RTIC

The hardware accelerated Rust RTOS

A concurrency framework for building real-time systems

Preface

This book contains user level documentation for the Real-Time Interrupt-driven Concurrency (RTIC) framework. The API reference is available here.

This is the documentation for RTIC v2.x.

Older releases: RTIC v1.x | RTIC v0.5.x (unsupported) | RTFM v0.4.x (unsupported)

Features

- **Tasks** as the unit of concurrency [^1]. Tasks can be *event triggered* (fired in response to asynchronous stimuli) or spawned by the application on demand.

Is RTIC an RTOS?

A common question is whether RTIC is an RTOS or not, and depending on your background the answer may vary. From RTIC's developers point of view; RTIC is a hardware accelerated RTOS that utilizes the hardware such as the NVIC on Cortex-M MCUs, CLIC on RISC-V etc. to perform scheduling, rather than the more classical software kernel.

Another common view from the community is that RTIC is a concurrency framework as there is no software kernel and that it relies on external HALs.

RTIC - The Past, current and Future

This section gives a background to the RTIC model. Feel free to skip to section RTIC the model for a TL;DR.

The RTIC framework takes the outset from real-time systems research at Luleå University of Technology (LTU) Sweden. RTIC is inspired by the concurrency model of the Timber language, the RTFM-SRP based scheduler, the RTFM-core language and Abstract Timer implementation. For a full list of related research see RTFM and RTIC publications.

Stack Resource Policy based Scheduling

Stack Resource Policy (SRP) based concurrency and resource management is at heart of the RTIC framework. The SRP model itself extends on Priority Inheritance Protocols, and provides a set of outstanding properties for single core scheduling. To name a few:

- preemptive deadlock and race-free scheduling
- resource efficiency
    - tasks execute on a single shared stack
    - tasks run-to-completion with wait free access to shared resources
- predictable scheduling, with bounded priority inversion by a single (named) critical section
- theoretical underpinning amenable to static analysis (e.g., for task response times and overall schedulability)

SRP comes with a set of system-wide requirements:

- each task is associated a static priority,
- tasks execute on a single-core,
- tasks must be run-to-completion, and
- resources must be claimed/locked in LIFO order.

SRP analysis

SRP based scheduling requires the set of static priority tasks and their access to shared resources to be known in order to compute a static *ceiling* (***π***) for each resource. The static resource *ceiling* ***π***(r) reflects the maximum static priority of any task that accesses the resource `r`.

Example

Assume two tasks `A` (with priority `p(A) = 2`) and `B` (with priority `p(B) = 4`) both accessing the shared resource `R`. The static ceiling of `R` is 4 (computed from `π(R) = max(p(A) = 2, p(B) = 4) = 4`).

A graph representation of the example:

```
graph LR
    A["p(A) = 2"] --> R
    B["p(B) = 4"] --> R
    R["π(R) = 4"]
```

RTIC the hardware accelerated real-time scheduler

SRP itself is compatible with both dynamic and static priority scheduling. For the implementation of RTIC we leverage on the underlying hardware for accelerated static priority scheduling.

In the case of the `ARM Cortex-M` architecture, each interrupt vector entry `v[i]` is associated a function pointer (`v[i].fn`), a static priority (`v[i].priority`), an enabled- (`v[i].enabled`) and a pending-bit (`v[i].pending`).

An interrupt `i` is scheduled (run) by the hardware under the conditions:

1. is `pending` and `enabled` and has a priority higher than the (optional `BASEPRI`) register, and
2. has the highest priority among interrupts meeting 1.

The first condition (1) can be seen as a filter, allowing RTIC to take control over which tasks should be allowed to start (and which should be prevented from starting).

The SRP model for single-core static scheduling, on the other hand, states that a task should be scheduled (run) under the conditions:

1. it is `requested` to run and has a static priority higher than the current system ceiling (Π)
2. it has the highest static priority among tasks meeting 1.

The similarities are striking and it is not by chance/luck/coincidence. The hardware was cleverly designed with real-time scheduling in mind.

In order to map the SRP scheduling onto the hardware we need to take a closer look at the system ceiling (Π). Under SRP Π is computed as the maximum priority ceiling of the currently held resources, and will thus change dynamically during the system operation.

Example

Assume the task model above. Starting from an idle system, Π is 0, (no task is holding any resource). Assume that `A` is requested for execution, it will immediately be scheduled. Assume that `A` claims (locks) the resource `R`. During the claim (lock of `R`) any request `B` will be blocked from starting (by Π = `max(`π`(R) = 4) = 4`, `p(B) = 4`, thus SRP scheduling condition 1 is not met).

Mapping

The mapping of static priority SRP based scheduling to the Cortex M hardware is straightforward:

- each task `t` are mapped to an interrupt vector index `i` with a corresponding function `v[i].fn = t` and given the static priority `v[i].priority = p(t)`.
- the current system ceiling is mapped to the `BASEPRI` register or implemented through masking the interrupt enable bits accordingly.

Example

For the running example, a snapshot of the ARM Cortex M Nested Vectored Interrupt Controller (NVIC) may have the following configuration (after task `A` has been pended for execution.)

Index	Fn	Priority	Enabled	Pending
0	A	2	true	true
1	B	4	true	false

(As discussed later, the assignment of interrupt and exception vectors is up to the user.)

A claim (lock(r)) will change the current system ceiling ($\boldsymbol{\Pi}$) and can be implemented as a *named* critical section:

- old_ceiling = $\boldsymbol{\Pi}$, $\boldsymbol{\Pi}$ = $\boldsymbol{\pi}$(r)
- execute code within critical section
- old_ceiling = $\boldsymbol{\Pi}$

This amounts to a resource protection mechanism, requiring only two machine instructions on enter and one on exit the critical section, for managing the `BASEPRI` register. For architectures lacking `BASEPRI`, we can implement the system ceiling through a set of machine instructions for disabling/enabling interrupts on entry/exit for the named critical section. The number of machine instructions vary depending on the number of mask

registers that needs to be updated (a single machine operation can operate on up to 32 interrupts, so for the M0/M0+ architecture a single instruction suffice). RTIC will determine the ceiling values and masking constants at compile time, thus all operations is in Rust terms zero-cost.

In this way RTIC fuses SRP based preemptive scheduling with a zero-cost hardware accelerated implementation, resulting in "best in class" guarantees and performance.

Given that the approach is dead simple, how come SRP and hardware accelerated scheduling is not adopted by any other mainstream RTOS?

The answer is simple, the commonly adopted threading model does not lend itself well to static analysis - there is no known way to extract the task/resource dependencies from the source code at compile time (thus ceilings cannot be efficiently computed and the LIFO resource locking requirement cannot be ensured). Thus, SRP based scheduling is in the general case out of reach for any thread based RTOS.

RTIC into the Future

Asynchronous programming in various forms are getting increased popularity and language support. Rust natively provides an `async`/`await` API for cooperative multitasking and the compiler generates the necessary boilerplate for storing and retrieving execution contexts (i.e., managing the set of local variables that spans each `await`).

The Rust standard library provides collections for dynamically allocated data-structures which are useful to manage execution contexts at run-time. However, in the setting of resource constrained real-time systems, dynamic allocations are problematic (both regarding performance and reliability - Rust runs into a *panic* on an out-of-memory condition). Thus, static allocation is the preferable approach!

From a modelling perspective `async/await` lifts the run-to-completion requirement of SRP, and each section of code between two yield points (`await`s) can be seen as an individual task. The compiler will reject any attempt to `await` while holding a resource (not doing so would break the strict LIFO requirement on resource usage under SRP).

So with the technical stuff out of the way, what does `async/await` bring to the table?

The answer is - improved ergonomics! A recurring use case is to have task perform a sequence of requests and then await their results in order to progress. Without `async`/`await` the programmer would be forced to split the task into individual sub-tasks and maintain some sort of state encoding (and manually progress by selecting sub-task). Using `async/await` each yield point (`await`) essentially represents a state, and the progression mechanism is built automatically for you at compile time by means of `Futures`.

Rust `async`/`await` support is still incomplete and/or under development. Nevertheless, it covers most common use cases and can be considered production ready.

An important property is that futures are composable, thus you can await either, all, or any combination of possible futures (allowing e.g., timeouts

and/or asynchronous errors to be promptly handled).

RTIC the model

An RTIC `app` is a declarative and executable system model for single-core applications, defining a set of (`local` and `shared`) resources operated on by a set of (`init`, `idle`, *hardware* and *software*) tasks. In short the `init` task runs before any other task returning a set of resources (`local` and `shared`). Tasks run preemptively based on their associated static priority, `idle` has the lowest priority (and can be used for background work, and/or to put the system to sleep until woken by some event). Hardware tasks are bound to underlying hardware interrupts, while software tasks are scheduled by asynchronous executors (one for each software task priority).

At compile time the task/resource model is analyzed under SRP and executable code generated with the following outstanding properties:

- guaranteed race-free resource access and deadlock-free execution on a single-shared stack (thanks to SRP)
    - hardware task scheduling is performed directly by the hardware, and
    - software task scheduling is performed by auto generated async executors tailored to the application.

The RTIC API design ensures that both SRP requirements and Rust soundness rules are upheld at all times, thus the executable model is correct by construction. Overall, the generated code infers no additional overhead in comparison to a handwritten implementation, thus in Rust terms RTIC offers a zero-cost abstraction to concurrency.

Starting a new project

A recommendation when starting a RTIC project from scratch is to follow RTIC's `defmt-app-template`.

If you are targeting ARMv6-M or ARMv8-M-base architecture, check out the section Target Architecture for more information on hardware limitations to be aware of.

This will give you an RTIC application with support for RTT logging with `defmt` and stack overflow protection using `flip-link`. There is also a multitude of examples provided by the community:

For inspiration, you may look at the RTIC examples.

RTIC on RISC-V devices

Even though RTIC was initially developed for ARM Cortex-M, it is possible to use RTIC on RISC-V devices. However, the RISC-V ecosystem is more heterogeneous. To tackle this issue, currently, RTIC implements three different backends:

- `riscv-esp32c3-backend`: This backend provides support for the ESP32-C3 SoC. In these devices, RTIC is very similar to its Cortex-M counterpart.
- `riscv-esp32c6-backend`: This backend provides support for the ESP32-C6 SoC. In these devices, RTIC is very similar to its Cortex-M counterpart.
- `riscv-mecall-backend`: This backend provides support for **any** RISC-V device. In this backend, pending tasks trigger Machine Environment Call exceptions. The handler for this exception source dispatches pending tasks according to their priority. The behavior of this backend is equivalent to `riscv-clint-backend`. The main difference of this backend is that all the tasks **must be** software tasks. Additionally, it is not required to provide a list of dispatchers in the `#[app]` attribute, as RTIC will generate them at compile time.
- `riscv-clint-backend`: This backend supports devices with a CLINT peripheral. It is equivallent to `riscv-mecall-backend`, but instead of triggering exceptions, it triggers software interrupts via the `MSIP` register of the CLINT.

RTIC by example

This part of the book introduces the RTIC framework to new users by walking them through examples of increasing complexity.

All examples in this part of the book are part of the RTIC repository, found in the `examples` directory. The examples are runnable on QEMU (emulating a Cortex M3 target), thus no special hardware required to follow along.

Running an example

To run the examples with QEMU you will need the `qemu-system-arm` program. Check the embedded Rust book for instructions on how to set up an embedded development environment that includes QEMU.

To run the examples found in `examples/` locally using QEMU:

```
cargo xtask qemu
```

This runs all of the examples against the default `thumbv7m-none-eabi` device `lm3s6965`.

To limit which examples are being run, use the flag `--example <example name>`, the name being the filename of the example.

Assuming dependencies in place, running:

```
$ cargo xtask qemu --example locals
```

Yields this output:

```
   Finished dev [unoptimized + debuginfo] target(s) in 0.07s
    Running `target/debug/xtask qemu --example locals`
INFO  xtask > Testing for platform: Lm3s6965, backend: Thumbv7
INFO  xtask::run > 👟 Build example locals (thumbv7m-none-eabi, release, "test-critical-section,thumbv7-backend", in examples/lm3s6965)
INFO  xtask::run > ✅ Success.
INFO  xtask::run > 👟 Run example locals in QEMU (thumbv7m-none-eabi, release, "test-critical-section,thumbv7-backend", in examples/lm3s6965)
INFO  xtask::run > ✅ Success.
INFO  xtask::results > ✅ Success: Build example locals (thumbv7m-none-eabi, release, "test-critical-section,thumbv7-backend", in examples/lm3s6965)
INFO  xtask::results > ✅ Success: Run example locals in QEMU (thumbv7m-none-eabi, release, "test-critical-section,thumbv7-backend", in examples/lm3s6965)
INFO  xtask::results > 🚀🚀🚀 All tasks succeeded 🚀🚀🚀
```

It is great that examples are passing and this is part of the RTIC CI setup too, but for the purposes of this book we must add the `--verbose` flag, or `-v` for short to see the actual program output:

```
❯ cargo xtask qemu --verbose --example locals
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
        Running `target/debug/xtask qemu --example locals --verbose`
 DEBUG xtask > Stderr of child processes is inherited: false
 DEBUG xtask > Partial features: false
  INFO   xtask > Testing for platform: Lm3s6965, backend: Thumbv7
 INFO  xtask::run > 👟 Build example locals (thumbv7m-none-eabi, release, "test-critical-section,thumbv7-backend", in examples/lm3s6965)
 INFO  xtask::run > ✅ Success.
 INFO  xtask::run > 👟 Run example locals in QEMU (thumbv7m-none-eabi, release, "test-critical-section,thumbv7-backend", in examples/lm3s6965)
 INFO  xtask::run > ✅ Success.
  INFO   xtask::results > ✅ Success: Build example locals (thumbv7m-none-eabi, release, "test-critical-section,thumbv7-backend", in examples/lm3s6965)
     cd examples/lm3s6965 && cargo build --target thumbv7m-none-eabi --features test-critical-section,thumbv7-backend --release --example locals
 DEBUG xtask::results >
cd examples/lm3s6965 && cargo build --target thumbv7m-none-eabi --features test-critical-section,thumbv7-backend --release --example locals
Stderr:
    Finished release [optimized] target(s) in 0.02s
 INFO  xtask::results > ✅ Success: Run example locals in QEMU (thumbv7m-none-eabi, release, "test-critical-section,thumbv7-backend", in examples/lm3s6965)
    cd examples/lm3s6965 && cargo run --target thumbv7m-none-
```

```
eabi   --features   test-critical-section,thumbv7-backend   --
release --example locals
 DEBUG xtask::results >
cd examples/lm3s6965 && cargo run --target thumbv7m-none-eabi
--features test-critical-section,thumbv7-backend --release --
example locals
Stdout:
bar: local_to_bar = 1
foo: local_to_foo = 1
idle: local_to_idle = 1

Stderr:
    Finished release [optimized] target(s) in 0.02s
          Running  `qemu-system-arm  -cpu  cortex-m3  -machine
lm3s6965evb           -nographic           -semihosting-config
enable=on,target=native     -kernel     target/thumbv7m-none-
eabi/release/examples/locals`
Timer with period zero, disabling

 INFO  xtask::results > 🚀🚀🚀 All tasks succeeded 🚀🚀🚀
```

Look for the content following `Stdout:` towards the end ouf the output, the program output should have these lines:

```
bar: local_to_bar = 1
foo: local_to_foo = 1
idle: local_to_idle = 1
```

NOTE: For other useful options to `cargo xtask`, see:

```
cargo xtask qemu --help
```

The `--platform` flag allows changing which device examples are run on, currently `lm3s6965` is the best supported, work is ongoing to increase support for other devices, including both ARM and RISC-V

The `#[app]` attribute and an RTIC application

Requirements on the app attribute

All RTIC applications use the `app` attribute (`#[app(..)]`). This attribute only applies to a `mod`-item containing the RTIC application.

The `app` attribute has a mandatory `device` argument that takes a *path* as a value. This must be a full path pointing to a *peripheral access crate* (PAC) generated using `svd2rust` **v0.14.x** or newer.

The `app` attribute will expand into a suitable entry point and thus replaces the use of the `cortex_m_rt::entry` attribute.

Structure and zero-cost concurrency

An RTIC `app` is an executable system model for single-core applications, declaring a set of `local` and `shared` resources operated on by a set of `init`, `idle`, *hardware* and *software* tasks.

- `init` runs before any other task, and returns the `local` and `shared` resources.
- Tasks (both hardware and software) run preemptively based on their associated static priority.
- Hardware tasks are bound to underlying hardware interrupts.
- Software tasks are schedulied by an set of asynchronous executors, one for each software task priority.
- `idle` has the lowest priority, and can be used for background work, and/or to put the system to sleep until it is woken by some event.

At compile time the task/resource model is analyzed under the Stack Resource Policy (SRP) and executable code generated with the following outstanding properties:

- Guaranteed race-free resource access and deadlock-free execution on a single-shared stack.
- Hardware task scheduling is performed directly by the hardware.
- Software task scheduling is performed by auto generated async executors tailored to the application.

Overall, the generated code infers no additional overhead in comparison to a hand-written implementation, thus in Rust terms RTIC offers a zero-cost abstraction to concurrency.

Priority

Priorities in RTIC are specified using the `priority = N` (where N is a positive number) argument passed to the `#[task]` attribute. All `#[task]`s can have a priority. If the priority of a task is not specified, it is set to the default value of 0.

Priorities in RTIC follow a higher value = more important scheme. For examples, a task with priority 2 will preempt a task with priority 1.

An RTIC application example

To give a taste of RTIC, the following example contains commonly used features. In the following sections we will go through each feature in detail.

```
//! examples/common.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [UART0, UART1])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};

    #[shared]
    struct Shared {}

    #[local]
    struct Local {
        local_to_foo: i64,
        local_to_bar: i64,
        local_to_idle: i64,
    }

      // `#[init]` cannot access locals from the `#[local]`
struct as they are initialized here.
    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        foo::spawn().unwrap();
        bar::spawn().unwrap();
```

```
        (
            Shared {},
            // initial values for the `#[local]` resources
            Local {
                local_to_foo: 0,
                local_to_bar: 0,
                local_to_idle: 0,
            },
        )
    }

    // `local_to_idle` can only be accessed from this context
    #[idle(local = [local_to_idle])]
    fn idle(cx: idle::Context) -> ! {
        let local_to_idle = cx.local.local_to_idle;
        *local_to_idle += 1;

        hprintln!("idle: local_to_idle = {}", local_to_idle);

             debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator

                  //  error:  no  `local_to_foo`  field  in
`idle::LocalResources`
        // _cx.local.local_to_foo += 1;

                  //  error:  no  `local_to_bar`  field  in
`idle::LocalResources`
        // _cx.local.local_to_bar += 1;

        loop {
            cortex_m::asm::nop();
        }
    }

    // `local_to_foo` can only be accessed from this context
```

```
    #[task(local = [local_to_foo], priority = 1)]
    async fn foo(cx: foo::Context) {
        let local_to_foo = cx.local.local_to_foo;
        *local_to_foo += 1;

                    //  error:  no  `local_to_bar`  field  in
`foo::LocalResources`
        // cx.local.local_to_bar += 1;

        hprintln!("foo: local_to_foo = {}", local_to_foo);
    }

    // `local_to_bar` can only be accessed from this context
    #[task(local = [local_to_bar], priority = 1)]
    async fn bar(cx: bar::Context) {
        let local_to_bar = cx.local.local_to_bar;
        *local_to_bar += 1;

                    //  error:  no  `local_to_foo`  field  in
`bar::LocalResources`
        // cx.local.local_to_foo += 1;

        hprintln!("bar: local_to_bar = {}", local_to_bar);
    }
}
```

Hardware tasks

At its core RTIC is using a hardware interrupt controller (ARM NVIC on cortex-m) to schedule and start execution of tasks. All tasks except `pre-init` (a hidden "task"), `#[init]` and `#[idle]` run as interrupt handlers.

To bind a task to an interrupt, use the `#[task]` attribute argument `binds = InterruptName`. This task then becomes the interrupt handler for this hardware interrupt vector.

All tasks bound to an explicit interrupt are called *hardware tasks* since they start execution in reaction to a hardware event.

Specifying a non-existing interrupt name will cause a compilation error. The interrupt names are commonly defined by PAC or HAL crates.

Any available interrupt vector should work. Specific devices may bind specific interrupt priorities to specific interrupt vectors outside user code control. See for example the nRF “softdevice”.

Beware of using interrupt vectors that are used internally by hardware features; RTIC is unaware of such hardware specific details.

Example

The example below demonstrates the use of the `#[task(binds = InterruptName)]` attribute to declare a hardware task bound to an interrupt handler.

```
//! examples/hardware.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965)]
mod app {
    use cortex_m_semihosting::{debug, hprintln};
    use lm3s6965::Interrupt;

    #[shared]
    struct Shared {}

    #[local]
    struct Local {}

    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        // Pends the UART0 interrupt but its handler won't run
until *after*
        // `init` returns because interrupts are disabled
              rtic::pend(Interrupt::UART0);  //  equivalent  to
NVIC::pend

        hprintln!("init");
```

```
        (Shared {}, Local {})
    }

    #[idle]
    fn idle(_: idle::Context) -> ! {
         // interrupts are enabled again; the `UART0` handler
runs at this point

        hprintln!("idle");

        // Some backends provide a manual way of pending an
        // interrupt.
        rtic::pend(Interrupt::UART0);

        loop {
            cortex_m::asm::nop();
               debug::exit(debug::EXIT_SUCCESS); // Exit QEMU
simulator
        }
    }

    #[task(binds = UART0, local = [times: u32 = 0])]
    fn uart0(cx: uart0::Context) {
        // Safe access to local `static mut` variable
        *cx.local.times += 1;

        hprintln!(
            "UART0 called {} time{}",
            *cx.local.times,
            if *cx.local.times > 1 { "s" } else { "" }
        );
    }
}
$ cargo xtask qemu --verbose --example hardware
```

```
init
UART0 called 1 time
idle
UART0 called 2 times
```

Software tasks & spawn

The RTIC concept of a software task shares a lot with that of hardware tasks. The core difference is that a software task is not explicitly bound to a specific interrupt vector, but rather bound to a "dispatcher" interrupt vector running at the intended priority of the software task (see below).

Similarly to *hardware* tasks, the `#[task]` attribute used on a function declare it as a task. The absence of a `binds = InterruptName` argument to the attribute declares the function as a *software task*.

The static method `task_name::spawn()` spawns (starts) a software task and given that there are no higher priority tasks running the task will start executing directly.

The *software* task itself is given as an `async` Rust function, which allows the user to optionally `await` future events. This allows to blend reactive programming (by means of *hardware* tasks) with sequential programming (by means of *software* tasks).

While *hardware* tasks are assumed to run-to-completion (and return), *software* tasks may be started (`spawned`) once and run forever, on the condition that any loop (execution path) is broken by at least one `await` (yielding operation).

Dispatchers

All *software* tasks at the same priority level share an interrupt handler acting as an async executor dispatching the software tasks. This list of dispatchers, `dispatchers = [FreeInterrupt1, FreeInterrupt2, ...]` is an argument to the `#[app]` attribute, where you define the set of free and usable interrupts.

Each interrupt vector acting as dispatcher gets assigned to one priority level meaning that the list of dispatchers need to cover all priority levels used by software tasks.

Example: The `dispatchers =` argument needs to have at least 3 entries for an application using three different priorities for software tasks.

The framework will give a compilation error if there are not enough dispatchers provided, or if a clash occurs between the list of dispatchers and interrupts bound to *hardware* tasks.

See the following example:

```
//! examples/spawn.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [SSI0])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};

    #[shared]
    struct Shared {}

    #[local]
```

```
    struct Local {}

    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        hprintln!("init");
        foo::spawn().unwrap();

        (Shared {}, Local {})
    }

    #[task]
    async fn foo(_: foo::Context) {
        hprintln!("foo");

            debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator
    }
}
```

```
$ cargo xtask qemu --verbose --example spawn
```

```
init
foo
```

You may `spawn` a *software* task again, given that it has run-to-completion (returned).

In the below example, we `spawn` the *software* task `foo` from the `idle` task. Since the priority of the *software* task is 1 (higher than `idle`), the dispatcher will execute `foo` (preempting `idle`). Since `foo` runs-to-completion. It is ok to `spawn` the `foo` task again.

Technically the async executor will `poll` the `foo` *future* which in this case leaves the *future* in a *completed* state.

```
//! examples/spawn_loop.rs

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

```
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [SSI0])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};

    #[shared]
    struct Shared {}

    #[local]
    struct Local {}

    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        hprintln!("init");

        (Shared {}, Local {})
    }

    #[idle]
    fn idle(_: idle::Context) -> ! {
        for _ in 0..3 {
            foo::spawn().unwrap();
            hprintln!("idle");
        }
              debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator
        loop {}
    }

    #[task(priority = 1)]
    async fn foo(_: foo::Context) {
```

```
        hprintln!("foo");
    }
}
$ cargo xtask qemu --verbose --example spawn_loop
init
foo
idle
foo
idle
foo
idle
```

An attempt to `spawn` an already spawned task (running) task will result in an error. Notice, the that the error is reported before the `foo` task is actually run. This is since, the actual execution of the *software* task is handled by the dispatcher interrupt (`SSI0`), which is not enabled until we exit the `init` task. (Remember, `init` runs in a critical section, i.e. all interrupts being disabled.)

Technically, a `spawn` to a *future* that is not in *completed* state is considered an error.

```
//! examples/spawn_err.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [SSI0])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};

    #[shared]
```

```
    struct Shared {}

    #[local]
    struct Local {}

    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        hprintln!("init");
        foo::spawn().unwrap();
        match foo::spawn() {
            Ok(_) => {}
                Err(()) => hprintln!("Cannot spawn a spawned
(running) task!"),
        }

        (Shared {}, Local {})
    }

    #[task]
    async fn foo(_: foo::Context) {
        hprintln!("foo");

             debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator
    }
}
```

```
$ cargo xtask qemu --verbose --example spawn_err
```

```
init
Cannot spawn a spawned (running) task!
foo
```

Passing arguments

You can also pass arguments at spawn as follows.

```
//! examples/spawn_arguments.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [SSI0])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};

    #[shared]
    struct Shared {}

    #[local]
    struct Local {}

    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        foo::spawn(1, 1).unwrap();
        assert!(foo::spawn(1, 4).is_err()); // The capacity of
`foo` is reached

        (Shared {}, Local {})
    }

    #[task]
    async fn foo(_c: foo::Context, x: i32, y: u32) {
        hprintln!("foo {}, {}", x, y);
```

```
                debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator
    }
}
$ cargo xtask qemu --verbose --example spawn_arguments
foo 1, 1
```

Divergent tasks

A task can have one of two signatures: `async fn({name}::Context, ..)` or `async fn({name}::Context, ..) -> !`. The latter defines a *divergent* task — one that never returns. The key advantage of divergent tasks is that they receive a `'static` context, and `local` resources have `'static` lifetime. Additionally, using this signature makes the task's intent explicit, clearly distinguishing between short-lived tasks and those that run indefinitely. Be mindful not to starve other tasks at the same priority level by ensuring you yield control with `.await`.

Priority zero tasks

In RTIC tasks run preemptively to each other, with priority zero (0) the lowest priority. You can use priority zero tasks for background work, without any strict real-time requirements.

Conceptually, one can see such tasks as running in the `main` thread of the application, thus the resources associated are not required the Send bound.

```
//! examples/zero-prio-task.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use core::marker::PhantomData;
use panic_semihosting as _;

/// Does not impl send
pub struct NotSend {
    _0: PhantomData<*const ()>,
}

#[rtic::app(device = lm3s6965, peripherals = true)]
mod app {
    use super::NotSend;
    use core::marker::PhantomData;
    use cortex_m_semihosting::{debug, hprintln};

    #[shared]
    struct Shared {
        x: NotSend,
    }
```

```
    #[local]
    struct Local {
        y: NotSend,
    }

    #[init]
    fn init(_cx: init::Context) -> (Shared, Local) {
        hprintln!("init");

        async_task::spawn().unwrap();
        async_task2::spawn().unwrap();

        (
            Shared {
                x: NotSend { _0: PhantomData },
            },
            Local {
                y: NotSend { _0: PhantomData },
            },
        )
    }

    #[task(priority = 0, shared = [x], local = [y])]
    async fn async_task(_: async_task::Context) {
        hprintln!("hello from async");
    }

    #[task(priority = 0, shared = [x])]
    async fn async_task2(_: async_task2::Context) {
        hprintln!("hello from async2");

                debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator
    }
}
```

```
$ cargo xtask qemu --verbose --example zero-prio-task
init
hello from async
hello from async2
```

> **Notice**: *software* task at zero priority cannot co-exist with the [idle] task. The reason is that `idle` is running as a non-returning Rust function at priority zero. Thus there would be no way for an executor at priority zero to give control to *software* tasks at the same priority.

Application side safety: Technically, the RTIC framework ensures that `poll` is never executed on any *software* task with *completed* future, thus adhering to the soundness rules of async Rust.

Resource usage

The RTIC framework manages shared and task local resources allowing persistent data storage and safe accesses without the use of `unsafe` code.

RTIC resources are visible only to functions declared within the `#[app]` module and the framework gives the user complete control (on a per-task basis) over resource accessibility.

Declaration of system-wide resources is done by annotating **two** `struct`s within the `#[app]` module with the attribute `#[local]` and `#[shared]`. Each field in these structures corresponds to a different resource (identified by field name). The difference between these two sets of resources will be covered below.

Each task must declare the resources it intends to access in its corresponding metadata attribute using the `local` and `shared` arguments. Each argument takes a list of resource identifiers. The listed resources are made available to the context under the `local` and `shared` fields of the `Context` structure.

The `init` task returns the initial values for the system-wide (`#[shared]` and `#[local]`) resources.

`#[local]` resources

`#[local]` resources are locally accessible to a specific task, meaning that only that task can access the resource and does so without locks or critical sections. This allows for the resources, commonly drivers or large objects, to be initialized in `#[init]` and then be passed to a specific task.

Thus, a task `#[local]` resource can only be accessed by one singular task. Attempting to assign the same `#[local]` resource to more than one task is a compile-time error.

Types of `#[local]` resources must implement a `Send` trait as they are being sent from `init` to a target task, crossing a thread boundary.

The example application shown below contains three tasks `foo`, `bar` and `idle`, each having access to its own `#[local]` resource.

```
//! examples/locals.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [UART0, UART1])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};

    #[shared]
    struct Shared {}

    #[local]
    struct Local {
        local_to_foo: i64,
```

```
        local_to_bar: i64,
        local_to_idle: i64,
    }

      // `#[init]` cannot access locals from the `#[local]`
struct as they are initialized here.
    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        foo::spawn().unwrap();
        bar::spawn().unwrap();

        (
            Shared {},
            // initial values for the `#[local]` resources
            Local {
                local_to_foo: 0,
                local_to_bar: 0,
                local_to_idle: 0,
            },
        )
    }

    // `local_to_idle` can only be accessed from this context
    #[idle(local = [local_to_idle])]
    fn idle(cx: idle::Context) -> ! {
        let local_to_idle = cx.local.local_to_idle;
        *local_to_idle += 1;

        hprintln!("idle: local_to_idle = {}", local_to_idle);

              debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator

                  //  error:  no  `local_to_foo`  field  in
`idle::LocalResources`
        // _cx.local.local_to_foo += 1;
```

```
                    //  error:  no  `local_to_bar`  field  in
`idle::LocalResources`
        // _cx.local.local_to_bar += 1;

        loop {
            cortex_m::asm::nop();
        }
    }

    // `local_to_foo` can only be accessed from this context
    #[task(local = [local_to_foo], priority = 1)]
    async fn foo(cx: foo::Context) {
        let local_to_foo = cx.local.local_to_foo;
        *local_to_foo += 1;

                    //  error:  no  `local_to_bar`  field  in
`foo::LocalResources`
        // cx.local.local_to_bar += 1;

        hprintln!("foo: local_to_foo = {}", local_to_foo);
    }

    // `local_to_bar` can only be accessed from this context
    #[task(local = [local_to_bar], priority = 1)]
    async fn bar(cx: bar::Context) {
        let local_to_bar = cx.local.local_to_bar;
        *local_to_bar += 1;

                    //  error:  no  `local_to_foo`  field  in
`bar::LocalResources`
        // cx.local.local_to_foo += 1;

        hprintln!("bar: local_to_bar = {}", local_to_bar);
    }
}
```

Running the example:

```
$ cargo xtask qemu --verbose --example locals

bar: local_to_bar = 1
foo: local_to_foo = 1
idle: local_to_idle = 1
```

Local resources in `#[init]` and `#[idle]` have `'static` lifetimes. This is safe since both tasks are not re-entrant.

Task local initialized resources

Local resources can also be specified directly in the resource claim like so: `#[task(local = [my_var: TYPE = INITIAL_VALUE, ...])]`; this allows for creating locals which do no need to be initialized in `#[init]`.

Types of `#[task(local = [..])]` resources have to be neither `Send` nor `Sync` as they are not crossing any thread boundary.

In the example below the different uses and lifetimes are shown:

```
//! examples/declared_locals.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965)]
mod app {
    use cortex_m_semihosting::debug;

    #[shared]
    struct Shared {}

    #[local]
```

```
    struct Local {}

    #[init(local = [a: u32 = 0])]
    fn init(cx: init::Context) -> (Shared, Local) {
        // Locals in `#[init]` have 'static lifetime
        let _a: &'static mut u32 = cx.local.a;

             debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator

        (Shared {}, Local {})
    }

    #[idle(local = [a: u32 = 0])]
    fn idle(cx: idle::Context) -> ! {
        // Locals in `#[idle]` have 'static lifetime
        let _a: &'static mut u32 = cx.local.a;

        loop {}
    }

    #[task(binds = UART0, local = [a: u32 = 0])]
    fn foo(cx: foo::Context) {
        // Locals in `#[task]`s have a local lifetime
        let _a: &mut u32 = cx.local.a;

         // error: explicit lifetime required in the type of
`cx`
        // let _a: &'static mut u32 = cx.local.a;
    }
}
```

You can run the application, but as the example is designed merely to showcase the lifetime properties there is no output (it suffices to build the application).

```
$ cargo build --target thumbv7m-none-eabi --example declared_locals
```

`#[shared]` resources and `lock`

Critical sections are required to access `#[shared]` resources in a data race-free manner and to achieve this the `shared` field of the passed `Context` implements the `Mutex` trait for each shared resource accessible to the task. This trait has only one method, `lock`, which runs its closure argument in a critical section.

The critical section created by the `lock` API is based on dynamic priorities: it temporarily raises the dynamic priority of the context to a *ceiling* priority that prevents other tasks from preempting the critical section. This synchronization protocol is known as the Immediate Ceiling Priority Protocol (ICPP), and complies with Stack Resource Policy (SRP) based scheduling of RTIC.

In the example below we have three interrupt handlers with priorities ranging from one to three. The two handlers with the lower priorities contend for a `shared` resource and need to succeed in locking the resource in order to access its data. The highest priority handler, which does not access the `shared` resource, is free to preempt a critical section created by the lowest priority handler.

```
//! examples/lock.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [GPIOA, GPIOB, GPIOC])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};
```

```
    #[shared]
    struct Shared {
        shared: u32,
    }

    #[local]
    struct Local {}

    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        foo::spawn().unwrap();

        (Shared { shared: 0 }, Local {})
    }

    // when omitted priority is assumed to be `1`
    #[task(shared = [shared])]
    async fn foo(mut c: foo::Context) {
        hprintln!("A");

        // the lower priority task requires a critical section
to access the data
        c.shared.shared.lock(|shared| {
            // data can only be modified within this critical
section (closure)
            *shared += 1;

                // bar will *not* run right now due to the
critical section
            bar::spawn().unwrap();

            hprintln!("B - shared = {}", *shared);

               // baz does not contend for `shared` so it's
allowed to run now
            baz::spawn().unwrap();
```

```
        });

        // critical section is over: bar can now start

        hprintln!("E");

             debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator
    }

    #[task(priority = 2, shared = [shared])]
    async fn bar(mut c: bar::Context) {
        // the higher priority task does still need a critical
section
        let shared = c.shared.shared.lock(|shared| {
            *shared += 1;

            *shared
        });

        hprintln!("D - shared = {}", shared);
    }

    #[task(priority = 3)]
    async fn baz(_: baz::Context) {
        hprintln!("C");
    }
}
```

```
$ cargo xtask qemu --verbose --example lock
```

```
A
B - shared = 1
C
D - shared = 2
E
```

Types of `#[shared]` resources have to be `Send`.

Multi-lock

As an extension to `lock`, and to reduce rightward drift, locks can be taken as tuples. The following examples show this in use:

```
//! examples/mutlilock.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [GPIOA])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};

    #[shared]
    struct Shared {
        shared1: u32,
        shared2: u32,
        shared3: u32,
    }

    #[local]
    struct Local {}

    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        locks::spawn().unwrap();

        (
            Shared {
                shared1: 0,
```

```
                shared2: 0,
                shared3: 0,
            },
            Local {},
        )
    }

    // when omitted priority is assumed to be `1`
    #[task(shared = [shared1, shared2, shared3])]
    async fn locks(c: locks::Context) {
        let s1 = c.shared.shared1;
        let s2 = c.shared.shared2;
        let s3 = c.shared.shared3;

        (s1, s2, s3).lock(|s1, s2, s3| {
            *s1 += 1;
            *s2 += 1;
            *s3 += 1;

              hprintln!("Multiple locks, s1: {}, s2: {}, s3:
{}", *s1, *s2, *s3);
        });

             debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator
    }
}
$ cargo xtask qemu --verbose --example multilock
Multiple locks, s1: 1, s2: 1, s3: 1
```

Only shared (`&-`) access

By default, the framework assumes that all tasks require exclusive mutable access (`&mut-`) to resources, but it is possible to specify that a task only requires shared access (`&-`) to a resource using the `&resource_name` syntax in the `shared` list.

The advantage of specifying shared access (`&-`) to a resource is that no locks are required to access the resource even if the resource is contended by more than one task running at different priorities. The downside is that the task only gets a shared reference (`&-`) to the resource, limiting the operations it can perform on it, but where a shared reference is enough this approach reduces the number of required locks. In addition to simple immutable data, this shared access can be useful where the resource type safely implements interior mutability, with appropriate locking or atomic operations of its own.

Note that in this release of RTIC it is not possible to request both exclusive access (`&mut-`) and shared access (`&-`) to the *same* resource from different tasks. Attempting to do so will result in a compile error.

In the example below a key (e.g. a cryptographic key) is loaded (or created) at runtime (returned by `init`) and then used from two tasks that run at different priorities without any kind of lock.

```
//! examples/only-shared-access.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [UART0, UART1])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};
```

```
    #[shared]
    struct Shared {
        key: u32,
    }

    #[local]
    struct Local {}

    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        foo::spawn().unwrap();
        bar::spawn().unwrap();

        (Shared { key: 0xdeadbeef }, Local {})
    }

    #[task(shared = [&key])]
    async fn foo(cx: foo::Context) {
        let key: &u32 = cx.shared.key;
        hprintln!("foo(key = {:#x})", key);

            debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator
    }

    #[task(priority = 2, shared = [&key])]
    async fn bar(cx: bar::Context) {
        hprintln!("bar(key = {:#x})", cx.shared.key);
    }
}
$ cargo xtask qemu --verbose --example only-shared-access
bar(key = 0xdeadbeef)
foo(key = 0xdeadbeef)
```

Lock-free access of shared resources

A critical section is *not* required to access a `#[shared]` resource that's only accessed by tasks running at the *same* priority. In this case, you can opt out of the `lock` API by adding the `#[lock_free]` field-level attribute to the resource declaration (see example below).

To adhere to the Rust aliasing rule, a resource may be either accessed through multiple immutable references or a singe mutable reference (but not both at the same time).

Using `#[lock_free]` on resources shared by tasks running at different priorities will result in a *compile-time* error -- not using the `lock` API would violate the aforementioned alias rule. Similarly, for each priority there can be only a single *software* task accessing a shared resource (as an `async` task may yield execution to other *software* or *hardware* tasks running at the same priority). However, under this single-task restriction, we make the observation that the resource is in effect no longer `shared` but rather `local`. Thus, using a `#[lock_free]` shared resource will result in a *compile-time* error -- where applicable, use a `#[local]` resource instead.

```
//! examples/lock-free.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965)]
mod app {
    use cortex_m_semihosting::{debug, hprintln};
    use lm3s6965::Interrupt;

    #[shared]
```

```
    struct Shared {
        #[lock_free] // <- lock-free shared resource
        counter: u64,
    }

    #[local]
    struct Local {}

    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        rtic::pend(Interrupt::UART0);

        (Shared { counter: 0 }, Local {})
    }

     #[task(binds = UART0, shared = [counter])] // <- same
priority
    fn foo(c: foo::Context) {
        rtic::pend(Interrupt::UART1);

        *c.shared.counter += 1; // <- no lock API required
        let counter = *c.shared.counter;
        hprintln!("  foo = {}", counter);
    }

     #[task(binds = UART1, shared = [counter])] // <- same
priority
    fn bar(c: bar::Context) {
        rtic::pend(Interrupt::UART0);
        *c.shared.counter += 1; // <- no lock API required
        let counter = *c.shared.counter;
        hprintln!("  bar = {}", counter);

             debug::exit(debug::EXIT_SUCCESS); // Exit QEMU
simulator
```

```
    }
}
$ cargo xtask qemu --verbose --example lock-free
  foo = 1
  bar = 2
```

App initialization and the `#[init]` task

An RTIC application requires an `init` task setting up the system. The corresponding `init` function must have the signature `fn(init::Context) -> (Shared, Local)`, where `Shared` and `Local` are resource structures defined by the user.

The `init` task executes after system reset, [after an optionally defined `pre-init` code section][1] and an always occurring internal RTIC initialization.

The `init` and optional `pre-init` tasks runs *with interrupts disabled* and have exclusive access to Cortex-M (the `bare_metal::CriticalSection` token is available as `cs`).

Device specific peripherals are available through the `core` and `device` fields of `init::Context`.

1

https://docs.rs/cortex-m-rt/latest/cortex_m_rt/attr.pre_init.html

Example

The example below shows the types of the `core`, `device` and `cs` fields, and showcases the use of a `local` variable with `'static` lifetime. Such variables can be delegated from the `init` task to other tasks of the RTIC application.

The `device` field is only available when the `peripherals` argument is set to the default value `true`. In the rare case you want to implement an ultra-slim application you can explicitly set `peripherals` to `false`.

```
//! examples/init.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, peripherals = true)]
mod app {
    use cortex_m_semihosting::{debug, hprintln};

    #[shared]
    struct Shared {}

    #[local]
    struct Local {}

    #[init(local = [x: u32 = 0])]
    fn init(cx: init::Context) -> (Shared, Local) {
        // Cortex-M peripherals
        let _core: cortex_m::Peripherals = cx.core;
```

```
        // Device specific peripherals
        let _device: lm3s6965::Peripherals = cx.device;

        // Locals in `init` have 'static lifetime
        let _x: &'static mut u32 = cx.local.x;

        // Access to the critical section token,
        // to indicate that this is a critical section
        let _cs_token: bare_metal::CriticalSection = cx.cs;

        hprintln!("init");

             debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator

        (Shared {}, Local {})
    }
}
```

Running the example will print `init` to the console and then exit the QEMU process.

```
$ cargo xtask qemu --verbose --example init
init
```

The background task `#[idle]`

A function marked with the `idle` attribute can optionally appear in the module. This becomes the special *idle task* and must have signature `fn(idle::Context) -> !`.

When present, the runtime will execute the `idle` task after `init`. Unlike `init`, `idle` will run *with interrupts enabled* and must never return, as the `-> !` function signature indicates. The Rust type `!` means "never".

Like in `init`, locally declared resources will have `'static` lifetimes that are safe to access.

The example below shows that `idle` runs after `init`.

```
//! examples/idle.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965)]
mod app {
    use cortex_m_semihosting::{debug, hprintln};

    #[shared]
    struct Shared {}

    #[local]
    struct Local {}

    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
```

```
        hprintln!("init");

        (Shared {}, Local {})
    }

    #[idle(local = [x: u32 = 0])]
    fn idle(cx: idle::Context) -> ! {
        // Locals in idle have lifetime 'static
        let _x: &'static mut u32 = cx.local.x;

        hprintln!("idle");

             debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator

        loop {
            cortex_m::asm::nop();
        }
    }
}
```

```
$ cargo xtask qemu --verbose --example idle
```

```
init
idle
```

By default, the RTIC `idle` task does not try to optimize for any specific targets.

A common useful optimization is to enable the SLEEPONEXIT and allow the MCU to enter sleep when reaching `idle`.

> **Caution**: some hardware unless configured disables the debug unit during sleep mode.
>
> Consult your hardware specific documentation as this is outside the scope of RTIC.

The following example shows how to enable sleep by setting the `SLEEPONEXIT` and providing a custom `idle` task replacing the default

`nop()` with `wfi()`.

```
//! examples/idle-wfi.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965)]
mod app {
    use cortex_m_semihosting::{debug, hprintln};

    #[shared]
    struct Shared {}

    #[local]
    struct Local {}

    #[init]
    fn init(mut cx: init::Context) -> (Shared, Local) {
        hprintln!("init");

         // Set the ARM SLEEPONEXIT bit to go to sleep after
handling interrupts
                                                  //      See
https://developer.arm.com/docs/100737/0100/power-
management/sleep-mode/sleep-on-exit-bit
        cx.core.SCB.set_sleepdeep();

        (Shared {}, Local {})
    }
```

```
    #[idle(local = [x: u32 = 0])]
    fn idle(cx: idle::Context) -> ! {
        // Locals in idle have lifetime 'static
        let _x: &'static mut u32 = cx.local.x;

        hprintln!("idle");

              debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator

        loop {
               // Now Wait For Interrupt is used instead of a
busy-wait loop
            // to allow MCU to sleep between interrupts
                                                              //
https://developer.arm.com/documentation/ddi0406/c/Application-
Level-Architecture/Instruction-Details/Alphabetical-list-of-
instructions/WFI
            rtic::export::wfi()
        }
    }
}
```

```
$ cargo xtask qemu --verbose --example idle-wfi
```

```
init
idle
```

Notice: The `idle` task cannot be used together with *software* tasks running at priority zero. The reason is that `idle` is running as a non-returning Rust function at priority zero. Thus there would be no way for an executor at priority zero to give control to *software* tasks at the same priority.

Communication over channels.

Channels can be used to communicate data between running tasks. The channel is essentially a wait queue, allowing tasks with multiple producers and a single receiver. A channel is constructed in the `init` task and backed by statically allocated memory. Send and receive endpoints are distributed to *software* tasks:

```
...
const CAPACITY: usize = 5;
#[init]
    fn init(_: init::Context) -> (Shared, Local) {
        let (s, r) = make_channel!(u32, CAPACITY);
        receiver::spawn(r).unwrap();
        sender1::spawn(s.clone()).unwrap();
        sender2::spawn(s.clone()).unwrap();
        ...
```

In this case the channel holds data of `u32` type with a capacity of 5 elements.

Channels can also be used from *hardware* tasks, but only in a non-`async` manner using the Try API.

Sending data

The `send` method post a message on the channel as shown below:

```
#[task]
async fn sender1(_c: sender1::Context, mut sender: Sender<'static, u32, CAPACITY>) {
    hprintln!("Sender 1 sending: 1");
    sender.send(1).await.unwrap();
}
```

Receiving data

The receiver can `await` incoming messages:

```
#[task]
async fn receiver(_c: receiver::Context, mut receiver: Receiver<'static, u32, CAPACITY>) {
    while let Ok(val) = receiver.recv().await {
        hprintln!("Receiver got: {}", val);
        ...
    }
}
```

Channels are implemented using a small (global) *Critical Section* (CS) for protection against race-conditions. The user must provide an CS implementation. Compiling the examples given the `--features test-critical-section` gives one possible implementation.

For a complete example:

```
//! examples/async-channel.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [SSI0])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};
    use rtic_sync::{channel::*, make_channel};

    #[shared]
    struct Shared {}
```

```
    #[local]
    struct Local {}

    const CAPACITY: usize = 5;
    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        let (s, r) = make_channel!(u32, CAPACITY);

        receiver::spawn(r).unwrap();
        sender1::spawn(s.clone()).unwrap();
        sender2::spawn(s.clone()).unwrap();
        sender3::spawn(s).unwrap();

        (Shared {}, Local {})
    }

    #[task]
      async fn receiver(_c: receiver::Context, mut receiver:
Receiver<'static, u32, CAPACITY>) {
        while let Ok(val) = receiver.recv().await {
            hprintln!("Receiver got: {}", val);
            if val == 3 {
                debug::exit(debug::EXIT_SUCCESS); // Exit QEMU
simulator
            }
        }
    }

    #[task]
       async fn sender1(_c: sender1::Context, mut sender:
Sender<'static, u32, CAPACITY>) {
        hprintln!("Sender 1 sending: 1");
        sender.send(1).await.unwrap();
    }

    #[task]
```

```
        async fn sender2(_c: sender2::Context, mut sender:
Sender<'static, u32, CAPACITY>) {
        hprintln!("Sender 2 sending: 2");
        sender.send(2).await.unwrap();
    }

    #[task]
        async fn sender3(_c: sender3::Context, mut sender:
Sender<'static, u32, CAPACITY>) {
        hprintln!("Sender 3 sending: 3");
        sender.send(3).await.unwrap();
    }
}
```

```
$ cargo xtask qemu --verbose --example async-channel --
features test-critical-section
```

```
Sender 1 sending: 1
Sender 2 sending: 2
Sender 3 sending: 3
Receiver got: 1
Receiver got: 2
Receiver got: 3
```

Also sender endpoint can be awaited. In case the channel capacity has not yet been reached, `await`-ing the sender can progress immediately, while in the case the capacity is reached, the sender is blocked until there is free space in the queue. In this way data is never lost.

In the following example the `CAPACITY` has been reduced to 1, forcing sender tasks to wait until the data in the channel has been received.

```
//! examples/async-channel-done.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]
```

```
use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [SSI0])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};
    use rtic_sync::{channel::*, make_channel};

    #[shared]
    struct Shared {}

    #[local]
    struct Local {}

    const CAPACITY: usize = 1;
    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        let (s, r) = make_channel!(u32, CAPACITY);

        receiver::spawn(r).unwrap();
        sender1::spawn(s.clone()).unwrap();
        sender2::spawn(s.clone()).unwrap();
        sender3::spawn(s).unwrap();

        (Shared {}, Local {})
    }

    #[task]
      async fn receiver(_c: receiver::Context, mut receiver:
Receiver<'static, u32, CAPACITY>) {
        while let Ok(val) = receiver.recv().await {
            hprintln!("Receiver got: {}", val);
            if val == 3 {
                debug::exit(debug::EXIT_SUCCESS); // Exit QEMU
simulator
            }
```

```
        }
    }

    #[task]
        async fn sender1(_c: sender1::Context, mut sender:
Sender<'static, u32, CAPACITY>) {
        hprintln!("Sender 1 sending: 1");
        sender.send(1).await.unwrap();
        hprintln!("Sender 1 done");
    }

    #[task]
        async fn sender2(_c: sender2::Context, mut sender:
Sender<'static, u32, CAPACITY>) {
        hprintln!("Sender 2 sending: 2");
        sender.send(2).await.unwrap();
        hprintln!("Sender 2 done");
    }

    #[task]
        async fn sender3(_c: sender3::Context, mut sender:
Sender<'static, u32, CAPACITY>) {
        hprintln!("Sender 3 sending: 3");
        sender.send(3).await.unwrap();
        hprintln!("Sender 3 done");
    }
}
```

Looking at the output, we find that `Sender 2` will wait until the data sent by `Sender 1` as been received.

NOTICE *Software* tasks at the same priority are executed asynchronously to each other, thus **NO** strict order can be assumed. (The presented order here applies only to the current implementation, and may change between RTIC framework releases.)

```
$ cargo xtask qemu --verbose --example async-channel-done --
features test-critical-section
Sender 1 sending: 1
Sender 1 done
Sender 2 sending: 2
Sender 3 sending: 3
Receiver got: 1
Sender 2 done
Receiver got: 2
Sender 3 done
Receiver got: 3
```

Error handling

In case all senders have been dropped `await`-ing on an empty receiver channel results in an error. This allows to gracefully implement different types of shutdown operations.

```
//! examples/async-channel-no-sender.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [SSI0])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};
    use rtic_sync::{channel::*, make_channel};

    #[shared]
    struct Shared {}

    #[local]
    struct Local {}

    const CAPACITY: usize = 1;
    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        let (_s, r) = make_channel!(u32, CAPACITY);

        receiver::spawn(r).unwrap();

        (Shared {}, Local {})
    }
```

```
    #[task]
      async fn receiver(_c: receiver::Context, mut receiver:
Receiver<'static, u32, CAPACITY>) {
                          hprintln!("Receiver   got:   {:?}",
receiver.recv().await);

              debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator
    }
}
```

```
$ cargo xtask qemu --verbose --example async-channel-no-sender
--features test-critical-section
```

```
Receiver got: Err(NoSender)
```

Similarly, `await`-ing on a send channel results in an error in case the receiver has been dropped. This allows to gracefully implement application level error handling.

The resulting error returns the data back to the sender, allowing the sender to take appropriate action (e.g., storing the data to later retry sending it).

```
//! examples/async-channel-no-receiver.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [SSI0])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};
    use rtic_sync::{channel::*, make_channel};
```

```
    #[shared]
    struct Shared {}

    #[local]
    struct Local {}

    const CAPACITY: usize = 1;
    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        let (s, _r) = make_channel!(u32, CAPACITY);

        sender1::spawn(s.clone()).unwrap();

        (Shared {}, Local {})
    }

    #[task]
        async fn sender1(_c: sender1::Context, mut sender: Sender<'static, u32, CAPACITY>) {
                    hprintln!("Sender 1 sending: 1 {:?}", sender.send(1).await);
                debug::exit(debug::EXIT_SUCCESS); // Exit QEMU simulator
    }
}
```

```
$ cargo xtask qemu --verbose --example async-channel-no-receiver --features test-critical-section
```

```
Sender 1 sending: 1 Err(NoReceiver(1))
```

Try API

Using the Try API, you can send or receive data from or to a channel without requiring that the operation succeeds, and in non-`async` contexts.

This API is exposed through `Receiver::try_recv` and `Sender::try_send`.

```
//! examples/async-channel-try.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [SSI0])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};
    use rtic_sync::{channel::*, make_channel};

    #[shared]
    struct Shared {}

    #[local]
    struct Local {
        sender: Sender<'static, u32, CAPACITY>,
    }

    const CAPACITY: usize = 1;
    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        let (s, r) = make_channel!(u32, CAPACITY);
```

```
        receiver::spawn(r).unwrap();
        sender1::spawn(s.clone()).unwrap();

        (Shared {}, Local { sender: s.clone() })
    }

    #[task]
      async fn receiver(_c: receiver::Context, mut receiver:
Receiver<'static, u32, CAPACITY>) {
        while let Ok(val) = receiver.recv().await {
            hprintln!("Receiver got: {}", val);
        }
    }

    #[task]
       async fn sender1(_c: sender1::Context, mut sender:
Sender<'static, u32, CAPACITY>) {
        hprintln!("Sender 1 sending: 1");
        sender.send(1).await.unwrap();
                hprintln!("Sender 1 try sending: 2 {:?}",
sender.try_send(2));
             debug::exit(debug::EXIT_SUCCESS); // Exit QEMU
simulator
    }

      // This interrupt is never triggered, but is used to
demonstrate that
      // one can (try to) send data into a channel from a
hardware task.
    #[task(binds = GPIOA, local = [sender])]
    fn hw_task(cx: hw_task::Context) {
        cx.local.sender.try_send(3).ok();
    }
}
```

```
$ cargo xtask qemu --verbose --example async-channel-try --features test-critical-section

Sender 1 sending: 1
Sender 1 try sending: 2 Err(Full(2))
```

Delay and Timeout using Monotonics

A convenient way to express miniminal timing requirements is by delaying progression.

This can be achieved by instantiating a monotonic timer (for implementations, see `rtic-monotonics`):

```
...
    #[init]
    fn init(cx: init::Context) -> (Shared, Local) {
        hprintln!("init");

        Mono::start(cx.core.SYST, 12_000_000);
        ...
```

A *software* task can `await` the delay to expire:

```
#[task]
async fn foo(_cx: foo::Context) {
    ...
    Mono::delay(100.millis()).await;
    ...
}
```

▶ A complete example

Interested in contributing new implementations of `Monotonic`, or more information about the inner workings of monotonics? Check out the Implementing a `Monotonic` chapter!

Timeout

Rust `Future`s (underlying Rust `async`/`await`) are composable. This makes it possible to `select` in between `Futures` that have completed.

A common use case is transactions with an associated timeout. In the examples shown below, we introduce a fake HAL device that performs some imagined transaction when you call `hal_get(n).await`. We have modelled the time it takes based on the input parameter (`n`) as `350ms + n * 100ms`.

Using the `select_biased` macro from the `futures` crate it may look like this:

```
            // Call hal with short relative timeout using `select_biased`
        select_biased! {
             v = hal_get(1).fuse() => hprintln!("hal returned {}", v),
            _ = Mono::delay(200.millis()).fuse() =>  hprintln!("timeout", ), // this will finish first
        }

             // Call hal with long relative timeout using `select_biased`
        select_biased! {
             v = hal_get(1).fuse() => hprintln!("hal returned {}", v), // hal finish first
                    _ = Mono::delay(1000.millis()).fuse() => hprintln!("timeout", ),
        }
```

Assuming the `hal_get` will take 450ms to finish, a short timeout of 200ms will expire before `hal_get` can complete.

Extending the timeout to 1000ms would cause `hal_get` will to complete first.

Using `select_biased` any number of futures can be combined, so its very powerful. However, as the timeout pattern is frequently used, more ergonomic support is baked into RTIC, provided by the `rtic-monotonics` and `rtic-time` crates. Here's another example, using `Mono::delay_until` and `Mono::timeout_after`:

```
        // get the current time instance
        let mut instant = Mono::now();

        // do this 3 times
        for n in 0..3 {
            // absolute point in time without drift
            instant += 1000.millis();
            Mono::delay_until(instant).await;

            // absolute point in time for timeout
            let timeout = instant + 500.millis();
                  hprintln!("now is {:?}, timeout at {:?}",
Mono::now(), timeout);

             match Mono::timeout_at(timeout, hal_get(n)).await
{
                  Ok(v) => hprintln!("hal returned {} at time
{:?}", v, Mono::now()),
                _ => hprintln!("timeout"),
            }
        }
```

In cases where you want exact control over time without drift we can use exact points in time using `Instant`, and spans of time using `Duration`. Operations on the `Instant` and `Duration` types come from the `fugit` crate.

`let mut instant = Mono::now()` sets the starting time of execution.

We want to call `hal_get` every 1000ms relative to this starting time. We accomplish this by incrementing our `instant` by 1000 ms and then using

`Mono::delay_until(instant).await`. Any additional delays incurred as we iterate around this loop are compensated for by delaying until 'previous + 1000' as opposed to 'now + 1000' (which would cause our loop timing to drift).

To show an alternative to the `select!` async timeout example above, we define a future point in time as `timeout`, and call `Mono::timeout_at(timeout, hal_get(n)).await`.

For the first iteration of the loop, with `n == 0`, the `hal_get` will take 350ms (as described above), and finishes before the timeout. For the second iteration, the delay is 450ms, which still finishes before the timeout. For the third iteration, with `n == 2`, `hal_get` will take 550ms to finish, in which case we will run into a timeout.

▶ A complete example

The minimal app

This is the smallest possible RTIC application:

```
//! examples/smallest.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _; // panic handler
use rtic::app;

#[app(device = lm3s6965)]
mod app {
    use cortex_m_semihosting::debug;

    #[shared]
    struct Shared {}

    #[local]
    struct Local {}

    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
              debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator
        (Shared {}, Local {})
    }
}
```

RTIC is designed with resource efficiency in mind. RTIC itself does not rely on any dynamic memory allocation, thus RAM requirement is

dependent only on the application. The flash memory footprint is below 1kB including the interrupt vector table.

For a minimal example you can expect something like:

```
$ cargo size --example smallest --target thumbv7m-none-eabi --
release

Finished release [optimized] target(s) in 0.07s
   text    data     bss     dec     hex filename
    924       0       0     924     39c smallest
```

Tips & tricks

In this section we will explore common tips & tricks related to using RTIC.

Resource de-structure-ing

Destructuring task resources might help readability if a task takes multiple resources. Here are two examples on how to split up the resource struct:

```
//! examples/destructure.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [UART0])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};

    #[shared]
    struct Shared {
        a: u32,
        b: u32,
        c: u32,
    }

    #[local]
    struct Local {}

    #[init]
    fn init(_: init::Context) -> (Shared, Local) {
        foo::spawn().unwrap();
        bar::spawn().unwrap();
```

```
        (Shared { a: 0, b: 1, c: 2 }, Local {})
    }

    #[idle]
    fn idle(_: idle::Context) -> ! {
            debug::exit(debug::EXIT_SUCCESS);  //  Exit  QEMU
simulator
        loop {}
    }

    // Direct destructure
    #[task(shared = [&a, &b, &c], priority = 1)]
    async fn foo(cx: foo::Context) {
        let a = cx.shared.a;
        let b = cx.shared.b;
        let c = cx.shared.c;

        hprintln!("foo: a = {}, b = {}, c = {}", a, b, c);
    }

    // De-structure-ing syntax
    #[task(shared = [&a, &b, &c], priority = 1)]
    async fn bar(cx: bar::Context) {
        let bar::SharedResources { a, b, c, .. } = cx.shared;

        hprintln!("bar: a = {}, b = {}, c = {}", a, b, c);
    }
}
$ cargo xtask qemu --verbose --example destructure

bar: a = 0, b = 1, c = 2
foo: a = 0, b = 1, c = 2
```

Using indirection for faster message passing

Message passing always involves copying the payload from the sender into a static variable and then from the static variable into the receiver. Thus sending a large buffer, like a `[u8; 128]`, as a message involves two expensive `memcpy`s.

Indirection can minimize message passing overhead: instead of sending the buffer by value, one can send an owning pointer into the buffer.

One can use a global memory allocator to achieve indirection (`alloc::Box`, `alloc::Rc`, etc.), which requires using the nightly channel as of Rust v1.37.0, or one can use a statically allocated memory pool like `heapless::Pool`.

As this example of approach goes completely outside of RTIC resource model with shared and local the program would rely on the correctness of the memory allocator, in this case `heapless::pool`.

Here's an example where `heapless::Pool` is used to "box" buffers of 128 bytes.

```
//! examples/pool.rs

#![no_main]
#![no_std]
#![deny(warnings)]

use panic_semihosting as _;
use rtic::app;

// thumbv6-none-eabi does not support pool
// This might be better worked around in the build system,
// but for proof of concept, let's try having one example
// being different for different backends
```

```
//
https://docs.rs/heapless/0.8.0/heapless/pool/index.html#target
-support
cfg_if::cfg_if! {
    if #[cfg(feature = "thumbv6-backend")] {
        // Copy of the smallest.rs example
        #[app(device = lm3s6965)]
        mod app {
            use cortex_m_semihosting::debug;

            #[shared]
            struct Shared {}

            #[local]
            struct Local {}

            #[init]
            fn init(_: init::Context) -> (Shared, Local) {
                debug::exit(debug::EXIT_SUCCESS); // Exit QEMU
simulator
                (Shared {}, Local {})
            }
        }
    } else {
        // Run actual pool example
        use heapless::{
            box_pool,
            pool::boxed::{Box, BoxBlock},
        };

        // Declare a pool containing 8-byte memory blocks
        box_pool!(P: u8);

        const POOL_CAPACITY: usize = 512;

        #[app(device = lm3s6965, dispatchers = [SSI0, QEI0])]
```

```
        mod app {
            use crate::{Box, BoxBlock, POOL_CAPACITY};
            use cortex_m_semihosting::debug;
            use lm3s6965::Interrupt;

            // Import the memory pool into scope
            use crate::P;

            #[shared]
            struct Shared {}

            #[local]
            struct Local {}

            const BLOCK: BoxBlock<u8> = BoxBlock::new();

                        #[init(local  =  [memory:  [BoxBlock<u8>;
POOL_CAPACITY] = [BLOCK; POOL_CAPACITY]])]
            fn init(cx: init::Context) -> (Shared, Local) {
                for block in cx.local.memory {
                    // Give the 'static memory to the pool
                    P.manage(block);
                }

                rtic::pend(Interrupt::I2C0);

                (Shared {}, Local {})
            }

            #[task(binds = I2C0, priority = 2)]
            fn i2c0(_: i2c0::Context) {
                // Claim 128 u8 blocks
                let x = P.alloc(128).unwrap();

                // .. send it to the `foo` task
                foo::spawn(x).ok().unwrap();
```

```
                    // send another 128 u8 blocks to the task `bar`

bar::spawn(P.alloc(128).unwrap()).ok().unwrap();
            }

            #[task]
            async fn foo(_: foo::Context, _x: Box<P>) {
                // explicitly return the block to the pool
                drop(_x);

                debug::exit(debug::EXIT_SUCCESS); // Exit QEMU simulator
            }

            #[task(priority = 2)]
            async fn bar(_: bar::Context, _x: Box<P>) {
                 // this is done automatically so we can omit the call to `drop`
                // drop(_x);
            }
        }
    }
}
$ cargo xtask qemu --verbose --example pool
```

'static super-powers

In `#[init]`, `#[idle]` and divergent software tasks `local` resources have `'static` lifetime.

Useful when pre-allocating and/or splitting resources between tasks, drivers or some other object. This comes in handy when drivers, such as USB drivers, need to allocate memory and when using splittable data structures such as `heapless::spsc::Queue`.

In the following example two different tasks share a `heapless::spsc::Queue` for lock-free access to the shared queue.

```
//! examples/static-resources-in-init.rs

#![no_main]
#![no_std]
#![deny(warnings)]
#![deny(unsafe_code)]
#![deny(missing_docs)]

use panic_semihosting as _;

#[rtic::app(device = lm3s6965, dispatchers = [UART0])]
mod app {
    use cortex_m_semihosting::{debug, hprintln};
    use heapless::spsc::{Consumer, Producer, Queue};

    #[shared]
    struct Shared {}

    #[local]
    struct Local {
        p: Producer<'static, u32, 5>,
        c: Consumer<'static, u32, 5>,
    }
```

```
    #[init(local = [q: Queue<u32, 5> = Queue::new()])]
    fn init(cx: init::Context) -> (Shared, Local) {
          // q has 'static life-time so after the split and
return of `init`
        // it will continue to exist and be allocated
        let (p, c) = cx.local.q.split();

        foo::spawn().unwrap();

        (Shared {}, Local { p, c })
    }

    #[idle(local = [c])]
    fn idle(c: idle::Context) -> ! {
        loop {
            // Lock-free access to the same underlying queue!
            if let Some(data) = c.local.c.dequeue() {
                hprintln!("received message: {}", data);

                // Run foo until data
                if data == 3 {
                     debug::exit(debug::EXIT_SUCCESS); // Exit
QEMU simulator
                } else {
                    foo::spawn().unwrap();
                }
            }
        }
    }

    #[task(local = [p, state: u32 = 0], priority = 1)]
    async fn foo(c: foo::Context) {
        *c.local.state += 1;

        // Lock-free access to the same underlying queue!
```

```
        c.local.p.enqueue(*c.local.state).unwrap();
    }
}
```

Running this program produces the expected output.

```
$ cargo xtask qemu --verbose --example static-resources-in-init
received message: 1
received message: 2
received message: 3
```

Inspecting generated code

`#[rtic::app]` is a procedural macro that produces support code. If for some reason you need to inspect the code generated by this macro you have two options:

- You can inspect the file `rtic-expansion.rs` inside the `target` directory.
- Use the `cargo-expand` sub-command

Using generated `rtic-expansion.rs`

Locating this file depends on how building is performed.

Using e.g. `cargo xtask build-example` within the main RTIC repo will place the file based on "platform" used:

```
$ cargo xtask example-build --example smallest
$ cargo xtask example-build --example monotonic --platform esp32-c3

$ fd -u rtic-expansion.rs
examples/esp32c3/target/rtic-expansion.rs
examples/lm3s6965/target/rtic-expansion.rs
```

In the regular cargo project case it goes directly in the `target` folder.

This file contains the expansion of the `#[rtic::app]` item (not your whole program!) of the *last built* (via `cargo build` or `cargo check`) RTIC application. The expanded code is not pretty printed by default, so you'll want to run `rustfmt` on it before you read it.

```
$ cargo build --example smallest --target thumbv7m-none-eabi
$ rustfmt target/rtic-expansion.rs
$ tail target/rtic-expansion.rs
#[doc = r" Implementation details"]
mod app {
    #[doc = r" Always include the device crate which contains the vector table"]
    use lm3s6965 as _;
    #[no_mangle]
    unsafe extern "C" fn main() -> ! {
        rtic::export::interrupt::disable();
                let mut core: rtic::export::Peripherals = core::mem::transmute(());
        core.SCB.scr.modify(|r| r | 1 << 1);
        rtic::export::interrupt::enable();
        loop {
```

```
            rtic::export::wfi()
        }
    }
}
```

Using `cargo-expand` tool

If not available, install:

```
$ cargo install cargo-expand
```

This sub-command will expand *all* the macros, including the `#[rtic::app]` attribute, and modules in your crate and print the output to the console.

```
# produces the same output as before
cargo expand --example smallest | tail
```

The magic behind Monotonics

Internally, all monotonics use a Timer Queue, which is a priority queue with entries describing the time at which their respective `Future`s should complete.

Implementing a `Monotonic` timer for scheduling

The `rtic-time` framework is flexible because it can use any timer which has compare-match and optionally supporting overflow interrupts for scheduling. The single requirement to make a timer usable with RTIC is implementing the `rtic-time::Monotonic` trait.

For RTIC 2.0, we assume that the user has a time library, e.g. `fugit`, as the basis for all time-based operations when implementing `Monotonic`. These libraries make it much easier to correctly implement the `Monotonic` trait, allowing the use of almost any timer in the system for scheduling.

The trait documents the requirements for each method. There are reference implementations available in `rtic-monotonics` that can be used for inspriation.

- `Systick based`, runs at a fixed interrupt (tick) rate - with some overhead but simple and provides support for large time spans
- `RP2040 Timer`, a "proper" implementation with support for waiting for long periods without interrupts. Clearly demonstrates how to use the `TimerQueue` to handle scheduling.
- `nRF52 timers` implements monotonic & Timer Queue for the RTC and normal timers in nRF52's

Contributing

Contributing new implementations of `Monotonic` can be done in multiple ways:

- Implement the trait behind a feature flag in `rtic-monotonics`, and create a PR for them to be included in the main RTIC repository. This way, the implementations of are in-tree, RTIC can guarantee their correctness, and can update them in the case of a new release.
- Implement the changes in an external repository. Doing so will not have them included in `rtic-monotonics`, but may make it easier to do so in the future.

The timer queue

The timer queue is implemented as a list based priority queue, where list-nodes are statically allocated as part of the `Future` created when `await`-ing a Future created when waiting for the monotonic. Thus, the timer queue is infallible at run-time (its size and allocation are determined at compile time).

Similarly the channels implementation, the timer-queue implementation relies on a global *Critical Section* (CS) for race protection. For the examples a CS implementation is provided by adding `--features test-critical-section` to the build options.

RTIC vs. the world

RTIC aims to provide the lowest level of abstraction needed for developing robust and reliable embedded software.

It provides a minimal set of required mechanisms for safe sharing of mutable resources among interrupts and asynchronously executing tasks. The scheduling primitives leverages on the underlying hardware for unparalleled performance and predictability, in effect RTIC provides in Rust terms a zero-cost abstraction to concurrent real-time programming.

Comparison regarding safety and security

Comparing RTIC to traditional a Real-Time Operating System (RTOS) is hard. Firstly, a traditional RTOS typically comes with no guarantees regarding system safety, even the most hardened kernels like the formally verified seL4 kernel. Their claims to integrity, confidentiality, and availability regards only the kernel itself (under additional assumptions its configuration and environment). They even state:

"An OS kernel, verified or not, does not automatically make a system secure. In fact, any system, no matter how secure, can be used in insecure ways." - seL4 FAQ

Security by design

In the world of information security we commonly find:

- confidentiality, protecting the information from being exposed to an unauthorized party,
- integrity, referring to accuracy and completeness of data, and
- availability, referring to data being accessible to authorized users.

Obviously, a traditional OS can guarantee neither confidentiality nor integrity, as both requires the security critical code to be trusted. Regarding availability, this typically boils down to the usage of system resources. Any OS that allows for dynamic allocation of resources, relies on that the application correctly handles allocations/de-allocations, and cases of allocation failures.

Thus their claim is correct, security is completely out of hands for the OS, the best we can hope for is that it does not add further vulnerabilities.

RTIC on the other hand holds your back. The declarative system wide model gives you a static set of tasks and resources, with precise control over what data is shared and between which parties. Moreover, Rust as a programming language comes with strong properties regarding integrity (compile time aliasing, mutability and lifetime guarantees, together with ensured data validity).

Using RTIC these properties propagate to the system wide model, without interference of other applications running. The RTIC kernel is internally infallible without any need of dynamically allocated data.

RTIC vs. Embassy

Differences

Embassy provides both Hardware Abstraction Layers, and an executor/runtime, while RTIC aims to only provide an execution framework. For example, embassy provides `embassy-stm32` (a HAL), and `embassy-executor` (an executor). On the other hand, RTIC provides the framework in the form of `rtic`, and the user is responsible for providing a PAC and HAL implementation (generally from the `stm32-rs` project).

Additionally, RTIC aims to provide exclusive access to resources at as low a level as possible, ideally guarded by some form of hardware protection. This allows for access to hardware without necessarily requiring locking mechanisms at the software level.

Mixing use of Embassy and RTIC

Since most Embassy and RTIC libraries are runtime agnostic, many details from one project can be used in the other. For example, using `rtic-monotonics` in an `embassy-executor` powered project works, and using `embassy-sync` (though `rtic-sync` is recommended) in an RTIC project works.

Awesome RTIC examples

See the `rtic-rs/rtic/examples` repository for complete examples.

Pull-requests are welcome!

Migrating from v1.0.x to v2.0.0

Migrating a project from RTIC `v1.0.x` to `v2.0.0` involves the following steps:

1. `v2.1.0` works on Rust Stable from 1.75 (**recommended**), while older versions require a `nightly` compiler via the use of `#![type_alias_impl_trait]`.
2. Migrating from the monotonics included in `v1.0.x` to `rtic-time` and `rtic-monotonics`, replacing `spawn_after`, `spawn_at`.
3. Software tasks are now required to be `async`, and using them correctly.
4. Understanding and using data types provided by `rtic-sync`.

For a detailed description of the changes, refer to the subchapters.

If you wish to see a code example of changes required, you can check out the full example migration page.

TL;DR (Too Long; Didn't Read)

1. Instead of `spawn_after` and `spawn_at`, you now use the `async` functions `delay`, `delay_until` (and related) with impls provided by `rtic-monotonics`.
2. Software tasks *must* be `async fn`s now. Not returning from a task is allowed so long as there is an `await` in the task. You can still `lock` shared resources.
3. Use `rtic_sync::arbiter::Arbiter` to `await` access to a shared resource, and `rtic_sync::channel::Channel` to communicate between tasks instead of `spawn`-ing new ones.

Migrating to `rtic-monotonics`

In previous versions of `rtic`, monotonics were an integral, tightly coupled part of the `#[rtic::app]`. In this new version, `rtic-monotonics` provides them in a more decoupled way.

The `#[monotonic]` attribute is no longer used. Instead, you use a `create_X_token` from `rtic-monotonics`. An invocation of this macro returns an interrupt registration token, which can be used to construct an instance of your desired monotonic.

`spawn_after` and `spawn_at` are no longer available. Instead, you use the async functions `delay` and `delay_until` provided by ipmlementations of the `rtic_time::Monotonic` trait, available through `rtic-monotonics`.

Check out the code example for an overview of the required changes.

For more information on current monotonic implementations, see the `rtic-monotonics` documentation, and the examples.

Using `async` software tasks.

There have been a few changes to software tasks. They are outlined below.

Software tasks must now be `async`.

All software tasks are now required to be `async`.

Required changes.

All of the tasks in your project that do not bind to an interrupt must now be an `async fn`. For example:

```
#[task(
    local = [ some_resource ],
    shared = [ my_shared_resource ],
    priority = 2
)]
fn my_task(cx: my_task::Context) {
    cx.local.some_resource.do_trick();
                          cx.shared.my_shared_resource.lock(|s|
s.do_shared_thing());
}
```

becomes

```
#[task(
    local = [ some_resource ],
    shared = [ my_shared_resource ],
    priority = 2
)]
async fn my_task(cx: my_task::Context) {
    cx.local.some_resource.do_trick();
                          cx.shared.my_shared_resource.lock(|s|
s.do_shared_thing());
}
```

Software tasks may now run forever

The new `async` software tasks are allowed to run forever, on one precondition: **there must be an `await` within the infinite loop of the task**. An example of such a task:

```
#[task(local = [ my_channel ] )]
async             fn             my_task_that_runs_forever(cx:
my_task_that_runs_forever::Context) {
    loop {
        let value = cx.local.my_channel.recv().await;
        do_something_with_value(value);
    }
}
```

`spawn_after` and `spawn_at` have been removed.

As discussed in the Migrating to `rtic-monotonics` chapter, `spawn_after` and `spawn_at` are no longer available.

Using `rtic-sync`

`rtic-sync` provides primitives that can be used for message passing and resource sharing in async context.

The important structs are:

- The `Arbiter`, which allows you to await access to a shared resource in async contexts without using `lock`.
- `Channel`, which allows you to communicate between tasks (both `async` and non-`async`).

For more information on these structs, see the `rtic-sync` docs

A complete example of migration

Below you can find the code for the implementation of the `stm32f3_blinky` example for v1.0.x and for v2.0.0. Further down, a diff is displayed.

v1.0.X

```
#![deny(unsafe_code)]
#![deny(warnings)]
#![no_main]
#![no_std]

use panic_rtt_target as _;
use rtic::app;
use rtt_target::{rprintln, rtt_init_print};
use stm32f3xx_hal::gpio::{Output, PushPull, PA5};
use stm32f3xx_hal::prelude::*;
use systick_monotonic::{fugit::Duration, Systick};

#[app(device = stm32f3xx_hal::pac, peripherals = true, dispatchers = [SPI1])]
mod app {
    use super::*;

    #[shared]
    struct Shared {}

    #[local]
    struct Local {
        led: PA5<Output<PushPull>>,
        state: bool,
    }

    #[monotonic(binds = SysTick, default = true)]
    type MonoTimer = Systick<1000>;

    #[init]
         fn init(cx: init::Context) -> (Shared, Local, init::Monotonics) {
        // Setup clocks
```

```
        let mut flash = cx.device.FLASH.constrain();
        let mut rcc = cx.device.RCC.constrain();

        let mono = Systick::new(cx.core.SYST, 36_000_000);

        rtt_init_print!();
        rprintln!("init");

        let _clocks = rcc
            .cfgr
            .use_hse(8.MHz())
            .sysclk(36.MHz())
            .pclk1(36.MHz())
            .freeze(&mut flash.acr);

        // Setup LED
        let mut gpioa = cx.device.GPIOA.split(&mut rcc.ahb);
        let mut led = gpioa
            .pa5
                .into_push_pull_output(&mut gpioa.moder, &mut
gpioa.otyper);
        led.set_high().unwrap();

        // Schedule the blinking task
                        blink::spawn_after(Duration::<u64,   1,
1000>::from_ticks(1000)).unwrap();

        (
            Shared {},
            Local { led, state: false },
            init::Monotonics(mono),
        )
    }

    #[task(local = [led, state])]
    fn blink(cx: blink::Context) {
```

```
        rprintln!("blink");
        if *cx.local.state {
            cx.local.led.set_high().unwrap();
            *cx.local.state = false;
        } else {
            cx.local.led.set_low().unwrap();
            *cx.local.state = true;
        }
                        blink::spawn_after(Duration::<u64,   1,
1000>::from_ticks(1000)).unwrap();
    }
}
```

V2.0.0

```
#![deny(unsafe_code)]
#![deny(warnings)]
#![no_main]
#![no_std]

use panic_rtt_target as _;
use rtic::app;
use rtic_monotonics::systick::prelude::*;
use rtt_target::{rprintln, rtt_init_print};
use stm32f3xx_hal::gpio::{Output, PushPull, PA5};
use stm32f3xx_hal::prelude::*;

systick_monotonic!(Mono, 1000);

#[app(device = stm32f3xx_hal::pac, peripherals = true,
dispatchers = [SPI1])]
mod app {
    use super::*;

    #[shared]
    struct Shared {}

    #[local]
    struct Local {
        led: PA5<Output<PushPull>>,
        state: bool,
    }

    #[init]
    fn init(cx: init::Context) -> (Shared, Local) {
        // Setup clocks
        let mut flash = cx.device.FLASH.constrain();
        let mut rcc = cx.device.RCC.constrain();
```

```
        // Initialize the systick interrupt & obtain the token
to prove that we did
            Mono::start(cx.core.SYST, 36_000_000); // default
STM32F303 clock-rate is 36MHz

        rtt_init_print!();
        rprintln!("init");

        let _clocks = rcc
            .cfgr
            .use_hse(8.MHz())
            .sysclk(36.MHz())
            .pclk1(36.MHz())
            .freeze(&mut flash.acr);

        // Setup LED
        let mut gpioa = cx.device.GPIOA.split(&mut rcc.ahb);
        let mut led = gpioa
            .pa5
                .into_push_pull_output(&mut gpioa.moder, &mut
gpioa.otyper);
        led.set_high().unwrap();

        // Schedule the blinking task
        blink::spawn().ok();

        (Shared {}, Local { led, state: false })
    }

    #[task(local = [led, state])]
    async fn blink(cx: blink::Context) {
        loop {
            rprintln!("blink");
            if *cx.local.state {
                cx.local.led.set_high().unwrap();
```

```
                *cx.local.state = false;
            } else {
                cx.local.led.set_low().unwrap();
                *cx.local.state = true;
            }
            Mono::delay(1000.millis()).await;
        }
    }
}
```

A diff between the two projects

Note: This diff may not be 100% accurate, but it displays the important changes.

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

 use panic_rtt_target as _;
 use rtic::app;
 use stm32f3xx_hal::gpio::{Output, PushPull, PA5};
 use stm32f3xx_hal::prelude::*;
-use systick_monotonic::{fugit::Duration, Systick};
+use rtic_monotonics::Systick;

  #[app(device = stm32f3xx_hal::pac, peripherals = true,
dispatchers = [SPI1])]
 mod app {
@@ -20,16 +21,14 @@ mod app {
         state: bool,
     }

-    #[monotonic(binds = SysTick, default = true)]
-    type MonoTimer = Systick<1000>;
-
     #[init]
           fn init(cx: init::Context) -> (Shared, Local,
init::Monotonics) {
         // Setup clocks
         let mut flash = cx.device.FLASH.constrain();
         let mut rcc = cx.device.RCC.constrain();

-        let mono = Systick::new(cx.core.SYST, 36_000_000);
+                                    let    mono_token    =
rtic_monotonics::create_systick_token!();
+         let mono = Systick::start(cx.core.SYST, 36_000_000,
```

```
mono_token);

         let _clocks = rcc
             .cfgr
@@ -46,7 +45,7 @@ mod app {
         led.set_high().unwrap();

         // Schedule the blinking task
-                        blink::spawn_after(Duration::<u64,   1,
1000>::from_ticks(1000)).unwrap();
+        blink::spawn().unwrap();

         (
             Shared {},
@@ -56,14 +55,18 @@ mod app {
     }

     #[task(local = [led, state])]
-    fn blink(cx: blink::Context) {
-        rprintln!("blink");
-        if *cx.local.state {
-            cx.local.led.set_high().unwrap();
-            *cx.local.state = false;
-        } else {
-            cx.local.led.set_low().unwrap();
-            *cx.local.state = true;
-                        blink::spawn_after(Duration::<u64,   1,
1000>::from_ticks(1000)).unwrap();
-    }
+    async fn blink(cx: blink::Context) {
+        loop {
+            // A task is now allowed to run forever, provided
that
+            // there is an `await` somewhere in the loop.
+            SysTick::delay(1000.millis()).await;
+            rprintln!("blink");
```

```
+            if *cx.local.state {
+                cx.local.led.set_high().unwrap();
+                *cx.local.state = false;
+            } else {
+                cx.local.led.set_low().unwrap();
+                *cx.local.state = true;
+            }
+        }
+    }
 }
```

Under the hood

This is chapter is currently work in progress, it will re-appear once it is more complete

This section describes the internals of the RTIC framework at a *high level*. Low level details like the parsing and code generation done by the procedural macro (`#[app]`) will not be explained here. The focus will be the analysis of the user specification and the data structures used by the runtime.

We highly suggest that you read the embedonomicon section on concurrency before you dive into this material.

Target Architecture

Cortex-M Devices

While RTIC can currently target all Cortex-m devices there are some key architecture differences that users should be aware of. Namely, the absence of Base Priority Mask Register (`BASEPRI`) which lends itself exceptionally well to the hardware priority ceiling support used in RTIC, in the ARMv6-M and ARMv8-M-base architectures, which forces RTIC to use source masking instead. For each implementation of lock and a detailed commentary of pros and cons, see the implementation of lock in src/export.rs.

These differences influence how critical sections are realized, but functionality should be the same except that ARMv6-M/ARMv8-M-base cannot have tasks with shared resources bound to exception handlers, as these cannot be masked in hardware.

Table 1 below shows a list of Cortex-m processors and which type of critical section they employ.

Table 1: Critical Section Implementation by Processor Architecture

Processor	Architecture	Priority Ceiling	Source Masking
Cortex-M0	ARMv6-M		✓
Cortex-M0+	ARMv6-M		✓
Cortex-M3	ARMv7-M	✓	
Cortex-M4	ARMv7-M	✓	
Cortex-M7	ARMv7-M	✓	
Cortex-M23	ARMv8-M-base		✓
Cortex-M33	ARMv8-M-main	✓	

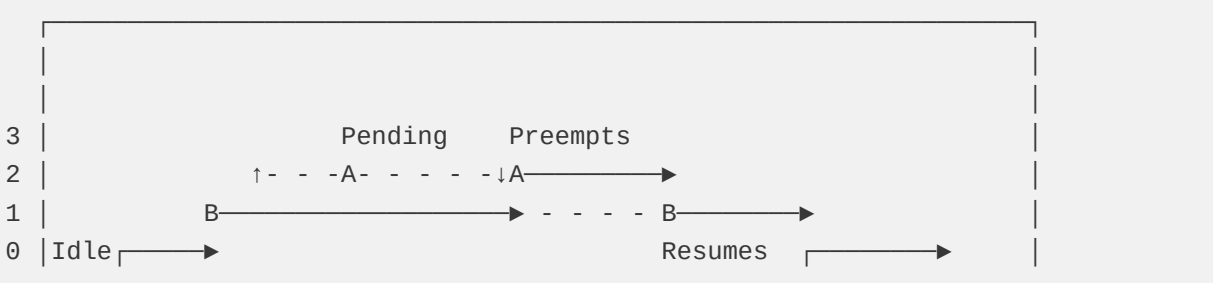
Priority Ceiling

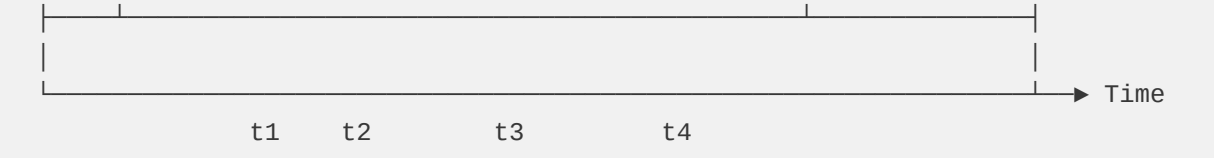
This is covered by the Resources page of this book.

Source Masking

Without a `BASEPRI` register which allows for directly setting a priority ceiling in the Nested Vectored Interrupt Controller (NVIC), RTIC must instead rely on disabling (masking) interrupts. Consider Figure 1 below, showing two tasks A and B where A has higher priority but shares a resource with B.

Figure 1: Shared Resources and Source Masking





At time *t1*, task B locks the shared resource by selectively disabling (using the NVIC) all other tasks which have a priority equal to or less than any task which shares resources with B. In effect this creates a virtual priority ceiling, mirroring the `BASEPRI` approach. Task A is one such task that shares resources with task B. At time *t2*, task A is either spawned by task B or becomes pending through an interrupt condition, but does not yet preempt task B even though its priority is greater. This is because the NVIC is preventing it from starting due to task A being disabled. At time *t3*, task B releases the lock by re-enabling the tasks in the NVIC. Because task A was pending and has a higher priority than task B, it immediately preempts task B and is free to use the shared resource without risk of data race conditions. At time *t4*, task A completes and returns the execution context to B.

Since source masking relies on use of the NVIC, core exception sources such as HardFault, SVCall, PendSV, and SysTick cannot share data with other tasks.

RISC-V Devices

All the current RISC-V backends work in a similar way as Cortex-M devices with priority ceiling. Therefore, the Resources page of this book is a good reference. However, some of these backends are not full hardware implementations, but use software to emulate a physical interrupt controller. Therefore, these backends do not implement hardware tasks, and only software tasks are needed. Furthermore, the number of software tasks for these targets is not bounded by the number of available physical interrupt sources.

Table 2 below compares the available RISC-V backends.

Table 2: Critical Section Implementation by Processor Architecture

Backend	Compatible targets	Backend-specific configuration	Hardware Tasks	Software Tasks	Number of tasks bounded by HW
`riscv-esp32c3-backend`	ESP32-C3 only		✓	✓	✓
`riscv-mecall-backend`	Any RISC-V device			✓	
`riscv-clint-backend`	Devices with CLINT peripheral	✓		✓	

`riscv-mecall-backend`

It is not necessary to provide a list of dispatchers in the `#[app]` attribute, as RTIC will generate them at compile time. Priority levels can go from 0 (for the `idle` task) to 255.

`riscv-clint-backend`

It is not necessary to provide a list of `dispatchers` in the `#[app]` attribute, as RTIC will generate them at compile time. Priority levels can go from 0 (for the `idle` task) to 255.

You **must** include a `backend`-specific configuration in the `#[app]` attribute so RTIC knows the ID number used to identify the HART running your application. For example, for `e310x` chips, you would configure a minimal application as follows:

```
#[rtic::app(device = e310x, backend = H0)]
mod app {
  // your application here
}
```

In this way, RTIC will always refer to HART `H0`.