

// Wait for button press
while !gpio.in_.read().pin5().bit() {}
```



Disclaimer: lots of important stuff omitted from code snippets for clarity, they won't work as-is!

Challenge #2: no OS

A peripheral in detail: MMIO + interrupts




```
#[interrupt]
fn GPIO() {
    // check if an event happened!
    if gpio.events_port.read().bits() != 0 {
        // acknowledge it
        gpio.events_port.reset();
        info!("button pressed!")
    }
}

// configure pin as input
gpio.pin_cnf[5].write(|w| {
    w.dir().input();
    w.input().connect();
    w.sense().high() // configure interrupts!
});

// we can go do other things, the peripheral will
interrupt us.
```

Disclaimer: lots of important stuff omitted from code snippets for clarity, they won't work as-is!

Let’s asyncify it!

```
struct InputFuture;
impl Future for InputFuture {
    type Output = ();
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<()> {
        WAKER.register(cx.waker());
        if gpio.in_.read().pin5().bit() {
            Poll::Ready(())
        } else {
            Poll::Pending
        }
    }
}
```

```
static WAKER: AtomicWaker = AtomicWaker::new();

#[interrupt]
fn GPIO() {
    if gpio.events_port.read().bits() != 0 {
        gpio.events_port.reset();
        WAKER.wake();
    }
}
```

Disclaimer: lots of important stuff omitted from code snippets for clarity, they won’t work as-is!

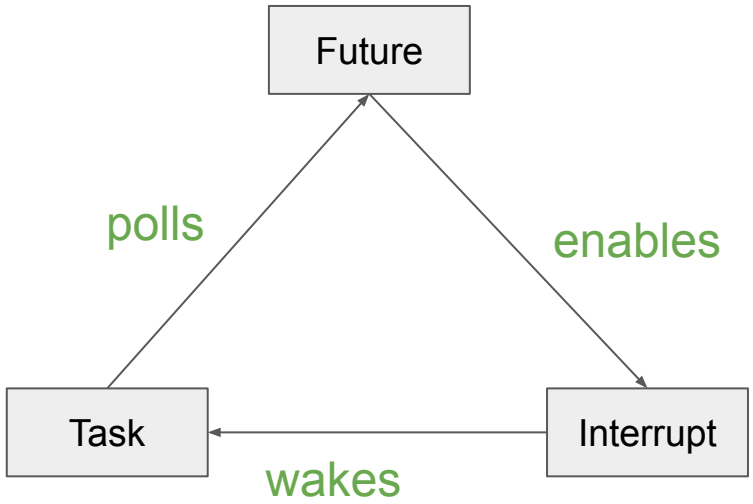
Let’s asyncify it!

```
struct InputFuture;
impl Future for InputFuture {
    type Output = ();
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<()> {
        WAKER.register(cx.waker());
        if gpio.in_.read().pin5().bit() {
            Poll::Ready(())
        } else {
            Poll::Pending
        }
    }
}
```

```
static WAKER: AtomicWaker = AtomicWaker::new();

#[interrupt]
fn GPIO() {
    if gpio.events_port.read().bits() != 0 {
        gpio.events_port.reset();
        WAKER.wake();
    }
}
```

And BOOM: `InputFuture.await;`

Disclaimer: lots of important stuff omitted from code snippets for clarity, they won’t work as-is!


Future
polls
enables
Task
Interrupt
wakes

Cool thing #1

No reactor!

No ecosystem fragmentation, you can mix executor and I/O driver crates.

Cool thing #2

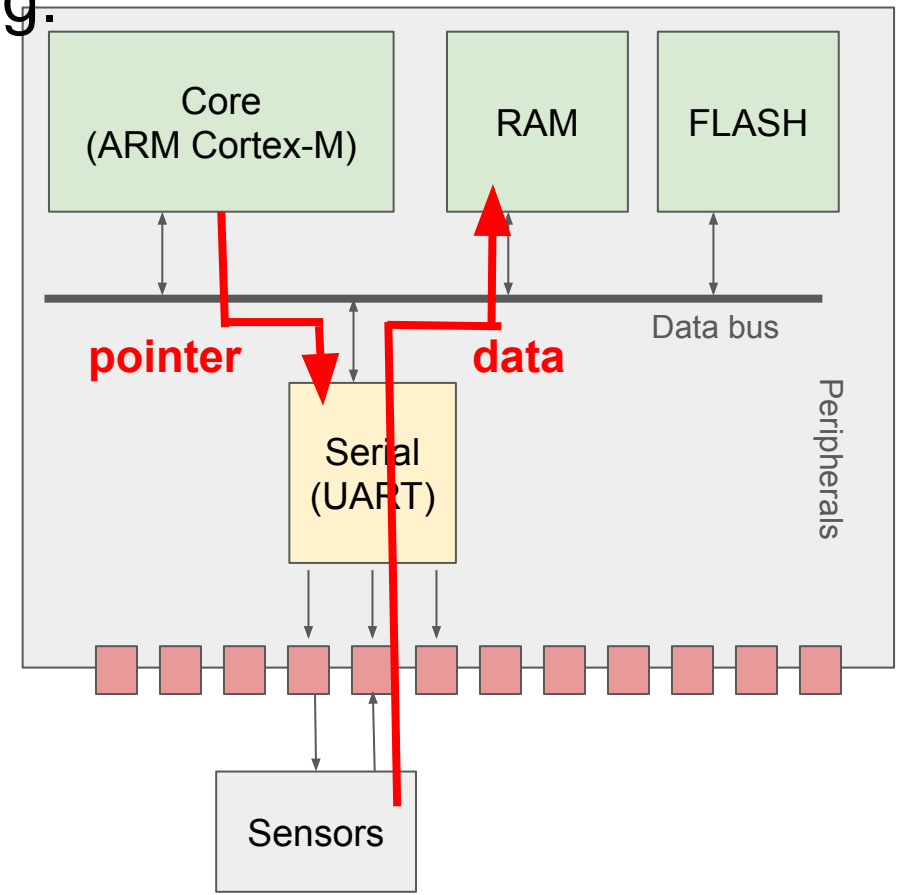
No OS needed!

We can use async *instead of* an RTOS. Not on top of it.

Problem: DMA and leaking.

DMA = Direct Memory Access

- You give a pointer to the peripheral
- It writes the data to RAM for you.



Problem: DMA and leaking.

```
let mut buf = [0u8; 256];
serial.read(&mut buf).await;  // borrowed buffer.
```

Problem: DMA and leaking.

```
let mut buf = [0u8; 256];
with_timeout(d, serial.read(&mut buf)).await;
```

Problem: DMA and leaking.

```
let mut buf = [0u8; 256];
let mut f = serial.read(&mut buf);
poll_once(f).await; // starts DMA
mem::forget(f); // releases the borrow, but doesn't stop DMA
return; // deallocates buf, DMA now corrupts memory
```

Problem: DMA and leaking.

Current solution:

Just don’t do that ¯_(ツ)_/¯

Problem: DMA and leaking.

Current solution:

Just don’t do that ¯_(ツ)_/¯

- Same problem in io_uring
- Use owned buffers (like `Vec`) -> can’t because no alloc.
- Use inline buffers (`[u8; 256]`) -> causes bloat due to moves.
- Use static buffers -> unergonomic, requires unsafe for static mut, or overhead of “locked” flags.
- Possible solution: `trait Leak` / `trait Forgettable`

Embassy current status

- The executor
- Real-time preemption (cooperative in priority level, preemptive across levels)
- join, select, Mutex, Channel…
- std-like Instant, Duration, sleep, timers.
- TCP/IP networking
- USB
- Bluetooth
- Hardware Abstraction Layers for nRF, STM32, RP2040 MCUs.
    - Though you can use any! Espressif's ESP-HAL has great async support, for example.

Embassy in the wild

akiles

- Bluetooth
- Ethernet
- WiFi
- Mobile internet
- Mesh networking
- RFID
- Low-power

Thank you!

Check out Embassy:
https://github.com/embassy-rs/embassy
https://embassy.dev