Async Rust for embedded systems

Scott Mabin, Juraj Sadel

Goals of this presentation

- Explore how async Rust works
- Apply it to an embedded context
- Create a working application using async Rust

Why use async?

ESPRESSIF DevCon23

- Single-threaded concurrency (multitasking), doing multiple things at once without needing threads
- Can run on a single stack, great for microcontrollers
- Composability of async/await instead of manually writing state machines.

A refresher on async Rust syntax

Last year's talk: EDC22 Day 1 Talk 7: Rust on Espressif chips.

`async` in Rust adds two new keywords to the language, `async` & `await` , where

- `async` defines a block or function to be asynchronous
- `await` defines yield points *within* an `async` block or function.

`async` & `.await`

- Building blocks for async blocks, closures, and functions that can be paused and yield control back to the caller
- `async` code returns a value that implements the `Future` trait

```rust
pub async fn say_hello(uart0: &mut Serial<UART0>) {
   let message = "Hello World!";
   uart0.write_bytes(message).await; // yield point here!
   let message = "Goodbye";
   uart0.write_bytes(message).await; // another yield point here!
}
```

A simple scenario

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

Busy Loop

- Repeatedly checks for a condition to be true before proceeding
- Simple to implement, very inefficient

```rust
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_up_input();

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

Interrupt - main

ESPRESSIF DevCon23

```
static BUTTON: Mutex<RefCell<Option<Gpio9<Input<PullDown>>>>> = Mutex::new(RefCell::new(None));
static STATE: AtomicBool = AtomicBool::new(false);
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));
    interrupt::enable(peripherals::Interrupt::GPIO, interrupt::Priority::Priority3).unwrap();

    loop {
        if STATE.load(Ordering::SeqCst) {
            led.set_high().unwrap();
        } else {
            led.set_low().unwrap();
        }
        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().unwrap() {
            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

Async

ESPRESSIF DevCon23

```rust
static EXECUTOR: StaticCell<Executor> = StaticCell::new();

fn main() {
    let io = IO::new(peripherals.GPIO, peripherals.IO_MUX);
    let mut output = io.pins.gpio7.into_push_pull_output();
    let input = io.pins.gpio9.into_pull_down_input();
    let executor = EXECUTOR.init(Executor::new());
    executor.run(|spawner| {
        spawner.spawn(toggle(input, output)).ok();
    });
}

async fn toggle(mut input: Gpio9<Input<PullDown>>, mut output: Gpio7<Output<PushPull>>) {
    loop {
        match select(
            input.wait_for_rising_edge().await.unwrap(),
            input.wait_for_falling_edge().await.unwrap(),
        ) {
            Either::First(_) => output.set_high(),
            Either::Second(_) => output.set_low(),
        }
    }
}
```

Async

- 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.

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.

The Future trait

ESPRESSIF DevCon23

```rust
pub trait Future {
    type Output;

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

```rust
enum Poll<T> {
    Ready(T),
    Pending,
}
```

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 an interrupt handler, a call back function or just another function.

Pseudo code `Future` implementation

ESPRESSIF DevCon23

```rust
impl Future for Socket<'_> {
    type Output = Vec<u8>;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        if self.has_data_to_read() {
            // The socket has data -- read it into a buffer and return it.
            Poll::Ready(self.read_buf())
        } else {
            // The socket does not yet have data.
            // Arrange for `wake` to be called once data is available.
            // When data becomes available, `wake` will be called, and the
            // user of this `Future` will know to call `poll` again and
            // receive data.
            self.set_readable_callback(cx);
            Poll::Pending
        }
    }
}
```

How to run futures - Executors

We've covered how futures work, but where do `Poll::Pending` futures yield to? They yield back to the *executor*.

The executor is the mechanism to run futures, it handles the response to a `wake` event and then `poll`'s that future again.

Executor is a general term, there is no trait for them, they can be implemented in various ways and each will have various features and limitations.

Embedded async - embassy

A popular executor for embedded systems is the embassy project. It aims to provide, not just an executor, but a collection of tools and utilities to create effective async applications.

```rust
#[embassy_executor::task]
async fn ping(mut pin: Gpio9<Input<PullDown>>) {
    loop {
        esp_println::println!("Waiting...");
        pin.wait_for_rising_edge().await.unwrap();
        esp_println::println!("Ping!");
        Timer::after(Duration::from_millis(100)).await;
    }
}
```

Async tasks in embassy

Usually, the top-level future is called a task. Within the task, many futures could be `await`ed. In embassy, tasks are statically allocated to avoid the need for an allocator. The `#[embassy_executor::task]` macro takes care of this for us.

Tasks are allowed to have infinite loops, much like a traditional RTOS task, but **MUST** contain at least one `await` point to avoid blocking other tasks.

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

Building an IoT temperature data logger with async

Links & Resources

- The esp-rs book
- esp-rs organisation
- esp-rs roadmap
- rust embedded book