

# **Rust embedded at Espressif**

**Scott Mabin**

# What I'll cover today



- What is an embedded system?
- Why Rust for embedded systems?
- `async` + embedded Rust
- Espressif's offerings

# What is an embedded system?



- A system created with a specific purpose
- Usually has some real-time computing and resource constraints

# What do we mean by real-time?



- Reacting to system events with as little latency as possible
- Usually with hard deadlines for response times
- Typically measured in the order of a few milliseconds

# Resource constraints



- Kilobytes of RAM, instead of Gigabytes found on modern computers
- 10's of Megahertz CPU frequencies, instead of the Gigahertz frequencies and multiple cores on modern computers

# Executing & Debugging programs



- Program(s) are stored in flash memory
- They need to be flashed from a host machine
- Debugging happens remotely

# Why Rust for embedded?



- Memory safety is even more important, most embedded systems do not have an MMU
- Ownership: Model physical hardware peripherals as singletons

# Why Rust for embedded - Tooling



- cargo, no more Makefiles!
- Package management
  - Interface trait crates like `embedded-hal`
  - Non-allocating data structure crates like `heapless`
- probe-rs, a debugging toolkit for embedded devices
- Wokwi - Simulating embedded systems in the browser



# Why Rust for embedded - `async`



- Works without alloc
- Provides single-threaded concurrency (multitasking)
  - Can run on a single stack, great for resource-constrained microcontrollers
- Write asynchronous code that has similar ergonomics and readability as synchronous code

# How does `async` work?



You can only `await` something that implements the `Future` trait.

The `Future` trait has one required method, `poll` which returns either `Poll::Ready(_)` if the asynchronous operation is complete, or `Poll::Pending` if it needs to be polled again later.

# When to poll?



You *could* just `poll` the future in a hot loop, but this is not very efficient and will block other `async` operations from running.

```
while let Poll::Pending = some_fut.poll() {  
    // 100% CPU used here waiting for `Poll::Ready(_)`  
}
```

We'd like to do other things until the `async` operation is ready. This is where the `Waker` concept is introduced.

# The Waker



A **Waker** is something that can be used to signal that a future should be polled again.

**wake** ing a **Waker** can happen from anywhere, some examples being a call-back function from a completed operation or just another function.

In many embedded cases, an interrupt handler is used.

# The embassy-executor



A popular executor for embedded is the embassy projects executor.

In Embassy, tasks are statically allocated to avoid the need for an allocator. The `#[embassy_executor::task]` macro takes care of this for us.

```
#[embassy_executor::task]
async fn task() {
    loop {
        Timer::after(Duration::from_millis(100)).await;
    }
}
```

# A trivial embedded scenario



Read the state of a button connected to a pin. Depending on whether the button is pressed, turn on or off an LED connected to another pin.

# Blocking



```
#[esp_riscv_rt::entry]
fn main() {
    let io = IO::new(peripherals.GPIO, peripherals.IO_MUX);
    let mut led = io.pins.gpio7.into_push_pull_output();
    let button = io.pins.gpio9.into_pull_down_input();

    loop {
        if button.is_high() {
            led.set_high();
        } else {
            led.set_low();
        }
    }
}
```

# Blocking



- Repeatedly checks for a condition to be true before proceeding
- Simple program flow
- Easy to write yet highly inefficient, causing 100% utilization of the CPU.



# Interrupt - main



```
static BUTTON: Mutex<RefCell<Option<Gpio9>> = Mutex::new(RefCell::new(None));
static STATE: AtomicBool = AtomicBool::new(false);

#[esp_riscv_rt::entry]
fn main() {
    let mut led = io.pins.gpio7.into_push_pull_output();
    let mut button = io.pins.gpio9.into_pull_down_input();
    button.listen(Event::FallingEdge);
    button.listen(Event::RisingEdge);
    critical_section::with(|cs| BUTTON.borrow_ref_mut(cs).replace(button));

    loop {
        if STATE.load(Ordering::SeqCst) {
            led.set_high();
        } else {
            led.set_low();
        }
        sleep(); // wait for interrupt here
    }
}
```

# Interrupt - handler



```
#[interrupt]
fn GPIO() {
    critical_section::with(|cs| {
        let button = BUTTON.borrow_ref_mut(cs).as_mut().unwrap();
        button.clear_interrupt();
        if button.is_high() {
            STATE.store(true, Ordering::SeqCst);
        } else {
            STATE.store(false, Ordering::SeqCst);
        }
    });
}
```

# Interrupt



- Hardware signal that interrupts the normal flow of programs execution
- Allows sleeping in the main thread
- More code is required, harder to write and read the code
- Only works for hardware events

```
#[embassy_executor::main(entry = "esp_riscv_rt::entry")]
async fn main(spawner: embassy_executor::Spawner) {
    let io = IO::new(peripherals.GPIO, peripherals.IO_MUX);
    let mut led = io.pins.gpio7.into_push_pull_output();
    let button = io.pins.gpio9.into_pull_down_input();

    loop {
        button.wait_for_any_edge().await;
        if button.is_high() {
            led.set_high();
        } else {
            led.set_low();
        }
    }
}
```

- Structurally, it's similar to a `busy loop` but with `async`, each `.await` point allows the CPU to do something else, or even sleep to save power.
- Uses interrupts behind the scenes but the user doesn't have to worry about setting them up.
- We can chain hardware driven `async` code with normal async code

# Demo



# Espressif's chip offerings



- RISC-V based ESP32-Cx, ESP32-Hx & ESP32-Px series
- Xtensa based ESP32, ESP32-Sx series
- WiFi
- Bluetooth
- IEEE 802.15.4
- Single-core and dual-core options available

# Espressif's Rust offerings



- [esp-rs/esp-hal](#) - `no_std` peripheral drivers, UART, I2C, SPI etc
- [esp-rs/esp-wifi](#) - `no_std` WiFi and Bluetooth drivers
- [esp-rs/esp-ieee802154](#) - `no_std` ieee802154 radio driver



# Bonus - STD support



- It's possible to use the Rust standard library with Espressif chips, we have a port upstream called `espidf`.
- It's based on [esp-idf](#), the C SDK which exposes a newlib environment which the Rust standard library can be built on top of.

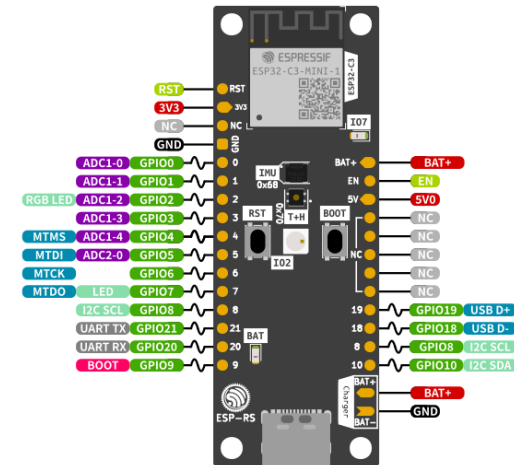
# A Rust development kit



Rust Board ESP32-C3



- The [esp-rs/esp-rust-board](#)



# Books, resources and trainings



- Our own mdbook for getting started with Rust on Espressif chips [esp-rs/book](#)
- We have a training pack created by Ferrous Systems available for free using this board [esp-rs/std-training](#).
- We also have a no\_std variant using the same training materials, if the no\_std option is more appealing [esp-rs/no\\_std-training](#)
- The [Rust embedded book](#) from the Rust embedded working group