Connected Embedded Rust - A Survey of system software and network stacks in Rust.

Elena Frank *Freie Universität Berlin*
Berlin Germany
e.frank@fu-berlin.de

***Abstract*—The Rust Programming Language is increasingly used for embedded systems programming as an alternative to C. Its ownership model and type system offer unique properties for compiler-enforced safety, hardware abstraction and concurrency. In this survey we analyze how several prominent embedded Rust projects leverage the programming language in their implementation. We discuss similarities and differences in the designs, and compare how platform specific details are abstracted on different layers. Finally, we discuss their networking stacks to study how complex low-level drivers are implemented in detail.**

***Index Terms*—Rust, Embedded Systems, Networking**

I. Introduction

The Rust programming language has proven to be a popular choice for embedded systems software. By now a large ecosystem has emerged around embedded Rust, with a dedicated working group [1] and over 2500 libraries [2]. Compared to the traditional C the Rust compiler enforces strict memory, thread and type safety, aiming for reliability and performance. Memory safety is realized through the concept of ownership, lifetimes and mutability. It ensures on one hand that references to objects can never be invalid and that allocated memory is freed once an objects owner goes out of scope. On the other hand it enforces special handling of mutable access to an object, which prevent runtime errors when concurrent threads operate on the same object.

Furthermore, the Rust type system itself offers great power. For starters, a lot of fundamental information can be encoded in Rust types, e.g. the information whether a certain periphery has been initialized or not. By using separate types for uninitialized vs initialized periphery an application could for example enforce that certain methods are only possible on one or the other, without needing a manual check. This is only a very basic example, but it shows how leveraging the type system can decrease code size, reduce state variables, and ease maintenance in general.

Rust also supports code de-duplication and the abstraction of implementation details through *Generics* and *Traits*. *Traits* can be compared to what is called an *Interface* in many other programming languages: they define shared behaviour that can be implemented by multiple types. Other structures can then leverage them in their implementation by placing *trait-bounds* on generic inputs or properties. The compiler enforces that only types that implement the required trait are valid for this generic.

In this paper we will discuss how these features are leveraged by several embedded Rust projects, resulting in very different designs. Given that embedded systems are often used in the context of the Internet-of-Things, we further put special focus on the topic of low-level networking stacks and their implementation. Section II introduced the basics of embedded development in Rust, section III provides a survey of relevant frameworks and operating systems, namely RTIC, Embassy, Tock, Hubris, Drone and R3-OS, section IV will discuss the networking stacks in surveyed software, and section V will compare and evaluate the discussed projects.

II. Embedded Rust

Embedded Rust targets platforms where the Rust standard library `libstd` is not available. `libstd` usually enables platform-agnostic access to the primitives that are implemented by the underlying system interface, e.g. an operating system. High-level applications and libraries typically depend on it since it provide a runtime, and basic features like panic handling, stack overflow protection and memory allocation. On baremetal systems however, no prior code has been loaded to the platform and thus no system interface is present. Embedded Rust therefore requires software to be implemented as called `no-std` applications. In this section we will discuss the layers and abstractions that shape the design of low-level

Rust applications, which will be relevant for our survey in section III.

A. Hardware Abstraction

`no_std` applications typically require hardware specific implementations on three layers: the micro-architecture, the peripheral access, and the board layer. Micro-architecture specific logic includes processor calls, memory layout, timer management, exception and interrupt handling, and specifying a program's entry point. An example for a micro-architecture crate is the `cortex_m` crate [3] for Cortex-M processors. The peripheral access layer then provides peripheral access through memory-mapped registers, e.g. for GPIO, SPI, I2C and UART, like the `nrf52832_pac` crate [4] for NRF52 microcontrollers. PAC crates may be implemented manually, however they can also be generated from SVD files. SVD files are published by microcontroller manufacturers and provide a standardized description of each register's functions based on XML [5]. The `svd2rust` [6] crate can then be used to generate the PAC crate code from the files.

In practice, there is typically not just a single processor or chip of its kind, but instead vendors produce a family of them, that share most specification but also differ in some. This often means that on both layers there are two or more crates for each platform: one with the features shared among all devices in the platform's family, and one with individual features.

To provide a common interface for different architectures and the peripheral access, the Rust community is putting quite a lot of effort towards Hardware-Abstraction-Layer (HAL) crates like `embedded_hal`. The goal of `embedded_hal` is to facilitate the writing of platform-agnostic drivers that higher layers can then interface with to "do I/O things". A very basic example that leverage the `embedded_hal` trait for writing to a serial connection is shown in Figure 1. The provided traits hide device-specific details and are generic within and across devices. Furthermore, they abstract the asynchronous model by leveraging the `nb` crate, a minimal non-blocking I/O layer.

Finally, board crates provide functionality for one concrete board and allow the user to directly write firmware for said board without having to deal with the underlying layers. While the previously discussed layers aim to abstract hardware specific details, the purpose of a board crate is now to tailor said layers for a concrete board. In some sense, the purpose of PACs and HALs is to provide a collection of building blocks, and the purpose of a board crate is now to assemble and configure the relevant blocks for one board.

```rust
fn write_all<S>(
  serial: &mut S,
  buffer: &[u8]
) -> Result<(), S::Error>
where
    S: embedded_hal::serial::Write<u8>
{
    for &byte in buffer {
        block!(serial.write(byte))?;
    }

    Ok(())
}
```

Fig. 1. Writing to an abstract serial device [7]. The function is trait-bounded to types that implement the embedded HAL *Write* trait for serial devices (6), so that we can then call the *write* function (9) without knowing the concrete type.

B. Asynchronous Programming and Concurrency

Any software that does some kind of I/O is faced with the challenge of asychrony, which is especially relevant for systems software that directly operates on hardware. Rust's primitive for asynchronous operations are so-called *futures*. As the name describes it, a future is a function that will return an output at some point in the *future*. There are no guarantees about **when** the output will be available, and a future might as well never return.

In practice, a future is implemented as a state machine through the `Future` trait, shown in Figure 2. `Future` only has a single function `poll`, that returns the `Poll` enum. Said enum has exactly to variants: `Poll::Ready(T)` with the final return value, and `Poll::Pending` in case that the return value is not available yet.

```rust
pub trait Future {
    type Output;

    // Probe for the final value.
    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Self::Output>;
}
```

Fig. 2. Rust's future trait for asynchronous code

It is the responsibility of the caller to continuously call the `poll` function to check if the return value is available. In the rest of the paper, we will use the term *polling* for this probing. Obviously, polling a future only makes sense if some progress has been made. Rust therefore couples futures with a waking mechanism through a `Waker` that is provided to `poll` as part of the `&mut Context`. Per contract, it is expected that `poll` is called again by the user each time the waker is triggered. On the other side, it is expected that inside the future implementation the waker is triggered each time some relevant progress has happened, e.g. after a related interrupt. This mechanism allows for each future to be driven to completion, while enabling the user to continue with other work until it is ready. It is common practice in applications with many async operations to never directly block, but instead poll the future state machines sequentially or hierarchically.

For blocking on a future, Rust provides the `async/await` primitives. Calling `.await` on a future waits for the future to be ready before continuing the execution. In return, a function that `awaits` a future itself becomes a future and must be annotated with the `async` keyword. Through this convention asynchronous code is propagated through all layers up to the user. The final handling commonly involves either blocking the executing thread, or spawning a new thread that owns the future and drives it to completion.

In the next section we will discuss how the different stack support asynchronous operations and concurrent programs in practice. Although not all of them use futures, we will see that the usage of state machines, together with some (often hardware enforced) waking mechanism is a common design.

III. Survey of Embedded Systems Software

We've discussed in the last section the fundamentals for writing low-level application in Rust. However, once we step from a small embedded application to complex systems software, e.g. to writing an Operating System, a lot more design and architectural questions are involved. The type system of the Rust programming language has great power in how it can be used to ensure safety, prevent runtime-errors, and define abstraction. In this section we will survey various embedded Rust projects that leverage the Rust type system in very different ways. For each of the systems we will individually discuss the general design and architecture, memory management, concurrency model, security considerations and (hardware) abstractions.

A. Concurrency Frameworks

Concurrency frameworks are embedded platforms for applications with concurrent units of execution, without the need for a full operating system. They typically provide an executor, and APIs for spawning concurrent tasks and accessing shared structures. The distinction between concurrency frameworks and real-time operating systems is rather mushy and subjective, but one can say that the main difference is that concurrency frameworks don't have a software kernel [8]. Thus, they usually don't do memory management, they run tasks on a single stack, and, if any, only offer very limited IPC.

The predominant concurrency frameworks in the Rust ecosystem are Embassy [9] and RTIC [10], which we will discuss in the following sections.

1) Embassy: The Embassy framework heavily makes use of the asynchronous programming primitives discussed in section II-B, i.e. async-await and futures. It provides two main features: the embassy executor that takes care of task management, and a hardware abstraction layer. Furthermore, it implements a networking stack that we will discuss in more detail in section IV-B.

The Embassy executor implements a runtime for concurrent tasks, a task here being a user-provided function. Tasks are statically allocated, which means that all tasks have to be defined at compile-time. This obsoletes the need for allocation through a heap with a dedicated allocator. Scheduling in embassy is cooperative. A task gets CPU time until it yields due to awaiting a asynchronous operation. The user is responsible for ensuring that extensive computations in one task doesn't block all others. A very simple embassy application is shown in Listing 3.

```
#[embassy::task]
async fn blink(pin: AnyPin) {
  let mut led = Output::new(pin, Level::Low,
    OutputDrive::Standard);
  loop {
    led.set_high();
    Timer::after(Duration::from_millis(150)).await;
    led.set_low();
    Timer::after(Duration::from_millis(150)).await;
  }
}
.
#[embassy::main]
async fn main(spawner: Spawner, p: Peripherals) {
  spawner.spawn(blink(p.P0_13.degrade())).unwrap()
  loop { }
}
```

Fig. 3. Simple embassy application [9]. A single task is implemented as an async function that manages an abstract pin. The main method spawns the task and provides it with the actual periphery.

Internally, a raw executor structure is used, that is implemented as a state-machine similar to a normal Rust future. The raw executor's poll method sequentially polls each task in the runqueue until they are yielding. It is efficient in the regard that the runqueue only contains tasks that have been woken, i.e. for which an interrupt was generated since they have been polled the last time. Waking happens through a arch-specific `Pender` that is responsible for notifying the executor if there is pending work, e.g. because an interrupt was triggered. On the user-facing layer the exposed `Executor` is an arch-specific wrapper of the state machine together with logic for interrupt handling and running the never-returning raw executor. Given that tasks are implemented as state machines, they all run in a single thread and share one stack. In practice, multiple embassy executors can be used if they run in separate processor threads. Supported architectures by the executor are the ARM Cortex-M series, 32-bit RISC-V and X-Tensa. Additionally, implementations for the Rust standard library and wasm are provided.

The interface to periphery is abstracted in the embassy Hardware Abstraction Layer (HAL). It wraps the `embedded-hal-async` crate, which is an asynchronous version of the `embedded-hal` crate that we discussed in section II-A. The embassy HAL provides blocking and non-blocking APIs for shared access to periphery. This is done by implementing the periphery traits specified in `embedded-hal-async` for their own generic wrapper types. The wrappers for I2C, SPI and Flash internally make use of the `embassy_sync::Mutex` to ensure exclusive access to the raw shared object. Additionally, the crate defines adapter traits for wrapping drivers that implement `embedded-hal` traits, but should be used in an async manner.

2) Real-Time Interrupt-driven Concurrency (RTIC) : Like embassy, RTIC [10] is an concurrency framework without a kernel. However, RTIC does not provide any HALs and currently only supports Cortex-M devices.

RTIC implements concurrency and resource management based on the *Stack Resource Policy (SRP)* [11], which itself extends the *Priority inheritance Protocols* [12]. Each task has a static priority and runs to completion unless it is preempted by a higher-priority task. The details of the protocols are out of scope for this paper, but the relevant effect is that it enables all tasks to run on a single stack with wait-free access to shared resources. The framework differentiates between hardware and software tasks. Hardware tasks run as interrupt handlers to completion and return. Software tasks are not explicitly bound to a specific interrupt vector, but instead to an async executor, the so-called *Dispatcher*. Software task may be started once and run forever and can only be preempted by higher-priority tasks. Thus, like in embassy, it is required that software tasks with non-returning loop cooperatively yield (by awaiting a future, e.g. a delay timer). Communication between tasks is implemented through a small global critical section that copies the payload from the sender to a static variable, and the static variable to the receiver.

Applications can define local and shared system-wide resources through macros. Local resources, e.g. a driver or a large object, are only accessible to a specific task. Shared resources on the other hand are mutex-protected and shared between multiple tasks. They can be accessed in a critical section that is implemented as a closure, and subject to bounded priority inversion. While the critical section is executed, a task can only be preempted by another higher-priority task if that task does not contend for the acquired shared object. A minimal RTIC application is shown in Figure 4.

```
#[app(device = stm32f3xx_hal::pac,
      peripherals = true,
      dispatchers = [SPI1]
)]
mod app {
  #[shared]
  struct Shared {}

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

  #[init]
  fn init(_: init::Context) -> (Shared, Local) {
    // Setup rcc and clock
    ...
    // Setup LED
    ...

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

    (Shared {}, Local { led })
  }
  #[task(local = [led])]
  async fn blink(cx: blink::Context) {
    // blink led
    ...
  }

}
```

Fig. 4. A minimal RTIC application with a single task for blinking an LED. [10] After initialization the LEDs peripheral object is stored in a global variable to which only that one task has access to.

B. Embedded Operating Systems

We are now going to discuss a number of embedded operating systems that are written entirely in Rust. It has to be noted that the covered Rust OSes are by far not the only interesting ones. That being said, we decided to only cover the most relevant ones in order to allow for a more thorough analysis and still keep an acceptable scope. We considered the following aspects in our selection:

1) Targeted platform: Embedded platforms may only have limited resources available, which requires specific considerations from systems software e.g. to reduce overhead and efficiently use memory. Operating systems that are not designed for such

restriction, and for example only/ or primarily target x86_64, are therefore not discussed in this survey. This concerns many well known Rust OSes, like Theseus [13], Redox [14], RustyHermit [15], Kerla [16] or rCore [17].
2) Entirely written in Rust: Operating systems like Google's Cantrip OS [18], that builds on top of the C seL4 kernel, or Rust interfaces to C operating systems, like FreeRTOS.rs [19] are not relevant.
3) Maturity and active maintenance: There are a ton of hobby operating systems that are either not actively maintained anymore, or haven't (yet) reached the level of maturity where a thorough analysis makes sense. Although the vast majority of them targets x86_64, there are also projects for embedded, like the Bern RTOS [20] or K5 microkernel [21], that might be interesting to look at in the future.

Based on this criteria, we found the most interesting embedded Rust OSes to be Tock [22], Hubris [23], Drone [24] and R3-OS [25].

1) Tock: Tock is probably the most well-know embedded Rust operating systems. It was first published to academia in 2017 [26], and since then has been further developed in an ongoing effort with more than 150 contributors [22]. The main objective of Tock is to enable secure multiprogramming on embedded system, i.e. to concurrently run multiple, untrusted applications. It promises isolation of software faults, memory protection, and dynamic, efficient memory management [26].

The Tock architecture differentiates between two classes of code: Capsules and Processes. Capsules are the units of composition in the Tock kernel, with a strict isolation enforced through the Rust type system. Most capsules are "untrusted", which means that they are not allowed subvert this type system sandbox by using unsafe code. Their interface explicitly defines the resources that they are being granted access to. Examples for untrusted capsules are drivers, logic for multiplexing hardware, or system-call capsules. Given that no unsafe code is allowed, untrusted capsules can not directly interact with hardware. Instead, a small number of trusted capsules are defined, where unsafe code is allowed. Trusted capsules provide low-level abstractions of MCU and Board peripherals by casting memory-mapped registers to type-safe structures, and implement core kernel functionalities like process scheduling. Each capsule owns a statically allocated event queue for receiving events from processes. Capsules themselves can not generate events and instead interact with each other directly through function calls or shared state variables. The kernel schedules capsules cooperatively and only polls them if there is work to be done.

Processes on the other hand are concurrently executed programs that each own a logical region of memory with a separate stack, heap and static variables. Separating the process stacks allows the kernel to schedule the processes in round-robin with preemption, which prevents that one process is consuming the whole CPU time. Processes are hardware-separated through isolated, MPU protected memory regions. Thus they don't have to be sandboxed through the Rust type system and can be written in any programming language. The architecture of Tock is visualized in Figure 5.

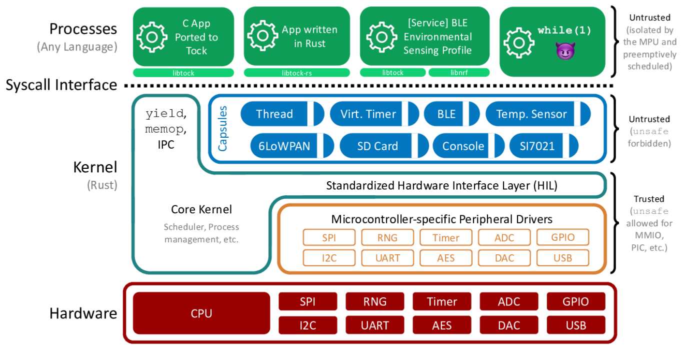

Fig. 5. Tock architecture [22]

Processes interact with the kernel through a non-blocking system call interface that only consist of 5 calls:

- `Command` invokes an operation on a capsule.
- `Allow` provides memory to a capsule to pass more complex data or provide a buffer for receiving I/O data.
- `Subscribe` registers a call-back function that a capsule can invoke. It is not serviced until the processes has yielded (see command below).
- `Memop` invokes the core kernel to increase the processes heap size.
- `Yield` block the process until an upcall completed.

For the interaction with capsules, a process is responsible for providing the necessary memory space that the capsule needs to service the command. This is done through so-called *Grants*. A Grant is kernel heap memory that is located within the processes memory region, but kernel controlled. The process itself is not able to access it. Memory access to a grant is offered to a capsule so it can use it for dynamic allocation when responding to a request. When a process dies, the grant is freed alongside with its other memory. Grants share their allocation space with the process heap, i.e. if the process heap is increase through the `Memop` syscall, the (free) grant space is decreased.

Grants also facilitate the inter-process communication (IPC). IPC is implemented as server-client communication, where a server announces a service to the kernel and a client may then discover the server through the kernel. To interact with the server, the client can to provide a buffer in the form of a grant, that is shared with the server. The server may then use or write to the buffer depending on the service protocol. Per process only one server can be defined.

Everything in Tock is build in-house, which includes all hardware-related logic. It provides its own Hardware Interface Layer (HIL) definitions in the kernel, which roughly correspond to the peripheral access layer discussed in section II-A. The implementation is provided by chip-specific crates, that build upon architecture crates and use svd2rust for setting up register mappings. The platform-specific integration of capsules is then done on the board layer. All capsules are platform agnostic, and instead a separate *components* crate is provided with helper modules for initializing each capsule. Board crates use this together with the matching chip crate to provide configurations of kernel, chip and peripherals for one specific hardware board. They are responsible for assigning peripherals to drivers, setting up virtualization layer and defining system call interfaces.

2) Hubris: Hubris' [27] design fundamentally differs from Tock, as it consist of a memory-protected, message-passing kernel and a collection of tasks. Compared to Tock, the design philosophy behind Hubris is to reduce complexity by avoiding unnecessary flexibility and not "trying to be smart". Instead simplicity is favored by limiting the features provided by the kernel and enforcing the user to follow certain assumptions. As part of this design choice, Hubris is intentionally static in respect to tasks and does not support dynamic memory management or the creation of tasks. Instead, tasks are separately compiled programs or drivers with fixed priorities at build time, specified in a `app.toml` configuration file. A task's executable code is immutable, which means that dynamic updating, creating or destroying of tasks is not possible. Tasks run in unprivileged processor mode and only a small kernel is privileged. Thus Hubris can be considered a micro-kernel, albeit the maintainers prefer not to use the term given its ambiguity [28]. Hubris schedules tasks strictly (preemptive) according to their priorities, within a priority class however the scheduling is cooperative. A dedicated supervisor task, specified at index 0 in the configuration file, is responsible for task management and crash recovery.

Inter-process communication in Hubris is synchronous apart from a very limited asynchronous notification API for interrupts and signals. The synchronous messages is implemented as "handoff", consisting of a single message copy in the kernel from sender to receiver. This obsoletes the need for a queue and additionally prevents task from being able to spam messages to other tasks. Messages are limited to 256 bytes. For larger messages Hubris introduces the notion of leases, which are small descriptor that enable the receiver to access part of the sender's memory. On the receiving side, a response needs to be returned for each message. Until the response is received, the sender is blocked, which means that receivers can coordinate senders and e.g. leverage this to enforce mutual exclusion. Obviously this communication mechanism as it is creates a large risk of deadlocks and priority inversion. As a result, Hubris enforces that a task may only send messages to higher priority tasks, resulting somewhat in a server-client hierarchy. Communication is nevertheless still possible in both direction via the previously discussed leases: a task can receive a message from higher-prioritized task by offering a write-only lease to it.

Interestingly, Hubris also provides an IPC API to the kernel. Compared to the additional usual system calls that provide fundamental operations used by each task, like IPC calls and timer management, the kernel's IPC interface facilitates operations where the kernel itself is involved. This is implemented through the concept of a "virtual kernel task" that each task can send messages to. The messages are then intercepted and handled by the kernel itself. In practice, the operations revolve around the supervisor's tasks management, e.g. reading a task's status, or restarting or stopping a task. They can currently be used by any task, which means that one misbehaving task right now can arbitrarily restart or stop any other tasks. It is planned to eventually restrict the kernel's IPC calls to only "privileged tasks" like the supervisor task [29].

The only asynchronous messaging mechanisms in Hubris are *Notifications*, which are written into a 32b notification set that each task has. The task is not interrupted, but instead has to manually check its notification mask. Notifications are used between tasks to notify without blocking, but also by the kernel to route hardware interrupts. Tasks have exclusive control over the interrupts, as specified during compile time through macros. One usecase for this is the implementation of drivers, which, given the microkernel-like nature of Hubris, are consequently implemented as tasks. More specifically, they are implemented as *Servers*, which is Hubris' name for tasks that implement some sort of IPC API. A driver typically consists of two crates: the inner driver crate that serves as interface for dealing with a device, by directly accessing the hardware or doing IPC calls to other servers, and the driver server that wraps the driver crate to provide an IPC interface. Servers typically consists of a looping main function that receives messages from other tasks through syscalls and responds to them. Figure 6 shows such a driver implementation: the RCC driver for the STM32F3/4.

```
#[export_name = "main"]
fn main() -> ! {
  // Pointer to the RCC register block.
  let rcc = unsafe { &*device::RCC::ptr() };

  let mut buffer = [0u32; 1];
  loop {
    hl::recv_without_notification(
        buffer.as_bytes_mut(),
        |op, msg| -> Result<(), ResponseCode> {
          let (msg, caller) = msg
            .fixed::<u32, ()>()
            .ok_or(ResponseCode::BadArg)?;
          ..
          match op {
            Op::EnableClock => { .. },
            Op::DisableClock => { .. },
            Op::EnterReset => { .. },
            Op::LeaveReset => { .. },
          }

          caller.reply(());
          Ok(())
      },
    );
  }
}
```

Fig. 6. Hubris driver for the STM32F3/4 Reset and Clock Controller (RCC) [23], drv/stm32fx-rcc/src/main.rs

This driver is board-specific, as most of Hubris drivers are. There is not clear Hardware Abstraction Layer in Hubris, but where possible platform-independent driver logic is extracted into separate servers, that are then used by the platform-specific implementations. A `<driver>-api` crates defines the platform independent API, i.e. a trait, for this server so that the client implementation can be platform independent. On the architectural level only ARM-M processors are supported. Hubris does not allow the usage of `embedded-hal` because `embedded-hal` implementations, like cortex-m, assume that they run in the processor's privilege mode and e.g. can disable interrupts, which is not the case in Hubris [28].

3) Drone: Drone [30] is a real-time operating system that, like the previously discussed RTIC framework, implements an interrupt-driven execution model. As a result, its design shares a lot of similarities with RTIC to achieve hard real-time. Like RTIC, Drone implements fully preemptive, strict priority-based scheduling where all task shared a single stack. By placing the stack at the boarder of the RAM, stack-overflows are prevented without the requiring an MMU or MPU. A task must completely relinquish the stack before completing or pausing.

Drone's primitive for asynchronous code are `Fibers` which internally use Rust's experimental `Generators` [31] API. `Generators` are stackless co-routines that rely on the Rust compiler to translate the co-routine to a state-machine. As shown in Listing 7, their execution can be suspended / yielded without loosing state.

```
let mut generator = || {
  let mut state = 0;
  while state < 3 {
      state += 1;
      yield state;
  }
  state
};
```

Fig. 7. Implementing a coroutine using Rust's experimental **Generator** [30]

The goal behind the `Generators` is to ease the implementation of asynchronous code. Drone leverages the feature for its `Fibers`, which implement a future-like API. The main difference is that `Fibers` are provided with an input when they are polled, and their intermediary `Yielded` state may include a value (contrary to the futures' `Poll::Pending`). Drone also provides an API for mapping a fiber into a future or stream.

Around the core `Fiber` trait, Drone defines additional APIs for writing asynchronous code: *Tasks*, *Threads* and *Processes*. Drone supports software and hardware triggered tasks, albeit the term *task* is only used for the former, and the latter are referred to as *threads*. Thus we will use the same terminology in this section. Threads are initialized on hardware interrupts, by utilizing the underlying MCUs nested vectored interrupt controller. Each thread manages a sequence of fibers in a linked list, the so called "fiber chain" that it runs LIFO order. A task on the other hand is any logical unit of work that should run in a separate thread, but is not related to a hardware interrupt. Like in RTIC, tasks can simply use unused hardware interrupts. Threads/ tasks always run to completion, i.e. they can't `yield` in their execution. For long-running jobs Drone introduces a third primitive: *Processes*. Processes are a special kind of fibers that used together with a supervisor, to which they hand back the execution context on each `yield`. Compared to fibers, which are stackless and internally use state machine, processes are stack-based and their implementation is platform specific. On Cortex-M platforms for example SVCall exception are used for yielding, so that the supervisor can then switch to the next processes' stack. Each process' stack is fix-sized and allocated from the heap upon process creation, and MPU protected if one is present.

For inter-process communication, three message-passing mechanism are supported: oneshot channel, ring channels, and a pulse functionality for repeatedly notifying another thread about some event. Their implementation leverages the atomic compare-and-swap operation for synchronization by encoding the channel's state as bits in an integer, that is then atomically updated. Drone generally makes heave use of atomic operations, instead of doing mutual exclusion through critical sections like it is done in RTIC. This is also apparent in its dynamic memory allocation. Unallocated regions of memory are chained together in `Pools` as linked list of blocks using atomic pointers, which allows allocation and deallocation to be lock-free. The size of the heap, its address, and the pools need to be defined by the user application through procedural macros, as its done for most platform-specific configuration. Drone provides macros for specifying thread/ interrupt mapping, the memory layout and peripheral mapping. On what it calls the "vendor-layer" these macros can then be used to specify bindings, alongside with register mappings that are generated from svd files. Furthermore, as we already briefly mentioned, an platform-specific implementation of the threading backend is required on the architecture-layer. Vendor- and architecture-specific implementations reside in their own crates, that a user application needs to depend on alongside with `drone-core`. For example the *Blue Pill Blink Demo for Drone OS* [32] depends on `drone-core`, `drone-cortexm` (the ARM® Cortex®-M platform crate) and `drone-stm32-map` (STM32 peripheral mappings crate).

4) R3-OS: Compared to the previously discussed operating systems, R3 is still very experimental. However, it makes very heavy usage of the Rust type system, macro and traits, to implement multiple layers of abstraction. This results in some very interesting properties, which makes it a relevant project to discuss.

For starters, it completely abstracts the kernel interface away from a kernel implementation. The `r3_core` crate only defines all the mandatory and optional traits a kernel should implement, while the separate `r3_kernel` crates provides the implementation of the "original" r3 kernel. This enables a user application to a) select the most suitable kernel with the appropriate properties for their use-case, and b) to easily port between hardware platforms by simply selecting the matching platform-specific kernel for their architecture. At the point of writing, only a single kernel implementation, the `r3_kernel`, exist, to which we will refer in the remainder of this section. We will call it *the* kernel implementation, but the reader should keep in mind that in the future additional implementations may emerge whose implementation can be entirely different.

The r3 kernel again abstracts its components with multiple traits, that are distinguished into platform-specific and platform-independent traits. The implementation of the traits is done through procedural macros. The rational behind using a macro (instead of trait-bounded properties) is that it allows to define static items, which r3 by design leverages for all of its kernel objects and to store internal state data. A basic example for the configuration of the r3 kernel is shown in Figure 8, which we will now discuss in more detail.

```
// Marker type around which r3 app it build
type System = r3_kernel::System<Traits>;
// Platform-specific implementations
r3_port_std::use_port!(unsafe struct Traits);

// Static objects
#[derive(Debug)]
struct Objects {
  task: StaticTask<System>,
}
const COTTAGE: Objects =
  r3_kernel::build!(Traits, configure => Objects);
// Application-specific configuration
const fn configure(
  cfg: &mut r3_kernel::Cfg<'_, Traits>
) -> Objects {
  cfg.num_task_priority_levels(4);

  let task = StaticTask::define()
    .start(task_body)
    .priority(3)
    .active(true)
    .finish(cfg);

  Objects {
    task,
  }
}

fn task_body() { ... }
```

Fig. 8. Basic configuration of an r3 application with a single task [25].

Platform-specific traits have to be implemented by so-called kernel *Ports*. By convention, a Port is expected to provide a `use_port!` macro that the user can simply call. It is required to implement traits for threading, controlling the system timer, and controlling interrupt lines. Furthermore, it must generate entry points to the kernel, i.e. the main function and interrupt handlers. The latter must then call user-defined interrupt service routines, whose registration we will discuss further below.

Everything else is defined in the platform-independent part, through the `r3_kernel::build!` macro. It defines most of the static kernel objects, for example the tasks queue that the platform-specific scheduler queries to get the next task. The decision if a context switch should happen and what tasks should run next is hereby platform-independent, and instead based on user-defined static and dynamic task priorities. The macro also configures the kernel based on user configuration. The configuration is done through a config function (`configure` in Figure 8) that, due to the usage of macro and traits, is (again) quite complex and abstract, which makes a detailed explanation out of scope for this paper. But the essence is that they enable the user to register start-up hooks, tasks, and the interrupt service routines that are called by the interrupt handlers we discussed above. This also presents a platform for driver implementation through board support packages, where the package simply provides a config function that should recursively be called to register the driver. Figure 9 shows such a config function that registers a USB serial devices based on peripheral types from a PAC crate, and user-defined options.

```
pub trait Options: 'static + Send + Sync {
    fn handle_input(_s: &[u8]) { ... }
    fn product_name() -> &'static str { ... }
    fn should_pause_output() -> bool { ... }
}

pub const fn configure<'a, C, O: Options>(
    b: &mut Cfg<'a, C>,
    rp2040_resets: _,
    rp2040_usbctrl_regs: _
) where
    C: CfgStatic + CfgInterruptLine { ... }
```

Fig. 9. Config function in the board support package for the RP2040 to register a USB serial device. The required periphery inputs are from the RP2040 PAC crate and have to be initialized in a separate startup-hook beforehand.

R3 does not offer primitives for IPC, apart from shared access to the static objects.

IV. Network stacks in Rust systems software

We have previously discussed the challenges of writing embedded Rust applications, and surveyed a number of different architectures and system designs. In this section we are going to deep-dive into one specific feature that is fundamental for most embedded systems: the networking stack. Its implementation presents a union of the individual issues that we've discussed: asynchronous operations, hardware abstraction, state management, and bounded resources.

In contrast to almost all other subjects, there is actually one dominant choice for low-level networking primitives in the Rust ecosystem: the `smoltcp` crate. It is essentially used by all embedded projects if they are not writing the whole stack in-house. Thus, we will provide an introduction to `smoltcp` in section IV-A and discuss its interface and abstractions. We will then discuss how some of the project introduced in the previous section integrate smoltcp, and the look into Tock's networking stack as an example for an in-house implementation without smoltcp.

A. *smoltcp*

smoltcp [33] is a TCP/IP stack that neither requires the Rust `std` library, nor heap-based allocation through `alloc`. It supports raw, ICMP, TCP and UDP sockets. Sockets are implemented following the Sans-I/O design principle [34]. Sans-IO means that a socket or protocol implementation itself doesn't do any I/O, but instead is provided by the user with the bytes and operates on them. Thus the implementation neither directly interacts with any hardware, nor does it do any sort of asynchronous flow control. This design facilitates the platform-agnostic implementation of its functionalities. In case of the sockets, these are buffering, packet construction & and validation, and the state machine implementation (for stateful sockets).

The actual I/O is done on the physical layer, where the interaction with platform-specific network devices is implemented. A network device needs to implement the `Device` trait for specifying device capabilities like MTU and the type of medium, and abstract functions for sending and receiving packets. Routing the packets from and to sockets from the device is managed in the `Interface` state machine. The central function of the `Interface` is (again) a `poll` function that has to be provided with an abstract device and a set of sockets, shown in Figure 10. In the function, queue packets in the given sockets are transmitted, and incoming packets queued in the devices are received. Furthermore, the interface also takes care of the physical addressing, neighbour discovery, and sending of control messages. Supported interfaces are Ethernet, IP, and IEEE 802.15.4 + 6LoWPAN.

```
impl Interface {
  /// Transmit packets queued in the given sockets,
  /// and receive packets queued in the device.
  pub fn poll<D>(
    &mut self,
    timestamp: Instant,
    device: &mut D,
    sockets: &mut SocketSet<'_>,
  ) -> bool
  where
      D: Device + ?Sized,
  { ... }
 ...
}
```

Fig. 10. smoltcp::iface::Interface poll methods.

As we've previously discussed, future-like poll functions typically get provided with a waker that can be triggered if some progress happened. This doesn't necessarily have to be done by the function itself, but instead a waker is often passed on in inner async method calls, e.g. to an I/O device so it can wake it if some I/O happened. In smoltcp however, no waker is passed to `Interface::poll`. Instead it is the responsibility of the user application to sleep for a reasonable amount of time before polling the interface again. The interface provides methods that return an *"advisory wait time"* for this. Concerning asynchronous code in general, smoltcp tries to be un-opinionated [35]; It doesn't have any async code or futures itself, but provides a feature-gated waking mechanism for sockets so they can be used with async code through higher-layer wrappers.

B. Usage and integration of smoltcp

For the discussion of smoltcp's integration in larger applications we decided to only focus on the crates that we introduced in section III because a) their high-level architecture is already known to the reader and b) they generally make broader usage of smoltcp's features and abstraction layers, compared to embedded applications that only need selected features. Given that Drone OS and R3 don't have a networking stack, and Tock provides an in-house implementation that we will discuss in section IV-C, this only concerns Embassy and Hubris. For RTIC, which itself is hardware-agnostic and doesn't provide any drivers, we will briefly discuss the STM32F Ethernet Driver [36]. It implements the smoltcp trait for is hardware and provides an example application for using RTIC together with smoltcp. We will start with this use-case, given that it is the simplest one.

The STM32F ethernet driver implements smoltcp's `Device` trait for its `EthernetDMA` type. In practice, this simply means that it configures the MTU, and implements receiving and transmission of packets by moving the packets into/ from the DMAs RX / TX ring buffers. Said buffers resides in a memory region to which the periphery has direct access so that it can transfer data, i.e. inbound and outbound packets, in the background.

The usage together with RTIC is also rather straightforward, as show in an example TCP echo server [36]. In the `init` function the periphery is set up and configured, sockets created and the dma interrupt enabled. A hardware task is bound to said interrupt. The task has access to a local state that wraps the necessary smoltcp types, i.e. the `Interface`, `SocketSet` and the physical device (the `EthernetDMA`). When the hardware task is executed, it polls the interface (so that potential incoming bytes are transferred from the DMAs RX buffer to the socket state machine) and then checks the sockets for new incoming bytes. If there are any it echos them back, and then polls the interface again so that the packets are transferred again from the socket state machine to the DMAs TX buffer. Obviously this implementation is hardware specific, mostly because the initial peripheral configuration direct operates on the stm32 peripheral access crate. To further abstract the example one could instead operate on the embedded_hal abstraction layer, which goes more into the direction of what embassy does.

Embassy splits the networking stack into a platform-agnostic interface `embassy-net`, and platform-specific drivers that build upon the `embassy-hal`. The `embassy-net` interface is a wrapper around the smoltcp types, device driver and needed memory resources, e.g. the socket list, with a higher-level, more opinionated API [37]. The non-returning `run` function (that should be called in a separate embassy task) continuously polls the smoltcp `Interface` with delays in-between based on the previously discussed advisory delay that the `Interface` specifies.

Hubris integration of smoltcp is very similar. Its networking support generally is still experimental, and only a platform-specific server implementation for the STM32H7 series is currently provided. It implements the `smol::phy::Device` trait for wrappers around the STM32h7 ethernet driver. The ethernet driver implementation itself is quite tailored to the usage within smoltcp. It manages relevant register blocks and provides methods for sending, receiving, and ethernet interrupt handling. It is wrapped together with the smoltcp sockets and interface by the server implementation. The server handles IPC calls and is continuously polled in the task's `main` function.

C. Tock Networking stack

In this section we briefly going to discuss Tock's in-house networking stack as an alternative to the usage of smoltcp. Tocks networking stack is implemented through a networking capsule that provides a `UDPDriver` based on Ipv6 with the 6loWPAN compression & fragmentation for a IEEE 802.15.4 wireless link. For the actually sending and receiving an abstract trait `MuxUdpSender` is defined that the user needs to implement. The integration between the layers is based on traits, which means that while this is currently the only supported stack, new protocol could be added on each layer in the future and used instead. It furthermore provides implementations of ICMPv6 and the *Thread* network protocol, but without driver implementations.

A `ieee802154` capsule additionally implements a dedicated driver for IEEE 802.15.4 on the link layer using the `net` capsule's `ieee802154` module. It implements a `RadioDriver` structure that operates on an abstract, platform-specific `MacDevice`. A client of the driver has to provide RX and TX buffers trough Grants, as well as set callback methods for when a transmission succeeded or a new packet was received.

Board crates can integrated the capsules by implementing the `MacDevice` for their hardware and either only integrate the `ieee802154` capsule, or make use of a provided component that implements the previously mentioned `MuxUdpSender` on top of the `ieee802154` capsule, to integrate the `UdpDriver`.

V. Comparison and Evaluation

Table I shows a general comparison of the discussed projects.

A. General Architecture and Design

1) Asynchronous execution: Async Rust functions are only supported by the concurrency frameworks, i.e. Embassy and Rust. Processes in Tock can yield their execution through a dedicated system call, and in Drone `Fibers` can yield as well. In Hubris tasks can not explicitly yield, but their execution is automatically blocked if they wait for a notification or the response to a sent message. In R3 yielding is not possible and tasks are simply scheduled based on a system timer.

2) Memory layout: In Embassy futures are polled in a list by the executor, which obsoletes the need for dedicated stack management. In RTIC and Drone tasks run on a single stack, where tasks always run to completion unless they are interrupted by a higher-priority task. Tock and Drone processes, and Hubris and R3 tasks, run in separate, MPU protected address spaces. Only Tock and Drone support dynamic creating of processes, all other stacks require their tasks to be statically defined at compile time.

3) Interrupt handling: Interrupt handling in Drone, RTIC and R3 is done by binding a specific interrupt handler task to the interrupt lines, that gets called and runs to completion when the interrupt occurs. In Hubris interrupts are routed to a statically configured task as a notification, however the task does not get interrupted and has to explicitly check for notifications. In Embassy and Tock interrupts are managed by the respective task/ capsule that controls the periphery.

B. Hardware Abstraction and Application portability

The topic of hardware abstraction needs to be approached from two perspective, that correspond to the two stakeholders in it. The first one is a kernel or OS developer's point of view, who is mainly concerned with the question: How easy is it to add support for a new board to the kernel or a specific driver? And the other one is that of the user, who wishes to port an application from one board to another with minimal effort. We need this differentiation because depending one how the platform-specific details are abstracted, adding support either requires additional work from one side, or the other. The most extreme opposite examples here are on one hand RTIC, which completely offloads all hardware management to the user, and on the other hand Tock, which defines its own hardware interface layer and implements support for a large number of architectures, chips and boards, so that users can write their application once and run it on all supported boards. The latter is facilitated through Tock's userland library libtock-rs [38].

The R3-OS kernel implementation follows a similar approach as Tock in the sense that it implements the majority of the kernel platform-agnostic, and then defines a number of traits that a platform-specific port has to implement. Drivers in R3 however are currently only provided through board-support crates and therefore don't have platform-agnostic code extracted like the Tock capsules. Each board-support crate therefore has to reimplement all drivers themselves, which also result in different traits that a user has to implement depending on which board is used.

Drone provides macros that architecture- and peripheral-access-layer crates can then use for platform-specific configuration. This seems to offer quite a convenient way to add support for new platforms. However, there is currently only support on the architectural layer for RISC-V and Cortex-M, and on the PAC layer for STM32 and Nordic Semi nRFx, which makes it difficult to judge if the macros are suitable for all devices. Like in R3, drivers are currently board specific, with the only supported board being the Raspberry Pi Pico (RP2040). Configuration of the heap memory in Drone is the user's responsibility.

Hubris only supports the Cortex-M architecture, and specifies chip and board specific mappings and adresses in .toml files that are parsed on compile time into static structures. Hubris and Embassy both implement their drivers similar to Tock by implementing a platform-agnostic crate with the driver logic, and requiring boards to supply platform-specific implementation of certain traits. In Embassy the portability of user applications is eased because platform specific crates implement the `embedded_hal` traits for their types. As a result, peripheral access in a user application doesn't differ be-

TABLE I
COMPARISON OF THE DISCUSSED SYSTEMS SOFTWARE

Crate	Task management	Safety measures	Hardware abstraction	Network stack
Embassy	**Tasks:** async functions; polled in a hierarchic state machine, single stack	Fair scheduling between tasks	embassy_hal implemented manually for all supported platforms; drivers split into platform-agnostic and platform-specific crates	smoltcp
RTIC	**Hardware Tasks**: interrupt handler that run to completion; **Software Tasks**: run forever, must cooperatively yield and store their state in local variable	Race-free shared access to objects through critical sections	Responsibility of the user	Responsibility of the user
Tock	**Processes**: dynamically created, own memory region with stack, heap and grants	No unsafe Rust allowed in most capsules; Processes isolated in memory through MPU	Defines own HIL for kernel interfaces; capsules are platform agnostic, but define traits that a board needs to implement	In-house
Hubris	**Tasks**: separately compiled programs in own address space	Tasks isolated address spaces and run in unprivileged processor mode	Peripheral mapping and memory layout specified through .toml files for each board; driver split into platform-agnostic and platform-specific crates	smoltcp
Drone	**Fiber**: state machines; **Tasks**: single stack, run-to-completion; **Threads**: interrupt handlers; **Processes**: long-running with dynamically allocated stack	Single stack placed on RAM border to prevent overflow, tasks run to completion	Peripheral mappings defined through provided macros, drivers implemented in board support crates	None
R3	**Tasks**: statically defined, separate stacks; scheduled based on priorities	”Object safety”: leverage Rust type system for accessing kernel objects	Platform-specific and -agnostic parts of kernel; drivers implemented in board support crates	None

tween the different boards. However, within embassy the usage of `embedded_hal` doesn’t offer any convenience when adding support for a new board since all mappings are manually defined.

VI. Conclusion

We have discussed the advantages of using Rust for embedded system programming and introduced fundamental primitives and abstraction layers. Based on this, we surveyed different systems software for embedded Rust. Concretely, we discussed the design and architecture of the RTIC and Embassy concurrency frameworks, and the embedded Operating Systems Tock, Hubris, Drone and R3-OS. We then analyzed the networking stacks in some of the implementations to discuss how a complex driver implementation is done in practice. Finally, we provided a comparison of the discussed stacks, with a special focus on hardware abstraction and application portability.

References

[1] Rust on embedded devices working group. [Online]. Available: https://github.com/rust-embedded

[2] Embedded development. [Online]. Available: https://crates.io/categories/embedded

[3] (2023, Jun.). [Online]. Available: https://docs.rs/cortex-m/0.7.7/cortex_m/

[4] nrf52832-pac crate. [Online]. Available: https://docs.rs/nrf52832-pac/0.12.2/nrf52832_pac/

[5] F. Skarman. (2020, Jul.) An overview of the embedded rust ecosystem. [Online]. Available: https://www.youtube.com/watch?v=vLYit_HHPaY

[6] (2023, Jun.) svd2rust. [Online]. Available: https://github.com/rust-embedded/svd2rust

[7] embedded-hal. V0.2.7. [Online]. Available: https://crates.io/crates/embedded-hal

[8] (2023, Jun.) Rtic book. [Online]. Available: https://rtic.rs/2/book/en/

[9] (2023, Jun.) Embassy. [Online]. Available: https://github.com/embassy-rs/embassy

[10] (2023, Jun.) Real-time interrupt-driven concurrency (RTIC). [Online]. Available: https://github.com/rtic-rs/rtic

[11] T. Baker, “Stack-based scheduling of realtime processes,” *Real-Time Systems*, vol. 3, pp. 67–99, 03 1991.

[12] L. Sha, R. Rajkumar, and J. Lehoczky, “Priority inheritance protocols: an approach to real-time synchronization,” *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[13] (2023, Jun.) Thesus OS. [Online]. Available: https://github.com/theseus-os/Theseus

[14] (2023, Jun.) Redox. [Online]. Available: https://gitlab.redox-os.org/redox-os/redox/

[15] (2023, Jun.) RustyHermit. [Online]. Available: https://github.com/hermitcore/rusty-hermit

[16] (2023, Jun.) Kerla. [Online]. Available: https://github.com/nuta/kerla

[17] (2023, Jun.) rCore. [Online]. Available: https://github.com/rcore-os/rCore
[18] (2023, Jun.) Project sparrow. [Online]. Available: https://github.com/AmbiML/sparrow-manifest
[19] (2023, Jun.) FreeRTOS. [Online]. Available: https://github.com/hashmismatch/freertos.rs
[20] (2023, Jun.) bern-rtos. [Online]. Available: https://gitlab.com/bern-rtos/bern-rtos
[21] (2023, Jun.) K5 microkernel. [Online]. Available: https://github.com/sphw/k5
[22] (2023, Jun.) Tock. [Online]. Available: https://github.com/tock/tock
[23] (2023, Jun.) Hubris. [Online]. Available: https://github.com/oxidecomputer/hubris
[24] (2023, Jun.) Drone OS. [Online]. Available: https://github.com/drone-os
[25] (2023, Jun.) R3-OS. [Online]. Available: https://github.com/r3-os/r3
[26] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, "Multiprogramming a 64kb computer safely and efficiently," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 234–251. [Online]. Available: https://doi.org/10.1145/3132747.3132786
[27] (2023, Jun.) Hubris reference. [Online]. Available: https://oxidecomputer.github.io/hubris/reference/
[28] (2023, Jun.) Hubris FAQ. [Online]. Available: https://github.com/oxidecomputer/hubris/blob/6c200c5/FAQ.mkdn
[29] (2023, Jan.) Hubris #1083: Plans to restrict kernel ipc to supervisor only? [Online]. Available: https://github.com/oxidecomputer/hubris/issues/1083
[30] (2023, Jun.) The drone embedded operating system. [Online]. Available: https://book.drone-os.com/
[31] core::ops::Generator. [Online]. Available: https://doc.rust-lang.org/nightly/core/ops/trait.Generator.html
[32] (2023, Jun.). [Online]. Available: https://github.com/drone-os/bluepill-blink
[33] (2023, Jun.) smoltcp. [Online]. Available: https://github.com/smoltcp-rs/smoltcp/
[34] B. Cannon. (2016) Network protocols, sans I/O. [Online]. Available: https://sans-io.readthedocs.io/
[35] (2020, Dec.) smoltcp #394: Async/await waker support. [Online]. Available: https://github.com/smoltcp-rs/smoltcp/pull/394
[36] (2023, Jun.) Rust ethernet driver for stm32f* microcontrollers. [Online]. Available: https://github.com/stm32-rs/stm32-eth
[37] embassy-net crate. [Online]. Available: https://docs.embassy.dev/embassy-net
[38] (2023, Jun.) libtock-rs: Rust userland library for tock. [Online]. Available: https://github.com/tock/libtock-rs