Delft University of Technology

Faculty of Electrical Engineering, Mathematics & Computer Science
Department of Software Technology

An Architecture-Agnostic Memory Protection Interface for the Tock Operating System

Daniel Stefanus Maria Verhaert

Stanford University

An Architecture-Agnostic Memory Protection Interface for the Tock Operating System

Thesis

submitted in partial fulfillment of the requirements for the

Master of Science

degrees in

Electrical Engineering

&

Embedded Systems

by

Daniel Stefanus Maria Verhaert
born in Amsterdam, The Netherlands

to be defended publicly on October 19, 2018 at 15:30

Student number: 4234685

Submission date: October 12, 2018

Advisors: Prof. dr. P. Levis, Stanford University
Prof. dr. K. G. Langendoen, Delft University of Technology

Thesis Committee: Prof. dr. K. G. Langendoen (Chair), Delft University of Technology
Dr. J.S. Rellermeyer, Delft University of Technology
Dr. M. Nasri, Delft University of Technology

ISBN: 978-94-6366-094-5

An Architecture-Agnostic Memory Protection Interface for the Tock Operating System

Author: Daniel Stefanus Maria Verhaert
Email: d.s.m.verhaert@student.tudelft.nl

Abstract

Tock is an embedded operating system that can run multiple concurrent, mutually distrustful processes, concurrently. Tock is written in Rust, a novel system programming language enforcing type safety at compile-time, and takes advantage of Rust's strong safety features. However, since Tock allows user-level applications to be written in any language, Rust is not sufficient in guaranteeing memory safety for user-level Tock. To obtain memory isolation, Tock takes advantage of MPUs provided by recent microcontrollers. Although Tock is supposed to be fully architecture agnostic, it is at present only able to support the MPU of the Cortex-M architecture. The lack of an architecture-agnostic MPU interface, process manager and the corresponding MPU implementations is the biggest remaining hurdle in making Tock architecture independent. In order to create such an interface, this work performs an analysis of state-of-the-art MPUs, comparing their key features and constraints. A feasibility study for the design of an MPU interface in Tock is carried out, resulting in a number of general changes to the current implementation of Tock, and leading up to the design, implementation and evaluation of two MPU interfaces. The first is a region-based interface, aiming to have an abstraction that is straightforward and completely agnostic of what the MPU is used for. The second interface is a process-based interface, that utilizes knowledge of what it is applied for in order to provide a solution that is more efficient given the constraints and optimizations of an arbitrary MPU. Implementations for the Cortex-M, Kinetis K and nRF51 MPUs are created that match these interfaces, and in addition fundamental adaptations to process management in the Tock kernel are made. On the Hail development board, these changes reduce context switching time by 25.34%, but lead to a cost of 848 bytes or 1.0% in flash memory overhead. Most importantly, with the introduction of these interfaces, the biggest hurdle for Tock in becoming a multi-architecture operating system is overwon.

Preface

Up until a year ago, I could never have imagined myself writing an MSc thesis on memory protection or operating systems, let alone getting to do this at Stanford University. I have had the most amazing time in the last nine months, and am extremely grateful to have gotten this opportunity.

I would start by thanking my thesis supervisors that made this adventure possible. Phil, I am extremely grateful to you for extending this life-changing invitation. Thank you for for your useful input and guidance. Koen, thanks for organizing everything on the Delft side and guiding me through the process of writing an MSc thesis, all while calling with me through a nine hour time difference.

Everybody on the Tock team, you have all been magnificent in helping me getting started. Amit, you in particular have been a tremendously helpful guide to me. Thank you.

My special thanks goes out to Conor. From having productive discussions to being vultures ahead of the wake while scavenging through Gates for free food, my experience working with you has been a ton of fun.

I am very grateful towards the Justus & Louise van Effen Excellence Scholarship and the Stanford Information Networks Group for funding me throughout the nine months I spent in the San Francisco Bay Area. Without this support, it would have been difficult to have had this experience.

I would like to thank the friends and family who helped me through the process of writing my thesis. You were an incredible motivation for me, helping me shape this work into what it is now (and providing some much needed distraction once in a while).

Thank you all for your unwavering support.

Danilo Verhaert
Palo Alto, California
October 12, 2018

Contents

List of Figures

List of Tables

Acronyms

CPU central processing unit

FPGA field-programmable gate array

HIL hardware interface layer

IoT internet of things

IPC inter-process communication

ISA instruction set architecture

LIFO last in, first out

MMU memory management unit

MPU memory protection unit

OS operating system

PAM process accessible memory

PMP physical memory protection

RAM random access memory

RWX read, write and execute

TBF Tock binary format

TLB translation lookaside buffer

Chapter 1

Introduction

With the ever increasing number of connected devices [1], new capabilities are made possible through access of rich new information sources. This creates the internet of things (IoT): the network of interconnected devices, providing the ability to share information. Frequently, IoT systems are composed of *embedded systems*. Embedded systems are small devices with a dedicated function that are embedded as part of a larger system. Embedded systems often have *microcontrollers* at their core: small computers on a single chip. Embedded systems are characterized by their low cost and low power consumption in comparison to their general-purpose counterpart, at the price of limited resources [2]. These properties make embedded systems economically attractive, and the drop in costs in recent years is what has spurred the growth of the IoT by making it cost-effective. The drawback of being low power and low cost is that embedded systems have very limited resources, particularly in terms of RAM and code space, making them more difficult to interact with. To make the most out of this, embedded systems are historically written in system-level languages like C and C++. These languages provide little abstraction from a computer's architecture, and therefore introduce very little overhead in terms of memory usage and execution time. However, they leave it up to the programmer to manually and explicitly handle memory management. This has proven to be a difficult task for many programmers, and is often the reason for bug-prone and vulnerable programs [3] [4].

Flaws caused by memory management in IoT systems have become increasingly apparent in recent years. For instance, a memory vulnerability in CD players of certain cars gave attackers control of safety critical subsystems in 2011 [5], and cardiac devices showed vulnerabilities that allowed an attacker to deplete the battery and administer incorrect pacing and shocks in 2017 [6]. In fact, poor memory management is the second to most common vulnerability in embedded devices next to poor login credentials [7] [8]. The strength of passwords is the responsibility of the user; on the system level little can be done to improve this. However, memory management need not be the responsibility of the user and should be correctly handled behind the scenes.

A new programming language enforcing memory safety is Rust: a system-level language introduced by Mozilla [9]. Rust is focused on safety, speed and concurrency, and maintains these goals without having a garbage collector, making it a useful language for writing low-level code among others. In Rust, variables are tracked by their lifetime and deallocated when they go out of scope. The safety guarantees of Rust are particularly appealing for an operating system kernel. It makes buffer overflows, integer overflows, and uninitialized data bugs impossible, which

constitute a significant fraction of kernel bugs [10]. Even with these safety guarantees that are generally only found in high-level languages, Rust has memory efficiency and performance close to C/C++.

Tock is an embedded operating system developed in Rust taking advantage of the safety benefits it provides [11] [12]. It is designed for running multiple concurrent, mutually untrusted processes on low-memory and low-power microcontrollers. Tock brings flexible multiprogramming to resource-constrained systems while isolating processes from the kernel and from each other. The kernel heap in Tock is split across processes, which allows the system to respond to resource demands from one process without impacting the available memory of the kernel and other processes.

1.1 Problem Statement

Tock allows for user-level processes to be written in any programming language. This means that although Rust's safety guarantees hold for the kernel, they do not hold for processes. Processes are scheduled preemptively and given their own separate memory region, making them have strong system liveness guarantees. However, this still is not enough to guarantee memory isolation: preventing a process from accessing memory that has not been allocated to it, as most other programming languages do not enforce this at compile-time. In general-purpose computing, a memory management unit (MMU) would be used to prevent this from happening, providing memory isolation in addition to address virtualization. MMUs however contain features that are generally unnecessary in embedded systems, leading to a needless increase in cost and complexity. Recent microcontrollers include a memory protection unit (MPU) instead: a trimmed-down version of the MMU that also provides memory protection, but no address virtualization. An MPU can be configured to protect a certain address range in memory, also called an MPU region. Tock takes advantage of MPUs to enforce memory isolation.

The market of microcontrollers has always been a fragmented and heterogeneous one, where MPUs too vary significantly per chip designer. Factors such as region alignment, (minimum) region sizes, access permissions and behaviour in the case of overlapping regions or a region miss show significant differences between MPUs. For this reason, having a system support multiple MPUs is a complex undertaking. In addition, the procedure of setting up MPU regions is fundamentally tightly interwoven with memory allocation: hence supporting multiple MPUs in an operating system is a tedious operation. In fact, currently no operating system exists with the ability to support multiple MPUs, and the few operating systems that do support an MPU only support one specific type of MPU, that is mostly the MPU of the ARM Cortex-M instruction set architecture (ISA) [13]. ISAs like these have historically been proprietary for business reasons, and companies have patents on these ISAs that prevent others from using them without expensive licenses. Due to the rise of open-source initiatives, new ISAs with different MPUs have been gaining momentum, with the most prominent example being RISC-V [14] [15].

Tock is one of the operating systems that does support an MPU, but has its existing MPU implementation specifically set up for the Cortex-M ISA. The current implementation makes it impossible to use other MPUs, and because of this, Tock is unable to provide its safety guarantees for any other architecture than the Cortex-M. Therefore, Tock is currently only supported for platforms with a Cortex-M processor.

1.2 Research Goal

In this thesis, the problem of operating systems not being able to support multiple MPUs is tackled by defining an efficient and general software abstraction for MPUs, using Tock as the target operating system. Tock is programmed to be modular with one key exception: the MPU. Because of this, a valid solution to our problem fitting Tock would be to create a modularly programmed MPU interface, that enables the addition of MPU functionality for other hardware platforms. As a result, the main research question of this thesis is as follows:

Can an Architecture-Agnostic MPU interface be created in Tock?

This question raises a number of sub-questions:

1. *What are the main challenges in creating a generic MPU interface for Tock?*
2. *Can a generic MPU interface be created that is completely independent from the target application?*
3. *Can a generic MPU interface be created in Tock that is efficient for a wide range of MPUs, not having less features than a stand-alone implementation?*

The benefit of answering these questions is creating a kernel that is simple and easy to port to a wide variety of platforms, and yet provides strong isolation guarantees for processes.

1.3 Contributions

The contributions this thesis proposes can be summarized in the following manner.

- An analysis of state-of-the-art MPUs, including an overview of their key features and constraints in addition to a more in-depth look at their discerning capabilities.
- A feasibility study to explore the challenges involved in creating an MPU interface, both in general and for Tock specifically.
- A variety of modifications to Tock ranging from making it architecture-agnostic, reducing its context switching overhead and simplifying its implementation.
- Design, implementation and evaluation of a region-based MPU interface in Tock, that as an interface aims to be as generic and operating-system agnostic as possible, but in turn loses some MPU-specific optimizations, struggles with dynamic memory allocation and has a higher complexity[1].
  - A corresponding MPU-specific implementation for the Cortex-M MPU.
  - A corresponding MPU-specific implementation for the Kinetis K MPU.
- Design, implementation and evaluation of a process-based MPU interface in Tock, that is targeted at a process memory model supporting regions with different access permissions that grow towards each other and leaves a high degree of freedom to the implementor, thereby being able to support all MPU-specific optimizations[2].
  - A corresponding MPU-specific implementation for the Cortex-M MPU.

[1] `https://github.com/dverhaert/tock/tree/region-based-interface`
[2] `https://github.com/dverhaert/tock/tree/process-based-interface`

- A corresponding MPU-specific implementation for the Kinetis K MPU.

1.4 Thesis outline

The structure of this thesis is as follows. Chapter 2 gives the reader an overview of fundamental background information regarding operating systems and memory, provides information on the Rust programming language and the Tock operating system, and discusses related work. Chapter 3 performs a comparative analysis of the key and in-depth features of MPUs, and discusses current memory protection in Tock. Next, Chapter 4 walks through the initial design decisions made for generalizing Tock, that are valid for both designed interfaces. The first interface, the region-based interface, is proposed in Chapter 5, where we dive into its design, implementation and evaluation. We go through similar aspects for the second interface, the process-based interface, in Chapter 6. Finally, we discuss the results and provide recommendations for future research in Chapter 7.

Chapter 2

Background

This chapter introduces the reader to several topics essential for grasping the full nature of this research. First, operating systems are discussed, going through the fundamentals and other topics relevant to this research in Section 2.1. Next, memory and the importance of memory management are covered in Section 2.2, along with an introduction to memory protection units. Hereafter, Section 2.3 covers the novelties and key features of the Rust programming language. This all leads up to the introduction of the Tock operating system in Section 2.4. The motivation, design and memory layout of Tock are elaborated on and illustrated. Finally, the related work relevant to this thesis is covered in Section 2.5.

2.1 Operating Systems

The ulterior goal of a computer is to enable a user to communicate with hardware; programs and operating systems exist to simplify this process [16]. A program –such as a mail client, a browser or a game– is a sequence of instructions written by a computer programmer in a programming language, existing to perform a certain task when executed by a computer.

An operating system (OS) is software that controls the general operation of a system, acting as an intermediary between programs and the computer hardware. The motivation of an operating system is to provide an environment that is efficient and convenient for a user. Traditionally, operating systems are only used in general-purpose computers, making their design focused on being as convenient as possible. Recently however, operating systems have started to appear in embedded systems. Since efficiency is vital in embedded systems, the emphasis of these embedded operating systems design is on efficiency over convenience.

A central processing unit (CPU), also called a *processor*, is the hardware that carries out a program's instructions. At a processor's center are one or more *cores* that are its basic computation units; embedded systems traditionally have one core. An operating system can communicate with the processor through the use of CPU *registers*. A register is a hardware component that can hold a sequence of bits. Software can read registers to get information from the hardware, and can write information to the registers to send information to the hardware. In embedded systems, registers are usually 32 bits wide. An important consequence of this is that only 4 GB (2^{32}) of memory can be directly accessed.

2.1.1 Kernels

The core of an operating system, responsible for controlling all other programs, is called the kernel. Unlike a user program, the kernel has full control of the operating system. Therefore, in order to achieve proper operation of an operating system, the execution of user-defined code and kernel code must be distinguishable. Consequently, modern operating systems operate between (at least) two distinct modes:

Supervisor Mode In supervisor mode, executing code has unrestricted access to the underlying hardware. Any instruction can be executed, and each memory address can be referenced. Supervisor mode is generally reserved for the most trusted functions of the operating system. Supervisor mode is often also called system mode, privileged mode or kernel mode.

User Mode In user mode, executing code cannot directly access the underlying hardware. Code running in user mode is restricted and can only access certain pre-specified addresses. User mode is utilized for most of the programs on a computer.

The current mode of the operating system is most often indicated by a bit present in the hardware of the computer and enforced by the CPU. At boot time, the system starts in supervisor mode. Once the operating system is fully loaded, programs are dispatched in user mode to protect the system from unauthorized or unwanted access of privileged resources. If a program running in user mode wants to request a service that can only be fulfilled by the kernel, it can do so by means of a a *system call*. A system call provides an interface between a program and the kernel, and is the only normal entry point for a user-level program into the kernel.

After a program is done executing, it is supposed to return control to the operating system by executing a system call. Programs that only voluntarily return control in this manner are *co-operatively* scheduled. To ensure programs do not keep control infinitely, frequently operating support setting up a timer to interrupt after a specified period, giving control back to the operating system. Programs that are executed in this way are called *preemptively* scheduled.

Kernel architectures can be written in two ways. A *monolithic* kernel is one that is big and complex, without having a specified structure. All services are bundled together, resulting in a smaller memory footprint and often higher efficiency. On the other hand a *microkernel* architecture has the goal of being as small as possible. Microkernels are predominantly known for their reliability and security; if one part of the system fails, the whole system does not crash unlike in monolithic kernels. Additionally, easier to extend and customize. The drawback associated with microkernels is their poor performance.

2.1.2 Processes

A program in execution is called a *process*. A process predominantly contains the following memory sections:

- The **text** section (also known as the code segment) contains the program's code.
- The **data** segment contains global or static variables.
- The **stack** section contains temporary data. During runtime, the stack can be dynamically adjusted according to a last in, first out (LIFO) structure.
- The **heap** area contains data dynamically allocated during process runtime.

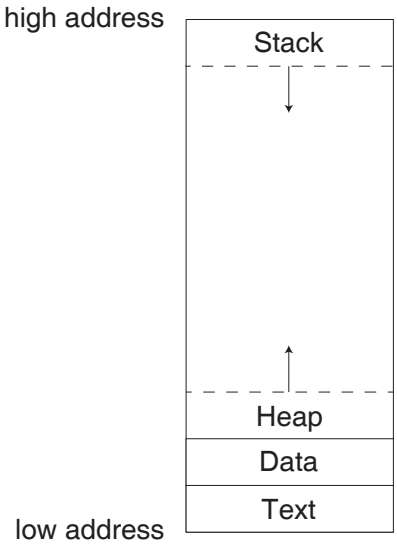

Figure 2.1: Memory overview of a process. The dashed lines and arrows indicate a region can expand in that direction.

Since the stack and heap can both grow during runtime, these are generally set to be growing towards each other in memory. A schematic of this general memory overview is shown in Figure 2.1.

Typically, operating systems are designed to support seemingly simultaneous running of multiple processes. To do this, they employ a technique called *context switching*, that essentially suspends the execution of one process and resumes the execution of another process. Context switching can only occur in supervisor mode. Doing a context switch is in most cases computationally intensive for an operating system. Consequently, avoiding unnecessary context switching has been a central point in the design of operating systems.

Independent processes are not affected by the execution of other processes. It can however be advantageous for processes to cooperate with each other for increasing computational speed, convenience and modularity. Inter-process communication (IPC) is a mechanism allowing processes to do this.

2.2 Memory Management

Memory management is the process of controlling and coordinating computer memory among programs, deciding how much memory each program gets at what time. In order for a computer system to function properly, proper memory management is essential. Flawed memory management can lead to slow performance, bugs, and even vulnerability to malware.

Memory management is done in either volatile or non-volatile memory. Volatile memory characterizes itself by being faster than non-volatile memory. On the other hand, non-volatile memory is distinguished by offering much more memory capacity and retaining saved data even without any power supply. In most modern general-purpose computers and embedded systems, both

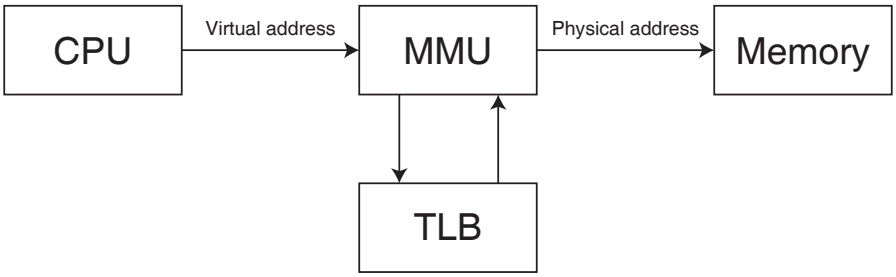

Figure 2.2: Simplified flow of a memory reference using an MMU.

forms of memory are used so that data can be stored both reliably and quickly. For embedded systems, usually a form of random access memory (RAM) is used for volatile memory, and flash is used for non-volatile memory.

Proper memory management aims to to minimize *fragmentation*: the inefficient use of storage space. Internal fragmentation is the wasted space within memory regions that results from allocating too much memory for a process. This is often the result of having to round up from the requested allocation to the allocation granularity. External fragmentation occurs when free memory is interspersed by allocated memory, i.e., having holes between memory regions.

2.2.1 Memory Management Units

Memory references on most general-purpose devices initially arrive at a memory management unit (MMU). This unit provides two main features. The first is address translation, that in essence is a method to abstract away memory. Using address translation, the operating system translates a request for a *virtual* memory address to an actual physical address. This enables programs to execute without having their entire address space be in RAM by relocating and accessing memory where convenient, resulting in reduced storage space and more flexibility in RAM. The virtual address space is regularly subdivided into equally-sized *pages*, and these are then stored in an on-chip *page table*. Usually, page sizes are in the order of kilobytes, and the corresponding page table has a number of pages in the order of thousands [17]. The downside of using address translation is that the translation costs time, leading to a performance reduction. To minimize this reduction an MMU generally contains a translation lookaside buffer (TLB). A TLB is a fast, on-chip memory cache that contains a number of past entries for an MMU's virtual-to-physical translations table. Even though an MMU's TLB greatly reduces the *average* memory access time, inevitably memory addresses not stored in the TLB yet will need to be accessed, and these incur a significantly higher performance cost.

The second main feature an MMU provides is memory isolation. An MMU can be configured to protect certain address ranges – also called regions – in memory. Whenever a process tries to access a certain address, the MMU checks if the process is authorized to do so. If not, the access will be blocked and a segmentation fault will occur, generally leading to an abort of the program. A simplified flow of a memory reference when using an MMU is shown in Figure 2.2.

Figure 2.3: Simplified flow of a memory reference using an MPU.

Feature	MMU	MPU
Address translation	Yes	No
Memory isolation	Yes	Yes
Protected segment	Fixed (page)	Flexible (region)
Typical number of entries	Thousands	Tens
Region/page table location	External Memory	On-chip registers

Table 2.1: Comparison between MMU and MPU Features.

2.2.2 Memory Protection Units

A full-fledged MMU is not the best fit for embedded systems. Address translation is an expensive operation in terms of memory and performance and having more unnecessary features like a TLB leads to increased cost and complexity. For this reason, most embedded systems choose not to implement an MMU and try to find another solution for memory isolation. One of the most recent solutions is a hardware component that does not have these drawbacks: the memory protection unit (MPU). Simply put, the MPU is a trimmed down version of the MMU that only provides memory isolation, but is far simpler and cheaper. An overview of the regular flow of memory references on a system using an MPU is shown in Figure 2.3.

Where MMU pages are fixed blocks in memory, MPU regions are generally defined as segments that have a variable size. The number of memory regions for an MPU is usually in the order of tens, contrary to the order thousands of pages for the MMU. For this reason, the MPU region table can be kept in on-chip memory, whereas MMU regions are generally kept on external memory. A comparison between MMU and MPU features is shown in Table 2.1. Because of this difference in features, MPUs generally have a lower cost, complexity and power consumption in comparison to MMUs.

2.3 Rust

The choice of a programming language is important in setting up an embedded system. Ideally, one wants a language that is both easy to understand and has a high performance. Regrettably, a trade-off has to be made. In order to more deeply understand this trade-off, it is useful to subdivide programming language into two classes:

- **Higher-level languages** enable programmers to produce software very close to the user (and far away from the hardware). Abstracting away from the hardware makes these languages very user friendly: they are platform independent, the syntax is easy to read and write, and the code is checked for mistakes. The focus is on reducing the time needed to write the program, and the performance of the program is a secondary concern.
- **Lower-level languages** enable programmers to produce software very close to the hardware (and far away from the user). Lower-level languages get the most performance out

of hardware by efficiently using available resources, therefore usually being very efficient in terms of memory usage and execution time. A great degree of hardware awareness is required to do low-level programming, making them much harder to code.

Since embedded systems must be as efficient as possible given their cost and power requirements, lower-level languages are used to program them. The most common languages for this are C and C++. Both of these are pretty close to the lowest level of machine code and therefore provide high performance, while also remaining relatively highly portable and easy to program. Unfortunately, a very useful property that these languages lack is *type safety*. Type-safe languages ensure no possible execution in any program written in the language can exhibit undefined behaviour. In the ideal world, code written by programmers would not contain any mistakes and therefore would also not exhibit undefined behaviour, but in reality mistakes are commonly made [10]. This makes languages like C and C++ bug prone and vulnerable. On the other hand, languages like Python, Java(Script), Ruby and Haskell are all type safe, but are relatively higher level and therefore lack the performance and efficiency of lower-level languages.

This apparent gap led to the introduction of Rust, a lower-level programming language designed to map directly to hardware while providing most features of high-level languages including type safety, generics, enumerations (algebraic data types) and much more [9]. Rust is a relatively new programming language that started out as a hobby project by a former Mozilla employee. Mozilla adopted the language and has released it as an open-source project, with its first stable release on May 15, 2015. Since then, Rust has been successfully used in large projects such as Dropbox's back end storage [18]. Rust provides two main features:

Memory Safety Security exploits often leverage bugs in the way languages (mostly C and C++) handle memory. Rust disallows the use of three things: null pointer dereferences, dangling pointers and buffer overruns. Instead, it uses options, borrowing and arrays with definite bounds.

Data-race free concurrency Writing (safe) multithreaded code is becoming increasingly important in modern devices [19], and is often complicated in traditional languages. Rust allows for concurrency to be used safely, and advocates developers to use it from the start.

Roughly 50% of all security critical bugs in Firefox's layout engine are related to mistakes in memory safety, even though its C++ programmers are experienced and have access to the best static analysis tools available [3]. This shows the severity of this problem, and the value of Rust's memory safety.

2.3.1 Ownership

All programs have to manage memory in some way. Some languages have garbage collection, continuously looking for unused memory while the program runs. Other languages enforce the programmer to explicitly allocate and free memory. Rust uses its own approach called *ownership* (an affine type system [20]), that is also its most unique feature [21]. This approach manages memory through a set of rules that the compiler checks at compile time. We proceed by explaining this through an example. Consider the code fragment in Listing 1

This code will print "$x = 1$, $y = 1$" when executed. Since x and y are both integers, their size is known at compile time. Copies are therefore easy to make, and Rust chooses to store primitive types entirely on the stack. That means now two variables x and y exist, both with a value of 1. A version where a reference to memory is used instead is illustrated in Listing 2

```rust
// Rust can statically infer a variable's type at compile time
let x = 1;
// Copy the value of x into a new variable y
let y = x;
// Print x and y
println!("x = {}, y = {}", x, y);
```

Listing 1: Basic example of copying in Rust.

```rust
// Let x be a vector of size 5
let x = vec![1, 2, 3, 4, 5];
// The pointer of x is copied into y, giving y ownership
let y = x;
// Compiling will fail at this line, since x no longer owns the heap memory
println!("x[0] = {}", x[0]);
// This line will compile successfully, since y now owns the heap memory
println!("y[0] = {}", y[0]);
```

Listing 2: Basic example of ownership in Rust.

Since this looks very similar to the previous code, one might assume it works the same and y will be a copy of x. However, this code will not compile for a subtle but important reason. The first line allocates memory for the vector object x on the stack, and allocates memory on the heap for the actual data (1, 2, 3, 4, 5). Creating the copy y now does not actually allocate the same vector on the heap, but just copies the vector object/pointer x. As this results in two pointers to the same memory, that could lead to a data race and violate safety guarantees, Rust forbids this copying. Instead, Rust transfers the ownership of the vector to y.

2.3.2 Borrowing

To enable sharing of objects, Rust uses a concept called *borrowing* Listing 3 gives an example of borrowing.

```rust
let mut x = 1;
{
    // y is defined as a mutable reference to x
    let y = &mut x;
    // Increment the value y points to (x) by 2
    *y += 2;
}
println!("{}", x);
```

Listing 3: Basic example of borrowing in Rust.

This code will print 3 when executed. First, note the `mut` keyword with which x is defined. Variables in Rust are immutable by default, and only mutable with the addition of this keyword.

After defining x, a new scope is entered where y is made a mutable reference to x, and 2 is added to the thing y points to. If x was not mutable, a mutable borrow to an immutable value could not have been taken. Rust has two fundamental rules for borrows. First, any borrow must not last for a greater scope than that of the owner. Second, there may either be exactly one mutable reference to a resource, or any number of immutable resources.

2.3.3 Lifetimes

Both ownership and borrowing are enforced through *lifetimes*. A lifetime is effectively just a name for a scope somewhere in the program. They exist to ensure a variable binding points to a valid resource, thereby enforcing memory safety. An illustration of lifetimes is shown in Listing 4.

```
fn main() {
    let mut x = 1; // Lifetime of x begins
    {
        let y = 2; // Lifetime of y begins
    } // Lifetime of y ends
    println!("y = {}", y) // This will fail, since y went out of scope
} // Lifetime of x ends
```

Listing 4: Basic example of lifetimes in Rust.

Each reference in Rust is marked with a lifetime specifying the scope it is valid for. The compiler lets you elide lifetimes in common cases, as was the case in the previous examples. Explicit lifetime annotations are required when for instance working with structs that contain references, or when a lifetime is required that is different from its scope. Lifetimes prevent the existence of dangling pointers.

Since ownership, borrowing and lifetimes are all enforced at compile time, there is no cost at runtime and therefore Rust accomplishes safety and speed. However, because this is a new concept, it does take quite some time to get used to. Many new users to Rust experience 'fighting the borrow checker', where a program the author thinks is valid does not get compiled by the Rust compiler. The good news is that the compiler is user-friendly, and usually provides hints as to what should be changed in order to make the program successfully compile. Once users overcome this rather steep learning curve, many of them grow to love Rust [22].

2.3.4 Unsafe Rust

Any operating system requires some code that does not enforce memory safety guarantees. By nature, static analysis is conservative, and furthermore, underlying computer hardware inherently requires some memory unsafe code. For this reason Rust has a second language hidden inside enabling these memory unsafe features called *unsafe Rust*. When code is encapsulated by an `unsafe` block, several things like dereferencing a raw pointer and accessing or modifying a mutable static variable are allowed. Within an `unsafe` block, it is up to the programmer to ensure these things do not break the system at runtime. Note that the ability for a part of code to use

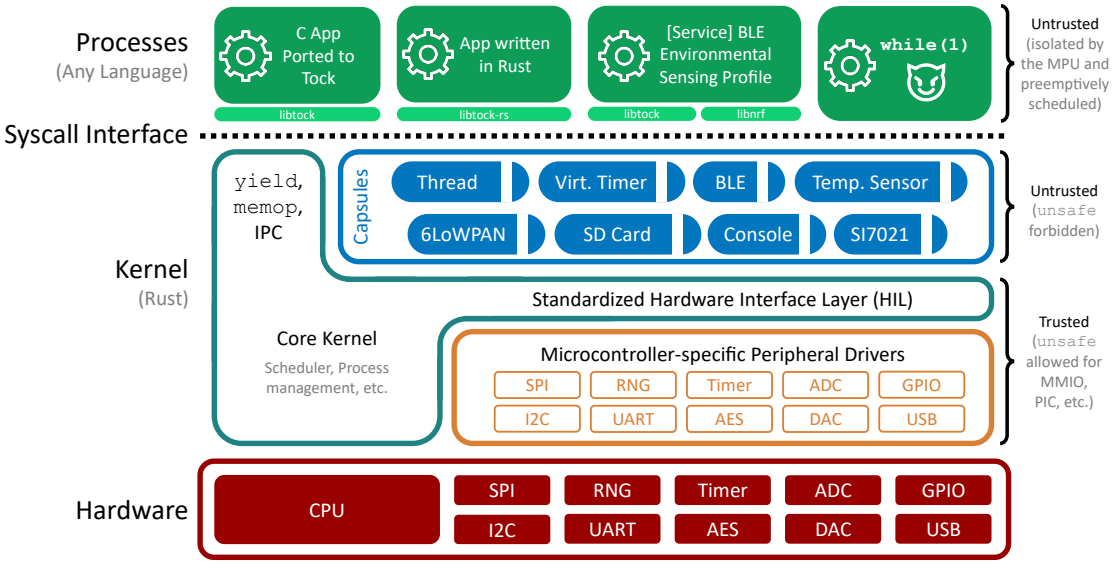

Figure 2.4: Overview of Tock's architecture. The kernel contains both the core (trusted) code, and the (untrusted) capsules.

an `unsafe` block can be rescinded if necessary. This is useful to for instance prevent user-level programs from calling `unsafe`.

2.4 Tock

The strict current-day power and cost budget has kept embedded platforms very simple, meaning RAM and other features like memory isolation are scarce. On most embedded systems, this simplicity causes faults to be non-isolated. Hence, the misconduct of a single application could lead to the entire system crashing, and even worse, devices are often very vulnerable to hacking. Another downside is that memory is often statically allocated, enforcing developers to guess how much memory to allocate for processes. Tock is an embedded operating system addressing these shortcomings by providing fault isolation and dynamic memory allocation [11]. Tock's kernel is written in Rust, and therefore inherits Rust's type safety, high memory efficiency and performance. In addition, Tock provides abstractions for processes using hardware isolation mechanisms found frequently in recent chips.

2.4.1 Overview

Tock is designed for running multiple mutually distrustful programs concurrently and independently on embedded systems [1] [23]. An overview of Tock's architecture is shown in Figure 2.4.

[1] `https://github.com/tock/tock`

Call	Core/Capsule	Description
`command`	Capsule	Invoke an operation on a capsule
`allow`	Capsule	Give memory for a capsule to use
`subscribe`	Capsule	Register a callback
`memop`	Core	Increase heap size
`yield`	Core	Block until an callback completes

Table 2.2: Tock system call interface [11].

Capsules

Recall from Section 2.1.1 that traditionally, operating systems distinguish between at least user-defined code and kernel code. Tock distinguishes between another type of code called *capsules*. Capsules are programs written in Rust that exist within the kernel; therefore, they are constrained by Rust's strong memory safety. Because of this, there is no overhead associated with safety and capsules require minimal error checking. Capsules are cooperatively scheduled, causing context overhead to be minimal by taking advantage of the kernel's short operations. However, this means they must also be trusted for system liveness. Even though capsules exists within the kernel, unlike kernel code they are not allowed to call `unsafe` code.

Processes

Tock characterizes user-defined code as *processes*. Processes are isolated from most parts of memory by hardware, allowing them to be written in any language. The maximum number of processes is only constrained by the available flash and RAM. Processes are scheduled preemptively and use a round-robin policy. For interacting with the kernel, processes can use system calls as elaborated on in Section 2.1.1. System calls are either routed to capsules or the core through a system call interface; this is shown in Figure 2.4. Five system calls currently exist in Tock and are shown in Table 2.2.

Grants

Capsules require the ability to allocate memory in response to process requests. For example, a virtual timer driver must allocate a structure to hold metadata for each new timer any process creates. Generally, one of two existing techniques would be used to address this problem:

- Static memory allocation. The limits of this technique are that with a too high estimated number, memory is wasted, whereas a too low estimated number limits concurrency.
- Using a global kernel heap. The drawback of using such a technique is that it can lead to unpredictable resource exhaustion.

Unfortunately, both techniques have their limitations. The novel solution Tock introduces to resolve this problem is by using so-called *grants*. The grant region is a dynamically-sized region in a process's memory, allowing the kernel to use a part of a process's memory for its own purposes. Having all memory related to a certain process in its own memory has two advantages. First, the required kernel memory for one process does not affect the available kernel memory for another process. That is, if one process requests an enormous amount of memory, it will not

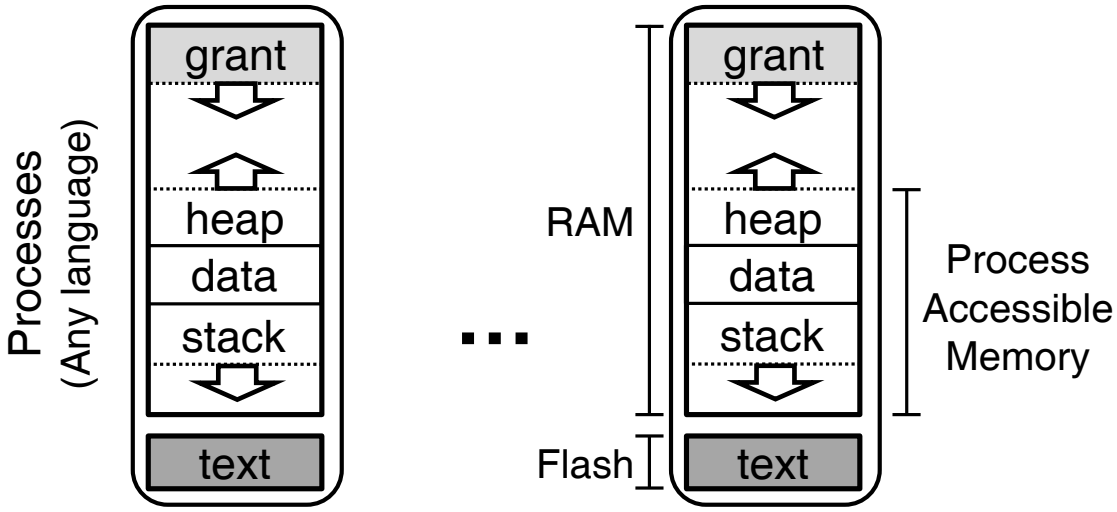

Figure 2.5: Memory layout of processes in Tock [11].

negatively influence other process. Secondly, if a process dies, all resources can be immediately freed.

The manner in which Tock layouts process memory is presented in Figure 2.5. Observe that this is different from a conventional operating system as seen in Figure 2.1 because of Tock's grant region. Normally, the stack and heap grow towards each other since these are the only dynamically sized regions; now, the grant is dynamically sized as well. This problem is tackled by having the maximum stack size be static instead, which is set at compile-time.

2.4.2 Memory Isolation

Rust preserves memory safety at compile time as covered in Section 2.3. However, since Tock processes can be written in any language, they are not protected by the Rust type system and therefore can still access certain addresses that they should not have access to in memory. In other words, Rust is not sufficient to provide memory isolation in user-level Tock, and some other component is necessary in order to safely support untrusted processes. For this reason, Tock uses MPUs (Section 2.2.2) provided by recent embedded microcontrollers. The MPU is set up so that processes are by default only allowed to access their own memory, excluding the grant region. Consequently, the MPU configuration is different for each process, and the MPU is reconfigured with each context switch to a process. The combination of the stack, data and heap region of a process which a process does have access to is referred to as process accessible memory (PAM). Together with the grant and the flexible memory between the grant and the heap, this consists of a process's RAM.

When the system is executing kernel code, Tock chooses to disable the MPU completely instead of using supervisor mode. This means there are no hardware restrictions preventing the kernel from accessing the entire address space. In other systems, this would lead to the system being vulnerable to kernel bugs. In Tock however, the Rust type system restricts what the kernel can do. For example, it is impossible for a capsule (which cannot use unsafe) to access a process's memory because it cannot create and dereference an arbitrary pointer. In general, Tock tries to

minimize the amount of trusted code (i.e. code that can call unsafe), and tries to encapsulate code that does need unsafe to make it clear what that code does and how to use it in a manner that does not violate overall system safety.

There are some exceptions to the memory isolation of Tock for processes. Processes can choose to explicitly share portions of their RAM in two ways:

- With the kernel. This is done through the use of `allow` system calls as discussed in Section 2.4.1 This gives capsules access to the process's memory for use with a specific capsule operation.
- With each other, through an IPC mechanism as discussed in Section 2.1.2. To use IPC, processes specify a buffer in their RAM to use as a shared buffer, and then notify the kernel that they would like to share this buffer with other processes. Then, other users of this IPC mechanism are allowed to use to this buffer.

Tock's kernel follows the microkernel design. Hardware-specific implementations are separated by their hardware platform. Processes, capsules and the kernel all have a completely architecture-agnostic implementation and use Rust's *traits* to interface with the hardware. Traits enforce a set of behaviours to which implementations must comply; they are similar to interfaces in other programming languages.

2.5 Related Work

In order to measure the value of using MPUs for memory isolation in Tock, it is important to see which existing systems and techniques already exist to solve this problem. For this reason, we start by looking at the embedded operating systems landscape, and discern what techniques are used to provide memory safety in these operating systems. Next, we split memory isolation methods up in software- and hardware-based techniques, and compare this with Tock's use of an MPU.

This section gives an overview of the embedded operating systems landscape

2.5.1 Embedded Operating Systems

While embedded operating systems have a focus on efficiency over convenience as discussed in Section 2.1, they must still be convenient enough for a user to get started quickly enough. Because of this, for now Linux – the best-known open-source operating system – is the largest operating system deployed on embedded systems, having a good balance between being resource-efficient and possessing a large amount of simple yet effective functionality [24] [25]. Moreover, Linux has a very large contributing open-source community, helping it advance at a fast pace.

The advantage of current-day open-source initiatives has not only spurred the growth of Linux, but also that of several open-source embedded operating systems. Contrary to the monolithic kernel design as discussed in Section 2.1.1, to which Linux adheres, a microkernel is the regular design choice for embedded operating systems due to its better reliability, extensibility and customization. The growth of these embedded operating systems has been very high in the last three years; in fact, it has been greater than the growth of Linux on similar platforms [25]. A list of leading embedded operating systems and their distinguishing features is as follows:

- **TinyOS** targets wireless sensor networks with scarce resources [2]. It is written in the nesC language, optimized for event-driven execution, flexible concurrency and component-oriented application design [26].
- **Contiki** has a focus on flexibility in low-power wireless IoT devices [27]. Contiki's key feature is its network mechanism, providing IPv4 networking, an IPv6 stack and a custom lightweight networking protocol.
- **RIOT**'s main design objectives are real-time support, modularity and multithreading [28].
- **FreeRTOS** has ease of use, a small memory footprint and robustness as its primary design goals [29].

Besides these, numerous other embedded operating systems are emerging [30] [31]. As of yet, few operating systems leverage the capabilities of emerging MPUs. Some that do are FreeRTOS [29] and Contiki [27]. Both of these have an implementation for the Cortex-M architecture's MPU. However, the interface with which they communicate to this MPU is tailored to this architecture, and they do not support any other MPUs. Such an interface is easier to construct, since the memory allocation model in the kernel code is pre-arranged so that the regions created always match to valid MPU regions.

2.5.2 Software-based memory isolation

Most existing embedded operating systems do not support MPUs. Instead, they use more traditional, software-based approaches to provide memory isolation. CCured [32] attempts to do compile-time verification, and inserts run-time checks where this is insufficient. However, in the process, it changes the programming language C's data representation by replacing regular C pointers with "fat pointers" that contain extra information. In addition, doing dynamic checks like this leads to an undesirable runtime overhead. Harbor [33], a memory protection system built on SOS [30], enforces memory isolation by distinguishing between distinct subsets of the overall memory space, where each process can only write into its own domain. Harbor ensures return addresses of function calls can not be corrupted by storing these in a protected area. Another approach is taken by *t-kernel* [34], that ensures memory isolation by doing extensive code modification at load time. Although both Harbor and *t-kernel* achieve static memory isolation just like Tock, they trade this off with a lower efficiency and a higher complexity due to the fact that they are language modifications. Moreover, language modifications seldomly tend to be sufficient to guarantee memory isolation [35] [36]. Since Rust already natively supports type safety due to its language constraints, and furthermore Tock is backed up by an MPU enforcing memory isolation, Tock is able to make stronger safety guarantees at a lower cost.

2.5.3 Hardware-based memory isolation

MPUs have only gained popularity in the last few years, simplifying the problem of memory isolation. Before this, several other parties were already trying their hand at hardware solutions. SMART [37] proposes a straightforward and efficient hardware-software primitive, demonstrating that minimal changes with a simple measurement routine in memory can lead to memory isolation on devices that did not have it before. Another similar solution is found by the authors of Sancus [38] [39], deploying a minimal hardware unit for extra CPU instructions that is used to create protected memory regions. However, both Sancus and SMART have their restrictions.

The interaction between protected processes in SMART is very slow, and Sancus has a relatively high hardware cost in addition to restricting processes' freedom. The authors of TrustLite [40] propose their own MPU instead: a Secure Loader hardware unit. This MPU records memory access rules in a configurable hardware table. Compared to Sancus and SMART, this allows for a more fine-grained protection scheme. However, TrustLite does not support dynamic process loading. For this reason, TrustLite has recently been expanded by introduction of the TyTAN architecture [41], enabling dynamic loading, real-time scheduling guarantees and secure IPC. Nevertheless, TyTAN still has a significant overhead associated with creating a secure task that is over 100 times greater than creating a normal task.

Since not all operating systems have an implementation for an MPU even though they have microcontrollers that support it, some take it upon themselves to create this implementation. Hardin et al. create such an implementation [42] for the TI MSP430FR5969 [43] microcontroller on the Amulet wearable platform [44]. An interesting insight from their work is that using the MPU has less than a 0.5% impact on battery lifetime. Unfortunately, the MSP430 has a very limited MPU that cannot protect the region below the allocation of the current app. This makes the authors resort to compiler-inserted bounds checks in order to provide memory isolation, which is far from an optimal solution. Because of this, the authors envision extending their work to more advanced MPUs.

Chapter 3

Memory Protection Units

The first and foremost challenge in creating a memory protection unit (MPU) interface for Tock is the diversity and heterogeneity of MPU designs on different microcontrollers. Recent platforms that support MPUs are rapidly emerging, and have a great amount of variance in semantics for controlling the MPU, in addition to varying constraints. In order to construct a generic MPU interface, it is essential to quantify these design differences and find a way to overcome the associated difficulties. Furthermore, it is of importance to analyze the feasibility of creating such an MPU interface on Tock, the target operating system.

This chapter starts off by introducing several MPUs as targets for analysis in Section 3.1. Next, in Section 3.2 an analysis of MPUs that are based on region-based protection is performed. The most fundamental MPU characteristics for a variety of region-based MPUs are analyzed, their key differences are compared using a table, and other important differences in terms of permissions, overlapping regions and default access permissions are explored. Afterwards, the working of barrier-based protection is briefly studied in Section 3.3. Consequently, the challenges that Tock provides are studied in Section 3.4, looking at the existing implementation that has been tailored for the Cortex-M MPU. This includes Tock's process manager, an initial attempt at an MPU interface, and the existing MPU implementation for the Cortex-M processor.

3.1 Choice of MPUs

A number of MPUs have been selected as targets to perform a comparative analysis on. The key factors taken into account in the selection of these MPUs were popularity, feasibility for embedded operating systems and uniqueness. After all, including a wider variety of MPUs will aid in fulfilling the research goal of this thesis: making a truly generic MPU interface. This has resulted in the following list of MPUs.

Cortex-M MPU ARM Cortex-M is a series of 32-bit processors widely used in modern microcontrollers [45]. The series is built on the ARM instruction set architecture (ISA), and the series' core is currently the most widespread used core in embedded systems [46]. The Cortex-M0+ [47] and the Cortex-M3 [48] both contain an MPU and are the most prevalent processors based on

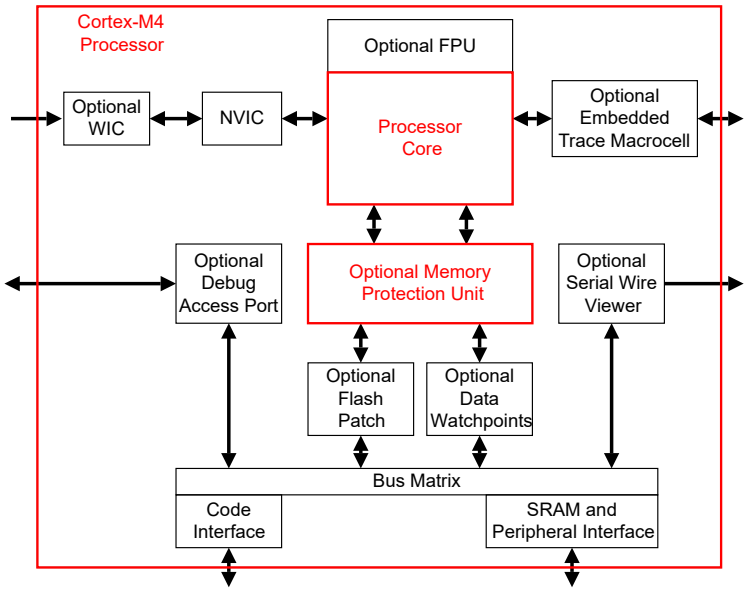

Figure 3.1: Schematic Overview of the Cortex-M4 processor [13]. Highlighted are the processor, its core, and the MPU.

the Cortex-M series, although many other types exist. In the Cortex-M series, the MPU is (optionally) contained within the processor. This is illustrated by the schematic overview of the Cortex-M4 in Figure 3.1.

Many microcontrollers use the Cortex-M processor and therefore also inherit its MPU. Microcontrollers that do this and are also implemented for Tock are the Nordic Semiconductor nRF52 [49] and Texas Instruments' TM4C129x [50] and CC26X [51].

RISC-V MPU RISC-V [15] [52] is a rapidly growing open-source ISA that is designed to be useful in a wide range of devices. RISC-V is overseen by the non-profit RISC-V foundation and, in contrast to the ARM ISA, is completely free and open, which makes it easier to implement and extend. In 2017, RISC-V has introduced an MPU in their architecture called physical memory protection (PMP).

Kinetis K MPU NXP Kinetis K [53] is a series of microcontrollers that offers high performance, scalable integration and low-power capabilities built on the Cortex-M4 core. Even though the microcontrollers of the Kinetis K series all contain a Cortex-M4 core, they do not contain its MPU, as this is located outside of the Cortex-M core as illustrated in Figure 3.1. Instead, the Kinetis K series contains its own MPU as a microcontroller peripheral.

Nordic nRF51 MPU Building on a Cortex-M0 core [54], the Nordic nRF51 series offers an ultra low-power wireless microcontroller [55]. Partly because of this ultra-low power requirement, the MPU on the nRF51 provides a very limited set of features. Like the Kinetis K series, the

MPU is not located in the processor or core, but outside of the processor as a peripheral on the microcontroller.

NIOS II MPU The NIOS II processor is a processor specifically designed to be embedded in Altera field-programmable gate array (FPGA) devices [56]. Using the NIOS II processor, a system designer can specify and generate a custom NIOS II core with an instruction set customized from the NIOS II ISA. Although the NIOS II processor contains an MMU, it also contains an optional MPU that is designed for environments that do not require virtual memory management.

Xtensa LX7 MPU The Xtensa ISA, just like NIOS II, is an architecture that allows designers to customize the instruction set automatically [57]. Processors based on the Xtensa ISA are able to be fine-tuned for a specific application at design time; one of these processors containing an MPU is the Xtensa LX7 processor [58]. Because The Xtensa LX7 processor has been designed with these reconfigurable and high-performance goals in mind, where everything is about increasing performance over reducing memory usage, the MPU is relatively coarse-grained.

TI Keystone MPU The TI Keystone II ISA is a multicore architecture that has been designed with high-performance digital-signal processing goals in mind, mainly targeted at embedded infrastructure applications [59]. For performance reasons, the TI Keystone MPU can often be found multiple times on the same architecture, in contrast to other MPUs that are usually only implemented on a microcontroller once. For instance, the 66AK2Hxx microcontroller contains 15 MPUs [60].

3.2 Region-Based Protection

Recall from Section 2.1.1 that in order to achieve proper operation of an operating system, user-defined code and kernel code must be distinguishable. Because of this, MPUs generally have the ability to set both user and supervisor permissions for an certain memory region. To create such a region, one must specify a beginning and an end address (or a size) in memory, along with permissions for this region in terms of the user and the supervisor. Frequently MPUs define three fundamental access types for which permissions can be set: read (R), write (W) and execute (X). These can be controlled for both user and supervisor mode. Most embedded devices consider flash and RAM as a continuous block of memory, sharing the available MPU regions across these.

Most MPUs:

1. Allow for the creation of regions.
2. Are capable of distinguishing between at least supervisor and user mode
3. Define read, write and execute permissions

These will be referred to as *region-based* MPUs. Of the selected MPUs, the only one that is not region-based is the nRF51; it will be referred to as being *barrier-based*. The design of the nRF51 will be discussed in more detail in Section 3.3. For now, region-based MPUs are looked into.

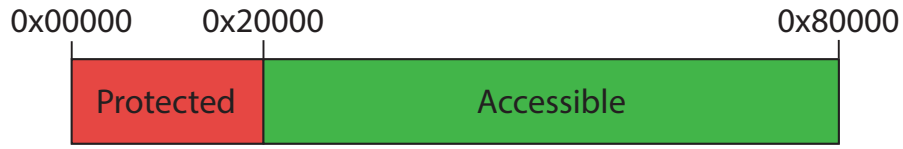

Figure 3.2: Example illustrating the general functionality of an MPU. The memory segment shown in red is protected by the MPU. The green segment represents the remaining unprotected memory.

In order to give the reader an idea of the general functionality of a region-based MPU, an example is shown in Figure 3.2. In this example, the memory size totals 512 kB, and the first 128 kB must be protected from any access by a process while leaving it accessible for the kernel. In hexadecimal, the total memory range from 0 to 512 kB is represented by `0x00000` to `0x80000`. The 128 kB region that requires protection ranges from `0x00000` to `0x20000`; so these addresses are set as the beginning and end address for an MPU region, respectively. Next the permissions for this region are configured as read, write and execute (RWX) for the supervisor, whereas no permissions for the user are set. We will now proceed by analyzing region-based MPUs by taking a look at their key features, the way they set up permissions, default access permissions and the behaviour on overlap.

3.2.1 Key Features

The key features of an MPU are the maximum number of regions, the region granularity, and the restrictions on the sizes and start addresses of these regions. An analysis of these features for the chosen MPUs is shown in Table 3.1. Some interesting things to consider given this table are the following:

Number of Regions The Cortex-M supports so-called *subregions*. Each MPU region consists of eight equally-sized subregions, which can be independently enabled and disabled. NIOS II on the other hand separates read/write regions from execute regions, and allows 32 regions for both to be set. The Xtensa LX7 and TI Keystone MPUs have various possible configurations that result in a different number of maximum regions.

Granularity The Xtensa LX7 and the TI Keystone architectures have been designed with high performance goals in mind, in contrast to the more traditional optimization targets of embedded systems like cost and energy consumption. For this reason, the granularity for the Xtensa LX7 and the TI Keystone is far coarser than that of the other architectures.

Region Size The Cortex-M, RISC-V and NIOS II MPUs have regions that must have a size equal to a power of two. This is fairly restrictive and can have a great impact on memory allocation. For instance, when a protected segment of 7 kB is desired, the closest possible region conforming to this constraint would be 8 kB. One solution for obtaining this exact segment could be to use three contiguous regions of 4 kB, 2 kB and 1 kB: however, this is a very architecture-specific solution and has a cost of using multiple regions. The Kinetis K, Xtensa LX7 and TI Keystone MPU have a size that must be divisible by a certain constant, where the Kinetis K is much more fine-grained than the other two.

Region Alignment The MPUs that have a power of two constraint on their size have the additional constraint that start addresses have to be divisible by their size. For example, if

MPU	Number of regions	Granularity	Region Size	Region Alignment
Cortex-M [13]	$8 \cdot 8$ subregions	32B	power of two	start % size = 0
Kinetis K [61]	12	32B	size % 32 = 0	start % 32 = 0
RISC-V [52]	16	4B	power of two	start % size = 0
NIOS II [56]	32 R/W, 32 X	64B	power of two	start % size = 0
Xtensa LX7 [58]	16 or 32	4kB	size % 4kB = 0	start % 4kB = 0
TI Keystone [62]	2, 4, 8 or 16	1kB	size % 1kB = 0	start % 1kB = 0

Table 3.1: Attributes of various MPUs.

a region is 8 kB in size, it must start on an 8 kB boundary. This not only forces the start addresses to be limited to addresses divisible by a power of two, but also causes them to be dependent on the region size as will be discussed in an example in Section 3.2.3. The non-power-of-two MPUs require region start addresses to be divisible by their granularity.

Generally, in terms of the region size and start address, observe there are two different themes. The first is MPUs where sizes align with a power of two, and the start address is divisible by the size. The second is MPUs where both the size and start address are divisible by the same constant: their granularity. We will separate these as *power-of-two-aligned* and *block-aligned* MPUs. Because of their different alignment and size constraints, the considerations that have to be made for these two types are very different.

Note that the assumption that all existing MPUs fall into one of these two types can not be made; therefore, these should not exclusively be used as a reference point in order to create a generic MPU interface.

3.2.2 Access Permissions

The manner in which read, write and execute permissions can be set differs per MPU. The easiest way to set permissions is by using a different bit for each of the RWX permissions. We call this procedure using *bit-based permissions*. Xtensa LX7 and TI Keystone take this approach for both the user and supervisor levels, resulting in a total of six bits per region. However, since some scenarios are generally not very useful – for instance, to let the user have more permissions than the supervisor – the Cortex-M and NIOS II choose an *encoded permissions* technique instead. In this approach, a specific set of permissions is defined. For instance, the set of permissions implemented for the Cortex-M devices is shown in Table 3.2. This leads to a total of 4 bits per region defining access permissions. RISC-V and the Kinetis K series choose a combination of these two approaches. For the user level, they choose a bit-based approach. For the supervisor level, a single bit indicates whether the supervisor should have full access (RWX) or the same permissions as the user.

All of these region-based MPUs have two distinct privilege levels, except for the RISC-V MPU, which has another privilege level that is meant for conventional use. This can be useful when for example simultaneously running two applications: one can be set to have more privileges than the other without having to do a context switch. A fourth privilege level in RISC-V, meant for hypervisor extensions, is planned for the near future [63].

In addition to setting permissions for processes running on the core, the Kinetis K and TI Keystone MPU allow permissions to be set individually for buses, e.g. a device connected by USB or the debugger.

AP[2:0]	Supervisor permissions	User permissions
000	No access	No access
001	RW	No access
010	RW	R -
011	RW	RW
101	R -	No Access
110	R -	R -

(a) Read and write permission encodings

XN	Supervisor permissions	User permissions
0	X	X
1	No access	No access

(b) Execute permission encodings

Table 3.2: Access permission encodings and corresponding permissions for the Cortex-M MPU [13]. The first column in each table shows the bit field value, and the second and third column show what the resulting access permissions for that region will be after setting this bit field, for both the supervisor and user mode.

3.2.3 Overlapping Regions

Overlapping MPU regions can be useful in order to use them more efficiently, and is in some cases even unavoidable. The behaviour of overlapping regions differs per MPU. We distinguish between three mechanisms for overlapping regions: priority, union and intersection.

Priority

With the priority ordering mechanism, regions are statically prioritized. The overlapping segment will adhere to the rules set by the region with the highest priority. The MPUs that make use of this mechanism are the Cortex-M, RISC-V, NIOS II and Xtensa LX7 MPUs. Generally, a higher region index corresponds to a higher priority. However, with RISC-V, a lower region index corresponds to a higher priority. In all four MPUs, any region has priority over the default access permissions.

To realize the scenario shown in the example from Figure 3.2 using the priority mechanism, the simplest solution would be to define a region with a higher priority and all access permissions for the user mode spanning from `0x20000` to `0x80000`. This is possible for the Xtensa LX7 MPU. Unfortunately, this is not possible with the other three priority MPUs due to their power of two constraint: `0x60000` is not a power of two. Two potential solutions to work around this constraint are:

1. Create two regions that together make up the full region. Region 0 from `0x20000` to `0x40000` and region 1 from `0x40000` to `0x80000`.

2. Create two regions. Region 0 allows all permissions and overwrites the background memory for the entire memory, that is from `0x00000` to `0x80000`. Region 1 blocks all user accesses from `0x00000` to `0x20000`; because it has priority, it takes precedence..

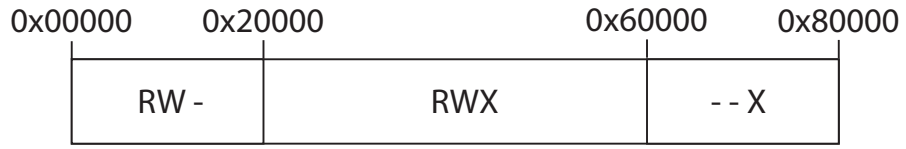

Figure 3.3: Example of the union mechanism for overlapping regions. Two regions are defined, one with read only permission from `0x00000` to `0x60000`, and the other with write only permission from `0x20000` to `0x80000`. In the overlapping segment the permission becomes the logical sum of permissions: RWX.

Union

The union mechanism gives priority to granting access over denying access in the case of overlapping regions. That is, the protection rights are logically summed together: like the boolean OR operator. An example of one scenario where it is useful to overlap regions with this mechanism is shown in Figure 3.3. Of the selected MPUs, the Kinetis K MPU makes use of this union mechanism.

Recreating our example (Figure 3.2) using the union mechanism can be simply done by creating an accessible region ranging from `0x20000` to `0x80000`.

Intersection

Using the intersection mechanism, priority is given to denying over granting access. The intersection mechanism behaves as the complementary of the union mechanism: like the boolean AND operator. Using the example of Figure 3.3 with the intersection mechanism would lead to the overlapping region permitting no access at all, contrary to the union mechanism where RWX would be allowed. Of the selected MPUs, the TI Keystone MPU makes use of this mechanism.

3.2.4 Default Access Permissions

The example from Figure 3.2 assumes that memory was by default unprotected. In most scenarios however, it is convenient to have all memory be protected by default: for instance, when having processes run on an operating system that should only have access to their own memory. This is also something that should be enforced in Tock, and therefore we will usually be exposing memory regions, in contrast to protecting them.

In some cases, overlap is inevitable: for instance, in some MPUs every region will overlap with their default access permission region as discussed in Section 3.2.4.

All MPUs offer a method to protect the entirety of memory by default:

Cortex-M Besides having 8 configurable regions, the Cortex-M supports a so-called background region. This region can be set to either block all accesses for both the user and supervisor, or to block accesses for the user only. Recall from Section 3.2.3 that the Cortex-M uses the priority mechanism. The priority of the background region is defined as -1: this implies every defined region will overwrite the background region.

MPU	User accessible	Supervisor accessible
Cortex-M	No	Yes/No
RISC-V	No	Yes
Kinetis K	No	No
NIOS II	No	No
Xtensa LX7	No	Yes
TI Keystone	Yes	Yes

Table 3.3: Access permissions for the default memory map not covered by any region.

RISC-V When no regions are set, the entire memory space for the RISC-V MPU is by default fully accessible to anyone. When the first MPU region is set, the default memory map is protected for the user; the supervisor still has full access to the default memory map.

Kinetis K On reset, by default region 0 is set up to map the entire 4 GB address space with RWX permissions for both the supervisor and user. When region 0 is changed to map a smaller chunk of memory, memory references that miss will fail, i.e., the actual background is protected. Because of the union mechanism, this means if region 0 is not changed this region overwrite the permissions of every other region. It is therefore imperative to adjust the settings of region 0 when using the Kinetis K MPU.

NIOS II Any memory reference that does not match with a valid MPU region will fail: the default memory map is protected by default for both the user and supervisor.

Xtensa LX7 By default, all supervisor references to memory not defined within an MPU region will succeed; user accesses will fail.

TI Keystone All memory is by default unprotected. In order to protect all memory, one can create a region with no access permissions that spans the entirety of memory. However, since the TI Keystone follows the intersection mechanism as covered in Section 3.2.3, any region that overlaps it will have no access permissions. Therefore, in order to properly protect memory, separate no-access regions must be created in all memory that is not covered by a region that does give access.

An overview of the default behaviour of each MPU for region misses is displayed in Table 3.3.

Since denying permissions by default is generally a desirable property, in subsequent sections we assume all memory not covered by a region is protected, by using the corresponding MPU-specific method to protect the default memory map for the user. Consequently, in contrast to protecting regions, we will talk about exposing memory regions within this protected memory.

3.3 Barrier-Based Protection

As mentioned in Section 3.2, contrary to all other region-based MPUs, the Nordic nRF51 is barrier-based. Instead of creating MPU regions, permissions are checked depending on the location of the instruction issuing the memory reference. Execution (X) permissions are always granted, regardless of where the call was made. Flash and RAM are seen as separate memory and are *both* divided in two regions: a supervisor and user region. To split these areas, the nRF51 supports a *barrier address* in both flash and RAM. These two barriers correspond to two registers in hardware. Every address with a lower address value than the barrier address is classified

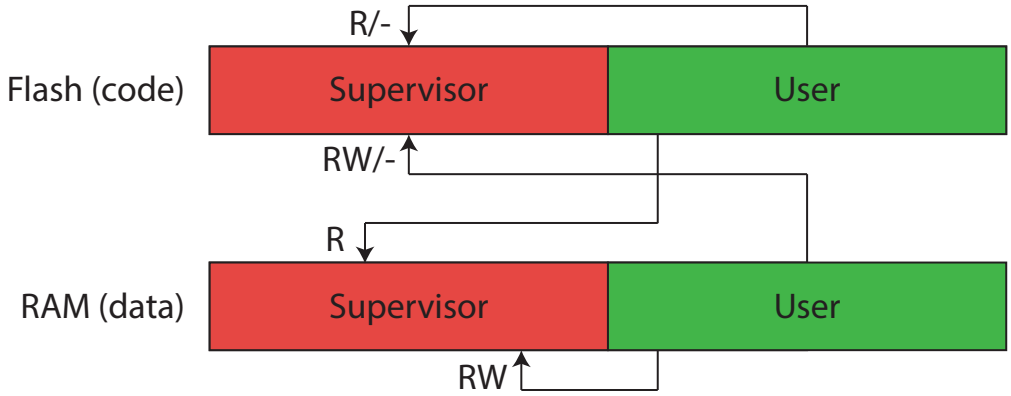

Figure 3.4: Overview of barrier-based protection for the nRF51. Shown are the memory references from user to supervisor memory and their possible access permissions. For instance, RW/- means the MPU can be configured to either allow both read and write accesses to that part of memory, or allow neither.

as the supervisor region, and everything with a higher address value is classified as the user region. The supervisor region has full access to the entire system, whereas the user region has full access to itself and configurable access to supervisor memory. This configurable access is different for flash and RAM:

- Memory references made from user flash to supervisor RAM are always read-only, whereas references to supervisor flash can be set to be read-only or no access.
- Memory references made from user RAM to supervisor RAM always have full access, whereas references to supervisor flash can be set to have full or no access.

This is illustrated in Figure 3.4, and shows that the functionality of the nRF51 has significant restrictions. Supervisor RAM is always accessible by user RAM; hence, if a user program has RAM assigned, it is always able to fully access supervisor RAM. This is a critical flaw that basically prevents the running of anything important in the kernel: user processes would be able to manipulate this data freely. Even if a program is only able to execute out of flash, it can still read supervisor RAM. The only part of memory that can be read and write protected in the nRF51 is supervisor flash, and even this part of memory is open to execute permissions. Therefore, this MPU is definitely not as secure as any of the region-based MPUs, and designing the interface around it is a secondary priority.

3.4 Memory Protection in Tock

Tock aims to guarantee memory isolation for processes by harnessing the power of MPUs as discussed in Section 2.4.2. Nevertheless, currently only the Cortex-M MPU [45] is supported on Tock owing to the lack of a generic MPU interface. Because of this, although Tock has been ported to other platforms that have a different MPU, such as the Kinetis K series and the Nordic nRF51, these ports currently lack an implementation for their MPU. Furthermore, the MPUs of platforms that Tock is currently being ported to, such as RISC-V, are also not able to be utilized.

Process	Flash	PAM	grant
`crc`	11,662 B	4,928 B	816 B
`ip_sense`	10,759 B	7,060 B	748 B
`ac`	7,694 B	4,172 B	724 B

Table 3.4: Memory requirement for each process shown in Figure 3.5, expressed in number of bytes.

In order to tackle this problem, we investigate the memory protection design in Tock and its existing shortcomings.

3.4.1 Process Memory Overview

Each process has three regions associated with it while running that require different permissions:

- Its stack, data and heap, also called its **process accessible memory (PAM)**. A process has full access to its PAM.
- The **flash** region where a process's code is stored. Tock also allocates a Tock binary format (TBF) header inside a process's flash that includes its flash size among others. The kernel uses the TBF header to traverse apps. Because the integrity of this header is essential to keep Tock running, processes do not have write access to their own memory (that is, they have R-X permissions).
- The **grant** region, where kernel memory is dynamically allocated. This region is fully protected by the MPU for the user level, so that it is not accessible in any way by a process.

Recall the overview of the organization of these regions shown in Figure 2.4. To clarify this overview and gain insight into realistic memory sizes, a numerical example of memory allocation and MPU regions is shown in Figure 3.5. In particular, these details are shown for three running processes: `crc`, `ip_sense` and `ac`, that are running on the Hail development board [1], containing a SAM4L microcontroller [64] with a Cortex-M4 processor and MPU [13]. These are processes that perform a cyclic redundancy check, broadcast periodic sensor readings and do an analog comparison, respectively, and their requirements in terms of memory are shown in Table 3.4. 0

Note that although this example may make it seem as if nine MPU regions are set concurrently, in fact only three MPU regions are set at the same time. The reason being that the operating system continuously context switches between processes and reconfigures the MPU, making it look as if the processes are running concurrently. In addition to these three regions, Tock allows processes to share their memory using an IPC region, as long as this region aligns to the Cortex-M MPU's restrictions. For instance, `ip_sense` could choose to share 1 kB of its data to `crc`. In that case, a fourth MPU region is defined inside the memory of `ip_sense`, that `crc` can access while it is executing.

One aspect this example aims to emphasize is that the power-of-two-constraint of the Cortex-M MPU plays a big role: the size of all MPU regions created in all three processes is rounded up to a power of two. The reason the PAM ends up being 15 kB for `ip_sense` and 7 kB for the other

[1] `https://github.com/lab11/hail`

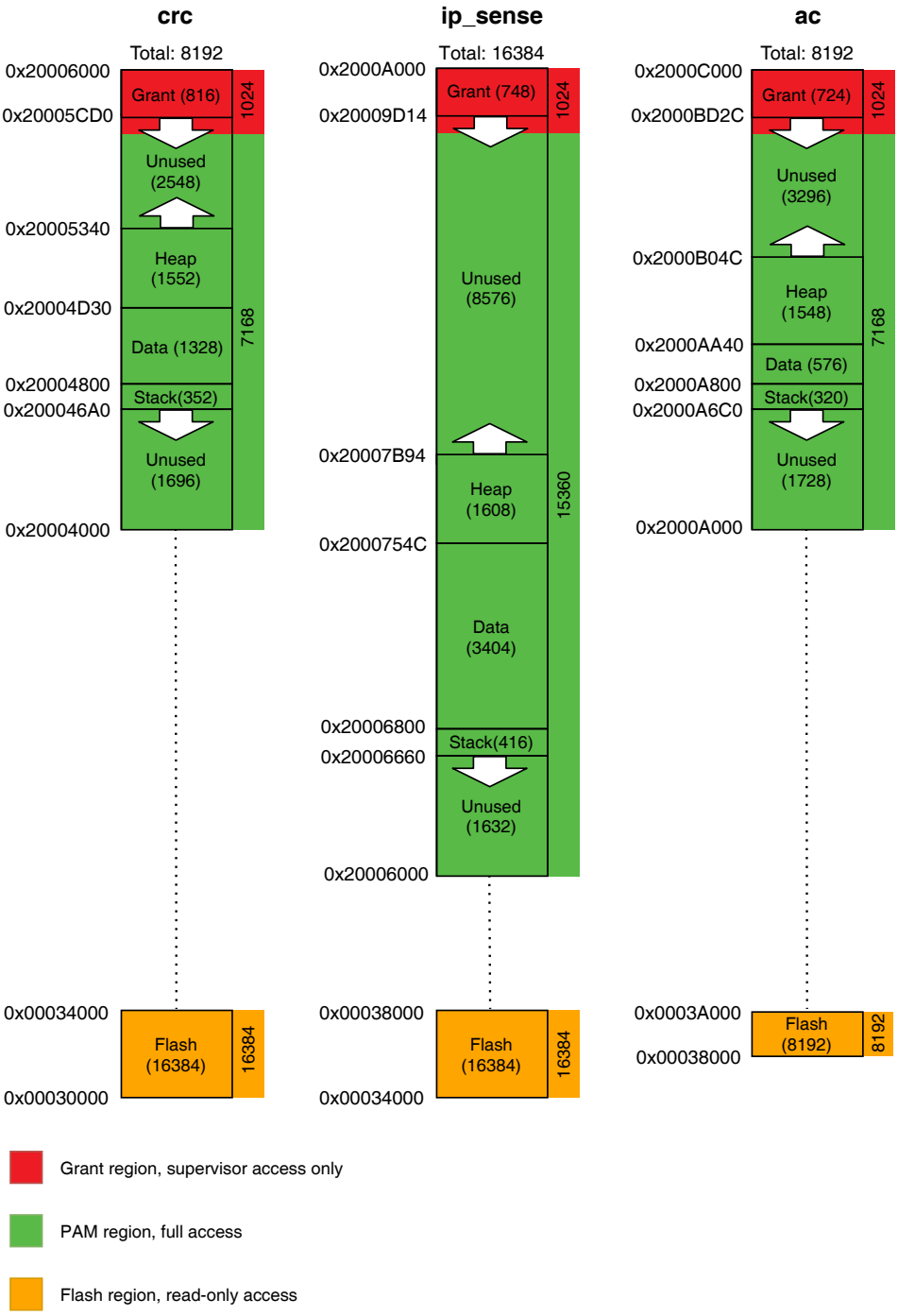

Figure 3.5: Overview of process memory with the existing Cortex-M MPU implementation in Tock. Shown is the memory and MPU allocation of three memory-contiguous processes running concurrently. A more detailed explanation is given in Section 3.4.1.

two processes is the priority overlapping property of the Cortex-M as discussed in Section 3.2.3, where in this case the grant region overlaps the PAM region. Another important consequence of power-of-two MPUs this example shows is the random access memory (RAM) of `ip_sense` being much bigger than the RAM of the other two regions with 16 kB instead of 8 kB, even though the majority of this 16 kB is unused and the minimum PAM requirement of `ip_sense` being 7,060 bytes. This is due to the kernel allocating some extra memory to processes' PAM in order to account for possible growth of the heap/grant regions, making the next valid size for `ip_sense` the nearest power of two: 16 kB.

When the heap grows into unused memory, no MPU regions have to be reconfigured as the unused memory is by default accessible for processes. Only after the PAM MPU region grows into the grant does the process run out of memory and crash. On the other hand, when the grant grows greater than its allocated MPU region of 1 kB, its MPU region has to be doubled in order to keep a length of a power of two. Every time it grows, it can keep doubling in this manner until it overlaps with the heap, after which the process runs out of memory. Having the grant grow in this manner leads to a finer granularity for low grant sizes, and a coarser granularity for higher grant sizes.

Note that the start address of `ip_sense` does not align with its size (`0x20006000` % `0x4000` $\neq$ 0), and therefore this region is ordinarily impossible to create on the Cortex-M MPU given its alignment requirements discussed in Section 3.2.1. To make this possible, Tock makes use of subregions provided by this MPU. A larger region is created that does align with the power-of-two requirement, and a number of subregions is disabled so that the correct amount of memory is exposed. The manner in which this is done is displayed in Figure 3.6.

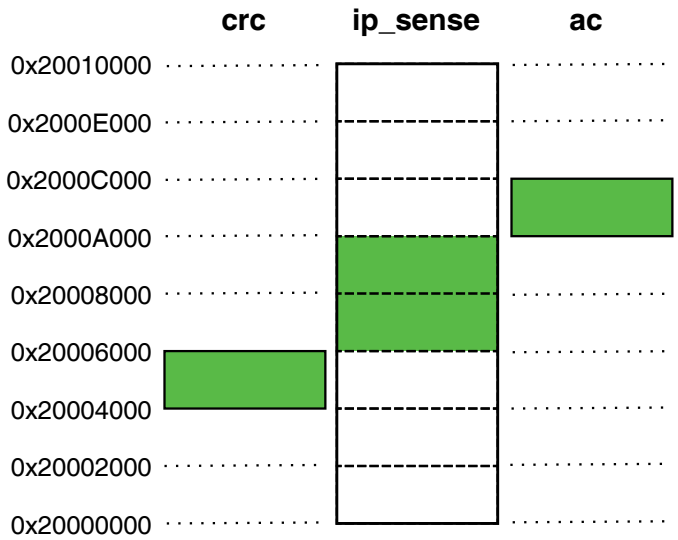

Figure 3.6: Overview of subregion usage for the PAM in the situation shown in Figure 3.5. Full lines represent regions, and dashed lines indicate subregions. In this case, subregion 4 and 5 are enabled for `ip_sense`, whereas the other two processes do not require using subregions.

3.4.2 Current Shortcomings

Tock distinguishes between different locations inside its kernel as shown in Figure 2.4. All code in Tock that involves the MPU is situated within this kernel, and is separated into three sections:

1. The memory for all processes is allocated by the **process manager** inside the core kernel. Here, the details of each process running within Tock are known, including its minimum required flash and RAM size. The length and location of each process's PAM, grant, flash and optional inter-process communication (IPC) region are determined, and accordingly, region requests are made for the MPU.

2. These region requests eventually arrive at the **MPU implementation**, which distinguishes itself as a microcontroller-specific peripheral driver. This implementation checks if the requested regions meet the requirements for the MPU, and is responsible for writing the region configuration to the hardware registers. The only MPU implementation that currently exists is that of the Cortex-M MPU. In this case, subregions are created if necessary (as was demonstrated in Figure 3.6), and the region configuration is written to two 32-bit registers on the MPU.

3. Linking the process manager and the Cortex-M MPU implementation together is the **MPU interface** situated within the hardware interface layer (HIL) in the kernel. The core of this interface is a Rust `trait`, defining functionality that every MPU implementation must implement. This trait contains functions attached to objects that are also called *methods*. Tock's microkernel design is what makes this interface so important, allowing different MPUs on different platforms to be coordinated through the use of a single abstraction.

Although there is nothing wrong with this structure, the existing implementation of it has a number of fundamental shortcomings that prevent the support of other MPUs.

MPU Interface

The MPU interface should be a generic abstraction that works for every MPU. Currently, code exists that is supposed to be an MPU interface: however, it is at this time not much more than a helper file for specifically configuring the Cortex-M MPU registers. It contains many Cortex-M specific details, and the trait defined inside it enforces properties that are unique to the Cortex-M. An overview of this existing trait is shown in Listing 5. Although the `enable_mpu` and `disable_mpu` functions are generic, the other two are not. The method `create_region` takes in the region index, start address, size and permissions and returns a `struct` – a Rust data structure that can contain different data types – called `Region`. This struct contains two 32-bit variables, corresponding to the two 32-bit region registers that the Cortex-M MPU supports. After a region has been created by `create_region`, the `Region` struct is written to the designated hardware registers by calling `set_mpu`. Rust's `self` keyword, which some of the methods take as a parameter, refers to the struct that implements this trait, similar to `foo` in `foo.bar()`.

In order to have this MPU interface be architecture agnostic, the `create_region` and `set_mpu` methods of this trait must be made more generic or replaced by other functions. Next, as currently only the permissions that the Cortex-M MPU supports are supported, and even the exact bits that have to be written to Cortex-M registers from Table 3.2 are specified, the defined permissions must be implemented in a generic way. Finally, a way should be found to not have the

```
pub trait MPU {
    /// Enable the MPU.
    fn enable_mpu(&self);

    /// Disable the MPU.
    fn disable_mpu(&self);

    /// Creates a new MPU region
    fn create_region(&self, region_index: usize, start: usize, size: usize,
    ↪ execute: ExecutePermission, access: AccessPermission,
    ) -> Region;

    /// Writes variables inside Region to hardware registers
    fn set_mpu(&self, Region);
}
```

Listing 5: Simplified overview of the existing MPU interface.

`Region` struct be Cortex-M specific. These main points and other small Cortex-M specific details must be moved out of the interface.

Process Manager

The existing process manager is far too interwoven with the constraints of the Cortex-M MPU, whilst it should not assume any details about MPUs. First of all, the existing process manager determines exactly what is passed to the MPU. Process memory is rounded up to a power of two to comply with the constraints of the Cortex-M MPU, and allocated in such a way that the start address always aligns to the size. Regions for flash, PAM and grant are created and set one by one with these exact constraints by calling `create_region` and `set_mpu` with these exact variables, not accounting in any way for the addresses or permissions the MPU supports. In addition, five empty IPC regions are created so that all eight MPU regions of the Cortex-M are used. Since constraints differ per MPU, the process manager should not make any assumptions like this, since even if such an assumption would work out for every MPU, it would in nearly all situations lead to an inefficient memory use for non-power-of-two MPUs.

Secondly, the process manager currently creates the grant region on top of the end of a process's RAM. It does this in order to prevent external fragmentation: should the grant be situated after the process's RAM, the subsequent process would not align to the Cortex-M region's alignment requirements anymore. In overlapping the grant with the process's RAM, the process manager assumes the behaviour on overlapping regions is the priority mechanism (Section 3.2.3). The grant MPU region is assigned a higher region index (which on the Cortex-M corresponds to a higher priority) with no permissions and thereby denies access to it for a process. Not all MPUs have this priority mechanism however, and some have a priority mechanism where a lower region index corresponds to a higher priority.

Thirdly, the process manager currently assumes a maximum of 8 regions, corresponding to the maximum number regions of the Cortex-M MPU as shown in Table 3.1. Ordinarily, a process requires three MPU regions: for its grant, PAM and flash. However, with the addition of IPC

MPU regions and the possible use of multiple regions to create a single region to provide more flexibility with the power-of-two requirement in mind, a process might require more regions. This could become problematic when the number of available regions on the MPU does not match the supported regions on Tock.

A fourth problem with the process manager is that it assumes memory is by default protected for the user and accessible for the supervisor, which is not the case for all MPUs as we saw in Table 3.3. The process manager currently relies on making calls outside of memory regions in supervisor mode, and when these calls fail at this time, Tock crashes. Also, the process manager assumes all user accesses to be blocked by default.

In all of these four scenarios the existing process manager blindly assumes the constraints of the Cortex-M MPU. These decisions must be made either by the Cortex-M MPU implementation instead of the process manager, or be made with knowledge of the utilized MPU.

MPU Implementation

Tock's process manager currently passes in regions that completely match the constraints of the Cortex-M MPU. In turn, the Cortex-M MPU implementation assumes region requests perfectly meet its requirements, and has no way to deal with the situation where this is not the case. For instance, when a region is not equal to a power of two, the existing behaviour is that the creation of a region fails and the entire operating system crashes. MPU implementations must either only receive valid region requests, or they must be able to work with region requests that do not exactly fit their constraints.

Chapter 4

Design Considerations and Methodology

Currently, the main obstacle preventing Tock from being an architecture-agnostic operating system is that the MPU interface is specifically tailored to the Cortex-M MPU. In this chapter, the design changes necessary in order to achieve a generic MPU interface are considered, and the various possibilities in doing so are explored. The primary objective of these design considerations is to generalize the MPU interface by removing the code that was specifically created for the Cortex-M MPU. A secondary goal is improving the performance and simplicity of the existing interface.

The structure of this chapter is as follows. We start by proposing storing the configuration of MPU regions in Section 4.1. Section 4.2 explores the challenges introduced by the overlapping of MPU regions, and the consequences that result when avoiding overlap. Next, we propose algorithms for each of the chosen MPUs that find the optimal region given their constraints Section 4.3. We follow up by presenting updates to the trait of the MPU interface in Section 4.4, and consider how to configure the MPU during kernel execution in Section 4.5. Finally, Section 4.6 lays out the methodology for creating an MPU interface.

4.1 Storing regions

In the existing design of Tock, the allocation of regions (asking the MPU to give back a valid configuration for a particular region request) is entangled with actually writing these regions to the MPU. This means that the entire process of communicating with the MPU regarding its regions, which for instance for the Cortex-M also includes checking for subregion alignment, is performed at every context switch. The maximum time a process is allowed to run on Tock is 10 milliseconds: therefore, context switching and thus reconfiguring MPU regions occurs at least 100 times per second (usually even much more), leading to a significant performance overhead. We propose making use of a Rust struct (a data structure that can hold multiple variables) for storing region information instead, so that regions do not have to be recomputed at every context switch. In this *region struct*, one entry corresponds to one MPU region, making its size depend

on the maximum number of regions an MPU supports. In every entry, a region is stored in two different ways: the logical and physical MPU region.

The *logical MPU region* consists of the start and end address of the region. This is what the process manager needs to know in order to properly align its next region. For instance, when a region request is made and the MPU implementation sends back the logical start and size, the end of the region can be computed by summing these two, and can be used as a start address for the next MPU region. Logical MPU regions exist to give the process manager information regarding the allocation of process memory, and separate regions by their access permissions.

Physical MPU regions are the actual variables that are written to the MPU corresponding to a logical region. Physical regions are opaque to the process manager, in the sense that the process manager has access to these register values, but does not derive any meaning from them. The reason both logical and physical MPU regions have to be returned is that these two do not always relate in a straightforward manner. For instance, in the case for the Cortex-M as shown in Figure 3.6, the logical region is the actual mapped memory whereas the physical region is the entire underlying memory region of which only two subregions are exposed. Physical regions are MPU-specific. The Cortex-M MPU for instance has two 32-bit hardware registers that are used for construing all regions, where one of these registers contains 3 bits to specify which region to write to. On the other hand, the Kinetis K66 has four 32-bit registers for each of its 12 regions. In order to enforce every MPU implementation to have a region struct but also allow them to be different, which is required because each MPU has different physical MPU regions, the proposed MPU interface uses a Rust *associated type*. An associated type makes the implementor of a trait specify the concrete type to be used in this type's place for the particular implementation. In this case, an associated type called `MpuConfig` is used to let each MPU implementation specify their own struct.

For instance, inside the Cortex-M MPU implementation, `MpuConfig` is set to `CortexMConfig`, being the region struct containing the logical and physical regions for each available region `CortexMRegion`:

```
pub struct CortexMRegion {
    location: Option<(*const u8, usize)>, // Logical region
    base_address: FieldValue<u32, RegionBaseAddress::Register>, // Physical region
    attributes: FieldValue<u32, RegionAttributes::Register>, // Physical region
}
pub struct CortexMConfig {
    regions: [CortexMRegion; 8],
}
```

With these changes, configuring the MPU can now be done by directly writing the registers with this stored data instead of (re-)calculating regions at every context switch. In line with this objective, calls for fulfilling a region request and writing the actual registers are separated. Fulfilling a region request – that is, returning logical and physical regions according to the MPU's constraint– is done with the `allocate_region` method. After fulfulling such a request, the logical regions returned by this method are used by the client to know what memory can be safely used: memory can be allocated in the corresponding region. The physical regions on the other hand are used in the `configure_mpu` method to write the actual MPU registers.

Storing regions has the benefit of not having to recompute regions at every context switch. However, since this recomputing in the original design also allowed for potential growth of certain MPU regions, and Tock supports dynamic allocation of heap and grant memory at runtime, a

new way to be able to grow MPU regions has to be found. Preferably, this should only happen on growth of the heap and grant, instead of at every context switch. Accordingly, we propose adding some functionality to the `brk()` and `alloc()` functions in Tock, that are called every time a process requires more heap and grant memory, respectively. When these functions are called, a check is done to see if memory is to be grown past the end of its MPU region. If the growth of memory does not exceed its logical end, this means the currently set MPU region is still sufficient to cover this growth, and nothing has to be done. If a region has grown outside of its bounds, all MPU regions are recomputed given the new constraints.

4.2 Overlapping Regions

The existing design assumes the priority mechanism for overlapping, where a higher region index corresponds to a higher priority as explored in Section 3.2.3. In overlapping the grant region with the process RAM, Tock uses that property to guarantee the grant not being accessible by a process. As overlapping mechanics differ per MPU, the use of overlapping regions has to be reconsidered.

4.2.1 Challenges in Overlapping

Not all MPUs have the same priority mechanism as the Cortex-M MPU. The RISC-V MPU has lower region indices corresponding to a higher priority on overlap. Furthermore, the Kinetis K and TI Keystone MPUs do not use the priority mechanism at all, but rely on the union and intersection mechanism, respectively.

For the problem of the opposite priority ordering for the RISC-V MPU, a possible and relatively easy solution is to invert the region indices in the MPU implementation, thereby writing the request for the first region (region 1) to the last region (region 16) instead and so on. Unfortunately, the problem with overlapping regions is much harder to solve for the Kinetis K and TI Keystone MPUs. The TI Keystone would in this specific case enforce the correct behaviour because of its intersection mechanism. However, MPUs that use the union mechanism cannot have these regions overlap. In this case, since of the chosen MPUs only the Kinetis K MPU uses the union mechanism, the straightforward engineering solution would be to not overlap the regions when using the Kinetis K MPU but change the start and/or size of the region that is being overlapped instead, utilizing its fine granularity. However, a solution like this would not be a generic solution at all, and undermines the main goal of this interface: being architecture-agnostic.

In addition, the reverse scenario has to be taken into account where one region overlaps over another that has fewer access permissions: for example, when overlapping a region with full access permissions over a part of a region with limited access permissions. In this case, the TI Keystone MPU is the one with the deviation in behaviour, since it will deny access to the overlapping part, whereas all other MPUs would allow access to it.

Furthermore, the process of overlapping regions leads to an added complexity in terms of returning logical regions. Specifically, where a logical region would normally have a 1-to-1 mapping with a physical region, it might have a M-to-N mapping in the case of overlap. For instance, in the example of Figure 3.3 for union and intersection MPUs, two physical regions would lead to three logical regions, whereas for priority MPUs it would lead to two logical regions. On the other hand, in a scenario of two physical regions where a region is situated in the middle of

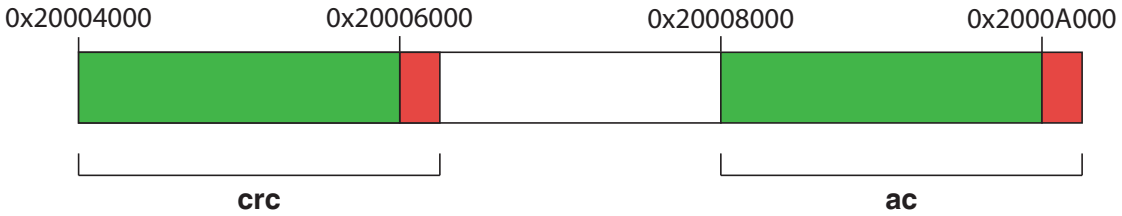

Figure 4.1: Example showing what occurs for power-of-two aligned MPUs when overlapping of regions is prohibited. Shown are two processes, `crc` and `ac` running concurrently. Now that the grant (red) does not overlap with the PAM (green) any longer, external fragmentation occurs.

another region, this leads to three logical regions for union and intersection MPUs, and one or three logical regions for priority MPUs, depending on the region indices.

4.2.2 Avoiding Overlap

As can be seen from the scenarios sketched out in Section 4.2.1, overlapping regions leads to a variety of complex situations. Therefore, the impact of avoiding overlap is now inspected.

For block-aligned MPUs, not being able to overlap is not a big problem since overlapping is only useful when really wanting to optimize the number of regions used. However, for general power-of-two aligned MPUs, the choice of not overlapping regions for the allocation of regions often leads to a significant amount of external fragmentation. One such situation can be shown by going through an example in which only the `crc` and `ac` applications from Figure 3.5 are running. Now that the grant is forbidden from overlapping and has to be placed after the process random access memory (RAM), processes will not align as neatly as they do now. Since the memory both processes require is greater than 4 kB, the next available region size for the process accessible memory (PAM) is 8 kB in order to align to a power of two. The smallest possible region size for the grants of both processes is 1 kB. For `crc`, this means the total process RAM (the combination of the PAM and the grant) is 9 kB. Now, because the next process contiguous in memory `ac` requires a size of at least 8 kB, the next available start address is at `0x20008000`, introducing external fragmentation of 7 kB. An overview of this situation is shown in Figure 4.1.

This is clearly not a preferred allocation of memory, and is the reason why in the existing design of Tock the PAM and grant overlap. However, solving overlap would require knowledge regarding the MPU, and thereby a significantly higher complexity in either the MPU interface or the MPU implementation. The challenge of overlapping is one of the main problems we try to tackle in this research, and both of the two proposed interfaces take another approach in solving this problem. The default behaviour in our design however will be forbidding overlap.

4.3 Flexible Region Ranges

As region constraints differ per MPU, it is impractical to decide upon the region ranges without knowledge of the MPU, which is something that is currently done in the process manager of Tock. We propose a design where the kernel communicates with the MPU in order to find a valid MPU region. On creation of a region, the kernel passes through a *minimum region size* the

MPU region should have, and a range in which this region should be placed defined by a *lower bound* and an *upper bound*. The objective of an MPU implementation is then threefold:

1. Create a region with a size that is greater or equal to the specified *minimum region size*, that is valid and as close this size as possible.

2. Align the start of the MPU region on or after the *lower bound* as specified by the kernel, as close to it as possible on a valid address.

3. Ensure the resulting end of the region does not exceed the value of the *upper bound*.

Since the region constraints are MPU-specific, the algorithm that determines the optimal region descriptors given these three inputs is specific to an MPU implementation. Recall from Section 3.2.1 that two fundamentally different implementations for the selection of MPUs are distinguished between: power-of-two aligned MPUs and block-aligned MPUs. The algorithm that finds the best suitable region is dependent on this type. We proceed by proposing algorithms for general power-of-two aligned MPUs and block-aligned MPUs given these three inputs, and in addition propose an algorithm for the Cortex-M MPU that makes use of its subregions as discussed in Section 3.2.1.

4.3.1 Block-Aligned Algorithm

An overview of the block-aligned algorithm that finds the most suitable region is shown in Algorithm 1. This is a pretty simple algorithm that rounds up $start$ and $size$ to the nearest multiple of the granularity, and checks if the given upper bound is exceeded. If it does, no valid region exists with these inputs and an error is returned. If it does not, and so the region is valid, the start and size of the MPU region for this request are returned.

Algorithm 1 Algorithm for calculating the nearest suitable region for block-aligned MPUs. Inputs are a minimum region size, a lower bound an an upper bound. Outputs are a start and a size.

1: $start \leftarrow lower\_bound$
2: $size \leftarrow minimum\_region\_size$
3: Round $size$ up to a multiple of $granularity$
4: Round $start$ up to a multiple of $granularity$
5: **if** $start + size \geq upper\_bound$ **then**
6: **return** error
7: **end if**
8: **return** $start$ and $size$

4.3.2 General Power-of-Two Aligned Algorithm

The algorithm we propose for power-of-two aligned MPUs is presented in Algorithm 2. It is equal to the block-aligned algorithm with the exception of line 3 and 4, where the size is rounded up to a power of two instead, and the start is rounded up to this size.

Algorithm 2 Algorithm for calculating the nearest suitable region for power-of-two aligned MPUs. Inputs are a minimum region size, a lower bound an an upper bound. Outputs are a start and a size.

1: $start \leftarrow lower\_bound$
2: $size \leftarrow \max(minimum\_region\_size, granularity)$
3: Round $size$ up to the next power of two
4: Round $start$ up to a multiple of $size$
5: **if** $start + size \geq upper\_bound$ **then**
6: **return** error
7: **end if**
8: **return** $start$ and $size$

4.3.3 Cortex-M algorithm

In the case of the Cortex-M, recall subregions can be used in order to partly circumvent the power-of-two constraint. This provides more flexibility in terms of starting addresses as was demonstrated in Figure 3.6, where no fragmentation occurred, whereas for other power-of-two MPUs there would be an external fragmentation of 8 kB between `crc` and `ip_sense`. In addition, this leads to an improved flexibility in region sizes: now, a size that is a multiple of an eighth of a power of two can be realized by having a different underlying region size than the actual MPU region. The existing algorithm that calculates the subregion configuration relies on the fact that the start and size that are passed in are fixed, making this problem relatively simple. In our design of the Cortex-M MPU specific implementation, we propose an algorithm that passes in the same parameters as the other algorithms, and uses subregions optimally in order to provide more flexibility: this is shown in Algorithm 3. In essence, this algorithm rounds the size up to the nearest power of two, and then tries a variety of start addresses and subregion sizes by rounding them up to possible values. A detailed explanation of the algorithm is as follows.

Algorithm Explanation

First, the size is rounded up to the granularity for the Cortex-M MPU, 32 bytes, if it is smaller than this. Otherwise, the size is rounded up to the next power of two P. Because subregions are $1/8$ of the region size and therefore also a power of two, the minimum possible subregion size is $1/8$ of the size rounded up to a power of two. As the granularity of subregions on the Cortex-M MPU is also 32, the subregion size is rounded up to 32 if it turns out to be smaller. This implies the actual underlying region size is higher than 256. The reason this was not rounded up in the first place is because subregions might not be necessary in a simple case such as $start = 0$ and $size = 32$.

Next, the while loop that tries to find a valid subregion configuration is entered by trying sizes equal to $P/8, P/4, P/2$ and P. This loop starts by rounding up the size to $subregion\_size$, so that when $size$ is a power of two and $start$ is divisible by $size$, a valid region configuration is found without having to resort to subregions, and the algorithm is finished. In such cases the algorithm finishes at line 9 by returning the valid *start* and *size*. If rounding up the size did not provide a valid region yet, the address under $start$ is found that aligns with the underlying (physical) region size, i.e., eight times the *subregion size*, as was the case in Figure 3.6. Given a value x and y greater than 0 and $x < y$, it is an invariant that a value can be found lying between y and $y-x$ that aligns with x. This trick is utilized to find an underlying region start that aligns with the start y

given the underlying region size x. Afterwards, a check is done to see if the underlying created region is big enough to cover the found *start* and *size*. If it is, the corresponding subregions that should be enabled are calculated, and the logical region and physical region are returned. If it is not, the procedure is repeated for a $subregion\_size$ of $P/4$, and afterwards $P/2$ and P. When after all these tries still no valid region has been found, the algorithm fails, meaning there is no valid region for the given constraints.

Simple Example

A simple example of the algorithm is one where immediately a size is found that is a power of two. For instance, consider the inputs $minimum\_region\_size = 0, lower\_bound = 0$ and $upper\_bound = 76$. The size is rounded up to 32 in line 2, the start remains unchanged and in line 8 the algorithm enters the `if` statement and returns the region $start = 0$, $size = 32$.

Subregion Example

An example of this algorithm using subregions is as follows. Consider a $minimum\_region\_size$ of 2.9 kB, a $lower\_bound$ of 3 kB and an $upper\_bound$ of 8 kB. Now, P is set to 4 kB, and the subregion size to 0.5 kB. In line 5, since $start$ already aligns to $subregion\_size$, it will remain 3 kB. Next, the while loop is entered and $size$ is rounded up to the $subregion\_size$ and becomes 3 kB. Since this does not align to a power of two, the else statement in line 14 is entered. An underlying region is constructed, with an $underlying\_region\_size$ of 4 kB and an $underlying\_region\_start$ of 0. This does not cover the end of the desired MPU region, as $4kB < 3kB+3kB$, so $subregion\_size$ is doubled to 1 kB. The $start$ again remains unchanged, since it still aligns with $subregion\_size$. The `if` statement in line 8 again fails, and another underlying region is tried given the new $subregion\_size$. Now, $underlying\_region\_size$ becomes 8 kB and $underlying\_region\_start$ becomes 0. This time, $8kB \geq 3kB + 3kB$ and the desired region fits within $upper\_bound$: the algorithm knows a suitable region can definitely be found and subregions are created. Here, $min\_subregion$ becomes 3 and $max\_subregion$ becomes 5. As Rust ranges are minimum inclusive and maximum exclusive, this means the fourth through sixth subregions will be enabled, corresponding to the memory from 3 kB to 6 kB. The algorithm has succeeded and returns the physical region, being the optimal region request given these inputs.

This algorithm can be applied to the problem shown in Figure 4.1, and will find a set of regions that reduce external fragmentation in most cases. Unfortunately, this does not completely get rid of external fragmentation, and therefore, this problem is something that we consider in both proposed MPU interfaces.

In comparison to the existing design, note that adding an algorithm computing flexible region ranges does lead to a small increase in flash and an increased complexity in the MPU-specific code: this is the price we must pay for becoming architecture-agnostic.

Algorithm 3 Algorithm for calculating the nearest suitable region for the Cortex-M MPU. Inputs are a minimum region size, a lower bound an an upper bound. Outputs are a start, a size and a possible subregion configuration, as this algorithm makes use of Cortex-M subregions.

$start \leftarrow lower\_bound$
$size \leftarrow \max(size, 32)$
Find the next power of two of the size, P
$subregion\_size \leftarrow \max(P/8, 32)$
Round $start$ up to a multiple of $subregion\_size$
while $subregion\_size <= P$ **do**
    Round $size$ up to a multiple of $subregion\_size$
    if $size$ is a power of two and $start \bmod size == 0$ **then**
        if $start + size \geq upper\_bound$ **then**
            return error
        else
            return $start$ and $size$
        end if
    else
        $underlying\_region\_size \leftarrow subregion\_size * 8$
        $underlying\_region\_start \leftarrow start$ rounded down to $underlying\_region\_size$
        $end = start + size;$
        $underlying\_region\_end \leftarrow underlying\_region\_start + underlying\_region\_size$
        if $underlying\_region\_end \geq end$ **then**
            if $start + size \geq upper\_bound$ **then**
                return error
            else
                $min\_subregion \leftarrow (start - underlying\_region\_start)/subregion\_size$
                $max\_subregion \leftarrow min\_subregion + size/subregion\_size - 1$
                return $start$, $size$ and subregion information
            end if
        end if
        $subregion\_size \leftarrow subregion\_size * 2$
        Round $start$ up to a multiple of $subregion\_size$
    end if
end while
return error

4.4 The MPU Trait

To enable the MPU interface to be compatible with various MPUs, Rust's *trait* mechanism is used to define a trait named `MPU` as discussed in Section 3.4.2. Data types implementing this trait (the MPU implementations) are normally forced to implement its methods. However, when the trait defines default method definition, the implementor is not forced to implement this method but will by default assume this implementation instead.

As an example, we take a look at the `number_total_regions` method in the `MPU` trait, that is a new method we present that returns the number of maximum regions for that MPU. Having this number of regions helps the process manager in making the right decisions, letting it know if regions are still available. The existing implementation assumes there will always be 8 regions, and has many dependencies relying on this invariant. For instance, it creates three regions for the flash, PAM and grant, and creates 5 additional regions that it sets to be empty by writing the corresponding register values and region indices to the MPU directly. Our changes remove these dependencies and make the implementation flexible with a variable number of maximum regions depending on the MPU.

Generally, MPUs contain a hardware register that defines the number of available regions; the MPU implementation should return this value to the interface. If such a register does not exist, the implementation of this method can be as simple as returning a predetermined number. The implementation of this method inside the `MPU` trait in the MPU interface is as follows:

```
fn number_total_regions(&self) -> usize {0}
```

Where `usize` is an unsigned integer that has a size dependent on the target. For example, on a 32-bit target it is 4 bytes. In this example of the method `number_total_regions`, the `MPU` trait defines a default implementation that returns 0. Now, if an MPU implementation does not have the `number_total_regions` method, it will assume this default implementation and return 0. We choose this approach of defining default implementations for every method inside the `MPU` trait in the MPU interface – over not defining a default implementation – in order to allow Tock to function on a microcontroller that does not have an MPU, by inheriting these default implementations.

For the Cortex-M MPU implementation, we do define a method for `number_total_regions`, overriding the default implementation. In this implementation,Cortex-M MPU's `DREGION` register is read out that contains the number of regions it supports:

```
fn number_total_regions(&self) -> usize {
    let regs = &*self.0;
    regs.mpu_type.read(Type::DREGION) as usize;
}
```

Where the `regs` variable is set to the hardware address at which the MPU is located, and we make use of the new register interface that is clarified in Appendix A.

4.5 Disabling the MPU

An additional design consideration is that of the MPU usage during kernel execution. Currently, instead of making use of the supervisor level while the kernel is running, Tock completely dis-

ables the MPU. In normal operating systems, disabling the MPU during kernel execution is not advisable: the operating system can contain bugs related to memory management, and correctly setting up the MPU can lead to such bugs being caught more easily. In Tock however, as the kernel is written in Rust and the `unsafe` keyword is sparingly used, disabling the MPU is a valid option. Considering this, the main decision criteria for using the supervisor level or not are complexity, power usage and performance:

Complexity Having the kernel run in supervisor mode would lead to an increased complexity, in the sense that the default access permissions for the supervisor level should then be similar across MPUs. For some MPUs, as discussed in Section 3.2.4, this is a complex undertaking. Therefore, in terms of complexity, preference goes to disabling the MPU.

Power Usage The difference in power usage for keeping the MPU enabled depends on the switching power consumption – that is, the power consumption of enabling and disabling the MPU – relative to the steady power consumption. Although not much data is available on this, the Kinetis K MPU datasheet states that disabling the MPU minimizes power dissipation [61].

Performance The effect on performance depends on the time it takes to enable and disable the MPU after every context switch.

In order to measure the power usage and performance, some tests were done aimed at measuring these factors for the scenario where the MPU was continuously enabled, and where it was disabled and re-enabled at every context switch for the SAM4L microcontroller [64] with the Cortex-M MPU. The results of these tests were an insignificantly distinguishable power dissipation and speed; therefore we choose to disable the MPU by default in order have a lower complexity.

4.6 Methodology

The most important design consideration in designing an MPU interface is that of the chosen level of abstraction. Ideally, code duplication should be minimized by having a concise interface and moving as much code into the process manager, which is reused for every MPU, as possible. On the other hand, no useful features or optimizations a specific MPU might have should be wasted, and therefore a certain degree of freedom should be given to MPU implementations, moving code away from the process manager. Seeking this Pareto point is the most difficult aspect in designing an MPU interface, and is the reason we proceed by designing two MPU interfaces. These interfaces are the result of seeking to answer research questions 2 and 3, respectively.

We specify the following requirements for the design of an MPU interface, with a descending order of importance:

Portability An MPU interface must be completely architecture agnostic. No assumptions regarding the target MPU must be made in the process manager and the MPU interface; everything specific to an MPU must be contained within the platform-dependent MPU implementation. Furthermore, it must be realizable on as many MPUs as possible.

Memory Usage Memory must be used in an efficient manner, keeping external and internal fragmentation to a bare minimum. Furthermore, the MPU interface should support the possible growth of regions during runtime.

Complexity As much complexity as possible should be moved to the process manager, resulting in a clearer abstraction and a simplification to the process of adding a new MPU implementation.

Due to the lack of comparable related work aiming to satisfy the same goals as our research, a comparison between this research and existing work is impossible. Hence, we evaluate our two MPU interfaces by comparing them to the existing MPU implementation in Tock and each other.

Chapter 5

A Region-Based MPU Interface

In this chapter, we present the region-based MPU interface. This interface is based around the idea of no information about the application of MPU regions being known in any way. Therefore, it does not assume any information about the current workings of Tock or any MPU: it is completely separate from memory allocation. Every region is created through the same function, and every region request is uncorrelated. This makes the interface straightforward and compact, moving most complexity to the client (which in the case of Tock is the process manager). In doing so, the interface remains independent of a specific application, as all changes will be contained to this client. A successful interface with such a generic design is future proof: it will not have to be changed when the memory layout of the operating system changes, and could even be used for other operating systems.

Unfortunately the region-based design, while seemingly very flexible and general, runs into problems due to the huge variations in MPU designs. The growth of MPU regions at runtime in particular is a big problem for the region-based design, due to the constraints of power-of-two aligned MPUs, and makes it unsuitable for Tock.

This chapter starts off by elaborating on the design of the region-based interface, discussing what design choices were implemented in order to create this abstract interface in Section 5.1. Afterwards, we discuss the implementation of this interface in Tock in Section 5.2,showing how this is done in Rust. Finally, the interface is evaluated in terms of its portability in Section 5.3.

5.1 Design

The key challenge in designing the region-based MPU interface is defining a method that can independently allocate regions, while encompassing all possible optimizations and accounting for the constraints of each MPU. In Section 4.3, we devise algorithms to find the optimal region given *minimum region size*, *lower bound* and *upper bound*. Unfortunately, such an algorithm does not support the growth of regions during runtime, since in that case extra constraints are necessary to take into account the memory that has already been allocated. For this reason, we propose two different type of region requests for the region-based MPU interface. Although the behaviour in finding a region is different for these requests, they both return a logical and

physical region that will be stored in the same way as discussed in Section 4.1, from which it is not distinguishable by which request each region was created.

5.1.1 Relative Region Request

The first of two types of region requests we propose is the *relative region request*. Relative region requests are meant for regions that have constraints regarding the minimum amount of memory to be covered, but no hard constraints regarding where exactly this memory region has to be located. In that sense, the procedure of a relative region request is very similar to the algorithm introduced in Section 4.3, with the exception of two additional features. The first is that when a lower bound is not defined, it will implicitly be set as the end of the previous region. Hence, relative regions will by default be aligned as close as possible to the end of the previous region. The second addition of relative region requests is that they allow for a region to be placed at least a specified distance from the lower bound. This can be either a specific address, or the last region that was set. This is a useful property when wanting to leave a gap between memory regions that are supposed to grow towards each other.

5.1.2 Absolute Region Request

The relative region request as discussed in Section 5.1.1 is suitable for the initial allocation of regions: allocating a region as flexibly as possible, making no guarantees with respect to the exact mapping of the start and end address. However, if memory has already been allocated and the size of an MPU region has to increase at runtime, relative regions are unfitting. In Tock for instance, when the heap and grant grow bigger during runtime and exceed their initial memory allocations, MPU regions must be recomputed without changing the start address for the process accessible memory (PAM), and end address for the grant, while making sure they do not overlap. On the other hand, updating these regions must also have some flexibility, since otherwise they can not meet all MPU-specific constraints while being MPU-agnostic.

Therefore, we propose *absolute region requests*, that promise to at least map a certain memory range, but have a configurable flexibility at their start and end addresses: regions must be able to support flexibility on both sides without giving up coverage of the already existing range. In other words, absolute region requests exist to specify a *start* and *end* address that definitely need to be covered by the memory region, and where (configurably) some other memory next to the start and/or end can be covered as well if necessary for proper alignment. The parameter *start_flexibility* defines the freedom in placement towards the left of the MPU region, i.e., the maximum number of bytes that the start may be moved to the left in order to find a valid start address. In fulfilling such a flexibility request, the MPU implementation should aim to deviate as little as possible from the start address, in order to not expose any unnecessary memory that would lead to internal fragmentation. The parameter *end_flexibility* defines the opposite: flexibility in placement towards the right of the MPU region, and other than that has similar requirements as the start flexibility.

We now propose an algorithm for both block-aligned and power-of-two aligned MPUs that aims to find the smallest region that covers all memory between *start* and *end*, has a start not more than *start_flexibility* bytes away from the start address, and an end not more than *end_flexibility* bytes away from the end address.

Block-Aligned Algorithm

The block-aligned algorithm that calculates the optimal region given an absolute region request is shown in Algorithm 4. First, the start/end are rounded down/up to a multiple of the granularity, respectively, to find a valid region start and end. Then, the algorithm checks if this start and end addresses fall within the specified flexibility. If they do, the smallest possible region covering the start and end has succesfully been found, and the region descriptors are returned. If not, the algorithm fails.

Algorithm 4 Algorithm that calculates absolute regions for Block-Aligned MPUs.

1: $region\_start \leftarrow start$
2: $region\_end \leftarrow end$
3: Round $region\_start$ down to a multiple of $granularity$
4: Round $region\_end$ up to a multiple of $granularity$
5: **if** $region\_start < start - start\_flexibility$ OR $region\_end > end + end\_flexibility$ **then**
6: **return** error
7: **end if**
8: **return** $region\_start$ and $region\_size$

Power-of-Two Aligned Algorithm

The proposed algorithm for power-of-two aligned MPUs given an absolute region request is shown in Algorithm 5. First, the minimum possible region size is found by rounding up the size of the region (*end - start*) to a power of two. If this is greater than the granularity, this is used as the size: otherwise, the granularity is used as the size. Next, a loop is entered that tries to find regions that fit within the specified lower and upper bound, that is within *start-start_flexibility* and *end+end_flexibility*. Since the start address has to be divisible by the size for power-of-two aligned MPUs, it is rounded down to a valid address. In line 8 of the algorithm a check is done to see if the region covers the memory from *start* to *end*. If it does, a region is found that conforms to both the power-of-two constraint and covers the desired memory region, so the region values are returned and the procedure is finished. If it does not, the size is multiplied by two, a suitable starting point is found again and the procedure is repeated. This eventually leads to a point where either a valid region is found that covers all the required memory, or a point where the constructed region lies outside of the desired boundaries. In the former case the region identifiers are returned to the client, and in the latter case the algorithm fails and returns an error.

5.1.3 Simultaneous Allocation

The choice of not overlapping regions for the allocation of regions unfortunately leads to a significant amount of external fragmentation for power-of-two aligned MPUs as elaborated on in Section 4.2. Seeing that in the existing design, regions are created consecutively as examined in Section 3.4.2, the MPU has no way of knowing what the implementor is looking for in creating regions. For instance, consider a situation in which an implementor would like to create three regions of 300 kB, 1 kB and 300 kB in a power-of-two aligned MPU. These regions will be rounded up to a power of two, resulting in regions of 512 kB, 1 kB and 512 kB. If regions are

Algorithm 5 Algorithm that calculates absolute regions for power-of-two aligned MPUs.

$region\_start \leftarrow start$
$region\_end \leftarrow end$
$region\_size \leftarrow \max(end - start, granularity)$
Round $region\_size$ up to the next power of two
while $region\_start \geq start - start\_flexibility$ AND $region\_end \leq end + end\_flexibility$ **do**
    Round $region\_start$ down to a multiple of $region\_size$
    $region\_end \leftarrow region\_start + region\_size$
    if $region\_end \geq end$ **then**
        return $region\_start$ and $region\_size$
    end if
    $region\_size \leftarrow region\_size * 2$
end while
return error

allocated consecutively, the first region will be created with 512 kB, after which the second region of 1 kB will be placed next to it, resulting in these two regions occupying the first 513 kB. Consequently, the next available start address for the third region would be 1024 kB, leading to 511 kB of external fragmentation. A scenario like this should be avoided at all costs.

We propose sending all the region requests simultaneously instead of consecutively, and letting the MPU implementation decide how to allocate regions according to the region size and alignment constraints, granularity, overlapping mechanisms and other optimizations. Doing so keeps the client (in the case of Tock, the process manager) and MPU interface generic, and moves all the MPU-specific details to the MPU implementation. For instance, in the aforementioned example, power-of-two aligned MPUs using the priority mechanism can choose to let the region of 1 kB overlap with the first 512 kB region instead, since the first region actually only needs 300 kB. Another solution would be to sort regions in such a way that external fragmentation is kept minimal, which in the case of this example would correspond to having the 1 kB region allocated last. In addition, MPUs with optimizations can reduce internal fragmentation even further, e.g. the Cortex-M can make this 512 kB region even smaller by only exposing subregions.

5.1.4 Access Permissions

Recall from Section 3.2.2 that access permissions differ per MPU. Considering the region-based interface should stay as generic as possible and not lose any functionality because of constrained MPUs, it should support all permissions for all MPUs: every possible combination of read, write and execute permissions should be specifiable.

In terms of user and supervisor levels, we support three possibilities for every permission: user and supervisor, supervisor only, or no access for both. We exclude giving the user more permission than the supervisor, since we do not see this ever being of use; furthermore this is not possible in most MPUs. This leads to 27 total possible options for read, write and execute permissions combined.

MPUs that use bit-based permissions for both the user and supervisor level (like the Xtensa LX7 and the TI Keystone) are perfectly able to support all of these permissions. However, MPUs that use any form of encoded permissions (all other chosen MPUs), would not be able to support

this. One of two possible types of behavior can be enforced for the MPU implementation when requesting a permission that is not defined:

1. The MPU implementation can fail, suggesting the client should request a region with defined access permissions.
2. The MPU implementation can pick the closest possible set of permissions that is defined. For instance, when a write-only permission is requested for the user level on the Cortex-M, of which the possible access permissions are shown in Table 3.2, the implementation will create and return a region with read and write permissions.

In order to not deviate from the requested behavior, we opt to make the MPU implementation return that the allocation has failed, and leave the responsibility for defining valid access permissions to the client.

5.1.5 Default Access Permissions

By default, the MPU should give a process access to its RAM and flash only: no access to any other part of memory. For this reason, when the MPU is enabled, it is ensured that the default memory map is protected for processes by using the corresponding MPU-specific technique as described in Section 3.2.4. This is easily possible for all MPUs but the TI Keystone MPU, that has the default memory map be accessible to user accesses by default as seen in Table 3.3. Therefore, in order to protect this default memory map, the MPU implementation of the TI Keystone must make sure to create regions with no permissions in locations where no regions by the user are defined on every call to the MPU. This would lead to a significant amount of complexity in the MPU implementation and a lowered number of maximum regions, since existing regions would have to be checked and possibly adjusted every time another region is adjusted/created, but would then show the correct behaviour.

Generally, operating in supervisor mode is used for the kernel. In line with this use case, on many MPUs having supervisor access implies being able to write to hardware registers, including hardware registers for the MPU itself. Since no other party than the kernel should by default have access to this, in our design we also assume the supervisor permission is reserved for kernel use, and therefore set the default memory map to be fully accessible for the supervisor permission.

Although most MPUs support this feature as shown in Table 3.3, the two MPUs that do not have this option built in by default are the Kinetis K and NIOS II MPUs. In order to achieve this feature on the Kinetis K MPU, a region must be defined that spans the entire memory and gives full access to the supervisor. Because of the union mechanism that the Kinetis K MPU adheres to on overlap, this implies that the supervisor always has access to the entire memory. As for the NIOS II MPU, since it follows the priority mechanism, if the region with the lowest priority is used to cover the entire memory, the MPU will function as if the default memory map is protected.

5.2 Implementation

We now present an implementation of the design of the region-based interface described in Section 5.1 in Tock. At its core, this implementation consists of a redesigned MPU interface and

process manager (Section 3.4.2) that are generic and therefore can be implemented for every MPU. In order to evaluate these changes and test their value, we present MPU implementations of this interface for a power-of-two aligned and a block-aligned MPU: the Cortex-M and Kinetis K MPU, respectively.

Now that regions can be created as either relative or absolute regions, a means is necessary to indicate which of these is being created on a region request. For this, a Rust *enumeration* or `enum` is used, that is a data structure that enumerates all possible values but can only take the value of one of them. Both `Relative` and `Absolute` are structs defined within this `enum`: one or the other can be picked for a region request.

5.2.1 Relative Region Request

The `Relative` enumeration contains four fields:

1. The `lower_bound` field indicates which address to use for the lower bound.
2. The `upper_bound` field indicates which address to use for the upper bound.
3. The `min_offset` field indicates the minimum distance that has to be between the `lower_bound` and and the start of this region.
4. The `min_region_size` field indicates the minimum size the region should have.

The `min_offset` field is useful for leaving a gap in memory that regions can grow into. In Tock for instance, when setting the grant after specifying the PAM, `min_offset` represents the minimum amount of unused memory between these.

The implementation of the algorithms for the relative region request is equal to the algorithms shown in Section 4.3, with two modifications to its input parameters. First, `lower_bound` is now wrapped in a Rust `Option`. When the `Option` is `Some`, it contains the value of the address to use as a lower bound for the region. If it contains `None`, the end of the previous logical region is used as the lower bound. Second, `min_offset` is added to `lower_bound` on the calculation of `start`.

In the initial setup of every region in Tock, only relative region requests have to be made, since there are at that point no strict constraints on where exactly the data has to be located.

5.2.2 Absolute region request

The `Absolute` enumeration contains four different parameters: `start`, `end`, `start_flexibility` and `end_flexibility`. Here, `start` and `end` represent the start and end address of the memory range that definitely has to be covered, and `start_flexibility` and `end_flexibility` define the flexibility in region placement as discussed in Section 5.1.2. The algorithms that have been implemented for the absolute region request are Algorithm 4 for block-aligned MPUs and Algorithm 5 for power-of-two aligned MPUs.

An absolute region request is useful in a situation where a region needs to grow. In the case of Tock, this means on growth of the heap or grant. On growing of the heap in Tock, `brk()` is called, and the region stored in the region struct for the PAM is removed. Then, a new region is created that has the same start, and a start flexibility of 0, but a different end corresponding to the newly required memory and an end flexibility such that the PAM does not extend into the

grant. On growth of the grant a similar procedure takes place: the Tock kernel calls `alloc()`, the grant region is removed and an absolute region request is made with a fixed end and flexible start.

5.2.3 Simultaneous Allocation

Section 5.1.3 proposes sending all region requests simultaneously in order to be able to enable the MPU implementation to decide on optimizations. In the implementation, this is done by first storing all region requests in a `regions` array, with each index corresponding to one region. The details for flash, PAM, grant and optional inter-process communication (IPC) regions for a process are stored in this array and sent to the MPU implementation at the same time, allowing this implementation to allocate in a more flexible way and account for the constraints and optimizations, a feature that is mostly useful for exploiting overlapping regions. Since in the region-based interface in Tock the PAM and grant are separated by a certain offset as explained in Section 5.2.1, there is no overlap. Therefore, no special algorithm for overlapping regions is created in both the Cortex-M and Kinetis K MPU implementations.

5.2.4 Access Permissions

A `Permission enum` is defined with three options: no access, supervisor access only and full access. Each read, write and execute permission inherits this `enum`, allowing the flexibility of all possible access permissions. The process manager requests a certain read, write and execute permission and sends this to the MPU implementation. These requests are then matched to register values. For instance, consider setting the execute permissions for the Cortex-M MPU, which is done by setting one bit as shown in Table 3.2b. If the supervisor and user are both given execute permissions, this bit is set, and if they are both denied access permissions this bit is not set. If a region is requested with supervisor execute permissions and no execute permissions for the user, the Cortex-M implementation returns an error, and Tock will panic with a message stating the unsupported permission.

To implement the default memory map to be accessible for the supervisor and inaccessible for the user, on enabling the MPU activates the background region for the Cortex-M MPU as explored in Section 3.2.4, and creates a supervisor region spanning the entire memory for the Kinetis K MPU.

5.2.5 Overview

An overview of the implementation of the region-based MPU interface in Tock is shown in Listing 6. The core of this interface are the methods `allocate_regions` and `configure_mpu` in the `MPU` trait.

In `allocate_regions`, a `Region` struct is passed to the MPU implementation, that contains the `enum` corresponding to the desired region request, in addition to `read`, `write` and `execute` permissions. The possible permission options are defined in the `Permission enum`. The process manager calls `allocate_regions` in three situations:

1. When Tock boots, setting up each process's memory.

2. When `brk()` or `alloc()` are called, recomputing the MPU regions for that process.
3. When an IPC region is added.

The return value of `allocate_regions` is the `config` containing the MPU-specific struct as proposed in Section 4.1. Finally, `configure_mpu` is called to write this struct to the hardware registers, which occurs in these three situation in addition to at every context switch.

Note that all methods have a default definition as explored in Section 4.4, enabling platforms that do not have an MPU to run Tock. For instance, as *allocate_regions* expects a Rust `Result` to be returned, the default implementation always returns `Ok`.

5.3 Evaluation

The region-based interface is evaluated by analyzing the impact of both relative and absolute region requests on memory. For this analysis, the allocation of memory and MPU regions in the example of three running processes for the existing implementation as shown in Figure 3.5 is considered, but now by using the region-based interface. In order to have a fair comparison this exact same situation is thoroughly analyzed, i.e., the same memory requirements and the same MPU. Hereafter, we discuss how this would apply to other MPUs, and provide a summary of the strengths and weaknesses of the region-based interface.

5.3.1 Relative Region Request

The result of using the Cortex-M subregions and our new algorithm for this as discussed in Section 4.3.3 leads to a process memory layout occupying less memory, and is shown in Figure 5.1. The corresponding subregion alignment is shown in Figure 5.2. Unfortunately, using the region-based interface for a power-of-two aligned MPU does lead to some complications. In order to see what these are and understand the exact procedure of the interface in practice, we now proceed by going through the allocation of MPU regions in a step-by-step manner.

Flash

The start of the flash region of `crc` is placed directly next to the end of the flash that the kernel needs, which in this case is `0x00030000`. To do this, the process manager makes a relative region request with `lower_bound` set to `Some(0x00030000)`, a `min_offset` of 0 from that location, an `upper_bound` of `0x20000000` corresponding to the end of flash, and a `min_region_size` of 11,662 bytes (as per its requirements shown in Table 3.4). In the old design, this would get rounded up to 16 kB: now, by using Cortex-M subregions and our proposed algorithm in Section 4.3.3, a region of 12 kB can be achieved by only exposing the first six out of eight subregions of a 16 kB region. When the process `ip_sense` is context switched to, its regions are set up given the constraints of previously allocated memory. This process requires a `min_region_size` of 10,759 B and *lower bound* set to `Some(0x00033000)`. Using subregions, a region of 12 kB can be created starting from that location by creating an underlying of region of 32 kB that aligns with `0x00030000` and exposing subregion four through six. As for the flash of `ac`, since the next power of two is 8 kB and the starting address (`0x00036000`) already aligns with this, a region can be created without resorting to subregions.

```rust
/// Access permissions.
pub enum Permission {
    //                  Supervisor  User
    //                  Access      Access
    NoAccess,       // No          No
    SupervisorOnly, // Yes         No
    Full,           // Yes         Yes
}

/// MPU region type.
pub enum RegionType {
    /// Absolute region, anchored to start and end
    Absolute {start: usize, end: usize, start_flexibility: usize,
    ↪ end_flexibility: usize},

    /// Relative region, flexible in alignment but having a minimum size
    Relative {lower_bound: usize, upper_bound: usize, min_offset: usize,
    ↪ min_region_size: usize},
    }

/// MPU region.
pub struct Region {
    region_type: RegionType,
    read: Permission,
    write: Permission,
    execute: Permission,
}

pub trait MPU {
    /// Enables the MPU.
    fn enable_mpu(&self) {}

    /// Disables the MPU.
    fn disable_mpu(&self) {}

    /// Returns the total number of regions supported by the MPU.
    fn number_total_regions(&self) -> usize {0}

    /// Allocates a set of logical regions for the MPU.
    /// Returns config with the resulting logical and physical regions.
    fn allocate_regions(regions: &mut [Region]) -> Result<Self::MpuConfig>
    {Ok(())}

    /// Configures the MPU with the provided region configuration.
    fn configure_mpu(&self, config: &Self::MpuConfig) {}
}
```

Listing 6: Simplified overview of the region-based MPU interface.

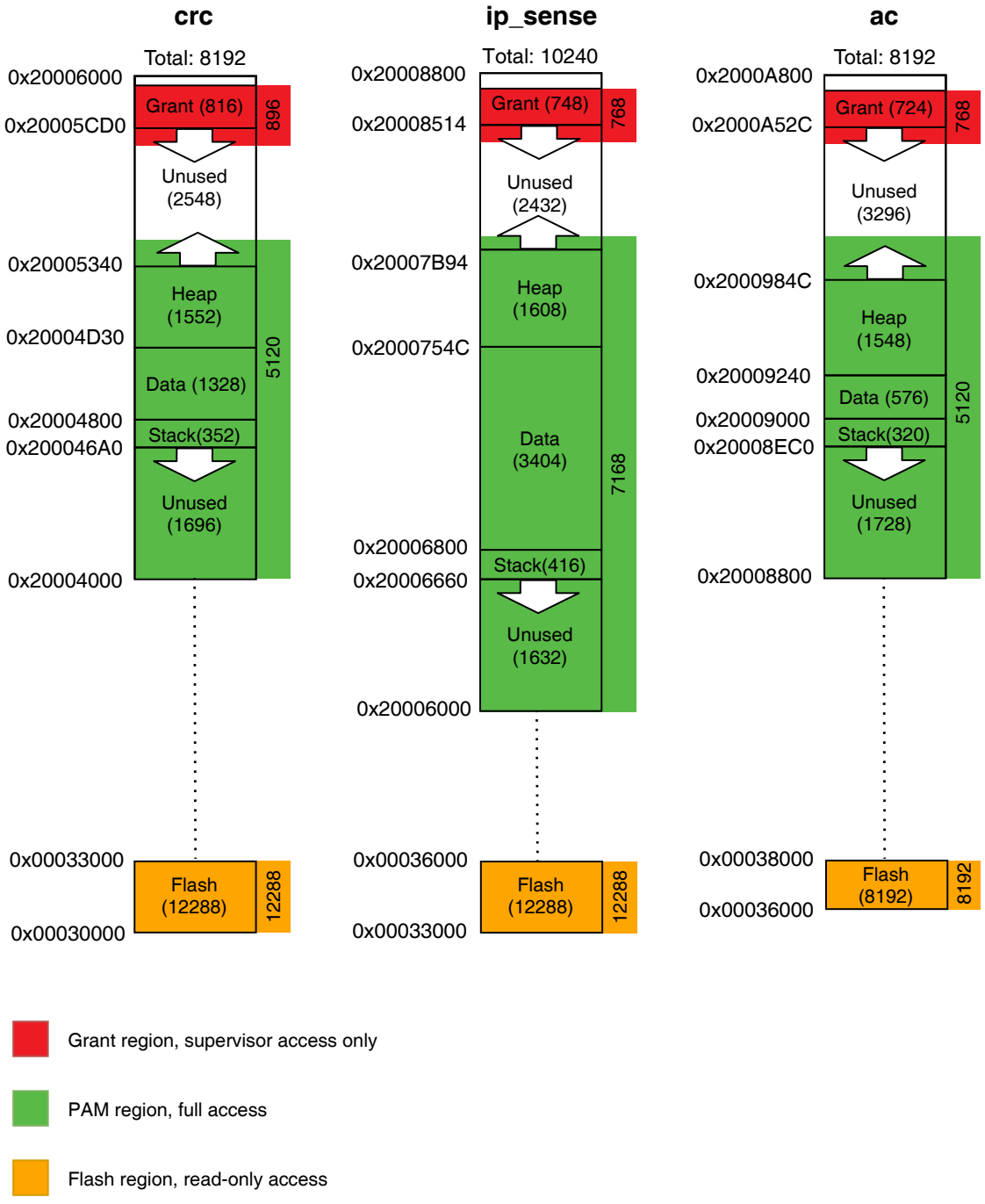

Figure 5.1: Overview of process memory using the region-based MPU interface for the Cortex-M MPU in Tock. Shown is the memory and MPU allocation of three memory-contiguous processes running concurrently using this interface. A more detailed explanation is given in Section 5.3.

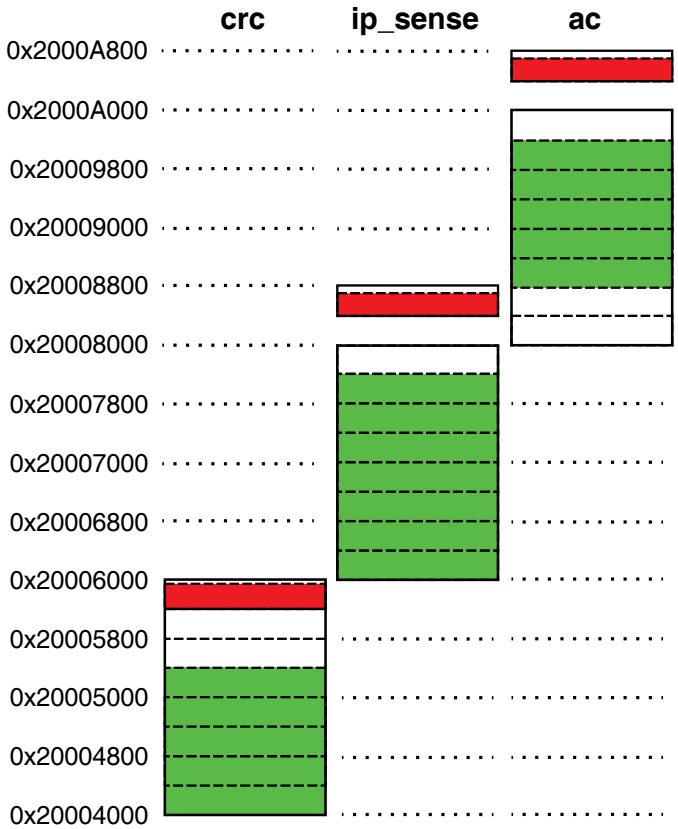

Figure 5.2: Overview of subregion usage for RAM in the situation shown in Figure 5.1. Full lines represent regions, dashed lines subregions, and the green and red areas are the PAM and grant, respectively. For clarity reasons, grant subregions are not shown.

There is little internal and no external fragmentation in flash, contrary to the situation in Figure 3.5, now that Cortex-M subregions are exploited. Note that although this situation has such small fragmentation, unfortunately many other situations in power-of-two aligned MPUs will lead to at least some external fragmentation.

RAM

The first region request for random access memory (RAM) is a relative region request for the PAM of the first process. Given the required kernel memory at the start of RAM, the `lower_bound` is set to `Some(0x20004000)`. Now, as the PAM requires a size of 4,928 bytes as seen in Table 3.4, the `min_region_size` is set to this accordingly, and `upper_bound` is set as the end of RAM. Making use of our Cortex-M subregion algorithm (Section 4.3.3), this leads to a logical region of 5,120 bytes, which uses five subregions in an underlying region of 8,192 bytes.

The subsequent region to be allocated (in the same context switch) is the grant region. Considering some unused space between the heap and the grant is necessary in order to allow these regions to grow, the `min_offset` parameter is used. The original design would add 2,048

bytes to the size of the PAM before allocating it: now, `min_offset` is set to 2,048 bytes. Setting `lower_bound` to `None` (Section 5.2.1) makes it become the end address of the PAM. The start address for the grant then becomes the sum of these two parameters. With the `min_region_size` of the `crc` grant size being 816, this results in a grant region of 896 bytes, where seven out of eight subregions of a region of 1,024 bytes are exposed. This is where the first problem of the region-based interface surfaces: an external fragmentation of 128 bytes occurs between `crc` and `ip_sense`, because no assumptions are made regarding the allocation of the next process. Optimally, the grant would have been placed at the end of this process as is done in the existing implementation, so that both the grant and PAM would have more room for growth.

For `ip_sense`, the combined size of the stack, the unused stack growth space, data and heap is 7060, making the best the Cortex-M can do exposing the first seven subregions in a region of size 8192, resulting in a logical region of 7168 bytes ending at the address `0x20007600`. With a `min_offset` for the grant region of 2048 bytes and its given size requirements, its start address will be allocated at `0x20008400` and the grant region will have a length of 768 bytes, enabling the first six subregions. In a similar manner, RAM for `ac` is allocated. Using its constraints an underlying region of 8 kB is created at `0x20008000` that enables use of the third through the seventh subregions, leading to the actual covered range being from `0x20008800` to `0x20009C00`.

By looking at these relative region requests, it becomes apparent that having a region-based interface for power-of-two aligned MPUs leads to external fragmentation, even though this is somewhat reduced by using the Cortex-M MPU's subregions. In the case of non-growing regions such as flash, having internal fragmentation is as bad as having external fragmentation since the memory is not going to be used in either case (would memory not be allocated consecutively, external fragmentation would even be slightly better since a smaller region could possible be placed within in this fragmented part). However, for growing regions internal fragmentation is preferred, as this means regions have more memory to grow into. This is the first weak point of the region-based interface.

5.3.2 Absolute Region Request

Consider the setting in which the PAM and the grant require growth during runtime, and request this by using an absolute region request. As the initial heap size of `crc` is 1552 and it has 192 bytes to grow before it exceeds its MPU region, every call to `brk()` for its initial growth will lead to no change in the MPU regions. When it grows bigger than this, `brk()` will call an absolute region request with a `start` of `0x20004000`, a `start_flexibility` of 0, an `end` corresponding to the new heap end and an `end_flexibility` of the difference between the end of the heap and the beginning of the grant. This results in a PAM of 6144 bytes: the same underlying region as before that has an additional subregion enabled. In this case, the growth of the PAM is perfectly fine, and similar to the existing scenario. Growth of the grant however leads to a less preferable scenario. The grant region now has to grow after only 80 bytes, instead of 208 bytes as in the existing design. With its end address being set already, even with subregions there is no way to allocate a bigger region that aligns to this end address, and the algorithm fails, resulting in the process running out of memory and failing. This brings us to the second and main weak point of the region-based interface: the inability to efficiently handle growing regions that have different access permissions given power-of-two constraints.

Even if the end address of the grant is during runtime somehow aligned to the start address of the next process and the contents of the grant is somehow realigned to reduce external fragmentation solving the first problem as discussed in Section 5.3.1, there is an abundance of possible

scenarios in which one of the growing regions would result in an impossible scenario for the region-based interface, and where a growth of 1 byte would already lead to the process running out of available memory, making it crash. Other power-of-two aligned MPUs than the Cortex-M MPU that do not have subregions suffer from this problem in an even more pronounced manner. For instance, giving the PAM an initial size of 8 kB, setting up a grant region with a distance 2 kB after it and consequently letting the PAM grow by merely one byte will lead to no valid region being available anymore. In addition, the process that gets allocated after it will not be able to find a start address that leads to no external fragmentation.

5.3.3 Summary

To summarize, two problems caused by the alignment constraints of power-of-two aligned MPUs make a region-based interface unsuitable as a generic MPU interface:

1. Allocating a left-growing region without knowing the start address of the next process often leads to external fragmentation, which could have instead been used as more space to grow into.
2. Because it is not known at allocation which regions grow towards each other, they can not be aligned accordingly, leading to a sub-optimal region placement.

Even were in some way a method be devised to know allocation beforehand and solve the first problem, e.g. moving the grant region after the PAM start address for the next process in memory is known, the second problem is unsolvable if an interface is to remain generic. Furthermore, although these problems can be reduced by using the properties of overlapping regions, since this is not allowed in the region-based interface, the efficiency goes down. Regions can not overlap if an 1-to-1 mapping is to be maintained, and adding an M-to-N mapping would lead to a significant increase in complexity. To solve these problems, a request is needed that takes in two regions (with differing access permissions) growing towards each other at the same time, and allocates memory accordingly.

Note that block-aligned MPUs on the other hand do not have these two problems, since alignment is consistent through memory and there are no complicated non-uniform alignment constraints. With block-aligned MPUs, the region-based interface performs much better than the existing interface, and is in fact optimal in placement given certain parameters. In addition, the region-based interface would be a valid solution – even for power-of-two aligned MPUs – would memory regions not require any growth during runtime. It is the combination of power-of-two aligned MPUs and growing regions that unfortunately make the region-based interface unsuitable for applications such as Tock.

Chapter 6

A Process-Based MPU Interface

In this chapter, we present the process-based MPU interface. Where the region-based MPU interface discussed in Chapter 5 is targeted at being as generic as possible and contains only one call to the MPU for creating regions, the process-based interface supports specific functionality for two regions that are growing towards each other. Our design for the process-based interface solves the problems introduced in Section 5.3.3, allowing for a minimal amount of external fragmentation and a more flexible support of region growth.

In this chapter, we first provide a detailed design of the process-based interface in Section 6.1. Subsequently, in Section 6.2 we describe its implementation in Tock. We end this chapter by evaluating the process-based interface in terms of memory usage, portability and performance in Section 6.3.

6.1 Design

In the process-based design, instead of remaining fully agnostic of what memory is being used for, assumptions are made regarding the target use case. This enables a design in which two memory regions that should have different access permissions grow towards each other, which is used in for instance Tock's process random access memory (RAM). For this reason, we propose two different region requests. The first is a *non-growing region request*, that has the goal of purely allocating a region as optimally as possible. The second is a *growing region request*, that exists for allocating two MPU regions with differing permissions growing towards each other.

6.1.1 Non-Growing Region Request

The region request for a non-growing region uses the same parameters and algorithms as mentioned in Section 4.3: *minimum region size*, *lower bound* and *upper bound*. Contrary to the region-based interface as shown in Section 5.1.1, in non-growing allocation there is no need to account for gaps in memory or aligning to the previous region within the same region request, and therefore only these three parameters are necessary for region alignment. Regions are still forbidden from overlapping: if a user requests a non-growing overlapping region, the implementation should fail.

6.1.2 Growing Regions Request

In order to deal with memory regions with differing permissions that grow towards each other, we propose the growing regions request. This request takes in the same parameters as the non-growing region request. Now, *minimum region size*, *lower bound* and *upper bound* are used in order to set the size of the memory containing both of these two growing regions. In addition, two new parameters are defined corresponding to the initially required size of each growing region.

In the process-based interface, the MPU implementation decides how regions are used and the client does not have full control of every region. This is in contrast to the region-based interface, where the the client knows which region index corresponds to a growing region, allowing it to clear that region entry and recompute it using an absolute region request. Shifting this responsibility to the MPU implementation allows optimizations unique to MPUs to be used, which is useful especially in the case of growing regions. For instance, in some MPUs the use of multiple (overlapping) regions can be exploited. In others, subregions can be used to divide memory in a more fine-grained manner. To enable the process manager to update this region on runtime growth, which is in this interface not in control of every region anymore, we propose a second method that exists solely to update the memory protection within the greater region at runtime: the *update growing regions request*. It does not take in any constraints regarding the greater region, considering this is already known at runtime, and purely changes the alignment of the regions within. On runtime region growth, only this updating growing regions request will need to be called, contrary to recomputing all regions as discussed in Section 4.1.

6.1.3 Access Permissions

Similar to the region-based interface, by default the MPU should give a process access only to its own memory, and no access to any other part of memory as discussed in Section 5.1.4: user accesses should be blocked on access violations. As for supervisor accesses to the default memory map, we decide to not take this into account two reasons:

1. Generally, the supervisor mode is used for the kernel to ensure some memory is not accidentally overwritten. However, since Tock's kernel is written in Rust, and the `unsafe` keyword is sparingly used, unwanted memory references should be extremely rare.
2. Currently, in Tock there is no use case for the supervisor permission, as the MPU is disabled while the kernel is running as discussed in Section 4.5.

For these two reasons, in the process-based design, all MPU implementations are configured in such a way that memory not lying within a region is by default protected for user-level processes, and the supervisor permission is not taken into account.

Disregarding supervisor permissions reduces the number of possible permission combinations defined in the MPU interface from 27 to 8. Of these resulting eight combinations, an additional three combinations are removed: 'write only', 'write and execute', and 'no access'. The first two are removed because these are combinations that are generally not useful in operating systems and especially not in Tock, as scenarios in which it is useful to have write permissions and no read permissions are extremely rare. The argument for removing the 'no access' permission is that the default memory map is not accessible anyways. Only in the case of overlapping regions for power-of-two aligned MPUs would a no access permission potentially be useful in order to have

a more fine-grained region alignment. However, since overlapping is only allowed within the growing region request in the MPU implementation, and this does not involve the permissions defined in the MPU interface as registers can in that case be directly written, there is no need for a 'no access' permission. This leaves only 5 permission combinations that are defined in this interface: execute only, read only, read execute only, read write only, and full access (RWX).

6.2 Implementation

We now present our implementation for the process-based interface in for Tock, following the design requirements of Section 6.1. The main additions to Tock are in the process manager, the MPU interface and MPU implementations for the Cortex-M and Kinetis K MPUs. In addition, minor changes are made to some other files in order to make this interface efficiently work with the existing version of Tock (version 1.2).

6.2.1 Non-Growing Region Request

The method used for creating a non-growing region request is `allocate_region`. As Tock has no use case for overlapping regions other than RAM that now has its own function, allocating all regions at the same time as was done in the region-based interface (Section 5.2.3) only increases complexity; therefore, `allocate_region` takes in constraints for one region at a time, allocating regions consecutively. However, instead of an upper bound, now a maximum size in which a region can grow is passed instead: that is, the size between the lower and upper bound. This leads to less error checking (for instance, making sure the passed upper bound is not smaller than the lower bound) and allows us to pass an integer instead of a pointer, reducing complexity by avoiding some type castings. In particular, the following parameters are passed through the `allocate_region` method:

1. The `unallocated_memory_start` field indicates which address to use for the lower bound.
2. The `unallocated_memory_size` field indicates the maximum size in which a region can grow. For instance, if this field is 8 kB and `unallocated_memory_start` is 16 kB, this is equal to *upper bound* being 24 kB.
3. The `min_region_size` field indicates the minimum size that the region to be created should have.
4. The `permissions` field indicates which of the five defined access permissions must be set.
5. The `config` struct contains all logical and physical MPU regions as discussed in Section 4.1.

Besides being used to see what regions are free (and so in which index the new region should be stored), the `config` struct is used by the MPU implementations of both the Cortex-M and the Kinetis K MPU to ensure none of the existing regions overlap with the region to be created. This is done by going over all the logical regions in the `config` struct and checking if any of them overlap with the memory chunk starting from `unallocated_memory_start` with a size of `unallocated_memory_size`.

The implementation of the region alignment follows the algorithm described in Section 4.3.3 using subregions for the Cortex-M MPU, and the block-aligned algorithm of Section 4.3.1 for the Kinetis K MPU. We would like to emphasize that the implementation for a block-aligned MPU

like the Kinetis K MPU is much simpler than the implementation of a power-of-two aligned MPU like the Cortex-M MPU, especially since Cortex-M subregions, when used properly, provide a great improvement to flexibility. In fact, the body of `allocate_region` in the Kinetis K MPU is only 23 lines long, whereas that of the Cortex-M MPU is 76 lines long.

6.2.2 Growing Regions Request

The implementation of the growing regions request in Tock consists of two methods: one for allocating process memory and another for updating it. As no MPU region has to be created for the grant since it is by default protected, only one MPU region is required for a process's RAM. In terms of allocating unused memory between the heap and the grant, we choose to allocate this to the kernel over the user, with the operating system goal in mind to not give a user needless access to it (even though it is empty).

Allocating Process Memory

The growing regions request is implemented in Tock specifically for a process's RAM. A process is also sometimes called an app, leading to this method's name: `allocate_app_memory_region`. It takes in the same five parameters as the method mentioned in Section 6.2.1, and two additional parameters: `initial_app_memory_size` and `initial_kernel_memory_size` that represent the initial memory required for the process accessible memory (PAM) and the grant, respectively.

The implementation of `allocate_app_memory_region` differs for MPUs, since different MPUs can use their own techniques to optimally subdivide this memory. Recall from Section 4.3.3 that previously, the Cortex-M implementation used subregions for changing the alignment and sizes of regions. In the `allocate_app_memory_region` implementation for the Cortex-M on the other hand we use one region for process memory and use its subregions in order to divide it between the PAM and the kernel. This way, only one MPU region has to be used, as the default memory map is protected. Furthermore, having the growth defined by subregions makes the growth of regions increment in logical steps of one eighth of the region.

For power-of-two aligned MPUs, it is an invariant that if memory regions are sorted by descending size and the first one aligns, they will always align to their start addresses. Since the size of processes of Tock is known at boot time, this property can be used to sort processes in descending order: this way, there will not be any external fragmentation given that the first region aligns. These statements are possible because of the fact that dynamic loading is not yet implemented in Tock (although possible by design), and the start of RAM in Tock aligns to a large power of two. Now that Cortex-M subregions are used for aligning permissions within the MPU region instead of for aligning *start* and *size*, its start and size constraints are similar to other power-of-two aligned MPUs; therefore, we also sort the process memory regions for the Cortex-M implementation in descending order.

For the implementation of the Kinetis K also only one region is used: one that covers the PAM and is rounded up to a granularity of 32. Again, the Kinetis K implementation is much simpler than that of the Cortex-M, with 35 lines of code instead of 80.

Updating Process Memory

To update the process RAM, a new method is implemented: `update_app_memory_region`. It is executed every time `brk()` and `alloc()` are called, and the check to see if a region has grown beyond its initially allocated memory is now done inside this method in the MPU implementation, contrary to in the process manager as was the case in the region-based interface. Hence, an MPU implementation can decide what to do when `update_app_memory_region` is called.

In our implementation for the Cortex-M MPU, we choose to simply recompute all regions at every call of either `brk()` or `alloc()` for two reasons:

1. In practice, `brk()` and `alloc()` are rarely ever called in Tock, and so the computational overhead caused by recomputing regions will be relatively low.
2. As the process manager only has information regarding the logical regions and cannot distinguish anything from physical regions, it would take an additional parameter to make the process manager aware of this. This is especially hard because of possible subregion usage, and adding such a parameter would lead to a needless increase in complexity and communication.

For the Kinetis K, we make the same decision in light of these two reasons in addition to a third one: since the Kinetis K has the fine granularity of 32 bytes, practically every call to `brk()` or `alloc()` will need to a recomputation of regions anyways.

Naturally, there still exists a point where a process grows too much at runtime and crashes. For the Cortex-M MPU implementation, as subregions are used to give permissions to the user level, this happens when the end of the last enabled subregion grows into the grant or vice versa. For the Kinetis K, this occurs only when the PAM and grant grow into each other with a precision of 32 bytes.

6.2.3 Access Permissions

We define a `Permissions enum` with five options as discussed in Section 6.1.3. It is the responsibility of the MPU implementation to convert these permissions into valid bit fields for register values. As all the chosen MPUs support these five options, there is no need to deal with an error case considering these permissions can always be set.

Because the default memory map now only has to be protected for the user level, it does not matter what happens to the background region for the supervisor level. For the Kinetis K implementation, this leads to an extra available region in comparison to the region-based interface, since it has to use one region to make the default memory map accessible for the supervisor mode as discussed in Section 5.2.4. As for the Cortex-M, because it has specific functionality for the default memory map anyways (Section 3.2.4), the background region is made accessible for the supervisor to allow support for a possible future use case.

6.2.4 Overview

An overview of the implementation of the process-based MPU interface in Tock is displayed in Listing 7. In the three methods `allocate_regions`, `allocate_app_memory_region` and `update_app_memory_region`, two additional parameters are passed besides the ones discussed.

The first is `permissions`, containing one of the five possible permissions as defined in the `Permission enum`. The second is the `config` struct in which regions are stored as proposed in Section 4.1. A key difference in comparison to the region-based interface shown in Section 5.2.5 is that the `config` struct is now passed in, instead of being returned. Where the region-based interface entirely recomputes the `config` every time a region changes, the process-based interface lets the MPU implementation maintain and update `config`, by passing it in as a mutable reference that all these three methods borrow (Listing 3) while executing. In doing so, the MPU implementation is also free to use more regions when necessary. Every time after one of these three methods is called, the hardware registers are written to by calling `configure_mpu` using the `config` struct. In addition, `configure_mpu` is called by the process manager at every context switch.

As in the region-based interface, all methods now have a default method definition as discussed in Section 4.4, allowing platforms that do not have an MPU to run on Tock. For `allocate_region` and *allocate_app_memory_region*, a region is returned that is equal to the requested start and size, and for `update_app_memory_region` a simple success value (`Ok`) is returned.

6.3 Evaluation

We proceed by evaluating the process-based interface. First, the interface is compared using our example of three concurrent processes, and a comparison is done with the existing and region-based interfaces. Next, the portability and performance are discussed, after which the evaluation is summarized.

6.3.1 Memory

For the implementation described in Section 6.2, we evaluate the performance in terms of efficiency in memory allocation for the same situation as the existing and region-based interface, with the three running processes `crc`, `ip_sense` and `ac`.

The allocation of flash is similar to the situation shown in the evaluation of the region-based interface in Section 5.3.1. The only two differences are the absence of the `min_offset` parameter, which for flash was set to zero anyways, and `lower_bound` (now `min_region_size`) is not wrapped in an `Option` anymore. Therefore, we now take a look at what has changed: the memory situation in RAM, and specifically the allocating and updating of its MPU regions.

Allocating Process Memory

First, processes are sorted by their size in RAM. Since `ip_sense` has a PAM of 7,060 bytes as shown in Table 3.4, it is placed first in memory, followed up by `crc` and `ac`. Booting proceeds by the process manager allocating memory with `unallocated_memory_start` set to `0x20004000`, `unallocated_memory_size` equal to the size until the end of RAM is reached and a `min_region_size` of $7,060+2,048 = 9,108$ bytes, according to the actual required memory and a margin for potential growth of these regions. Now that the Cortex-M implementation does not use subregions for alignment anymore, this size is simply rounded up to a power of two resulting in a region size of 16 kB. Also, there is no need to compute an MPU region for the grant region, since memory is by default protected. As for the enabled subregions, the algorithm calculates that using four

```rust
/// Access permissions.
pub enum Permissions {
    ReadWriteExecute, ReadWriteOnly, ReadExecuteOnly, ReadOnly, ExecuteOnly
}

/// MPU region.
pub struct Region {
    start_address: *const u8, size: usize
}

pub trait MPU {
    /// Enables the MPU.
    fn enable_mpu(&self) {}

    /// Disables the MPU.
    fn disable_mpu(&self) {}

    /// Returns the total number of regions supported by the MPU.
    fn number_total_regions(&self) -> usize {0}

    /// Allocates a new MPU region.
    /// Returns the start and size of the allocated MPU region.
    fn allocate_region(&self, unallocated_memory_start: *const u8,
    ↪ unallocated_memory_size: usize, min_region_size: usize, permissions:
    ↪ Permissions, config: &mut Self::MpuConfig,
    ) -> Option<Region>
    {Some((unallocated_memory_start, min_region_size))}

    /// Allocates MPU region(s) for a process's RAM.
    fn allocate_app_memory_region(&self, unallocated_memory_start: *const u8,
    ↪ unallocated_memory_size: usize, min_memory_size: usize,
    ↪ initial_app_memory_size: usize, initial_kernel_memory_size: usize,
    ↪ permissions: Permissions, config: &mut Self::MpuConfig,
    ) -> Option<(*const u8, usize)>
    {Some((unallocated_memory_start, min_region_size))}

    /// Updates MPU region(s) for a process's RAM.
    fn update_app_memory_region(&self, app_memory_break: *const u8,
    ↪ kernel_memory_break: *const u8, permissions: Permissions, config: &mut
    ↪ Self::MpuConfig,
    ) -> Result<(), ()> {Ok(())}

    /// Configures the MPU with the provided region configuration.
    fn configure_mpu(&self, config: &Self::MpuConfig) {}
}
```

Listing 7: Simplified overview of the process-based MPU interface.

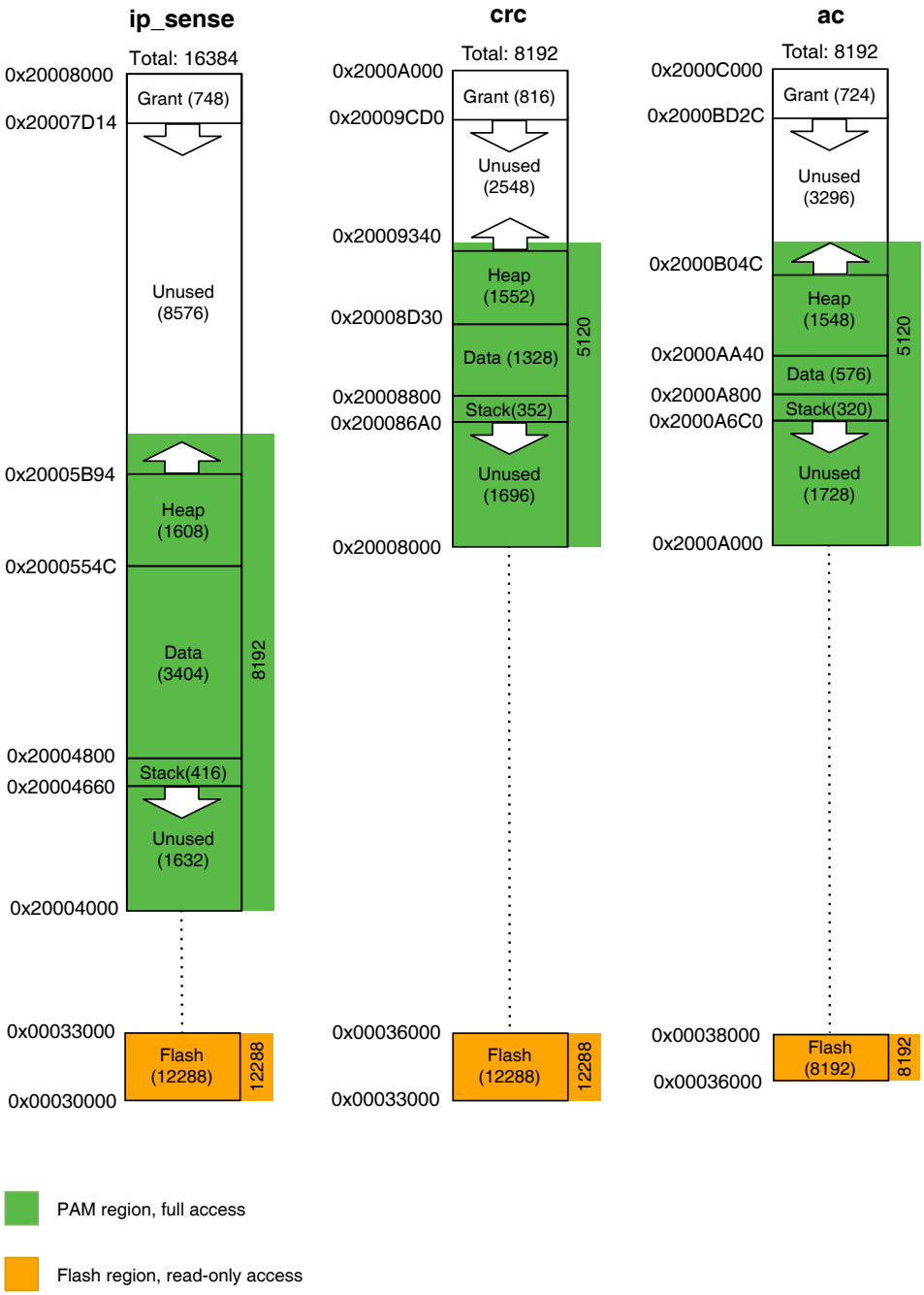

Figure 6.1: Overview of process memory using the process-based MPU interface for the Cortex-M MPU in Tock. Shown is the memory and MPU allocation of three memory-contiguous processes running concurrently using this interface. A more detailed explanation is given in Section 6.3.

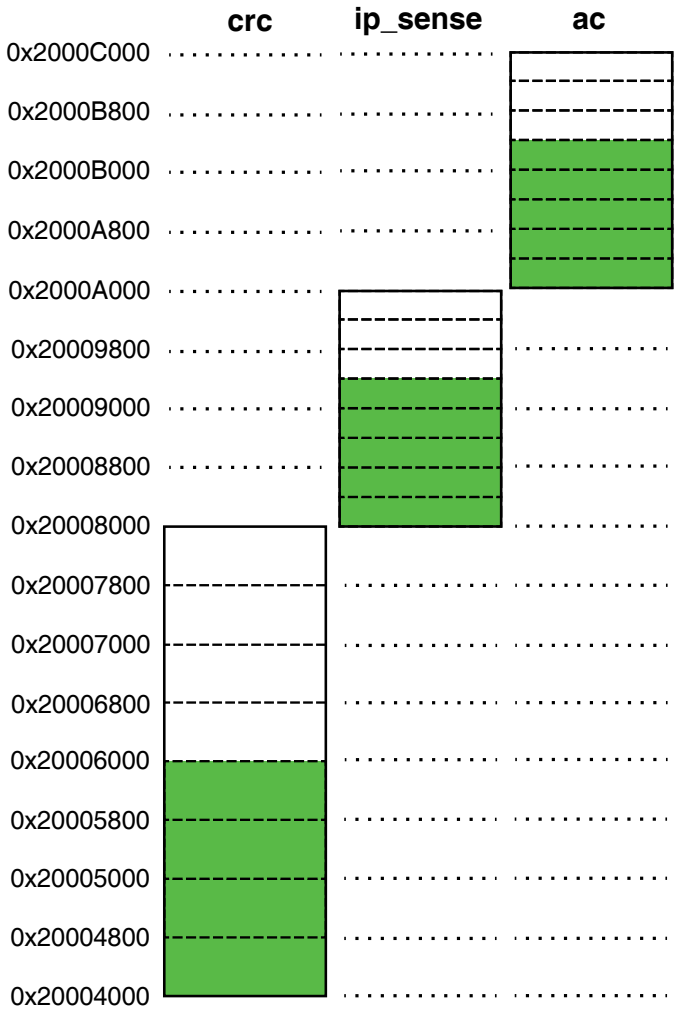

Figure 6.2: Overview of subregion usage for the PAM in the situation shown in Figure 6.1. Full lines represent regions, and dashed lines indicate subregions.

of its eight subregions is sufficient to cover the stack, data and heap, resulting in a logical region size of 8 kB ($16\text{ kB}/8 * 4 = 8\text{ kB}$).

When `crc` is context switched to, memory is allocated for it with a start address of `0x20008000`, a `min_region_size` of 4,928 + 2,048 = 6,976 bytes and similar other values, resulting in a region of 8,192 bytes. Now, the first five subregions are enabled to cover a range of 5,120 bytes greater than the required 4,928 bytes. On the context switch to `ac` a similar thing occurs: a region of 8 kB is allocated with 5 kB of subregions enabled.

Updating Process Memory

The PAM and grant can be updated at runtime by calling `update_app_memory_region`, that is called in both `brk()` and `alloc()`, and recomputes the region configuration for the active pro-

cess. With `ip_sense` for instance, the first 1,132 bytes of growth for the heap will not change the subregion configuration at all. With a growth of 1,133 bytes however, five out of eight subregions are enabled, and all memory until 10,240 bytes (10 kB) is covered.

As for growth of the grant, nothing in particular has to happen since the unused memory is by default allocated to it. What does happen on every call of `update_app_memory_region` is a check to see if the grant has grown into the PAM, in this case being the end of the last enabled subregion. If this happens to be so, memory has ran out for the process leading to its failure. Region growth of the other process is really similar to this scenario, where we simply grow and shrink subregions whenever necessary. This is opposed to the region-based interface as evaluated in Section 5.3, where we had to deal with complicated power-of-two region alignments.

Comparison

A comparison of the memory usage for the existing, region-based and process-based interface given the example of three running processes is displayed in Table 6.1. A brief summary of the three interfaces given this table is as follows:

Existing Interface As the existing interface uses the Cortex-M subregions only for aligning the start address, all regions are a power of two leading to an inefficient usage of flash memory. Regions are allowed to overlap leading to a somewhat higher possible (and unlikely) PAM growth for `ip_sense`, because this interface is specifically created for the Cortex-M. However, in all other aspects the existing interface performs worse in memory than the other two interfaces, even though it does not even work on any other MPU.

Region-Based Interface Because in the region-based interface processes are independently placed as efficiently as possible, this leads to a low total memory usage in both flash and RAM while being architecture agnostic. However, the region-based interface does suffer from external fragmentation, and most importantly, supports region growth very poorly (especially that of the grant).

Process-Based Interface The process-based offers the solution for being both architecture agnostic and having a relatively low memory overhead. No external fragmentation occurs, but some more internal space is reserved for the PAM and grant to grow into. In addition, because of the way Cortex-M subregions grow, region growth takes place in more fine-grained linear steps for both the PAM and grant region.

We would like to emphasize that these examples are given in order to clarify the subtle differences between the interfaces. The greatest contribution this work proposes is the architecture agnosticism of the interface, which is something both the region-based and process-based interface achieve.

Flash Memory Footprint

Since the additions made add to the number of lines of code in the Tock kernel, this leads to an increased memory footprint. By checking the size of the binaries, the difference in size between the existing and process-based interface is found. In the current Tock version, Tock 1.2, the total memory size in terms of code/text memory Tock occupies for the Hail development board[1] is

[1] `https://github.com/lab11/hail`

Feature	Existing	Region-Based	Process-Based
Total RAM Usage	32 kB	26 kB	32 kB
Total Flash Usage	40 kB	32 kB	32 kB
External Fragmentation	0 kB	0.625 kB	0 kB
Maximum PAM Size for `crc`	7 kB	7 kB	7 kB
Maximum PAM Size for `ip_sense`	15 kB	8 kB	14 kB
Maximum PAM Size for `ac`	7 kB	7 kB	7 kB
Maximum Grant Size for `crc`	2 kB	1.875 kB	3 kB
Maximum Grant Size for `ip_sense`	8 kB	1.75 kB	8 kB
Maximum Grant Size for `ac`	2 kB	1.75 kB	3 kB
Region Growth Steps	Exponential	Varies	Linear

Table 6.1: Comparison of memory usage for the existing, region-based and process-based interface given the example of three running processes. Both the maximum possible region sizes after growing and the total memory usage are shown.

83,620 bytes. With the inclusion of the new process manager, MPU interface, MPU implementations and other small changes, the flash size is increased to 84,468, corresponding to an increase of 848 bytes or 1.0%.

6.3.2 Portability

The process-based interface is able to support all region-based MPUs. Contrary to the region-based interface, it is able to support barrier-based MPUs like the nRF51 MPU as discussed in Section 3.3. Recall that barrier-based MPUs are only able to protect memory at one side of a barrier, and therefore Tock only supports the running of one process in order to guarantee processes are not able to read each others memory. Because of this, the sum of $start$ and $size$ can simply be used as an address value for the barrier: those of `allocate_region` for flash and `allocate_app_memory_region` for RAM.

Another aspect making this interface more portable is that it is easier to implement and clearer to read. The existing interface is Cortex-M specific and unclear in its details, and the region-based interface has additional complexity for supporting absolute region requests. Updating RAM with one method as in the process-based interface makes the interface simpler to grasp.

6.3.3 Performance

Each process has its own MPU region configuration. As such, the MPU has to be reconfigured at every context switch, and the existing design does this by recomputing MPU regions every time. In Section 4.1 the storage of physical regions is proposed, aiming to reduce needless computational overhead in case a process's region configuration has not changed from its previous execution slot. For the process-based interface, we have implemented this storage of regions and now proceed by evaluating its impact.

In order to measure the context switching overhead in Tock, we have configured a GPIO pin to produce a high voltage after taking back control from a process, and a low voltage right before

starting the next. We have used a logic analyzer[2] to measure the time difference between this toggling. This test was performed on the Hail development board on Tock 1.2, with both the existing and the new MPU interface in which regions are stored. The results of this test are an average context switching overhead of 54.4 μs for the existing interface and 43.4 μs for the new interfaces, resulting in a speedup of 25.34%. Note that the performance results are equal for the region-based and process-based interface, as their differing aspects do not involve the performance.

6.3.4 Summary

The process-based interface presents different methods for creating non-growing and growing regions. Because of this differentiation, regions that do not grow at runtime such as flash can be allocated as efficiently as in the region-based interface, and memory that does require growth such as process RAM can now also be efficiently allocated. Letting the MPU implementation decide on region allocation makes the use of multiple regions and MPU-specific optimizations such as the Cortex-M subregions possible. In addition, having the MPU implementation be aware of how memory is allocated (where the next process RAM starts) allows it to optimally allocate the grant region without any external fragmentation.

When using the region-based interface, the two main problems are allocating regions efficiently without having external fragmentation, and region growth for power-of-two aligned MPUs. With the process-based interface we have alleviated these problems, and arrived at a generic interface which supports growing regions and has minimal external fragmentation.

[2]Saleae Logic 8, https://www.saleae.com/

Chapter 7

Conclusions and Future Work

A new generation of memory isolation mechanisms naturally draws the attention of many due to a large number of bugs related to memory management in resource-constrained systems. Tock brings flexible multiprogramming to this tier of computing, leading to new opportunities and capabilities for low-power embedded systems, while providing memory safety using the novel mechanisms that Rust and MPUs provide. Although Tock is designed to be deployed on a variety of hardware platforms, its implementation currently only supports platforms containing the Cortex-M processor.

In this thesis, we investigate the feasibility of supporting a wide range of MPUs in Tock efficiently and without losing utility by answering the following research question:

Can an Architecture-Agnostic MPU interface be created in Tock?

We now return to this main research question and the formulated sub-questions, discussing how these questions are answered. Moreover, we identify remaining problems and improvements in order to provide future research directions.

7.1 Contribution & Results

In this research, the objective has been to explore the ability to make platforms memory-safe for user processes by utilizing the memory safety features of Rust in combination with the hardware isolation provided by MPUs through using Tock. Although several approaches have been suggested to provide memory safety on more than one differing platform, previous work has been unable to provide memory safety in an efficient way due to additions in runtime overhead, complex language modifications and the lack of dynamic allocation.

In Section 1.2, we identify three sub-questions of the main research question. To answer the first sub-question, a study of MPUs in the embedded system landscape is performed by selecting a variety of MPUs and comparing their key differences in terms of alignment. In addition, the number of available regions, possible access permissions, semantics in overlapping regions and default access permissions are analyzed for each of these MPUs. Furthermore, an analysis of the memory allocation model in Tock discerns the main problems of its current state, and in particular its entanglement with the Cortex-M MPU. Based on these analyses, the conclusion

is drawn that the main research challenge of designing a generic MPU interface for Tock lies within region allocation for two different types of MPUs: power-of-two aligned MPUs and block-aligned MPUs.

The second sub-question describes the problem of creating an interface that does not assume any knowledge regarding what kind of application the MPU will be used for. In order to do this, several design considerations are made; in particular algorithms for flexible and efficient region alignment are designed for all selected MPUs. The design of a region-based interface is proposed, which is based on the premise of offering true platform independency. It assumes no information regarding the application of MPU regions, leading to a straightforward interface that moves most complexity to the client. The region-based interface uses a relative and absolute region request in order to separate initially allocating regions and growing regions at runtime. Since the region-based interface is generic and independent of the target application, it positively answers the second sub-question.

The third and final sub-question investigates the ability of a generic MPU interface to work efficiently, not losing any MPU-specific optimizations. Unfortunately, as the region-based interface runs into problems due to the large variation in MPU designs, it fails to answer this research question. In particular, the two main problems are the inability to consistently support region growth and external fragmentation in power-of-two aligned MPUs.

In order to find a positive answer to the third sub-question, the process-based interface is proposed. This interface contains functionality geared towards both non-growing and growing regions. Non-growing region requests are used to allocate regions that do not grow during runtime such as flash, whereas growing region requests are used for situations in which two regions grow towards each other. In particular, a growing region request gives an MPU implementation the freedom to choose proper regions according to memory requirements, contrary to forcing the MPU implementation to allocate regions independently as was the case in the region-based interface. This enables use of optimizations such as Cortex-M subregions, resulting in minimal external fragmentation in addition to support for growing regions. In terms of performance, the process-based interface delivers a speedup of 25.34% at the cost of 848 bytes or an 1.0% overhead in flash. Most importantly, the process-based interface provides a level of memory efficiency and optimality required from individual MPUs, and thereby allows Tock to become architecture-agnostic in an effective manner.

7.2 Future Work

Power-of-two aligned MPUs are more prominent than block-aligned MPUs because they are less expensive in terms of cost and performance. In terms of hardware efficiency, this is good. However, the power-of-two alignment constraint significantly complicates the process of creating an MPU interface. Nearly all the problems discussed in this research are caused by MPUs having a power-of-two constraint, and because of this the region-based interface is unsuitable and a process-based interface is required. Would the MPU landscape ever converge to a block-aligned MPUs, the interface should change accordingly.

Although an MPU implementation for more hardware platforms would be a great way to test the value of the proposed MPU interfaces, unfortunately this would require Tock to support more hardware platforms first. Since a port for the MK66 already existed with the only main thing missing being an MPU implementation, creating and testing these implementations us-

ing the new MPU interface was a realizable feat. However, porting an entire operating system like Tock to other hardware platforms involves much more factors than just the MPU. Seeing as how the process-based interface we propose is relatively straightforward and well documented, and so are the implementations for the Cortex-M and Kinetis K MPU, we believe supporting other platforms safely should now be a much easier task. The implementation of block-aligned MPUs is a straightforward task. As for the implementation of other power-of-two MPUs than the Cortex-M (that do not support subregions), we propose using at least more than one MPU region within a process's random access memory (RAM), in order to ensure a more fine-grained alignment within the process RAM.

The deletion of regions is not yet implemented in the process-based MPU interface. As the flash and RAM regions for a process do not require deletion during runtime, and Tock currently reallocates all memory on the addition or deletion of a process, such a feature is simply not useful in Tock at present. Even inter-process communication (IPC) is in a very experimental stage in Tock 1.2, such that dynamic reallocation of IPC is not even supported in the kernel. As these features evolve however, the MPU interface will require functionality for deletion of regions.

Appendix A

Register Interface

An update is made to simplify writing to hardware registers by updating the register interface for the Cortex-M MPU implementation. Instead of writing to memory by completely exposing registers, they are marked as `ReadOnly`, `WriteOnly` or `ReadWrite`, matching the way the hardware exposes them. Furthermore, with this updated register interface text is translated into bitfields. For instance, enabling the MPU in the old interface was done as follows:

```
regs.control.set(0b101);
```

A programmer that wants to create a driver in Tock requires knowledge of the exact registers in order to make sure that this is correct, making this a complex and bug-prone line of code. Therefore, we now do this with an updated register interface:

```
regs.control.write(Control::ENABLE::CLEAR);
```

A register interface like this is more expressive and prevents mistakes when implemented. For instance, if a programmer of the kernel accidentally writes to the `RegionBaseAddress` register instead of the *Control* register, the code would no longer compile since it does not contain an `ENABLE` field. As the register interface is created by using the `unsafe` keyword, there must be programmed with care.

In order to even further simplify this process, we define a dedicated function that handles the conversion of variables to register values, thereby taking complexity out of the function that allocate regions. This entails transforming variables such as region start and size. For instance, in the existing implementation, setting up the bit string to write to the registers is done as follows:

```
Some(unsafe
    Region::new(
        (start | 1 << 4 | (region_num & 0xf)) as u32,
        1 | (region_size.exp::<u32>() - 1) << 1 | ap << 24 | xn << 28,
    )
})
```

Whereas the new implementation does this in a more clear manner:

```
CortexMRegion::new(start, size, region_start, region_size, region_num,
↪  subregion_mask, permissions)
```

Note that since we now want to return both logical and physical regions as discussed in Section 4.1, we need to send the start, size and permissions of the covered memory region to the interface in addition to the Cortex-M specific underlying region start, underlying region size, and subregion mask (the bitfield that indicates which subregions are enabled). Furthermore, notice that because of this new register interface, there is no need to use the `unsafe` keyword anymore: all use of it is moved to the kernel.

Bibliography

[1] W. Obile, "Ericsson mobility report: on the pulse of the networked society," *Ericsson mobility report*, June 2015.

[2] 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," in *Ambient intelligence*. Springer, 2005, pp. 115–148.

[3] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, "Engineering the servo web browser engine using rust," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 81–89.

[4] S. M. Alnaeli, M. Sarnowski, M. S. Aman, A. Abdelgawad, and K. Yelamarthi, "Vulnerable c/c++ code usage in iot software systems," in *Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum on*. IEEE, 2016, pp. 348–352.

[5] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno *et al.*, "Comprehensive experimental analyses of automotive attack surfaces." in *USENIX Security Symposium*. San Francisco, 2011, pp. 77–92.

[6] M. Khera, "Think like a hacker: Insights on the latest attack vectors (and security controls) for medical device applications," *Journal of diabetes science and technology*, vol. 11, no. 2, pp. 207–212, 2017.

[7] R. Morris and K. Thompson, "Password security: A case history," *Communications of the ACM*, vol. 22, no. 11, pp. 594–597, 1979.

[8] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, "A large-scale analysis of the security of embedded firmwares," in *USENIX Security Symposium*, 2014, pp. 95–110.

[9] N. D. Matsakis and F. S. Klock II, "The rust language," in *ACM SIGAda Ada Letters*, vol. 34, no. 3. ACM, 2014, pp. 103–104.

[10] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM, 2011, p. 5.

[11] 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*. ACM, 2017, pp. 234–251.

[12] 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*. ACM, 2017, p. 1.

[13] ARM, "Cortex M4 Generic User Guide," *ARM DUI 0553A (ID121610)*, pp. MPU: 4–37 to 4–47, 2010. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439b/DDI0439B_cortex_m4_r0p0_trm.pdf

[14] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[15] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual Volume I: User-Level ISA, Document Version 2.2," RISC-V Foundation, Tech. Rep., May 2017. [Online]. Available: https://riscv.org/specifications/

[16] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014.

[17] I. C. Bertolotti and T. Hu, *Embedded Software Development: The Open-Source Approach*. CRC Press, 2016.

[18] C. Metz, "The Epic Story of Dropbox's Exodus From the Amazon Cloud Empire," Jun 2017. [Online]. Available: https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/

[19] R. R. Schaller, "Moore's law: past, present and future," *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[20] J. A. Tov and R. Pucella, "Practical affine types," in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 447–458.

[21] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, 2018.

[22] Stack Overflow, "Developer survey results: 2018," *Available at: https://insights.stackoverflow.com/survey/2018/ (accessed 10 September 2016).*, 2018.

[23] "Tock: An embedded operating system designed for running multiple concurrent, mutually distrustful applications on low-memory and low-power microcontrollers." [Online]. Available: https://www.tockos.org/

[24] E. Brown, "Embedded linux keeps growing amid iot disruption, says study," Mar 2015. [Online]. Available: https://www.linux.com/news/embedded-linux-keeps-growing-amid-iot-disruption-says-study

[25] C. Sabri, L. Kriaa, and S. L. Azzouz, "Comparison of IoT constrained devices operating systems: A Survey," in *Computer Systems and Applications (AICCSA), 2017 IEEE/ACS 14th International Conference on*. IEEE, 2017, pp. 369–375.

[26] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," *Acm Sigplan Notices*, vol. 49, no. 4, pp. 41–51, 2014.

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

[28] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. C. Schmidt, "Riot os: Towards an os for the internet of things," in *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*. IEEE, 2013, pp. 79–80.

[29] R. Barry, *The FreeRTOS reference manual*. Amazon Web Services, 2009.

[30] C.-C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava, "Sos: A dynamic operating system for sensor networks," in *Proceedings of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys)*. Citeseer, 2005, pp. 1–2.

[31] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms," *Mobile Networks and Applications*, vol. 10, no. 4, pp. 563–579, 2005.

[32] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," in *ACM SIGPLAN Notices*, vol. 37, no. 1. ACM, 2002, pp. 128–139.

[33] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: software-based memory protection for sensor nodes," in *Proceedings of the 6th international conference on Information processing in sensor networks*. ACM, 2007, pp. 340–349.

[34] L. Gu and J. A. Stankovic, "t-kernel: Providing reliable os support to wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 2006, pp. 1–14.

[35] F. Sant'Anna, N. Rodriguez, R. Ierusalimschy, O. Landsiedel, and P. Tsigas, "Safe system-level concurrency on resource-constrained nodes," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2013, p. 11.

[36] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in cyclone," *ACM Sigplan Notices*, vol. 37, no. 5, pp. 282–293, 2002.

[37] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "Smart: Secure and minimal architecture for (establishing dynamic) root of trust." in *NDSS*, vol. 12, 2012, pp. 1–15.

[38] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base." in *USENIX Security Symposium*, 2013, pp. 479–494.

[39] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A low-cost security architecture for iot devices," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 3, p. 7, 2017.

[40] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 10.

[41] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "Tytan: tiny trust anchor for tiny devices," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 34.

[42] J. Hester, T. Peters, T. Yun, R. Peterson, J. Skinner, B. Golla, K. Storer, S. Hearndon, K. Freeman, S. Lord *et al.*, "Amulet: An energy-efficient, multi-application wearable platform," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*. ACM, 2016, pp. 216–229.

[43] Texas Instruments, "Msp430fr5969 16 mhz ultra-low-power microcontroller," *SLAS704G*, 2016.

[44] T. Hardin, R. Scott, P. Proctor, J. Hester, J. Sorber, and D. Kotz, "Application Memory Isolation on Ultra-Low-Power MCUs," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018, pp. 127–132.

[45] J. Yiu, *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*. Newnes, 2013.

[46] ARM, "Embedded segment market update," *China Technical Seminar Series*, 2015. [Online]. Available: https://www.arm.com/zh/files/event/1_2015_ARM_Embedded_Seminar_Richard_York.pdf

[47] ——, "Cortex m0+ generic user guide," *ARM DUI 0662B (ID011713)*, 2012. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0662b/DUI0662B_cortex_m0p_r0p1_dgug.pdf

[48] ——, "Cortex m3 generic user guide," *ARM DUI 0552A (ID121610)*, 2010. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0552a/DUI0552A_cortex_m3_dgug.pdf

[49] Nordic Semiconductor, "nRF52 Series API Reference," *Nordic Semiconductor*, 2015. [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v14.0.0%2Fgroup__nrf__mpu.html

[50] Texas Instruments, "Tiva C Series TM4C129x," *SPMU363*, 2014. [Online]. Available: http://www.ti.com/lit/ug/spmu363a/spmu363a.pdf

[51] ——, "CC13x2, CC26x2 SimpleLink Wireless MCU," *SWCU185A*, 2018. [Online]. Available: http://www.ti.com/lit/ug/swcu185a/swcu185a.pdf

[52] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Version 1.10," RISC-V Foundation, Tech. Rep., May 2017. [Online]. Available: https://riscv.org/specifications/privileged-isa/

[53] NXP, "Kinetis K Series Microcontrollers (MCUs) Selector Guide," *A Performance and Integration Series Based on 32-bit ARM®Cortex®-M4 Cores*, 2018. [Online]. Available: https://www.nxp.com/docs/en/product-selector-guide/KINETISKMCUSELGD.pdf

[54] ARM, "Cortex m0 generic user guide," *ARM DUI 0497A (ID112109)*, 209. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0497a/DUI0497A_cortex_m0_r0p0_generic_ug.pdf

[55] Nordic Semiconductor, "nRF51 Series Reference Manual," *Nordic Semiconductor*, pp. 28–36, 2014. [Online]. Available: http://infocenter.nordicsemi.com/pdf/nRF51_RM_v3.0.pdf

[56] Altera, "Nios ii classic processor reference guide," *Altera*, pp. 3–2 to 3–54, 2016. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf

[57] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE micro*, vol. 20, no. 2, pp. 60–70, 2000.

[58] Cadence, "Xtensa LX7 Microprocessor," *RG-2017.5*, pp. 33–39, 2017.

[59] Texas Instruments, "White paper: enhancing the KeyStone II architecture with multicore RISC processing," *SPRY223*, 2013. [Online]. Available: https://www.multivu.com/assets/54044/documents/54044-ARM-A15-in-KeyStone-II-White-Paper-original.pdf

[60] ——, "66AK2Hxx Multicore DSP+ARM®KeyStone II System-on-Chip (SoC)," *SPRS866G*, p. 79 to 92, 2017. [Online]. Available: http://www.ti.com/lit/ds/symlink/66ak2h12.pdf

[61] NXP Semiconductors, "Kinetis K66 Sub-Family Reference Manual," *NXP Semiconductors*, pp. 449–472, 2017. [Online]. Available: https://www.nxp.com/docs/en/data-sheet/K66P144M180SF5V2.pdf

[62] Texas Instruments, "KeyStone Architecture Memory Protection Unit (MPU)," *SPRUGW5A*, pp. 3–1 to 3–4, 2013. [Online]. Available: http://www.ti.com/lit/ug/sprugw5a/sprugw5a.pdf

[63] P. Bonzini, J. Hauser, and A. Waterman, "Risc-v hypervisor exstension," *7th RISC-V Workshop*, 2017. [Online]. Available: https://content.riscv.org/wp-content/uploads/2017/12/Tue0942-riscv-hypervisor-waterman.pdf

[64] Atmel, "The sam4l series: an arm-based flash mcu," *42023H*, 2016.