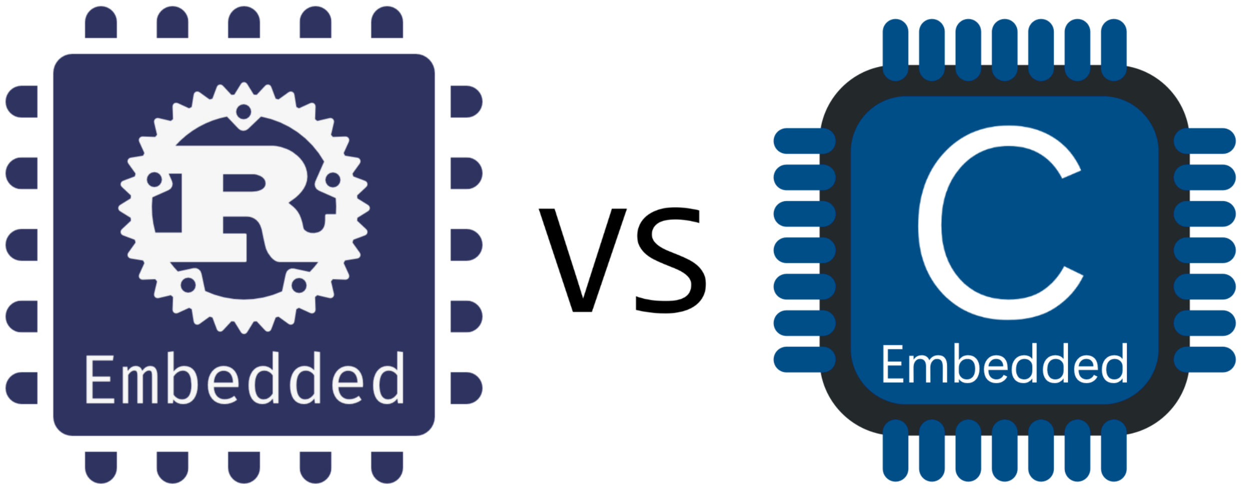
January 31, 2022

Async Rust vs RTOS showdown!

Dion
Embedded software engineer

embedded IoT rust



It's time for another technical blog post about async Rust on embedded. This time we're going to pitch Embassy/Rust against FreeRTOS/C on an STM32F446 microcontroller.

They will both be running applications that perform the same actions. We're then going to judge them on the basis of interrupt latency, program size, ram usage and ease of programming. There are already a lot of articles that compare C and Rust, so we're not could be tuned to give better performance with a lot of work. Doing that can be a nearly endless task. So as a guideline, the applications will be:

- Portable(-ish) to other chips and architectures (aside from the dependency on the HAL)
- Straightforward
- Tuned with normal options and settings like compiler optimizations, rtos settings and thread priorities

In the end, we should have a basic understanding of how RTOS'es and async executors (can) work.

I am biased, but I hope this blog post gives a fair comparison. If you have suggestions, please let us know!

We'll be testing with the STM32F446ZET6 microcontroller at 180Mhz and some of the measurements will be done with a Rigol DS1054Z oscilloscope.

Async Rust

An async function in Rust is syntax sugar for a function that returns a future.

Rust

```rust
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

The function is transformed into a state machine object that can be polled. The state machine allows the code to jump into the function, resuming where it previously stopped. It also keeps track of all the its completion, all you have to do is to continuously call the `poll` function until it stops returning the `Pending` state and returns the `Ready(Output)` state.

This is straightforward, but not very efficient.

To fix that, there are also `Wakers`. A waker can signal to the executor that a future ought to be polled again. This waker can be called by the future itself or can be given to another process/thread the future depends on. In general, an executor calls the `poll` function once and then only calls it again when the waker is triggered.

A future can call other futures and incorporate them into itself. For an executor, any top-level future it polls is usually called a `task`.

Lots more can be said. Luckily I don't have to because there are some really good resources out there:

- Under the Hood: Executing Futures and Tasks
- How Rust optimizes async/await
- Understanding Rust futures by going way too deep

In Embassy

Embassy uses this mechanism as well but adds a couple of constraints.

- Tasks have to be statically allocated
  Embassy doesn't want to depend on an allocator
  All tasks must be known at compile time
- A nightly compiler is required
  The `type_alias_impl_trait` preview feature is required
  This is because we can't use boxed trait objects, due to having no allocator

For many peripherals, Embassy has made an async interface. This

```rust
#[embassy::task]
async fn my_task(mut button: ExtiInput<'static, PC13>) {
    loop {
        button.wait_for_rising_edge().await;
        info!("Pressed!");
        button.wait_for_falling_edge().await;
        info!("Released!");
    }
}
```

A couple of things are happening here.

The `wait_for_rising_edge` creates a new future and returns it. The constructor of the future configures the interrupt of the pin. On the first poll, the future puts its waker into a global array of EXTI wakers. When an EXTI interrupt happens, the appropriate waker in that array is used to wake up the right task.

So when the interrupt exits, the executer polls the task again, the `wait_for_rising_edge` future notices its interrupt has fired and returns that it is ready. And so the program continues.

One thing Embassy doesn't do is pre-emption, which means that the active task is only switched to a more important one when it awaits something. This is called cooperative multitasking. But Embassy has some other features that make this missing feature a non-issue, which will be covered later on in this article.

RTOS

A real-time operating system divides everything up into independent threads. Different from tasks is that threads don't run a state machine, but run normal code. This means that you don't have to program your code in a special way. Any old function can be run in an RTOS.

because the thread is running normal code. When that code resumes, it will require the processor context to be the same again.

This design of multithreading lends itself to pre-emptive threads. This means that the kernel can give fair execution time to all threads, that the user can specify priorities and that the kernel can respond to events and interrupts in a predictable amount of time.

This description doesn't even scratch the surface of an RTOS. To get a better understanding, here are some articles if you're interested:

- How to build a Real-Time Operating System
- FreeRTOS Kernel Developer Docs

Let the showdown begin!

Now that we know a bit about the two models, we're going to pitch them against each other by implementing the same program in both.

The program

We can't build a fully realistic program because that would just take too long to build. But let's try to have something that is not too simple.

There are a couple of things we need to be able to claim to be approaching realism:

- Multiple tasks
- Data sharing between tasks
- Responding to interrupts

So, what our program will do is the following three (literal) tasks:

- Blink an LED every 200ms for 100ms
  Be in a loop and use the delay function of the executor

- Keep track of the user button
  Set up a gpio interrupt so we can detect a signal change
  Communicate in a shared (atomic) boolean whether the button is high or low
  When the button state changes, put a string on the message queue with the text `Button is <0/1> (N)\n` where `<0/1>` is 0 if the button is low and 1 if the button is high and `N` is the number of triggers
- Print the message queue to serial
  Wait for the message queue to contain a string
  Print it to serial

What we're measuring

This showdown can be won on the basis of these things:

Performance

How long does the button gpio interrupt take?

- When the interrupt fires, we will set a pin high
- When the interrupt ends, we will set the pin low
- The time in between is measured by an oscilloscope

How long does the button thread take until it waits again?

- When the thread stops waiting, we will set a pin high
- When the thread starts waiting again, we will set the pin low
- The time in between is measured by an oscilloscope

Interrupt (processing) latency

What is the time between the start of the button gpio interrupt and the button thread resuming?

- The time between the rise of the interrupt pin and the rise of the thread pin is measured by an oscilloscope

.text section as reported by arm-none-eabi-size

Static memory usage

.data + .bss section as reported by arm-none-eabi-size

- All tasks and threads are statically allocated

We're only looking at static memory usage because dynamic memory usage is difficult to measure. A program that statically allocates a lot of memory will likely use less stack memory than a similar program that doesn't. However, since RTOS'es can struggle with this, I think it's a relevant metric to compare.

Ease of programming

Very subjective, I know

To reiterate from the start, we're not looking for the most optimized solution. The goal is to have a relatively normal program.

Expectations

I don't really know what to expect except that an RTOS is made to really optimize performance and latency. So based on that, here are my predictions:

Performance

The RTOS will set a flag in the thread directly, this is probably faster than having to find an async waker and triggering it.

Aside from how the code is resumed and suspended, there's not much difference for the button thread between the two

Interrupt (processing) latency

The RTOS will probably be more optimized for this. Embassy can't pre-empt running tasks, so it's less worthwhile to optimize this a lot.

Program size

Rust programs are usually a bit bigger due to more expensive formatting and compiler inserted runtime checks. Since the rest of the program is essentially the same, I expect the C implementation to use less flash memory.

Static memory usage

Because Rust's compiler-generated futures only store the variables that are held across an await point and doesn't have to fully allocate a full-stack size, the Rust implementation should win.

Ease of programming

Ignoring the 'Rust vs C' side, I think the async model will be nicer to work with. In the web world async/await has already won from threads, so that will probably be the case here as well.

Let's look at the code

The repository can be found here: github The C project is made in STMCube 1.8 and the Rust project is a standard cargo binary.

Getting the button interrupt noticed

We're not going to process everything in the interrupt, we're just notifying the executor that the interrupt has happened.

For Rust, we don't need to do anything because this is exactly what Embassy already does.

C++

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (GPIO_Pin == USER_Btn_Pin) {
        osThreadFlagsSet(buttonWaiterHandle, 1);
    }
}
```

Blinking the led

We're going to use the normal delay function of each executor to wait for our time. To determine if the button is pressed, we have an atomic bool that we need to read. In C that bool is stored in a global because the tasks are created globally and getting it from the void pointer argument is not very nice.

Rust

Rust

```
#[embassy::task]
async fn blink_led(mut led: Output<'static, PB0>, button_high: &'static AtomicBool) {
    loop {
        Timer::after(Duration::from_millis(100)).await;
        if !button_high.load(Ordering::SeqCst) {
            led.set_high().unwrap();
        }
        Timer::after(Duration::from_millis(100)).await;
        led.set_low().unwrap();
    }
}
```

In Rust we need to annotate our task function so it can be statically allocated. The LED is also given as an argument because the

The atomic types in C are an optional part of the C11 spec and luckily our compiler implements them. This makes using atomic types a lot more comfortable.

C++

```
void StartBlinkLedTask(void *argument)
{
    for (;;) {
        osDelay(100);

        if (atomic_load(&buttonPressed) == GPIO_PIN_RESET) {
            HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_SET);
        }

        osDelay(100);
        HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_RESET);
    }
}
```

Writing the message queue to serial

The messages we send to the writing task are basically strings. The Rust implementation uses the `ArrayVec` library to get access to a good stack allocated `ArrayString` type.

In C we don't have as much luxury, so I made a simple type for it:

C++

```
typedef struct {
    char data[32];
} UartMessage;
```

will wait for a new message to show up and then print it to the uart.

Rust

The Rust implementation is pretty straightforward:

Rust

```
#[embassy::task]
async fn uart_writer(
    mut usart: Uart<'static, USART3, DMA1_CH3>,
    mut receiver: Receiver<'static, Noop, ArrayString<32>, 8>,
) {
    loop {
        let message = receiver.recv().await.unwrap();
        usart.write(message.as_bytes()).await.unwrap();
    }
}
```

C

In C we need to do a bit more memory and size management:

C++

```
void StartUartWriter(void *argument) {
    for (;;) {
        UartMessage message;

        CheckStatus(
            osMessageQueueGet(uartQueueHandle, &message, NULL, osWaitForever)
        );

        size_t messageLength = strnlen(message.data, sizeof(message.data));
        CheckStatus(
}
```

Waiting on the button

The button logic is split into a couple of parts.

First, the button pin is configured to generate an interrupt on a rising edge. The interrupt is then waited on. For the measurement, the `button_processed` pin is also turned high and low around the waiting line.

After the waiting is over, the trigger count is upped, the button pressed variable is set high and a message is formatted and sent to the message queue.

This is then repeated with the interrupt set to the falling edge.

Observant readers might notice that there isn't any debouncing for the button and that's definitely a problem. But I felt that if I put in a delay here, it would ruin the measurements we're going to do. So no debouncing is done, which makes the state of the `button_pressed` variable a bit unreliable.

Rust

Anyway, here is the Rust code:

Rust

```rust
#[embassy::task]
async fn button_waiter(
    mut button: ExtiInput<'static, PC13>,
    button_pressed: &'static AtomicBool,
    sender: Sender<'static, Noop, ArrayString<32>, 8>,
    mut button_processed: Output<'static, PG1>,
) {
    loop {
        button_processed.set_low().unwrap();
        button.wait_for_rising_edge().await;
        button_processed.set_high().unwrap();

        trigger_count += 1;
        button_pressed.store(true, Ordering::SeqCst);
        if sender.send(format_message(trigger_count, true)).await.is_err() {
            panic!("SendError");
        }

        button_processed.set_low().unwrap();
        button.wait_for_falling_edge().await;
        button_processed.set_high().unwrap();

        trigger_count += 1;
        button_pressed.store(false, Ordering::SeqCst);
        if sender.send(format_message(trigger_count, false)).await.is_err() {
            panic!("SendError");
        }
    }
}
```

I found out that unwrapping the `sender.send()` result leads to unreasonably expensive formatting code (size-wise) while it doesn't show any relevant information. So it now does just a simple panic.

C

In the C code, we need to change the pin interrupt direction ourselves.

C++

```cpp
void StartButtonWaiterTask(void *argument) {
    int triggerCount = 0;
```

```
        for (;;) {
            // Only react to rising edges
            EXTI->RTSR |= USER_Btn_Pin;
            EXTI->FTSR &= ~USER_Btn_Pin;

            HAL_GPIO_WritePin(ButtonProcessed_GPIO_Port, ButtonProcessed_Pin, GPIO_PIN_RESE
            osThreadFlagsWait(1, osFlagsWaitAny, osWaitForever);
            HAL_GPIO_WritePin(ButtonProcessed_GPIO_Port, ButtonProcessed_Pin, GPIO_PIN_SET)
            triggerCount++;

            // Set the button pressed variable
            atomic_store(&buttonPressed, true);
            message = FormatMessage(triggerCount, true);
            CheckStatus(
                osMessageQueuePut(uartQueueHandle, &message, 0, osWaitForever)
            );

            // Only react to falling edges
            EXTI->RTSR |= USER_Btn_Pin;
            EXTI->FTSR &= ~USER_Btn_Pin;

            HAL_GPIO_WritePin(ButtonProcessed_GPIO_Port, ButtonProcessed_Pin, GPIO_PIN_RESE
            osThreadFlagsWait(1, osFlagsWaitAny, osWaitForever);
            HAL_GPIO_WritePin(ButtonProcessed_GPIO_Port, ButtonProcessed_Pin, GPIO_PIN_SET)
            triggerCount++;

            // Set the button pressed variable
            atomic_store(&buttonPressed, false);
            message = FormatMessage(triggerCount, false);
            CheckStatus(
                osMessageQueuePut(uartQueueHandle, &message, 0, osWaitForever)
            );
        }
    }
```

That's pretty much all of the code aside from the setup.

provides its own interrupt function, so that had to be modified.

In the exti file of the embassy_stm32 file, I added it here:

Rust

```rust
macro_rules! impl_irq {
    ($e:ident) => {
        #[interrupt]
        unsafe fn $e() {
            pac::gpio::Gpio(0x40021800 as *mut u8).odr().modify(|odr| odr.set_odr(0, st
            let x = on_irq();
            pac::gpio::Gpio(0x40021800 as *mut u8).odr().modify(|odr| odr.set_odr(0, st
            x
        }
    };
}
```

After all this time, let's look at what the results are!

Results

First off, I really like the async await model. Once you accept the idea that you can await something that you'd normally have an interrupt for, it writes very nicely! Managing threads is not a lot of fun, so I'm not really missing that part.

Because Embassy is built around interrupts, its design feels really nice and integrated. Handling the interrupt in FreeRTOS is a lot less ergonomic. For me, this is a win for Embassy.

Let's run the tests so we can look at the numbers.

I will push the button a hundred times so we'll get two hundred

Test	C	Rust	Difference	Difference %
Interrupt time (avg)	2.962us	1.450us	-1.512us	-51.0%
Interrupt time (stddev)	124.8ns	4.96ns	-119.84ns	-96.0%
Thread time (avg)	16.19us	11.64us	-4.55us	-28.1%
Thread time (stddev)	248.2ns	103.0ns	-145.2ns	-56.2%
Interrupt latency (avg)	4.973us	3.738us	-1.235us	-24.8%
Interrupt latency (stddev)	158.0ns	45.3ns	-112.7ns	-71.3%
Program size	20676b	14272b	-6404b	-31.0%
Static memory size	5480b	872b	-4608b	-84.1%

Oh...

Wow...

I genuinely did not expect this.

These numbers are also repeatable on different days (with slight variations of course).

It looks like Embassy/Rust won in every category! Ok, let's at least look at something where FreeRTOS/C did actually beat Rust. If we look at the time between the end of the interrupt and the start of the thread awaking, we get the following numbers: `(Interrupt latency - Interrupt time)`

- C: `4.973 - 2.962 = 2.011us`
- Rust: `3.738 - 1.450 = 2.288us`

This shows that purely the context switching and resuming the thread is faster in the RTOS. But in the face of an interrupt that takes the thread signalling model? To answer that we'd have to test more RTOS'es and more HALS. That's maybe something for another time.

One of the biggest improvements we could make in the RTOS code is moving the button logic to inside of the interrupt. This is something that is not possible in Embassy, because it itself creates the interrupt functions for us. There's a tradeoff here. Freedom in FreeRTOS/C and ease of development for Embassy/Rust.

The winner

I can only declare Embassy/Rust as the winner here.

Not only is it nicer to program in my opinion, all the numbers seem to favor it too.

Wrap up

There's just one thing that may still worry you. An RTOS can be used in actual real-time applications. Because the async tasks can't be pre-empted, it is not always possible to execute another task in time. While this is true for a set of tasks in one executor, Embassy allows us to use additional executors that run inside interrupt contexts.

A waker will not only trigger the executor to run a task, if the executor is on an interrupt context, the waker also sets the executor's interrupt pending. This way if the executor is on an interrupt that has a higher priority, it will pre-empt other executors on lower priorities.

Here's the example that Embassy gives on Github.

I'd like to thank the creator of Embassy, Dario, and Sjors from Jitter for giving feedback on this post and for answering my questions.

Discussions on /r/rust and /r/embedded.

Edit

It was brought to my attention that I had left the heap turned on in the C project even though it was not used. This caused the static memory size to be 15kb bigger than it really had to be. I've subtracted the heap size from the static memory size. The conclusion is still the same though.

RTIC addendum (17-02-2022)

We got a pull request on our repo to add an implementation for RTIC. Thanks Rafael Bachmann! (barafael)

RTIC is a fully interrupt-driven runtime that is used quite a bit in the rust embedded ecosystem. You can find more here: https://rtic.rs/1/book/en/preface.html.

I've changed the PR a little bit so that it falls in line with the FreeRTOS and Embassy implementations and have run the numbers again.

It is important to say, though, that the RTIC implementation is not entirely fair to the other two implementations. Because RTIC defines its interrupt handlers inside of a macro (so I can't modify them), I can't set one of the gpio pins high immediately. However, since RTIC is a really small layer on top of the interrupts, I still think setting the pin high in the user-provisioned interrupt function will represent the performance just fine.

We're going to use Embassy as our baseline.

Test	RTIC	Embassy	Difference	Difference %
Interrupt time (avg)	650.8ns	1450ns	799ns	122.8%

Thread time (stddev)	279.9ns	103.0ns	-176.9ns	-63.2%
Interrupt latency (avg)	1.184us	3.738us	2.554us	215.7%
Interrupt latency (stddev)	77.75ns	45.3ns	-32.45ns	-41.7%
Program size	8888b	14272b	5384b	60.0%
Static memory size	392b	872b	480b	122.4%

So, RTIC shows some impressive results. That's of course very logical. The less runtime you bring along, the less you have to carry.

RTIC is a clear improvement over manually implementing interrupts. I usually say that if you keep implementing features on interrupts, then eventually you'll get a worse version of RTIC, so just use RTIC.

Attribution

The Rust Embedded Working Group Logo, based on the Rust logo, was designed by Erin Power.

Dion
Embedded software engineer
dion@tweedegolf.com

`embedded` `IoT` `rust`

Stay up-to-date

Stay up-to-date with our work and blog posts?