CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Abstraction Layers and Energy Efficiency in TockOS, a Rust-based Runtime for the Internet of Things

Master's thesis in Computer Systems and Networks

FILIP NILSSON & SEBASTIAN LUND

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Abstraction Layers and Energy Efficiency in TockOS, a Rust-based Runtime for the Internet of Things

SEBASTIAN LUND

FILIP NILSSON

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Abstraction Layers and Energy Efficiency in TockOS, a Rust-based Runtime for the Internet of Things
SEBASTIAN LUND, FILIP NILSSON

Supervisor: Olaf Landsiedel, Computer Science and Engineering
Examiner: Philippas Tsigas, Computer Science and Engineering

Master's Thesis 2018
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 2018

Abstraction Layers and Energy Efficiency in TockOS, a Rust-based Runtime for the Internet of Things, FILIP NILSSON & SEBASTIAN LUND
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The advent of the Internet of Things (IoT) has led to an increasing number of connected devices with the need to run several applications concurrently. This calls for an operating system with a complete network stack, customized for embedded systems with the requirements to be up and running for very long periods of time.
In this thesis, we demonstrate how Tock, an operating system written in Rust, easily can be ported to a new hardware platform and provide similar results in terms of performance and energy-efficiency as other state-of-the-art operating systems for the IoT.
Our thesis revolves around the CC26xx family of microcontrollers from Texas Instruments. These microcontrollers provide a wide range of features for power management, such as peripheral clock management, and support for several different power modes. We show how software constructs can be used to facilitate the use of these power saving resources and decide what power mode to use depending on the workload.
Besides comparing Tock with its competitors, we document the process of working with Rust in an embedded setting and research if Tock manages to leverage the features of Rust to its advantage through an adequate abstraction level.

Keywords: Tock, TockOS, Rust, IoT, Bluetooth Low Energy, Embedded Operating System, Energy efficiency, Power modes

Acknowledgements

We would like to thank our supervisor, Olaf Landsiedel, for his support and guidance throughout the project. We also want to thank Amit Levy and the whole core team behind Tock for sharing their knowledge and reviewing our code. Lastly, we would like to thank Philippas Tsigas for being our examiner.

Filip Nilsson, Sebastian Lund, Gothenburg, June 2018

Contents

List of Figures **xiii**

List of Tables **xv**

Acronyms **1**

1 Introduction **3**
1.1 Context . . . 3
1.2 Goals and Contributions . . . 4
1.2.1 Problem statement . . . 4
1.2.2 Contributions . . . 4
1.3 Report structure . . . 4

2 Background **7**
2.1 Embedded Operating Systems . . . 7
2.2 Hardware . . . 8
2.2.1 UART . . . 8
2.2.2 I^2C . . . 8
2.2.3 Memory protection unit . . . 8
2.2.4 Bluetooth low energy . . . 9
2.3 Platforms . . . 9
2.3.1 Simplelink Sensortag . . . 9
2.3.2 Launchpads . . . 9
2.4 The Rust Programming Language . . . 10
2.4.1 Ownership . . . 10
2.4.2 Borrowing with references . . . 11
2.4.3 Lifetimes . . . 11
2.5 Tock OS . . . 12
2.5.1 Kernel . . . 12
2.5.2 Userland . . . 13
2.5.3 Scheduling . . . 13
2.5.4 Grants . . . 13
2.6 Energy estimation . . . 13
2.6.1 Simulation . . . 14
2.6.2 Hardware-based energy measurement . . . 14
2.6.3 Software-based on-line energy estimation . . . 14

3 Related work **15**
3.1 TinyOS . . . 15
3.1.1 Discussion . . . 15
3.2 Contiki . . . 16
3.2.1 Discussion . . . 16
3.3 TI-RTOS . . . 17
3.3.1 Discussion . . . 17
3.4 Centralized power management in Linux . . . 17
3.4.1 Discussion . . . 18

4 Design **19**
4.1 General hardware abstraction . . . 19
4.1.1 Device drivers . . . 19
4.1.2 Hardware interface layer . . . 20
4.1.3 Hardware platform layer . . . 20
4.1.4 Discussion . . . 21
4.2 Tock hardware abstraction . . . 21
4.2.1 Capsules . . . 22
4.2.2 Hardware interface layer . . . 22
4.2.3 Hardware platform layer . . . 23
4.2.4 Discussion . . . 23
4.3 Energy efficiency . . . 24
4.3.1 On-demand resource management . . . 24
4.3.2 Sleep modes . . . 24
4.3.3 Peripheral management . . . 25
4.3.4 Energy efficiency in Tock . . . 26
4.3.5 Discussion . . . 27

5 Implementation **29**
5.1 General implementation . . . 29
5.1.1 Capsule . . . 29
5.1.2 Hardware interface layer . . . 31
5.1.3 Hardware platform layer . . . 32
5.2 Device driver details . . . 34
5.2.1 UART . . . 34
5.2.2 I^2C . . . 35
5.2.3 Radio . . . 35
5.3 Energy efficiency . . . 36
5.3.1 Power manager . . . 36
5.3.2 Peripheral manager . . . 37
5.3.3 Sleep/Power modes . . . 38
5.3.4 Configuring the Sensortag for low-power consumption . . . 39
5.3.5 Power saving features . . . 40

6 Evaluation **43**
6.1 Evaluation setup . . . 43
6.1.1 Energy efficiency . . . 43
6.1.2 BLE . . . 44
6.1.3 Measurement setup . . . 44
6.2 Results . . . 45
6.2.1 Power consumption during inactivity . . . 45
6.2.2 Tock power consumption . . . 47
6.2.3 Blink . . . 48
6.2.4 BLE . . . 49
6.2.5 Expected lifetime . . . 50
6.2.6 Abstraction overhead . . . 51

7 Discussion **53**
7.1 Rust . . . 53
7.2 Sleep modes . . . 54
7.2.1 Transition responsibility . . . 55
7.3 Tock . . . 55
7.3.1 Architecture . . . 55
7.3.2 Abstractions over unsafe code . . . 56
7.3.3 Portability . . . 56
7.4 Choice of platform . . . 56

8 Conclusion **59**

Bibliography **64**

List of Figures

2.1 TockOS Architecture 12

4.1 Applications communicate with device drivers 20
4.2 Hardware abstraction layers 21
4.3 Tock abstraction layers 22
4.4 On-demand dynamic resource management design 24
4.5 Interaction between peripherals and the peripheral manager 25
4.6 Tock Kernel Main Loop 26

5.1 Power saving features 40

6.1 Measuring setup using an Oscilloscope 45
6.2 Standby power consumption comparison 46
6.3 Tock Power modes current draw 47
6.4 Tock wakeup with and without deep sleep 48
6.5 Toggling LED on Sensortag with Tock 49
6.6 BLE advertising on Sensortag with Tock 50

List of Tables

2.1 External sensors found on the Sensortag platform. 9
2.2 Hardware specification for the CC2650 microcontroller. 10

6.1 Expected lifetime in different use cases of a Sensortag device running Tock with our modifications. 50

Acronyms

BLE Bluetooth Low Energy

GPIO General Purpose Input/Output

HAL Hardware Abstraction Layer

HIL Hardware Interface Layer

HPL Hardware Platform Layer

I^2C Inter-Integrated Circuit

IoT Internet of Things

MCU Microcontroller Unit

MMU Memory Management Unit

MPU Memory Protection Unit

OS Operating System

TI Texas Instruments

UART Universal Asynchronous Receiver/Transmitter

1
Introduction

An increasing variety of physical objects connect to the Internet. Embedded systems underlie everything from washing machines to light bulbs. The wide range of devices gaining connectivity have coined a new term: The Internet of Things (IoT). The IoT makes it possible to collect and act on information from a lot of different sources. This information can be used to regulate sophisticated systems like power grids or self-driving cars.

Embedded systems have traditionally been simple systems focusing on a very specific set of tasks. As hardware has gotten more capable, they can now provide more advanced applications to its users and run several different applications at once. An advanced software platform with connectivity to the internet needs an operating system (OS) to coordinate all the work on the system. This calls for an OS that can provide multiprogramming and respect strict hardware constraints. Embedded systems need to stay functional for long periods of time and as a consequence of this, fault-tolerance and energy efficiency are also high-priority goals for embedded systems.

Tock is an OS providing many of the features mentioned above. It focuses on being resource efficient and safe. For this purpose the kernel is written in Rust, a type-safe language without garbage collection [1]. Rust is a low-level language like C, but provides many features from object-oriented languages. Memory management also works in a radically different way, by using ownership and lifetimes.

This thesis describes the process of porting Tock to new hardware platforms and how the kernel can be extended with new features for improved energy efficiency.

1.1 Context

Writing hardware-specific applications in order to decrease the power consumption is cumbersome, and a more general approach is preferable. Therefore, it is motivated to provide a flexible way for power management in the kernel so that software developers can focus on other parts of their applications instead.

Many of the hardware platforms which Tock supports have several different features (such as sleep modes) which could be used to decrease the power consumption. These features were not being utilized by Tock before this work.

Contiki [2], TinyOS [3], and other embedded OS today support sleep modes for their applications in order to reduce the energy consumption. Tock currently supports concurrency and de-prioritization of processes in order to conserve energy [4]. Tock has the potential to achieve better energy efficiency than it has today and

this is crucial for battery-driven devices that needs to be running for several months or even years.

1.2 Goals and Contributions

Our main goal is to evaluate and analyze the energy efficiency and the abstraction Tock uses to leverage Rust's features to its advantage in an embedded environment. We have therefore decided upon two research questions and three contributions to Tock.

Tock is currently only supported on three different platforms. Porting Tock to an additional platform and documenting process, increases the availability of the OS and helps with future development when adding support for new platforms.

1.2.1 Problem statement

- How do the abstraction layers in Tock affect portability?
- How does an embedded OS implemented in Rust compare to other state-of-the-art implementations in terms of energy efficiency?

1.2.2 Contributions

- Port Tock to an additional development board by adding support for a new family of microcontrollers and writing device drivers for the board's different peripherals.
- Enable platforms that use Tock to decrease their power consumption.
- Evaluate the energy efficiency of Tock during different levels of activity.

1.3 Report structure

The report starts with a theoretical background, giving the fundamentals to understand the rest of the report. We talk about different hardware technologies, Rust, and give an introduction to Tock. Following the background is a section about related work. Related work concerns the alternatives to Tock, and describes a couple of other popular operating systems for sensor networks.

The design section gives more information about Tock's abstraction layers and describe, at a higher level, how a driver should be designed to fit with the rest of Tock. Then we give some more details about each driver and our abstractions for power management in the implementation section. The evaluation section talks about testing our design and the metrics we use. This is also where we show the results from our benchmarks.

The discussion section is where we talk about how suitable Rust is as a language for embedded operating systems, and if Tock can leverage its features to its

advantage. Our results regarding energy-efficiency are discussed and we share our experiences about porting Tock to a new platform.

The report ends with a conclusion that summarizes our findings and discusses the future of Tock.

2
Background

This chapter describes concepts and knowledge necessary to understand the rest of the thesis. It begins with a description of embedded operating systems in Section 2.1. It then continues with a brief description of the peripheral hardware used in Section 2.2, followed by information about the platforms picked in Section 2.3. The chapter then moves on to the Rust programming language in Section 2.4. Afterwards, Section 2.5 goes through parts of Tock not covered by this thesis for completeness. Finally, the chapter concludes with energy estimation in Section 2.6.

2.1 Embedded Operating Systems

In [4], we can read about five key features that embedded operating systems need to have in order to support the applications of tomorrow. The mentioned features are dependability, concurrency, fault isolation, efficiency, and run-time updates. They focus mainly on resource sharing between processes and preventing and recovering from faults.

Dependability Embedded systems are often hard to access and can be found in very remote locations. Because of this, they need to be up and running for an extended period of time without the need for maintenance. An example could be a distant weather station that constantly needs to forward data from its sensors.

Concurrency As embedded systems get more complex, we need support for an environment that can run several different applications concurrently. This also increases power efficiency since we can allow the device to enter sleep mode when asynchronous operations occur.

Efficiency Embedded systems running on limited power (e.g. batteries) need to save their energy whenever possible to prevent the need for frequent maintenance. Another resource that is very scarce when it comes to MCUs is memory. Applications should ideally not allocate more memory than needed. Allocating memory dynamically to applications is one of the most efficient ways, but this comes with the disadvantage that some applications might fail when memory is exhausted by other applications.

Fault isolation Having multiple applications running on the same system means that we have to make them all get along, even when something goes wrong.

One way to ensure damage control is to isolate memory between applications. Modern MCUs make this easy with memory protection units that can be exploited by the operating system.

Updates at runtime IoT devices are often designed to run without downtime. To make it possible to run applications this way, there needs to be a way to communicate and apply updates to the system without interfering with the ongoing work.

2.2 Hardware

This section introduces some of the hardware and protocols encountered during the project. It is not meant to be an in-depth explanation of the technologies, but more as a short overview to get a notion of their purpose.

2.2.1 UART

UART (Universal Asynchronous Receiver/Transmitter) is a piece of hardware used for asynchronous serial communication. Asynchronous means that the beginning and start of a packet is signaled through start and stop bits, and not controlled by a shared clock.

The transmitting device sends data from its TX-pin to the receiving device's RX-pin. Transmission speed is decided by the *Baud rate* which expresses the speed in bits per second (bps). For two devices to communicate with each other, they must use roughly the same baud-rate.

UART requires little setup and uses a simple protocol. It is often used as a debug tool and provides a way to get information from the system.

2.2.2 I^2C

I^2C is a way for integrated circuits to communicate with each other. It is often used by the MCU to control slower peripheral devices on the same board. More generally, I^2C provides for several *slave* devices to communicate with one or more *master* nodes. The hardware is simple in the sense that only two wires are needed for the communication. One wire is used for the clock signal (SCL) and the other for data (SDA). By manipulating the signals on these buses, certain transmission conditions (START/STOP/ACK), can be generated.

2.2.3 Memory protection unit

To support advanced memory architectures, many modern processors include a Memory Management Unit (MMU) to provide support for virtual memory. An MMU also provides protection between different memory areas by preventing errant or malicious applications to access parts of memory they are not authorized to use.

A Memory Protection Unit (MPU) is a simpler device that only provides the memory protection part of an MMU. MPUs is commonly found in embedded systems

with limited memory size where there is not enough resources for virtual memory to be useful.

2.2.4 Bluetooth low energy

Bluetooth is a technology for short-range wireless communication. Each Bluetooth device has a radio for transmitting and receiving data over the ISM-band. Bluetooth Low Energy (BLE) is a more power efficient and less costly version of Bluetooth that still maintains the same range as the original version. The key difference is that BLE does not continuously stream data, but remains in sleep-mode until there is something to send. BLE is common for portable, battery-driven devices that only sends data periodically. It is also found in sensor networks and other areas related to the Internet of Things.

2.3 Platforms

The main platform for this project is the Simplelink Sensortag [5] from Texas Instruments (TI). The Sensortag is a powerful platform with a lot of different sensors and peripherals. This platform is however not fully compatible with Tock's hardware requirements to provide isolation between applications. Therefore, Tock is also ported to a Launchpad with a similar microcontroller to enable us to get more feedback from the Tock core-team. A more elaborate discussion about the choice of platform is held in Section 7.4.

2.3.1 Simplelink Sensortag

The Sensortag consists of a CC2650 MCU together with a wide variety of external sensors (Table 2.1) [5].

Table 2.1: External sensors found on the Sensortag platform.

Article number	**Description**
MPU9250	9-axis Motions Sensor
TMP007	IR Thermopile Temperature Sensor
HDC1000	Digital Humidity Sensor
BMP280	Altimeter/Pressure Sensor
OPT3001	Ambient Light Sensor

The CC2650 MCU has a Cortex-M3 [6] processor and a radio-unit with support for Bluetooth Low Energy (BLE). Some of the microcontroller's specifications can be seen in Table 2.2.

2.3.2 Launchpads

Launchpads are evaluation boards available from TI to test their microcontrollers. They are circuit boards with just enough logic to load programs and get acquainted with the technology.

Table 2.2: Hardware specification for the CC2650 microcontroller.

Specification	Value
Clock Speed	48 MHz
Flash	128 KB
Cache	8 KB
SRAM	20 KB

In this project, Tock is ported to a launchpad running a prototype of a microcontroller (CC2652R1) which belongs to the same family as the microcontroller used by the Sensortag.

2.4 The Rust Programming Language

Rust is a programming language focusing on safety and concurrency. It uses a very strict compiler and type system to ensure safety and makes most of the security checks at compile time to save time during execution. Its syntax is similar to C and C++, but it takes a new approach to memory management. Instead of allocating and freeing memory dynamically during runtime, Rust introduces the concept of ownership. The language is also very strict when it comes to how variables changes their state. Variables are always immutable by default and can only change their value if they are declared as mutable.

2.4.1 Ownership

Ownership is one of Rust's most defining properties. Ownership means that all resources are owned by a specific variable, and once it goes out of scope, the resource is released. A resource might however be moved from one variable to another, which is called ownership transferal. It is also possible for one scope to borrow a variable from another scope, and then return it once it is finished. The rules of transferals is checked during compile time, which removes the need for extra overhead (e.g a garbage collector). Listing 2.1 illustrates the concept of ownership, and transferal, with a simple example of what happens when the rules are broken.

Listing 2.1: Transferal of ownership in rust

```
{
    let s = String::from("Test"); // Beginning of scope for s
    stringFunction(s);            // Transferal of s to function
    println!("{}", s);            // Fail: value has moved
}                                 // End of scope if not for fn call
```

2.4.2 Borrowing with references

It is possible for variables to be borrowed between scopes, which is done by lending a different scope a reference instead of the variable itself. The referenced variable could either be mutable or immutable, which is explicitly stated. It is not possible to borrow a mutable and an immutable reference at the same time, since this can result in race conditions [7].

Listing 2.2: Borrowing with references in Rust. The variables $v1$ and $v2$ is borrowed as references to the function *foo*, and then returned when foo returns.

```
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
    // do stuff with v1 and v2
    // return the answer
    42
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];
let answer = foo(&v1, &v2);
// we can use v1 and v2 here!
```

2.4.3 Lifetimes

Ownership and references is complicated in rust, and many failures can occur when sharing resources between scopes (e.g. a resource is freed by scope A, but scope B still has a reference to it). Lifetimes is used to explicitly inform the compiler of how long any given resource will exist [8]. You do not have to explicitly declare a lifetime for each resource, the compiler can automatically infer lifetimes for resources. See Listing 2.3 for an example of when the lifetime of a resource is shorter than the lifetime of a resource with a reference to it, causing an error.

Declaring lifetimes of resources is done by using $\{<, >\}$ brackets, e.g. $Foo <' a >$ where *'a* indicate the lifetime of the resource. You can use multiple lifetimes in a struct in order to assign different lifetimes to members.

Listing 2.3: Premature deallocation of a referenced resource in rust

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                    // x goes into scope
                              //
```

```
8       {                           //
9           let y = &5;             // y goes into scope
10          let f = Foo { x: y }; // f goes into scope
11          x = &f.x;               // error here
12      }                           // f and y go out of scope
13                                  //
14      println!("{}", x);          //
15  }                               // x goes out of scope
```

There is a special case of lifetime called *'static*, which means that the lifetime is infinite. *'static* resources will only be released once the program ends, and exists indefinitely. Borrowing a *'static* resource as mutable can present memory leaks [9, 10].

2.5 Tock OS

Tock is an embedded operating system for the IoT developed in Rust [4, 11]. The safety and performance traits of Rust makes it an interesting alternative to classical low-level languages like C for OS development. Tock tries to leverage the advantages of Rust to provide a fully featured OS for modern applications, running on limited hardware with strict power constraints. An overview of how Tock is designed can be seen in Figure 2.1.

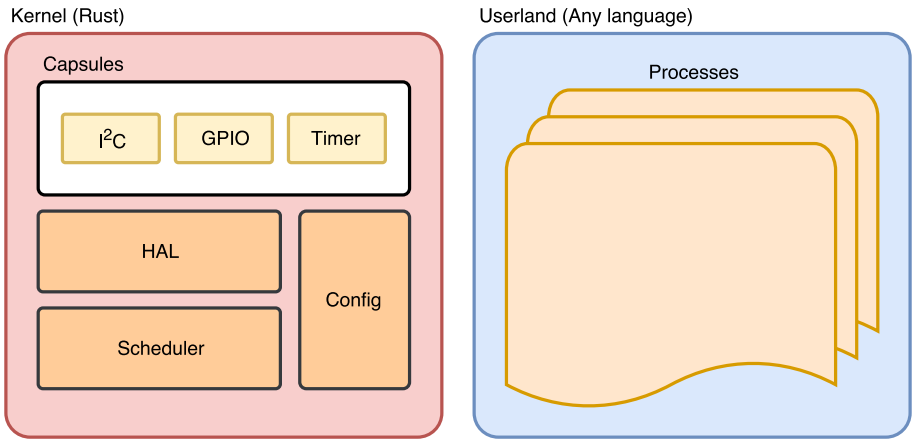

Figure 2.1: TockOS Architecture. The red box to the left is the Tock kernel. It consists of two main parts: capsules and core modules. Capsules are untrusted (no unsafe code allowed), and core modules in orange are trusted. The blue box to the right represents the userland, and the running processes/applications.

2.5.1 Kernel

Tock uses a micro-kernel design [4], and is built from several components called *capsules*. Capsules are sandboxed by Rust's strict type-system and untrusted by

default. Untrusted means that they are not trusted to execute unsafe code. The kernel is modular and only a subset of capsules are chosen at compile-time. Capsules are responsible for the communication with userland processes and is the main entry point when a process wants to use any type of hardware resource.

Some parts of the kernel need to leave Rust's type-system in order to perform their function. These modules reside outside of capsules and include the scheduler, board-specific configurations and the Hardware Abstraction Layer (HAL). Because of their use of unsafe language features, these modules are trusted to be well designed.

2.5.2 Userland

Userland contains applications which run with limited privileges [4]. Applications are not trusted and have limited memory access. They are designed to run without interfering with the kernel, meaning that they can be uploaded, enabled, and disabled during runtime.

Userland applications can be developed in any language, as their safety and reliability is ensured by the underlying operating system. The most common languages used today for userland applications are C and C++ [12].

Applications have a local memory area for their data which is independent from the rest of the system and other processes. This independence makes it easy to load and replace applications in comparison to capsules that need to be integrated with the kernel at compile-time.

2.5.3 Scheduling

Processes are scheduled with a round-robin algorithm while scheduling of capsules is event-based. Kernel events decide when capsules should run and these events always have higher priority than processes. Event handling in the kernel is not preempted which gives capsules the possibility to block other capsules and processes by doing long running computations.

2.5.4 Grants

Each process has a small heap that they can allocate memory from. When it comes to the kernel, it too needs to be able to dynamically allocate resources in certain situations. The way this has been solved in Tock, is that a fraction of each process's memory space is reserved for the kernel. These memory regions are called *grants* and they give us a dynamic kernel heap where processes can exhaust their own part of this memory without affecting other processes. Capsules can only work on grant memory and Rust's type-system and lifetimes make sure that references can't leave this area. It also enables us to quickly re-allocate memory if a process dies.

2.6 Energy estimation

Low power consumption is crucial for IoT devices. Many devices have limited power supplies and need to make informed decisions to preserve their lifetime. This section

discuss different approaches, both hardware and software-based, on how to estimate the power consumption.

2.6.1 Simulation

One way to estimate power consumption is to use a cycle-accurate simulator of the system where all different parameters can be controlled and measured. This gets increasingly difficult as the complexity of the system grows. One alternative approach is to create a profile of the hardware empirically and simulate power consumption using a model created from the profile [13]. A big question in this case is if the results from such a simulation can be generalized to the real world. In [14] the authors argue that the only way to see how temperature and the physical environment affects the result is to run experiments in the real environment and at scale.

2.6.2 Hardware-based energy measurement

Using a hardware-based approach to measure the power consumption is often accurate, but might require modification of the existing hardware [15]. Many methods for achieving accurate results are often complicated and costly [14]. A classic approach is to measure the voltage drop around a shunt resistor. Using Ohm's law one can calculate the current and ultimately the power by multiplying the current with the voltage over the device (Equation 2.1).

$$P_{device} = \frac{V_{shunt}}{R_{shunt}} V_{device} \tag{2.1}$$

2.6.3 Software-based on-line energy estimation

An alternative approach presented in [15] is to measure the time a device spends in certain power-modes together with the time peripherals are activated. A table keeps track of all relevant components. When a component is started, a time-stamp is created and when the component is turned off the difference is calculated from the initial time-stamp and added to the corresponding entry in the table. This approach is non-intrusive if the platform supports timers in its lower power modes.

This way of measuring power consumption is implemented in Contiki and is easy to port to similar operating systems [15]. How accurate this approach is needs more research [15], but it should be able to give good estimates when doing comparisons on the same platform. Equation 2.2 shows how the energy consumption can be calculated by multiplying the current draw (I_c) for different components together with the time (t_c) they spend in active-mode and then summing up the result for each component.

$$\frac{E}{V} = \sum_i I_{c_i} t_{c_i} \tag{2.2}$$

3
Related work

This section describes some of the popular alternatives to Tock today. Contiki is an open-source OS that already has support for the Sensortag platform and many of the drivers implemented will be inspired by the drivers that Contiki uses. It also describes TinyOS, since it is one of the earliest systems for sensor networks and some of the main contributors of TinyOS is now working on Tock. Another operating system worth mentioning is TI developed, namely TI-RTOS, and is the default operating system for the Sensortag platform supplied by TI.

3.1 TinyOS

TinyOS is one of the first operating systems for sensor networks. It focuses on providing a flexible platform for low-energy devices and follows a reactive programming model. TinyOS is not a stand-alone OS, but a series of modular components that are built into the application to create an application specific environment [3].

TinyOS uses its own dialect of C (NesC) that creates programs out of components that are *wired* together to form the final application. Components in TinyOS uses three different forms of abstraction. There is *commands* and *events* which is used for communication between components, and *tasks* which is a way to achieve concurrency within a specific component.

Commands and events create well-defined *interfaces* between components. Commands are requests to specific components to perform something. Once a command is completed, the status of the command is then signaled through an event. This makes all communication non-blocking. Each computational request is turned into a task that can be scheduled individually. This allows a command to return immediately and the result to be reported back at a later time through an event.

Each application has a *wiring specification* that defines how different components should be fit together. Wiring specifications work as blueprints for the final program and are independent of how components are implemented.

3.1.1 Discussion

TinyOS is in a sense a predecessor to Tock, and some of the contributors of TinyOS is also developers of Tock. This effectively means that Tock includes improvements from lessons learned during the development of TinyOS. Tock does not suffer from the unsafety which comes with using C (or NesC) in TinyOS, and have more concern about safety in its design.

One major difference between Tock and TinyOS is the possibility to add applications at run-time. This is an advantage that Tock has as applications can be uploaded separately, contrary to TinyOS where applications are an integral part of the OS.

3.2 Contiki

Contiki is a lightweight operating system for large sensor networks. It provides a flexible environment for applications while still taking limited hardware resources into account. The architecture allows dynamic loading and replacement of applications and runs an event-based kernel [16].

An event-based kernel means that programs only run when triggered by events from the kernel. Once an event-handler has started, it runs to completion. This can cause problems when long-running calculations need to be run. However, there's an optional library that can be linked in to enable preemptive multi-threading [17].

One of Contiki's main features is the possibility to load and replace programs during runtime. The kernel informs a process that it is going to be replaced through a special event. It is then up to the process to shut down and clean up after itself. If information needs to be passed from the old process to the new one, there is a way to transfer the internal state by passing a pointer to the state description on to the new application.

Contiki is written in C and supports a large number of platforms, including the Sensortag from TI [2]. Another interesting feature is the possibility to estimate power consumption and locate where this power was spent.

3.2.1 Discussion

Contiki is widely adopted by the industry today. It is open source and has seen many contributions since its creation in 2002 [18]. It can already be used on the Sensortag, and the port is even supported by TI themselves. The wide adoption of platforms gives Contiki a clear advantage over its competitors.

Like Tock, Contiki is designed towards embedded systems with limited resources. There are, however, some disadvantages of Contiki - namely its safety. Contiki does not impose as strict safety regulations as Tock does by default. An example of this is the possibility to directly manipulate hardware registers without any safety guarantees. This can even be done from user applications, which is strictly prohibited by Tock. Being able to remotely upload applications, together with unrestricted memory access, is a potential security risk.

Tock focus more on safety and isolation, when Contiki provides very efficient features in terms of practicality. Tock views processes as potentially malicious, whereas Contiki views them merely as a unit of modularity. There is a higher risk for memory related crashes in Contiki (e.g. null pointers, deallocated memory, etc.) which is not likely to happen in Tock because of Rust. Tock has a more distinct segregation of its dependencies to increase reliability and safety, providing static verification during compile-time to guarantee that no memory issues occur during runtime.

3.3 TI-RTOS

TI-RTOS is a real-time OS developed by TI to be used with their own MCUs. The system provides a multi-tasking environment with protocol stacks and drivers so that developers can focus on applications instead of system software [19]. Since the operating system runs on hardware from the same developer, TI-RTOS is very efficient in terms of using its resources and power consumption. A power manager leverages the different energy saving features of the underlying hardware for minimal power consumption [20]. Some of the power saving features found in TI-RTOS are listed below.

Management of peripheral clocks and power domains is optimized to only enable certain power domains and clocks on demand.

Power states are used to turn off certain peripherals and run the MCU on a lower clock frequency. The OS keeps track of the peripherals used at any given time and decides which power-mode to use.

Tick suppression schedules ticks to appear together with other time-driven functions so that the MCU is not awoken from a low-power-mode when there is nothing to do.

3.3.1 Discussion

TI-RTOS is specifically designed for TI's own platforms. This means that they know all esoteric details about the hardware when writing their code (and vice versa), and hence can create very optimized solutions. There is an API provided from TI to make their platforms more accessible for those who does not wish to use TI-RTOS. This API is however written in C, and using it directly in Tock would circumvent many of the safety and reliability features provided by Rust.

3.4 Centralized power management in Linux

Power management is often implemented as an afterthought and as an effect of that, power management is often coarse-grained and hard to implement. Complex drivers make it hard to reason where it is safe to put code for power management.

Xu et al. [21] suggests a solution to this problem by introducing a centralized power manager in the Linux kernel. The power manager keeps track of whether a device has any pending tasks. If no tasks are pending, a power off request will be sent to the device. Two different ways are suggested to infer if a device has any pending tasks:

Software-based inference Access to device registers are monitored to see if there is any ongoing activity on a device. If a device is idle for a longer period of time than a certain threshold, the device is assumed to have no pending tasks and can be disabled.

Hardware-assisted inference Another approach is to extend the hardware with extra register bits, indicating if devices are busy or idle. These bits are then polled regularly to see there is any ongoing activity on a device. The reasoning behind this approach is that, once a task is finished, a device will immediately start processing a pending task. This means that the time threshold for when a device is assumed to be idle can be much shorter than in the software-based approach.

3.4.1 Discussion

A centralized power manager relieves driver developers from having to implement manual power management inside of drivers. Xu et al. acknowledges that software-based inference is less aggressive than power management in device drivers and that supervising access to hardware registers incurs some overhead.

Adding to this, hardware-assisted inference assumes devices capable of indicating when tasks are being processed. Another concern is the ability of a centralized manager to make use of devices with more sophisticated power-saving features. Many devices support several different modes of operation rather than just *enabled* and *disabled*.

4
Design

This chapter elaborates on the design of Tock's kernel and how its different components interface with each other. In Section 4.1 we begin with a theoretical description of how software can be executed independently of the hardware platform it runs on. Later, we get more specific in Section 4.2 where we look at how Tock implements several levels of abstraction to facilitate portability. Finally in Section 4.3 we take a look at energy-efficiency, and how Tock can benefit from several extensions to improve its power consumption.

4.1 General hardware abstraction

Adding support to several underlying hardware platforms increase the complexity of software design [22], and may result in excessive duplication of code. Therefore, abstraction layers are needed to provide common interfaces to the underlying hardware which the system utilizes. This makes porting easier and enables the execution of the same software on multiple hardware platforms.

4.1.1 Device drivers

In order to allow software to execute independently of what hardware platform it runs on, a software abstraction layer is needed. This is commonly done with an abstraction called device drivers [22]. Device drivers are designed to provide a generic interface towards the underlying hardware. They are responsible for configuration, listing, and communication with hardware devices.

Generally, device drivers have two different interfaces: one towards the hardware and one towards user applications [23, 22]. The interface towards applications usually looks the same for all device drivers. For example, there could be three different commands: configure, send, and receive. These commands are general enough to perform a variety of functions. To distinguish between different devices, the commands are used with different arguments and data. Libraries in userland can then wrap these commands to make it easier for applications to communicate with device drivers.

The interaction between applications and device drivers is illustrated in Figure 4.1. Note that there can be dependencies between drivers, as they can utilize the functionality of each other (e.g. a radio driver that utilizes a timer driver to schedule a transmission).

Device drivers are not only used to communicate directly with the hardware

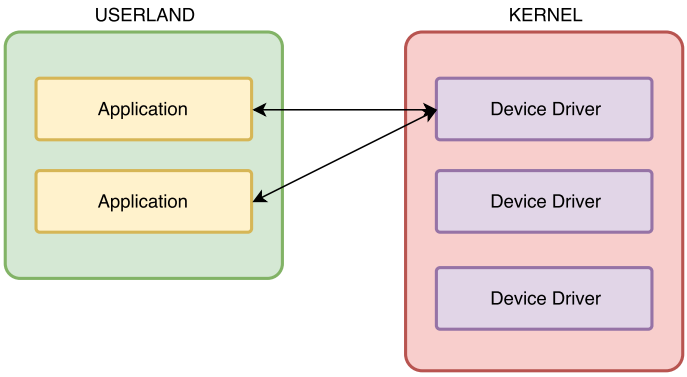

Figure 4.1: Applications communicate with device drivers in order to interact with hardware, regardless of what platform they run on.

[23], but also include platform independent logic, for example, implementations of protocols such as BLE. The reason for this is that some protocols are tightly bound to a specific type of hardware, like the radio.

How device drivers are used in embedded operating systems differ between implementations [24, 25, 26]. Most operating systems use device drivers as components attached to the kernel. The advantage of this modular approach is that many device drivers can be seen as optional. This also makes it easier to combine device drivers into new components that use utilize functionality from several drivers.

4.1.2 Hardware interface layer

The Hardware Interface Layer (HIL) provides a platform-independent abstraction that hides differences in underlying hardware [23]. Hardware often differs in its implementation and interaction depending on who manufactured it and what version it is, making it more complex to add support for new platforms. The HIL is composed of several hardware interface models, which each creates a contract of how the underlying hardware behaves - regardless of the platform.

An interface model is an abstraction of how a specific type of hardware functions (e.g. I^2C, BLE). It sets an expectation of what features and functionality the underlying hardware should provide. This abstraction allows an interface to be set up between device drivers and the hardware, thus allowing device drivers to interact with the hardware independently of what platform is used. Figure 4.2 illustrates how a device driver can be compatible with several hardware platforms if both parts follow the contract defined in the HIL.

4.1.3 Hardware platform layer

The Hardware Platform Layer (HPL) is the layer closest to the hardware. This layer handles all details about the hardware so that the layers above do not have to. The HPL implements the functionality defined in the HIL to make the platform compatible with the device driver. Each platform needs its own HPL in order to be compatible with the OS, which is illustrated in Figure 4.2. The HPL is ideally the only layer that needs to be ported to add support for a new platform.

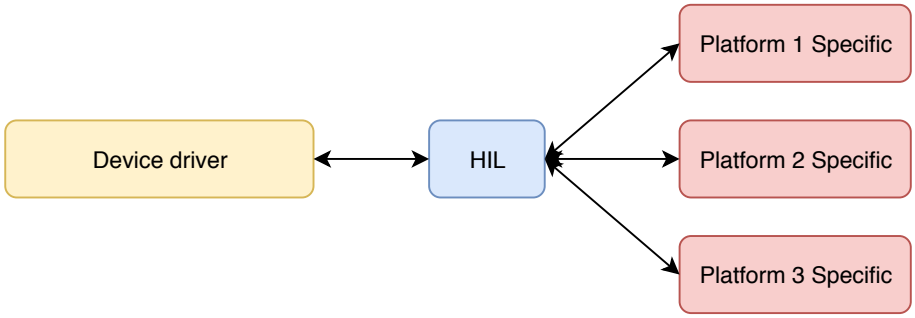

Figure 4.2: The abstraction layers used to interface to different platform-specific hardware. How the device driver uses the hardware interface layer in order to use platform-dependent implementations of hardware interaction.

4.1.4 Discussion

There are always some limitations on how general an abstraction can be. The more complex a device is, the harder it is to find a suitable abstraction. It is easy to describe an abstraction for something as simple as an LED or a button, but harder for a radio with support for multiple protocols. Complex hardware works in very different ways depending on the manufacturer, and what design choices were made during development.

Hardware abstractions may cause portability to suffer [27], as certain assumptions often are made in higher abstraction layers. An example of this would be if the OS assumes the existence of a specific hardware device. This would limit the portability of the OS to only support platforms which use this device.

Embedded operating systems and general purpose systems have different ways to manage hardware abstractions, with different goals and purposes in mind. Embedded systems work with a lot of restrictions in terms of resources, which makes it harder to use advanced and complex abstractions. Using Rust to design the abstractions gives a huge advantage since everything is checked during compilation, and not during run-time. This minimizes overhead and allows for flexible and portable abstractions.

4.2 Tock hardware abstraction

Tock uses different abstraction layers in order to easily extend or modify specific parts of its functionality. Figure 4.3 shows a high-level overview of Tock's current design. Notice how the kernel consists of capsules that interacts with both hardware and software.

In this section, we describe the four noticeable abstraction layers for the kernel: capsules, HIL, HAL, and board-specific configurations. Capsules are described in Section 4.2.1 and the HIL in Section 4.2.2. The HAL together with board-specific configurations are what Tock consider to be its platform dependent implementation, and is described in Section 4.2.3.

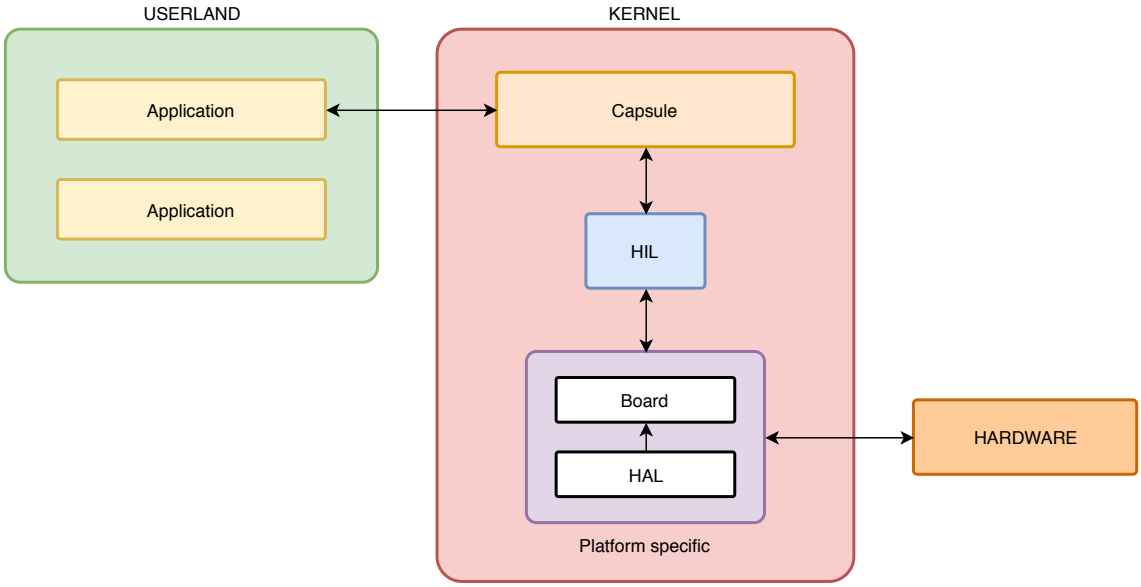

Figure 4.3: The abstraction layers of Tock. Applications communicate with capsules, which uses the HIL to interact with platform-specific implementations, which in turn interacts directly with the hardware.

4.2.1 Capsules

Capsules are Tock's main abstraction for device drivers. They use Rust's type and module system to provide isolated units that communicates through well-defined interfaces. It is a robust design since they are not allowed to use any unsafe code.

Capsules are meant to provide platform independent implementation of device functionality (e.g. protocol implementations, virtual multiplexing) with a higher execution privilege than regular applications. They can act as standalone units to provide features which share resources between applications, or directly act as an intermediary between applications and specific hardware - or even a combination of them both.

Applications communicate with capsules using a generic interface which is capsule independent, often in conjunction with a userland library to facilitate communication. This allows applications to communicate with every capsule regardless of its functionality, or what hardware it uses.

4.2.2 Hardware interface layer

The HIL is used to interface between capsules and the HPL. The HIL should be as generic as possible and depends on what type of hardware it interfaces against. The information included should add an abstraction over the actual hardware interaction, making the capsule oblivious to how the communication with the hardware is implemented.

The HIL is the only way for capsules to communicate with and use hardware in Tock since unsafe code is not allowed in capsules. This allows capsules to be ported much easier to new platforms, since there would only be a need to add a new HIL implementation for a new platform in order to use a specific capsule.

4.2.3 Hardware platform layer

The hardware platform layer in Tock has been divided into two parts: the Hardware Abstraction Layer (HAL), and board-specific configuration. A certain type of MCU is often used in several development boards, but with a different set of peripherals connected to it. It makes sense to design the HPL in such a way that MCU specific logic is kept in the HAL and can be used by multiple boards which share the same MCU.

4.2.3.1 Hardware abstraction layer

The HAL covers the MCU specific logic and implementation of all HIL modules that are going to be supported. The aim is to keep the design general towards a family of MCUs to avoid duplicate code. One module implements all the common parts for a certain family and then a sub-module extends this with more specific settings for a certain MCU.

4.2.3.2 Board-specific configuration

This part of the hardware abstraction configures the OS for a specific board by deciding what capsules to use and connecting these with the HAL. This is also where all pins are configured to match the layout of the board.

4.2.4 Discussion

The abstraction layers in Tock make it easy to add support for new platforms, and the only layers that need to be implemented are the HAL and board-specific configuration. However, the other parts of Tock need to be taken into consideration as well.

Capsules are embedded into the kernel and cannot be changed dynamically during runtime. This design choice makes it necessary to explicitly include capsules for each platform in the board-specific configuration. If an application needs to be added that depends on another set of capsules, the whole kernel needs to be recompiled. Capsules can also cause issues as they make assumptions of how the underlying hardware works. This can increase the complexity of the HAL as workarounds might be needed to comply with these assumptions. This problem is hard to avoid, and both the HIL and capsules need to be updated as more platforms get supported by Tock.

The HAL contains MCU specific logic, which is shared between MCUs of the same family. By keeping the HAL platform agnostic, Tock can easily be ported to platforms which use hardware components that are already supported.

Handziski et al. describes how HIL models have to be actively developed in order to cope with changes in the underlying hardware [23] - especially for wireless sensor networks. Hardware is constantly improved and changed, which means the HIL will sometimes need to be updated. This could pose a problem in Tock since all of the abstraction's implementations on all supported platforms would need to be updated.

4.3 Energy efficiency

Energy efficiency is highly integrated into the hardware of IoT devices today. It is desirable for any IoT device to last as long as possible, consuming the minimum required amounts of energy to extend its lifetime. Several techniques can be used to reduce the energy consumption of a device, and this section will discuss some of the most common ones.

4.3.1 On-demand resource management

Resources needs to be dynamically managed in order to reduce the energy consumption for a certain board. There is no need for a clock to tick or a peripheral to have power if they are not used.

Dynamic management of resources can be designed in many ways depending on the underlying hardware. Ideally, power management should be general enough to support several boards, even if this is challenging since they often have very different power saving features.

Peripherals need different resources in order for them to work. For example, if a peripheral resides in a specific power region, that power region needs to be enabled for it to work. Certain resources are also shared between peripherals. On-demand resource management need to keep track of each resource, and their usage, to know when to enable or disable them. This can be done by creating a power manager, which is shown in Figure 4.4.

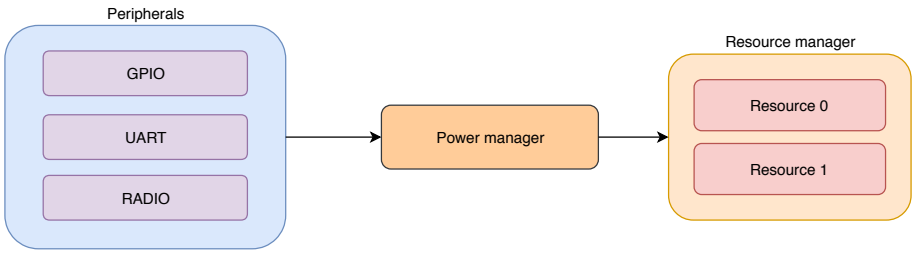

Figure 4.4: Resources are requested and released by peripherals through a power manager. The power manager keeps track of the number of peripherals who use a certain resource and can on-demand decide if it needs to be powered on.

The management of resources could be more abstract, which requires more assumptions to be made about the underlying hardware. It would minimize the amount of platform-specific implementation, but limit the number of platform-specific energy efficient features that could be used. Using these platform-specific features is a huge advantage in terms of energy efficiency, which is why our design kept the entire resource management platform-specific.

4.3.2 Sleep modes

In order to reduce the energy consumption during inactivity, it is common to utilize sleep modes for peripherals and the MCU. This suspends the peripheral or the

MCU until a certain event has occurred, thus reducing the energy consumption drastically. There is often several kind of sleep modes depending on the situation, which all offers trade-offs in energy consumption and functionality. Lower energy consumption often means reduced functionality during these periods. An example of the predefined sleep modes for the Sensortag [6] can be seen below:

Active An application is currently running on the MCU.

Idle No application is running, the MCU is powered off, but all peripherals are functional and available.

Standby No application is running, the MCU is powered off, and only the sensor controller and alarm is available.

Shutdown Everything is powered off.

In order to know how far into sleep it is possible to transition, we need to keep track of what peripherals that are being used and what sleep modes they are compatible with. Even if certain peripherals are not being used currently, they might be waiting for something to happen, e.g. waiting on an interrupt from a sensor.

4.3.3 Peripheral management

To avoid interfering with ongoing peripheral activity, there needs to be a way to decide if it is safe to drop to a certain sleep mode. This section describes a manager that peripherals can use to receive notifications about power mode transitions and prevent certain transitions if needed.

It should be possible to query the peripheral manager to determine the lowest sleep mode currently supported by the board. The peripheral manager loops through all subscribers and simply ask them which sleep mode they support at the moment. The lowest sleep mode supported by all subscribers is then returned.

Since certain peripherals need to be disabled before certain sleep modes, the peripheral manager should be able to notify peripherals before and after power mode transitions. Peripherals can then release their acquired resources safely, and then re-acquire them upon wakeup. This is useful for peripherals that perform periodic tasks and do not need to be powered on in-between (e.g. a radio sending advertisements periodically). See Figure 4.5 for an overview of the manager.

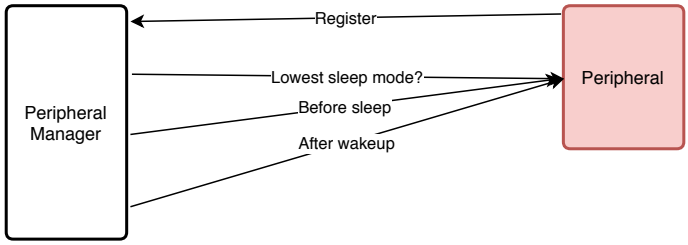

Figure 4.5: The peripheral manager lets peripherals subscribe to it, and then invokes several hooks for these once transitions between power modes are to occur.

4.3.4 Energy efficiency in Tock

Tock uses an event-based kernel, that waits for interrupts during inactivity. Tock uses the default wait-for-interrupt (WFI) instruction supported by all ARM processors (the main architectural target of Tock). This instruction reduces the power consumption of the CPU when it is not needed. The WFI instruction is not enough in many cases, however, and most hardware platforms support even lower power modes by limiting their functionality (see Section 4.3.2). Tock needs to be extended to support these power modes to improve its energy-efficiency further.

Tock always precedes the WFI instruction with a platform-specific invocation to prepare the chip for inactivity, but not upon wakeup. Most hardware needs to be configured before transitioning into inactivity, as well as afterwards. An illustration of Tock's current main loop is shown in Figure 4.6.

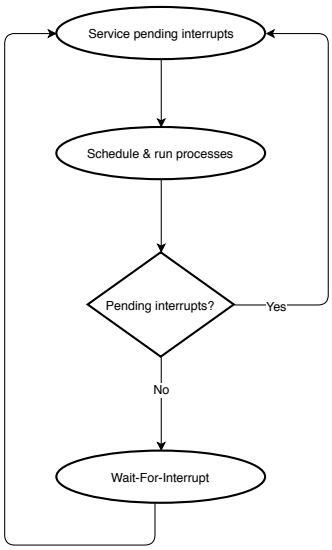

Figure 4.6: The main loop of the Tock kernel. It serves pending interrupts by invoking their handlers, and then schedule each process until any application yield, finally it transitions into hardware specific sleep.

4.3.4.1 Peripheral manager

Tock already has an implementation of a peripheral manager [28]. It enables notifications and hooks for peripherals before and after access to hardware registers. It is an abstraction which can be used to remove much of the unsafe code currently in the HAL.

The Sam4L chip supported in Tock uses it to toggle clock gates for peripherals before and after hardware register access, and ultimately uses the status of the clock gates to determine if it is safe or not to transition into a lower power mode (Section 4.3.2 for further information).

4.3.5 Discussion

Power management in Tock is currently very limited, and much work is being done in order to improve this (e.g. safe peripheral management [29]). There is much to gain by implementing on-demand resource management (Section 4.3.1) and support for sleep modes (Section 4.3.3). More sophisticated sleep modes would reduce the power consumption drastically, especially for IoT-devices with large periods of inactivity.

The power management in Tock today is very platform specific, which means that performance will vary when new boards are added. Since the platforms have different features for saving power, this is a necessary evil, but with the different managers described in this chapter, we hope to make it somewhat easier to implement support for these features in Tock.

The current peripheral manager in Tock does not handle certain edge-cases related to asynchronous operations. If a clock is disabled during an asynchronous operation, the operation would be disabled as well. This adds a conditional when we disable the clock. Therefore, another hardware register access would have to be made once all asynchronous operations has been completed to successfully trigger the hook after access, to disable the clock. This last hardware access would need to be made before entering a lower power mode, else it would unnecessarily prevent it. The current peripheral manager could potentially be merged with our own design mentioned in Section 4.3.3 to fix this issue.

The peripheral manager described in Section 4.3.3 needs to poll all peripherals to decide the lowest power mode safe to enter. An improvement to this could be to have peripherals explicitly inform the manager when their minimum permissible sleep state changes. Doing this means that the manager only has to re-evaluate what sleep mode to use when something changes. This could provide a more scalable solution, but the manager still has to inform all peripherals to prepare them for the transition. However, it is safe to assume that the number of peripherals attached to an IoT device is fairly small, which means that the overhead from polling all peripherals should be negligible.

5
Implementation

In this chapter, we look more closely at how the abstraction models from Chapter 4 are implemented in Tock, as well as the energy efficiency features. We begin with a general description of how drivers are written in Rust before diving into details about some of the drivers developed during the project. This is followed by details about how the energy improvements were implemented, and what trade-offs there are in their implementations.

5.1 General implementation

This section gives an overview of how different layers of the hardware abstraction are implemented in Tock. It includes details as well as code examples that show how the different software constructs are used. The implementation of capsules are described in Section 5.1.1, the HIL in Section 5.1.2 and the HPL in Section 5.1.3.

5.1.1 Capsule

A capsule acts as a device driver. It communicates with user applications through system calls and hardware devices through the HIL. Capsules often include abstracted logic, for example how a communication protocol works at a higher level.

When the kernel is invoked through a system call, the kernel performs several checks to see if the capsule exists and if it is busy. If the checks fall through, the scheduler invokes specific functions in the correct capsule. Therefore, each capsule need to implement a common communication interface:

allow(appid, driver-num, slice)
: Allows the kernel to use memory allocated by the application. It is used to transfer results back from the kernel to the application.

subscribe(driver-num, callback)
: Subscribe allows the application to pass on a callback to the capsule, which is invoked once a certain criteria is fulfilled.

command(driver-num, data, appid)
: Command instructs the capsule to perform a specific operation. This interface allows a limited amount of data to be transferred with the command.

The implementations of these functions differ depending on what the job of the capsule is. Many capsules implement extra commands via the *command* function,

and has extended functionality due to it. An example of how these functions are used together would be to invoke a certain *command* and give access to some memory using *allow*; and once the capsule is finished, make it invoke a callback which was registered using *subscribe*.

Capsules depend on the HIL to communicate with the hardware. To allow two-way communication, capsules themselves implement a client interface. This allows the hardware to notify the capsule when a specific operation has been completed.

Capsules are represented as structures in Rust, using traits as a templates for the HIL modules. This allows great flexibility when designing capsules, as traits are highly generic and simply define an interface - a contract of what behaviour is to be expected from a certain type of hardware. The actual HIL modules which the capsule uses is assigned during the runtime configuration. Listing 5.1 contains an example of how a capsule is constructed and connected with the HIL.

Listing 5.1: Example of capsule implementation. Notice the communication interface on line 8, 13, and 18, that is expected by the *Driver* trait. On line 7 it is explicitly defined that the *kernel::hil::Module* trait is used to communicate with the underlying hardware. Both communication directions use abstract traits to form a contract of communication - effectively an abstract protocol between applications, the capsule, and the hardware.

```
1  // Capsule 'capsule', using kernel::hil::Module for HIL interface.
2  struct Capsule<'a, H: kernel::hil::Module + 'a> {
3      hil_interface: &'a H,
4      // ...
5  }
6
7  impl<'a, H: kernel::hil::Module> Driver for Capsule<'a, H> {
8      fn allow(&self, appid: AppId, allow_num: usize,
9          slice: AppSlice<Shared, u8>) -> ReturnCode {
10         // implementation of allow
11     }
12
13     fn subscribe(&self, subscribe_num: usize, callback: Callback)
14         -> ReturnCode {
15         // implementation of subscribe
16     }
17
18     fn command(&self, command_num: usize, data: usize, appid: AppId)
19         -> ReturnCode {
20         // implementation of command
21     }
22 }
```

5.1.2 Hardware interface layer

The Hardware Interface Layer (HIL) in Tock is a work in progress. It is constantly being developed and improved to cope with the needs of tomorrow. HIL models depicts how a capsule expects the hardware platform layer to function in order to provide the necessary service to applications. Capsules often expect information about certain events from the hardware as well, and a way to disable or enable these events.

Events are most often handled by creating another interface which the HAL will use to notify capsules of certain callbacks. The capsules is expected to implement these callback interfaces, effectively creating an abstraction using a two-way protocol of interfaces between Capsules and the platform layer.

In Listing 5.2 is an example of a HIL model. It is a simple rust trait, where the functions *async_read, async_write* is used to communicate with the hardware device. It is also common to create a callback interface (see line 76) to be notified when certain operations has been completed by the hardware.

Listing 5.2: Hardware interface module example. It shows how a HIL model can be created to allow asynchronous communication and notification when operations has been completed by using a callback interface *Client*.

```
1  pub trait Module {
2      // Perform asynchronous read
3      fn async_read(&self);
4      // Perform asynchronous write of `data`
5      fn async_write(&self, data: DataType);
6
7      // Callbacks are often invoked by using interrupts, thus
8      // we need to be able to either enable or disable them.
9      fn enable_interrupt(&self);
10     fn disable_interrupt(&self);
11 }
12
13 // Callback interface towards the capsule using the HIL
14 pub trait Client {
15     // Invoked when a read operation has been finished
16     fn read_done(&self, data: DataType);
17
18     // Invoked when a write operation has been finished
19     fn write_done(&self);
20 }
```

5.1.3 Hardware platform layer

The hardware platform layer in Tock consists of two parts: the HAL, and board-specific configuration. The HAL contains logic for the MCU and its peripherals. It interacts directly with the hardware and is only compatible with a certain MCU or family of MCUs. The board-specific configuration is in turn responsible for connecting modules in the HAL with capsules and configuring what pins they use.

5.1.3.1 Hardware abstraction layer

The HAL contains MCU specific logic which performs direct configuration of hardware peripherals. Its purpose is to implement different parts of the HIL to establish an interface between capsules and hardware. The HAL contains unsafe code since it needs to perform raw memory operations. This means that it is not restricted by Rust's type-system and needs to be trusted by the kernel.

Listing 5.3 shows a bare-bones example of a HAL module. Almost all of the modules follow this pattern. They each interface directly with a set of hardware registers, which is mapped to specific memory locations. To be able to write to memory without any restrictions, we need to explicitly declare that the code is unsafe by using an unsafe-block. This can be seen in *start_peripheral()* at line 22, where we dereference a pointer to the registers.

Listing 5.3: Chip module example. The module interfaces directly with the registers found on line 1. The registers are mapped to a memory address (provided by hardware design), shown on line 7.

```
1  pub struct Registers {
2      pub control: VolatileCell<u32>,
3      _reserved: [VolatileCell<u8>, 0x10],
4      pub status: VolatileCell<u32>,
5  }
6
7  const MODULE_BASE: u32 = 0x1234_1234;
8
9  pub struct Module {
10     regs: *const Registers,
11     state: Cell<u8>,
12 }
13
14 impl Module {
15     const fn new() -> Module {
16         Module {
17             regs: MODULE_BASE as *const Registers,
18             state: Cell::new(0),
19         }
20     }
21
```

```
22      fn start_peripheral(&self) {
23          let regs = unsafe { &*self.regs };
24          regs.control.set(1);
25      }
26
27      fn return_state(&self) -> u8 {
28          self.state.get()
29      }
30  }
31
32  impl kernel::hil::module::Module for Module {
33      // HIL implementaions
34  }
```

Tock uses several software abstractions to manipulate ownership and make it easier to share data contained in unsafe and low-level structs. The *Cell* abstraction used for the *state* variable in Listing 5.3 allows it to be accessed several times by creating copies of the value contained inside of it [30]. An alternative for more advanced members, where copying involves significant overhead, is the *TakeCell* construct. In short, TakeCell allows us to define a closure where we can manipulate a variable before returning ownership to the original owner.

The HAL module also configures the device for interrupts. The kernel will forward interrupts to the module by calling a *handle_interrupt* function. This function handles the interrupt accordingly and forwards the event to the capsule by using a callback client.

5.1.3.2 Board-specific configuration

HAL modules are written to be compatible with all boards that support a certain MCU. This avoids code duplication but requires some initial configuration of the board. Each module needs to be attached to its corresponding capsule and callback clients need to be set for handling interrupts. The HAL modules must also be configured to use the pin layout of this particular board.

All of this happen in *main.rs* which is defined for each board in Tock. This is where the whole board is set up before loading the application and starting the OS. Listing 5.4 shows how a GPIO driver is set up for an example board before jumping to the operating system's entry point.

Listing 5.4: A somewhat simplified example of how to set up a board with a GPIO driver. The *Platform* struct holds the device driver which is initialized with the *GPIOPin* chip module on line 6 and 14. On line 18 we give each pin module the capsule as a callback client.

```
1  pub struct Platform {
2      gpio: &'static capsules::gpio::GPIO<'static, cc26xx::gpio::GPIOPin>,
```

```
3  }
4
5  pub unsafe fn reset_handler() {
6      let gpio_pins = static_init!(
7          [&'static cc26xx::gpio::GPIOPin; 3],
8          [
9              &cc26xx::gpio::PORT[0],
10             &cc26xx::gpio::PORT[1],
11             &cc26xx::gpio::PORT[2],
12         ]
13     );
14     let gpio = static_init!(
15         capsules::gpio::GPIO<'static, cc26xx::gpio::GPIOPin>,
16         capsules::gpio::GPIO::new(gpio_pins)
17     );
18     for pin in gpio_pins.iter() {
19         pin.set_client(gpio);
20     }
21
22     let platform = Platform { gpio }
23
24     // Init kernel with platform struct and run main
25 }
```

The platform struct holds all device drivers the platform is going to use. In Listing 5.4 this corresponds to the GPIO capsule using the *GPIOPin* chip module. Everything is initialized in the reset handler which is run every time the system boots. The *static_init* macro helps us instantiate static variables at run-time.

At line 18 we can see how all instances of the hardware module gets the capsule as a callback client. This client is used to signal the capsule after interrupts.

5.2 Device driver details

This section describes specific drivers and hardware modules in greater detail. For each peripheral, we describe what functionality the driver provides to the system and some details about the HIL functions they implement.

5.2.1 UART

This module handles all interaction with the UART hardware in the MCU. It provides a simple way of transmitting messages to an external device. Receiving is not implemented at the moment and interrupts are not enabled. UART is only used for debugging in this project, which is why only a handful of hardware features are supported.

From the HIL we implement a function for initializing the hardware and a another for transmissions. The *init* function powers up the peripheral, disables interrupts and configures the hardware by setting the baud rate and enabling requests to be queued in FIFO order. After all configuration parameters are set, this function enables the hardware by writing to the corresponding registers.

The transmission function takes a byte array together with its length as arguments and issues the bytes to be written once there is room in the FIFO queue. Once a transmissions has been completed, a callback is made to the kernel indicating that the command is complete.

5.2.2 I^2C

I^2C is used to communicate with external sensors on the board. There are two different interfaces/buses it can communicate on. Before communication can take place, an interface and an address need to be selected. The hardware module allows a master node to communicate with one slave node at a time. The operations are *read*, *write* and *read_write*. Read and write takes a data buffer as argument together with the number of bytes to read/write. *read_write* combines the two operations and performs first a write operation and then a read. In this last operation, both read and write share the same buffer.

The I^2C module is used exclusively by modules controlling the external sensors. We do not provide a way for user applications to utilize the I^2C driver on this board.

5.2.3 Radio

This module handles the interaction with the radio hardware, and thus part of the Bluetooth stack. The Tock BLE implementation is still unfinished, and the design of how to handle radio communication is a work in progress. Currently, only advertisements are supported and transmissions can only occur one at a time for every dedicated BLE channel.

The CC26xx family of MCUs use a dedicated extra Cortex-M0 MCU to handle all Radio communication. The communication with the main MCU (Cortex-M3/M4) and the Radio MCU is performed through shared memory and dedicated registers. Commands are sent to configure the radio and communicate using either BLE, IEEE802.15.4, or FM. It is possible to use all of the aforementioned protocols at the same time, thus achieving high flexibility in terms of wireless communication.

There are mainly two types of commands used in communication between the MCUs:

Direct commands are simple and often used to toggle a single function. They often have no arguments and are very simplistic in their nature (e.g. ping).

Immediate commands are dedicated memory structures that form more complex commands (e.g. transmission, and settings) with a wide range of variables and arguments.

It is possible to share commands between the MCUs as the Radio MCU has unrestricted access to the entire RAM of the main MCU [6, p. 1586]. The main

MCU issues commands by assigning a specific memory-mapped register to either an integer indicating a direct command or a pointer to a memory address indicating an immediate command.

The way CC26xx handles radio communication differs from how other chips manage the radio in Tock. The capsule is very specific to the radios currently supported and the HIL is being redesigned to change this. The final implementation of the radio for the CC26xx family will have to be updated in the future to reflect these changes. Hopefully, this will lead to a solution with fewer workarounds because of assumptions made higher up in the hardware abstraction.

5.3 Energy efficiency

To coordinate the power management of the board, we have designed two different software constructs that keep track of different resources being used and decide what power mode we can drop to when sleeping. The two constructs are the *Power manager* (Section 5.3.1) and the *Peripheral manager* (Section 5.3.2).

5.3.1 Power manager

This construct facilitates the management of different hardware resources on a chip related to power. By keeping track of the number of references to a certain resource, we can determine if the resource needs to be powered on or not.

For a resource to be tracked it needs to be registered with the power manager. The power manager then controls all registered resources through a resource manager. The resource manager knows the hardware specific details of how to enable/disable the resources.

Resources are arbitrary objects identified using an integer identifier. This allows flexibility in terms of resources as they differ between hardware platforms. A resource management facility is supplied to allow further flexibility in terms of the type of resource. Another way would be to supply a type of resource to the power manager, and then allow each resource to implement a trait which would enable or disable that specific resource. The required boilerplate to implement this would, however, become tedious and cumbersome to developers, and further increase the difficulty of supporting new platforms with low energy consumption.

Listing 5.5 shows an example which shows how a set of resources can be controlled through the power manager. A peripheral that wants to use a certain resource simply requests the resource through the power manager and releases it once it is done. This is can be seen in Listing 5.6.

Note that it is possible to request a resource multiple times and never release it. This was a choice to avoid overhead and try to keep the power manager unaware of the requester.

Listing 5.5: Example showing how we instantiate a *power manager* with a *resource manager*. The power manager keeps track of when resources are needed and then calls the functions defined in the resource manager to turn them on and off. In order to track the usage of resources, they first need to be registered with the power manager. This can be seen in the *init* function.

```
// Requests for resources (regions) will go through this power manager
pub static mut PM: PowerManager<RegionManager> =
    PowerManager::new(RegionManager);

pub struct RegionManager;

impl ResourceManager for RegionManager {
   fn enable_resource(&self, resource_id: u32) {
     // Enable the resource of identifier resource_id
   }

   fn disable_resource(&self, resource_id: u32) {
     // Disable the resource of identifier resource_id
   }
}

// Registers all resources we want the power manager to keep track off.
pub unsafe fn init() {
   for resource in RESOURCES.iter() {
       PM.register_resource(&resource);
   }
}
```

Listing 5.6: An example of how a peripheral might request a certain resource to do some work and then release it once the work is done.

```
PM.request_resource(power_region_id);
// Do some work which requires the power region to be on.
PM.release_resource(power_region_id);
```

5.3.2 Peripheral manager

The peripheral manager is a way for peripherals to get notified during transitions between power modes, and to decide which transitions are possible in order for peripherals to retain their functionality. Peripherals sometimes need to save and restore

their state, or disable and enable their features, before and after sleep. To enable this, we provide a way for peripherals to register themselves, each implementing a Rust trait to enable the peripheral manager to invoke them when needed.

There are three functions each peripheral must implement in order to be able to interact with the peripheral manager:

lowest sleep-mode
: Returns the lowest possible power mode the peripheral can transition to right know.

before sleep
: Gets called before going into sleep in order to prepare the peripheral for the transition.

after wakeup
: Re-initializes the peripheral after waking up from sleep mode.

There are often an arbitrary number of peripherals available, as peripherals can be attached and removed manually by users. There would need to be a way to register new peripherals on the fly to be able to keep track of every peripheral. This is why each peripheral needs to be manually registered for the peripheral manager to be aware of their existence. In order to register peripherals, they need to be statically allocated and appended to a linked list upon board configuration. The memory for the linked list is reserved during compile-time, which ensures safe operation during run-time.

Peripherals once registered can never be unregistered. The ability to unregister peripherals would require dynamic memory management. Tock does not support dynamic management of static memory, as it is considered unsafe and prone to memory leaks if not handled carefully. It is neither possible to remove members of the linked list, as the links are formed by immutable fields, and changing the structure may result in undefined behaviour.

5.3.3 Sleep/Power modes

Sleep/power modes are used during inactivity to reduce the power consumption. An IoT device often spends a great amount of its lifetime waiting for something, which makes inactive periods very important in terms of energy efficiency.

The device should transition into the deepest sleep mode whenever possible. This can, however, interfere with the functionality of the device. To prevent transition when an asynchronous operation is pending (i.e. sensor reading in progress), the peripheral manager described in Section 5.3.2 is used.

The setup of different sleep modes differ between hardware platforms, as they provide different features in terms of energy efficiency. One common denominator for the Cortex-based architecture of MCUs is that, in order to transition into efficient deep sleep, an indicator bit has to be set in the system control block that allows the MCU to disable certain features when waiting for interrupts.

An example of how the transition into sleep mode might look like is presented in Listing 5.7. It begins by a query to the peripheral manager to get the lowest sleep

mode possible. Depending on the returned value, it either prepares and transitions into deep sleep, or simply waits for interrupts until it continues.

Listing 5.7: An example of how a sleep routine may look like, invoked to reduce the energy consumption during inactivity. We retrieve the lowest possible sleep mode from the PeripheralManager, and then transition into the appropriate sleep mode - and do not interfere with the peripherals functionality or pending operations.

```
// Get the lowest possible sleep mode
let sleep_mode: SleepMode = SleepMode::from(peripherals::M.lowest_sleep_mode());

match sleep_mode {
    // DeepSleep is defined to be the lowest possible
    SleepMode::DeepSleep => {
        // We need to prepare peripherals for deep sleep
        peripherals::M.before_sleep(sleep_mode as u32);

        // Enable deep sleep & disable certain services during this period
        // This also sets the deepsleep bit in the system control block
        power::prepare_deep_sleep();

        // Transition into deep sleep by invoking the WFI
        // (Wait-For-Interrupt) instruction
        support::wfi()

        power::prepare_wakeup();

        // Peripherals might need to setup in order to
        // properly function again
        peripherals::M.after_wakeup(sleep_mode as u32);
    },

    // Do not transition into deep sleep (no setup), just
    // wait for next interrupt
    _ => support::wfi(),
}
```

5.3.4 Configuring the Sensortag for low-power consumption

For the lowest power settings, everything on the board needs to be powered off according to the sleep modes described in Section 4.3.2. However, this configuration only applies to the MCU, but the rest of the board also needs to be put in a low power mode. Mainly this means turning off external sensors and configuring the GPIO pins in a mode that does not allow any leakage currents to occur.

Of the sensors listed in Table 2.1, only the temperature sensor (TMP007) is on by default. The other sensors starts up in a low power mode and needs to be initialized before they power on and can be used. For low power consumption, the temperature sensor has to be powered off manually. The communication with the sensors happens through the I^2C-bus. Usually, a single command to a configuration register is everything that needs to be written in order to disable a sensor.

In general, all GPIO pins need to be configured in a way that prevents leakage currents. By default, all pins are configured in a low-leakage mode that leaves them floating with both input and output disabled. Some pins have external pull up or pull down resistors connected to them (see the Sensortag schematic [31]) and in those cases the pins need to be configured to match the pull to avoid voltage differences.

The Sensortag also has an external flash memory that can draw excessive current if it is not configured properly. In our case, this means simply pulling the SPI-pins (used for communication with external devices) low since they are currently not used.

5.3.5 Power saving features

The CC26xx family of MCUs has a wide variety of features in order to save energy. Mainly, there are four different ways to control the power consumption and they are organized into a hierarchy where each power saving feature includes or depends on the levels below. An overview of the topology can be seen in Figure 5.1.

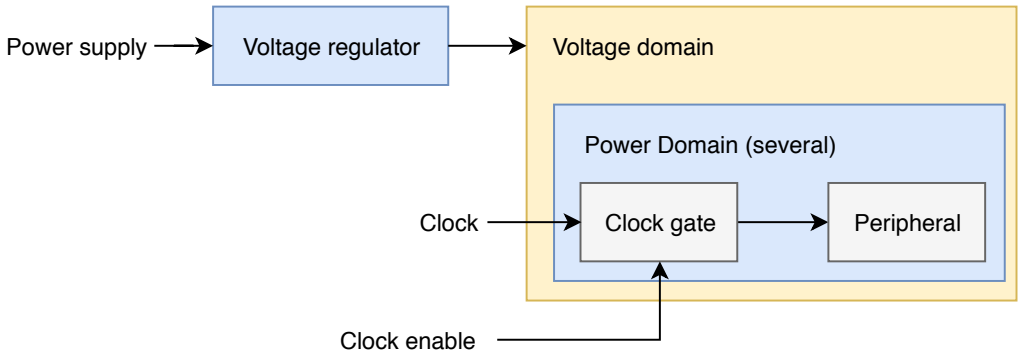

Figure 5.1: The figure shows the hierarchy of different power saving features in the CC26xx family of MCUs. The simplest feature is to toggle the peripheral clocks. Several peripheral clocks are in turn controlled by a power domain. Power domains are grouped into voltage domains and lastly, we have the voltage regulator that controls the power to the whole board.

Following is a short description of each power saving feature. In general, they are all about turning off parts of the chips that are not in use to conserve energy.

Clock gating
Prunes the clock tree by disabling the flip-flops controlling a specific peripheral.

Power domains
Each power domain powers the logic for a certain type of peripherals. For

example, turning off the *Serial* power domain will stop the UART and the I^2C bus. Turning a power domain off overrides the clock gating for all the peripherals found in the domain.

Voltage domains
Includes several power domains and regions otherwise considered *always-on*.

Voltage regulator
Turning off the voltage regulator provides the lowest power configuration. The chip reboots on power up and all peripherals need to be re-initialized.

The higher we go in the hierarchy the more energy we save, but the power cycling also takes longer time. For example, toggling the peripheral clocks are almost instantaneous while waking up from turning off the voltage regulator requires almost a whole millisecond [32]. This is something that needs to be taken into consideration if the board is in a lower power mode and needs to wake up in order to meet a scheduled event.

6
Evaluation

This chapter shows the results of our work and compares them to other operating systems ported to the same hardware platform. We also describe the setup for measuring the power consumption and the different benchmarks.

6.1 Evaluation setup

The evaluation consists of comparing the energy consumption and overall efficiency in Tock with other state-of-the-art operating systems for IoT devices. More precisely we, compare Tock with Contiki [2] and TI-RTOS [19] on the Sensortag. These are both operating systems which share the following properties with Tock:

- Support for the Sensortag platform
- Source code available
- BLE support
- Low power configurations for improved energy efficiency

The source code is important in order to compare the different implementations with each other. Support for BLE and low power configurations are relevant because of our work to support these features in Tock for the Sensortag.

6.1.1 Energy efficiency

Measuring the power consumption on a device is hard since it varies over time and heavily depends on what task the device is performing. IoT-devices could constantly be monitoring a sensor, or alternatively, be configured to only do sensor readings periodically. The power requirements differ between the two cases and devices probably use different techniques to save energy depending on how often they need to wake up to perform something. To cover most use-cases, we evaluate three different scenarios:

Constant/Frequent readings
: To cover the use-case of constant, or very high-frequent readings, we use BLE advertisements. It is common for IoT devices to broadcast data over BLE (e.g. sensor readings), and we want to evaluate the usage of BLE in conjunction with implemented energy improvements.

Intermittent readings

To cover the use case of intermittent readings, we use an application that intermittently toggles an LED on the Sensortag board. Toggling the LED is meant to represent a sensor reading.

Infrequent readings

Some IoT devices are inactive for long periods of time, which makes them suitable for very low sleep modes. In these cases, the actual jobs performed between inactivity have very little impact on the lifetime due to how infrequent they are. To evaluate this use-case, we investigate how low the power consumption can be while still being able to wake up.

6.1.2 BLE

To evaluate the BLE implementation, a way to detect and inspect raw BLE packets is needed. In our case, we use an nRF52 [33] board configured as a BLE sniffer.

The BLE sniffer is used in conjunction with the measurement setup described in Section 6.1.3 to view the energy consumption during and between BLE operations. This is done to evaluate how our energy efficiency improvements work in conjunction with regular functionality of the device (e.g. advertising its existence via BLE).

6.1.3 Measurement setup

In order to measure the power consumption, a shunt resistor is connected in series with the board and the power supply, Figure 6.1 depicts the schematic of the setup. By measuring the voltage V_R over the shunt resistor, we can derive the current using Ohm's law (Equation 6.1). The benefit of using an oscilloscope to measure the voltage is that we can see how the power consumption varies over time.

$$I_{sensortag} = \frac{V_R}{10\Omega} \tag{6.1}$$

The measurements include external sensors and peripherals rather than just the MCU. Measuring over the whole board is easier than singling out individual components and it gives a more realistic estimate of Tock's energy-efficiency. The ability of Tock to communicate with external components is crucial for minimizing leakage currents and something that needs to be included in our evaluation.

The oscilloscope of choice was the Rigol DS1054Z [34] which has a *Roll mode* that enables us to observe slow signals in real-time without having to wait for the wave-form to complete. To reduce the effects of random noise on the waveform, the signal was averaged with the *High-resolution* acquisition mode. Another benefit of using a digital oscilloscope is that the waveform can be frozen and saved to an external storage device for later analysis.

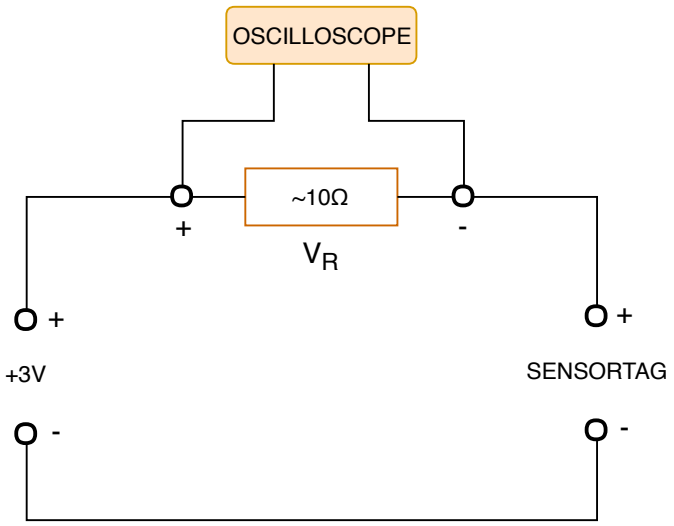

Figure 6.1: Schematic of the measuring setup. An oscilloscope measures the voltage over a resistor in series with the board. Derive the current by using Ohms law: $I_{sensortag} = \frac{V_R}{10\Omega}$, where $I_{sensortag}$ denotes the current through the SensorTag.

6.2 Results

This section explains the benchmarks in greater detail and shows the results from running Tock on the Sensortag platform.

A comparison is made between the different operating systems supported on the platform in terms of standby power consumption. The operating systems perform with the same energy efficiency during the active and idle power modes, which makes the standby power consumption more interesting.

We begin with the results for the power consumption during inactivity in Section 6.2.1. Then, we continue with measurements and results for Blink and BLE advertising in Sections 6.2.3 and 6.2.4 respectively. Following the results of our benchmark applications, Section 6.2.5 gives an estimation of the energy efficiency on the Sensortag board, presenting an estimated lifetime of the device.

6.2.1 Power consumption during inactivity

Power consumption during inactivity is important and has the possibility to extend the lifetime of IoT devices greatly if reduced. Measurements of the power consumption on the Sensortag during inactivity are shown in Figure 6.2. The voltage over the board is constant which makes the power consumption directly proportional to the current draw ($P = U \times I$). This is why the power consumption is shown as current draw in all figures in this chapter.

Tock seems to have better energy-efficiency than both Contiki and TI-RTOS. This is surprising since all energy savings are done by turning off hardware features and they all run on the same board. The reason for this is probably that the other operating systems support more features on the Sensortag that might need to stay on during sleep to be used after wakeup.

Both Contiki and Tock outshine the default TI-RTOS implementation since they

configure the GPIO pins according to the attached peripherals by default - something TI-RTOS does not. TI-RTOS has to have the pins manually configured in order to get the results depicted in Figure 6.2.

The measurements are performed with a simple application which yields as soon as possible, without doing any work. It is also worth pointing out that the measurements are done with the MCU in standby mode (Section 4.3.2). There is an even lower power mode called shutdown. Putting the device in shutdown mode is not used in Contiki by default and our implementation does not yet have support for it. TI-RTOS might have an advantage here since the board can be allowed to enter shutdown by just defining a different power policy [35]. The savings gained from entering shutdown instead of standby [32, p. 38] are so small that it might not be worth the trouble.

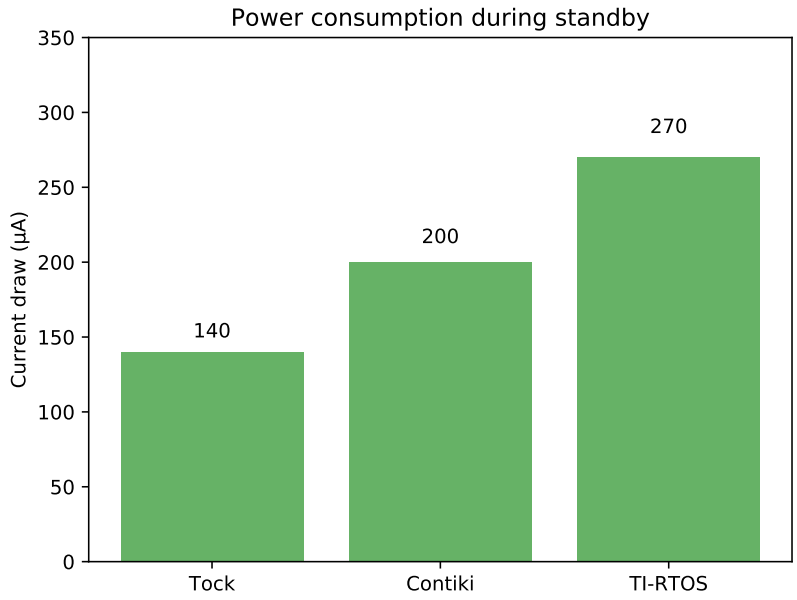

Figure 6.2: Power consumption in Tock, Contiki, and TI-RTOS during standby. The operating systems use different defaults which yield different current draw in standby.

6.2.2 Tock power consumption

Measuring and comparing the power consumption in different power modes illustrates the potential energy savings that can be made during inactivity. Figure 6.3 shows the difference between the power modes available on the Sensortag.

It is clear that standby mode should be used as much as possible to extend the lifetime of the device. However, the disadvantage of using standby is that the wakeup time gets noticeable longer. This is illustrated in Figure 6.4 where the wakeup time for Blink is measured when waking up from idle and standby. The extra delay needs to be taken into account when working with strict timing-constraints.

There is also a noticeable current surge (about 6 mA) that occurs when waking up from standby. If wakeups are too frequent, this surge might lead to increased power consumption. This is not specific to Tock, but something that occurs for all operating systems running on the Sensortag.

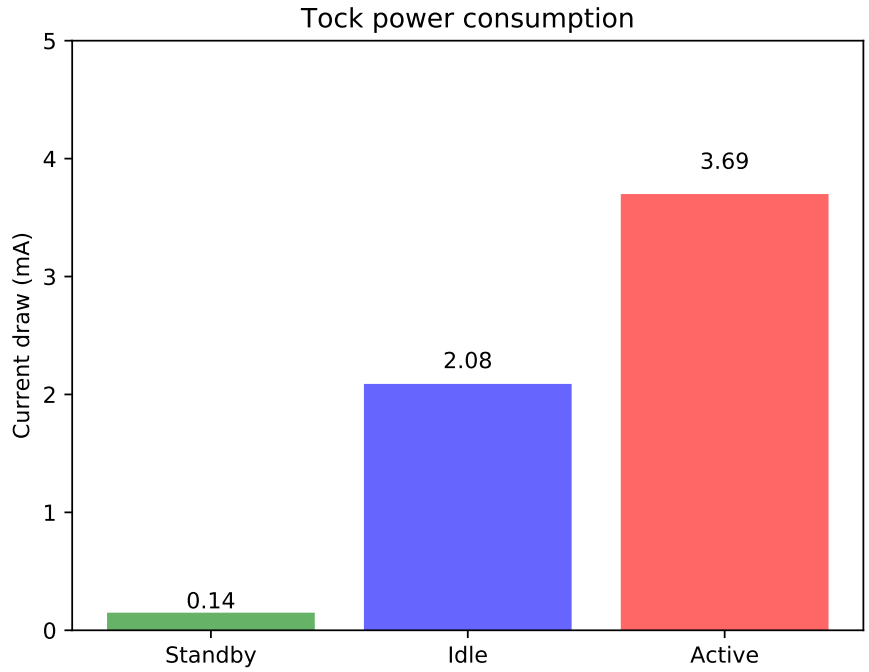

Figure 6.3: Power consumption in different power modes, when running Tock on the Sensortag.

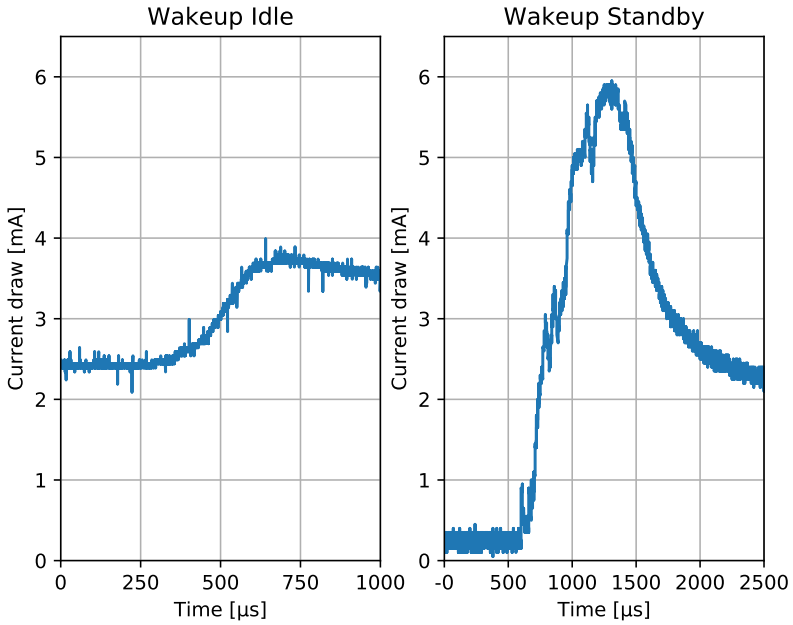

Figure 6.4: Comparison of Tock wakeup times between idle and standby. The wakeup time was measured with the Blink application when waking up to toggle the LED on.

6.2.3 Blink

Blink is the simplest application of the benchmarks. It toggles an LED on the platform at a certain interval. The benchmark shows that we properly can enter sleep mode and then wake up without any issues. Before the transition into sleep, we enable an IO-latch on the platform, which freezes the state of all GPIO pins in order to retain them until we wake up. This is why the LED can still be powered on in sleep mode.

A measurement of the power consumption can be seen in Figure 6.5. The device wakes up to toggle the LED and then goes back to sleep as soon as possible for a predetermined time of one second. What we see in the figure is the pulse of the LED, either powered on or off (low edge represents LED off, high edge LED on) during sleep. The brief window of activity is hard to notice in the figure, as the device only performs a single syscall before going back to sleep. During sleep, the device is power-cycled in order to retain its volatile memory (i.e. RAM). The recharge can be seen as periodic spikes in Figure 6.5.

The Blink application is proof that Tock works as expected on the Sensortag, and that the porting of Tock to the new platform was successful.

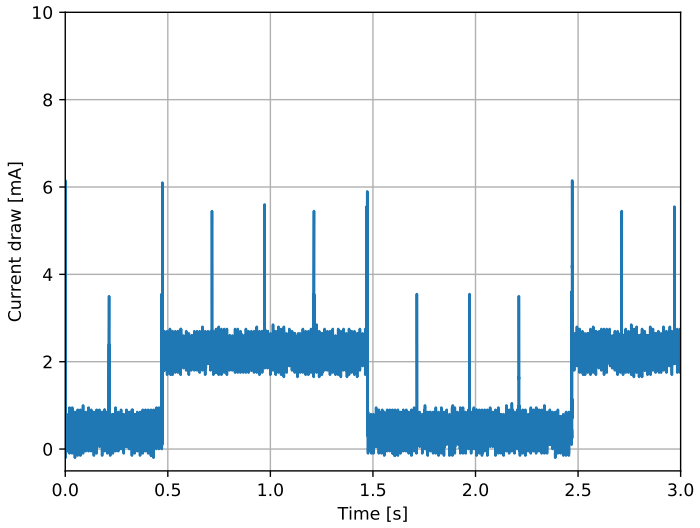

Figure 6.5: Toggling the red LED on the Sensortag. The LED is toggled every two seconds and the board enters sleep mode once the state of the LED has changed. There is some noise due to electrical disturbance, as the current is very low.

6.2.4 BLE

The radio is arguably the most sophisticated peripheral on the Sensortag. The BLE benchmark shows that our implementation of sleep modes can handle more complex scenarios. The radio is more complex in the sense that it depends on more hardware resources and has stricter timing constraints.

BLE advertisements increase the power consumption drastically. Wireless communication requires a burst of electricity in order to transmit over the air. A BLE advertisement round consists of three separate transmissions on three separate radio channels, which are specifically dedicated for advertisements in the BLE protocol.

A typical BLE advertisement round in Tock can be seen in Figure 6.6, where each peak represents one transmission on a specific channel. From the figure, we can see that the peak current draw is about 18 mA during transmissions.

The energy efficiency of BLE heavily relies upon the frequency of advertisement rounds, or transmission rounds. Gomez et al. [36] state that it could impact the lifetime of a device in terms of years and that one should be careful when adjusting this frequency. Tock has a frequency of 300 ms in its BLE advertising sample and allows this to be dynamically changed during advertisements.

The power consumption during an advertisement round is almost identical on all tested platforms. The main difference between the platforms is the idle power consumption between rounds, which is described in further detail in Section 6.2.1.

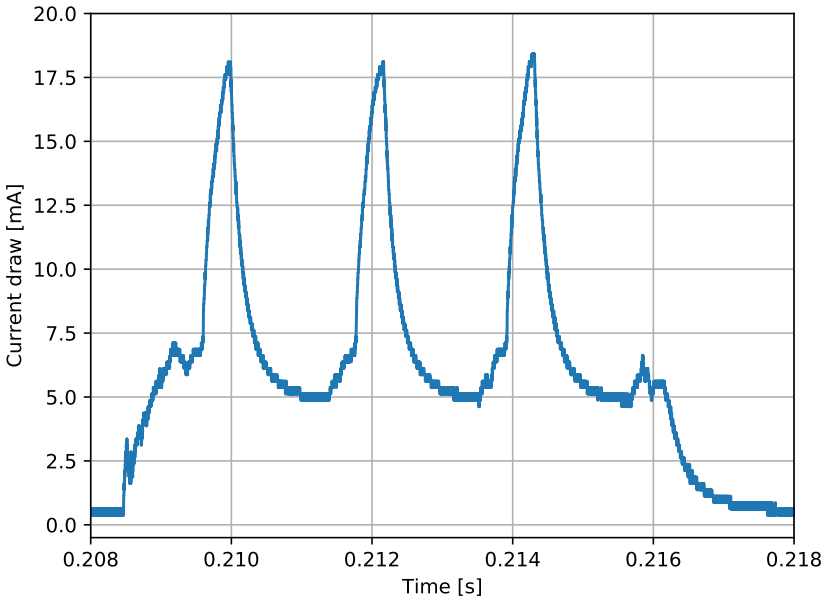

Figure 6.6: Oscilloscope trace of one BLE advertising round. The three peaks correspond to three transmissions on three different channels.

6.2.5 Expected lifetime

An estimation of the expected lifetime can be done using a theoretical model depending on the use cases mentioned in Section 6.1.1. The model is divided into two parts: active and inactive. The time periods for the active and inactive part are modified depending on the use case, and the current draw is taken from the measurements performed in Section 6.2.3 and 6.2.4. An expected lifetime is then calculated using Equation 6.3.

$$I_{average} = \frac{(I_{active} \times T_{active}) + (I_{inactive} \times T_{inactive})}{T} \tag{6.2}$$

$$Lifetime = \frac{BatteryCapacity}{I_{average}} \tag{6.3}$$

The battery capacity used is 240 mAh, which is the same capacity as the CR2032 battery shipped with the Sensortag.

Table 6.1: Expected lifetime in different use cases of a Sensortag device running Tock with our modifications.

Description	T_{active} (s)	I_{active} (mA)	$T_{inactive}$ (s)	$I_{inactive}$ (μA)	**Lifetime** (days)
BLE, frequent	20	18	40	80	1.65
Blink, intermittent	4	4	56	80	29.30
Blink, infrequent	0.2	4	59.8	80	107.44

As can be seen in Table 6.1, the expected lifetime drastically increases during large periods of inactivity. This depends on the fact that Tock has a very small amount of current draw during these periods. Active periods draw a lot of current, and it is up to each user to design their applications with this in mind.

This does not mean that BLE is energy-inefficient in any way, and it was only used in the frequent sample since we had practical values of the current draw of a transmission over BLE. As mentioned in Section 6.2.4 the energy efficiency of BLE is highly affected by the frequency of transmissions, and there are many tricks one can use to improve this [37, 38].

6.2.6 Abstraction overhead

The abstractions introduced to handle resource management (Section 4.3.1) and peripherals (Section 4.3.3), introduce extra overhead in Tock when used. This means that one should be wary when using these abstractions since they both affect the wakeup time and memory consumption.

6.2.6.1 Power Manager

The space complexity of the power manager is proportional to the number of resources. Let R denote the resources for a specific platform, the space complexity of the power manager would thus be $O(|R|)$. This can become a problem since resources can be almost anything on the platform (e.g. clock gates, peripherals, power regions, etc.). The power manager yields a flexible and platform independent way to manage resources, but it comes at the cost of memory. IoT devices often have limited memory, and the OS running should not consume a majority of it.

As described in Section 5.3.1, a unique integer identifier is used for each resource. When requesting or releasing a specific resource, there is a need to find the reference counter paired with the resource by iterating over all resources to find a match. This means that the worst-case lower bound on the request/release time complexity is $O(|R|)$. This presents a scalability issue as there is no bound on the number of resources.

6.2.6.2 Peripheral Manager

The space complexity of the peripheral manager is proportional to the number of peripherals attached to the platform. Let P denote the peripherals for a specific platform, the space complexity would thus be $O(|P|)$. IoT platforms generally do not have a large number of peripherals, which limits the upper bound of the space complexity. However, using the peripheral manager would increase the memory consumption proportionally to the number of peripherals on the platform.

The peripheral manager has an impact on the wakeup time of the device, as described in Section 5.3.2. The time complexity of getting the lowest power mode, and notifying peripherals upon transitions, would both be $O(|P|)$ since there is a need to poll all peripherals.

7
Discussion

This chapter reflects our general thoughts about Rust and Tock together with the key insights gained from porting Tock to a new platform and improving its energy-efficiency.

The chapter starts with a discussion in Section 7.1 about Rust and its use in embedded systems. Section 7.2 follows with a discussion about sleep modes and what needs to be done in order to transition to them. Section 7.3 is concerned with the design of Tock and the flexibility of its components. Finally, Section 7.4 explains why the thesis includes two hardware platforms rather than just one.

7.1 Rust

Rust provides a certain set of features that makes it very attractive for operating systems. Its strict memory management avoids problems related to pointers in C-like languages such as buffer overflows and memory leaks. This is crucial for safety-critical systems that needs to be up and running for long periods of time. Rust's approach to memory management together with its type system makes it possible to write a safe kernel where only a small part of the code needs to be trusted [1].

When relying on software for security checks, there is always the question of performance. Rust makes most memory checks at compile time which avoids a lot of overhead during runtime. This means that Rust still retains reasonable overhead thanks to its strict compiler and the lack of a garbage collector.

Rust is a relatively new language and its userbase is quite small. For newcomers to the language, the concept of lifetimes and ownership (Section 2.4) can seem intimidating at first, and the learning curve is steep. Rust requires some getting used to if you come from other object oriented languages such as C or C++. The benefits of using Rust are obvious, but it will take some time to fully appreciate its features.

Writing Rust code for an embedded system provides an additional challenge: an operating system interacts directly with the hardware through its device drivers, and this often requires direct memory access. This means that we occasionally have to leave the safety features behind and use unsafe Rust code. This code is more similar to C and should be kept to a minimum if one wants to leverage the advantages of the language. Finding good abstractions that hides the unsafe code behind safe interfaces, is one of the main challenges when developing an operating system in Rust.

The work of writing hardware specific modules includes a lot of unsafe Rust code

because of the memory mapped registers for all peripherals. This means that the porting process in our case revolve around writing very C-like code to be able to interact with these memory areas. There is, however, ways to mitigate this which is described in a pull request [28], and Section 7.3.2.

7.2 Sleep modes

Sleep modes are a way for IoT-devices to lower their power consumption when the MCU is idle or waiting for an event. It is generally easy to put a device in sleep mode, but there are several factors that need to be taken into consideration before doing so. The CC26xx family of MCUs supports several different power modes. One complex issue is deciding what sleep mode we can enter without interrupting any ongoing activity. Different sleep modes keep different parts of the hardware powered. Therefore, we need to ask all peripherals what power mode they support at a given moment. This process can be somewhat simplified by the peripheral manager described in Section 5.3.2.

The decision of what sleep mode to use is, however, more complex than just looking at what peripherals are currently being powered. There is also several timing constraints that need to be taken into consideration. The lower we go, the more time it will take to wake up once there is something to do. If the radio broadcasts BLE advertisements at a certain interval, we need to wake up in time to not delay the advertisement. The reference manual for the MCU normally contains information about the time it takes to wake up from a certain state, but on a full board, we also need take external peripherals into account.

Adding support for new peripherals which are dependent on other peripherals or system resources often means we have to modify several drivers to still support all peripherals during sleep. In this thesis, we have tried to create several software constructs that deal with this problem. Our intent is to keep them general enough to reside in the kernel with the potential to be used by MCUs outside the CC26xx-family. Taking timing-constraints in mind, further functionality needs to be added that makes use of timers to see when peripherals need to be awakened to decide if a certain sleep mode is safe to enter in order to meet all deadlines.

There is a trade-off between energy-savings and complexity. Contiki and TI-RTOS keep things simple by mainly using two different sleep modes. The fist mode is just to run the WFI-instruction to prevent the CPU from busy-waiting. The other mode, *deep-sleep*, performs a more sophisticated configuration of the hardware before entering sleep. Even in this mode, some power features is always kept on since they are almost always being used or turning them off would require extensive configuration when waking up.

Very fine-grained power management might be cumbersome and not provide that much of an improvement to be worth the trouble. Be it smart-watches or sensor-nodes, IoT-devices seldom need to be fully powered all the time. The degree of power management that is needed depends on the lifetime expectancy of the device and the frequency of different events.

7.2.1 Transition responsibility

It is important to consider the responsibility of when to perform transitions between power modes, and where the logic for this should reside. There are three different alternatives: userland, kernel, or a combination of both. All alternatives pose problems in their own way and impose extra complexity into the operating system.

Several applications should be able to run on the system without interfering with each other. Giving the responsibility of transitions to userland gives applications a lot of freedom, but might be problematic if several applications have to get along. Resources used by other applications must be taken into consideration before transitioning to not disrupt their execution. Backwards compatibility can also be an issue if different applications are built with different versions of the same userland library.

Giving the responsibility of transitions to the kernel makes them seamless to all applications, but increase the overall complexity of the kernel. The kernel would inspect peripherals and their usage in order to determine if a transition should occur or not. User applications would no longer be able to directly affect the transition between sleep modes.

A compromise would be to share the responsibility between userland and kernel by adding an extra syscall API. This would increase the complexity of both applications and the kernel, and both applications and peripherals would have to be taken into consideration by the kernel when performing transitions. Applications would be able to prevent transitions and have more control over them. The kernel would be far more complex, as more conflicts would arise between applications and peripherals. This approach might not be beneficial in all situations, but it is the most flexible approach without compromising reliability.

7.3 Tock

This section addresses Tock's architecture and discusses how easy it is to adapt to a new hardware platform. The discussion will explain how Rust's strengths are emphasized while keeping the code safe and modular.

7.3.1 Architecture

The architecture of Tock is interesting in its design, in the sense that it revolves heavily around safe memory management which it achieves with Rust's safety mechanisms. The downside with these safety mechanisms is that all resources have to be statically allocated during compilation. This means that all capsules used during compilation will keep this memory during runtime and never be released. In other words, this feature provides memory safety at the cost of increased memory usage and flexibility of how memory is used during runtime since capsules which might not be used are still allocated.

7.3.2 Abstractions over unsafe code

In order to perform low level operations in Rust, it is necessary to use unsafe code. This code can do unsafe operations such as raw manipulation of memory, which is necessary in order to communicate with the hardware through memory-mapped registers. The usage of unsafe code means that Rust is stripped of its safety features, which should be avoided unless absolutely necessary.

Tock tries to minimize its usage of unsafe Rust to provide a safe environment, but it cannot be removed completely. It is desirable to hide the unsafe implementation and add abstractions which use unsafe code that is rigorously tested and verified. Using these verified abstractions allow the overlaying logic to exploit Rust's safety mechanisms. An example is the *Cell* wrapper from Rusts standard library [39], which is an extension to allow interior mutability [40] using unsafe code. This allows a hidden mutable field within safe structures, to allow inherently immutable structures to mutate their fields.

7.3.3 Portability

Adding support for a new platform in Tock is only as complex as the new platform. As more platforms get support, the easier it will be to add another, since many embedded platforms share properties and implementation details. The architecture of Tock divides porting into three parts:

Capsules are entirely platform independent and should be designed with independence in mind. When porting, it is sometimes necessary to add new capsules to support new types of hardware, or to extend old capsules to be more general.

The HAL implements hardware specific features needed by the capsules. HAL modules are specific to a certain MCU family but should be configurable to support several boards.

Boards are entirely platform dependent, and should not share implementation details with other boards. Both the HAL and capsules are tied together in the board configuration, and this is the main entry-point and setup for the kernel.

The abstractions are well designed and highly flexible. There are however some assumptions Tock has made that makes it incompatible with some platforms. One example of such an assumption is the existence of an MPU. The operating system still works without one, but applications will not be completely sandboxed from each other.

7.4 Choice of platform

The Sensortag uses a Cortex-M3 based MCU, which lacks an MPU as required by Tock. For this reason, we also port the operating system to an early evaluation board from TI which uses an almost identical MCU, but with an added MPU. However, the evaluation board lacks much of the appeal of the Sensortag since it does not have

any external sensors. To solve this, we keep our code compatible with both boards to still be able to upstream our work and get feedback from the core-developers. The ideas is to keep doing this until an updated version of the Sensortag is released.

8

Conclusion

This chapter summarizes our findings and keeps a broader discussion about the subject in general and what the future might hold for Tock.

The IoT is growing and its growing fast. The variety of different devices gaining connectivity are constantly increasing. Embedded systems are often inaccessible once in production and the sensitive data these devices handle need to be well protected. Tock handles these demands by providing a flexible platform, with inherent safety and reliability features. Thanks to Rust, it provides a safe environment for applications.

Tock has an active community with rapid development cycles. The safety features and portability is two major features that makes it easy to start using Tock and develop applications for it. However, Tock is still relatively new and all pieces are not in place yet. Tock does not have a complete network stack and power management is still in its early stages.

Tock has begun to shift more of its focus towards energy efficiency [29]. Tock must find a general way to manage power consumption to keep its flexibility and increase the incitement to port it to other platforms. Good energy-efficiency is crucial for IoT-devices and a feature that must be supported by the kernel so that the application developers can focus on other parts of their program.

In this thesis, we contribute to the open source community of Tock by adding support to two new platforms, and new hardware modules for their peripherals. The platforms offer new opportunities in terms of energy efficiency, which we take advantage of in our implementation. We show that by successfully adding support for two new platforms, the abstraction level of Tock is in its favor and easy to port to new platforms, which answers our first research question. We also increase the energy efficiency by providing new ways for Tock to manage peripherals and resources, and further decrease the power consumption by utilizing lower power modes. We show that our implementation holds up well against other state-of-the-art embedded operating systems in terms of energy efficiency, which answers our second research question: that an operating system implemented in Rust compares well.

Tock is designed with safety and reliability in mind. It uses the inherent safety features of Rust to provide a safe environment. This is however not always in its favor and may increase the complexity of adding support for new platforms by embedding assumptions at higher abstraction layers. The current flexibility provided by the abstractions and the assumptions made did not hinder us in our work, and the hurdles proved to be necessary. We argue that the trade-offs in assumptions Tock has made is justified, and perhaps necessary to achieve the desired safety in

an embedded operating system.

Tock is still new to the embedded scene of operating systems, which is reflected by how its design changes occasionally. Research is being made on how to utilize the flexibility and safety of Rust to improve Tock even further. However, Rust is a new language with a steep learning curve, and adoption comes at a slower rate because of this.

Bibliography

[1] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis, "The Case for Writing a Kernel in Rust," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, (New York, NY, USA), pp. 1:1–1:7, ACM, 2017.

[2] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, Nov 2004.

[3] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, *et al.*, "TinyOS: An operating system for sensor networks," *Ambient intelligence*, vol. 35, pp. 115–148, 2005.

[4] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, "Multiprogramming a 64kB Computer Safely and Efficiently," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), pp. 234–251, ACM, 2017.

[5] T. Instruments, "The SensorTag Story - IoT made easy." `http://www.ti.com/ww/en/wireless_connectivity/sensortag/`, Visited: Oct. 2017.

[6] "CC2650 Technical Reference Manual." `http://www.ti.com/lit/pdf/swcu117`, Visited: Feb. 2018.

[7] "References and Borrowing - The Rust Programming Language." `https://doc.rust-lang.org/1.9.0/book/references-and-borrowing.html`, Visited: Feb. 2018.

[8] "Lifetimes - The Rust Programming Language." `https://doc.rust-lang.org/1.9.0/book/lifetimes.html`, Visited: Feb. 2018.

[9] "borrowck is unsound in the presence of ''static mut's 27616." `https://github.com/rust-lang/rust/issues/27616`, Visited: Feb. 2018.

[10] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto, "Ownership is Theft: Experiences Building an Embedded OS in Rust," in *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, PLOS '15, (New York, NY, USA), pp. 21–26, ACM, 2015.

[11] "TockOS Website." `https://www.tockos.org`, Visited: Mar. 2018.

[12] "TockOS Userland Source code." https://github.com/helena-project/tock/tree/ff133577d0326c8283fcc0d8f3159cbfc8fcf23f/userland, Visited: Mar. 2018.

[13] V. Shnayder, M. Hempstead, B.-r. Chen, G. W. Allen, and M. Welsh, "Simulating the Power Consumption of Large-scale Sensor Network Applications," in *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, (New York, NY, USA), pp. 188–200, ACM, 2004.

[14] X. Jiang, P. Dutta, D. Culler, and I. Stoica, "Micro Power Meter for Energy Monitoring of Wireless Sensor Networks at Scale," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, (New York, NY, USA), pp. 186–195, ACM, 2007.

[15] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He, "Software-based On-line Energy Estimation for Sensor Nodes," in *Proceedings of the 4th Workshop on Embedded Networked Sensors*, EmNets '07, (New York, NY, USA), pp. 28–32, ACM, 2007.

[16] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, Nov 2004.

[17] "Contiki Multithreading Library." https://github.com/contiki-os/contiki/wiki/Multithreading, Visited: May 2018.

[18] "Contiki: Bringing IP to Sensor Networks." https://ercim-news.ercim.eu/en76/rd/contiki-bringing-ip-to-sensor-networks, Visited: May 2018.

[19] "TI-RTOS." http://processors.wiki.ti.com/index.php/TI-RTOS, Visited: Mar. 2018.

[20] "RTOS power management: Essential for connected MCU-based IoT nodes." http://www.ti.com/lit/pdf/spry282, Visited: Mar. 2018.

[21] C. Xu, F. X. Lin, Y. Wang, and L. Zhong, "Automated os-level device runtime power management," *SIGARCH Comput. Archit. News*, vol. 43, pp. 239–252, Mar. 2015.

[22] K. Popovici and A. Jerraya, *Hardware Abstraction Layer*, pp. 67–94. Dordrecht: Springer Netherlands, 2009.

[23] V. Handziski, J. Polastre, J. H. Hauer, C. Sharp, A. Wolisz, and D. Culler, "Flexible hardware abstraction for wireless sensor networks," in *Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.*, pp. 145–157, Jan 2005.

[24] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. Haskins, "A survey of configurable, component-based operating systems for embedded applications," *IEEE Micro*, vol. 21, pp. 54–68, May 2001.

[25] A. A. Jerraya and W. Wolf, "Hardware/software interface codesign for embedded systems," *Computer*, vol. 38, pp. 63–69, Feb 2005.

[26] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin, "Actor-Oriented Design of Embedded Hardware and Software Systems," *Journal of Circuits, Systems and Computers*, vol. 12, no. 03, pp. 231–260, 2003.

[27] J. Liedtke, "On Micro-kernel Construction," *SIGOPS Oper. Syst. Rev.*, vol. 29, pp. 237–250, Dec. 1995.

[28] "Tock Automatic peripheral clock management pull request." `https://github.com/tock/tock/pull/760`, Visited: May 2018.

[29] "TockOS Blog Post about Peripheral Management." `https://www.tockos.org/blog/2018/peripheral-management/`, Visited: May 2018.

[30] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis, "The Case for Writing a Kernel in Rust," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, (New York, NY, USA), pp. 1:1–1:7, ACM, 2017.

[31] "Sensortag Schematics." `www.ti.com/lit/df/swrr134c/swrr134c.pdf`, Visited: May 2018.

[32] "CC2650 Datasheet." `www.ti.com/lit/ds/symlink/cc2650.pdf`, Visited: May 2018.

[33] N. Semiconductor, "NRF52832/NRF52DK BLE Board - Nordic Semiconductor." `http://www.nordicsemi.com/eng/Products/Bluetooth-low-energy/nRF52832`, Visited: May 2018.

[34] "Rigol ds1000z Manual." `https://www.batronix.com/pdf/Rigol/UserGuide/DS1000Z_UserGuide_EN.pdf`, Visited: May 2018.

[35] "Power Management for CC26xx SimpleLink Wireless MCUs." `www.ti.com/lit/ug/sprui20/sprui20.pdf`, Visited: May 2018.

[36] C. Gomez, J. Oller, and J. Paradells, "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology," *Sensors*, vol. 12, no. 9, pp. 11734–11753, 2012.

[37] A. Nagy and O. Landsiedel, "Towards Energy Efficient, High-speed Communication in WSNs," in *ASCoMS: Proceedings of the Workshop on Architecting Safety in Collaborative Mobile Systems held in conjunction with the 33nd International Conference on Computer Safety, Reliability and Security (SafeComp)*, Sept. 2014.

[38] B. A. Nahas and O. Landsiedel, "Towards Low-Latency, Low-Power Wireless Networking under Interference," in *EWSN: Proceedings of the International Conference on Embedded Wireless Systems and Networks, Dependability Competition*, Feb. 2016.

[39] “Rust Standard Library Documentation - core::cell::Cell.” https://doc.rust-lang.org/core/cell/struct.Cell.html, Visited: May 2018.

[40] “Rust Documentation - Interior Mutability.” https://doc.rust-lang.org/reference/interior-mutability.html, Visited: May 2018.