Implementing async APIs for microcontroller peripherals

2024-11-30 :: tags: #async/await #atsamd-hal #embedded #rust

`async/await` support has been a long time coming in the atsamd-hal project. Some time around 2020, I discovered embassy, a comprehensive framework that exploits asynchronous programming on a wide range of microcontrollers. It immediately appealed to me; to my brain, it made complete sense to harness cooperative multitasking on small, single-core systems. It enables you to write seemingly linear and straightforward code, without needing to manually deal with complex finite states machines, and without the drawbacks of threading on a more "traditional" RTOS. Having been involved with atsamd-hal for some time at that point, I started thinking about how we could support async programming in the HAL.

At that time, most async-enabled HALs were tightly integrated into the embassy ecosystem. I had already poured a good deal of effort into atsamd-hal, and didn't necessarily want to start a new HAL from scratch, fragmenting the (already small) community of Rust devs deploying their code on the ATSAMD chip line. I therefore set out to integrate async support directly in atsamd-hal. A first proof of concept was proposed in a PR in October 2022. It was finally merged in November 2024, a little more than two years later.

Nowadays, while most async HALs out there are still maintained by the embassy organization, they're much less coupled to the executor. In fact, most HALs out there will be useable with a multitude of executors, like embassy-executor and rtic - that's also the case for atsamd-hal.

With that bit of history out of the way, let's get cooking!

Understanding async in an embedded context

`async/await` is a fantastic tool to leverage concurrency on single-core systems such as microcontrollers. Some tasks may be offloaded to hardware peripherals, freeing up the CPU to move onto better things. For example, think of a UART waiting on some bytes to arrive on the wire, or a timer that is bound to expire some time in the future. In both these cases, the CPU doesn't have anything useful to do until *something* happens: the peripherals are already doing all the work in parallel.

Instead of busy-waiting until the task is complete (no multitasking), or using RTOS-style threads or plain interrupts and a state machine (*preemptive* multitasking), we can abstract that complexity away to the compiler, and write code that has the appearance of being straightforward and linear.

That's the whole idea of *cooperative* multitasking: let the CPU do some work when it has some work to do. But don't let it block waiting on something to happen! That time could be better utilized doing some more useful work elsewhere, or sleeping to save some power.

If you'd rather first see a complete usage example of an async peripheral before diving into its internals, check this out.

The three building blocks of async/await in Rust

To make cooperative multitasking possible in an embedded context, we will need three things:

1. A Future

Futures are the cornerstone of asynchronous operations in Rust. You might have seen them called a Promise in other languages. Conceptually, a Future represents a value that might not be available yet. It allows us to continue doing more work until the value is needed. Practically, a Future is a trait that looks like this:

```
pub trait Future {
    type Output;

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

It returns Poll, which is a wrapper around the value which we are expecting eventually:

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

Notice that `Future::poll` takes a `Pin<&mut Self>` argument. It behaves similarly to a normal `&mut self`, but with additional restrictions, which we won't cover in this post. If you want to learn more about pinning, check out the Additional Reading section. The method also takes a `&mut Context` argument: we will come back to that in a moment.

The async keyword

An `async` function is syntax sugar for a function which returns a Future:

```
async fn some_function() {
    // ...
}

// Roughly desugars to
fn some_function() -> impl Future<Output = ()> {
    // ...
}
```

A Future can also be constructed by using an async block, which behaves similarly to a closure:

```
fn some_function() -> impl Future<Output = ()> {
    async {
        // ...
    }
}
```

Furthermore, since Future is just a trait, we can simply implement it on an arbitrary struct:

```
use core::future::Future;
use core::pin::Pin;
use core::task::{Context, Poll};

struct MyFuture;

impl Future for MyFuture {
    type Output = ();

    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::
        // When implementing Future directly, it's all about what
        // happens in the poll() method.
    }
}
```

The await keyword

However, Futures aren't enough on its own. A Future does nothing until it is first polled; we need a way to drive that Future to make progress. Enter the `await` keyword:

Awaiting a future boils down to calling `poll()` until the future returns `Poll::Ready`. If it returns `Poll::Pending`, it will be polled again at some later time. Note that a Future is free to panic (but not to display undefined behavior) if it is polled again after it has returned `Poll::Ready`.

See what happens if we forget to await or otherwise poll a Future:

```
async fn some_function() {
   println!("Hello, world!");
}

fn main() {
    some_function();
}
```

```
$ cargo run
(nothing happens)
```

However we do get a compile-time warning:

```
warning: unused implementer of `Future` that must be used
 --> src/main.rs:6:5
  |
6 |     some_function();
  |     ^^^^^^^^^^^^^^^
  |
  = note: futures do nothing unless you `.await` or poll them
  = note: `#[warn(unused_must_use)]` on by default
```

The compiler helpfully reminds us to .await our Future. Alright, let's try that:

```
async fn some_function() {
   println!("Hello, world!");
}

fn main() {
    some_function().await;
}
```

Trying to run this code gives us a compile error:

```
error[E0728]: `await` is only allowed inside `async` functions anc
 --> src/main.rs:6:21
  |
5 | fn main() {
  | --------- this is not `async`
6 |     some_function().await;
  |                     ^^^^^ only allowed inside `async` functio
```

The compiler is telling us that we can't await a Future outside of an async function or block. But how do we turn an synchronous function into an async one? For that we need...

2. An executor

The executor is the runtime responsible for scheduling the tasks to be ran. It's responsible for checking in with tasks when they are ready to make progress, by handing Futures a Waker, which is used to notify the executor that it's ready for some work. It's also responsible for parking tasks that aren't quite ready yet. In the embedded world, the two most popular executors are probably embassy-executor and rtic. In the normal computing world, tokio is the uncontested champion of the async runtime popularity contest, with honorable mentions for async-std and smol. Note that I didn't use the word *executor* here, because a third building block is necessary to make async tasks work:

3. A reactor

The name says it all: it *reacts* to stuff. Their job is to listen to external events, and wake the executor when a task is ready to make progress. Typically, `std` runtimes bundle reactors along with the executor to enable async operations like reading from a file, or waiting on a TCP packet. In the embedded world, we will need to provide those reactors ourselves when building our futures from scratch.

How do we know when a task is ready?

A naive executor could repeatedly poll a future in a busy loop until it returns:

```
async fn some_future() {
    // Do some async things here
}

fn naive_spawn(){
    let future = some_future();

    let result = loop {
        if let Poll::Ready(value) = future.poll(/* what should we
            break value;
        }
    };
}
```

However, you can probably see that this completely defeats the purpose of using cooperative multitasking: our CPU is now completely tied up in repeatedly checking if our task is ready! Therefore, we can't free it up to do other things while we're waiting for this task to complete. So how does the executor know when a task is ready to make progress?

Wakers

As we say earlier, the `Future::poll` method takes a `&mut Context` argument, which itself is a wrapper around a `Waker`. Since the executor is ulimately responsible for polling top-level Futures, it must provide a Waker when calling `poll()`. In return, it expects the Future to call `Waker::wake` (or `wake_by_ref`) when it wants to make more progress. When constructing our Futures, we need some way to wake the Waker when we're ready to move forward. The executor, by design, will **not** poll a Future again after it has returned `Poll::Pending` - unless the Future's Waker has been woken. So we need a way to be preemptively notified when a task has completed, without relying on manually checking for progress...

You guessed it: interrupts!

Conveniently enough, microcontrollers have interrupts! They're perfect for this - they can preempt whatever code is running at the time, wake the Waker, and let the executor know that we're ready to move forward. The interrupt handlers are our reactors - they react to external events (the peripherals finishing their assigned duties), and can wake the executor to signal that progress has been made.

Let's build a Future from scratch -

microcontroller style

Let's now dive into the specifics of implementing async support for a peripheral. The code presented is loosely based on atsamd-hal. As an example, we will work with the SAMD21 External Interrupt Controller (or in simpler terms, GPIO interrupts), as it's probably the easiest peripheral to understand from a hardware perspective.

Our goal is to start with an `ExtInt`, which represents a GPIO pin capable of generating interrupts based on its input state, and end up with the following `wait()` method, as seen in the async usage example:

```rust
pub enum Sense {
    /// No detection
    None,
    /// Rising edge
    Rise,
    /// Falling edge
    Fall,
    /// Any edge
    Both,
    /// Generate interrupts as long as the
    /// pin is in High state
    High,
    /// Generate interrupts as long as the
    /// pin is in Low state
    Low,
}

impl ExtInt
where
    Self: embedded_hal::digital::InputPin
{
    /// Wait for the specified input state to occur on the pin
    pub async fn wait(&mut self, sense: Sense) {
        // ...
    }
}
```

To get there, there are multiple steps we need to consider:

1. We must figure out a way to tie a particular peripheral to its interrupt handler. We want the peripheral to take ownership of the handler - we don't want to rely on the user calling the right functions in the handler, which could be very error prone.
2. Then, inside our `wait()` method, we must ensure that the (correct) interrupt will be fired when the pin reaches the desired state;
3. Finally, when an interrupt occurs, we must wake the Waker inside the interrupt handler.

1. HAL generalities: declaring interrupt handler bindings

Disclaimer: the concepts shown in this section were heavily inspired by the embassy project.

We want our async peripherals to be as self-contained as possible. Since for our use case, Wakers are intimately tied to interrupt handlers, we'd like to provide users with a way to pass ownership of the handlers to the peripheral. That way, we can guarantee that the correct code will be ran upon the interrupt firing. At the same time, we can leverage the type system to prove at compile time that the user gave the peripheral ownership of the *correct* handler. Surprising things would happen if a peripheral expects some interrupt to fire, but some other handler is ran instead!

To provide a reusable way to bind an interrupt handler to an interrupt, we create three traits. Note that these traits are generic across the HAL; all peripherals reuse these three trait declarations:

The first one, `InterruptSource`, represents the basic NVIC interrupt operations an async peripheral must be able to perform on an interrupt source:

```
// src/async_hal/interrupts.rs

pub trait InterruptSource: crate::typelevel::Sealed {
    unsafe fn enable();
    fn disable();

    fn unpend();

    fn set_priority(prio: Priority);
}
```

The HAL creates a type for each interrupt source that may be used in an async peripheral, and implements `InterruptSource` for that type. In reality, `InterruptSource` may represent a single interrupt, or multiple interrupts related to a single peripheral; on some chips, there is more than one interrupt per peripheral. The async peripherals in this HAL consolidate all interrupts of a single peripheral into a single handler for simplicity.

The second trait, `Handler`, represents a struct that holds an interrupt handler. Each peripheral that wishes to take ownership of a handler must declare their own types that implement `Handler`. The type parameter `I` lets perhiperals declare which interrupt sources they are able to accept:

```
// src/async_hal/interrupts.rs

pub trait Handler<I: InterruptSource>: Sealed {
    /// The actual interrupt handler
    unsafe fn on_interrupt();
}
```

Finally, `Binding` is a marker trait used to statically prove that an interrupt source has been tied to a specific handler which can accept it:

```
// src/async_hal/interrupts.rs

pub unsafe trait Binding<I: InterruptSource, H: Handler<I>> {}
```

Registering interrupts: the bind_interrupts macro

Interrupt handlers must de declared in the final binary crate[1]. This means we need a user-facing mechanism to bind an interrupt source to the correct interrupt handler. The easiest way (for HAL authors and for the end-users) to do this is by providing a macro that does the heavy lifting. This `bind_interrupts` macro does 3 things:

- Takes a user-provided link between an interrupt source and a peripheral Handler. Even though peripherals can statically check that the correct source is bound to the correct hander, this step must still be done manually.
- Create a zero-sized struct that implements `Binding`. This struct may be passed to any peripheral to statically prove that the correct interrupt source has been bound to the correct interrupt handler for a given peripheral.
- Declares the interrupt handler, and calls `Handler::on_interrupt()` inside the function.

The actual macro declaration is of little interest in this post. However, if you're interested in seeing the code, you can take a look here. What's more interesting is how it's used and what the generated code looks like:

```
use atsamd_hal::async_hal::interrupts;

atsamd_hal::bind_interrupts!(struct Irqs {
    EIC => atsamd_hal::eic::InterruptHandler;
});
```

Which inlines to:

```
use atsamd_hal::eic;
use atsamd_hal::async_hal::interrupts::{
    Binding, EIC, Handler, SingleInterruptSource
};

// Zero-sized type used to statically prove interrupt bindings
#[derive(Copy,Clone)]
struct Irqs;

// The "real" interrupt handler. Can only appear once in the compi
// Its sole job is to call the on_interrupt trampoline.
#[allow(non_snake_case)]
#[no_mangle]
unsafe extern "C" fn EIC(){
    <eic::InterruptHandler as Handler<EIC>>::on_interrupt();
}

// Implement the binding
unsafe impl Binding<EIC, eic::InterruptHandler> for Irqs
where
    EIC: SingleInterruptSource
{

}
```

Implementing our async peripheral

We now have the ground work laid out to start actually working with Futures! Let's remember our to-do list:

1. We must figure out a way to tie a particular peripheral to its interrupt handler. We want the peripheral to take ownership of the handler - we don't want to rely on the user calling the right functions in the handler, which could be very error prone.
2. Then, inside our `wait()` method, we must ensure that the (correct) interrupt will be fired when the pin reaches the desired state;
3. Finally, when an interrupt occurs, we must wake the Waker inside the interrupt handler.

We've only taken care of half the problem in item 1 so far: we can declare interrupt bindings, but our EIC peripheral still needs to take ownership of them.

1. (continued): Take ownership of the handler

The core struct that manages setting up the external interrupt peripheral is `Eic`. Its definition looks like this:

```
use crate::pac;
use crate::typelevel::NoneT;
```

```
pub struct Eic<I = NoneT>{
    eic: pac::Eic,
    _irq: PhantomData<T>,
}
```

We also have another helper type that will let us define methods on `Eic` only when it has async support enabled:

```
// An empty enum can only exist at the type level
pub enum EicFuture {}
```

We can define our async methods on `Eic` when its type paramater is `EicFuture`, like this:

```
impl Eic<EicFuture> {
    // Implement our async methods here
}
```

The `NoneT` type is used pervasively across atsamd-hal. It's a zero-sized type that semantically represents the *absence* of a type, similar to `Option::None` but at the type level. Very useful indeed, especially as a default parameter in this case. Now we just need to turn an `Eic<NoneT>` into an `Eic<EicFuture>`. This is where we'll check that the interrupt handler binding invariants have been upheld:

```
use crate::typelevel::NoneT;
use crate::async_hal::interrupts::{self, Binding, Handler};

impl Eic<NoneT> {
    pub fn into_future<I>(self, _irq: I) -> Eic<EicFuture>
    where
        I: Binding<interrupts::EIC, InterruptHandler>,
    {
        // Unpend any potentially pending interrupts that could me
        // our handler before we've registered a waker
        interrupts::EIC::unpend();
        // Enable the NVIC interrupt
        unsafe { interrupts::EIC::enable() };

        Eic {
            eic: self.eic,
            _irqs: PhantomData,
        }
    }
}
```

The `Binding` trait bound guarantees that the user has linked the correct interrupt source to our handler. We can assume that the `Eic` peripheral now "owns" the handler for the EIC interrupt source[2]. Note that we haven't talked about the `InterruptHandler` type just yet; for now, all we need to know is that it's the type that implements `Handler` for the EIC peripheral.

For coherency, I want to point out that the `Eic` struct is eventually turned into an `ExtInt`, which is the type that actually does the waiting on interrupts. The details of how this is done are unimportant for this post, apart from the fact that `ExtInt` inherits `Eic`'s `EicFuture` type parameter if it is present.

2. The wait() method

We can now start writing the method which will actually be waiting on the GPIO pin to reach the desired state. It goes as follows:

```
use core::convert::Infaillible;
use embedded_hal::digital::Input;

impl<P, Id> ExtInt<P, Id, EicFuture>
where
    P: EicPin,
    Id: ChId,

    // ExtInt must implement InputPin, because we
    // will be reading its input state
    Self: InputPin<Error = Infallible>,
{
```

```
/// Wait on the pin to reach the state specified in `sense`
pub async fn wait(&mut self, sense: Sense) {
    use core::{future::poll_fn, task::Poll};

    // We start by disabling interrupts as to not interfere
    // with the handler before we had a chance to register a w
    self.disable_interrupt();

    // Before starting to deal with futures, we have an opport
    // return early if the pin is already in the desired state
    match sense {
        Sense::High => {
            if self.is_high().unwrap() {
                return;
            }
        }
        Sense::Low => {
            if self.is_low().unwrap() {
                return;
            }
        }
        _ => (),
    }

    // Ensure that the interrupt will wake the cpu. At this po
    // interrupt itself isn't enabled yet.
    self.enable_interrupt_wake();

    // sense() sets up the pin to generate an interrupt when t
    // desired state is reached
    self.sense(sense);

    // Start building the Future that will actually be polled.
    // The closure will be called every time the future is pol
    poll_fn(|cx| {

        // is_interrupt checks if the interrupt flag is set.
        // Remember that this closure is called at least once,
        // but it's also called every time the Waker is woken
        // (ie, the interrupt handler runs). Thus we need to c
        // if our task is complete before going through anothe
        // register-waker/enable-interrupt/wait-for-interrupt
        if self.is_interrupt() {
                self.clear_interrupt();
                self.disable_interrupt();

                // Reset the pin so it doesn't try to sense anythi
                self.sense(Sense::None);

                // We're ready to make more progress!
                // The closure will not be called again.
                return Poll::Ready(());
            }

            // If we reached this point, we haven't succeeded in r
            // We need to register the Waker.
            //
            // The interrupt handler will wake it when the task ha
            // at which point this entire closure will be executed
            // again **from the top**.
            WAKERS[ChId::ID].register(cx.waker());

            // The interrupt must be enabled **after** the waker i
            self.enable_interrupt();

            // Between the time we checked the interrupt flag and
            // there's a chance the interrupt has fired; this give
            // another chance to return early without actually hav
            // to wait on anything.
            if self.is_interrupt() {
                self.clear_interrupt();
                self.disable_interrupt();
                self.sense(Sense::None);
                return Poll::Ready(());
            }

            // If we failed to return early, we'll have to wait on
            // interrupt handler to run.
            // Return Poll::Pending and give up control back to th
            Poll::Pending
        })
        .await;
    }
}
```

poll_fn is a nice way of avoiding having to create a newtype struct that implements the Future trait, especially if we want to capture variables inside the closure. When using it, we just need to be mindful of the fact that while the code looks linear, the closure we provide might be executed more than once, as it will be called every time the Waker is woken.

You also might have noticed this line, where we register a waker:

```
// ChId::ID is the mechanism by which the ExtInt knows which EIC c
// it's using, at compile time.
WAKERS[ChId::ID].register(cx.waker());
```

What's with the `WAKERS` variable? We haven't declared it anywhere. Well, interrupt handlers can only communicate with other parts of the code via static variables. Therefore, we need to have a place to keep our wakers in static storage, since they are registered in the main thread, but are woken from the handler. Let's look at the declaration:

```
use embassy_sync::waitqueue::AtomicWaker;

// One waker per EIC channel.
// NUM_CHANNELS is defined elsewhere in the HAL.
static WAKERS: [AtomicWaker; NUM_CHANNELS] = [
    const { AtomicWaker::new() }; NUM_CHANNELS
];
```

We keep an array of wakers - one each for each EIC channel. That way, each channel can work independently from one another; if we only had one waker, we'd open ourselves up to the possibility of crosstalk, and the wrong task could be woken. `AtomicWaker` is a convenient way of keeping a Waker in static storage, as it only requires shared, `&self` references for registering or waking it.

3. The interrupt handler

Our final puzzle piece is the interrupt handler implementation itself. Earlier, we've looked at the `Handler` trait, which has one method: `unsafe fn on_interrupt()`. When the interrupt fires, it will trampoline to whatever code we have in `on_interrupt()`. Let's define an empty `InterruptHandler` struct, and implement `Handler` for the `EIC` interrupt source:

```
use crate::async_hal::interrupts::{self, Handler};
pub struct InterruptHandler {
    // Add a private field to prevent anyone from creating an inst
    // of the struct.
    _private: (),
}

impl Handler<interrupts::EIC> for InterruptHandler {
    unsafe fn on_interrupt() {
        // We need to steal the EIC peripheral here.
        // This is safe, as long as we only touch the interrupt
        // related fields and don't otherwise mess with the
        // peripheral's configuration
        let eic = pac::Peripherals::steal().eic;

        // Since we only have a single interrupt for all EIC chann
        // we have to iterate over all of them that have a pending
        // interrupt flag
        let pending_interrupts = BitIter(eic.intflag().read().bits

        for channel in pending_interrupts {
            let mask = 1 << channel;
            // Disable the interrupt but don't clear the flag; wil
            // when future is next polled.
            eic.intenclr().write(|w| w.bits(mask));

            // Wake the waker!
            WAKERS[channel as usize].wake();
        }
    }
}
```

The interrupt flag is our main way of communicating between the handler and the Future in its `poll()` method. It acts as a signal that the task is complete. The best part about it is we don't even need to store that signal anywhere; it's already in the peripheral's registers.

As a convention, in most async peripherals (at least in atsamd-hal's case), the interrupt handler is NOT responsible for clearing the flag. Rather it's the the Future being polled's responsibility. However, the interrupt handler *does* have to *disable* the interrupt: otherwise its handler would be repeatedly called in an infinite loop, never giving the Future a chance to clear the flag and break the cycle.

There we have it! We managed to write a future from scratch, that will react to interrupts. Putting all the pieces together:

```
use core::convert::Infaillible;
use embassy_sync::waitqueue::AtomicWaker;
use crate::async_hal::interrupts::{self, Handler};
use embedded_hal::digital::InputPin;

static WAKERS: [AtomicWaker; NUM_CHANNELS] = [
    const { AtomicWaker::new() }; NUM_CHANNELS
];

pub struct InterruptHandler {
    _private: (),
}

impl Handler<interrupts::EIC> for InterruptHandler {
    unsafe fn on_interrupt() {
        // We need to steal the EIC peripheral here.
        // This is safe, as long as we only touch the interrupt
        // related fields and don't otherwise mess with the
        // peripheral's configuration
        let eic = pac::Peripherals::steal().eic;

        // Since we only have a single interrupt for all EIC chann
        // we have to iterate over all of them that have a pending
        // interrupt flag
        let pending_interrupts = BitIter(eic.intflag().read().bits

        for channel in pending_interrupts {
            let mask = 1 << channel;
            // Disable the interrupt but don't clear the flag; wil
            // when future is next polled.
            eic.intenclr().write(|w| w.bits(mask));
```

```
                // Wake the waker!
                WAKERS[channel as usize].wake();
            }
        }
    }

impl<P, Id> ExtInt<P, Id, EicFuture>
where
    P: EicPin,
    Id: ChId,

    // ExtInt must implement embedded_hal::digital::InputPin, beca
    // will be reading its input state
    Self: InputPin<Error = Infallible>,
{
    /// Wait on the pin to reach the state specified in `sense`
    pub async fn wait(&mut self, sense: Sense) {
        use core::{future::poll_fn, task::Poll};

        // We start by disabling interrupts as to not interfere
        // with the handler before we had a chance to register a w
        self.disable_interrupt();

        // Before starting to deal with futures, we have an opport
        // return early if the pin is already in the desired state
        match sense {
            Sense::High => {
                if self.is_high().unwrap() {
                    return;
                }
            }
            Sense::Low => {
                if self.is_low().unwrap() {
                    return;
                }
            }
            _ => (),
        }

        // Ensure that the interrupt will wake the cpu
        self.enable_interrupt_wake();
```

```
// sense() sets up the pin to generate an interrupt when t
// desired state is reached
self.sense(sense);

// Start building the Future that will actually be polled.
// The closure will be called every time the future is pol
poll_fn(|cx| {

    // is_interrupt checks if the interrupt flag is set.
    // Remember that this closure is called at least once,
    // but it's also called every time the Waker is woken
    // (ie, the interrupt handler runs). Thus we need to c
    // if our task is complete before going through anothe
    // register-waker/enable-interrupt/wait-for-interrupt
    if self.is_interrupt() {
        self.clear_interrupt();
        self.disable_interrupt();

        // Reset the pin so it doesn't try to sense anythi
        self.sense(Sense::None);

        // We're ready to make more progress!
        // The closure will not be called again.
        return Poll::Ready(());
    }

    // If we reached this point, we haven't succeeded in r
    // We need to register the Waker.
    //
    // The interrupt handler will wake it when the task ha
    // at which point this entire closure will be executed
    // again **from the top**.
    WAKERS[P::ChId::ID].register(cx.waker());
    self.enable_interrupt();

    // Between the time we checked the interrupt flag and
    // there's a chance the interrupt has fired; this give
    // another chance to return early without actually hav
    // to wait on anything.
    if self.is_interrupt() {
        self.clear_interrupt();
        self.disable_interrupt();
        self.sense(Sense::None);
```

```
                    return Poll::Ready(());
                }

                // If we failed to return early, we'll have to wait or
                // interrupt handler to run.
                // Return Poll::Pending and give up control back to th
                Poll::Pending
            })
            .await;
        }
    }
```

A usage example

To demonstrate how we can use our new async API, we'll implement a simple program that toggles a LED when a button is pressed. We'll implement the same program in both sync and async mode, using the Feather M0 development board. We'll also use the `feather_m0` Board Support Package, which in turn uses `atsamd-hal`.

```
# Cargo.toml

[package]
name = "external-interrupts"
version = "0.1.0"

[dependencies.feather_m0]
version = "0.20.0"
features = ["rt", "async"]

[dependencies.cortex_m]
version = "0.7"

[dependencies.embassy-executor]
version = "0.6.2"
features = ["arch-cortex-m", "executor-thread", "task-arena-size-(
```

Sync version

```
// src/bin/eic_sync.rs

#![no_std]
#![no_main]

use panic_halt as _;

use feather_m0 as bsp;

use bsp::{entry, hal, pac};

use pac::{interrupt, CorePeripherals, Peripherals};

use core::cell::RefCell;
use core::sync::atomic::{AtomicBool, Ordering};

use bsp::hal::ehal::digital::StatefulOutputPin;
use cortex_m::{interrupt::Mutex, peripheral::NVIC};

use hal::{
    clock::GenericClockController,
    eic::{Ch2, Eic, ExtInt, Sense},
    gpio::{Pin, PullUpInterrupt, PA18},
};

type ButtonPin = ExtInt<Pin<PA18, PullUpInterrupt>, Ch2>;

// To avoid unsafely passing the button pin to the interrupt handl
// we must use a Mutex. This can also be done safely by using a fr
// like RTIC.
static BUTTON_PIN: Mutex<RefCell<Option<ButtonPin>>> = Mutex::new(

// Used to signal to the main thread that the interrupt has fired
// from the interrupt handler
static INTERRUPT_FIRED: AtomicBool = AtomicBool::new(false);

#[entry]
fn main() -> ! {
    // -- Setup clocks and peripherals
    let mut peripherals = Peripherals::take().unwrap();
    let mut core = CorePeripherals::take().unwrap();
    let mut clocks = GenericClockController::with_external_32kosc(
        peripherals.gclk,
        &mut peripherals.pm,
        &mut peripherals.sysctrl,
        &mut peripherals.nvmctrl,
    );
    let _internal_clock = clocks
        .configure_gclk_divider_and_source(ClockGenId::Gclk2, 1, (
        .unwrap();
    clocks.configure_standby(ClockGenId::Gclk2, true);

    enable_internal_32kosc(&mut peripherals.sysctrl);

    // Configure a clock for the EIC peripheral
    let gclk2 = clocks.get_gclk(ClockGenId::Gclk2).unwrap();
    let eic_clock = clocks.eic(&gclk2).unwrap();

    let pins = bsp::Pins::new(peripherals.port);

    // Take the LED pin and set it to output mode
    let mut red_led: bsp::RedLed = pins.d13.into();

    // Setup the External Interrupt Controller and split it into i
    let eic_channels = Eic::new(&mut peripherals.pm, eic_clock, pe

    // Enable EIC interrupt in the NVIC
    unsafe {
        core.NVIC.set_priority(interrupt::EIC, 1);
        NVIC::unmask(interrupt::EIC);
    }

    // Take the user button pin
    let button: Pin<_, PullUpInterrupt> = pins.d10.into();
    // Turn the pin into an external interrupt using EIC channel 2
    let mut extint = eic_channels.2.with_pin(button);

    // Setup the button pin to wake the CPU upon interrupt
    extint.enable_interrupt_wake();

    // Setup the pin to sense falling edges. It will generate
    // interrupts on every falling edge, not juste the first one
    extint.sense(Sense::Fall);

    // Enable the pin's interrupt
    extint.enable_interrupt();

    // Store the button pin in static storage so that the interruı
    // handler can access it
    cortex_m::interrupt::free(|cs| BUTTON_PIN.borrow(cs).borrow_mı

    loop {
        // Check if our interrupt has fired
        if INTERRUPT_FIRED.load(Ordering::Acquire) {
            // Toggle the LED! We don't use the return
            // value, because toggling a pin is infaillible
            // in atsamd-hal.
            let _ = red_led.toggle();

            // Reset the signal for the next loop
            INTERRUPT_FIRED.store(false, Ordering::Release);
        }
        // Put the CPU to sleep while we wait for an interrupt
        // to happen
        cortex_m::asm::wfi();
    }
}

/// The external interrupt controller handler
#[interrupt]
fn EIC() {
    // Clear the interrupt so we don't reenter the handler
    // infinitely
    cortex_m::interrupt::free(|cs| {
        let mut button = BUTTON_PIN.borrow(cs).borrow_mut();
        let button = button.as_mut().unwrap();
        button.clear_interrupt();
    });

    // Send a signal to the main thread
    INTERRUPT_FIRED.store(true, Ordering::Release);
}
```

Async version

```
// src/bin/eic_async.rs
```

```rust
#![no_std]
#![no_main]

use panic_halt as _;

use bsp::pac;
use bsp::{hal, pin_alias};
use feather_m0 as bsp;
use hal::{
    clock::{enable_internal_32kosc, ClockGenId, ClockSource, Gener
    ehal::digital::StatefulOutputPin,
    eic::{Eic, Sense},
    gpio::{Pin, PullUpInterrupt},
};

atsamd_hal::bind_interrupts!(struct Irqs {
    EIC => atsamd_hal::eic::InterruptHandler;
});

// We use embassy-executor to turn our main function in an async
#[embassy_executor::main]
async fn main(_s: embassy_executor::Spawner) {
    // -- Setup clocks and peripherals
    let mut peripherals = pac::Peripherals::take().unwrap();
    let _core = pac::CorePeripherals::take().unwrap();

    let mut clocks = GenericClockController::with_external_32kosc(
        peripherals.gclk,
        &mut peripherals.pm,
        &mut peripherals.sysctrl,
        &mut peripherals.nvmctrl,
    );
    let pins = bsp::Pins::new(peripherals.port);

    let _internal_clock = clocks
        .configure_gclk_divider_and_source(ClockGenId::Gclk2, 1, (
        .unwrap();
    clocks.configure_standby(ClockGenId::Gclk2, true);

    enable_internal_32kosc(&mut peripherals.sysctrl);

    // Configure a clock for the EIC peripheral
    let gclk2 = clocks.get_gclk(ClockGenId::Gclk2).unwrap();
    let eic_clock = clocks.eic(&gclk2).unwrap();

    // Take the LED pin and set it to output mode
    let mut red_led: bsp::RedLed = pins.d13.into();

    // Setup the External Interrupt Controller and split it into 
    let eic_channels = Eic::new(&mut peripherals.pm, eic_clock, pe
        .into_future(Irqs)
        .split();

    // Take the user button pin
    let button: Pin<_, PullUpInterrupt> = pins.d10.into();
    // Turn the pin into an external interrupt using EIC channel 
    let mut extint = eic_channels.2.with_pin(button);

    loop {
        // Wait for a falling edge...
        extint.wait(Sense::Fall).await;
        // ...and toggle the LED!
        let _ = red_led.toggle();
    }
}
```

Not only is the async version of the program almost 50% shorter, it's also much more linear and readable. Plus, as a bonus, it doesn't use any unsafe code, and doesn't require passing data between an executor handler and the main thread through a critical section![3] The async loop might look linear, but the CPU is actually free do take care of other async tasks during the await call while it's waiting for the interrupt to arrive, or go to sleep like in the sync example.

Wrapping up

There you have it - a working async implementation. We skimmed over lots of details that are specific to the atsamd-hal way of doing things - I plan on covering some of those in a later post, especially how the type system works. However, in my opinion, what we did cover really do captures the essence of working with asynchronous periperals on microcontrollers, and how they interact with interrupts and their handlers.

Until next time!

Additional Reading

- Writing an OS in Rust: Async/Await: A truly excellent post that goes through all the concepts necessary to write and understand Futures and executors.
- Pin and suffering: A deep-dive into the intricacies of using the Pin type for async tasks and Futures.

Footnotes

[3] Okay, that's a bit of a lie. Deep down in the HAL's internals, registering and waking a waker does take a critical section. Although, that can easily be optimized. We could also (unsafely) steal the EIC peripheral in the sync version to avoid statically storing it behind a Mutex. ↩

[1] That's not entirely true; interrupt handlers can be declared in a library, but things get ugly and rife with linker errors pretty quickly. ↩

[2] It is not the same thing as the commonly understood Rust ownership semantics: The `Eic` interrupt handler is bound to the EIC interrupt source for the entirety of the program's lifetime. It can not "return" ownership of the handler to the caller. ↩