CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Rust-based Runtime for the Internet of Things

Fredrik Nilsson
Niklas Adolfsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

A Rust-based Runtime for the Internet of Things

Fredrik Nilsson
Niklas Adolfsson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

A Rust-based Runtime for the Internet of Things
Fredrik Nilsson
Niklas Adolfsson

Supervisor: Olaf Landsiedel
Examiner: Magnus Almgren

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

A Rust-based Runtime for the Internet of Things
Fredrik Nilsson
Niklas Adolfsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As the number of connected devices increases, known as the Internet of Things (IoT), reliability and security become more important. C and C++ are common programming languages for IoT devices, mostly due to their fine-grained memory management and low runtime overhead. As these languages are not memory-safe and memory management has proven to be difficult to handle correctly, problems such as buffer overflows and dangling pointers are likely to increase as the IoT devices become more prevalent. These problems can result in reliability and security issues. In contrast to C and C++, Rust is a memory-safe systems programming language that, just like C and C++, provides low runtime overhead and fine-grained memory management. Potentially, Rust can reduce the number of unreliable and insecure connected devices as a result of memory safety.

Tock is an embedded operating system implemented in Rust, and thus it is inherently safe when it comes to memory management. In this thesis, we investigate if Rust is suitable to develop device drivers. We implement four different device drivers covering a range of functions and evaluate the energy consumption as well as the execution time of our drivers compared to other current state-of-the-art embedded operating systems. These device drivers enhance the usability of Tock for connected devices. With the device drivers, we show that Tock introduces an overhead in terms of execution speed but has a similar energy consumption in comparison to the current state-of-the-art embedded operating systems. Despite the increased runtime overhead, we argue that the benefits of Tock and Rust, i.e., increased reliability and security along with similar power consumption exceeds the negative aspect. Finally, we conclude that Rust and Tock are appropriate for developing device drivers for IoT devices.

Keywords: IoT, Tock, Rust, Device drivers, Embedded systems, Bluetooth Low Energy, Embedded Operating System

Acknowledgements

We would like thank our supervisor, Olaf Landsiedel, for providing this very interesting thesis topic and for all his guidance and support. Second, we want to thank all contributors of Tock for helping us. Especially, Amit Levy one of the masterminds behind Tock for very interesting discussions and help along the way. Lastly, we would also want to thank our examiner, Magnus Almgren, for being a good examiner.

Fredrik Nilsson, Gothenburg, June 2017
Niklas Adolfsson, Gothenburg, June 2017

Contents

List of Figures

List of Tables

1
Introduction

The number of connected devices rapidly increases as traditional physical systems such as refrigerators, cars and toothbrushes are becoming connected [1]. Embedded systems are most often the core of IoT devices as a result of their low power consumption and low price. Internet of Things (IoT) or cyber-physical systems (CPS) commonly describe such connected devices.

Resource constraints in embedded systems limit the number of suitable programming languages [2]. The C programming language is a popular language to use when programming embedded systems [2]. C introduces very little overhead in terms of memory usage and runtime. In C, it is up to the programmer to explicitly handle the memory management. Manual memory management has proven to be a difficult task for many programmers to grasp and is often the reason for faulty programs [3]. An incorrect implementation of the memory management can result in increased time and expenses when it comes to development because finding such bugs is hard. In addition, when such bugs are not detected before deployment, it can result in safety, security and privacy issues [4]. As an example, a faulty implementation, e.g., a buffer overflow vulnerability, can yield in memory corruption that in turn can lead to unintended execution. Therefore, malicious parties can exploit such faulty implementations to gain control of a system. A team of researchers demonstrated an example of such an attack in 2011. The researchers exploited a buffer-overflow vulnerability in the CD player of a car and acquired control of safety critical subsystems in the whole car [4]. This example shows that programming in C can lead to dangerous results where subsystems such as braking and steering can become attacked.

1.1 Problem statement

Embedded systems are known to have very limited resources but they still (in most cases) require a runtime or operating system to simplify development as they are becoming more complex. An operating system adds an extra layer of abstraction to provide protection and enable several programs to run on the processor in an efficient manner. C and C++ are the most common programming language in the current state-of-the-art embedded operating systems, such as Apache Mynewt, Contiki, ARM mbed and Zephyr. As the number of IoT devices increases so does the usage of these operating systems. To keep the devices reliable and secure, it is essential that the implementations of the operating systems are correct. As the operating

system can be the host of many applications potential bugs can, in turn, make these applications less reliable and secure. By implementing the operating system in C or C++, the risk of introducing such critical bugs increases.

Mozilla introduced Rust [5] as an alternative to C in 2009. Rust is a systems programming language with strong safety guarantees that prevents memory corruption. Rust has the potential to solve the problems that can occur when using C, e.g, dangling pointers and buffer overflows. Tock [6] is an embedded operating system developed in Rust. Currently, Tock is missing a Bluetooth low energy (BLE) stack and encryption support. In most IoT devices BLE is a heavily used component that requires encryption [7]. We introduce the following research questions of this thesis:

- Is Rust suitable for programming device drivers for embedded operating systems?
- How does an embedded operating system implemented in Rust compare to current state-of-the-art implementations in C?

1.2 Contribution

This thesis contributes with the following:

1. Design of device drivers for Tock in Rust that enable communication via BLE, true random number generator (TRNG), temperature sensor and AES encryption and decryption.
2. Implementation of the designed device drivers that take advantage of the runtime that Tock provides.
3. Evaluation of Tock and some of the device drivers compared to current state-of-the-art implementations in C.

1.3 Scope

Design and implementation of device drivers for the embedded operating system Tock comprise the scope of this master thesis.

1.4 Method

This thesis strives to follow the design principles of Tock. These design principles allow us to contribute with device drivers to Tock and merge them into the main repository. By submitting the code to the main repository, the code community should review it which probably will increase the code quality.

1.5 Key results

The key results of this thesis are twofolded. First, we contribute with design and implementation of device drivers to the open source embedded operating system Tock. The device drivers are safe and reliable by relying on the safety guarantees of Rust and Tock. We contribute with four device drivers: BLE, AES, TRNG and a temperature sensor. Second, we provide comparisons between our device drivers in Tock and the current state-of-the-art embedded operating systems. The main results of the evaluation are:

- The power consumption for BLE in Tock is on par with or lower than the other operating systems.
- The performance of the AES128 driver is significantly slower than the other operating systems that rely on mbed TLS. Another implementation, TinyCrypt does, however, perform way worse than Tock. The system calls in Tock include, compared to the other operating systems, a significant overhead.

1.6 Thesis outline

The rest of this thesis consists of six additional chapters. Chapter 2 presents the required background material for embedded systems, IoT devices, operating systems, programming languages and Tock. Chapter 3 presents the related work. Chapter 4 presents and discusses potential designs of device drivers along with our designs of the device drivers. Chapter 5 presents the implementation of the device drivers in Tock. Chapter 6 presents an evaluation the implemented device drivers in Tock compared to state-of-the-art embedded operating system device driver along with a general discussion about Tock and Rust. Finally, Chapter 7 summarizes and concludes the thesis.

2
Background

This chapter introduces the reader to relevant topics regarding the thesis and it is divided into 5 sections. Section 2.1 presents the concept of Internet of Things. Section 2.2 presents a new safe systems programming language named Rust which is an alternative to C and C++ when it comes to the development of Internet of Things applications. Section 2.3 presents what an operating system is and what is special with operating systems when it comes to Internet Of Things. Section 2.4 presents a safe operating system for the Internet of Things named Tock that relies on the safety features of Rust. Finally, Section 2.5 presents an overview of the Bluetooth Low Energy protocol, a slimmed version of Bluetooth protocol for low power applications.

2.1 Internet of Things

The number of connected devices rapidly increase as our society gets more and more connected [1]. The increase reflects mostly that traditional physical and embedded devices are becoming connected. Previously, these systems did not have wireless communication capabilities but are now equipped with chips that enable wireless connectivity via several network communication protocols such as WiFi, BLE, ANT+, and ZigBee. Most of the data that these systems generate are sensor data. This data is available for further processing or simply transferred to other devices. We commonly refer to these connected devices as Internet of Things (IoT) and sometimes as Cyber-Physical-Systems (CPS). Example of such devices are:

- medical devices (e.g. pacemaker)
- wearables (e.g. smartwatch, fitness bracelets)
- smart homes (e.g. refrigerators, thermostats)
- connected cars

The characteristics of IoT devices are that they consist of embedded computers with very limited resources, e.g., computing power, memory, and energy, and have access to environmental sensors. The two main reasons for the limited resources are cost and convenience. The average user would not like to spend two times as much money for an electrical toothbrush due to a more powerful processor that enables real-time analysis of the brushing. The other factor is convenience which relates to power consumption. A typical end user would neither like a power cord connected to the toothbrush nor having to charge it after each brush. Hence, the power con-

sumption should be low so that the devices do not need external power. It should also be able to run on battery for a very long time without replacement or recharging.

Embedded computers typically consist of an 8, 16 or 32-bit microcontroller with RAM and flash in the magnitude of hundreds of kilobytes, and wireless networking capabilities such as Bluetooth. Due to the low performance of the embedded computers, one should carefully consider where to spend the resources. One such consideration is the choice of programming language. Most high-level languages such as Java and Python include a runtime overhead. Due to the low overhead of C and C++, these are the main languages for developing embedded systems.

2.1.1 nRF51-DK

nRF51-DK is a development board designed for prototyping IoT applications with BLE, ANT and 2.4 GHz radio capabilities. Two processors are present on the nRF51-DK development board. One of them is a small interface processor that uses a virtual COM-port over USB (UART) to program the main processor via an external computer. The processor also provides debugging via SWD[1]. The other processor is the main processor which handles all other tasks on the chip, e.g., provide all the functionality to access the hardware. This is an ARM Cortex-M family processor that is common in the IoT industry. This processor comes from the ARM Cortex-M family which is commonly used in the IoT industry. More specifically, it is an M0 32-bit processor that operates with a 16 MHz crystal. The development board also comes with 256 kB of flash and 32 kB of RAM. Apart from featuring radio communication via the 2.4 GHz radio, AES encryption and decryption, error detecting with CRC, temperature sensor, random generation, current measurement, LEDs and push buttons are available and supported by the hardware[2].

To give a greater understanding of what the aforementioned numbers mean in terms of performance, Table 2.1 provides a simple comparison of common of the shelf hardware and the nRF51.

Device	CPU (GHz)	RAM (MB)
Macbook Pro 2016	2.9	8192
Raspberry Pi 3	1.2	1024
Raspberry Pi Zero	1	512
nRF51822	0.016	0.016
Arduino UNO	0.016	0.032

Table 2.1: Hardware comparison

[1]https://www.arm.com/products/processors/serial-wire-debug.php
[2]http://infocenter.nordicsemi.com/pdf/nRF51_RM_v3.0.1.pdf

2.2 Rust

Rust [5] is a relatively young systems programming language. A systems programming language supports the development of operating systems and software that requires direct access to the hardware. The Rust project started out as a hobby project by a former Mozilla employee named Graydon Hoare in 2006. Mozilla adopted the language and has released it as an open source project [5]. The main motivation for Mozilla's interest in Rust has been to reduce the number of bugs and security vulnerabilities in their web browser engine Gecko. Roughly 50 % of all security vulnerabilities detected in Gecko are related to memory corruption namely use after free, array out-of-bounds access, or integer overflow [3]. Incorrect implementations of memory management, mostly in C and C++, cause these issues and bugs, that are generally hard to track and debug. The developers at Mozilla are experienced C++ developers with access to advanced static analysis tools. Despite this, bugs related to buffer overflows and dangling pointers occur, which show the severity of the problem with manual memory management. Rust is a safe programming language, i.e., it handles memory management in a safe matter. Hence, by using Rust Mozilla can reduce the number of problems related to faulty memory management and concurrency. To further reduce the number of bugs and security issues, Mozilla Research develops a web browser engine written in Rust, named Servo [3] that takes advantage of the powerful type system in Rust.

Rust is a statically and strongly typed language, i.e., data types of variables are known at compile time and specified by the programmer or inferred by the compiler [5]. Bugs and errors are therefore easier to find and correct early in the development process. Dynamically typed languages, e.g., Python and Ruby do not have a static analysis of the source code. This means that detection of type errors is more likely to occur at runtime in dynamically typed languages than in statically typed languages but has no impact on logical errors [8]. A fundamental design decision of Rust is the zero cost of abstractions, where Rust prioritizes checks at compile time over runtime. To enable zero cost abstractions without significant checks at runtime, Rust relies on its type system [5] instead of using a garbage collector. Rust prevents a set of issues: 1) race conditions, 2) use-after-free 3) integer overflow, 4) buffer overflows and 5) iterator invalidation. Memory management is therefore safe in Rust with strong safety guarantees by default. In contrast, in languages that are not memory safe, e.g., C and C++, the compilers cannot detect memory corruption faults.

Furthermore, Rust is also a multi-paradigm language that aims to combine features from both imperative languages and functional languages. It provides high-level features such as generics, type-classes, high-order functions, algebraic data types and anonymous functions, as well as statements with side-effects and fine-grained memory control as C/C++ but in a memory safe matter.

Due to the absence of a garbage collector, the execution is predictable without additional runtime overhead that a garbage collector introduces. Predictable execution and low runtime overhead are important features in applications with limited resources and timing constraints.

2.2.1 Ownership

Variables in Rust are immutable by default, i.e., modifications of the variables are not possible. The “let“ keyword declares variables and “mut” declares that modification of the data is possible, i.e., a mutable variable. A modification of an immutable variable generates a compilation error. The variables can be either explicitly typed or the compiler can determine the type which Listing 2.1 shows.

Listing 2.1: Variable bindings

```
1    // type-inferred
2    let a = 1;
3    // explicitly typed
4    let b: u32 = 1;
5    // mutable variable
6    let mut c: 1;
7    // compilation error
8    a = 1337;
```

Rust’s object model relies on a concept named ownership where each resource has exactly one owner. The ownership model enforces that a resource can have exactly one variable binding. A variable binding that moves to a new context is no longer valid, i.e., it becomes consumed. Listing 2.2 shows an example where the ownership of the variable “str” moves from the main function to the function “func”. The compiler generates an error if anyone tries to access a moved variable, in this case, “str” which line 7 shows.

Listing 2.2: Ownership move (move semantics)

```
1    // owned in the context
2    fn main {
3        let str = "HelloWorld".to_string();
4        // move i.e. ownership is transferred to the function func
5        func(str);
6        // compilation error: value moved
7        print!("str: {}", str);
8    }
```

Another important aspect regarding ownership is that each variable has a scope for how long it is the bound to the resource. Unless the ownership of a resource changes, resources are freed once their bound variables go out of scope. In Listing 2.3 we show an example where the variable “b” is deallocated once the program exits the block on line 8.

Listing 2.3: Variable scope

```
// owned in the context
fn main {
    let a: &usize;
    // local scope
    {
        let b: usize = 1337;
        a = &b;
    }
    // b gets deallocated here
    // compilation error
    print!("a: {}", a);
}
```

As Rust keeps track of ownership at compile both programs in Listing 2.2 and Listing 2.3 generate errors. However, for primitive types, such as integers the ownership does not change since Rust can simply copy those types.

2.2.2 Borrowing

To enable sharing of objects without transferring ownership, Rust uses borrows. Fundamentally, a borrow is a reference to the owned variable, "&" character declares a borrow in Rust. Any borrow must have a scope that is not exceeding the owner. There are two types of borrows, immutable and mutable borrows. A resource can have either an arbitrary number immutable borrows or exactly one mutable borrow. However, it is only possible to have one of the borrow types at the same time. Listing 2.4 presents a valid program that passes an immutable borrow to the function "func" and then prints the string. This is a modification to the program in Listing 2.2.

Listing 2.4: Immutable borrow

```
// owned in the context
fn main {
    let str = "HelloWorld".to_string();
    // immutable borrows of str
    func(&str);
    // ok
    print!("str: {}", str);
}

fn func(s: &String) {
    print!("{}\r\n", s);
}
```

2.2.3 Lifetimes

A common problem while programming in C and C++ is dangling pointers. Dangling pointers are pointers that point to a variable that no longer exists. To avoid dangling pointers can be quite a complex task, especially when passing references around to different entities, then it can become hard to keep track of whether a reference is valid or not. Rust relies on the concept called lifetime to cope with for example dangling pointers. Lifetimes apply to both owned variables and borrows. Rust utilizes lifetimes to deduce whether a given variable binding points to a valid resource or not and thus, prevents errors regarding memory corruption. Listing 2.5 illustrates how lifetimes work in Rust.

Listing 2.5: Lifetimes in Rust

```
fn main() {
    let mut i = 1; // ‘i‘ lifetime starts.
    {
        let x = 5; // ‘borrow1‘ lifetime starts
        i = x     // error, x has a shorter lifetime than i
    } // ‘x‘ lifetime ends
}  // ‘i‘ lifetime end
```

Most often, the compiler can infer the lifetime, but sometimes the programmer must explicitly annotate the lifetimes. Complex data types such as structs, typically need manual lifetime annotations compared to primitive data types such as integers. An example when the programmer must manually assign the lifetime is when a function returns a reference to a variable. The following describes the syntax for assigning lifetime to a reference in a function: foo<’lifetime> bar(‘lifetime variable: &some_struct){ ... }

2.2.4 Unsafe Rust

As Rust is a systems programming language, it is essential that the programmer can circumvent the type system to enable, for example, dereferencing raw memory, inline assembly code or invoke C code which are tasks that are common in systems programming. Rust provides such functionality by another language inside of Rust namely “unsafe Rust” and it provides the same functionality as ordinary Rust with a few exceptions. From the documentation of Rust [9], one can read that unsafe Rust allows:

- *Dereferencing a raw pointer*
- *Calling an unsafe function or method*
- *Accessing or modifying a mutable static variable*
- *Implementing an unsafe trait*

These features make unsafe Rust not memory safe and should be used with care. Unsafe Rust basically tells the compiler that the programmer is aware of the risk but must perform an operation that is not supported by ordinary Rust. To perform

one or more unsafe operations in Rust, the code must reside in an unsafe block. An unsafe block is an ordinary code block with the unsafe annotation. To clarify, the compiler still provides unsafe Rust with all safety checks, except from the four aforementioned operations.

2.3 Operating systems

An operating system is a program that handles a computer system's hardware resources and provides a platform for user applications to run on [10]. The operating system acts as an intermediate device between hardware and user applications. Its responsibilities include utilizing hardware resources, execute user applications and provide protection mechanisms. However, there exist no clear definition what comprises an operating system. Thus, the size of the operating systems can differ from a few kilobytes to several gigabytes. Different designs of operating systems achieve distinctive goals, e.g., high performance, low memory footprint or high reliability and security. Therefore, different designs of operating systems serve different purposes. The kernel is the common divisor of all operating systems, as all operating systems have one.

2.3.1 Kernels

The core of an operating system is the kernel. A kernel has the highest privileges in an operating system and thus, also full control. Its main responsibility is to handle memory, determine which instructions the CPU should execute (scheduling) and handle peripheral devices, e.g., disks, displays, and keyboard. In most operating systems, the first part to load at boot-up is the kernel. The kernel remains in RAM during the entire execution, hence the size of the kernel is of great importance. Moreover, the kernel's address space is, in most operating systems, located in a protected region to which only entities with the highest privileges are granted access. Hardware determines whether an entity is granted access to the protected address space or not. Changing between operating modes is known as context switching. Processes interact with the kernel via system calls. A system call is an abstraction for a user space application to use resources, for example, interact with I/O without direct access to the hardware. The system call triggers a context switch from user mode to privilege mode. Many operating systems use software-based interrupts (traps) to trigger a context switch. Typically, the kernel and user applications are operating in different address spaces. Therefore, a system call will also include an address space switch. Consequently, the overhead of system calls can be significantly higher and because of this, some operating systems run device drivers with full privileges to avoid the overhead that comes with it. There are mainly two different architectures of kernels when it comes to operating systems, monolithic and micro. The following sections explains the key aspects and the differences between the two kernel architectures.

2.3.1.1 Monolithic kernels

Monolithic kernels were the first kind of kernels. In 1964, IBM developed one of the first monolithic kernels for the operating system IBM System/360 [11]. In general, a big and complex static binary with functionality such as device drivers, process and memory management, resource allocation and interrupts characterize the monolithic kernel. It is also typical for a monolithic kernel to execute all operations in the same address space with the highest privileges. With this design there is a single point of failure where potentially a single bug can crash the entire operating system or malicious users can exploit it to gain access to the kernel. However, the kernel in Windows and Linux, which are two of the most popular operating systems, both build on top of a monolithic kernel. As an example of the size of a monolithic kernel, the Linux kernel is 20 million lines of code [12]. The main drawbacks of the monolithic kernels are threefolded: 1) the kernel size 2) lack of extensibility and 3) maintainability [13]. However, Linus Torvalds argues in a kernel debate[3] that it is easier to debug and develop a monolithic kernel compared to other architectures.

2.3.1.2 Microkernel

Microkernels are the direct opposite of monolithic kernels. The goal of a microkernel is to have as small kernel as possible. Apple's operating system for mobile devices, iOS, builds on top of the L4 microkernel [14]. The biggest kernel size in the L4 family consists of 37.6 thousand lines of code [15]. Thus, it is significantly smaller than the monolithic kernel of Linux. A microkernel is only responsible for the core of the operating system and should at least manage address space, inter-process communication (IPC) and thread management [16]. Other services, known as servers, e.g., device drivers, file system, memory management, network stack and user applications, execute as separate processes. These processes are traditional operating system processes, i.e., they have one or more stacks, heap, data, and text segment. Furthermore, the fault isolation that a process has, in terms of independent execution, is valid for servers as well. A server can then potentially crash and then restart without affecting the kernel. Consequently, it makes the microkernel more reliable and secure, but the kernel itself is still a single-point-failure. A smaller kernel is also likely to contain fewer bugs as it contains less code according to previous studies on software fault density [17]. Microkernels require more resources than monolithic kernels since each server executes as a separate process and that the microkernel communicates with other services via IPC instead of invoking them directly via function calls. Moreover, the communication via IPC is far more expensive than function calls. Thus, it results in more context switches and worse performance than monolithic kernels. The first generation of microkernels had an implementation that favored high extensibility which included a big overhead as a result. As an example, one of the first microkernels Mach had 800 cycles ($\approx$ 500 instructions) of pure overhead in its context switch [16]. However, Liedtke [16] showed that there is no fundamental issue with the microkernel itself rather that the implementation is highly architecture dependent.

[3] http://www.oreilly.com/openbook/opensources/book/appa.html

2.3.2 Embedded OS

Embedded systems are systems that serve a specific purpose in a larger system. One such example is all the computers that handle the maneuvering, e.g., braking and steering, of a car. A microprocessor is most often the core in such systems and they typically have limited resources and low performance. Moreover, the design of embedded systems most often serves specific purposes, compared to general-purpose systems. Due to the nature and location of the systems, e.g., placed in an integrated system without (easy) access, physical interaction is most often not possible or very limited. The limited access and in many cases a large number of devices make restarting the device inconvenient, which otherwise can be an easy way to recover from an erroneous state. This makes reliability more important for an embedded operating system than for a general-purpose operating system. Thus reliability is favored in the design of an embedded system rather than performance.

To develop a secure and reliable operating system for embedded systems is hard. One aspect that increases the difficulty is that embedded devices typically does not have a memory protection mechanism such as memory management unit (MMU). Thus, embedded operating systems cannot rely on virtual address spaces as protection. For example, in a general-purpose operating system, once a process tries to write to memory outside its dedicated memory segment, it generates a segmentation fault. Contrary, in an embedded operating system, illegal writes to critical memory addresses are possible and can crash the system. Thus, there is no protection that prevents writing to critical addresses and as a result, the programmer must pay extra attention to avoid this. A possible scenario is that the programmer overwrites the program counter and unexpected executions are possible.

Real-time and embedded operating systems are closely related and commonly used interchangeably. However, a real-time operating system (RTOS) can handle applications with real-time constraints where specific deadlines must be met. For example, in a car, once the driver presses the breaks the task of breaking the car must start within a certain time frame. In this thesis, there is a clear distinction between an RTOS and an embedded operating system. By embedded operating system we refer to an operating system without real-time constraints.

2.4 Tock

Tock [6] [18] is an open source embedded operating system developed in Rust for ARM Cortex-M based devices. The main contributors of Tock are researchers from universities in the United States where a handful commits full time to the project. A couple of the contributors have also been involved with TinyOS [19] which is a well-known embedded operating system. By using Rust, Tock aims to be safe in terms of memory management and supports event-based multitasking. The design goals of Tock are that the operating system should require very little resources, as most embedded operating systems, and at the same time provide high security and reliability.

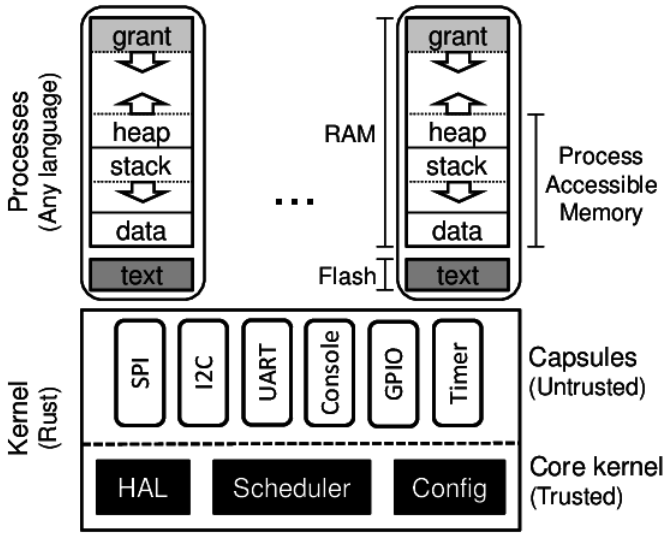

Figure 2.1: Tock architecture. The box at the bottom is the Tock kernel. Inside it, the black boxes are the trusted modules and the white boxes are untrusted. Trusted means that the modules are allowed the unsafe keyword in Rust and untrusted are not. The stacks, located at the top are processes, user space processes, which are executed as separate processes within Tock which have their own grant, heap, stack, data, and text. Furthermore, processes are restricted by the use of MPU and thus cannot access kernel memory. Figure adapted from Tock Design by Tock https://www.tockos.org/documentation/design. Reprinted with permission.

2.4.1 Kernel

Tock's kernel architecture follows the microkernel design [18]. The kernel consists of two parts, one core, and several capsules and Figure 2.1 visualizes the architecture of Tock. It is the core that is responsible for the critical tasks of the operating system, e.g., interaction with the hardware and scheduling of processes, and thus it is relatively small. The core is the only part of the operating system which can and must, use the unsafe keyword of Rust to fulfill its responsibilities. By containing unsafe code, it is up to the programmer to verify the correctness of the code, i.e., Tock regards the core as trusted. Capsules are the other part of the kernel and contain implementations for device drivers and non-system-critical tasks. Tock does not trust capsules but they can communicate with the core and other capsules to accomplish their tasks. Rust's type system guarantees memory isolation between capsules. This mimics the goal of isolated processes, i.e., they are safe and cannot corrupt other capsules. Furthermore, capsules do not introduce any runtime overhead while they still enable secure drivers by checks at compile time. However, a capsule can in fact crash, but this is very unlikely. What is more troublesome is that nothing prevents capsules from starving the kernel. As an example, capsules characteristically implement device drivers such as Bluetooth and timers which rely on functionality from the core, e.g., radio access and usage of a real-time clock. To make capsules available to the rest of the system, e.g., user processes and other capsules, Tock initiates and configures them at boot time.

2.4.2 Userland

Userland contains processes that run with limited privileges outside the kernel [18], in most cases as user applications. However, it is also possible to put services as separate processes. In the case where a process crashes, it can restart without interfering with the kernel. Tock supports execution of several processes concurrently. It is also possible to load processes at runtime without restarting the kernel. Application developers commonly create user applications or services that utilize device drivers and modules provided by the kernel. Examples of user applications are toggling LEDs, reading environmental sensors and utilizing both cryptography and radio communication.

Compared to general-purpose operating systems, Tock does not provide the notation of virtual memory. The absence of virtual memory is because Tock is based on the ARM-Cortex-M architecture, which does not provide an MMU. MMUs are known to consume a lot of energy and require lots of memory, which motivates the design choice to not include it.

Tock enables development of services and user applications in the most programming languages that support Position Independent Code (PIC) and the ARM Cortex-M architecture family. However, currently, C and C++ are the most common languages for writing user applications. Libraries are available when programming user applications in Tock and one can use most of the C standard libraries provided by NewLib. NewLib aims to enable most common C libraries but for embedded systems [20]. Processes can communicate with the kernel via system calls (further described in Section 2.4.3) and between themselves via inter-process communication (IPC).

2.4.3 System calls

This section describes the different system calls available in Tock and their typical use cases. The system calls in Tock enable user space processes to interact with the kernel via a dedicated interface and decouples the user space processes from the kernel.

2.4.3.1 Command

The command system call instructs or commands the kernel to execute a specific task [18]. This system call provides an interface for sending commands with limited data to the kernel. Each command takes three arguments, driver number, command number, and subcommand number. Together these arguments instruct the kernel what operation to execute. The first argument, driver number, specifies which driver that should handle the request. Drivers can accept several commands and hence each command must have a unique command number which is the second argument. The third argument, subcommand, provides more fine-grained instructions to the command. One example of a command could be to toggle a LED. Firstly, the kernel deduces which driver to call, indicated by the driver number. Once the kernel has

assigned the task to the correct driver, the capsule will execute the desired command specified by the command number, in this case, toggle the LED. Finally, the capsule commands the LED corresponding to the last argument to blink.

2.4.3.2 Allow

The allow system call enables sharing of data between the userland and the kernel space [18], i.e., allow the kernel to use memory allocated by the user space process. Applications require in many cases a return value or a result from the kernel to make the application more interactive, i.e., read sensor data and present it to the end user. Like the command call, a driver number specifies the driver and an allow number specifies the specific allow call. Additionally, a pointer to a predefined buffer specifies where the kernel should either read or write data from and to the user space process respectively. Since the pointer does not contain any additional information such as the size of a buffer, the size of the buffer is also provided in the system call. A common scenario is to combine the allow and command calls. The allow call first defines where to store the result. With a command call, the user space process instructs the kernel, for example, to read sensor data to the "allowed" buffer. In this way, the user space processes can share data with the kernel.

2.4.3.3 Subscribe

The subscribe system call assigns a callback function, defined in the user application, for the kernel to call once it reaches some specific criteria [18]. This system call enables the user space process to continue to execute while waiting for a result or response from the kernel. The subscribe call carries, besides the driver number and subscribe number, a pointer to a function, i.e., a callback function, defined by the user space process. As the kernel assigns the correct capsule to manage the system call, it also stores the callback. Once some specific criteria or state is meet or reached in the kernel, it fires the callback. Delays and timers typically utilize the subscribe call. One example is to delay the execution of a function. As the kernel's clock reaches the specified delay, it fires the callback.

2.4.3.4 Yield

The yield system call suspends the current process [18]. In cases where the process is dependent on the work of the kernel, e.g., waiting for I/O to complete, the kernel can suspend the process while the process waits. The scheduler will not schedule the process until it receives a callback from the kernel. Hence, a yield command is dependent on the subscribe call.

2.4.3.5 Memop

Memop provides information about the process itself that enables expansion of the process size, during runtime [18]. There are two ways to expand the process size. The first way is to specify the number of extra bytes for the expansion. The second method is to specify the end of the requested memory segment, i.e., from one memory address to another.

2.4.4 System call data flow

This section explains the data flow from a user space process to the kernel. First, the user space process, i.e., a process, invokes a system call. The scheduler continuously runs an event loop which checks for events, e.g., interrupts and system calls. As the scheduler detects the system call it also classifies it, i.e., which kind of system call it is. Once classified as a supported system call, the kernel assigns the system call to the corresponding capsule, determined by the driver number. Capsules use the supplied command number to take appropriate action, e.g., turn on a LED. To determine which LED to turn on, the capsules use the additional subcommand number. The capsules have access to the hardware via a hardware interface layer. Through this interface, the capsule can indirectly modify hardware, i.e., turn on the LED, and finalize the system call. As the system call finalizes, the kernel signals to the user space process whether the call was successful or not. Figure 2.2 illustrates the data flow from the start of a system call to its end. Red boxes illustrate code that may use the unsafe keyword in Rust, e.g., perform unsafe operations. With Tock terminology, this code is trusted. Green boxes illustrate code that must not use the unsafe keyword in Rust. Hence, Rust enforces all safety guarantees, and with Tock terminology this code is untrusted.

2.4.5 Example of user application

Programs that interact with the kernel can consist of several system calls. By combining different system calls, the kernel can read and write data from and to a user application respectively, without halting the user application. The following is a description of the system calls that construct a simple encryption application (assumed that the encryption key is preconfigured):

1. **Subscribe call** - Specifies the callback function that the kernel should fire once it finishes the encryption. The callback function can, for example, print the encrypted data.
2. **Allow call** - Shares a buffer with the kernel to enable it to read a buffer. In this scenario, the buffer contains the plaintext.
3. **Command call** - Instructs the kernel to start the encryption operation.

As the encryption operation starts, the kernel reads the plaintext into a buffer owned by the kernel. The kernel encrypts the buffer and writes the ciphertext back to the shared buffer. Once this operation finishes, the kernel fires the callback. In the user application, the callback function finally reads the buffer and prints the encrypted plaintext, i.e., ciphertext.

2.5 Bluetooth Low Energy

Bluetooth is a standard for radio communication which evolved to replace wired serial connections. Bluetooth low energy (BLE) is a version of Bluetooth that is suitable for low powered and inexpensive devices with small sizes and short ranges of operation [21]. Today, many devices feature Bluetooth. To transfer files, stream music and control remote devices are typical use cases of Bluetooth. Bluetooth is an

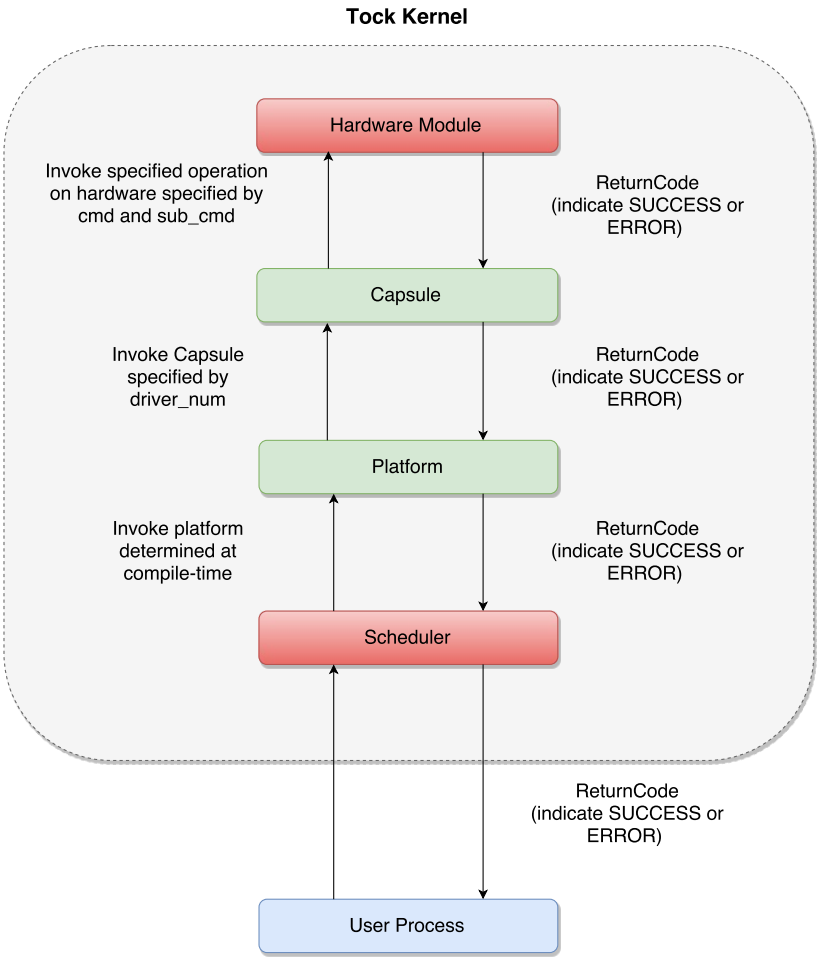

Figure 2.2: Tock system call. Showing the interaction flow upon a system call from a user space process to the Tock kernel. The blue box at the bottom is the user process outside the Tock kernel and the big gray box is the Tock kernel. Inside the Tock kernel, red boxes are the trusted modules and green boxes are the untrusted modules. The return code is returned to the user space process after the Tock kernel has handled the system call and it is either a success or an error code.

ad hoc protocol since it does not rely on any external infrastructure. An example of a protocol that requires preexisting infrastructure, i.e., the opposite of ad hoc, is WiFi. A router is required in order to enable communication between devices over WiFi. Communication between devices over WiFi requires a router. On the contrary, to initiate and utilize the Bluetooth protocol, the only requirements are two or more devices that can use the Bluetooth protocol, hence the description as an ad hoc protocol. Bluetooth does not require additional configuration or installation if drivers and hardware are available for the devices that should communicate. This design makes it fast and easy to setup connections between different devices.

The Bluetooth's Special Interest Group (SIG) provides standards that all variants and versions of Bluetooth must follow. This standard defines everything from different layers in the protocol stack to how certain devices shall communicate. For example, the standard provides details of how devices such as heart rate monitors

and temperature sensors shall communicate. It also defines the format of data, e.g., if sensor data should contain a timestamp or not. By following these standards, it becomes easy to connect independent devices. Designing an application that acquires heartbeats from a heart rate sensor does not require any prior knowledge about the actual hardware.

There are two common configurations of BLE devices, broadcasters and scanners. Broadcasters are devices with a single purpose to only send advertisements, with or without data. A typical scenario where a broadcaster is sufficient is in cases where the device should send data to multiple devices and exchange of data is not necessary. Scanners are the counterpart of broadcasters, i.e., they are the ones to receive or scan for advertisements. It is possible to change the configuration, e.g., a swap between the broadcaster and scanner configuration, of the device at runtime. This enables a device to start out as an advertiser, which announces that it is possible to connect to it, and then change the configuration so that it can have direct communication with a single device. Connections between devices provide a private channel for devices to communicate through. To establish a connection, the advertiser must advertise that it is a connectable device. As another device recognizes the advertising device, it can send a connection request. The two devices establish a bidirectional connection once the advertiser accepts the request [7].

The BLE protocol can use 40 different channels for transmissions and they span from 2402 MHz to 2480 MHz. Between each channel, there is a 2 MHz separation. Channel hopping prevents congestion and interference from other devices in the network. Upon connection, the devices share the settings for channel hopping to enable communication on the same channels. Advertisements use a simple form of channel hopping. To improve the chances of that advertisements reach their destination, the broadcaster sends advertisements on three channels “simultaneously”. The only delay between the transmissions depends on how big the payload is and how fast it can send one packet after the other. The three advertisement channels are 37, 38 and 39. Channel 37 is the first channel in the channel span and 39 the last, 38 is roughly in the middle. The placement of the channels further reduces interference from other devices and protocols. Between each advertisement cycle, there is a 10 milliseconds to 10 seconds delay dependent on the application. This process continues until the advertisement stops.

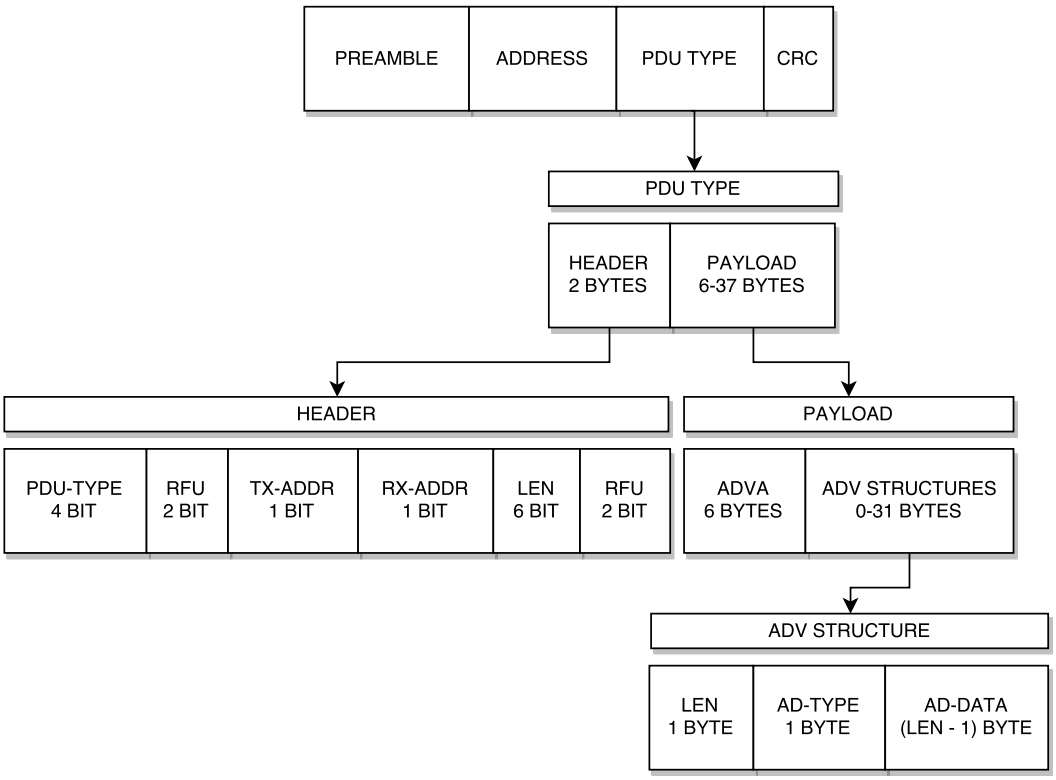

Figure 2.3: BLE advertisement packet structure. The figure illustrates a BLE packet specified by SIG. It describes the different layers from over the air structure to the different data structures to transmit in the BLE packet. The payload can consist of zero or several advertisement structures as long as it does not exceed 31 bytes.

2.5.1 Packet structure

To comply with the BLE standard, one must follow the packet structure of Figure 2.3. Basically, each packet can be explained with the three layers of the figure. The over the air packet must consist of a preamble, address, PDU type and a CRC field. The address specifies which address to send to. The address could either be the address of a receiver or the predefined address for BLE advertisements, i.e, *0x8E89BED6*. PDU type is the type of the BLE packet which in the case of advertisements, consists of a header and a payload. The header describes the entire BLE packet. PDU type defines the behavior of the device, for example, if it is connectable or not. RFU are reserved bits for further development. TX and RX define if the addresses are public or random. The payload can hold zero or several advertisement structures. Each structure consists of a length, type, and data. Figure 2.3 demonstrates the package when one advertisement structure is present.

3
Related work

This section consists of two parts. First, we compare and discuss Tock to a general-purpose operating system, Singularity, that has a design which favors reliability and security. Second, we discuss Tock in comparison to another IoT operating system, Contiki.

3.1 Singularity

Singularity [22] is a research operating system developed by Microsoft Research. The authors argue that the usage of computers is different today compared to back in the 1960's and 1970's, e.g., networking plays a bigger role today. Furthermore, the authors claim that modern commercial operating systems have not been adopted to ensure sufficient protection mechanism for the end-user. The design of Singularity favors dependability over performance. By relying on a microkernel architecture, Singularity can keep the kernel as small as possible which increases dependability as the trusted computing base becomes smaller. It enables other modules to execute outside the kernel as separated processes such as device drivers. For most commercial operating systems, e.g., Linux and Windows, the goal is rather to provide high performance to the end user.

Most parts of Singularity are based on the programming language Sing# [23]. Sing# is type-safe with a garbage collector designed and implemented specifically for Singularity. Sing# extends C# and Spec# and supports low-level programming, i.e., systems programming, which is essential for developing system-level software. Singularity introduces the following architectural features:

- Software-Isolated Processes (SIPs)
- Contract-Based Channels
- Exchange heap
- Manifested-Based Programs (MBP)

SIPs are like traditional operating-system processes which consist of text, data, heap and one or more threads (stacks). SIPs differ from traditional processes in three major ways:

- SIPs rely on software rather than hardware for protection and safety.
- Memory between SIPs is not shared.
- SIP operations are restricted.

Sing# provides static verification at compile-time, i.e., it ensures that a SIP cannot perform any illegal operations that violate the type-system. For instance, if a SIP tries to operate outside its dedicated memory segment it will generate a compiler error instead of a run-time error, e.g., segmentation fault. Each SIP has its own disjoint heap to ensure that a SIP cannot overwrite another SIP's heap. This ensures that a SIP can only operate in its dedicated memory segment. SIPs operations are highly restricted. Prohibited operations include dynamic loading of libraries, sharing of memory and self-modifying code. The authors argue that such operations are inherently dangerous and thus excluded them. These restrictions also simplify the static verification.

Exchange heap is a specific heap for the SIPs to enable sharing of memory outside the SIPs. The regular SIP heap is private, i.e., it is not shared with other SIPs. Thus, the exchange heap enables IPC between SIPs and other parties. Pointers within the exchange heap can only point to data inside the exchange heap itself, ensured by static verification. By only allowing access to a single thread at the time, the exchange heap provides mutual exclusion. Linear types are advanced type-system features that ensure that a resource can only have exactly one owner at the time, which in turn ensures correctness of the exchange heap. Thus, static verification validates data as well as the mutual exclusion for the exchange heap.
The ownership of the exchange heap transfers by data exchange. As an example, when the SIP that owns the exchange heap sends data to a receiving SIP, the sending SIP loses ownership to the receiving SIP. Furthermore, this design does not introduce any overhead by copying the actual data as it relies on pointer passing to the exchange heap.

Contract-based channels provide IPC between SIPs and are based on the exchange heap. A channel is a bi-directional message buffer with exactly two endpoints. The endpoints are asymmetric where they are responsible for receiving and transmitting data respectively. A contract, on the other hand, is a set of rules that specifies how two entities can communicate with each other. For example, a contract can specify protocol states and message declarations. In Sing# these contracts are a part of the language which the compiler verifies at compile time.

MBP are programs defined by a manifest. The manifest describes the program in terms of hardware resources and dependencies on other programs. No code or program can execute in Singularity without a manifest. The manifest verifies the program prior to execution, thus prevents interference between programs and guarantees the availability of hardware.

3.1.1 Discussion

Singularity and Tock are similar in the sense that they are both designed and implemented to favor reliability and safety. Their kernel architectures are also closely related as they are both based on the micro-kernel architecture to decouple the kernel from other modules such as device drivers. Additionally, the type-safe programming languages in both of the operating systems further reduce faulty implementations of memory management.

The biggest difference between Singularity and Tock is probably their goals when it comes to system requirements. Tock must not require many resources, which disqualifies features such as a resource demanding garbage collector. As Singularity "competes" with other general purpose operating systems more resources are possible to expect from the system. Consequently, Singularity can rely on Sing# which, in turn, relies on a garbage collection. As another consequence, device drivers have been designed to be isolated in separated processes in Singularity. Tock, on the other hand, utilizes the safety features of Rust to isolate device drivers but they run in the same process due to a limited amount of memory. This means that Singularity provides higher resource isolation than Tock and for example, device drivers in Singularity can crash without crashing the entire kernel. Contrary, Tock can not provide such guarantees because if a device driver crashes the entire kernel crashes. Furthermore, another difference is that Tock relies on hardware protection of user-space processes via an MPU whereas Singularity relies on software protection for SIPs.

Communication between device drivers in Tock is mainly done via function calls whereas Singularity utilizes its contract-based channels, i.e., IPC. Sing# provides advanced features such as static verification of contract-based channels. This is a direct result of the coevolution of Sing# and Singularity as the operating-system developers have influenced the development of the language. These features enable Singularity to provide more sophisticated dependability features than Tock. As Tock only utilizes Rust, it can sometimes result in workarounds rather than customized language features. A typical case where workarounds are necessary is when dealing with affine types (ownership) in Rust. Whether the extra and in some cases, more sophisticated features of Singularity and Sing# provide higher reliability and security is hard to conclude without further and deeper analysis. The comparison is, as stated earlier, in many cases unfair due to the different amount of resources that the different platforms expect.

3.2 Contiki

Contiki is an embedded operating system developed by Dunkels et al. [24]. The authors' goals have been to implement a lightweight and flexible operating system for IoT devices. It requires only tens of kilobytes of RAM and flash to run. Contiki supports many different platforms as well as wired and wireless communication for protocols such as IPv4, IPv6, 6LoWPAN, RPL and CoAP [25]. These are features which have boosted the popularity of Contiki. The C programming language is the primary language of Contiki.

Dynamic loading of applications is one of the key features of Contiki. Manually and physically updating the software of IoT devices in remote areas quickly becomes infeasible and extremely time-consuming. Loading and updating software dynamically, especially over the air is hence of great importance. To enable this dynamic loading at run time, Contiki separates applications from the kernel. Furthermore, applications are, compared to the kernel, relatively small. Small applications speed up the loading over the air which the authors argue is of great importance to save energy, as IoT devices most often have limited energy resources, while downloading new software. As the core should not change very often, it can be quite big and completely isolated from the applications. The main motivation for these design choices is that IoT devices should be easy to update after deployment.

The Contiki kernel is a hybrid model, i.e., it supports both event-based concurrency and preemptive multithreading. Both models have their benefits and drawbacks. Event-based concurrency simplifies synchronization as it is single threaded and thus locking becomes easier and each event runs to completion. The model does not work well for longer tasks which may block the kernel to react to other events. For longer tasks, the preemptive multithreading is better suited where the tasks can be preempted. Contiki implements event-based concurrency by default. However, libraries provide preemptive multithreading to individual tasks.

3.2.1 Discussion

Contiki and Tock both target embedded systems with limited resources and are in a sense, "competitors". Both operating systems use event-based concurrency and separate their kernels from the user applications. The separation of kernel and user applications enable loading of applications without rebuilding the kernel. Currently, though, Tock does not support any dynamic loading of applications over the air and hence it still requires a connection to a programmer and or computer. As Dunkels et al. [24] state, it is a big advantage of being able to send updated versions of the software over the air. This is especially true if one targets wireless sensor nodes and due to often a large number of devices in wireless sensor networks. However, based on the design of Tock, a similar implementation should be possible to implement in the future. The kernel architecture differs as Tock implements the micro-kernel architecture whereas Contiki implements the layered architecture. Contiki emphasizes to keep the kernel as big as possible to limit the network traffic upon loading user

applications over the air. Contrary, Tock emphasizes to keep the trusted computing based on the kernel as small as possible to decouple dependencies of other modules, e.g., device drivers to enhance reliability and safety. Static verification of Tock ensures that it is less likely to crash due to memory corruption, such guarantees cannot Contiki provide.

4
Design

This chapter presents the design of the device drivers for Tock and consists of four sections. First, in Section 4.1 we present and discuss different design choices for the device drivers. Second, in Section 4.2 we present our device driver architecture that matches the architecture of Tock, which favors reliability and safety. Third, in Section 4.3 we present the hardware requirements. Finally, in Section 4.4 we present the design of the device drivers.

4.1 General design of device drivers

In this section, we discuss alternative design decisions for device drivers. When designing device drivers, Rubin et al. [26] argue that device drivers should provide functionality rather than to dictate it. This means that a device driver should, for example, provide functionality to turn a LED on or off, but not dictate the duration of the different states. Such a design enables the application developers to build software with a large amount of freedom without modifying the kernel. With the previous LED example, it is quite straightforward when it comes to the granularity of the driver whereas more complex device drivers require additional reasoning. A device driver that manages, for example, a radio must provide more functionality, e.g., turn on/off the radio, set transmission frequency, manage addresses, provide integrity and confidentiality. The design must consider which parameters that should be available to the user or if basically everything should be encapsulated and wrapped inside the kernel. We have come up with three different designs where one follows Rubin et al. [26] and the other two ignore these recommendations in favor for fine-grained control and low memory footprint.

4.1.1 Micro design

The micro design places all or most of the logic of the device drivers in different user space libraries and the kernel mainly manipulates hardware without any further checks. Figure 4.1 illustrates this design where the logic of the device drivers along with the user space application are in the blue box and the kernel is in the red box. Moreover, the user space libraries wrap system calls in an arbitrary way to provide customized functionality to interact with the kernel. This means that system calls expose methods which interact with the hardware without any additional checks by the kernel, e.g., power off the radio while another application is utilizing the radio. Thus, this design entails that the user space library considers such scenarios and

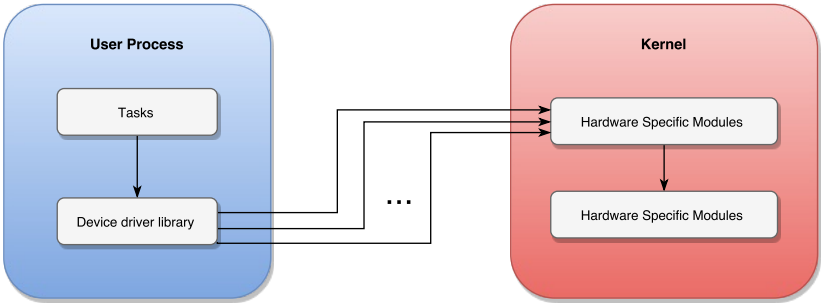

Figure 4.1: Device driver design micro. Favor flexibility to the developer whereas the majority of the complexity is maintained by the device driver library in user space, located in the blue box to left. In order to interact with the hardware, a system call needs to be performed. Thus, the kernel in the red box to the right does only manage basic hardware operations and introduce a system calls for every specific task which the arrows indicate.

other state transitions. With this design, a developer can design and implement customized functionality, e.g., communication protocols without any greater knowledge about the kernel (only how to perform system calls).

In a way, this design moves responsibility from the kernel to the user space libraries and/or the user application. This design introduces more context switches due to the large number of fine-grained system calls that the user space library requires, e.g., turn on radio, set transmitting frequency and set addresses, which otherwise could be handled by the kernel by, for example, one system call, which the arrows indicate in Figure 4.1. Lastly, this design favors flexibility and freedom for the developer.

4.1.2 Hybrid design

In contrast to the micro design, the hybrid design provides the opposite when it comes to flexibility and freedom for the developer of a user application. Figure 4.2 illustrates this design where most of the complexity of the device driver is integrated into the kernel, indicated by the red box. The blue box illustrates the user space application that consist of fewer building blocks, i.e., only device drivers calls to utilize the device drivers. Moreover, the logic of the device drivers, e.g., protocol specific code, integrates into the kernel in this design. Rather than for example, turning the radio on and setting transmitting frequency, the user application only has to tell the kernel that it wants to transmit a packet at a certain frequency and the rest is up to the kernel to handle. Since the kernel has more responsibilities, it is easier to provide a robust and consistent design where kernel developers provide application critical configurations. One such responsibility could be to keep track of the current state of the device driver, e.g. reject system calls that do not comply with the protocol specification. This design requires updates to the kernel to make modifications of the device driver. While this design increases the difficulty to make modifications to the implementation for a regular developer, one could argue that a developer that desires to make such changes, can modify the kernel anyhow.

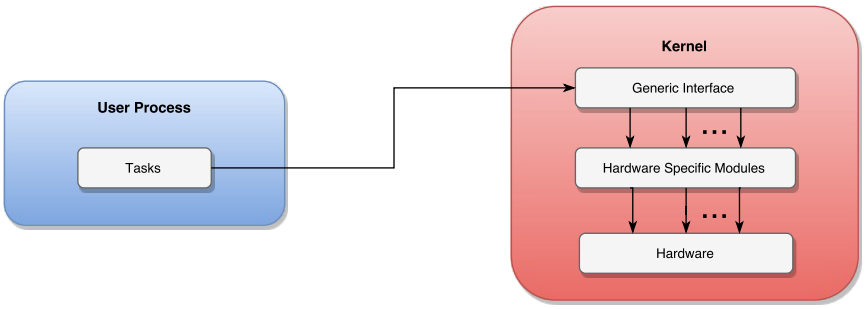

Figure 4.2: Device driver design hybrid. All or most of the device driver complexity is placed in the kernel, in the red box to the right. The user space application in the blue box to the left only invokes a function and the kernel handles the rest. However, this might increase the code size of the kernel but increase the performance. Further, the multiple arrows in the kernel indicate that a system call can utilize multiple kernel resources.

4.1.3 Monolithic design

The most extreme design would be to put all the functionality in the kernel. Such a design would cut off a dedicated area for the user application and require a custom built kernel for each application including the user application. Figure 4.3 illustrates this design where all functionality is integrated into the kernel which the red box indicates. This implies that all functionality are executed in the same address space and do not need to switch between user mode and kernel mode which decreases the context switch overhead and can end up in better performance. However, it is still possible to have some context switches i.e., if the kernel itself supports multi-threading. A possible scenario for this design is when the device has very limited amount of resources and one wants a very fast slimmed kernel for more efficient context switching.

4.1.4 Discussion

The system architecture of Tock (a microkernel with device drivers integrated into the kernel) favors reliability and security by relying on the memory safety of Rust. The goals with our device drivers are to be more secure and more reliable than existing state-of-the-art device drivers and to also be compliant with the design goals of Tock. In this subsection, we discuss the advantages and disadvantages of the different designs from Sections 4.1.1 (micro), 4.1.2 (hybrid) and 4.1.3 (monolithic).

Tock focuses on reliability and security but device drivers cannot execute as separated processes because embedded systems have limited resources. The monolithic design primary achieves a fast memory-optimized kernel with no dedicated user space area. A negative aspect of this design is that it does not provide sufficient abstractions as everything execute in the same address space with direct access to the hardware. Reliability and security are hence not favored and therefore, it is not an option for our design. In contrast, the other two designs provide a dedi-

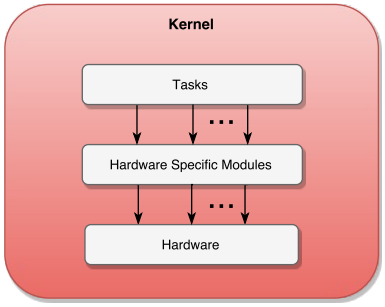

Figure 4.3: Device driver design monolithic. All functionality is integrated into a stripped variant of the kernel with the user space application as a part of the kernel. Unused kernel modules can be removed to provide lower memory footprint.

cated user space area with separate address spaces. The micro design provides more flexibility to the end-user with a relatively small kernel which can be slow since a system call must be invoked to utilize each kernel resource. In many cases, it is not desirable to give unnecessary control to user applications. Especially, since device drivers are part of the same process as the user application. Consequently, a faulty device driver can crash the user application but the kernel would handle it because the user application runs as a separate process outside the kernel. Therefore, the kernel itself would be very reliable and secure in the micro design due to low complexity but user applications become less resilient as these may not be implemented in memory safe languages. The micro design could work in Tock for simple device drivers but it becomes hard to use for more complex device drivers. When it comes to the hybrid design, the device drivers are controlled by the kernel and the user space applications run outside the kernel. By leaving the critical features of the device drivers in the kernel, one can provide simple but yet reliable device drivers to the user application via a well defined interface. In addition to that, the device drivers rely on type safety and are not allowed to use unsafe Rust. This keeps user space applications as simple as possible since library calls can wrap many system calls that the kernel later handles. As a result, it reduces the complexity of the user application and also the likelihood of crashes.

We conclude that the hybrid design matches our requirements the best and all the device drivers will be based on the hybrid design. A simple user space library for each device driver will also be needed to wrap existing system calls. The intention is that these should be as simple as possible but not limit the user. Furthermore, to ensure the reliability and security of the device driver, the kernel is highly responsible for the device driver. However, we argue that the micro design can be used in cases of simple device drivers.

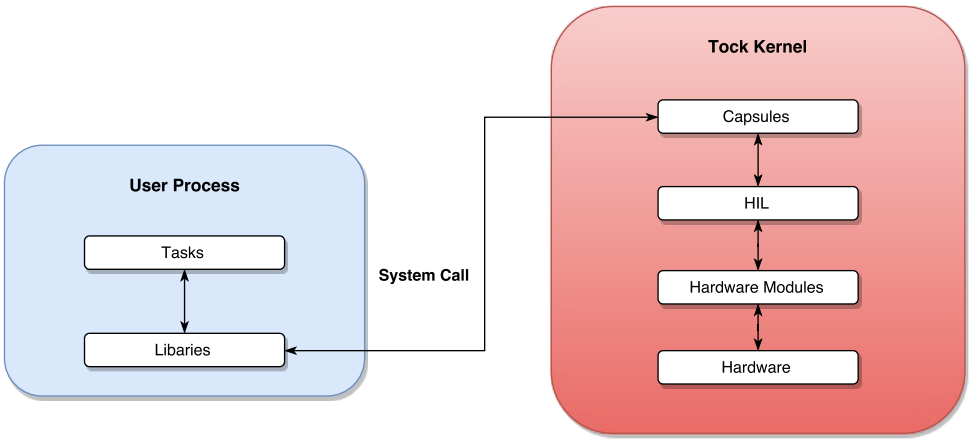

Figure 4.4: Tock device driver architecture. The blue box to the left represents the architecture for the user space application which contains two parts 1) the actual state machine 2) the libraries that provide a uniform usage of the device drivers and standard C functions. Furthermore, the red box to the right is the kernel architecture of the device drivers whereas capsules and hardware modules interact via a dedicated interface, HIL, to utilize hardware resources.

4.2 Device driver architecture

In the following section, we present the building blocks of device drivers in Tock. Each device driver consists of the following components: user space library, capsule, hardware interface layer (HIL) and hardware module.

4.2.1 User space library

The user space library provides specific libraries for each device driver to facilitate the use of the device driver. The general use case of the library is to wrap and provide functionality around the native system calls in Tock. In turn, that provides a uniform way to utilize the device drivers instead of each user space process having to provide its own functionality around the native system calls.

4.2.2 Capsule

Capsule is a module that controls each device driver. It is a module which sits between the user space and the hardware module, further described in Section 4.2.4. System calls ultimately end up in a capsule and it handles the request from user space. A capsule is, in other words, the interface for user space processes to interact with the kernel. As an example, to achieve the blinking effect of a LED, the capsule wraps calls to hardware that turns the LED on and off. Moreover, the capsule is also responsible for managing the hardware resources to prevent multiple interactions with the hardware as it is busy. The capsules are classed as untrusted and are therefore not allowed to contain unsafe Rust. This means that they are very unlikely to crash since they enhance the safety guarantees provided by Rust.

4.2.3 Hardware interface layer (HIL)

The HIL specifies the interface between hardware modules and capsules. Ideally, this should be the only way to access hardware modules and all other functionality should be hidden inside each hardware module. The functions in the interface should be independent of the hardware, i.e., the same functionality should be available for all platforms. As an example, most developers should not be aware of how a LED is turned on in the hardware, hence, the HIL defines a function turn_on_led() for this. HIL typically defines a lot of similar functions, such as to turn a LED on and off.

4.2.4 Hardware module

The hardware module enables interactions with the hardware. These modules should implement the methods specified in the HIL for capsules to utilize a specific hardware dependent features. As opposed to the other modules, the code in hardware module is hardware dependent. Each specific platform may have its own implementation of the HIL since there is no uniform way to interact with hardware dependent details. These modules are unique to each platform and they should contain as little logic as possible, i.e., only code that is unique to the hardware. If this module would contain code that is not dependent on the hardware, several modules would require unnecessary reimplementation when porting the code to new hardware. This is prone to produce errors since changes for one hardware require updates for all other hardware as well, i.e., more code to review. An example of hardware dependent code is how to turn on the power to the radio module and manage interrupts.

4.3 Choice of hardware

Some of the drivers are dependent on the hardware, e.g., radio and temperature sensor. To test and verify those device drivers, we need an actual chip, rather than an emulator. This is especially true for the radio since the protocol must be verified. The specifications of the hardware should meet the requirements of Tock in terms of resources and architecture. Tock only requires 16 kB of RAM and 512 kB of flash, so a lot of chips satisfy these requirements. However, as we evaluate the suitability of Tock and Rust as building blocks of IoT devices, the performance of the hardware should not exceed the performance of common IoT devices. A tighter requirement is the architecture of the processor. Currently, Tock only supports ARM Cortex-M processors. Furthermore, a set of peripherals should be available to utilize drivers and enable wireless connectivity for the device.

4.4 Design of device drivers

In this section, we present the designs of the device drivers in Tock where all the designs are based on the hybrid design, described in Section 4.1.2. The section is divided into four parts where we in the first part presents the design for the BLE

device driver. In part two, we present the design for the AES device driver. In part three, we present the design for the TRNG device driver. Finally, we present the design for the temperature sensor device driver.

4.4.1 BLE device driver design

The goal of BLE driver is to enable transmissions in terms of BLE advertisements. Figure 4.5 illustrates the design. The user application should be able to start and stop BLE advertisement by sending requests to the kernel. Two different requests should be available to the user application, one to start and one to stop the transmission of advertisement, respectively. The start advertising request should contain the advertisement data. Two data fields should compose the advertisement data, one containing the local name and the other the actual data to send, such as sensor measurement. The local name is the name that shows up when most commercial products, e.g., smart phones, scan for Bluetooth devices. Once, the user application sends the request, a capsule should proceed with the request. The capsule should also ensure compliance with the protocol specific requirements for BLE, e.g., transmitting on the right channel and with the right address. Another responsibility is to ensure mutual exclusion of the hardware, i.e., that no more than one user space process can utilize the radio at the same time. The major settings that the capsule is responsible for are:

- Transmission address
- Transmission frequencies
- Timing requirements

The device driver should advertise on address 8e89bed (hexadecimal format) and on three different frequencies. As described in Section 2.5 the use of different channels is to prevent interference with other devices and these channels are 37, 38 and 39. In other words, an advertisement comprises radio transmission on the channels 37, 38 and 39 with no or a very short delay between them. Lastly, the capsule is responsible to continuously send advertisements. Between each advertisement, there should be a delay in the range from 10 milliseconds to 10 seconds. Once the capsule has configured the radio per the requirements, a transmission can start. The capsule should start the transmission by sending a request to the hardware module. After the request is finished, the capsule should notify the user application indicating whether the request was successful or not. The stopping of advertisement relies on the same design as starting but omitting the data.

4.4.2 AES device driver design

The goal of the AES device driver is to provide both encryption and decryption of data in Tock. AES is a state-of-the-art encryption standard for symmetric block ciphers which provides confidentiality but lacks integrity [27]. Implementations of AES are possible both in hardware and software. This device driver relies on hardware support for AES. By utilizing the hardware for encryption, the focus of the device

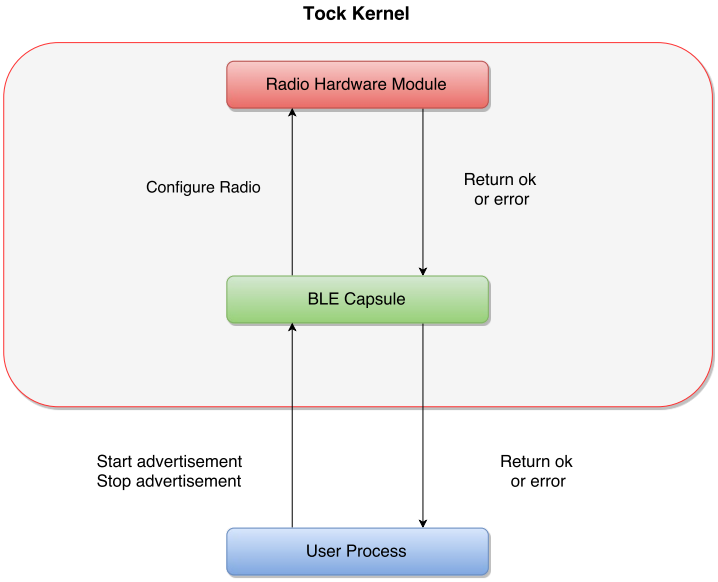

Figure 4.5: Tock BLE device driver design. The figure illustrates the design of the device driver for the BLE device driver. Green indicates that the module is safe, i.e., will not crash, whereas red contains unsafe code. Red modules rely on the fact that the programmers know what they are doing i.e., some of Rust safety features are disabled.

driver itself is prioritized over the implementation of the algorithm. The National Institute of Standards[1] (NIST) recommends five different block cipher modes to provide confidentiality: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR) [28]. ECB is not secure for encrypting more than one block with the same key, i.e., it reveals information about the plaintext [29]. Thus, it is not an alternative as block cipher mode for this driver. CBC, CFB, OFB, and CTR are on the other hand proven to be chosen plaintext attack secure [29] under some restrictions. An example of such a restriction is that nonce-based encryption must only use each counter value once. Rogaway [29] has performed an evaluation of the different modes of operation for block ciphers. He concludes that CTR is the best choice due to its simplicity, small memory footprint, high speed, and good security. The small memory footprint that CTR allocates is important and favorable since the design targets embedded systems with limited resources. The simplicity of CTR entails that only the encryption function of AES along with a nonce is used for converting plaintext and ciphertext respectively. In other words, it mimics or acts as a stream cipher. Moreover, each character of ciphertext is independent of each other and does not rely on padding, which means that it can run in parallel. These properties are especially attractive for embedded systems that implement AES in software. CBC, on the other hand, uses both the encryption and decryption functions and are using padding. Thus, each message must be a multiple of the block even if the message is shorter than the given block length. CFB and OFB are hybrids of CTR and CBC and not further described. We conclude that CTR is the most suitable block cipher mode for IoT devices.

[1] https://www.nist.gov

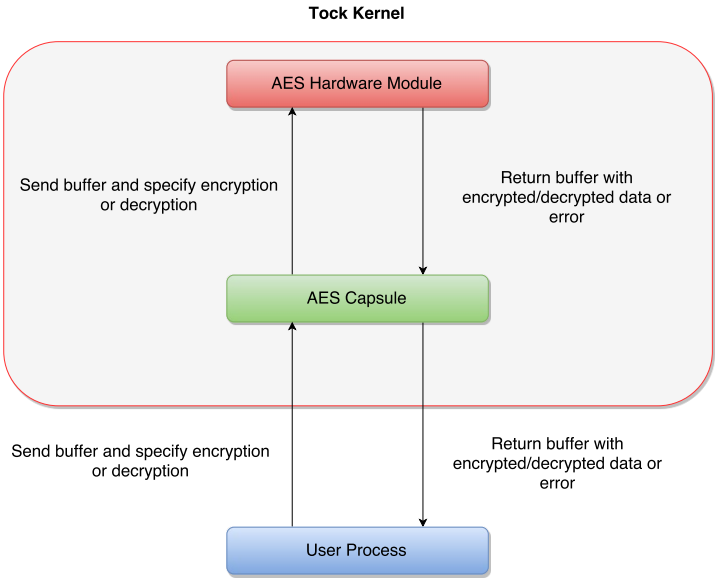

Figure 4.6: Tock AES device driver design. The figure illustrates the design of the device driver for the AES128 counter mode device driver. Green indicates that the module is safe, i.e., will not crash, whereas red contains unsafe code. Red modules rely on the fact that the programmers know what they are doing i.e., some of Rust safety features are disabled.

A user application should be able to initiate the decryption and or encryption operation via a system call. Together with the call to the kernel, the call requires a buffer with the ciphertext or plaintext. Once the call reaches the kernel, a capsule should proceed with the request. The capsule should contain the logic which makes the encryption and decryption comply with the AES standard. Such logic includes checks for key length. Insurance of mutual exclusion is also a responsibility of the capsule. If the capsule does not detect any errors, it should continue with the process by providing the hardware module with the buffer with data from the user application. In the case of errors, the capsule should notify the user application. The AES hardware module should encrypt or decrypt the entire buffer. Once the operation has finished, it should return the result in a buffer to the AES capsule. Finally, the capsule sends the buffer back to the user application.

4.4.3 TRNG device driver design

Currently, Tock supports a couple of different TRNGs but not the one on nRF51. Thus, the goal of this device driver design is to utilize the TRNG on nRF51. The major parts of the device driver already exist in Tock, including the user application, the user space libraries, and the capsule. Therefore, our design of the device driver only includes the hardware module which must comply with the existing design. Figure 4.7 illustrates existing components in yellow and new components in red. The existing design relies on that the user application requests the number of random bytes to generate and passes it to the capsule. After that, the capsule handles the request and in turn requests the hardware module to generate the number of requested random bytes. Once the request is invoked, the hardware module on nRF51

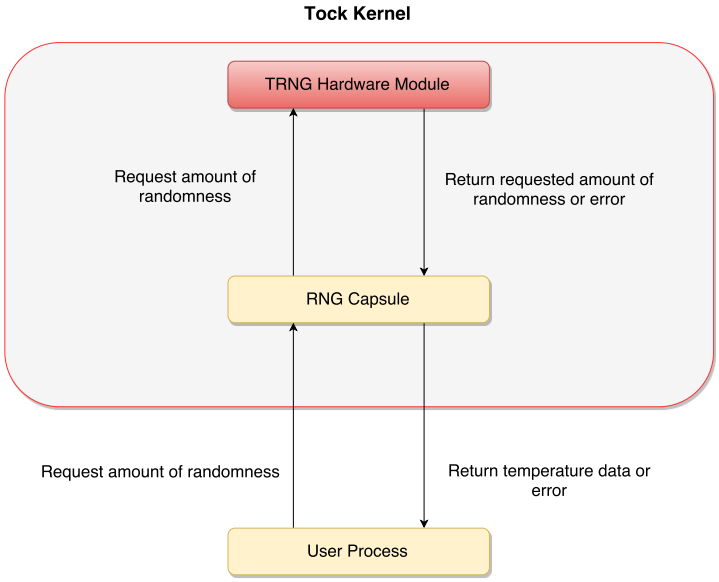

Figure 4.7: Tock TRNG device driver design. The figure illustrates the design of the device driver for the TRNG device driver. Green indicates that the module is safe, i.e., will not crash, whereas red contains unsafe code. Red modules rely on the fact that the programmers know what they are doing, i.e., some of Rust safety features are disabled.

will generate the requested number of random bytes and the capsule continuously keeps track of the number of random bytes generated. When the requested number of bytes has been generated by the hardware module, the capsule notifies the user application and the capsule finishes the invocation to the hardware modules. Figure 4.7 illustrates this sequence.

4.4.4 Temperature sensor device driver design

The goal of this device driver is to utilize the hardware based temperature sensor on nRF51. Currently, in Tock, device drivers for other temperature sensors exist but no general capsule is available. The biggest design decision of the device driver is whether the device driver shall or shall not provide functionality for sampling of temperature data. This design does not consider sampling in the device driver. It just reads a temperature value and sends it back to the user process. Therefore, it is up to the user application how to sample the temperature sensor data. We believe this design decision will provide the basic functionality rather than dictate it.

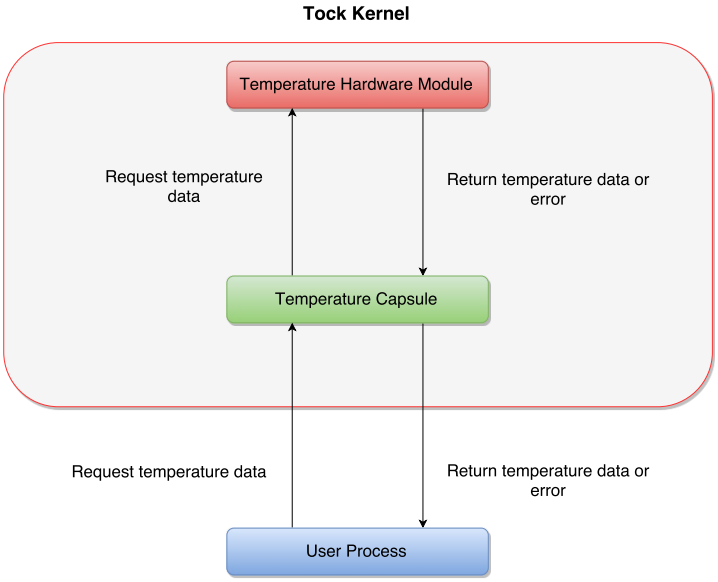

Figure 4.8: Tock temperature sensor device driver design. The figure illustrates the design of the device driver for the temperature sensor device driver. Green indicates that the module is safe, i.e., will not crash, whereas red contains unsafe code. Red modules rely on the fact that the programmers know what they are doing i.e., some of Rust safety features are disabled.

A user application should start the operation with a request to the kernel to retrieve temperature data. The temperature capsule should forward the request to the kernel and provide mutual exclusion for the user application. Both upon successful and non successful attempts, the capsule should notify the user application. Once the hardware module has measured the temperature, it will return the temperature data to the capsule. Finally, the capsule should send the temperature data back to the user application. Figure 4.8 illustrates the design of the device driver.

5
Implementation

This chapter presents the implementation of the device drivers for BLE, AES counter mode, TRNG and temperature sensor for nRF51 in Tock. Along with the device drivers, it also presents user applications that utilize and test the device drivers. The rest of the chapter is structured as follows: First, Section 5.1 presents the implementation details that all device drivers have in common. Second, Section 5.2 presents the implementation of the BLE device driver. Third, Section 5.3 presents the implementation of the AES counter mode device driver. Fourth, Section 5.4 presents the implementation of the true random generator. Finally, Section 5.5 presents the implementation of the temperature sensor device driver.

5.1 Common implementation

This section presents a template device driver implementation in Tock that all other device drivers build on. The reason why is to avoid repeating implementation details such as how to implement different system calls and interchange data with the kernel. Furthermore, we go through the implementation of main components of the device driver architecture in Tock, as described in Section 4.2.

5.1.1 Capsule implementation

The capsule is the main component of the device driver; it manages the system calls from user applications and communicates with the hardware modules. In a sense, the capsule can be regarded as the brain of each device driver. An example of a use case of a capsule is that, upon request from user space, it checks whether the system call is valid or not, i.e., if the kernel supports the requested operation, and it also ensures that the driver is not occupied by another entity. If the checks are positive, the capsule will request the hardware modules to perform the specified operation while providing mutual exclusion. Another common task for the capsule is to follow protocol standards. For example, for BLE, it should build upon the BLE protocol standard to provide communication according to the protocol. Protocol specific logic can include transmitting frequency, channel hopping and packet structure.

Two separate features enable the communication from user applications to the hardware modules in the kernel. First, the capsule must implement the system call interface. This makes the capsule assignable from the scheduler, i.e., the scheduler is able to point a system call to the capsule. Only through system calls is the user

application able to communicate with the kernel. Second, the capsules rely on the different HILs that describe accessible functions in the hardware modules.

A capsule in Tock is a struct with associated data and member functions. The system call interface, that most capsules implement, is a trait in Rust. A trait contains one or more function definitions that are optional to implement. The system call trait defines three functions; command, allow and subscribe. Capsules must implement at least one of these functions to enable the system call interface, but it does not have to implement all of them.

As already mentioned, capsules use the HIL to communicate with hardware modules. However, a capsule can also implement HIL functions. This enables communications in the other direction, i.e., from the hardware module to the capsule. On a high level, this works like a callback since the capsule instructs the hardware module to execute some task, to avoid busy-waiting for the task to finish, the hardware module can call the capsule via the HIL interface once the task completes. These traits are most often called Client traits since the capsule becomes the client which the hardware module calls.

Listing 5.1 shows an example of a theoretical capsule named *Capsule*, lines 26 to 28 show the data of the struct. Furthermore, the *Capsule* implements two traits: Driver, for the system calls on lines 31-41, and Client, for the "callbacks" on lines 44-50. All our device driver implementations build on top of this example capsule.

Listing 5.1: Example Capsule

```
1  pub struct Capsule<'a, D: DeviceDriver + 'a> {
2      driver: &'a D,
3      // add data to struct here
4      data: Cell<usize>,
5  }
6  // system call interface implementation
7  impl<'a, D: DeviceDriver> Driver for Capsule<'a, D> {
8      fn allow(&self, appid: AppId, allow_num: usize, slice:
           AppSlice<Shared, u8>) -> ReturnCode {
9          // implementation of the allow system call
10     }
11     fn subscribe(&self, subscribe_num: usize, callback: Callback) ->
           ReturnCode {
12         // implementation of the subscribe system call
13     }
14
15     fn command(&self, command_num: usize, data: usize, appid: AppId) ->
           ReturnCode {
16         // implementation of the command syscall
17     }
18 }
19 // Client HIL implementation
20 impl<'a, D: DeviceDriver> Client for Capsule<'a, C> {
```

```
21      fn call_from_chips() -> ReturnCode {
22          // implementation of call from chips
23      }
24  }
```

5.1.2 Hardware module implementation

The hardware modules introduce the lowest layer of abstraction, i.e., they interact directly with the hardware by reading and writing to the registers in the processor. Much like the capsules, a hardware module consists of a struct with associated data and methods. These methods are private by default, i.e., not publicly accessible. Initialization and interrupt handlers are, however, publicly available because they are needed outside the hardware module itself for general initialization and interrupt handling. The hardware module implements the predefined methods in the HIL. These methods should be available to the capsules and are therefore also publicly available.

As Tock is single-threaded and relies on event-based concurrency in order to prevent a lengthy process from suspending the kernel, most hardware modules rely on interrupts. The hardware module initiates an operation and enables an interrupt to indicate when an operation is finished. This enables other tasks to execute concurrently which busy-waiting does not. Upon completion, the hardware triggers an interrupt and the kernel eventually handles the associated interrupt. When the hardware module completes a request, it signals the capsule via the client trait by invoking the callback function.

In Rust, manipulation of processor registers is not safe, i.e., reading and writing to raw addresses by dereferencing raw pointers. Thus, hardware modules must be able to circumvent the type system in Rust. No other module than the hardware modules should contain the unsafe keyword in the device drivers. To show what this looks in Rust, we show a common use case of a simplified hardware module in Listing 5.2 that writes directly to memory register which circumvents the type system. First, line 3 shows a declaration variable of *regs*, a constant raw pointer to address 0x4000D000. The address is just for illustration and not the real address. Second, lines 6-10 show a declaration of a struct in a memory map which starts at address 0x4000D000 and ends at 0x4000D008 since the u32 data type size is 4 bytes. Third, on line 15 the pointer *regs* is dereferenced. Note that assigning a raw pointer is safe in Rust but dereferencing a pointer is not. Finally, on line 17 the value 1 is written into address 0x4000D000.

Listing 5.2: Hardware module

```
pub struct HardwareModule<'a> {
    // raw pointer
    regs: *const REGISTER,
}

const REGISTER_BASE: usize = 0x4000D000;
pub struct REGISTER {
    pub START: VolatileCell<u32>,
    pub STOP: VolatileCell<u32>,
}

impl HardwareModule {
    fn start(&self) {
        // dereference pointer regs
        let regs = unsafe { &*self.regs };
        // write 1 to address 0x4000D000
        regs.START.set(1);
    }
}
```

5.1.3 Sharing data between userland and the kernel

Sharing data between user space and the kernel space is important to enable user applications to process data both to and from sensors or other inputs. In Tock, one must follow two steps to share data from the user application to the kernel. Firstly, the user application shares a reference to a, by the user application, allocated buffer with the kernel via the allow system call. We show this in Listing 5.3.

Listing 5.3: Pass buffer by reference from userland

```
    unsigned char data[40] = {0};
    allow(DRIVER_NUM, COMMAND_NUM, (void*)data, sizeof(data));
```

Secondly, the kernel wraps the buffer in an AppSlice data structure, which is unique to Tock. An AppSlice contains the starting address of the buffer and its size. As the capsule handles the allow request, it receives the AppSlice object. The kernel encapsulates this object in two nested data types, first a Container and then an Option data type. Rust requires nested data types to cope with the type safety. Capsules (structs) are immutable and this prevents changes of the internal data, which is troublesome when we want to write to the buffer. The Container data type, which is unique to Tock, solves this by providing interior mutability[1]. In the standard library in Rust, the containers Cell and RefCell provides interior mutability but is not suitable for complex data structures such as an AppSlice. The Option[2] data type is common in functional languages. It encapsulates values of any given

[1]https://doc.rust-lang.org/book/mutability.html
[2]https://doc.rust-lang.org/std/option/

type such as integers and floats but can also be empty. An option data type either contains data or it does not. Variables inside an Option data type can be taken, i.e., moved. Thus, replacement of the option data type, i.e., the buffer, is possible and required when writing to the buffer.

To illustrate how this is done in Rust, Listing 5.4 presents a simplified version of a capsule that receives a shared buffer, transforms the buffer and passes it along to hardware modules. We break down the task into four parts and explain it step-by-step. First, lines 15-19 show the struct itself that contains a reference to the hardware module in *driver*, a container that encapsulates the AppSlice and a TakeCell (a container of static mutable data). The variable *driver* enables the communication to the hardware module via the HIL, the Container contains the AppSlice and we use the TakeCell to convert the AppSlice into a mutable static array. Second, on lines 24-25, a closure accesses to the container object within the Capsule struct. Third, line 26 replaces the existing Option object with a new Option object containing an AppSlice with the buffer passed by the user application. Fourth and finally, lines 31-42 show how to read already stored data in a capsule and pass it along to the hardware modules. To be more specific, line 31 shows how to get access to the Option within the struct by using an iterator. Then to access the content of the Option in the Container requires a closure, line 32 shows this. Moreover, to get access inside the Container, the AppSlice which holds the buffer and the length of the buffer, another closure is needed which line 33 shows. However, because the AppSlice holds a reference to a shared buffer it must be converted into a type that the HIL supports which is a mut [u8] array with static lifetime. Due to that, the capsules are not allowed to contain unsafe code the TakeCell is used to transform the shared buffer into a mut [u8] array with static lifetime. Inside the third closure, the zip function traverses the shared buffer and the TakeCell simultaneously on line 36 and lastly copies the contents of the shared buffer to the TakeCell. The length of the AppSlice buffer determines how many bytes to copy. To conclude, line 39 passes the TakeCell with contents of the AppSlice to the hardware module. Note that, the take function of the TakeCell moves the underlying object and it must be replaced with some other static mut [u8] data to be used further since the object becomes None.

Listing 5.4: Buffer within the Tock kernel

```
1
2  // container of AppSlice
3  pub struct App {
4      buffer: Option<AppSlice<Shared, u8>>,
5  }
6
7  impl Default for App {
8      fn default() -> App {
9          App {
10             buffer: None,
11         }
12     }
```

```
}

pub struct Capsule<'a, D: DeviceDriver + 'a> {
    driver: &'a D,
    apps: Container<App>,
    kernel_tx: TakeCell<'static, [u8]>,
}

impl<'a, D: DeviceDriver> Driver for Capsule<'a, D> {
    fn allow(&self, appid: AppId, allow_num: usize, slice:
        AppSlice<Shared, u8>) -> ReturnCode {
        // replace AppSlice
        self.apps
            .enter(appid, |app, _| {
                app.buffer = Some(slice);
                ReturnCode::SUCCESS
            })
        }
        // collect data in AppSlice and send to chips
        for cntr in self.apps.iter() {
            cntr.enter(|app,_| {
                app.buffer.as_ref().map(|slice| {
                    let len = slice.len();
                    self.kernel_tx.take().map(|buf| {
                        for (out, inp) in
                            buf.iter_mut().zip(slice.as_ref()[0..len].iter())
                            {
                            *out = *inp;
                        }
                        self.driver.hardware_module(buf, len);
                        });
                    });
                });
            });
        }
    }
}
```

5.2 BLE

This section presents the implementation of the BLE device driver that builds on the design presented in Section 4.4.1. The device driver consists mainly of three different modules within: BLE user library, BLE capsule, and BLE hardware module.

5.2.1 User space library

The BLE user space library is built on top of the existing system calls available in Tock. It provides two functions available to the user applications:

- *start_ble_advertisement(name, name_len, data, data_len)* wraps three system calls that starts the BLE advertisement. First, two allow calls share the local name and data with the kernel respectively. Second, the wrapper tells the kernel to start advertising by sending a command.
- *stop_ble_advertisement()* wraps a command system call to tell the kernel to stop sending advertisements

5.2.2 User application

The purpose of the user application is to demonstrate and evaluate the device driver. To utilize the driver, the user application uses the BLE library which provides two functions, one to start sending BLE advertisement and one to stop the transmission. The user application allocates two buffers to advertise, one with the device name and the other one with the data. The device will continue to send advertisements until the user application explicitly calls the stop function.

5.2.3 Capsule

The capsule is the core of the device driver. It enables communication between the user application and makes sure that the device driver follows the BLE standard. To communicate and interact with the user application, the capsule manages system calls. The capsule accepts two allow calls to read the data from the user application, one for the device name and one for the data. Together, the device name and data constitutes the payload of a BLE packet. Due to the limitations of BLE advertisement packets, i.e., the size of the packets, the capsule must ensure that the size of the device name and the data does not exceed a predefined threshold, in this case 31 bytes including the headers. If the combined size is too big the capsule will deny the user application to start sending advertisements. Two commands manage the starting and stopping of the advertisements respectively. Upon a start command, the capsule starts sending advertisements, with the data received from the user application, by invoking the hardware module. Before the hardware module can send the advertisement, the capsule has to configure it. Configuration of the hardware module is more or less just to follow the BLE standard. It is however highly important to enable communication with commercial devices. The standard specifies the settings, e.g., transmitting frequency and address, and the hardware

module exposes functions to configure these parameters to the capsule. The advertisements are periodic and the timing of the advertisements is important. With a larger interval between advertisements the device consumes less energy, but with a too large interval, the advertisements might not reach other devices due to packet collisions or simply bad timing. It is up to the designers to define a good interval since it is not specified in the BLE standard.

The capsule relies on timers to achieve periodicity. As the start command reaches the capsule, it sends the first of three advertisements (see Section 2.5 about BLE). Upon completion of one send request, the capsule starts a timer. The timer capsule manages the timer and hence the BLE capsule can stop its execution. Once the timer fires, the timer calls callback function in the BLE capsule. This callback function starts the transmission of the next advertisement. For the last advertisement, the capsule increases the time of the timer until it all starts over. This process provides bursts of advertisements in a periodic fashion.

5.2.4 Hardware module

The hardware module manages all communication with the hardware, i.e., the radio chip. It is responsible for configuring and handling the radio chip and for setting up the BLE packet. To configure the radio, the hardware module modifies registers that are defined in the data sheet. The packet configuration is slightly more complicated. It requires a mapping between the radio packet and the BLE packet. Three registers in the radio chip constitute the header of a BLE packet: S0, LENGTH, and S1. Figure 2.5.1 shows the structure of a BLE packet, by setting S0 to 2 bytes, LENGTH to 6 bits and S1 to 2 bits, these registers maps to the BLE header. Furthermore, a state machine describes the radio chip where a state register holds the current state. For each state change, the chip generates an interrupt and the state register tells which state the radio is in. This enables an interrupt based design, rather than a busy wait design where the driver occupies all the resources.

5.2.5 Discussion

This device driver requires further work to comply with more of the BLE standard. In the current state, only simple advertisements are available. We motivate this by arguing that our contributions are device drivers rather than a full BLE stack. However, the community of Tock shows a big interest in the driver for their projects. Some other projects rely entirely on an infrastructure around BLE advertisements.

The user application allocates two buffers, one for the device name and one for the data. With this implementation, the capsule has to perform less parsing compared to if one buffer contained all information. When constructing the BLE packet, an array contains all the bytes without any formatting or helper functions.

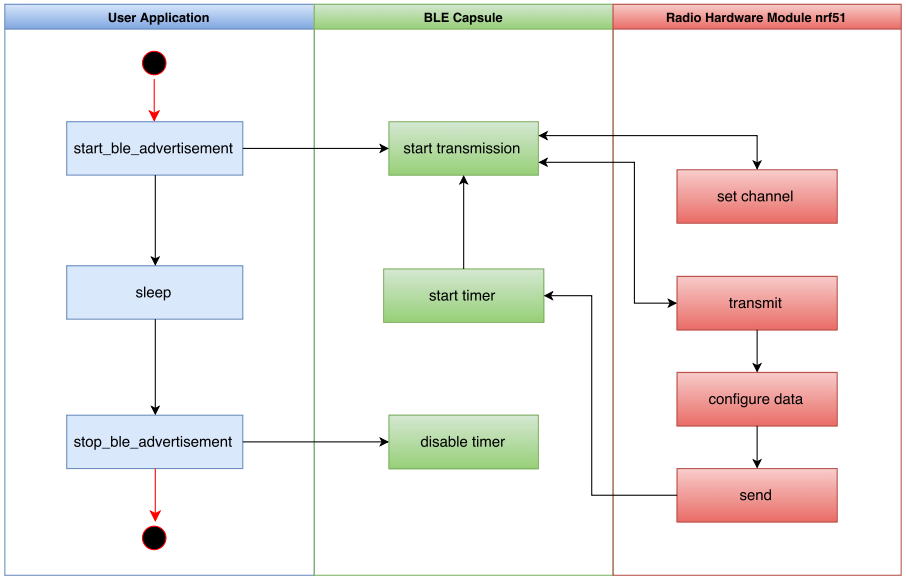

Figure 5.1: Tock BLE device driver implementation. The figure shows the entire BLE device driver. To the left, blue boxes illustrate the user space application with the primary purpose to interact with the Tock kernel via two system calls to the device driver to start and stop periodic advertisements. The middle green boxes are untrusted Tock modules which manage the BLE state machine and virtual timers to send periodic advertisements without suspending the entire kernel. To the right, red boxes illustrate trusted modules which directly manages the radio.

Updating or modifying the packet, other than device name and data, is a troublesome task. Further work should introduce predefined data structures for different types of advertisements with helper functions that easily constructs and modifies the packet. As a summary, we provide a framework for a BLE driver. The driver is functional and can perform simple tasks. Further work includes extending and improving the driver.

5.3 AES-128-CTR

This section presents the implementation of AES-128-CTR device driver in Tock for nRF51 that builds on the design presented in Section 4.4.2. The device driver consists mainly of four different modules within Tock namely AES-128 library, SymmetricEncryption capsule, SymmetricEncryption HIL and AES hardware module for nRF51.

5.3.1 Library

The library supports AES-128-CTR in both software and in hardware. It is assumed that the user application provides a key, initial counter, and plain/ciphertext to the library. The key is the secret key, required for encryption and decryption. The initial counter enables a *random* start of the encryption.

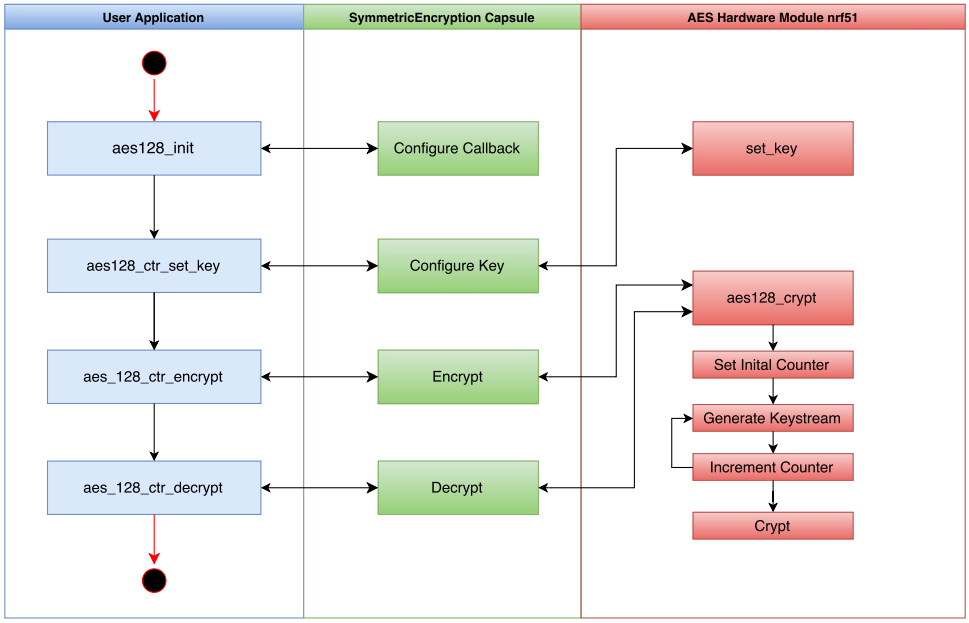

Figure 5.2: Tock AES device driver implementation. The figure shows the entire AES-128 counter mode device driver. To the left, blue boxes are different parts of the user space application which are device driver calls that are asynchronous i.e., return once the Tock kernel has handled the call. In the middle, green boxes are untrusted kernel modules i.e., the capsule and the HIL. For simplicity reasons, all system call implementations are referred to as "System Calls" which enables configuring of a key and to encrypt or decrypt a payload. Note, that the modules in the HIL only defines the interface and actual implementation elsewhere. To the right, red boxes are trusted kernel modules that configure the AES chip.

The library provides three functions which wrap the system calls command, allow and subscribe:

- *aes128_init(callback)* wraps a subscribe call which configures the callback function to be used by kernel once a task has finished, e.g., an encryption operation has finished.
- *aes128_configure_key(key)* wraps two calls: allow and command. The allow call shares the key between the kernel and the user application. To configure the key, in this case, to set it in the hardware, the function invokes the command call.
- *aes128_{encrypt,decrypt}_ctr(plaintext, initial counter)* wraps three system calls, one allow to share the plain/ciphertext, another allow to share the initial counter and lastly a command call to start the encryption or decryption.

Even though the library provides functions for both encryption and decryption, they are equivalent, i.e., there is no difference between them. Encrypting a text twice with the same initial counter value yields the plaintext with AES-CTR. However, the code is easier to read if the function name signales the intended purpose.

5.3.2 User application

The user application is relatively simple and its purpose is to demonstrate the device driver and to verify that the counter mode implementation is correct. It utilizes the AES-128 library, further described in Section 5.3.1, to initialize AES, configure a key, encrypt some data and decrypt the encrypted data. The blue marked parts of Figure 5.2 illustrate the implementation of the user application. As encryption and decryption can sometimes be lengthy processes, the user application is event driven. Once an event occurs, which in this case is the completion of a task (the blue box illustrates the tasks), the kernel calls the assigned callback function. Dependent on the data in the callback, the application will either continue or upon completion, exit. For example, after the user application invokes *aes_128_ctr_set_key* it goes to sleep and cannot continue with *aes128_ctr_encrypt* until it receives a callback. This implementation enables other tasks to execute while encrypting or decrypting or even put the processor to sleep which can save power. A test, specified by NIST [28], verifies that the driver is correct. It encrypts and decrypts 64 bytes, where given the key, initial counter and a plaintext should produce a known ciphertext to pass the test.

5.3.3 Capsule

The SymmetricEncryption capsule builds on the example capsule and handles configuration of the driver, encryption and resource allocation. It handles two system calls to configure the driver. One subscribe call for setting the callback function that the driver calls upon task completion. The other sets the encryption and decryption key via the allow call. For each encryption, two allow calls share the initial counter and the plain/ciphertext with the capsule respectively. The capsule keeps track of the current state of the driver with four states: not configured, encrypting, decrypting and idle. With these states, the capsule can manage resources and verify the status of the driver. Both the hardware module and the capsule utilizes the state but only the capsule updates it.

5.3.4 Hardware module

This module manages the interactions with the encryption hardware. In Figure 5.2, the red parts illustrate the overall functionality of the AES hardware module. The module exposes two functions via the HIL to the capsules, one to set the key and one to invoke the encryption/decryption. To set the key is quite simple, just write it to the register. The second function is a wrapper for all the other configurations and invocations of encryption and decryption. In terms of configuration, it initializes all associated data, keeps track of the state of the hardware module and configures the initial counter. The function also generates the keystream and XORs the keystream with the input data. In order to encrypt or decrypt data on the nRF51, the hardware module must configure the AES-ECB registers to point to the beginning of 48 continues bytes in RAM. The first 16 bytes correspond the key, the second 16 bytes correspond to the plaintext and the last 16 bytes corresponds the ciphertext. The AES-ECB module supports direct memory access (DMA) to read and write into

RAM without an involvement of the CPU once it is enabled. The DMA together with an interrupt based implementation makes the module nonblocking, i.e., the kernel can perform other tasks while the module is encrypting or decrypting.

We present the sequence of the high level aspects during an encryption operation. First, all data needed for the AES hardware module is initialized. This is required for every new encryption/decryption operation. The struct in the hardware module encapsulates all this data. Two types of containers, Cell or TakeCell, store the data in the struct and provide interior mutability. Listing 5.5 shows all this data stored in the struct. The data objects in the struct have the following responsibilities:

- regs - pointer to AES module in RAM
- client - object to call SymmetricEncryption capsules
- ctr - 16 bytes containing the counter value
- input - data to be encrypted or decrypted
- keystream - data containing the keystream
- remaining - value indicating how many more bytes of keystream to generate
- len - value indicating the length of input data to be encrypted or decrypted
- offset - cursor to access in correct place in the arrays.

Listing 5.5: AES Hardware Module Struct

```
pub struct AesECB {
    regs: *mut AESECB_REGS,
    client: Cell<Option<&'static Client>>,
    ctr: Cell<[u8; 16]>,
    // input either plaintext or ciphertext to be encrypted or decrypted
    input: TakeCell<'static, [u8]>,
    // keystream to be XOR:ed with the input
    keystream: Cell<[u8; 128]>,
    remaining: Cell<usize>,
    len: Cell<usize>,
    offset: Cell<usize>,
}
```

Second, the function enables the AES module and starts the generation of the keystream. After each generation of the keystream, the module updates the state and increments the counter. In the case where the module did not generate all bytes for the keystream, i.e., the keystream is smaller than the input, the process continues until the generated keystream is as big as the input data. Since the keystream must be XOR:ed with the input data (byte by byte), it must be as long as the input data. For instance, if the input data is longer than 16 bytes more than one AES operation has to be performed to generate the entire keystream. Once the keystream contains as many bytes as the input data it is finished. Finally, we XOR the input data and the keystream with each other and send it back to the capsule.

5.3.5 Discussion

The implementation of this device driver differs from the original design principle where the hardware module should contain as little logic as possible. The biggest reason for this is that the *SymmetricEncryption* capsule should support both software and hardware encryption. Rather than creating a helper capsule, containing logic specific to the hardware module, some of the logic is therefore put in the hardware module.

There are two types of hardware AES-128 modes available on the nrf51, AES128-ECB, and AES128-CCM. The AES128-CCM manages the encryption/decryption of BLE packets and this task has the highest priority, i.e., a task not connected to the radio module might be aborted while encrypting/decrypting with the AES128-CCM module. Technically, "regular" encryption/decryption can use the AES-128-CCM, but with the risk of the radio aborting the process and/or creating race conditions. To reduce those risks, our encryption AES128-CTR builds on top of the AES128-ECB in software. However, both AES128-CCM and AES-ECB use the same underlying AES hardware module but the chip ensures mutual exclusion.

In the current implementation, we generate the entire keystream at once and then XOR the input with the keystream byte by byte. As the hardware generates 16 bytes at a time and the input might not be a multiple of 16, the keystream can be longer than the input. In those cases, we only XOR the keystream corresponding to the input, i.e., we ignore some bytes. A more intuitive solution might be to allocate a buffer with the total length of the input data. Then, for each 16 bytes of generated keystream, we XOR:ed it with the corresponding part of the input data and the result of the XOR operation is added/appended to the buffer. This process repeats until all the keystreams are generated and XOR:ed. The problem with this solution is that the capsule requires a static mutable array for the final output. All operations on static mutable arrays are unsafe in Rust. Our solution relies on the TakeCell data type. Operations on the TakeCell are safe but it only enables a one-time operation, i.e., each read or write operation consumes the TakeCell. Since only one operation is possible, we generate and XOR everything at once. This implementation reduces the number of unsafe blocks.

5.4 TRNG

This section presents the implementation of the TRNG device driver in Tock for nRF51 that builds on top of the design presented in Section 4.4.3. The device driver consists of three components: user application, user library, capsule, and hardware module. However, this device driver already exists for other platforms, i.e., the user library, and capsule are available, but the hardware module is missing for nRF51. As a result, this section only covers the implementation of the hardware module, which maps directly to the already available capsule. Figure 5.3 illustrates the complete device driver. The yellow parts are the already existing parts of the device driver, while the red ones illustrate our contribution to the nRF51 platform.

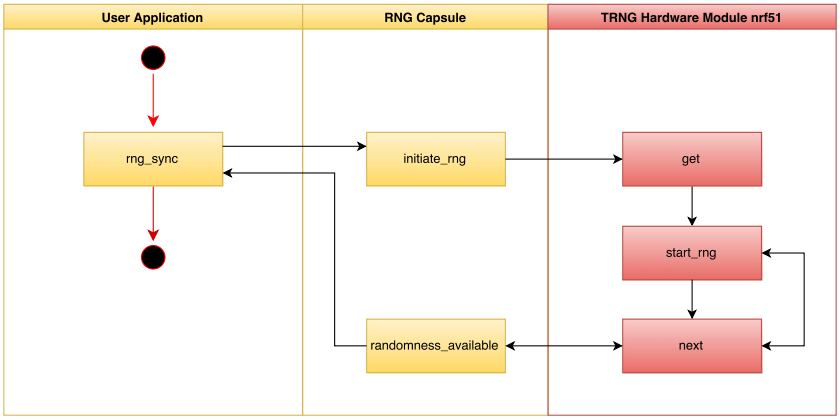

Figure 5.3: Tock TRNG device driver implementation. The figure shows the entire TRNG device driver. To the left and in the middle, yellow boxes are existing parts of Tock i.e., not implemented by us. To the right, red boxes are modules that directly configures the TRNG chip for nRF51 which have been implemented a part of this thesis.

5.4.1 Hardware module

This module interacts with the hardware to generate random numbers. Figure 5.3 presents the design of the driver and the red section visualizes the hardware module. The hardware module exposes one function to the HIL/capsule named *get*. This function starts the generation of random numbers. For each 4 bytes of random numbers, the hardware module notifies the capsule via a callback function. In the case where the user application requests more than 4 bytes of random numbers from the capsule, the capsule will instruct the hardware module to provide 4 more bytes, until the hardware module has generated all the requested bytes.

5.4.2 Discussion

As this device driver already exists, we have follow the "requirements" of the design. What is troublesome is that the design has a constraint to the hardware. For each generation of bytes, it requires 4 bytes to be generated from the hardware module. However, nRF51 can only generate 1 byte at a time and this forces us to put unnecessary logic in the hardware module. To continue the generation of additional bytes, we invoke a new call to generate additional bytes within the interrupt handler until we reach 4 bytes. Internal calls to member functions within the interrupt handler generates new stack frames, which should be limited in embedded systems.

5.5 Temperature sensor

This section presents the implementation of the temperature sensor device driver in Tock for nrf51 that builds on the design presented in Section 4.4.4. The device driver consists mainly of four different modules within Tock: temperature sensor library, temperature sensor capsule, temperature sensor HIL and temperature sensor hardware module for nRF51.

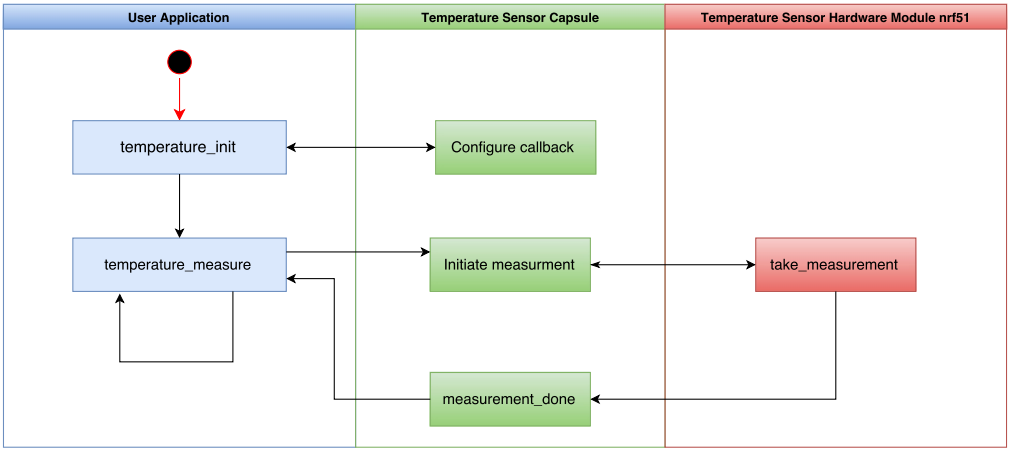

Figure 5.4: Tock temperature sensor device driver implementation The figure shows the entire temperature sensor device driver. To the left, blue boxes are the parts of the user applications which initiate the temperature sensor device driver and then runs an infinite loop to measure the temperature one time per second. In the middle, green boxes are the temperature capsule and the temperature HIL. Note, that the HIL itself do not implement any modules, it only defines an interface. To the right, the red box configures and starts the temperature sensor.

5.5.1 User application

The user application is a sample application with the purpose to demonstrate and to verify that the temperature sensor device driver works. The blue parts of Figure 5.4 illustrate the user application. It utilizes the temperature sensor library to initialize the temperature sensor and to perform temperature measurements. To read from the sensor, the driver requires a one-time initialization. After the initialization, the user application can read an arbitrary number of temperatures. The kernel signals that the reading is complete by firing a callback. This sample application reads from the temperature sensor and sleeps for one second. It also prints the value to the serial interface in the callback routine. This process repeats in an infinite loop until the system is shut down.

5.5.2 Library

The library provides two functions:

- *temperature_init(callback)* wraps a subscribe call which sets a callback function that the kernel calls when a new temperature reading is available.
- *temperature_measure()* wraps a command call that initiates the temperature measurement.

5.5.3 Capsule

The responsibility of the capsule is to provide the user application with temperature measurements. To acquire the measurements, the capsule must interact with the

hardware module. Once a temperature reading is available, the capsule sends this to the user application. Only one user application can request readings from the temperature sensor at the same time, which the capsule ensures. The capsule is implemented on top of the example capsule. Figure 5.4 illustrates the driver and the green parts mark the capsule.

5.5.4 Hardware module

The hardware module manages all interactions with the hardware, i.e., the temperature sensor. Figure 5.4 illustrates the hardware module in red. To enable the measurements, the hardware module initiates and starts the temperature sensor on the chip. Once a reading is available, the chip triggers an interrupt. The driver provides one sample per measurement. It is up to the user application to request more values to measure over time or to simply get more samples.

6
Evaluation

In this chapter, we present the evaluation of the device drivers in Tock compared to the current state-of-the-art embedded operating system and discuss the results. The evaluation creates the base to answer the following research questions:

- Is Rust suitable for programming device drivers for embedded operating systems?
- How does an embedded operating system implemented in Rust compare to current state-of-the-art implementations in C and C++?

As Tock aims to be more secure and reliable than the current state of the art embedded operating systems, we evaluate what the cost for this is in terms of power consumption and performance. Further, the chapter is divided into three sections, in the first Section 6.1, we present the evaluation framework that includes methods on how to evaluate power consumption and performance which include both software and hardware configurations. Second, in Section 6.2 we present and discuss the measurement results from the evaluation framework. Third and finally, in Section 6.3 we zoom out and reflect on the bigger picture regarding Tock, Rust, and other relevant topics.

6.1 Evaluation setup

We compare our device drivers in Tock with similar implementations in three other state-of-the-art embedded operating systems: ARM mbed [30], Apache Mynewt [31] and Zephyr [32]. These are embedded operating systems with the following properties:

- open source
- BLE support for nRF51
- AES-128 counter-mode support
- implemented in C/C++
- favor security and reliability
- support for many popular IoT processors and development kits

Table 6.1 shows the versions of the evaluated embedded operating systems. The rest of this section describes the different frameworks and approaches for the evaluation.

Embedded operating system	Kernel version
Apache Mynewt	1.0.0
mbed	5.4.0
Tock	d215e792
Zephyr	1.6.0

Table 6.1: Software versions

6.1.1 Power consumption

Power consumption in IoT devices is very important due to the nature of the devices, i.e., they should consume little power to be able to run on battery for a very long time, thus an evaluation of the power consumption is essential. nRF51-DK provides two pins for measuring the current power consumption of the processor. The pins are located after the voltage regulator to enable accurate results. The board requires a few small modifications to enable the measurement. By cutting a soldering connection, current will flow through the measuring pins instead of via the previous soldering connection. A 10-ohm's resistor between these pins finalizes the modification which enables measuring the voltage drop over the resistor. The voltage drop by itself is not the standard unit for measuring the power consumption. A more standardized unit is watt; hence we convert the voltage drop into watt with the following formula:

$$P = (\frac{U_{voltagedrop}}{R_{10ohm}}) \quad * \quad U_{supplyvoltage} \tag{6.1}$$

Measuring with good precision is only possible when running on the battery, rather than powered by USB [33]. Therefore, power consumption measurements are only performed when a battery powers the nRF51-DK. We mainly want to measure the power consumption during a specified period, i.e., to both measure the average power consumption and the instantaneous power consumption. To do this, we use two external tools, the Saleae Logic - 8-Channel USB Logic Analyzer[1] and an oscilloscope. Both tools provide a sampling rate over 1 kHz which provides enough measurement data to calculate the average power consumption.

6.1.1.1 BLE

This section presents the user application and the configurations used during the evaluation. As wireless communication is an important part of IoT devices, the evaluation focuses on the BLE driver to measure and compare the power consumption. All the operating systems in the evaluation have support for sending periodic BLE advertisement packets.

Since the radio consumes significantly more power in contrast to other peripheral devices and operations on the processor, the configurations have to be the same or similar across all evaluated operating systems. Four settings are important for

[1] http://downloads.saleae.com/Saleae+Users+Guide.pdf

power consumption in BLE: the advertisement interval, the transmission size, the transmission power, and the advertisement type [34]. The advertisement interval is the time between transmission. This property specifies how many times the radio should be active throughout a given time interval and thus it is important for the power consumption of the system. An advertisement interval can be set to between 10 milliseconds and 10 seconds. With a 10 milliseconds interval, the radio consumes a lot of power but enhances performance in applications such as localization. On the other hand, 10 seconds reduces the power consumption but would not be good for localization or pairing. For this evaluation, all systems have a similar advertisement interval of approximately 150 milliseconds.

The size of the packet to transmit affects the time the radio must be active to finish each transmission. A larger packet size requires the radio to be on for a longer time compared to smaller packets. 22 bytes constitute the payload size of the PDU type for all tests and transmission on all three channels take approximately 2 milliseconds.

Transmission power manages the range of the signal. A higher transmission power increases the range that the radio can reach but increases power consumption. nRF51 has eight settings for transmission power with a range from -30 dBm to +4 dBm. The current consumption varies between 5 mA to 15 mA between the highest and lowest transmission power [35]. In most of the evaluated systems, 0 dBm have been default, hence all systems use this value.

The last parameter which influences the power consumption is the advertisement type. An advertisement can either be connectable or non-connectable. Connectable advertisements, even though they are not connected to, activates the radio in receiving mode to listen for incoming connections. This means that the radio will be active for a longer time compared to the non-connectable advertisement that never activates the radio in receiving mode. Finally, all device drivers send advertisements every 150 milliseconds during a 10 seconds' interval.

Throughout the measurement, the power consumption between each advertisement, i.e., idle mode, will play a significant role in the average power consumption. To compare the different parts of the power consumption, we evaluate them separately. Idle power consumption is the power consumption between advertisement cycles where we assume that the CPU does not execute anything important. To get the idle data, we manually select it with the logic analyzer. Since we do not know when the idle mode stops and the radio starts, we separate the idle power consumption and radio power consumption from the total power consumption. By separating the measurements, we can argue about the results and suggest further improvements.

6.1.2 Performance

There is no built-in setting or tool to measure performance on nRF51-DK. By performance, we refer to the time it takes for an operation to complete. Evaluating performance requires a more hands-on approach compared to how we measure power consumption. The GPIO pins on the development board provide a simple way to measure performance with good precision. By setting a pin high when an operation starts and low when it stops, it is possible to calculate the time between the high and low voltage with, for example, an oscilloscope. This enables measuring and comparing different implementations.

6.1.2.1 AES128 counter mode

To evaluate the performance of the AES-128 counter mode device driver in Tock we compare it to other AES-128 counter mode implementations in terms of execution speed by encrypting data of different sizes. The different implementations are Apache Mynewt, ARM mbed, and Zephyr. All the evaluated embedded operating systems rely on encryption in software, except Tock. The software encryption in the other operating systems utilizes two different libraries: mbed TLS [36] and TinyCrypt [37]. Table 6.2 shows the methods for encryption for the different operating systems.

Embedded OS	AES128 counter mode options
Apache Mynewt	mbed TLS
mbed	mbed TLS
Tock	hardware
Zephyr	mbed TLS TinyCrypt

Table 6.2: Comparison AES-128 counter mode

The evaluation compares software and hardware implementations of AES128 counter mode which are two very different approaches to perform encryption. The evaluation is still interesting as we evaluate the following aspects:

- hardware encryption with counter mode in software compared to its counterpart in software
- differences between different software implementations
- the performance of the different operating systems

The advantage of a software implementation is that it does not require any system calls and context switches. In the hardware implementation, the hardware must be provided with the input and it must be transferred or shared via one or more system calls from the user application to the kernel. However, the software implementation still needs to send the data to the kernel for further process, e.g., transmitting the data via radio, which a hardware implementation does not need since the data is

already accessible to the kernel. Additionally, a disadvantage with encryption in hardware is that the implementation must provide functionality to encrypt more bytes than the hardware can encrypt at once. In an event-driven embedded operating system, a hardware implementation is also more attractive because it does not block the entire kernel. In other words, once data is being encrypted, other tasks can execute until the encryption task is finished.

In order to evaluate all the different configurations for the different operating systems, we use six different user applications. The basic functionality of the user applications are to set a GPIO pin high, encrypt data and set the GPIO pin low after the encryption. The time between the high and low GPIO pin, which we measure with an oscilloscope, determines the execution speed. We conduct three measurements for each of the operating systems with different sizes: 64 bytes, 512 bytes, and 1024 bytes. Each measurement consists of 10 encryption rounds and we use the average time of these encryptions as the results. We did not include the initialization of AES and configuring of the key because in Tock, it only introduces a system call overhead and in the other operating systems it is only function call overhead which is much smaller.

6.1.2.2 System call overhead

To evaluate the different operating systems without any external peripherals or operations, we measure the system call overhead. The time it takes for a system call to return after the user application invokes it determines the overhead. We argue that measuring the system call overhead is a good method to determine the overhead of the operating system since this is how fast the kernel can switch between user space and kernel space, and in other words, react to requests and system calls. The overhead affects tasks such as scheduling and context switching. To measure the overhead, we use one of the simplest drivers: the GPIO driver. By using a simple driver, we do not rely on any external process which could make the results less consistent. Embedded systems are generally I/O bound which means that the CPU spends more time to wait for I/O than performing computational heavy tasks. Nevertheless, we argue that this metric is good to compare the efficiency of the different operating systems.

We introduce four different user applications in each operating system respectively which toggle a GPIO pin 10,000 times and then we measure the execution time. A division of the results with 10,002 (including two additional operations for the measurement) returns the average execution time of a system call. Toggling of a pin in some operating systems can require more system calls since it might require a read call to get the current status from the GPIO. Therefore, in all user applications, we implement our own toggle function by using the underlying function to set the GPIO pin low and high. A boolean variable determines whether to set the GPIO pin high or low.

6.2 Results

This section presents and discusses the results of the measurements. We discuss and compare the device drivers in Tock to the state-of-the device drivers in Apache Mynewt, ARM mbed, and Zephyr in terms of power consumption and performance.

6.2.1 BLE power consumption

As described in Section 6.1.1.1 we measure the power consumption during 10 seconds of BLE advertisements. Each advertisement consists of three transmissions where Figure 6.1 visualizes one advertisement cycle. Furthermore, we divide the results into three different parts where all of them show the average power consumption: the total power consumption (both idle mode and during transmission), idle and radio transmission.

Figure 6.2 shows the results of the three aforementioned parts during approximately 66 BLE advertisements. Mynewt has the highest total average power consumption and the difference between Mynewt and Tock is about 0.5 mW. mbed consumes almost half as much as the other operating systems. During transmission, i.e., when the radio is active, the average power consumption is highest in Tock and lowest in mbed. In idle-mode, mbed consumes significantly less, a factor of approximately 10, than the other operating system. Mynewt consumes most in idle and approximately 0.5 mW more than Tock and Zephyr.

6.2.1.1 Discussion

During the measurement of the operating systems, we found negative values in the data set. We argue that this is because the power consumption is sometimes so low that current in capacitors get discharged and "returns". Approximately 80 % of the measured values in mbed are negative, whereas the rest of the measurements in the other operating systems contain less than 1 % negative values. When processing the results, we set all negative values to 0 and we argue that this is the fairest approach since both including them and removing them would affect the results in a strange way.

The results show that mbed is far superior to the other systems both in terms of average power consumption during advertisements and in idle mode. What is interesting is that mbed has an extremely low power consumption while in idle. This affects the results heavily since the time it takes to send a packet is very short compared to the time that the operating system is idle, hence the idle power consumption has a big impact on the results. The large power consumption in idle mode for Mynewt can be the result of keeping the radio on for a longer time than necessary and not powering it off. Another explanation can be that the Mynewt keeps the UART active throughout the measurement. The UART provide real-time logging of the status of the BLE stack and we did not manage to disable this feature.

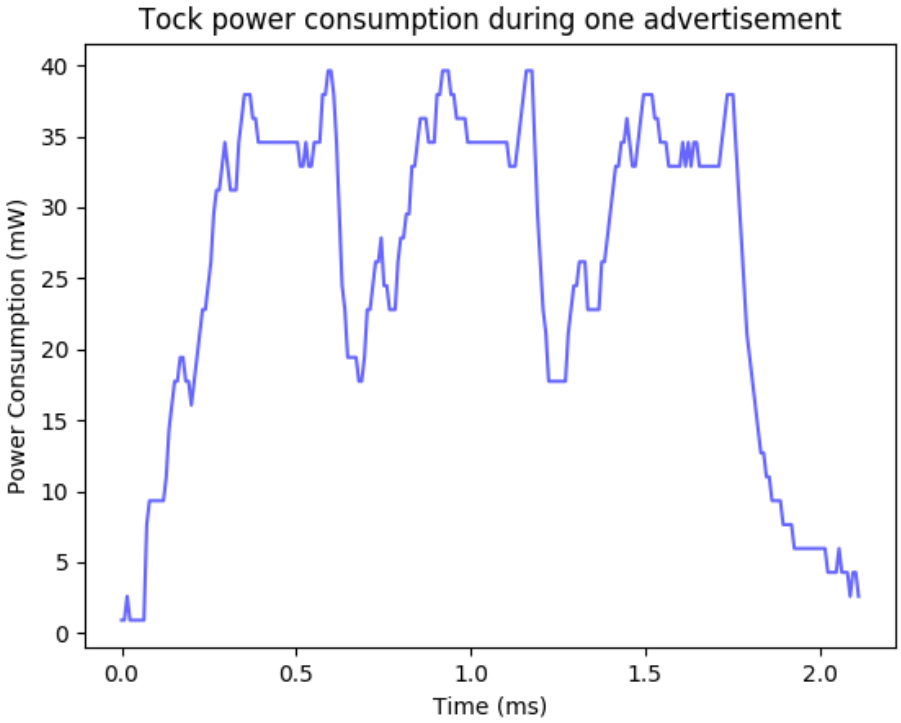

Figure 6.1: Tock power consumption during one advertisement. The figure shows the characteristics for the power consumption for one advertisement. During transmission, the power consumption increases which the spikes symbolize. The three spikes show the transmissions on the three advertisement channels. The y-axis shows the power consumption in mW and the x-axis shows the time in milliseconds.

We also see that the power consumption while transmitting advertisements is significantly lower in mbed. Mynewt, Tock and zephyr have approximately the same power consumption and the consummations only differ in a few mW but Tock consumes the most.

From the results, we learn that it is important to configure the processor to go into deep sleep while not operating. It also shows that our concerns with the configuration of the radio and advertisement are important since this is most likely the reason for the high power consumption in Tock while advertising. We conclude that it is important to investigate the use case of the device and to configure it correctly to achieve optimal power consumption.

6.2.2 AES performance

Figure 6.3 shows the results of the execution time for AES128 counter mode in the different configurations in each operating system, further described in Table 6.2. mbed TLS is the fastest and the operating system has minor impact on the results. Hardware encryption in Tock is about 4 times slower than mbed TLS and TinyCrypt is about 10 times slower than mbed TLS.

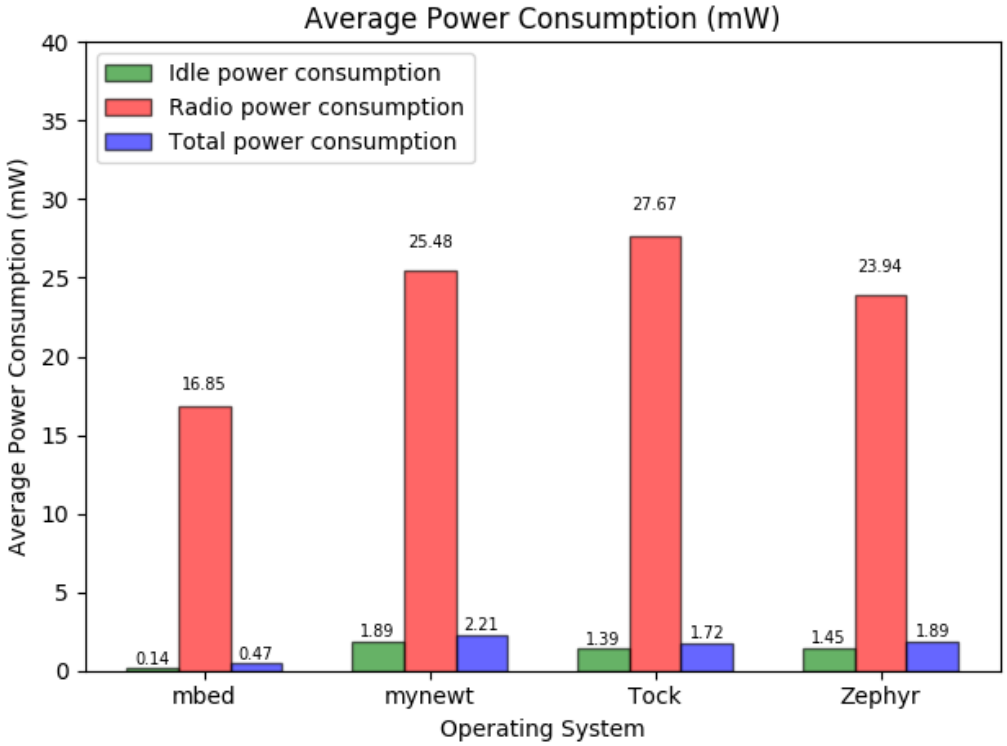

Figure 6.2: Embedded OS comparison BLE average power consumption. The y-axis shows the average power consumption in mW and the x-axis shows three different metrics of the average power consumption per operating system during BLE advertisements. The red bar shows the total average power consumption, the blue bar shows the average power consumption during the active time of the radio, and the green bar shows the average power consumption only during idle mode.

6.2.2.1 Discussion

The first result to highlight is the difference between AES in hardware and software. Surprisingly, the hardware implementation in Tock is slower than mbed TLS. However, it is not surprising that the hardware implementation is slower but that is four times slower is worth noticing.

The results of the comparisons of TinyCrypt and mbed TLS is quite interesting. TinyCrypt is about 10 times slower than mbed TLS despite both implementations rely purely on software without system calls and context switches. This shows that the implementation plays a much bigger role in terms of performance than the operating system itself. Since studies and implementations of cryptography have been extensively studied in the past, it is surprising that the implementations differ so much. The results also show that the execution speed of mbed TLS in mbed, Mynewt and Zephyr is more or less the same. This shows that the operating system plays a very little role when it comes to software implemented encryption. One reason for this is most likely due to that such tasks does not require any additional kernel involvement, e.g., system calls, because they execute in user space.

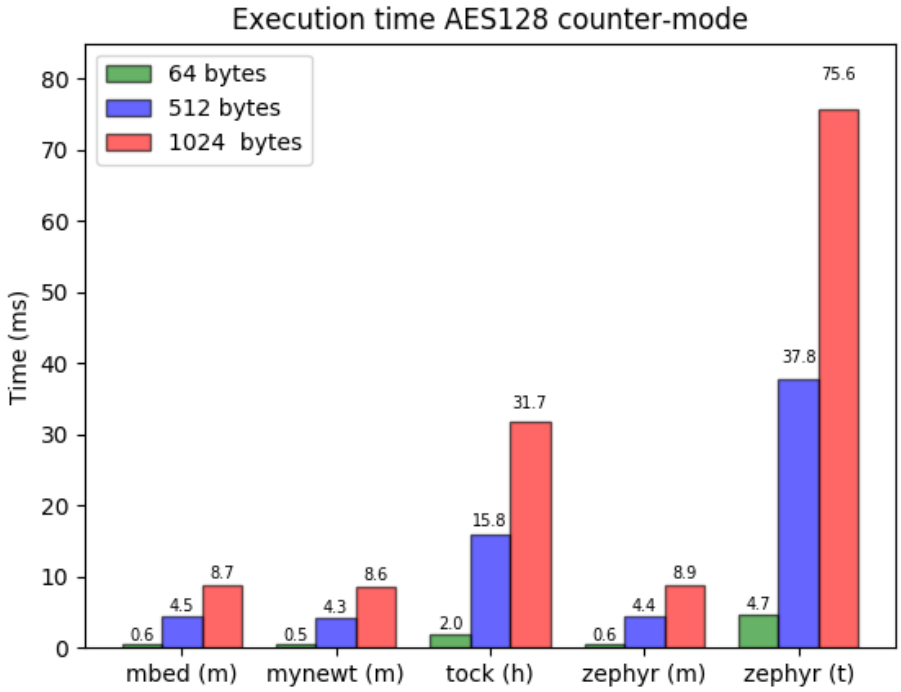

Figure 6.3: Embedded OS comparison execution speed AES-128 counter mode The y-axis is the execution time for AES-128 counter mode in milliseconds. The x-axis is the operating system and (m) is an acronym for mbed TLS, (t) is an acronym for TinyCrypt, and (h) is an acronym for hardware encryption in Tock. Furthermore, the green bar is 64 bytes payload size. The blue bar is 512 bytes payload size and the red bar is 1024 bytes payload size. The results show that mbed TLS library is the fastest, around 4 times faster than the hardware implementation in Tock and around 10 times faster than TinyCrypt. In the software implementations, the different operating systems have a minor impact on the execution speed.

To conclude, we discuss the advantages and disadvantages of hardware encryption in Tock. The advantages with hardware encryption are twofolded. First, it does not block the kernel from performing other tasks during encryption. This is especially attractive in event-based operating systems where tasks are executed until completion. Second, it allocates less memory because everything is performed in hardware. When it comes to disadvantages, they are twofolded. First, hardware encryption is not hardware independent and requires different implementations for different chips, which a software implementation does not need. This makes the implementation less portable and requires more maintenance during hardware upgrades. Second, hardware encryption is slower than software encryption.

We think that there are limited use cases for encrypting bigger payloads than 1024 bytes and even during such a large encryption, it only takes 8 milliseconds to complete. Therefore, it will only block an event-based operating system for 8 ms while encrypting 1024 bytes data which should be affordable in our opinion. Thus, we conclude that a software-based implementation for AES128 counter mode is superior to the hardware based implementation.

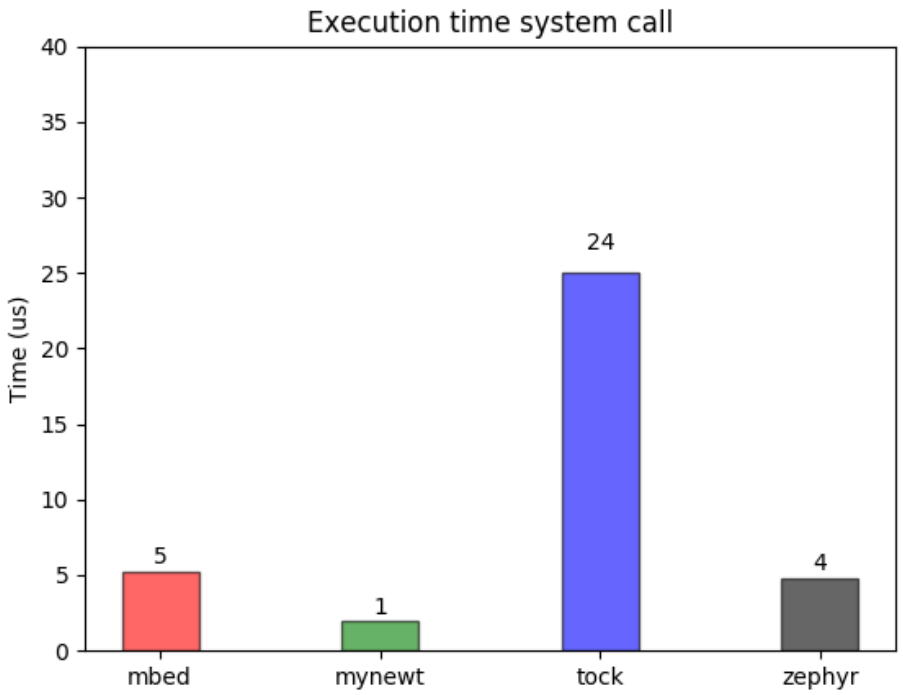

Figure 6.4: Embedded OS comparison average execution time of a system call. The y-axis is the execution time for one system call in μs and the x-axis is one for each of the operating systems. The results show that Mynewt is the fastest, mbed, and Zephyr are around 5 times slower and Tock is around 24 times slower. Tock stands out in this comparison and one reason is that the context switch assumes there exist an MPU which requires about 100 extra clock cycles.

However, we did not conduct any experiment regarding different hardware encryption. The case can be that the AES chip on nRF51 is slow compared to other processors with built-in encryption. We have not seen any numbers on how fast the AES chip of nRF51 is, but based on our estimation it is approximately 62.5 kB/s.

6.2.3 System call performance results

This section presents the average time to execute a system call. 10,000 system calls lay the base for the average calculation. Figure 6.4 shows the differences between the systems where Mynewt outperforms the other operating systems and Tock is the slowest by a big margin.

6.2.3.1 Discussion

Figure 6.4 shows a big difference between Tock and the other operating systems where Tock is approximately 5 times slower than the competitors. A conversion to clock cycles (1 μs corresponds to 16 clock cycles) shows that the fastest system call takes 16 clock cycles compared to Tock where a system call requires 384 clock cycles. The results are not surprising because Tock introduces several layers along with

that each user application is executed as its own separate process, which increases the overhead. In addition, Tock assumes that each processor has an MPU which introduce approximately 100 clock cycles in overhead in the context switch which corresponds to 6.25μs on nRF51. Note that the MPU of nRF51 is not supported by Tock because the MPU does not provide sufficient granularity, hence all MPU instructions are NOP's. Regardless whether an MPU is present or not, the context switch is significantly higher in Tock. Even though the differences are quite big in terms of percentage, the values are relatively small. In most applications, the system call overhead that Tock introduce would probably not be critical. 20 μs is a very short time and most often, the waiting for other peripheral devices or operations takes longer time which makes the system call in most cases neglectable.

6.3 Discussion

In this section, we discuss the experience by developing device drivers in Tock. The section is divided into two parts. First, we discuss our experience by learning and using Rust. Second, we discuss our experience by using Tock as an embedded operating system for developing device drivers.

6.3.1 Rust

As Rust is a relatively new programming language, it is not commonly used by students and "regular" developers. We belong to these categories and hence we had very little experience with Rust before starting this thesis. It has, however, been both an interesting and rewarding journey to learn Rust.

Rust enables safety guarantees by default, which increase development time for novice Rust developers. Safety guarantees include borrowing, lifetimes and ownership, and together they introduce a very steep learning curve. It can take some time to pass the learning curve and to become productive in Rust. One of the safety guarantees is the ownership paradigm which is very different in contrast to our earlier experiences. Passing variables by value to functions requires more thinking than in a "normal" programming language since Rust transfers ownership of the variable to the invoked function. The advantage of these strict rules is that once the Rust code compiles it executes reliable, i.e., crashes are rare, and thus, we have experienced very few runtime errors which is an attractive property for operating systems. The compiler is however helpful and informative; it gives hints about what is wrong and how to make it right. One example of such a compiler error is *"error borrow 'foo' as mutable more than once at a time"* and it tells the user that foo can only be borrowed mutable once at the time which is helpful.

Although Rust has many positive sides, changing or learning a new programming language overnight is not an easy task, especially not when programming embedded devices which are generally more complex than general applications. The community of Rust often claims that the learning curve is quite steep. It takes some time be proficient and to work with the compiler rather than against it, but once achieved,

it is very rewarding. Most functionality of C exists in Rust; hence it can be an alternative to use Rust instead of C. Rust will not replace C overnight but if it gains traction and popularity it is fully capable of doing so. A transition from C to Rust would increase the reliability and security of many applications and simplify development in many cases.

6.3.1.1 Unsafe Rust

To be a fully compatible system programming language, unsafe Rust plays an important role since for example, it enables direct modification of registers in a processor. By separating the safe and unsafe code with blocks it becomes very clear where the code can crash and it helps developers to understand why a given operation is safe or unsafe. The compiler also helps since it will prevent code that is unsafe and warn about code that is safe but is still within an unsafe code block. This can decrease the number of unsafe code blocks, which is good since ideally, Rust code should contain as little unsafe code as possible to be reliable and secure.

6.3.1.2 State machines

By combining the enums, structs, and traits, one can build finite state machines in Rust. The compiler can then verify the state transitions at compile time. For example, consider a state machine with three states: Off, On and Sending. Assume it has two valid state transitions: Off $\rightarrow$ On and On $\rightarrow$ Sending. All other state transitions such as Off $\rightarrow$ Sending are not valid. By conforming different types in Rust the compiler can then verify whether a state transition is valid or not. An invalid state transition would result in a compiler error. In such a simple state machine a compile time varification may not be particularly useful. However, in more complex state machines, for example as in the BLE, protocol such functionality is attractive. Verification of the state machine at compile time would catch faulty state transitions early on and decrease time spent on debugging. This feature introduces no runtime overhead since the compiler manages everything.

An equivalent solution in C or C++ would require runtime checks to determine whether a state transition is valid or not. Another way is to, way simple but yet common, put inline comments that warn the user that the radio must be powered on before configuration.

6.3.2 Tock

Tock is expanding rapidly with both active maintainers and community. In the beginning of the project, a lot of documentation was missing and changes in the build system increased the difficulty of the already complex task to extend an embedded operating system with device drivers. We have spent time to research how to debug and patch existing debugging tools. However, the amount of documentation and tutorials is increasing, which can help new developers to get into Tock much faster and easier. Once one pass the first big hurdles, Tock becomes quite easy to understand and to extend.

For a relatively young and small project, Tock performs very well or on par with the state-of-the-art embedded operating systems according to our benchmarks. However, these results are just benchmarks, for many applications and developers the availability of support, high reliability and security are probably worth more than the execution speed in many cases.

6.3.2.1 Tock architecture

The architecture of Tock is probably one of the most promising and interesting parts with regard to how it takes advantage of the safety features of Rust. With the separation of capsules and hardware modules, the system becomes both easier to maintain and less prone to crashes. Rust provides safety to the capsules with no runtime overhead, i.e., capsules cannot crash due to memory corruption, since they do not use unsafe Rust. In a system without these features, the same level of safety would require the "capsules" to run in separate processes and communicate via IPC in order to achieve the same level fault isolation but with a higher runtime overhead and memory requirement. The increased memory usage would probably be too big for most embedded systems, especially if all device drivers were statically allocated. Another architectural feature is that user space applications run in separate processes protected by an MPU. In a sense, the MPU is a lightweight MMU that prevents user space applications from operating in other memory than its dedicated memory and this protects other applications as well as the kernel from a malicious or faulty user application. Thus, a malicious or faulty user space application can never crash the Tock kernel. We argue that this design is very attractive when it comes to reliability and security.

In comparison to the current design of state-of-the-art embedded operating systems, we argue that in theory, they are not as reliable and secure as Tock. This is mainly due to two reasons: 1) they are implemented in a memory unsafe language such as C/C++ 2) many of them have no memory corruption protection at all because the kernel and user space share the same stack and thus a buffer overflow can crash the entire kernel. However, this does mean that the current state-of-the-art is insecure by default. The implementation itself is probably more important and they are probably implemented with defensive programming principles in C/C++.

A nice aspect of the safety features of Rust is that one can be more fearless. The developer does not need to worry about the code crashing nor to have null checks all over the code.

Tock naturally contains unsafe Rust but compared to the current state-of-the-art embedded operating systems it contains significantly less unsafe code segments. C or C++ code is equivalent to unsafe Rust, hence all code in the current state-of-the-art embedded systems is unsafe. Tock limits unsafe Rust to small parts of the core of the kernel and there are even flags in the compiler to refuse the use of unsafe code outside these predefined regions. This makes it far more obvious where Tock can crash.

6.3.2.2 Resource allocation

As Tock does not support dynamic allocation, resources are statically allocated where the only exception is for user space applications which Tock can load at runtime. They run outside the kernel with restricted privileges but can dynamically allocate resources. Relying on the static allocation of resources have both advantages and disadvantages. One advantage is that the compiler analyzes and verifies all code in Tock to increase reliability, security, and performance by compiler optimization. We argue that these properties are essential to provide reliable and secure kernel modules because dynamic loading means that these can have been written in C, C++, assembly or similar. Moreover, Swift et al. [38], also report that loadable extensions are the leading cause of operating system failures. We see this as an indication that static allocation is favourable for reliability. SIPs in Singularity further described 3.1, shares the design where it does not allow self-modification of the code and dynamic loading.

On the other hand, one disadvantage is that Tock statically allocates all capsules and hardware modules at compile time, and deallocation is not possible. The allocation also includes some internal buffers in the kernel. For example, to provide functionality to encrypt a payload of 1 kB, the kernel requires 1 kB of RAM extra. This is not ideal for embedded systems since the RAM is rather limited and when many device drivers require dedicated static buffers, the kernel will require more RAM. The allocations are however platform dependent, i.e., each platform requires individual allocation of capsules and hardware modules. That means that the allocation is only specific for each platform, e.g., nRF51-DK, and thus, Tock does not allocate device drivers that are not available for the chosen platform. As an example, it is not possible for an application to only allocate the BLE device drivers and nothing else without recompiling the Tock kernel. This design requires more memory since Tock allocates all modules independently regardless whether they are needed or not. This can turn into a problem if the number of capsules increases, then the processor might not have enough memory to load them all. In the end, this would result in customization of the kernel to meet different requirements.

6.3.2.3 Debugging

Debugging in Tock has been a burdensome experience compared to the current state-of-the-art embedded operating systems. At first, there were no debugging symbols available in the Rust cross compiler. Debugging without debugging symbols is not very easy. In later versions, however, they are now available. Despite that debugging symbols now are available, nRF51-DK has too little memory for them anyway. This implies that gdb only provides ARM assembly, which at least for the authors of this thesis has been a difficult task to benefit from. Moreover, to debug the code, one must put annotations to prevent compiler optimizations and to ensure that the linker keeps symbolic names of functions in order to be able to assign breakpoints to them.

There are now great tutorials on the Tock homepage explaining this in more detail. Throughout the project, debugging by inserting panics have been the most used debugging technique. This has been quite time consuming but now there is a debug macro which can print to the serial interface which greatly improved the debugging.

6.3.2.4 Implementation

Embedded systems are mostly event-driven and can end up in a scenario where multiple mutable references to an object are needed. The Rust compiler prevents such scenarios because race conditions can occur in multi-threaded environments. As Tock is single-threaded these checks are unnecessary for safety, which means that the compiler rejects valid single-threaded programs. Therefore, a workaround for this in Tock is the TakeCell data structure which prevents the compiler to check for race conditions. The TakeCell can only be consumed once, after that it becomes None. An attempt to consume a TakeCell which is None will generate a runtime error. However, we have used TakeCell extensively to be able to share the contents of buffers between user space and kernel space without many crashes. We think that the TakeCell forces us to be more careful in our implementations and to ensure that it is not None. In a sense, this is very similar to check whether a pointer is null in other unsafe languages. Most of the crashes we have encountered have been due to consumption of an empty TakeCell. This means that a bad implementation with the TakeCell can crash the kernel and reduce the reliability of Tock. We consider two approaches to fixing this problem: 1) replace the TakeCell with an empty data type internally in the TakeCell when it gets consumed 2) create a deep copy of the object instead of transferring ownership.

7
Conclusion

This chapter summarizes and concludes the thesis. Most of the conclusions are on a higher lever and with a broader perspective when compared to the discussions in the previous chapters.

In this thesis, we contribute to the open source community with designs and implementations of device drivers for Tock. The device drivers enable foremost connectivity via Bluetooth low energy and security primitives via AES128 encryption, but also random numbers via a true random generator and usage of a temperature sensor. BLE and AES128 drivers have a high value for embedded systems as both connectivity and security increase in popularity. We show that designing and developing device drivers for embedded systems in Rust work well with the safety features in Rust by the successful implementation of device drivers. This answers the first research question if Rust is suitable for programming device drivers for embedded operating systems and we argue that Rust is fully capable as a systems programming language for IoT. The evaluation shows that Tock is on par with the current state-of-the-art embedded operating systems when it comes to power consumption. In terms of performance there is an overhead but in many cases, embedded systems are I/O bound. We conclude that power consumption has a bigger role and impact and Tock is doing well according to such metrics. This answers research question two, how does an embedded operating system implemented in Rust compare to current state-of-the-art implementations in C?

We argue that Tock is a good platform for IoT devices and a worthy contribution to the embedded operating systems' family. Many applications would benefit from running a more reliable and secure embedded operating system and with the small overhead of Tock, there is no reason not to use it. For non-kernel development, Tock is easy to use out of the box. Source code for existing applications is available which can be modified to fit a specific use case. A regular developer, i.e., an application developer with very little systems programming knowledge, can set up and run applications relatively easy. The possibility to write user applications in C enables an easy transition from other embedded operating systems while increasing the reliability and security with the Tock kernel. As Tock now supports user applications in Rust, even the user applications can benefit from the security features of Rust.

We have encountered very little problems with the design and implementation of Tock as an operating system. Most components, e.g., system calls and hardware abstractions, are available as one might expect from an embedded operating system. The most troublesome aspects are to share data between user space and kernel space. This includes accessing static arrays by reference in nested data types. For a novice Rust developer, the nested data types required can seem a bit obscure but it increases throughput as entire buffers do not need to be copied to the kernel.

7.1 The future of Tock

Tock is still missing some essential features to "compete" with current state-of-the-art embedded operating systems, such as a complete network stack and support for more hardware platforms. In our opinion, the highest priority in Tock should be a complete network stack in Rust. Tock now supports 802.15.4, which is a step in the right direction. It is, however, possible to run other network stacks outside Tock, but for reliability and security implementations purely in Rust is beneficial. Our contribution with the BLE driver is a start to provide more connectivity to Tock. Hopefully, more work to this driver can result in a more sophisticated BLE stack. The work includes direct connections, rather than just advertisements, security, i.e., authentication to connections and support more PDU types.

It would also be interesting to see support for real-time constraints to compete with RTOS's. Safety-critical applications would especially benefit from Rust and Tock since potential memory corruption issues can endanger human lives. By relying on memory safety, one can facilitate and simplify the development of safety-critical software. These are applications that require high reliability and more sophisticated development processes which the type-system in Rust can simplify. As these safety-critical applications are becoming connected, the threats against these applications increase. Therefore, we consider that Tock is a good candidate to use when reliability and security are essential. Support for real-time constrained applications can increase the attention to Tock since the need for reliable and secure systems is increasing, especially when the technology regarding autonomous cars progresses. To further increase the popularity of Tock (and in some cases Rust), it is also important that more tools (compilers and development environments) for embedded systems become available in Rust. To conclude, we think that Tock can compete with the current state-of-the-art embedded operating systems if development continues and more hardware gets supported.

7.2 Ethics and sustainability

In this thesis, we evaluate and implement safe device drivers in Rust and Tock for the Internet of Things. The increasing number of connected devices, makes safety, privacy, and reliability important, especially if big parts of our society rely on them. The fact that we monitor our lives to a larger extent, for example, our health and physical activities, exposes our privacy. With insecure implementations of IoT, hackers can exploit the implementations and gain an understanding of our lives. Insecure devices can also be used to attack companies or countries. Potentially, the concepts in this thesis can prevent accidents that endanger human life in safety-critical systems and also increase privacy by reducing the risk that malicious parties will get access to our private data.

Further, by evaluating the power consumption, we highlight that it is an important metric for sustainability with the increase of global warming in mind. Thus, manufacturers and customers need to think about energy consumption and not just performance when developing IoT devices.

Bibliography

[1] Ericsson. 26 billion connections 2020, June 2015. URL http://www.ericsson.com/res/docs/2015/ericsson-mobility-report-june-2015.pdf.

[2] Mouaaz Nahas and Adi Maaita. *Choosing Appropriate Programming Language to Implement Software for Real-Time Resource-Constrained Embedded Systems*. INTECH Open Access Publisher, 2012.

[3] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. Engineering the Servo Web Browser Engine Using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 81–89, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4205-6. doi: 10.1145/2889160.2889229. URL http://doi.acm.org/10.1145/2889160.2889229.

[4] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *USENIX Security Symposium*. San Francisco, 2011.

[5] The Rust Programming Language, 2017. URL https://www.rust-lang.org.

[6] Amit Levy, Michael P Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is Theft: Experiences Building an Embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, pages 21–26. ACM, 2015.

[7] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology. *Sensors*, 12(9):11734–11753, 2012.

[8] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An Empirical Study on the Impact of Static Typing on Software Maintainability. *Empirical Softw. Engg.*, 19(5):1335–1382, October 2014. ISSN 1382-3256. doi: 10.1007/s10664-013-9289-1. URL http://dx.doi.org/10.1007/s10664-013-9289-1.

[9] The Rust programming language. The Rust Programming Language, Second edition, 2017. URL https://doc.rust-lang.org/nightly/book/second-edition/.

[10] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating System Concepts*, volume 4. Addison-wesley Reading, 1998.

[11] Gene M Amdahl, Gerrit A Blaauw, and FP Brooks. Architecture of the IBM System/360. *IBM Journal of Research and Development*, 8(2):87–101, 1964.

[12] Wikipedia. Source Lines of Code, 2017. URL `https://en.wikipedia.org/wiki/Source_lines_of_code`.

[13] Benjamin Roch. Monolithic kernel vs. Microkernel. *TU Wien*, 2004.

[14] Apple. iOS 10 Security, March 2017. URL `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`.

[15] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522720. URL `http://doi.acm.org/10.1145/2517349.2522720`.

[16] J. Liedtke. On Micro-kernel Construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, December 1995. ISSN 0163-5980. doi: 10.1145/224057.224075. URL `http://doi.acm.org/10.1145/224057.224075`.

[17] Norman E. Fenton and Niclas Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software engineering*, 26(8):797–814, 2000.

[18] Tock. Tock is a Safe, Multitasking Operating System for Low-Power, Low-Memory Microcontrollers., 2017. URL `https://tockos.org`.

[19] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. TinyOS: An Operating System for Sensor Networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[20] Newlib, 2017. URL `https://sourceware.org/newlib`.

[21] Naresh Gupta. *Inside Bluetooth Low Energy*. Artech House, 2013.

[22] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007. ISSN 0163-5980. doi: 10.1145/1243418.1243424. URL `http://doi.acm.org/10.1145/1243418.1243424`.

[23] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, April 2006. ISSN 0163-5980. doi: 10.1145/1218063.1217953. URL `http://doi.acm.org/10.1145/1218063.1217953`.

[24] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Local Computer*

Networks, 2004. 29th Annual IEEE International Conference on, pages 455–462. IEEE, 2004.

[25] Contiki. Contiki: The Open Source OS for the Internet of Things, 2017. URL `http://www.contiki-os.org`.

[26] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly Media, Inc., 2001.

[27] NIST FIPS Pub. 197: Advanced Encryption Standard (AES). *Federal Information Processing Standards Publication*, 197(441):0311, 2001.

[28] M Dworkin. NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation, 2001.

[29] Phillip Rogaway. Evaluation of Some Blockcipher Modes of Operation. *Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan*, 2011.

[30] ARM. ARM mbed, 2017. URL `https://www.mbed.com/en`.

[31] Apache Software Foundation. Apache Mynewt: An OS to Build, Eeploy and Securely Manage, 2017. URL `https://mynewt.apache.org/about`.

[32] Linux Foundation. Zephyr Project, 2017. URL `https://www.zephyrproject.org/about`.

[33] Nordic Semiconductor. nRF51 Development Kit, User Guide v1.1, May 2016. URL `http://infocenter.nordicsemi.com/pdf/nRF51_DK_UG_v1.1.pdf`.

[34] Raphael Schrader, Thomas Ax, Christof Röhrig, and Claus Fühner. Advertising power consumption of Bluetooth Low Energy systems. In *Wireless Systems within the Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS), 2016 3rd International Symposium on*, pages 62–68. IEEE, 2016.

[35] Nordic Semiconductor. nRF51 Series Reference Manual Version 3.0.1, December 2016. URL `http://infocenter.nordicsemi.com/pdf/nRF51_RM_v3.0.1.pdf`.

[36] ARM. mbed TLS, 2017. URL `https://tls.mbed.org/`.

[37] Intel Open Source Technology Center. TinyCrypt, 2017. URL `https://01.org/tinycrypt`.

[38] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Trans. Comput. Syst.*, 23(1): 77–110, February 2005. ISSN 0734-2071. doi: 10.1145/1047915.1047919. URL `http://doi.acm.org/10.1145/1047915.1047919`.