DEGREE PROJECT

Real time Rust on multi-core microcontrollers

Jorge Aparicio Rivera

Computer Science and Engineering, master's level (120 credits)
2020

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

LULEÅ UNIVERSITY OF TECHNOLOGY

Abstract

Today the majority of embedded software is written in C or C++ using the thread paradigm. C and C++ are memory unsafe programming languages that often appear in CVE (Common Vulnerability and Exploits) reports. Threads are a popular concurrency paradigm in SMP (Symmetric Multi Processor) systems; however, threads can deadlock and are hard to statically analyze for schedulability. At the same time, security is becoming more and more important thanks to the exponential grow of IoT (Internet of Things) devices; meanwhile, vendors are starting to ship more and more heterogeneous multi-core devices where the thread paradigm can not be applied. In this thesis, we present an alternative programming framework for building real time, safety critical and general purpose embedded software that is memory safe by construction and suitable for single-core, homogeneous multi-core and heterogeneous multi-core systems.

Acknowledgments

First of all, I want to thank Luleå University of Technology for funding my masters through a stipend.

I would also like to thank professor Per Lindgren for his guidance and the many interesting discussions we had during the development of this degree project; and also Grepit AB for lending me hardware to test out the multi-core implementation of Real Time For the Masses.

My gratitude also goes to the many people that tried out Real Time For the Masses, reported bugs and gave feedback on the API and user documentation of the framework. The framework would not look as polished as it does today without their input.

Finally, I wish to thank my parents and brother for their unconditional support during my whole studies.

Contents

1 Introduction **6**
- 1.1 Trends in embedded software . . . 6
- 1.2 Real Time For the Masses . . . 6
- 1.3 The Rust programming language . . . 7
- 1.4 Static analysis . . . 7
- 1.5 Contributions of this thesis . . . 7
- 1.6 Outline . . . 8

2 Theoretical background **9**
- 2.1 The tasks and resources model . . . 9
- 2.2 Stack Resource Policy . . . 10
- 2.3 Multi-core environment . . . 11

3 Rust **13**
- 3.1 Memory safety . . . 13
  - 3.1.1 Ownership . . . 13
  - 3.1.2 References . . . 14
  - 3.1.3 Lifetimes . . . 14
    - 3.1.3.1 `'static` . . . 15
  - 3.1.4 Panicking . . . 15
  - 3.1.5 `unsafe` . . . 15
- 3.2 Concurrency . . . 16
  - 3.2.1 `Sync` . . . 16
  - 3.2.2 `Send` . . . 17
- 3.3 Other features . . . 18
  - 3.3.1 Generics . . . 18
    - 3.3.1.1 Const generics . . . 18
  - 3.3.2 Polymorphism . . . 18
    - 3.3.2.1 Enumerations . . . 18
    - 3.3.2.2 Trait objects . . . 19
  - 3.3.3 Procedural macros . . . 20
  - 3.3.4 Compilation targets . . . 20
  - 3.3.5 Conditional compilation . . . 21
- 3.4 `core` library . . . 21
  - 3.4.1 Raw pointer methods . . . 21
  - 3.4.2 `MaybeUninit` . . . 21
  - 3.4.3 Atomic . . . 22

4 Single-core RTFM **25**
- 4.1 Overview example . . . 25
- 4.2 Design decisions . . . 27
- 4.3 The API . . . 27

4.3.1 Resources . . . 27
4.3.2 Tasks . . . 27
4.3.3 `#[init]` . . . 28
4.3.4 `#[idle]` . . . 28
4.3.5 `Resources` . . . 28
4.3.6 `Spawn` . . . 29
4.3.7 `Schedule` . . . 29
4.3.8 `Monotonic` . . . 29
4.3.9 Device bindings . . . 30
4.4 Memory safety analysis . . . 30
4.4.1 Uninitialized memory . . . 30
4.4.2 Aliasing . . . 31
4.4.2.1 Serial access . . . 31
4.4.2.2 Preemption . . . 32
4.4.2.3 Nesting `locks` . . . 32
4.4.2.4 Duplicating resources . . . 33
4.4.3 Pointer invalidation . . . 34
4.4.4 `Send` bound . . . 34
4.4.5 `device` . . . 35
4.5 Implementation . . . 36
4.5.1 Base binary interface . . . 36
4.5.2 The ARM Cortex-M ISA . . . 38
4.5.2.1 Thread mode . . . 38
4.5.2.2 Interrupts . . . 38
4.5.2.3 NVIC . . . 38
4.5.2.4 BASEPRI . . . 39
4.5.2.5 PRIMASK . . . 39
4.5.2.6 ARMv6-M . . . 40
4.5.3 Hardware tasks . . . 40
4.5.4 Scope control . . . 41
4.5.5 Resource initialization . . . 42
4.5.6 `lock` . . . 43
4.5.6.1 Resource proxies . . . 44
4.5.6.2 BASEPRI invariant . . . 44
4.5.7 Message passing . . . 45
4.5.7.1 SPSC queue . . . 45
4.5.7.2 `spawn` . . . 46
4.5.7.3 Task dispatcher . . . 50
4.5.7.4 Priority ceiling analysis . . . 51
4.5.7.5 Queue capacity . . . 51
4.5.8 Timer queue . . . 51
4.5.8.1 Binary heap . . . 51
4.5.8.2 The system timer . . . 52
4.5.8.3 `TimerQueue` . . . 52
4.5.8.4 `schedule` . . . 52
4.5.8.5 Handler . . . 56
4.5.8.6 Task dispatcher . . . 57
4.5.8.7 Priority ceiling analysis . . . 57
4.6 WCET . . . 59
4.6.1 `lock` . . . 59
4.6.1.1 vs `Spinlock` . . . 59
4.6.2 `spawn` . . . 62
4.6.3 Task dispatcher . . . 62
4.6.3.1 `match` . . . 62

4.6.3.2 Overhead . . . 64
4.6.4 `schedule` . . . 65
4.6.5 `SysTick` . . . 65
4.7 Alternative implementations . . . 66
4.7.1 Multi-producer multi-consumer (MPMC) queue . . . 66
4.7.2 Lock-free memory pool . . . 66
4.8 Users . . . 66

5 Multi-core extension 69
5.1 Overview example . . . 69
5.2 API . . . 69
5.2.1 `cores` . . . 70
5.2.2 `core` . . . 70
5.2.3 `late` . . . 70
5.2.4 `#[shared]` . . . 71
5.3 Implementation . . . 72
5.3.1 Base binary interface . . . 72
5.3.2 `lock` . . . 72
5.3.3 `xpend` . . . 72
5.3.4 Message passing . . . 72
5.3.4.1 `spawn` . . . 73
5.3.4.2 Task dispatcher . . . 73
5.3.4.3 SPSC . . . 74
5.3.5 Timer queue . . . 74
5.3.6 Synchronization barriers . . . 74
5.3.6.1 `Barrier` . . . 74
5.3.6.2 `spawn` . . . 74
5.3.6.3 Cross-core resource initialization . . . 75
5.3.6.4 Time zero . . . 76
5.3.7 Code and data placement . . . 76
5.3.8 Example memory layout . . . 77
5.4 WCET . . . 82
5.4.1 Blocking exchange . . . 82
5.4.2 `xpend` . . . 83
5.4.2.1 vs `pend` . . . 84
5.4.3 Message passing . . . 84
5.4.3.1 `spawn` . . . 84
5.4.3.2 Task dispatcher . . . 85
5.4.3.3 End to end . . . 86
5.4.3.4 Contention effect . . . 87
5.4.4 Timer queue . . . 87
5.5 Code sharing . . . 87

6 Heterogeneous devices 89
6.1 μAMP . . . 89
6.1.1 `#[shared]` . . . 90
6.2 `heterogeneous` . . . 90
6.3 Additional restrictions . . . 91
6.4 No code sharing . . . 91

7 Stack usage analysis 93
7.1 Stack overflows . . . 93
7.2 Stack overflow protection . . . 94
7.2.1 Memory Protection Unit (MPU) . . . 94

7.2.2 MSPLIM . . . 95
7.2.3 Swapped memory layout . . . 95
7.3 `cargo-call-stack` . . . 95
7.3.1 Capabilities . . . 95
7.3.1.1 Filtering . . . 95
7.3.1.2 Cycles . . . 96
7.3.1.3 Function pointers . . . 96
7.3.1.4 Trait objects . . . 97
7.3.2 Implementation . . . 101
7.3.2.1 Per function stack usage . . . 101
7.3.2.2 Call graph . . . 101
7.3.2.3 Graph traversal . . . 104
7.3.3 `core::fmt` . . . 105
7.3.3.1 `ufmt` . . . 107

8 Conclusions and future work **109**
8.1 Future work . . . 109
8.1.1 Shared-exclusive locks . . . 109
8.1.2 Canceling and rescheduling tasks . . . 110
8.1.3 Automatic capacity selection . . . 111
8.1.4 Schedulability and security . . . 112
8.1.5 Stack analysis of RTFM applications . . . 112
8.1.6 A RTFM organization . . . 112

References **113**

Chapter 1

Introduction

1.1 Trends in embedded software

Ericsson forecasts 18 billions of IoT devices by 2022: a threefold increase with respect to the almost 6 billions deployed by 2016 [1]. Some of these new devices will be deployed in industrial settings as part of control systems that need to meet hard real time requirements for their correct operation. Others will be deployed in safety critical environments like trains and airplanes where software needs to be certified to strict standards. Yet others will be deployed in rural settings and remote places where energy efficiency is of particular importance.

In response to the increasing expectations on functionality provided by embedded software, vendors are starting to ship an increasing number of multi-core microcontrollers and most of them opt for heterogeneous multi-core devices where a fast core is paired with a slower one. This setup is meant to minimize power consumption by having the slow core deal with most non-CPU intensive tasks, mainly dealing with I/O events, and the fast core with the CPU intensive tasks.

As all IoT devices are connected to the internet, security ought to be at the front and center during software development yet most IoT devices are programmed using the C and C++ programming languages. These programming languages are chosen due to the constraints on memory and processing power that embedded devices have and the need for low level control over the hardware. C/C++ is not particularly known for producing secure software as evidenced by the large number of vulnerabilities reported against software written in them on a yearly basis. The main cause of security vulnerabilities are memory safety bugs like data races and buffer overflows.

Many of the C/C++ multitasking frameworks and embedded operating systems are based on the time-sliced thread paradigm. Although well suited for composing long running processes, this paradigm may not be the best option for building reactive embedded applications that need to deal with external events with tight timing constraints. To meet timing constraints in some cases the thread abstraction is avoided in favor of direct use of interrupt handlers but OSes usually provide little or no support to safely share data between threads and interrupt handlers increasing the chance of memory safety bugs.

1.2 Real Time For the Masses

The original Real Time For the Masses (RTFM) language (rtfm-lang) [2] is a Domain Specific Language (DSL) layered on top of the C programming language designed with reactive real time systems in mind [3] [4]. Instead of time-sliced threads this DSL is based around tasks that either start in response to a hardware event (hardware tasks) or are scheduled from software (software tasks). Its task model is inherently reactive and more closely fits the kind of software that usually runs on embedded systems.

In this model, tasks have run to completion semantics and can be assigned priorities. There is no context switching between tasks running at the same priority like in time-sliced threaded systems, but high priority tasks can preempt lower priority ones.

Under this model tasks can interact using a shared memory abstraction known as *resource* or communicate via message passing. Access to shared resources is synchronized using critical sections created by temporarily raising the dynamic priority of a task, which prevents other tasks that contend for the resource from starting.

Software tasks can be directly spawned onto the scheduler or scheduled to run at some point in the future [5]; in either case a message (data) can be passed along to become the input of the task.

All these abstractions can be efficiently implemented on architectures that support interrupt nesting like the ARMv7-M and ARMv7-R architectures resulting in abstractions with predictable, constant-time overhead.

However, being based on C the DSL is prone to memory safety bugs and requires user to carefully use the `lock` API where required to access resources in a data race free manner.

1.3 The Rust programming language

Rust is a modern programming language that targets the same wide application space as C and C++ but promises to be free of memory safety bugs by design. The borrow checker that is run as part of the compilation process catches bugs like use after free, pointer invalidation and data races.

The Rust standard library ships with a thread API and multiple synchronization primitives that are compile time checked to be memory safe but the core of the language provides functionality to build custom concurrency models and synchronization primitives with similar compile time guarantees.

Furthermore, the language provides metaprogramming features that can be used to extend the Rust syntax and create DSLs without the need to build a separate parser or compiler, as was the case with rtfm-lang. Also, as exemplified in [6], Rust's linear type system permits the efficient implementation of techniques that improve the security and reliability of software.

Rust is one of the few alternatives to C and C++ in the embedded software development space that can equally compete in terms of performance, low level control and platform support.

1.4 Static analysis

Static analysis and in particular Worst Case Execution Time (WCET) and stack usage analyses are required components of the certification process of safety critical software. Due to their age and popularity plenty of C and C++ tooling exists for static analysis: from *sanitizers* that find data races and memory bugs in software to static stack usage analysis tools that find the worst case stack usage of an application. Only a handful of these tools, namely sanitizers, can be used on Rust programs.

1.5 Contributions of this thesis

This thesis set out to create better alternatives to the existing C and C++ solutions for writing embedded software. The author believes that security cannot be achieved without memory safety so it is instrumental that memory safety is a property of the framework and not a concern of the user; that is why Rust was chosen as the implementation language.

As part of the work in this thesis the Real Time For the Masses framework was ported to the Rust programming language and the task model was extended to support multi-core devices. Furthermore, a stack analysis tool capable of fully analyzing Rust programs was developed: `cargo-call-stack`.

To the author's knowledge this RTFM port is the first Rust framework that targets heterogeneous multi-core microcontrollers and `cargo-call-stack` is the first whole program static stack usage analysis tool that targets Rust programs.

1.6 Outline

The first chapter of the thesis corresponds to this introduction. The second chapter covers the theory behind RTFM's task model. The third chapter provides an overview of the main tenets of the Rust programming language and the features that play a crucial role in the implementation of the Rust port of Real Time For the Masses. The fourth chapter describes the single-core RTFM Rust API and its implementation, analyzes its promise of memory safety and its real time suitability: Worst-Case Execution Time (WCET) analysis is done on the entire API. The fifth chapter describes the API extension that provides multi-core support and its implementation on a homogeneous multi-core device; it also zooms in on the parts of the implementation relevant to the memory safety and analyzes the WCET of the multi-core implementation. The sixth chapter covers an alternative backend for the multi-core API that targets heterogeneous multi-core devices. The seventh chapter covers the features and the implementation of `cargo-call-stack`. The last chapter summarizes this document with conclusions and outlines potential improvements to be done as future work.

Chapter 2

Theoretical background

RTFM uses the tasks and resources model to logically structure applications and the Stack Resource Policy (SRP) to schedule the tasks and synchronize concurrent access to resources [3]. This section discusses both the theory behind the task model and SRP; it also explains how RTFM approaches multi-core systems.

2.1 The tasks and resources model

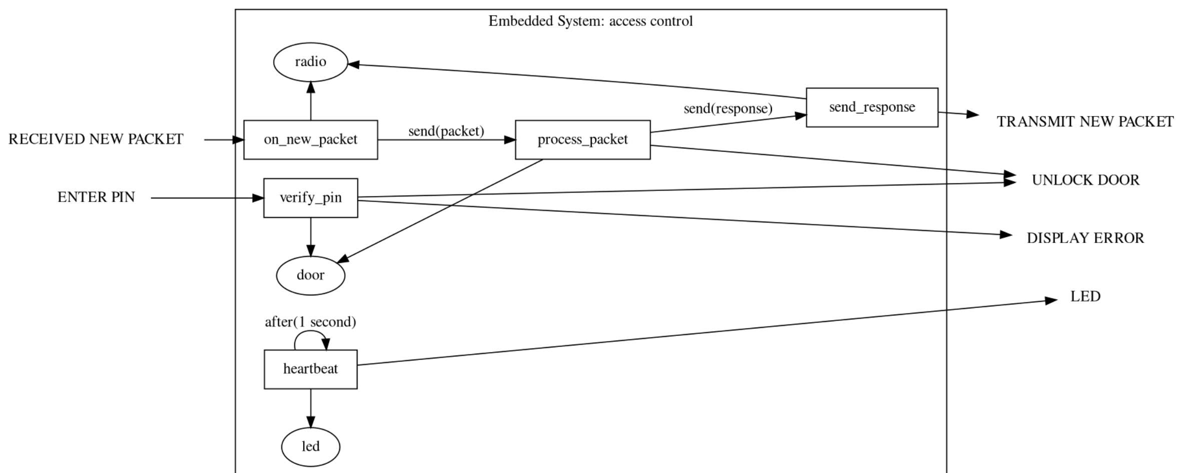


Figure 2.1: Tasks and resources model

One can think of an embedded system as a black box with inputs and outputs. The inputs are actions of the physical world on the system whereas the outputs are actions of the system on its surrounding environment. fig. 2.1 is a simplified depiction of an embedded system that controls access to some restricted area. Access can be gained by entering a pin on a numeric pad or by sending an HTTP request. If the pin or HTTP request are correct then the door is unlocked. The inputs to this system are radio packets and pin entries; the outputs are radio packets, a display over the numeric pad, the door lock and an LED.

Under the *tasks and resources* model the state of the system is divided in *resources*. In the example, the state of the LED, the state of the door and the radio buffer are *resources*. Under the model, the logic of the system – the instructions it will execute – are split in *tasks*. Tasks can start in response to inputs (external events) or can be initialized by the software. When a task spawns another it can pass a message to the receiving task. In the example, the `on_new_packet` task starts in response to the arrival of a new radio packet and this task spawns and passes a message, a packet, to the `process_packet` task. Tasks can be

spawned immediately or to run at some point in the future. In both cases, the operation is asynchronous; the spawned task may not be executed immediately or even before the spawning task ends. In the example, the `heartbeat` task spawns itself one second in the future; this creates a periodic task.

The system is inherently reactive: it performs work in response to external events. When no task is being performed, the system executes a never ending background task referred to as `idle`. In the scenario where no work needs to be performed in the background the system can be put in power saving mode.

This model does not specify how tasks are scheduled or how concurrent access to resources is synchronized.

2.2 Stack Resource Policy

The Stack Resource Policy (SRP) is a resource allocation policy that gives the tasks and resources model several properties: schedulability can be easily tested, priority inversion is bounded and absence of deadlocks is guaranteed [7]. These are particularly desirable properties for building hard real time applications and the absence of deadlocks is a desirable property for all kind of applications.

Under SRP tasks are assigned a static priority and scheduling is based on priorities. Tasks with higher priority are to be executed first. Under SRP tasks have *run to completion* semantics. This means that if two tasks, A and B, have the same priority then task A must run from start to finish before task B can or the other way around (B runs first); there is no context switching between tasks that have been assigned the same priority.

SRP allows priority-based preemption. In the scenario where a higher priority task becomes available, the current task is suspended and the higher priority task is executed from start to finish; after the higher priority task ends the previous lower priority task is resumed.

Before a task can access or modify a resource it must first *claim* it. The process of claiming a resource consists of temporarily raising the priority of the task to a *ceiling priority*. The act of raising the priority prevents the start of task with static priority equal to or less than the ceiling priority. Each resource is assigned a ceiling priority; the ceiling priority is chosen to be the maximum priority among the tasks that may access the resource.

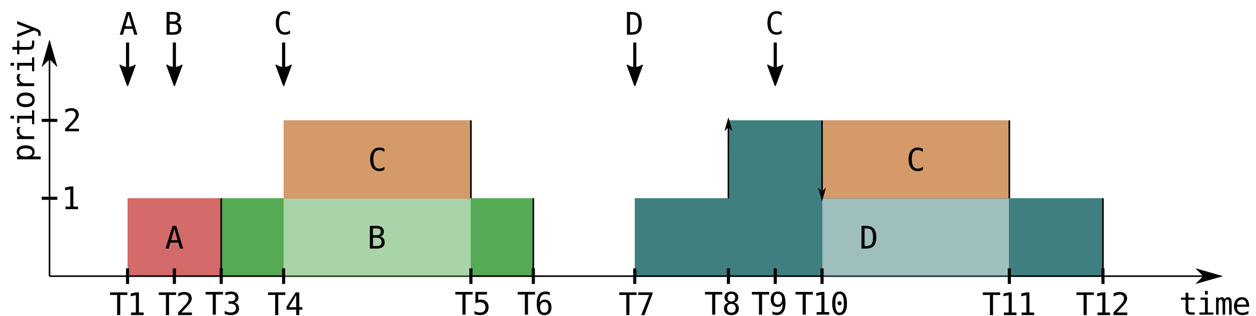


Figure 2.2: Task scheduling and resource claiming

The example in fig. 2.2 illustrates task scheduling and resource claiming. In this example 4 tasks and 1 resource are involved; the tasks are named A, B, C and D and all are triggered by external events; tasks A, B and C have a priority of 1; and task D has a priority of 2. The resource is shared between tasks C and D thus it is assigned a priority ceiling of 2.

The events in the example occur as follows:

- At time T1 an external event starts task A.
- At time T2 the external event that starts task B arrives. As tasks A and B have the same priority nothing immediately occurs.
- At time T3 task A ends and the pending task B starts execution.

- At time T4 the external event that starts task C arrives. Preemption occurs: task B is suspended and task C starts.
- At time T5 task C ends and task B resumes.
- At time T6 task C ends.
- At time T7 an external event starts task D.
- At time T8 task D claims the resource; this raises the priority of task D to 2, the ceiling priority.
- At time T9 the external event that starts task C arrives. As the current priority of task D is equal to the static priority of task C, nothing immediately happens.
- At time T10 task D releases the resource; this lowers the priority of task D back to 1 again. This causes the pending task C to preempt task D.
- At time T11 task C ends and task D is resumed.
- At time T12 task D ends.

SRP does not cover message passing. The properties of this abstraction are implementation dependent and are studied in more depth in the implementation sections of chapters 4 and 5.

As a side note, the response time analysis developed by Baker for SRP makes the following assumption: when several tasks with the same priority are pending then the *oldest* one will be serviced first. Our implementation does not emulate this behavior – though it could at extra overhead – so Baker's response time analysis would need to slightly modified to be applicable to our implementation.

2.3 Multi-core environment

Several algorithms have been devised for sharing a resource in multi-core real-time applications. The locking mechanism can be divided in suspension-based and spin-based. The most notable algorithms from each group are MPCP (Multiprocessor Priority Ceiling Protocol) [8] and MSRP (Multiprocessor Stack Resource Policy) [9]. Also worth mentioning is MrsP [10], a recent hybrid between the suspension-based and spin-based mechanisms.

MPCP allows both local and global resources; the former are constrained to a specific core; the latter are shared between different cores. Under MPCP claiming a local resource is done using the Priority Ceiling Protocol (PCP). Claiming a global resource is done using a synchronization processor; the critical section is executed on the synchronization processor. Processors that are unable to claim a global resource suspend their current task. The suspension consists of switching to a lower priority task. When a processor releases a global resource the highest priority task that was waiting for the resource is unblocked and becomes allowed to lock the global resource. MPCP does not allow nesting of global locks because nesting leads to excessive blocking time.

MSRP also allows both local and global resources. Local resources are claimed using SRP. Global resources are claimed using a critical section and a spinlock. When contention occurs on a global resource the other processors block (busy wait) at the highest priority (preemption is disabled). Nesting global critical sections is not allowed as it leads to deadlocks.

MrsP is very similar to MSRP except that spinning is done at a local ceiling priority rather non-preemptively and a *helping* mechanism is used to reduce the amount of busy waiting. Helping consists of having spinning processors opportunistically execute the global critical section of a different processor. To illustrate the concept consider the following scenario: tasks T1 and T2 run on different processors and both need to access a global resource R. Task T1 claims R and, immediately after, T2 also claims R. T1 gets the lock and T2 is put to spin at some local ceiling priority. Next, T1 gets preempted by a higher priority task while executing its global critical section. This is where helping occurs: T2 stops spinning and proceeds to continue executing the global critical section that T1 was executing. After the second core finishes executing T1's global critical section, it proceeds to execute T2's global critical section. MrsP permits nested locks and avoids deadlocks by statically ordering access to global resources.

All these algorithms are not without flaws. MPCP requires that global critical sections run on a synchronization processor; this makes it unsuitable for heterogeneous multi-core processors where the cores may not

have compatible instruction sets. Furthermore, it is not possible to have all tasks share the same call stack in MPCP due to the task suspension it performs on contention. The main downside of MSRP is the wasted CPU time that occurs on lock contention. MrsP addresses the problem of busy waiting with the concept of helping but helping requires symmetric processors; this makes MrsP unsuitable for heterogeneous multi-core processors.

In multi-core RTFM, neither MPCP nor MSRP is used: resource sharing between cores is simply not allowed. The only option for cross-core communication is message passing. Like in MSRP task partitioning is used: tasks must be assigned to a core – system partitioning was also explored in rtfm-lang in the context of mixed criticality systems [11]. That is each task will run *only* on a host core; not allowing task migration makes multi-core RTFM compatible with heterogeneous multi-core microcontrollers.

Fig. 2.3 depicts a multi-core port of the single-core application that was shown in fig. 2.1. In this multi-core version tasks and resources have been split between the two cores. One core owns the radio and takes care of receiving and transmitting radio packets; the other core is in charge of processing the radio packets and unlocking the door. The first core communicates with the second using message passing: the received packet is sent from the first core to the second core for processing.

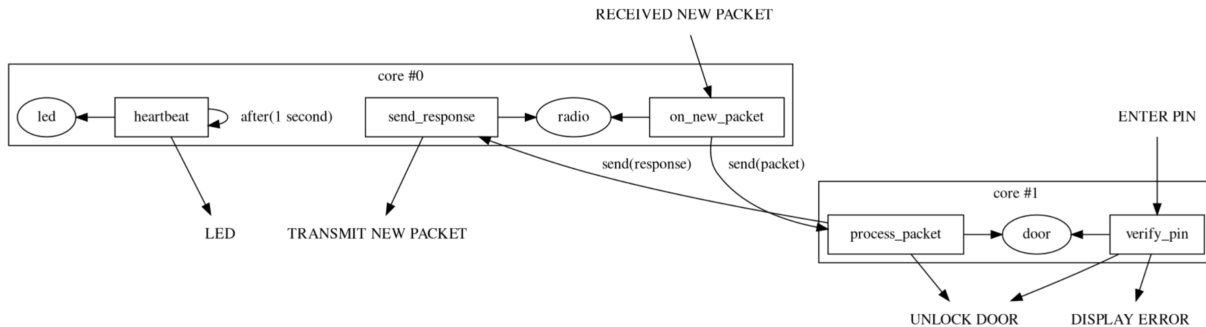


Figure 2.3: Multi-core task partitioning

The Stack Resource Policy is used to independently schedule tasks and manage resources within each core. The properties of SRP will hold for the whole system only if the implementation of cross-core message passing preserves them. This last point will be further discussed in the implementation section of chapter 5.

Chapter 3

Rust

Rust is a systems programming language designed to avoid the memory safety bugs that commonly occur in languages like C and C++. Problems like data races and use after free are eliminated at compile time by incorporating the concepts of ownership and borrowing into the language (type system).

3.1 Memory safety

This section covers the core features of Rust that set it apart from mainstream programming languages and guarantee memory safety at compile time.

3.1.1 Ownership

The compiler tracks the lifetime of resources, like memory allocations, through variable assignments and function calls. A resource assigned to a variable is said to be *owned* by the variable. Passing a variable to a function causes the ownership of the resource to be transferred to the callee; this operation in known as a *move* in Rust terminology. After moving the resource out of a variable the variable can no longer be used to access the resource. When a variable goes out of scope the resource it owns is *freed*; this free operation consists of calling the resource *destructor*, if it has one.

Listing 3.1 Ownership example

```
1  fn main() {
2      let x = Box::new(0);
3      foo(x);
4      //  - value moved here
5      // println!("{}", x);
6      //                ^ value borrowed after move
7  }
8
9  fn foo(y: Box<i32>) {
10     println!("{}", y);
11 } // `y` goes out of scope
```

Lst. 3.1 showcases the move and free operations previously described. On line 2, an integer is allocated on the heap and the resulting memory allocation, the *owning* pointer (`Box`), is bound to variable `x`. On line 3, the memory allocation is *moved* from the variable `x` to the function `foo`.

The function `foo` receives the memory allocation and binds it to variable `y` (line 9). On line 10 the contents of the memory allocation are printed to the console. On line 11 the variable `y` goes out of the scope and its associated memory allocation is deallocated (returned to the heap).

If line 5 is uncommented then the compiler rejects the program. That line tries to print the contents of the variable `x` but the variable no longer owns any resource so the operation is rejected. Lines 4 and 6 show the error messages reported by the compiler. Had the compiler accepted this program it would have resulted in a use after free error at runtime.

3.1.2 References

When a move is not desired *borrowing* can be used. Borrowing a value creates a reference to it. References, denoted by the `&` (ampersand) sign, in Rust are pointers that are guaranteed to be valid at compile time. References are never null and always point to a live and valid memory location.

At the core of all data races is an incorrect mix of aliasing and mutation. Aliasing refers to the process of creating many pointers to the same memory location. Using these pointers to modify the same memory location from different threads is a data race that results in *undefined behavior* where the compiler is free to optimize the program in ways to change the intended semantics of the program.

To cope with the problem of unsynchronized mutation the Rust programming languages provides two kinds of references: *immutable* references (`&-`) and *mutable* references (`&mut-`). The language allows *many* immutable references to the same memory location OR a *single* mutable reference to exist at any time. This restriction is referred to as the "Rust aliasing rule".

The official names, "immutable" and "mutable", are a bit misleading because it is possible to mutate a memory location through an *immutable* reference (`&-`) in some cases. In this work the terms *shared* references (`&-`) and *unique* references (`&mut-`) will be used instead as these better match the concept of *aliasing* which is what the compiler is checking.

Listing 3.2 (incorrect) borrowing example

```
1 fn main() {
2     let mut xs = vec![0];
3     let x: &i32 = &xs[0];
4     //             -- immutable borrow occurs here
5     xs.push(2);
6     //~^ error: cannot borrow `xs` as mutable because it is also borrowed as immutable
7     let z: i32 = *x;
8 }
```

These two types of references are used by abstractions to prevent problems like dangling pointers and data races. Consider the example in lst. 3.2, which is rejected by the compiler. In line 3, a shared reference to the first and only element in the vector (growable array) `xs` is stored in `x`. In line 5, a new element is added to the vector `xs`; this operation may cause the vector to be relocated in memory. In line 7, the reference (pointer) `x` is read and the loaded value is stored in the variable `z`.

Because of the potential relocation in line 5 this program could run into undefined behavior (deference of a dangling pointer). However, the Rust compiler rejects this program because it does not adhere to the Rust aliasing rule. The `push` operation requires a *unique* reference (`&mut-`) to the vector `xs`. However, a shared reference to the vector, `x`, already exists at that point so the borrow operation performed by `push` is not allowed by the compiler.

The unique borrow (`&mut-`) required by the `push` operation is a restriction that says that no other reference to the vector or any element in it is allowed to exist while the operation is performed. This restriction is required for memory safety because the operation can turn any such reference into a dangling pointer due to the potential relocation that may occur.

3.1.3 Lifetimes

The compiler uses *lifetimes* to track the liveness of variables. A lifetime, or a *region* as originally referred to in academic literature [12], corresponds to a span of code; the span can be as small as a single function

Listing 3.3 Rust lifetimes visualized

```
fn main() {                      'x
    let mut x = 0;               +   'y
    let y: &'w mut i32 = &mut x; |   + +
                                 |   | |
    let z: &'w mut i32 = y;      | + + |
                                 | |   |
    drop(z);                     | +   +
                                 | 'z  'a
    bar(&x);                     |
       +--+ 'b                   |
}                                +
```

argument or several lines long. Lifetimes rarely need to be annotated in code but when they do they appear with the syntax: `'identifier`.

Consider lst. 3.3, a Rust program with invalid syntax – most lifetimes cannot be named within a non-generic function. Lifetimes `'x` (lines 2-11), `'y` (lines 3-5) and `'z` (lines 5-7) are the lifetimes of variables `x`, `y` and `z`, respectively. References have a lifetime in their type that indicates the lifetime (span) of the borrow. In the listing, lifetime `'a` (lines 3-7) corresponds to the span of the unique borrow of `x`. Lifetime `'b` corresponds to a short-lived shared borrow of `x`.

The borrow checker checks that a unique borrow to a variable, like `'a`, does not overlap with a shared borrow of the same variable, like `'b` – that would be a violation of the Rust aliasing rule. It also checks that borrows do not outlive (have a span longer than) the lifetime of the data they refer to; in the example, neither `'a` nor `'b` can outlive `'x`.

3.1.3.1 `'static`

`'static` is a special lifetime that corresponds to no particular span of code. When it appears within a reference it indicates that the value behind the reference will never be deallocated. `&'static` - references occur naturally when interacting with static variables. Borrowing a static variable produces a `&'static` - reference.

3.1.4 Panicking

Not all operations can be verified to be memory safe at compile time. One example is indexing a slice, a partial view into an array. In this case, the indexing operation contains a runtime check to check if the index is within bounds; at runtime slices carry information about their length for this purpose. If the index is out of bounds then the result is a *panic*. When using the standard library, a panicking condition can result in either aborting the whole program or just unwinding the stack of the thread that ran into the panic. The unwinding process walks up the stack freeing all live resources before terminating the thread. Panicking is used not only to enforce memory safety but also to uphold contracts at the function call boundary.

3.1.5 `unsafe`

The Rust language also has the concept of (raw) pointers (`*mut _`) in the C sense. These pointers are exempt from the borrow checker so they may be null, may dangle or may point to dead / freed memory and thus are dangerous to work with. To indicate that these pointers need special attention from the programmer the language has the concept of *unsafe* operations. Unsafe operations cannot be proven to be memory safe by the compiler; the programmer must manually verify that they are indeed memory safe.

Dereferencing a raw pointer and unsynchronized access to a static (`static mut`) variable are examples of language-level unsafe operations but functions can also indicate if they are a safe or unsafe operation. Unsafe

operations must be wrapped in an `unsafe` block. This makes them easy to spot and facilitates the process of auditing Rust code for memory bugs.

Applications are usually written using only *safe*, that is non-`unsafe`, code; this means that if the code passes compilation then the program is proven to be memory safe. On the other hand, libraries that provide abstractions to applications may occasionally need to perform `unsafe` operations in the implementation of their abstractions. However, they tend to do so in a way that the abstraction provides a safe, not `unsafe`, API for applications to use.

It needs to be stressed that `unsafe` is only used to establish a clear boundary between code that has been machine checked to be memory safe (safe Rust) and code that needs to be manually verified to be memory safe (unsafe Rust). `unsafe` does *not* disable compile time checks like the borrow checker and the type checker nor does it disable move semantics; those checks and properties also apply to unsafe Rust.

3.2 Concurrency

Rust provides two *marker traits* in the core library that are used to build concurrency primitives: `Send` and `Sync`. A type can implement a marker trait or not. Certain operations require that the involved types implement one of these traits; trying to use types that do not implement the trait results in a compile error.

The definitions of these marker traits, according to the standard library documentation, are the following:

- `Send`, types that can be transferred across thread boundaries. [13]
- `Sync`, types for which it is safe to share references between threads. [14]

Where "thread" refers to the time-sliced thread abstraction found in general purpose operating systems.

In this work a more general definition of these traits is used. Implementers of the `Sync` trait are understood as types for which *concurrent access* through a shared reference (`&-`) is memory safe. Where concurrent access could mean 2, or more, cores accessing the value in parallel but can also mean two *contexts* running on the same core accessing a value concurrently. Examples of the latter scenario include POSIX signal handlers on a single-core system and time-sliced threads running on a single-core system. `Send` implementers are understood as types which are memory safe to *move* from one context to another where both contexts are *running concurrently*. Here, running concurrently means that the execution of the contexts may overlap, due to parallelism or context switching.

At the language level it is defined that the type `&'a T` implements the `Send` trait if `T` is `Sync` and it is required that `static` variables must hold types that implement the `Sync` trait. The latter requirement enforces memory safety: any context can get a shared reference (`&-`) to a `static` variable that is in scope so the `Sync` requirement ensures that any access from said shared reference is memory safe.

3.2.1 Sync

The prime example of a `Sync` type are atomic types like `AtomicU8` (8-bit integer), `AtomicU16` (16-bit integer), etc. These types can be modified concurrently and in a data race free manner using Compare-And-Swap (CAS) loops.

In lst. 3.4 two time-sliced threads get a shared reference to the atomic variable `X` and then modify the same memory location concurrently. The first thread gets the shared reference explicitly (line 4) whereas the second one gets the reference implicitly (line 10): the method call automatically borrows the variable `X` for the duration of the invocation.

A type that allows mutation through a shared reference but does not implement the `Sync` trait is the `Cell` type. Lst. 3.5 shows an unsound example that is rejected by the compiler − line 8 shows the error message reported by the compiler. The example is unsound because the write operation in line 11 is not atomic on 32-bit architectures: it requires two write instructions at the machine code level. As both time-sliced threads can overlap in execution there could be a context switch from the first thread to the second at line 11, after

Listing 3.4 Concurrent modification of an atomic variable

```rust
static X: AtomicU8 = AtomicU8::new(0);

fn thread0() {
    let x: &AtomicU8 = &X;
    let prev = x.fetch_add(1, Ordering::AcqRel);
    // ..
}

fn thread1() {
    let prev = X.fetch_add(1, Ordering::AcqRel);
    // ..
}
```

Listing 3.5 (unsound) Non-`Sync` type in a `static` variable

```rust
#[repr(u64)]
enum E {
    A = 0x0000_0000_ffff_ffff,
    B = 0xffff_ffff_0000_0000,
}

static X: Cell<E> = Cell::new(E::A);
//~^ error: `Cell<E>` cannot be shared between threads safely

fn thread0() {
    X.set(E::B);
}

fn thread1() {
    match X.get() {
        E::A => { /* .. */ }
        E::B => { /* .. */ }
    }
}
```

the first write instruction occurs but before the second write instruction is executed. This condition is known as a *torn write* and could result in the second thread observing `X` as containing an all zeros value or an all ones value – neither of which is a valid variant of the enumeration `E` – so the result is undefined behavior.

3.2.2 Send

An example of a `Send` type is `Arc<T>`, the atomically reference counted type [15]. `Arc<T>` is a smart pointer that is cheap to clone: the cloning operation consists of increasing the reference count and then handing back a copy of the pointer. The value `T` and reference count behind the pointer are allocated on the heap; when an `Arc<T>` value is destroyed it first decreases the reference count and then, if the count has fallen to zero, calls `T`'s destructor and frees the heap allocation behind the pointer. The reference count is an atomic type so both the clone and drop operations are safe to execute concurrently.

Lst. 3.6 showcases a program where a `Send` type is used. The first thread creates an `Arc` (line 2) and immediately clones it (line 3). Then it proceeds to spawn a new thread (lines 5-7) using the `thread::spawn` API [16]. This API executes the closure passed to it and requires that all values captured by the closure implement the `Send` trait. In this case the closure includes a capture, by value, of the variable `y`; semantically it is said the variable `y` has been *sent* to the second thread. Each thread prints the value behind the smart

Listing 3.6 A `Send` type

```
1  fn thread0() {
2      let x = Arc::new(0); // count = 1
3      let y = x.clone(); // count = 2
4      thread::spawn(|| {
5          println!("{}", y);
6          drop(y); // count -= 1
7      });
8      println!("{}", x);
9      drop(y); // count -= 1
10 }
```

pointer it owns and then proceeds to destroy (`drop`) it. Each `drop` operation decreases the same reference count; due to the time-sliced nature of these threads the operation may occur concurrently so it is unknown which thread will free the memory allocation behind the smart pointer but the deallocation would occur only once.

An example of a type that does not implement the `Send` trait is `Rc<T>`, a reference counted type [17] that is the non-atomic version of `Arc<T>`. Substituting the `Arc` in lst. 3.6 would result in an unsound program that is rejected by the compiler. The program would be unsound because the two unsynchronized `drop` operations would result in a data race that may cause the heap allocation to be freed twice.

3.3 Other features

This section covers other features that are used in the Rust implementation of Real Time For the Masses or are relevant to the stack usage analysis.

3.3.1 Generics

In Rust "generics" refers to parametric polymorphism as found in C++ templates and Java Generics. Instead of C++'s class inheritance Rust provides a trait system similar to Java interfaces.

Lst. 3.7 showcases the features that make up the concept of "generics" in Rust. `Pair` in line 1 is a generic struct with two unnamed fields of the same type. In lines 4 and 5 the generic struct `Pair<T>` is instantiated with different type parameters as `Pair<i32>` and `Pair<bool>`, respectively. Function `foo` (lines 21-23) is a generic function that takes an argument whose type must implement the `Frob` trait. This function is instantiated for types `X` and `Y` in lines 6 and 7. The commented out operation in line 8 is not allowed by the type system because `Z` does not implement the `Frob` trait.

3.3.1.1 Const generics

Const generics is a recent addition to the type system that allows using integers as type parameters. The main use case for this feature is creating data structures that are generic over some sort of size or capacity. Arrays (`[T; N]`) can be seen as a built-in data structure that is generic over its length. Lst. 3.8 shows a user defined `Array` type that is also generic over its length.

3.3.2 Polymorphism

Rust provides two (dynamic) polymorphism features: enumerations (`enum`) and trait objects (`dyn Trait`).

3.3.2.1 Enumerations

Enumerations (`enum`) are a form of *closed set* polymorphism that in their simplest form are equivalent to C `enum`s. Consider the example in lst. 3.9. Each `enum` type has a *finite* set of variants (lines 2-3); unlike

Listing 3.7 Generics in Rust

```
struct Pair<T>(T, T);

fn main() {
    let a: Pair<i32> = Pair(0, 1);
    let b: Pair<bool> = Pair(false, true);
    foo(X);
    foo(Y);
    // foo(Z); //~ error: the trait `Frob` is not implemented for `Z`
}

trait Frob {
    fn frob(&self);
}

struct X;
impl Frob for X { /* .. */ }
struct Y;
impl Frob for Y { /* .. */ }
struct Z;

fn foo(x: impl Frob) {
    x.frob();
}
```

Listing 3.8 Const generics in Rust

```
struct Array<T, const N: usize>([T; N]);

fn main() {
    let x: Array<bool, 2> = Array([false, true]);
    let y: Array<i32, 3> = Array([0, 1, 2]);
}
```

variants in C `enum` variants can have fields like `struct`s have (line 2). At runtime an `enum` value carries a tag that identifies in which variant the `enum` currently is; for instance the variable `x` is in the `A` variant in line 7 but in the `B` variant in line 9. To access the data within an `enum` value the `match` operator must be used (line 13); this operator forces one to declare how to handle each potential variant the `enum` may be in. At runtime the `match` operation checks the `enum` tag to decide which arm to execute (line 8).

3.3.2.2 Trait objects

Trait objects are a form of *open set* polymorphism. A trait object is a pointer into a value that implements some known trait (interface). The type behind the pointer is not known at compile time but all the trait methods can be used on the trait object. At runtime the correct implementation for a trait method invocation is selected using a virtual table stored in the trait object (as a pointer) – this process is referred to a dynamic dispatch.

The example in lst. 3.10 illustrates the capabilities of a trait object. Two zero sized structures `X` and `Y` (lines 5 and 8) implement the trait `Frob` (line 1). Instances of both structures are created in line 12. In line 13, a `Frob` trait object is created and made point to a `X` value. Then, in line 14, the method `frob` is invoked on the trait object; this causes `X`'s implementation of `frob` to be executed. In line 15, the trait object is modified and made point to a `Y` value. In line 16, `frob` is invoked on the trait object again but this time it causes `Y`'s implementation of `frob` to run. In general, it is not possible to store a reference to a type (e.g. `&X`) in a variable and then modify the variable to hold a reference to a *different* type (e.g. `&Y`); trait objects allow

Listing 3.9 Using an enumeration

```rust
enum E {
    A(u8),
    B,
}

fn main() {
    let mut x: E = E::A(0);
    print(&x);
    x = E::B;
}

fn print(e: &E) {
    match e {
        E::A(a) => println!("variant is A with an inner value of {}", a),
        E::B => println!("variant is B"),
    }
}
```

this operation but only as long as the types implement the same trait.

In enumerations the number of the variants the enumeration has is determined when the enumeration is declared (closed set polymorphism) but with trait objects the set of potential types that could be behind a trait object can grow by linking to third party libraries because these libraries can define new implementations of the trait (open set polymorphism).

3.3.3 Procedural macros

Procedural macros are a metaprogramming feature that performs a source code level transformation on some input source code. The source code output by a procedural macro is referred to as its *expansion*.

The most commonly used procedural macros are the `#[derive]` attributes. These attributes are used to automatically implement a trait for a structure or enumeration.

Lst. 3.11 shows how the `#[derive]` attribute is used to automatically implement the `Clone` trait. Lst. 3.12 shows the expansion of lst. 3.11.

Procedural macros are defined in code as functions that take an input token stream and produce an output token stream. Lst. 3.13 shows the implementation of an `#[identity]` attribute that performs no transformation on the input item.

3.3.4 Compilation targets

The official Rust compiler, `rustc`, supports a wide range of platforms and architectures. Information about different platforms are encoded as *compilation targets*. Each compilation target is identified by a *target triple*, a string of the form `one-two-three-four`.

Passing a compilation target to `rustc` makes it compile the source code to machine code optimized for the target platform. tbl. 3.1 lists compilation targets that correspond to some of the embedded variants of the ARM architecture.

Table 3.1: Embedded compilation targets built into `rustc`

Compilation target	Architecture	Cores
`thumbv6m-none-eabi`	ARMv6-M	Cortex-M0, Cortex-M0+
`thumbv7em-none-eabi`	ARMv7E-M	Cortex-M4, Cortex-M7

Compilation target	Architecture	Cores
`thumbv7em-none-eabihf`	ARMv7E-M	Cortex-M4F, Cortex-M7F
`thumbv7m-none-eabi`	ARMv7-M	Cortex-M3

3.3.5 Conditional compilation

Conditional compilation is a compiler feature that controls which code is included in an application based on some condition holding or not. Most of the conditionals the compiler checks by default are related to the compilation target: code can be included, or omitted, based on the target architecture (ARM vs x86), target endianness (big vs little), pointer width (32-bit vs 64-bit), etc. But it is possible to define new conditionals that are enabled by passing the `--cfg` flag to the compiler.

Consider the example in lst. 3.14. The `#[cfg]` attribute (lines 1 and 4) can be used to include, or omit, entire items. The `cfg!` macro returns `true` if the conditional evaluates to `true` (line 8). Conditionals can be chained using `any` or `all`; `any` is equivalent to a logical OR and `all` is equivalent to a logical AND.

`target_arch` is one of the built-in conditionals. `single_core` and `core` are user-defined conditionals (line 12) that are enabled with the compiler flags `--cfg single_core` and `--cfg core=0`, respectively.

3.4 `core` library

When writing embedded Rust code only a subset of the standard library is available: the `core` library. This section covers the abstractions provided by `core` that are used in the implementation of RTFM.

3.4.1 Raw pointer methods

Raw pointers have `read` and `write` methods to perform reads and writes on the memory location indicated by the pointer; these methods are preferred to using the dereference operator (`*`) which can cause unintentional moves.

The `read` method (lst. 3.15 line 2) reads the value behind the pointer without *moving* it. The operation leaves the memory location unchanged. This operation is `unsafe` because the caller needs to make sure that the pointer is pointing to valid memory (e.g. not deallocated), that the pointer is properly aligned and that the operation does not cause aliasing (e.g. `*const Vec<u8>`).

The `write` method (lst. 3.15 line 3) writes `val` into the memory location indicated by the pointer *without* first destroying the value that was already there. This operation is `unsafe` because the caller needs to make sure that the pointer is pointing to valid memory and that the pointer is properly aligned.

Raw pointers also have *volatile* variants of the `read` and `write` methods (lst. 3.15 lines 5-6). The compiler is not allowed to change the order or the number of volatile operations. These semantics match the semantics of C11's volatile modifier.

3.4.2 `MaybeUninit`

`MaybeUninit` is a newtype, a data type that has the same memory layout as its only inner field, used to represent a memory location that may be uninitialized.

Lst. 3.16 lists the `MaybeUninit` API. The `uninitialized` method (lst. 3.16 line 2) is a constructor that creates a `MaybeUninit` value in a uninitialized state. At runtime the `MaybeUninit` value does not track whether the memory location has been initialized or not. For that reason the main way to interact with a `MaybeUninit` value is through a raw pointer which can be obtained from the `as_ptr` or `as_mut_ptr` method (lst. 3.16 lines 3-4).

Listing 3.10 Using an enumeration

```
trait Frob {
    fn frob(&self);
}

struct X;
impl Frob for X { /* .. */ }

struct Y;
impl Frob for Y { /* .. */ }

fn main() {
    let (x, y) = (X, Y);
    let mut to: &dyn Frob = &x;
    to.frob();
    to = &y;
    to.frob();
}
```

Listing 3.11 A `derive` attribute

```
#[derive(Clone)]
struct Pair { x: i32, y: i64 }
```

3.4.3 Atomic

`core` provides several atomic types like `AtomicBool`, `AtomicU8` and `AtomicUsize`. The semantics of these types match the semantics of C11 atomics.

The API of `AtomicU8` is shown in lst. 3.17. The `load` and `store` methods (lines 10-11) perform an atomic read or write to the memory location. The `Ordering` (lines 1-7) argument indicates what kind of memory barrier to attach to the atomic read / write operation. The correct `Ordering` argument must be used with `load` and `store` to properly synchronize the operation across different cores.

The `compare_exchange` method (line 12) atomically updates the memory location to the `new` value if and only if the memory location held the `current` value at the start of the method invocation. This method is commonly used in loops to implement lock-free data structures – these loops are referred to as Compare And Swap (CAS) loops.

The `fetch_add` (line 13), and the other `fetch_` variants, atomically update the memory location. `fetch_add`, in particular, increments the memory location by `val`. These methods are usually internally implemented with CAS loops.

Listing 3.12 The expansion of lst. 3.11

```rust
struct Pair { x: i32, y: i64 }

impl Clone for Pair {
    fn clone(&self) -> Self {
        Pair { x: self.x.clone(), y: self.y.clone() }
    }
}
```

Listing 3.13 The implementation of the `identity` attribute

```rust
#[proc_macro_attribute]
pub fn identity(args: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

Listing 3.14 Different forms of conditional compilation

```rust
#[cfg(target_arch = "arm")]
const ARCH: &str = "ARM";

#[cfg(any(target_arch = "x86_64", target_arch = "x86"))]
const ARCH: &str = "x86";

fn is_arm() -> bool {
    cfg!(target_arch = "arm")
}

fn first_core() -> bool {
    cfg!(single_core) || cfg!(core = "0")
}
```

Listing 3.15 Raw pointer methods

```rust
impl<T> *mut T {
    unsafe fn read(self) -> T { /* .. */ }
    unsafe fn write(self, val: T) { /* .. */ }

    unsafe fn read_volatile(self) -> T { /* .. */ }
    unsafe fn write_volatile(self, val: T) { /* .. */ }
}
```

Listing 3.16 `MaybeUninit` API

```rust
impl<T> MaybeUninit<T> {
    fn uninitialized() -> Self { /* .. */ }
    fn as_ptr(&self) -> &T { /* .. */ }
    fn as_mut_ptr(&mut self) -> &mut T { /* .. */ }
}
```

Listing 3.17 `Atomic*` API

```
pub enum Ordering {
    Relaxed,
    Release,
    Acquire,
    AcqRel,
    SeqCst,
}

impl AtomicU8 {
    fn load(&self, order: Ordering) -> u8 { /* .. */ }
    fn store(&self, val: u8, order: Ordering) { /* .. */ }
    fn compare_exchange(
        &self, current: u8, new: u8, success: Ordering, failure: Ordering,
    ) {
        /* .. */
    }
    fn fetch_add(&self, val: u8, order: Ordering) -> u8 { /* .. */ }
}
```

Chapter 4

Single-core RTFM

This chapter covers the Rust port of single core RTFM. It describes the API and its implementation and then proceeds to analyze the memory safety and execution time of all the abstractions provided by the framework.

4.1 Overview example

Listing 4.1 Overview example

```
pool!(P: [u8; 128]);

#[rtfm::app(/* .. */)]
const APP: () = {
    struct Resources { radio: Radio }

    #[init]
    fn init(cx: init::Context) -> init::LateResources {
        // ..
        init::LateResources { radio: radio }
    }

    #[task(binds = EXTI0, priority = 2, resources = [RADIO], spawn = [process_packet])]
    fn on_new_packet(cx: on_new_packet::Context) {
        let new_packet = cx.resources.radio.next_packet();
        if let Some(buffer) = P::alloc() {
            let packet = new_packet.read_into(buffer);
            cx.spawn.process_packet(packet);
        } else {
            new_packet.discard();
        }
    }

    #[task(priority = 1, capacity = 4, /* .. */)]
    fn process_packet(cx: process_packet::Context, packet: Box<P>) { /* .. */ }
};
```

Real Time For the Masses is implemented as a DSL (Domain Specific Language) comprised of attributes on top of regular Rust items. In this section an example is presented (see lst. 4.1) to provide an overview of the main API of the framework.

Listing 4.2 The rest of the overview example

```
#[task(priority = 1, capacity = 4, resources = [radio], schedule = [turn_off_lights])]
fn process_packet(cx: process_packet::Context, packet: Box<P>) {
    // ..
    if some_command {
        cx.resources.radio.lock(|radio: &mut Radio| radio.send(response));
    } else if other_command {
        let when = cx.scheduled + Duration::from_secs(n);
        cx.schedule.turn_off_lights(when);
    }
    // ..
}

#[task(priority = 1)]
fn turn_off_lights(cx: turn_off_lights::Context) { /* .. */ }
```

The context of the example is a network application that performs actions based on packets received over a 802.15.4 radio. The radio interface is external and has limited memory: it can only hold one received packet in memory. That packet must be read, or discarded, before the interface can receive a new packet.

An RTFM application consists of a module to which the `#[app]` attribute is attached. Everything inside this module uses the RTFM DSL. Items inside the DSL can refer to normal Rust items defined outside the module. For example, a memory pool [18] is declared in line 1, outside the DSL, and then used in line 16, within the DSL.

A resource named `radio` is declared in line 5. This *late* resource is initialized at runtime; its initial value comes from the return value of the `init` function in lines 7-11.

The `init` function performs the initialization of the system. Although omitted, the memory pool `P` and several peripherals are initialized in that function, including the external radio. After being initialized, the `radio` value becomes the initial value of the `radio` resource.

The `on_new_packet` function in lines 13-22 is a *hardware* task bound to the `EXTI0` (external interrupt 0) interrupt. That interrupt fires when the radio has finished receiving a new packet and signals this to the microcontroller through a digital I/O pin. This task has access to the `radio` resource, from which it reads *metadata* about the newly received packet (`new_packet` in line 15). After reading the metadata the task tries to get a new memory block from the memory pool `P` (line 16). If there is enough free memory the task copies the *contents* of the packet from the radio into the memory block (line 17). If there is not enough memory the task tells the radio interface to discard the packet (line 20). In either case the radio interface can start receiving a new packet. After reading the contents of the packet the data is sent to the *software* task `process_packet` for further processing using message passing (line 18).

The software task `process_packet` receives the packet data (`packet` argument), parses it and performs some action based on its contents. After the task is done with the packet it returns the memory block to the memory pool `P` (not shown).

Each of these two tasks is given a different priority. The software task is given a lower priority, meaning that the `on_new_packet` task can preempt it. That way new packets can be copied out of the radio interface while old ones are being processed or pending processing.

The `process_packet` task is assigned a capacity of `4`; this means that its message buffer can hold a maximum of 4 messages (packets). This buffer lets the application deal with sporadic packet bursts.

Lst. 4.2 zooms into the `process_packet` task and shows the rest of the overview example. In some cases the software task may need to send a response to the client (line 5); to do that it needs to use the `radio` interface. When two or more tasks need to access the same resource, the lower priority one needs to `lock` it first to prevent a data race. `lock` creates a critical section, which appears as a closure in the code, and

only within this critical section the task has access to the radio interface (`&mut Radio`). The application has functionality to perform actions in the future. The `schedule` API is used in line 8 to schedule a task, that will turn off some light, `n` seconds in the future.

4.2 Design decisions

The framework requires user input like task priorities in order to compute the priority ceiling of resources, create buffers with the right size, etc. To make the framework as unintrusive as possible the implementation uses attributes on standard Rust items, like functions, to gather this additional information.

A closure API was chosen for the `lock` operation; this ensures that nesting locks is done in a LIFO manner. The alternative API for this operation is the "guard pattern", which is used by the `Mutex` abstraction in the standard library [19], but it is not possible to enforce strict LIFO nesting with that API.

4.3 The API

This section contains a summary of the RTFM API exposed to end users. For a more in depth explanation of the API along with usage examples the reader is encouraged to check the RTFM book [20] and the RTFM API reference [21].

4.3.1 Resources

Resources are declared using a `Resources` structure. Each field in this `struct` is a different resource. All resources default to being *late* resources, resources initialized at runtime. However, one can assign an initial value to a resource using the `#[init]` attribute; this turns it into an early resource. Lst. 4.3 shows the `struct Resources` syntax, an early resource and a late resource.

Listing 4.3 Declaration of resources

```
struct Resources {
    #[init(0)]
    early: u32,
    late: u32,
}
```

4.3.2 Tasks

Tasks are declared by attaching the `#[task]` attribute to functions. As described in the task model section (sec. 2.1) there are two kind of tasks: hardware tasks, tasks that start in response to external events, and software tasks, tasks spawned by other tasks; both use the `#[task]` attribute. Hardware tasks use the `binds` argument to bind the task to a particular interrupt; software tasks use the `capacity` argument to declare the size of their message buffers. Apart from those two type-exclusive arguments either type of task can use these arguments:

- `priority`, the static priority of the task
- `resources`, list of resources this task can access
- `schedule`, list of tasks this task can `schedule`
- `spawn`, list of tasks this task can `spawn`

The signature of a task handler must be `fn(task::Context, /* inputs */)`. The first argument is a task-specific `Context` structure whose fields reflect the capabilities of the task (see lst. 4.4). The following arguments, if any, correspond to the inputs of the task, that is the message passed to the task. Only software tasks can have inputs.

Apart from the `resources`, `schedule` and `spawn` fields, which reflect the capabilities of the task, the `Context` struct also contains a `scheduled` (software) or a `start` (hardware) field when the `schedule` API is being

Listing 4.4 the `Context` structure of a task

```
struct Context {
    resources: Resources,
    schedule: Schedule,
    scheduled: Instant, // or `start`
    spawn: Spawn,
}
```

used. This field represents the time at which the task was `schedule`d to run (software) or the time at which the task started executing (hardware), respectively. If a software task was `spawn`-ed instead of `schedule`-d then the `scheduled` field inherits its value from the task that spawned it.

4.3.3 `#[init]`

The `#[init]` attribute is used to declare the initialization function. This attribute can take any of the following arguments: `resources`, `schedule` and `spawn` whose semantics match that of `#[task]`'s arguments. The `#[init]` function must have signature `fn(init::Context) [-> init::LateResources]`. The return type must be used when the application has late resources and omitted when it does not.

The `init::Context` structure, shown in lst. 4.5, contains the `resources`, `schedule` and `spawn` fields seen in the task `Context`. In addition to those fields, it also contains a `core` field that packs all the Cortex-M peripherals and `start` field that represents the start time of the system, that is time *zero*.

Listing 4.5 `init::Context` structure

```
struct Context {
    core: rtfm::Peripherals,
    resources: Resources,
    schedule: Schedule,
    spawn: Spawn,
    start: Instant,
}
```

4.3.4 `#[idle]`

The `#[idle]` attribute is used to declare the background `idle` context. This attribute can take the following arguments: `resources`, `spawn` and `schedule`, whose semantics match that of `#[task]`'s arguments. The `#[idle]` function must have signature `fn(idle::Context) -> !`.

The `idle::Context` structure, shown in lst. 4.6, contains the `resources`, `schedule` and `spawn` fields seen in the task `Context`.

Listing 4.6 `idle::Context` structure

```
struct Context {
    resources: Resources,
    schedule: Schedule,
    spawn: Spawn,
}
```

4.3.5 `Resources`

The `Resources` structure that appears in the `Context` structure is a collection of resources that the task can access (example in lst. 4.7). Each field represents a different resource. The resource may directly appear as a unique reference (`&mut-`) to the resource data or as a proxy that must be `lock`-ed before the data can be

accessed. References appear when the resource is accessed by a single task and when the resource is shared but accessed from the highest priority task. When the resource is contended between two or more tasks, it appears as a proxy.

Listing 4.7 `Resources` struct

```
struct Resources<'a> {
    direct: &'a mut Type,
    proxy: Proxy<'a>,
    // ..
}
```

Lst. 4.8 shows the signature of the `lock` API present on all proxies. The method takes a closure as an argument. This closure is given temporary access to the resource data and it is executed only once. Due to the implicit lifetime constrains in the signature the reference given to the closure cannot escape the closure.

Listing 4.8 `lock` API expressed as a trait

```
trait Lock {
    type Data;

    fn lock<R>(&mut self, f: impl FnOnce(&mut Self::Data) -> R) -> R;
}
```

4.3.6 `Spawn`

The `Spawn` structure that appears in the `Context` structure provides an API to spawn tasks. Each task defined in the `spawn` list appears as a method on this structure. The signature of a task method is shown in lst. 4.9. The method is non-blocking and fallible; if the message buffer of the requested task is full the method returns an error (`Result::Err` variant) that wraps the message payload.

Listing 4.9 `spawn` API

```
impl<'a> Spawn<'a> {
    fn task(&self, payload: i32) -> Result<(), i32> { /* .. */ }
}
```

4.3.7 `Schedule`

The `Schedule` structure that appears in the `Context` structure provides an API to schedule tasks. Each task defined in the `schedule` list appears as a method on this structure. The signature of a task method is shown in lst. 4.10. The first argument of these methods is an instant that indicates when the task should run; the type of this argument is user defined (see sec. 4.3.8). Like in the `spawn` API, all methods are non-blocking and fallible.

4.3.8 `Monotonic`

To use the `schedule` API a monotonic timer must be specified using the `monotonic` argument of the `#[app]` attribute. This timer is used for time keeping and it is polled by the runtime to decide whether it's time to dispatch an scheduled task or not. The monotonic timer selected by the user must fulfill the `Monotonic` trait shown in lst. 4.11.

The `Instant` type associated to this trait represents an instant in time. This type must sortable (`Ord` bound in line 2) and subtracting two instances of it must return some duration type (`Sub` bound in line 2) that represents a span of time. The `now` method returns an `Instant` value that correspond to "now". The `reset` method resets the monotonic timer (counter) to some value considered "zero" – this method is called by the runtime *exactly once* after the initialization function returns and before tasks are allowed to run.

Listing 4.10 spawn API

```
impl<'a> Schedule<'a> {
    fn task(&self, when: Instant, payload: i32) -> Result<(), i32> { /* .. */ }
}
```

Listing 4.11 the Monotonic trait

```
trait Monotonic {
    type Instant: Ord + Sub;

    fn now() -> Self::Instant;
    unsafe fn reset();
}
```

4.3.9 Device bindings

Hardware tasks are bound to device interrupts and software tasks are also dispatched using unused device interrupts. Information about the interrupts available on the target device are passed to the runtime using the `device` argument of the `#[app]` attribute.

The `device` argument is a path to a module that must contain an `Interrupt` enumeration and an `NVIC_PRIO_BITS` constant.

`Interrupt` must be an enumeration of all the interrupts available on the target device; this `enum` must implement the `Nr` trait (shown in lst. 4.12) which maps each variant to its interrupt number. The interrupt number is the position of the interrupt in the device vector table. Two `enum` variants must *not* map to the same interrupt number in the implementation of the `Nr` trait; this is part of the contract of implementing the trait and breaking this contract can break memory safety. For this reason `Nr` is an `unsafe` trait.

Listing 4.12 the Nr trait

```
unsafe trait Nr {
    fn nr(&self) -> u8;
}
```

`NVIC_PRIO_BITS` must be a constant with type `u8` whose value is the number of priority bits supported by the device. A value of `2` indicates that the device supports 4 priority levels, `3` indicates 8 priority levels and so on.

4.4 Memory safety analysis

All the API exposed by the framework is presented as safe, thus using it with any other safe API should not result in a memory unsafe program. In this section, the correctness of this API is analyzed from the perspective of memory safety. Several scenarios where Rust safety guarantees could potentially be broken by the end user are studied.

4.4.1 Uninitialized memory

Late resources are initialized after `init` returns and are in an uninitialized state before that point. It is undefined behavior if any task accesses a late resource before it is initialized. Consider the example in lst. 4.13 where both a hardware task and software task are triggered during `init`.

The resource `X` is a boolean: it is only valid states are `false` and `true`. In machine code this boolean is implemented as a byte where the value `0` represents `false` and `1` represents `true`. At boot RAM starts in a randomized state so this boolean byte could have any value in the inclusive range `0` to `255`. If the software encounters a boolean byte who value is not `0` or `1` then undefined behavior occurs.

Listing 4.13 Attempting to access uninitialized memory

```
struct Resources { x: bool }

#[init(spawn = [a])]
fn init(cx: init::Context) -> init::LateResources {
    rtfm::pend(Interrupt::EXTI0); // raise signal EXTI0
    cx.spawn.a();
    init::LateResources { x: false }
}

#[task(binds = EXTI0, resources = [x])]
fn exti0(cx: exti0::Context) {
    let x: &mut bool = cx.resources.x;
    if *x { /* .. */ }
}

#[task(resources = [x])]
fn a(cx: a::Context) {
    let x: &mut bool = cx.resources.x;
    if *x { /* .. */ }
}
```

The program could encounter such scenario if `exti0` or `a` preempted the `init` function. However, by definition tasks can only start after `init` returns and the runtime will keep tasks blocked until after late resources are initialized so tasks `exti0` and `a` always observe resource `X` in an initialized state.

Another scenario where uninitialized memory could be observed is the program in lst. 4.14 where the user wants to access a late resource during the initialization phase.

Listing 4.14 Attempting to access a late resource from `init`

```
struct Resources { x: bool }

#[init(resources = [x])]
fn init(cx: init::Context) -> init::LateResources {
    let x: &mut bool = cx.resources.x;
    init::LateResources { x: false }
}
```

However, the DSL rejects this program at compile time: late resources cannot be assigned to `init`.

4.4.2 Aliasing

Accessing resources must adhere to Rust aliasing rules (sec. 3.1.2).

4.4.2.1 Serial access

Two same priority tasks can access a resource without a lock. Both tasks get an unique reference (`&mut-`) to the resource. This is safe because the tasks execute serially: one gets to run only after the other *finishes*; there is no preemption or context switching between them, therefore the lifetime of both tasks never overlaps. Lst. 4.15 depicts this situation.

It is possible to break memory safety in this scenario by changing the static priority of one of the tasks – this would make preemption possible and let the execution of the tasks overlap – but such operation is considered `unsafe`.

Listing 4.15 Serial access to a resource with no locks

```
#[task(binds = EXTI0)]
fn a(cx: a::Context) {
    let x: &mut u64 = cx.resources.x;
}

#[task(binds = EXTI1)]
fn b(cx: b::Context) {
    let x: &mut u64 = cx.resources.x;
}
```

4.4.2.2 Preemption

When two, or more, tasks declare access to a resource the lower priority task needs to use a critical section to access the resource data. A unique reference (`&mut-`) to the data is handed out to the lower priority task during this critical section. If the higher priority task were to start at this point two unique references (`&mut-`) to the resource data would exist and undefined behavior would arise. However, that cannot happen by definition: the critical section prevents the higher priority task from starting.

In the example in lst. 4.16 the `higher_priority` task can only preempt the `lower_priority` task in sections (I) and (III). This prevents the creation of a second unique reference.

Listing 4.16 the `lock` API is preemption safe

```
#[task(priority = 1, resources = [x])]
fn lower_priority(mut cx: lower_priority::Context) {
    // (I)
    cx.resources.x.lock(|x: &mut u64| { /* (II) */ });
    // (III)
}

#[task(priority = 2, resources = [x])]
fn higher_priority(cx: higher_priority::Context) {
    let x: &mut u64 = cx.resources.x;
}
```

The invariant would be broken if the reference could escape the critical section but the lifetime constraint of the `lock` API does not allow this operation. Lst. 4.17 showcases this scenario.

Listing 4.17 the reference to the data cannot escape the critical section

```
#[task(priority = 1, resources = [x])]
fn lower_priority(ctxt: lower_priority::Context) {
    let x: &mut u64 = ctxt.resources.x.lock(|x| x);
    //~^ error: cannot infer an appropriate lifetime due to conflicting requirements
}
```

4.4.2.3 Nesting `lock`s

The `lock` API creates a unique reference (`&mut-`) to the data. If two `lock`s to the *same* resource are nested this would result in two unique references to the same data existing at the same time. However, this operation is not allowed by the compiler due to the signature of the `lock` API: it requires a unique reference to the proxy so the proxy is uniquely borrowed (`&mut self`) during the whole execution of the critical section.

Lst. 4.18 is an example of this situation. The first `lock` operation (line 4) uniquely borrows the proxy `x` for the span of lines 4-6. The second `lock` operation (line 5) tries to uniquely borrow the proxy again but this

Listing 4.18 [Invalid] Nesting `lock` operations on the same resource

```
#[task(priority = 1, resources = [x])]
fn a(cx: a::Context) {
    let mut x = cx.resources.x;
    x.lock(|a: &mut u64| {
        x.lock(|b: &mut u64| { /* (I) */ });
    });
}
```

is rejected by the compiler. If that were not the case then at checkpoint (I) two unique references (`&mut-`), `a` and `b`, to the same memory location would exist.

Listing 4.19 Nesting `lock` operations on different resources

```
#[task(priority = 1, resources = [x, y])]
fn lower_priority(cx: lower_priority::Context) {
    let (mut x, mut y) = (cx.resources.x, cx.resources.y);
    x.lock(|a: &mut u64| {
        y.lock(|b: &mut u64| { /* .. */ });
    });
}
```

Note that nesting `lock`s to *different* resources adheres to aliasing rules and it is allowed by the DSL. The example in lst. 4.19 is accepted by the compiler.

4.4.2.4 Duplicating resources

More than one instance of a resource proxy cannot exist at any point in time as this would let the user get multiple unique references (`&mut-`) to the resource data by `lock`-ing each proxy separately.

The program in lst. 4.20 is invalid as it would create two instances of the same proxy. Line 3 bootstraps task `a`. On it first run task `a` sends its resource proxy `x` to its next instance using message passing (line 12). The second instance of task `a` would receive the proxy (line 9) as input and have access to a second instance of the proxy through the `Context` structure (line 10).

Listing 4.20 [Invalid] Using message passing to clone a resource proxy

```
#[init(spawn = [a])]
fn init(cx: init::Context) {
    cx.spawn.a(None);
}

#[task(priority = 1, resources = [x], spawn = [a])]
fn a(cx: a::Context, input: Option<resources::x>) {
    if let Some(y) = input {
        let alias: resources::x = y;
        let x: resources::x = cx.resources.x;
    } else {
        cx.spawn.a(Some(cx.resources.x));
    }
}
```

This kind of program is rejected by the DSL because, in this example, the message payloads must fulfill the `'static` bound and resource proxies do not fulfill this bound. This bound will be explained in sec. 4.5. Why resource proxies do not fulfill the `'static` bound will be covered in sec. 4.5.6.

4.4.3 Pointer invalidation

In general it is not known at compile time *when* a task will be dispatched after being `spawn`-ed or `schedule`-d. In most instances, however, the task will be dispatched *after* the task that `spawn`-ed / `schedule`-d it has finished. For this reason is it not safe to use references to values allocated on the stack frame of a task as the payload of a message. To reject this scenario all message payloads must fulfill the `'static` bound meaning that only references with `'static` lifetime can be part of a message payload.

The example in lst. 4.21, which is rejected by the DSL, showcases the problem. Task `a` sends a pointer into its stack (`&x`) to task `b`. As both tasks have the same priority task `b` will run *after* task `a` returns. At that point the stack variable `x` has already been deallocated so the dereference operation in line 9 would dereference an invalid pointer.

Listing 4.21 [Invalid] sending a pointer into the stack to another task

```
#[task(spawn = [b])]
fn a(mut cx: a::Context) {
    let x = 0;
    cx.spawn.b(&x); //~ error: `x` does not live enough
}

#[task]
fn b(mut cx: b::Context, input: &i32) {
    let x = *input;
}
```

4.4.4 Send bound

Listing 4.22 [Invalid] Sending types that do not implement the `Send` trait

```
struct Resources { x: Rc<i32> }

#[init(spawn = [a, b])]
fn init(cx: init::Context) -> init::LateResources {
    let x = Rc::new(0);
    cx.spawn.a(x.clone());
    init::LateResources { x: x }
}

#[task(priority = 1)] //~ may be preempted by `b` at any point
fn a(cx: a::Context, x: Rc<i32>) {
    let y = x.clone(); // increase reference count
    drop(y); // decrease reference count
    drop(x); // decrease reference count
}

#[task(binds = SomeInterrupt, priority = 2, resources = [x])]
fn b(cx: b::Context) {
    let x: &mut Rc<i32> = cx.resources.x;
    let y = x.clone(); // increase reference count
    drop(y); // decrease reference count
}
```

Consider the example in lst. 4.22 which is rejected by the DSL. In this example a reference counter pointer, `Rc<i32>`, is created in `init` and then copies of it are *sent* to tasks `a` and `b` – the latter is an indirect send operation caused by late resource initialization. As these tasks have different priorities their execution may

overlap – `b` has higher priority so it can preempt `a`. This results in tasks `a` and `b` modifying the reference count concurrently which is a data race because the count cannot be concurrently modified.

To prevent this and similar scenarios the DSL enforces a `Send` bound on message payloads according to these two rules:

1. If a value is *moved* from task A to task B and the two tasks run at different priorities then the value must implement the `Send` trait. For the effect of this analysis `init` and `idle` are both considered to have a priority of 0 and late resource initialization is also considered a *move*.
2. Sharing a resource with `init` also requires a `Send` bound, unless the resource is owned by `idle`. The reason for this is that one can emulate late resource initialization using this approach as exemplified in lst. 4.23.

As per these two rules the program in lst. 4.24 that makes uses of the `Rc` type is accepted by the DSL. Tasks `a` and `b` execute serially because they have been assigned the same priority. For this reason, the reference count of `x` is never modified concurrently.

Listing 4.23 Runtime initialization using an `Option`-al early resource

```
struct Resources {
    #[init(None)]
    x: Option<SomeType>,
}

#[init(resources = [x])]
fn init(cx: init::Context) {
    *cx.resources.x = Some(SomeType::new());
}

#[task(priority = 1, resources = [x])]
fn a(cx: a::Context) {
    if let Some(x) = cx.resources.x.take() {
        // value `x` has been _moved_ from `init` into `a`
    }
}
```

Listing 4.24 Message passing a type that does not implement the `Send` trait

```
#[task(spawn = [b])]
fn a(cx: a::Context) {
    let x = Rc::new(0);
    cx.spawn.b(x.clone()); // increase reference count
    drop(x); // decrease reference count
}

#[task]
fn b(cx: b::Context, input: Rc<i32>) {
    drop(input); // decrease reference count
}
```

4.4.5 `device`

The `device` module describes the capabilities of the target device though the `NVIC_PRIO_BITS` constant and the `Interrupt` enumeration.

Getting the `NVIC_PRIO_BITS` wrong does not break memory safety. Using a lower value results in under utilizing the priority levels that the devices has. Consider a device that physically has 3 priority bits but the

user specifies `NVIC_PRIO_BITS = 2`. This results in user defined priorities of `1` (192), `2` (128), `3` (64) and `4` (0) mapping to physical priorities of `2`, `4`, `6` and `8`.

Using a higher value results in contiguous user declared priorities mapping to the same physical priority. Consider a device that physically has 2 priority bits but the user specifies `NVIC_PRIO_BITS = 3`. This results in 2 contiguous user defined priorities, like `1` (224 / `0b1110_0000`) and `2` (192 / `0b1100_0000`), mapping to the same physical priority, like `1` (192 / `0b1100_0000`), because the lower bits of the byte are ignored.

Listing 4.25 Incorrect, memory unsafe implementation of the `Nr` trait

```
enum Interrupt { A, B, C }

unsafe impl Nr for Interrupt {
    fn nr(&self) -> u8 {
        match self {
            Interrupt::A => 2,
            Interrupt::B => 3,
            Interrupt::C => 2, // should be 4
        }
    }
}
```

Getting the `Nr` implementation of `Interrupt` wrong can break memory safety: if two interrupt names are specified as the same interrupt number this can cause interrupt priorities to be wrongly configured.

Consider a device that defines its `Interrupt` enumeration as shown in lst. 4.25. If an RTFM application uses interrupts `A` and `C` with priorities `1` and `2` the runtime code that runs before `#[init]` will set the priority of interrupt number `2` to priority `1` and then to `2` resulting in interrupt `A` having a priority of `2` instead of `1`.

4.5 Implementation

As part of this work Real Time For the Masses has been ported to different platforms: single-core ARM Cortex-M microcontrollers [22]; the LPC55S69, an homogeneous dual-core ARM Cortex-M microcontroller (2x Cortex-M33) [23]; the LPC54114, an heterogeneous dual-core ARM Cortex-M microcontroller (Cortex-M4 + Cortex-M0+) [24]; the Realtime Processing Unit (RPU) on the Zynq UltraScale+, a dual-core ARM Cortex-R processor [25]; the HiFive1, a single-core RISCV microcontroller [26]; and the Linux operating system [27][] (see [28] for a threaded implementation of rtfm-lang). This document will focus on the version v0.5.x of the implementation for the ARM Cortex-M architecture as it is the most complete implementation. All the other implementations have the exact same API and similar implementation details but are lacking some functionality in one way or another.

4.5.1 Base binary interface

The DSL relies on an underlying runtime crate (library) which performs the following tasks:

- Boot process
- Initialization of static variables
- Placement of the vector / interrupt table in memory

And provides a binary interface to the framework through the symbols `main` and an arbitrary number of interrupt handlers. The `main` symbol must have signature `fn() -> !`; all the interrupt handlers must have signature `fn()`. Lst. 4.26 shows the signatures.

The contract is that the runtime crate calls into `main` provided by the DSL *after* the boot process has been completed and all static variables have been initialized.

Listing 4.26 Symbols used to interface the runtime crate

```
#[no_mangle]
extern "C" fn main() -> ! { /* .. */ }

#[no_mangle]
extern "C" fn SomeInterruptHandler() { /* .. */ }
```

In the ARM Cortex-M port of RTFM this interface is provided by the `cortex-m-rt` crate [29], which is a crate widely used in the Rust ecosystem.

4.5.2 The ARM Cortex-M ISA

4.5.2.1 Thread mode

Cortex-M devices boot in *thread mode*. Within the RTFM task model this thread mode can be seen as a context with the lowest priority of `0`.

4.5.2.2 Interrupts

Interrupts are a mechanism commonly found in embedded devices that allows prioritization of parts of the program. An interrupt is comprised of two parts: an interrupt signal and an interrupt handler.

An interrupt signal is an event that usually occurs asynchronously to the execution of the program. Examples of signals are: an electrical pin changed its logical state from 0 to 1 or vice versa, a counter (timer) reached a certain value, data became available on some communication interface.

In response to these signals the processor executes an interrupt handler, a special subroutine. In the case of the ARM Cortex-M architecture all interrupts are higher priority than the thread mode so handlers will *preempt* code running in thread mode.

Preemption works as follows: the processor suspends the execution of the current subroutine, saves the state of that subroutine – processor registers – onto the stack and then jumps to the interrupt handler. The interrupt handler runs to completion and returns. Upon returning from the interrupt handler the state of the suspended routine is restored by popping the registers off the stack and the preempted routine is resumed.

In the Cortex-M architecture each interrupt signal is serviced by a different interrupt handler and up to 240 different interrupts are supported – the exact number depends on the implementation; most vendors only use a few dozens of interrupts. All these interrupts are managed by the NVIC peripheral. Apart from these interrupts the architecture defines up to 15 *exceptions*; these are interrupts not managed by the NVIC.

The Cortex-M architecture allows interrupt nesting: an interrupt can preempt another interrupt handler if the first has higher *priority*. All interrupts and most exceptions have configurable priorities.

An interrupt signal changes the state of an interrupt to *pending*. If the priority of the interrupt is high enough the interrupt is serviced: its pending state is cleared and the associated handler is executed. If the interrupt priority is equal or lower than the priority of the current context then the interrupt is kept in the pending state until the priority of the context being executed is low enough.

4.5.2.3 NVIC

Listing 4.27 NVIC API

```
1 impl NVIC {
2     pub fn unmask(interrupt: Interrupt) { /* .. */ }
3     pub unsafe fn set_priority(interrupt: Interrupt, nvic_priority: u8) { /* .. */ }
4     pub fn pend(interrupt: Interrupt) { /* .. */ }
5 }
```

The Nested Vector Interrupt Controller (NVIC) is a standardized interrupt controller found on all Cortex-M devices. The NVIC provides an interface to control and configure interrupts in the form of memory mapped registers.

The registers used in the Cortex-M port of RTFM are:

- ISER, Interrupt Set Enable Registers. These registers are used to individually enable interrupts.
- IPR, Interrupt Priority Registers. These registers are used to change the static priority of an interrupt.
- ISPR, Interrupt Set Pending Registers. These registers are used to change the pending state of a register.

Interaction with these registers is limited to the API shown in lst. 4.27. `NVIC::unmask` writes to an ISER register to enable the interrupt indicated in its argument. `NVIC::set_priority` writes to an IPR register to set the static priority of the interrupt indicated in the first argument to the value passed as the second argument. `NVIC::pend` writes to an ISPR register to change the state of the interrupt passed as first argument as pending.

It is important to note that NVIC priorities are encoded in the *higher* bits of a byte and that number of supported priority levels is device specific. Implementations support only a certain number of priority bits. tbl. 4.1 maps logical priorities to NVIC priorities for 2 and 3 priority bits:.

Table 4.1: Mapping between NVIC priorities and logical priorities for 2 and 3 priority bits

logical / bits	2	3
1	`0b1100_0000` / `192`	`0b1110_0000` / `224`
2	`0b1000_0000` / `128`	`0b1100_0000` / `192`
3	`0b0100_0000` / `64`	`0b1010_0000` / `160`
4	`0b0000_0000` / `0`	`0b1000_0000` / `128`
5	~	`0b0110_0000` / `96`
6	~	`0b0100_0000` / `64`
7	~	`0b0010_0000` / `32`
8	~	`0b0000_0000` / `0`

4.5.2.4 BASEPRI

Listing 4.28 BASEPRI API

```
impl BASEPRI {
    pub fn read() -> u8 { /* .. */ }
    pub unsafe fn write(nvic_priority: u8) { /* .. */ }
}
```

The Base Priority Mask Register (BASEPRI) is a special purpose register that controls the *dynamic* priority of a context. The thread mode has a static logical priority of `0` and interrupts have static logical priorities greater than `1`. It is the dynamic priority of a context what allows or disallows preemption. Preemption only occurs when the static priority of an interrupt is greater than the dynamic priority of the context currently being executed.

The dynamic priority is always equal or greater than the static priority. Writing a value to BASEPRI has no effect if the value would lower the dynamic priority below the static priority of the context. The BASEPRI register uses NVIC encoded priority values.

Interaction with this register will be limited to the API shown in lst. 4.28. `BASEPRI::read` returns the current value of the BASPRI register. `BASEPRI::write` writes the given argument into the BASEPRI register.

4.5.2.5 PRIMASK

Listing 4.29 PRIMASK API

```
impl PRIMASK {
    pub fn set() { /* .. */ }
    pub fn clear() { /* .. */ }
}
```

The Priority Mask Register (PRIMASK) is a special purpose register that can prevent the activation of all interrupts with configurable priority. This register only has two states: it can be active or not.

Interaction with this register will be limited to the API shown in lst. 4.29. `PRIMASK::set` sets the PRIMASK register, disabling all interrupts with configurable priorities. `PRIMASK::clear` clears the PRIMASK register, re-enabling the interrupts.

4.5.2.6 ARMv6-M

There are a few variants of the ARM Cortex-M architecture: ARMv6-M, ARMv7-M and ARMv8-M. The Cortex-M port of RTFM targets all of them. Each sub-architecture adds functionality to the previous iteration. The ARMv6-M has the less features of the three variants and supporting it has led to some implementation decisions.

The main differences between the ARMv6-M variant and the other two are:

- There is no BASEPRI register in this variant.
- There are no LDREX, STREX and CLREX instructions in this variant. These instructions are used to implement atomic CAS loops.

4.5.3 Hardware tasks

Listing 4.30 A minimal application

```
#[init]
fn init(cx: init::Context) { /* .. */ }

#[idle]
fn idle(cx: idle::Context) -> ! { /* .. */ }

#[task(binds = EXTI0)]
fn exti0(cx: exti0::Context) { /* .. */ }
```

Listing 4.31 Expansion of lst. 4.30

```
fn init(cx: init::Context) { /* .. */ }
fn idle(cx: idle::Context) -> ! { /* .. */ }
fn exti0(cx: exti0::Context) { /* .. */ }

#[no_mangle]
unsafe fn main() -> ! {
    PRIMASK::set();
    NVIC::set_priority(Interrupt::EXTI0, logical2nvic(1));
    NVIC::enable(Interrupt::EXTI0);
    init(/* .. */);
    PRIMASK::clear();
    idle(/* .. */)
}

#[no_mangle]
unsafe fn EXTI0() {
    exti0(/* .. */);
}
```

RTFM tasks are implemented using interrupts. The way the NVIC handles interrupts exactly matches how the RTFM task model prioritizes tasks. This fact is used to provide a highly efficient implementation: the RTFM runtime lets the NVIC, the hardware, do all the task scheduling so no bookkeeping needs to be done in software.

Lst. 4.30 is a minimal application that contains a hardware task, `exti0`, and the `idle` background task. Lst. 4.31 shows the code generated by the DSL (the expansion of the procedural macro). Lines 1-3 (lst. 4.31) contain the user code, which is kept as it is. The `main` function (lines 5-13) in the expansion is the entry point of the program; this function runs in *thread mode*. At the start of this function, the interrupts are disabled (line 7) and then the `EXTI0` interrupt is configured (lines 8-9). While the interrupts are disabled the user-defined `init` function is executed. After `init` returns, the interrupts are re-enabled and the user-defined `idle` function is executed.

The function `EXTI0` (lines 15-18) is an interrupt handler. This interrupt handler is used to dispatch the hardware task `exti0` (line 17). As all tasks are dispatched from interrupts disabling the interrupts ensures that no task will preempt `init`, as specified in the task model. As `idle` runs in thread mode all interrupts, and thus all tasks, can preempt it; this also matches the task model.

4.5.4 Scope control

Listing 4.32 A minimal application that uses resources

```
struct Resources {
    #[init(0)]
    x: u64,
}

#[task(binds = EXTI0, resources = [x])]
fn a(cx: a::Context) { /* .. */ }
```

Listing 4.33 Expansion of lst. 4.32

```
fn a(cx: a::Context) { /* .. */ }

mod a {
    pub struct Context<'a> { pub resources: Resources<'a> }
    pub struct Resources<'a> { pub X: &'a mut u64 }
}

const APP: () = {
    static mut X: u64 = 0;

    #[no_mangle]
    unsafe extern "C" fn EXTI0() {
        crate::a(a::Context { resources: a::Resources { x: &mut X } })
    }
};
```

Internally, RTFM makes extensive use of `static mut` variables. Static variables, both `static` and `static mut`, have global scoping that makes them accessible to all functions defined in the same scope. The DSL is careful about the placement of `static mut` variables so that they are never exposed to user code.

For the program shown in lst. 4.32, the DSL produces the code shown in lst. 4.33. Line 1 (lst. 4.33) corresponds to the user input, which is kept as it is; lines 3-6 are the public API that the user code can use; and lines 8-15 are implementation details. As task `a` is bound to interrupt `EXTI0` the corresponding interrupt handler (lines 11-14) calls into the user code (line 13).

The `static mut` variable behind the resource `x` is placed inside the `const APP` item (line 9) making it impossible to access from user code. References to this hidden `static mut` variable are passed back to the user code using a function call (lines 13).

4.5.5 Resource initialization

Listing 4.34 Minimal program that uses a late resource

```
#[init]
fn init(cx: init::Context) -> init::LateResources {
    init::LateResources { x: 0 }
}

#[task(binds = EXTI0, resources = [x])]
fn exti0(cx: exti0::Context) { /* .. */ }
```

Listing 4.35 Expansion of lst. 4.34

```
mod init {
    pub struct LateResources {
        pub x: u64,
    }
}

const APP: () = {
    static mut x: MaybeUninit<u64> = MaybeUninit::uninit();

    #[no_mangle]
    unsafe extern "C" fn main() -> ! {
        PRIMASK::set();
        let late = crate::init(init::Context { /* .. */ });
        x.as_mut_ptr().write(late.x);
        PRIMASK::clear();
        // ..
    }

    #[no_mangle]
    unsafe extern "C" fn EXTI0() {
        crate::exti0(exti0::Context {
            resources: exti0::Resources { x: &mut *x.as_mut_ptr() },
        });
    }
};
```

Resource initialization was discussed in the memory safety section (sec. 4.4.1); it was confirmed that semantically the user cannot use uninitialized resources. In this section, compiler optimizations and out of order execution are factored into the memory safety equation.

Consider the program in lst. 4.34 and the relevant parts of its expansion in lst. 4.35. In the expansion, the `main` function is the entry point of the program and `x` in line 8 (lst. 4.35) is a static variable with undefined initial value that holds the data of resource `x`. Interrupts are disabled (line 12) as soon as `main` starts executing and the user defined `init` function is executed while interrupts are disabled (line 13). The `LateResources` value returned by `init` is used to initialize the resource `x` in line 14. After late resources are initialized interrupts are re-enabled (line 15). After interrupts are re-enabled tasks like `exti0` can start; `exti0` always observes an initialized resource (line 22).

At the source code level initialization of late resources is done *before* re-enabling interrupts. This, however, may not be sufficient in the general case for the compiler can reorder memory operations as part of it optimizations and some architecture execute instructions and memory operations *out of order*.

To prevent the compiler from moving the initialization of `x` to *after* interrupts are re-enabled a *compiler*

Listing 4.36 Implementation of the `PRIMASK::clear` API

```
impl PRIMASK {
    pub fn clear() {
        unsafe { asm!("cpsid i" : : : "memory" : "volatile") }
    }
}
```

barrier is internally used in `PRIMASK::set` and `PRIMASK::clear`. Lst. 4.36 shows the implementation of `PRIMASK::clear`. The `"memory"` clobber in the `asm!` block acts like a fence that prevents the compiler from reordering memory operations across it.

To deal with the processor executing code out of order, which could result in `X` being initialized after interrupts are re-enabled a *memory barrier* instruction – `DMB` (Data Memory Barrier) instruction on ARM Cortex-M – may be needed in addition to the compiler barrier but this depends on the target architecture. In the specific case of the ARM Cortex-M architecture a memory barrier is *not* required in this case because the architecture only reorders memory operations and the ARM Cortex-M implementation guarantees that "all loads and stores always complete in program order, even if the first is buffered" [30].

4.5.6 `lock`

Listing 4.37 `lock` function

```
unsafe fn lock<T, R>(
    priority: &Cell<u8>, ceiling: u8, ptr: *mut T, f: impl FnOnce(&mut T) -> R,
) -> R {
    let current = priority.get();
    if current < CEILING {
        priority.set(CEILING);
        BASEPRI::write(logical2nvic(CEILING));
        let r = f(&mut X);
        BASEPRI::write(logical2nvic(current));
        priority.set(current);
        r
    } else {
        f(&mut X)
    }
}
```

Listing 4.38 Implementation of `BASEPRI::write`

```
impl BASEPRI {
    pub unsafe fn write(priority: u8) {
        asm!("msr basepri, $0" : : "r"(priority) : "memory" : "volatile");
    }
}
```

All resource proxies use the `lock` function shown in lst. 4.37 – on ARMv6-M the implementation uses `PRIMASK` instead of `BASEPRI`, because the latter is not available in its ISA. The `priority` argument is used to track the dynamic priority of the caller context, `ceiling` is the priority ceiling of the resource, `ptr` is a raw pointer into the static variable that holds the resource data and `f` is the closure to run within a critical section. A critical section is created by temporarily raising the dynamic priority of the caller context using the `BASEPRI` register (lines 7 and 9). The implementation tries to minimize `BASEPRI` writes by checking if the current priority is equal or greater than the priority ceiling; if that is the case then `BASEPRI` manipulation is omitted and the closure `f` is run directly (line 13).

Listing 4.39 Minimal application that contains a contended resource

```
1  #[idle]
2  fn idle(cx: idle::Context) -> ! {
3      // (I)
4      NVIC::pend(Interrupt::EXTI0);
5      // (V) ..
6  }
7
8  #[task(binds = EXTI0, priority = 1, resources = [x])]
9  fn exti0(cx: a::Context) {
10     // (II)
11     cx.resources.x.lock(|_| { /* (III) */ });
12     // (IV)
13 }
```

The implementation of the `BASEPRI::write` method which uses inline assembly is shown in lst. 4.38 and it is relevant to the memory safety of `lock`. The `"memory"` clobber is used to create a *compiler* barrier: it prevents memory operations *within* the closure `f` from being reordered to outside the critical section. The architectural requirements indicate that a *memory* barrier is *not* needed after interrupts are disabled by an `MSR` instruction [30].

4.5.6.1 Resource proxies

Consider the program in lst. 4.39 and its expansion in lst. 4.40. Lines 2-4 (lst. 4.40) show the resource proxy `x`; the proxy contains a `priority` field used to track the dynamic priority of the context that owns the proxy. This field makes the proxy *not* satisfy the `: 'static` which is required for memory safety as discussed in sec. 4.4.2.4.

Lines 11-14 show the implementation of the `lock` method for the proxy `x`. The implementation simply calls the `lock` *function* defined in lst. 4.37. The chosen priority ceiling (line 12) comes from the ceiling analysis performed by the DSL.

Lines 17-25 show the interrupt handler `EXTI0`. Hardware task `exti0` is bound to the `EXTI0` interrupt so the user defined handler is called from the `EXTI0` handler. The `priority` variable is used to track the dynamic priority of the context; all resource proxies owned by the task shared a reference (`&-`) to this variable. Interrupt handler `EXTI1` and task handler `exti1` have been omitted in the expansion.

4.5.6.2 BASEPRI invariant

The implementation of the `lock` method never reads the BASEPRI register; it only writes to it. This can result in the original value of the BASEPRI register being lost. It is important that the value of BASEPRI upon returning from an interrupt handler matches the value it had on entry. If this invariant is not preserved the static priority property of the RTFM task model would be broken compromising schedulability.

The schedulability issue can be observed in lst. 4.39 assuming the BASEPRI invariant is *not* upheld. At checkpoint (I) BASEPRI is `0` and the static and dynamic priorities are also `0`. The `pend` operation in line 4 causes task `exti0` to preempt idle. At checkpoint (II) BASEPRI is still `0` but the static and dynamic priorities become `1`. At checkpoint (III), inside the critical section, `BASEPRI` becomes `192` (assuming 3 priority bits); this raises the dynamic priority to `2`. At checkpoint (IV), `BASEPRI` becomes `224`; this lowers the dynamic priority back to `1`.

At checkpoint (V), after `exti0` returns, `BASEPRI` will have a value of `224` if the `BASEPRI` invariant is not preserved. This causes the dynamic priority of `idle` change from `0`, at checkpoint (I), to `1`. This dynamic priority is maintained for the rest of the execution of the program and prevents tasks that have a priority of `1` from ever running again.

Listing 4.40 Expansion of lst. 4.39

```
1  mod resources {
2      pub struct x<'a> {
3          priority: &'a Cell<u8>,
4      }
5  }
6
7  const APP: () = {
8      static mut X: u64 = 0;
9
10     impl resources::X<'_> {
11         fn lock<R>(&mut self, f: impl FnOnce(&mut u64) -> R) -> R {
12             const CEILING: u8 = 2;
13             lock(self.priority(), CEILING, &mut X, f)
14         }
15     }
16
17     #[no_mangle]
18     unsafe extern "C" fn EXTI0() {
19         let initial = BASEPRI::read();
20         let priority = Cell::new(1);
21         crate::exti0(exti0::Context {
22             resources: exti0::Resources { X: resources::X::new(&priority) },
23         });
24         BASEPRI::write(initial);
25     }
26 };
```

To prevent the issue the generated code, see lst. 4.40, includes code to read the `BASEPRI` register on interrupt entry (line 20) and restoring that original value before returning (line 26). A special case are tasks with a priority of `1`. On entry `BASEPRI` is always `0` (not active) so the `BASEPRI` read (line 20) is omitted and the write operation (line 26) always sets the `BASEPRI` register to `0`.

4.5.7 Message passing

Message passing consists of two parts: posting a message and dispatching the task associated to the message. The former is done by the user through the `spawn` API.

4.5.7.1 SPSC queue

Central to the `spawn` API is the Single-Producer Single-Consumer queue, whose implementation is shown in lst. 4.41. A previous iteration of this implementation was covered in [31] under the `RingBuffer` name.

The queue has a fixed capacity of `N` (line 4) and the `enqueue` and `dequeue` operations are non-blocking. The `dequeue` operation returns `None` if the queue is observed as empty at the moment it is invoked (line 13).

The `enqueue` operation is *unchecked*: it does not check if the queue is full before inserting a new item; it is unsound to insert new items in a queue that is full: it can result in destructors being run twice on the same value and cloning values that do not implement the `Clone` trait. For this reason the method is `unsafe` to call. The implementation of `spawn` and other APIs are structured so that calling this method does not break memory safety.

The queue must be used in single-producer single-consumer mode meaning that two `dequeue` operations should never happen concurrently (overlap in execution); likewise with the `enqueue` operation.

Listing 4.41 Single producer single consumer (SPSC) queue

```
pub struct Queue<T, const N: usize> {
    read: AtomicU8,
    write: AtomicU8,
    buffer: UnsafeCell<[MaybeUninit<T>; N]>,
}

impl<T, const N: usize> Queue<T, { N }> {
    pub fn dequeue(&self) -> Option<T> {
        let r = self.read.load(Ordering::Relaxed);
        let w = self.write.load(Ordering::Acquire); // ↓

        if r == w {
            None
        } else {
            let item =
                unsafe { (self.buffer.get() as *const T).add(r as usize % N).read() };
            self.read.store(r + 1, Ordering::Release); // ↑
            Some(item)
        }
    }

    pub unsafe fn enqueue_unchecked(&self, item: T) {
        let w = self.write.load(Ordering::Relaxed);
        (self.buffer.get() as *mut T).add(w as usize % N).write(item);
        self.write.store(w + 1, Ordering::Release); // ↑
    }
}
```

The use of the `Acquire` and `Release` orderings ensure that concurrent access results in an updated buffer being observed. Namely, the `Release` stores ensure that the preceding memory operation on the buffer has completed before the `read` / `write` pointer is updated. The `Acquire` load in `dequeue` synchronizes with the `Release` store in `enqueue` so that the `write` operation on the buffer has propagated across the core boundary before the `read` operation is performed.

4.5.7.2 `spawn`

Consider the example in lst. 4.42 and the relevant parts of its expansion in lst. 4.43. In the example, two tasks, `a` and `b`, running at different priorities can spawn a third task `c`, which takes a 64-bit integer as input. Because task `b` can preempt task `a` two instances of the `spawn.c()` invocation can overlap.

There are several pieces to the `spawn` API. The involved data structures are an `INPUTS` buffer (lst. 4.43 line 3) that holds the messages sent to a task, a *free queue* (line 4) that acts as a free list that keeps track of empty slots in the `INPUTS` buffer and a *ready queue* (line 5) that keeps track of software tasks ready to be dispatched.

Instead of a single ready queue, there is one ready queue per priority level to minimize the number of required `lock` operations. Each entry in the ready queue indicates which task is ready to be dispatched using an enumeration. There is one enumeration of tasks per priority level; line 1 shows the enumeration of tasks dispatched at priority `1`.

The `spawn` API is exposed as a method on a `Spawn` struct (lines 17-21). There is one such struct per task as to limit the tasks a task can spawn according to the user specification.

The process of posting a message (depicted in fig. 4.1) consists of these steps:

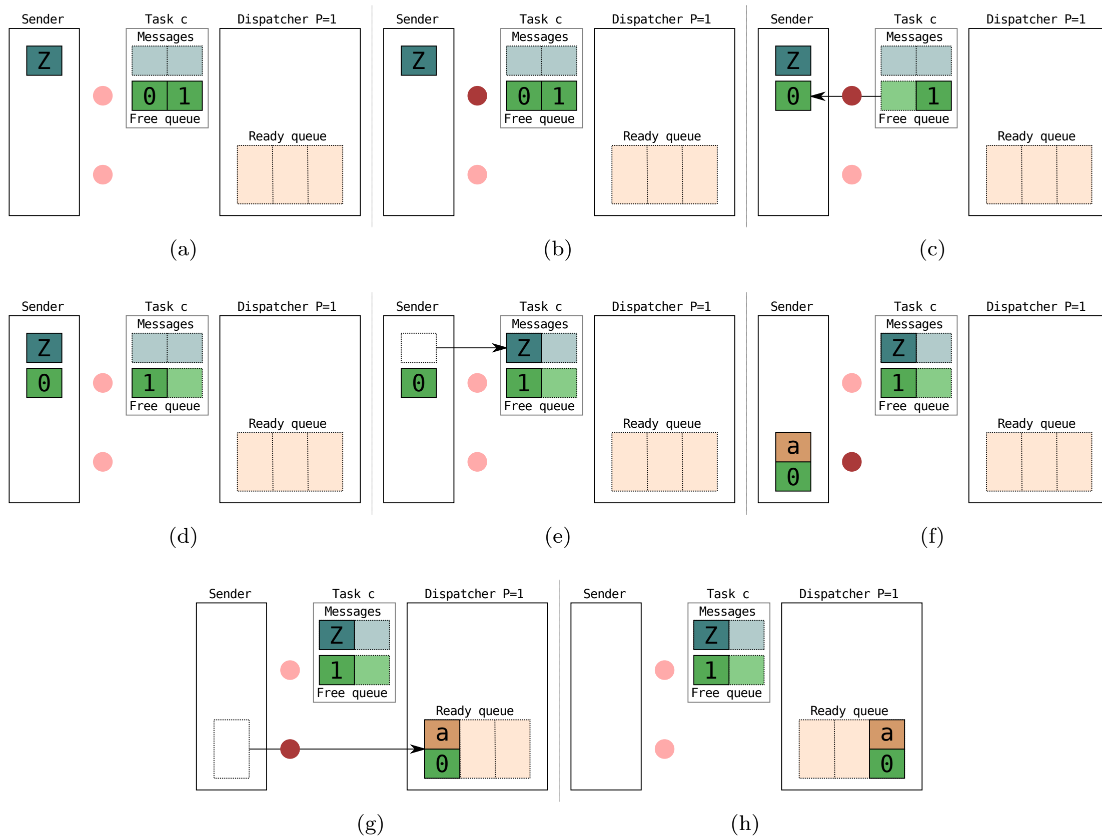


Figure 4.1: Steps for sending a message

Listing 4.42 Minimal application that uses message passing with and without payload

```
#[task(priority = 2, spawn = [c, d])]
fn a(cx: a::Context) {
    cx.spawn.c('Y');
}

#[task(priority = 3, spawn = [c])]
fn b(cx: b::Context) {
    cx.spawn.c('Z');
}

#[task(capacity = 2, priority = 1)]
fn c(cx: c::Context, input: char) { /* .. */ }

#[task(priority = 1)]
fn d(cx: d::Context) { /* .. */ }
```

1. Lst. 4.43 line 8: lock the task free queue (fig. 4.1b); get an empty slot in the `INPUTS` buffer (fig. 4.1c); release the lock (fig. 4.1d)
2. line 9: write the message payload into the empty slot (fig. 4.1e)
3. line 10: lock the ready queue (fig. 4.1f); enqueue the new message (fig. 4.1g); release the lock (fig. 4.1h)
4. line 11: send an interrupt signal to the interrupt handler that dispatches tasks

The locks are needed because two calls to `spawn_c` can overlap, as it may occur in the example (lines 3 and 8 of lst. 4.42). Writing into the `INPUTS` buffer does not require a lock because the `index` pointer indicates ownership over the free slot.

If the free queue is observed as empty in step 1 then the message buffer is full and the `spawn` invocation fails and the payload is returned back wrapped in an `Err` variant case (line 13).

Listing 4.43 Expansion of lst. 4.42

```
enum T1 { c, d }

static mut c_INPUTS: [MaybeUninit<char>; 2] = [MaybeUninit::uninit(); 2];
static mut c_FQ: Queue<u8, 2> = Queue::new();
static mut RQ1: Queue<(T1, u8), 3> = Queue::new();

unsafe fn spawn_c(priority: &Cell<u8>, payload: char) -> Result<(), char> {
    if let Some(index) = (c_FQ { priority }).lock(|fq| fq.dequeue()) {
        c_INPUTS.get_unchecked_mut(index as usize).write(payload);
        (c_RQ { priority }).lock(|rq| rq.enqueue_unchecked((T1::c, index)));
        NVIC::pend(Interrupt::EXTI0);
    } else {
        Err(payload)
    }
}

impl a::Spawn<'_> {
    pub fn c(&self, payload: char) -> Result<(), char> {
        spawn_c(self.priority(), payload)
    }
}
```

4.5.7.3 Task dispatcher

Listing 4.44 One of the task dispatchers of lst. 4.42

```
#[no_mangle]
unsafe fn EXTI0() {
    let priority = Cell::new(1);
    while let Some((task, index)) = RQ1.dequeue() {
        match task {
            T1::c => {
                let input = c_INPUTS.get_unchecked(index as usize).read();
                c_FQ.enqueue_unchecked(index);
                crate::c(c::Context { /* .. */}, input);
            }
            T1::d => { /* .. */ }
        }
    }
    BASEPRI::write(0); // BASEPRI invariant
}
```

Interrupts not used by the application are used by the runtime to dispatch software tasks. All software tasks that must be executed at the same priority level are dispatched from the same interrupt handler. The code in charge of dispatching tasks is called the task dispatcher. Each task dispatcher uses one of the many ready queues.

Lst. 4.44 shows one of the task dispatchers of lst. 4.42. This task dispatcher runs from interrupt handler `EXTI0` (lst. 4.44 line 2) and runs at a priority of `1` (line 3).

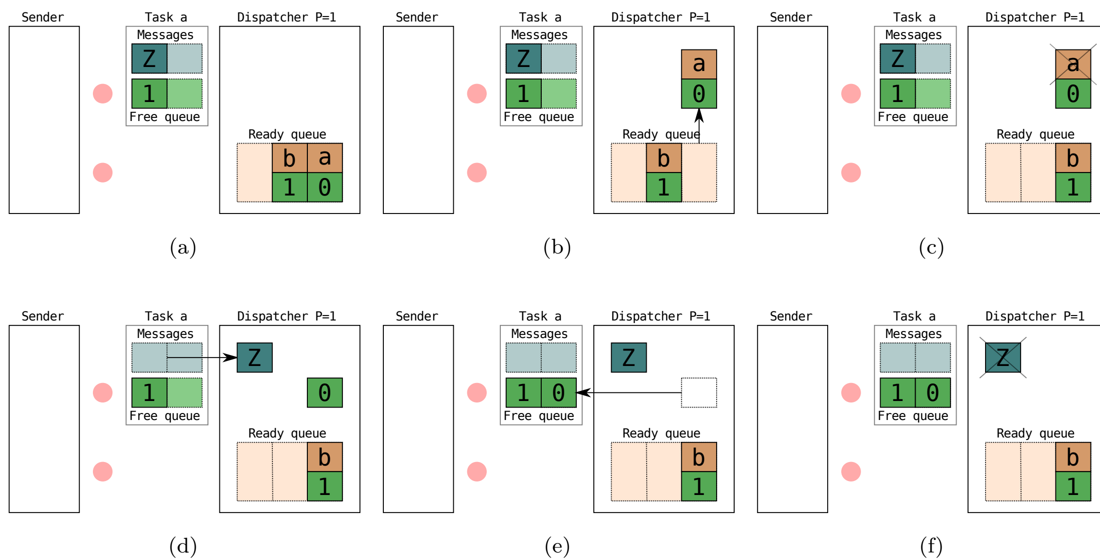


Figure 4.2: Steps for dispatching a message

The steps to dispatch a message (depicted in fig. 4.2) are:

1. Lst. 4.44 line 4: remove an entry from the ready queue (fig. 4.2b)
2. line 5 and 7: based on the task name indicated in the entry (fig. 4.2c) read out the message payload from the appropriate `INPUTS` buffer (fig. 4.2d) at the designated `index`.
3. line 8: return the `index` back to the free queue (fig. 4.2e)

4. line 9: Execute the corresponding user defined task handler. This action consumes the message payload (fig. 4.2f).

These steps are repeated until the ready queue is empty.

The task dispatcher does not need any `lock` because it is the only context that dequeues from its ready queue and enqueues empty slots back into the free queue of tasks.

4.5.7.4 Priority ceiling analysis

To compute the priority ceilings of the ready queues and free queues all the potential `spawn` invocations need to be taken into account. Each `spawn` call will access the free queue of the spawnee and the ready queue that corresponds to the priority level of the spawnee.

Using lst. 4.42 as an example for the priority ceiling analysis:

- Task `a`, which runs at priority 2, may access `c_FQ` and `RQ1` to spawn task `c` and may access `d_FQ` and `RQ1` to spawn task `d`
- Task `b`, which runs at priority 3, may access `c_FQ` and `RQ1` to spawn task `c`.

From this information the chosen priority ceilings are: `3` for `c_FQ`, `2` for `d_FQ` and `3` for `RQ1`.

4.5.7.5 Queue capacity

The capacity of each free queue comes from the task `capacity` declared in the specification. Where not declared the capacity is assumed to be `1`.

The capacity of each ready queue is selected to be the sum of the capacities of the tasks dispatched from it. This capacity is large enough to ensure that the unchecked `enqueue` operation in every `spawn` invocation never fails.

4.5.8 Timer queue

The RTFM runtime internally uses a timer queue to provide the `schedule` API. This section goes into the implementation details of the timer queue.

4.5.8.1 Binary heap

Listing 4.45 Binary Heap API

```
1  pub struct BinaryHeap<T, const N: usize> {
2      buffer: MaybeUninit<[T; N]>,
3      len: usize,
4  }
5
6  impl<T, const N: usize> BinaryHeap<T, { N }>
7  where T: Ord
8  {
9      pub fn push_unchecked(&mut self, item: T) { /* .. */ }
10     pub fn pop_unchecked(&mut self) -> T { /* .. */ }
11     pub fn peek(&self) -> Option<&T> { /* .. */ }
12     pub fn is_empty(&self) -> bool { /* .. */ }
13 }
```

The main data structure behind the timer queue is a binary heap with `O(log N)` `push` and `pop` operations. The `pop` operation returns the smallest item stored in the heap. This data structure is used as a priority queue where tasks are kept sorted by earliest scheduled time.

The runtime interacts with binary heaps using the API shown in lst. 4.45. `push_unchecked` and `pop_unchecked` (lines 9-10) are push and pop operations that do not check if the binary heap is full / empty before adding / removing an item; they are `unsafe` methods. The `peek` method (line 11) returns a shared reference (`&-`) to the task with earliest scheduled time if the heap is not empty; otherwise it returns `None`. The `is_empty` method (line 12) returns `true` if the heap is empty and `false` otherwise.

4.5.8.2 The system timer

Listing 4.46 System timer API

```rust
impl SYST {
    pub fn set_timeout(from: u32) { /* .. */ }
    pub fn pend() { /* .. */ }
}
```

The system timer, or SysTick, is a Cortex-M peripheral available on all Cortex-M devices. This 24-bit timer is clocked at the CPU frequency and can generate timeout interrupts by counting down to zero from a specified value.

Interaction with this peripheral is limited to the API shown in lst. 4.46. The `set_timeout` method (line 2) prepares a new timeout interrupt by starting a new countdown from the given argument. The `pend` method (line 3) sets the system timer interrupt as pending.

4.5.8.3 `TimerQueue`

Listing 4.47 `TimerQueue` abstraction

```rust
pub struct Task<T, M>
where M: Monotonic {
    name: T,
    index: u8,
    instant: M::Instant,
}

pub struct TimerQueue<T, M, const N: usize>
where M: Monotonic {
    tasks: BinaryHeap<Task<T, M>, { N }>,
}
```

The `TimerQueue` API used by the runtime is a thin abstraction over a binary heap; its API is shown in lst. 4.47. The items in the heap are scheduled `Tasks` (line 1-8) that are kept sorted according to the user defined `Instant` they were scheduled to run at.

The methods of this `TimerQueue` abstraction are shown in lst. 4.48. The `enqueue_unchecked` operation (lines 2-7) adds a new task into the underlying binary heap but also sets the `SysTick` exception as pending if the new task was scheduled to run before any of the tasks already in the heap or the heap was empty.

The `dequeue_ready` operation (lines 9-26) only removes the top task *if* its time to run has come (line 12). If none of the tasks is ready to run this operation sets a new timeout (line 22). The timeout (line 18) is computed by subtracting the current time (line 11) from the `instant` the earliest task was scheduled to run at (line 10) and then scaling using the ratio (line 20) indicated by the monotonic timer chosen by the user. The timeout is limited to `MAX_TIMEOUT` (line 16) which is the maximum timeout the system timer supports.

4.5.8.4 `schedule`

Consider the example in lst. 4.49 and the relevant parts of its expansion in lst. 4.50.

Listing 4.48 `TimerQueue` methods

```
impl<T, M, const N: usize> TimerQueue<T, M, { N }> {
    pub fn enqueue_unchecked(&mut self, task: Task<T, M>) {
        if self.tasks.peek().map(|earliest| &task < earliest).unwrap_or(true) {
            SYST::pend()
        }
        self.tasks.enqueue_unchecked(task);
    }

    pub fn dequeue_ready(&mut self) -> Option<(T, u8, bool)> {
        if let Some(instant) = self.tasks.peek().map(|earliest| earliest.instant) {
            let now = M::now();
            if instant < now {
                let task = self.tasks.pop_unchecked();
                return Some((task.name, task.index, !self.tasks.is_empty()));
            } else {
                const MAX_TIMEOUT: u32 = (1 << 24) - 1;
                let r = M::ratio();
                let timeout = (now - instant)
                    .try_into()
                    .map(|diff| cmp::min(MAX_TIMEOUT, diff * r.numerator / r.denominator))
                    .unwrap_or_else(MAX_TIMEOUT);
                SYST::set_timeout(timeout);
            }
        }
        None
    }
}
```

The data structures involved in the implementation of the `schedule` API are `INPUTS` buffers (lst. 4.50 line 3), free queues (line 5) and ready queues (line 6) as seen in the `spawn` API. In addition there are `INSTANTS` buffers (line 4) for each task, these that hold the scheduled time of each message, and a single `TimerQueue` instance (line 7) that holds all tasks that have been scheduled but have not yet become ready.

The process of scheduling a task (lines 9-22), depicted in fig. 4.3, consists of these steps:

1. Lst. 4.50 line 12: lock the task free queue (fig. 4.3b); get an empty slot in the `INPUTS` buffer (fig. 4.3c); release the lock (fig. 4.3d).
2. line 13: write the message payload into the empty slot (fig. 4.3e)
3. line 14: write the scheduled time into the `INSTANTS` buffer at the same index as the payload (not shown in fig. 4.3)
4. line 15-17: Lock the timer queue (fig. 4.3f); insert the task into it (fig. 4.3g); release the lock (fig. 4.3h)

The last step may trigger or schedule a run of the system timer handler.

Listing 4.49 Minimal application that uses the `schedule` API

```
#[task(priority = 2, schedule = [c])]
fn a(cx: a::Context) {
    cx.schedule.c(some_instant, 'Y');
}

#[task(priority = 3, schedule = [c])]
fn b(cx: b::Context) {
    cx.schedule.c(some_instant, 'Z');
}

#[task(priority = 1, capacity = 2, schedule = [a])]
fn c(cx: c::Context, input: char) { /* .. */ }
```

Listing 4.50 Expansion of lst. 4.49

```
enum T { a, c }

static mut c_INPUTS: [MaybeUninit<char>; 2] = [MaybeUninit::uninit(); 2];
static mut c_INSTANTS: [MaybeUninit<Instant>; 2] = [MaybeUninit::uninit(); 2];
static mut c_FQ: Queue<u8, 2> = Queue::new();
static mut RQ1: Queue<(T1, u8), 2> = Queue::new();
static mut TQ: TimerQueue<T, Monotonic, 2> = TimerQueue::new();

unsafe fn schedule_c(
    priority: &Cell<u8>, instant: Instant, payload: char,
) -> Result<(), char>{
    if let Some(index) = (c_FQ { priority }).lock(|fq| fq.dequeue()) {
        c_INPUTS.get_unchecked_mut(index as usize).write(payload);
        c_INSTANTS.get_unchecked_mut(index as usize).write(instant);
        (TQ { priority }).lock(|tq| {
            tq.enqueue_unchecked(Task { name: T::c, index, instant });
        });
        Ok(())
    } else {
        Err(payload)
    }
}
```

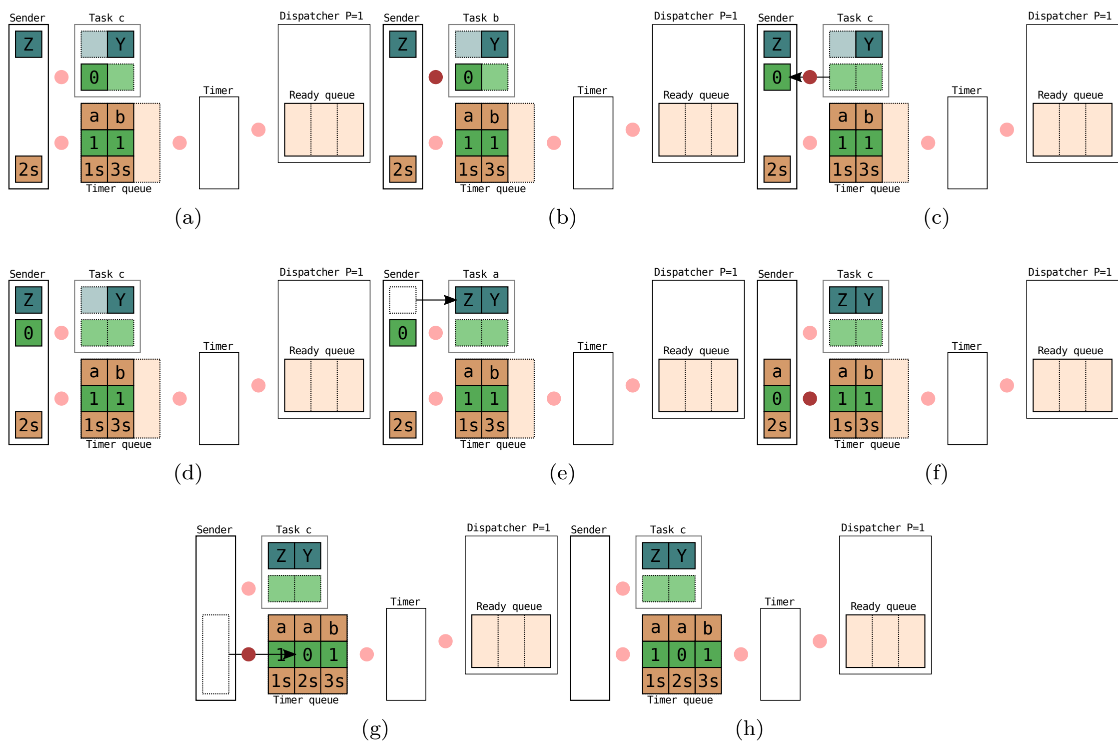


Figure 4.3: Steps for scheduling a task

4.5.8.5 Handler

Listing 4.51 System timer handler

```
#[no_mangle]
unsafe extern "C" fn SysTick() {
    let priority = &Cell::new(2);
    while let Some((task, index)) = (TQ { priority }).lock(|tq| tq.dequeue_ready()) {
        match task {
            T::a => {
                (RQ1 { priority }).lock(|rq| rq.enqueue((task, index)));
                NVIC::pend(Interrupt::EXTI0);
            }
            T::c => { /* similar to the other branch */ }
        }
    }
}
```

The timer queue handler, shown in lst. 4.51, forwards messages ready to be dispatched into the corresponding ready queue. The process, depicted in fig. 4.4, consists of these steps:

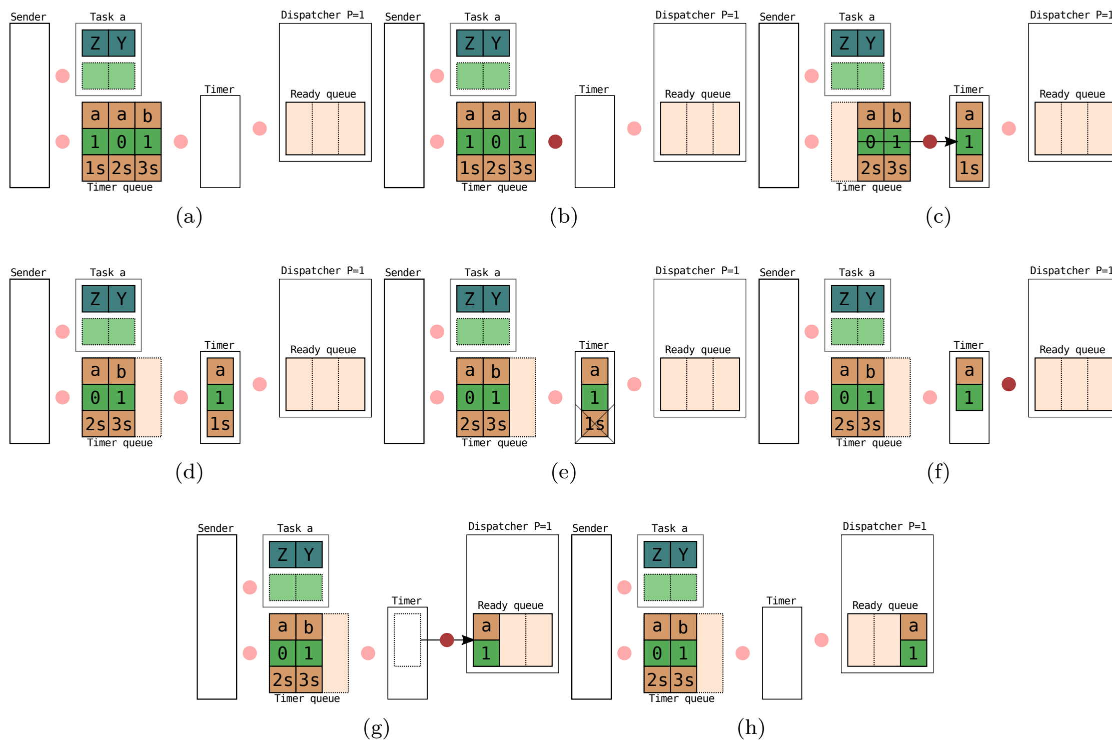


Figure 4.4: Forwarding a ready task from the timer queue to the ready queue

1. Lst. 4.51 line 4: lock the timer queue (fig. 4.4b); take the entry at the top, which corresponds to the task with nearest scheduled time (fig. 4.4c); release the lock (fig. 4.4d).
2. line 5: based on the task name in the entry the corresponding ready queue is selected. At this point the scheduled time that was in the entry is discarded (fig. 4.4e).
3. line 7: lock the ready queue (fig. 4.4f); insert the task name-slot index pair into the ready queue (fig. 4.4g); release the lock (fig. 4.4h).

4. line 8: signal the corresponding task dispatcher to run.

These steps are repeated until no *ready* task is left in the timer queue – the queue may not necessarily be empty at this point. `dequeue_ready` will set the timeout for the next pending message if there are no more ready tasks in the queue.

4.5.8.5.1 Priority The priority of the timer queue handler cannot be configured by the user; it is selected by the framework. The priority of this handler is selected to match the highest priority among all the `schedule`-able tasks. The rationale for this is avoiding unintended priority inversion.

To see why this is the case consider this scenario: tasks A and B with priorities 2 and 3 can be scheduled. If A is scheduled to run slightly before B and the priority of the timer queue handler is chosen to be 2 then task A will be forwarded and dispatched first; the higher priority task, B, will not be forwarded to the task dispatcher until after the lower priority task A returns.

If instead a priority of 3 is chosen then when task B becomes ready the handler will preempt task A and forward the task B to the dispatcher; as task B's dispatcher has higher priority it will preempt task A and execute task B to completion.

Choosing a higher priority does not improve scheduling of tasks. Consider the scenario where two tasks, A and B, with the same priority are scheduled to run at around the same time. Setting the priority of the timer queue handler higher than those tasks will cause the handler to preempt task A and forward task B to its dispatcher but since tasks A and B have the same priorities task B will only run after task A finishes. Lowering the priority of the timer queue handler to match the priorities of tasks A and B produces the same result except that the timer queue handler runs between tasks A and B.

4.5.8.6 Task dispatcher

When the `schedule` API is used task dispatchers and task `Contexts` are updated to include the scheduled time, as shown in lst. 4.52. After reading the message payload (line 7) and before returning the `index` back into the free queue (line 9), the `scheduled` time is read from the `INSTANTS` buffer (line 8). This `scheduled` time is passed to the user defined task handler as part of the task `Context` structure (line 10).

Listing 4.52 One of the task dispatchers of lst. 4.49

```
#[no_mangle]
unsafe extern "C" fn EXTI0() {
    let priority = &Cell::new(1);
    while let Some((task, index)) = RQ1.dequeue() {
        match task {
            T1::c => {
                let input = c_INPUTS.get_unchecked(index as usize).read();
                let scheduled = c_INSTANTS.get_unchecked(index as usize).read();
                c_FQ.enqueue_unchecked(index);
                crate::c(c::Context { scheduled /*..*/ }, input);
            }
        }
    }
}
```

4.5.8.7 Priority ceiling analysis

To compute the priority ceiling of the timer queue all the potential `schedule` invocations need to be considered. All these invocations may access the timer queue. The timer queue is also accessed from the system timer handler whose priority was discussed in sec. 4.5.8.5.1.

The analysis from sec. 4.5.7.4 needs to be updated to include `schedule` operations. All operations access the free queue of the target task.

Using lst. 4.49 as an example for the priority ceiling analysis:

- Tasks `a`, which runs at priority `2`, and `c`, which runs at priority `1`, are `schedule`-able tasks. The highest priority (`2`) is chosen to be the priority of the system timer handler `SysTick`.
- Task `a`, which runs at priority `2`, may access `c_FQ` and `TQ` to `schedule` task `c`.
- Task `b`, which runs at priority `3`, may access `c_FQ` and `TQ` to `schedule` task `c`.
- Task `c`, which runs at priority `1`, may access `a_FQ` and `TQ` to `schedule` task `a`.

From this information the chosen priority ceilings are: `3` for `TQ`, `3` for `c_FQ` and `1` for `a_FQ`.

4.6 WCET

In this section WCET analysis is performed on all the single-core API. All the measurements were done on a STM32F103 microcontroller, a device with a single ARM Cortex-M3 core, running code off Flash with zero Flash wait cycles and at a frequency of 8 MHz.

4.6.1 `lock`

Listing 4.53 Nested lock for WCET analysis

```
#[task(priority = 1, resources = [x, y])]
fn a(cx: a::Context) {
    let mut x = cx.resources.x; // ceiling = 2
    let mut y = cx.resources.y; // ceiling = 3

    asm::bkpt();
    x.lock(|_| {
        asm::nop();
        y.lock(|_| asm::wfi());
        asm::nop();
    });
    asm::bkpt();
}
```

Listing 4.54 Optimized machine code of lst. 4.53

```
8000152:        BKPT    #0
8000154:        msr     basepri, r12
8000158:        NOP
800015a:        WFI
800015c:        NOP
800015e:        msr     basepri, lr
8000162:        BKPT    #0
```

Thanks to the FIFO nature of the `lock` API and the whole-program compile-time information about task priorities and priority ceilings the compiler is able to optimize away the branching within the implementation of `lock` (sec. 4.5.6).

Consider the example in lst. 4.53. The example depicts two nested `lock` calls. The priority ceilings of resources `x` and `y` are 2 and 3 respectively. The first `lock` call on `y` increases the dynamic priority level to 3; the second `lock` call does not need to increase the priority level because it is high enough to safely access `x`.

The optimized machine code for this example is shown in lst. 4.53. Note that the second `lock` call compiles down to nothing: there are no instructions between `WFI` and the surrounding `NOP`s. The first `lock` call produces two writes to the BASEPRI register to create the critical section. In the worst-case it takes 2 instructions to enter and leave the critical section created by the `lock` API.

4.6.1.1 vs Spinlock

To have a reference point the overhead and properties of a spinlock are studied in this section. The implementation of the spinlock is shown in lst. 4.55. The `lock` method uses an atomic boolean to claim the spinlock. If the lock has already been taken it busy waits until it is released; if it is free then it changes the state of the boolean to indicate that it currently holds the lock. While the lock is held the closure is executed and then the lock is released by reverting the state of the atomic boolean.

To measure the overhead of entering and leaving the critical section created by the spin lock consider the program in lst. 4.56, which compiles down to the machine code shown in lst. 4.57. In the *best* case the

Listing 4.55 A spinlock

```
struct Spinlock<T> {
    lock: AtomicBool,
    data: UnsafeCell<T>,
}

impl<T> Spinlock<T> {
    fn lock<R>(&self, f: impl FnOnce(&mut T) -> R) -> R {
        while self.lock.compare_and_swap(false, true, Ordering::Relaxed) != false {}
        let r = f(unsafe { &mut *self.data.get() });
        self.lock.store(false, Ordering::Relaxed);
        r
    }
}
```

Listing 4.56 Program to measure the overhead of `Spinlock.lock`

```
static X: Spinlock<i64> = Spinlock::new(0);

fn main() {
    asm::bkpt();
    X.lock(|x| asm::nop());
    asm::bkpt();
}
```

overhead of the spinlock is 15 cycles. The best case is when the spinlock succeeds on its first try.

Unlike the resource abstraction the spinlock abstraction can deadlock because it does not prevent other tasks from starting. The example in lst. 4.58 demonstrates a deadlock. Assume that task `a` runs first; it claims the spinlock and within the critical section it spawns task `b`. Task `b` preempts task `a` because it has higher priority; when it tries to claim the spinlock it observes that the lock is taken so it busy waits. This is a deadlock because task `a` needs to resume for the lock to be released but that cannot occur in the RTFM model because there is no context switching from a high priority task to a lower priority task.

Listing 4.57 Optimized machine code of lst. 4.56

```
8000404:       BKPT    0x0000
8000406:       adds    r1, r0, #4
8000408:       b.n     800040e
800040a:       clrex
800040e:       ldrexb  r3, [r1]
8000412:       cmp     r3, #0
8000414:       bne.n   800040a
8000416:       strexb  r3, r2, [r1]
800041a:       cmp     r3, #0
800041c:       bne.n   800040e
800041e:       movs    r1, #0
8000420:       NOP
8000422:       strb    r1, [r0, #4]
8000424:       BKPT    0x0000
```

Listing 4.58 Deadlocking a spinlock

```
static X: Spinlock<i64> = Spinlock::new(0);

#[task(priority = 1, spawn = [b])]
fn a(cx: a::Context) {
    X.lock(|_| cx.spawn.b());
}

#[task(priority = 2)]
fn b(cx: b::Context) {
    X.lock(|_| { /* .. */ });
}
```

4.6.2 spawn

Listing 4.59 Program used to measure the overhead of `spawn`

```
#[task(spawn = [b])]
fn a(cx: a::Context) {
    asm::bkpt();
    cx.spawn.a([0; N]);
    asm::bkpt();
}

#[task]
fn b(cx: b::Context, input: [u32; N]) {}
```

A `spawn` invocation consists of these operations:

- `lock` on the free queue. `O(1)` overhead
- `write` on `INPUTS` buffer. `O(sizeof(T))` overhead (`memcpy`)
- `write` on `INSTANTS` buffer. `O(sizeof(Instant))` overhead (`memcpy`)
- `lock` on the ready queue. `O(1)` overhead
- `NVIC::pend`. `O(1)` overhead

Neither of these operations depends on the state of the queues: the execution time is the same regardless of how full they are. The size of both `T`, the type of the task input, and the user-defined `Instant` are known at compile time and constant so all `spawn` calls run in constant time.

The following table shows the typical overhead for a `spawn` call for different sized payloads. The measurement was done on the program shown in lst. 4.59 by reading the `CYCCNT` cycle counter, a monotonic timer, at lines 3 and 5 and subtracting the values.

Table 4.2: WCET overhead of `spawn` for different payload sizes

Payload size (bytes)	Overhead (cycles)
0	32
4	33
8	36
16	48
32	70

4.6.3 Task dispatcher

In this section the overhead of dispatching a software task is measured.

4.6.3.1 match

A `match` operation appears in the code of the task dispatcher. This section characterizes the overhead of a match operation using the program shown in lst. 4.60 for a different number of `Task` variants. The overhead was measured by reading the `CYCCNT` cycle counter at line 4 and at any of the other breakpoints and then subtracting the two values.

The execution time was found to be linear for a small number of tasks but starting at 5 tasks the compiler built a dispatch table making the execution time constant regardless of the number of tasks. The results are shown in tbl. 4.3. The machine code for 3 and 5 variants can be found in lst. 4.61 and lst. 4.62 respectively.

Listing 4.60 Program used to measure the overhead of a `match` operation

```
enum Task { A, B, /* .. */ }

fn a(task: Task) {
    asm::bkpt(0);
    match task {
        Task::A => asm::bkpt(1),
        Task::B => asm::bkpt(2),
        // ..
    }
}
```

Table 4.3: WCET of `match` for different number of `Task` variants

# tasks	WCET
2	2
3	4
4	6
>5	9

Listing 4.61 Optimized Machine code of lst. 4.60 when `Task` has 3 variants

```
8000132:        cmp     r0, #1
8000134:        BKPT    0x0000
8000136:        beq.n   8000140
8000138:        cmp     r0, #2
800013a:        bne.n   8000144
800013c:        BKPT    0x0003
800013e:        bx      lr
8000140:        BKPT    0x0002
8000142:        bx      lr
8000144:        BKPT    0x0001
8000146:        bx      lr
```

Listing 4.62 Optimized Machine code of lst. 4.60 when `Task` has 5 variants

```
8000132:        BKPT    0x0000
8000134:        subs    r0, #1
8000136:        cmp     r0, #3
8000138:        bhi.n   8000146
800013a:        tbb     [pc, r0]
800013e:        .short  0x0602
8000140:        .short  0x0a08
8000142:        BKPT    0x0002
8000144:        bx      lr
8000146:        BKPT    0x0001
8000148:        bx      lr
800014a:        BKPT    0x0003
800014c:        bx      lr
800014e:        BKPT    0x0004
8000150:        bx      lr
8000152:        BKPT    0x0005
8000154:        bx      lr
```

4.6.3.2 Overhead

Listing 4.63 Program used to measure the overhead of task dispatching

```
#[task(priority = 2)]
fn a(cx: a::Context, input: [u32; N]) {
    asm::bkpt();
}
```

The operations performed by the task dispatcher to invoke a software tasks are the following:

- `dequeue` operation on the ready queue. `O(1)` overhead
- `match` on the task name. `O(1)` overhead
- `read` the `INPUTS` buffer. `O(sizeof(T))` overhead
- `read` the `INSTANTS` buffer. `O(sizeof(Instant))` overhead
- `enqueue` operation on the free queue. `O(1)` overhead

The overall run time does not depend on the state of the queue. The sizes of the the message payload `T` and the user-defined `Instant` are known at compile time and fixed so the whole process completes in constant time.

The program in lst. 4.63 was used to measure the overhead of dispatching a software task. The overhead was measured as the number of CPU cycles it took to go from the the interrupt handler to the breakpoint instruction (line 3). tbl. 4.4 presents the overhead for different payloads sizes.

Table 4.4: Task dispatcher worst-case overhead for different payload size

Payload (bytes)	Overhead (cycles)
0	36
4	37
8	41
16	45
32	49

The overhead of stacking registers to enter and leave the interrupt needs to be added to this number to obtain the response time of spawning a task. The ARM Cortex-M architecture has two optimizations around interrupt handling: tail chaining and lazy stacking of floating pointer (FP) registers. Tail chaining occurs when the core leaves an interrupt and immediately enters a same or lower priority interrupt; registers are not popped and pushed in that scenario, lowering overhead. FP registers are only pushed onto the stack if they were being used at the time the interrupt was taken. tbl. 4.5 presents the overhead of interrupt entry / exit with and without optimizations [32].

Table 4.5: Overhead of entering and leaving an interrupt

Tail-chaining / FP registers	no	yes
no	12	29
yes	8	25

4.6.4 `schedule`

Listing 4.64 Program used to measure the WCET of `schedule`

```
#[task(schedule = [b])]
fn a(cx: a::Context) {
    asm::bkpt();
    cx.schedule.b(cx.scheduled, [0; N]);
    asm::bkpt();
}

#[task]
fn b(cx: b::Context) { /* .. */ }
```

The operations performed in an invocation of the `schedule` API are:

- `lock` on the free queue. `O(1)` overhead
- `dequeue` operation on the free queue. `O(1)` overhead
- `write` on the `INPUTS` buffer. `O(sizeof(T))` overhead
- `write` on the `INSTANTS` buffer. `O(sizeof(Instant))` overhead
- `lock` on the timer queue. `O(1)` overhead
- `enqueue` operation on the timer queue. `O(log2(n))` overhead where `n` is the *current* size of the queue.

The overall process does not complete in constant time because it depends on the size of the queue, which changes at runtime. However, it is possible to set an upper bound because the capacity of the queue is known at compile time and the runtime size of the queue can never exceed the capacity. Therefore the WCET for the operation is `O(sizeof(Instant) + sizeof(T) + log2(N - 1))` where `N` is the capacity of the timer queue.

tbl. 4.6 presents the execution time of `schedule` for different sized payloads, under the scenario that the timer queue is one element short of being full.

Table 4.6: WCET of `schedule` for different payload sizes and timer queue capacities

Payload size (bytes) / Capacity	1	2	4	8	16	32
0	53	105	128	143	165	185
4	59	107	130	145	170	190
8	70	113	136	152	174	196
16	80	121	146	161	183	205
32	96	141	165	180	202	225

4.6.5 `SysTick`

In this section the overhead of forwarding a task from the timer queue to the ready queue is characterized. The overhead consists of running `SysTick` to completion; the operations executed in `SysTick` are:

- `lock` on the timer queue. `O(1)` overhead
- `dequeue_ready` operation on the timer queue. `O(log2(n))` overhead where `n` is the *current* size of the queue, assuming that the calls to the `Monotonic` timer are constant time (they usually are).
- `match` on the task name. `O(1)` overhead
- `lock` on the ready queue. `O(1)` overhead
- `enqueue` operation on the ready queue. `O(1)` overhead
- The first two operations are repeated but `dequeue_ready` returns `None`.

The execution time is not constant time because it depends on the size of the timer queue. However, the execution can be bounded because the capacity of the timer queue is known at compile time and the size of

the queue cannot exceed its capacity. The worst case occurs when the queue is full; the execution time in that case is `O(log2(N))` where `N` is the capacity of the timer queue.

tbl. 4.7 lists the WCET for different timer queue capacities.

Table 4.7: System handler overhead for different timer queue capacities

Capacity	WCET
1	95
2	126
4	165
8	191
16	221
32	247

4.7 Alternative implementations

This section discusses alternative message passing implementations and the trade offs that were considered to not opt for these alternatives.

4.7.1 Multi-producer multi-consumer (MPMC) queue

There are fixed capacity lock-free MPMC queue implementations [33] that could have been used for the ready queues instead of the chosen SPSC implementation. The advantage of an MPMC queue would have been the removal of the `lock`s around the enqueue operations on the ready queue.

However, MPMC queues come with several disadvantages.

These queues are implemented using CAS loops which cannot be lowered to machine code when targeting the ARMv6-M architecture. This would have meant either not supporting the ARMv6-M architecture, making RTFM less portable, or using a different implementation depending on the target architecture, increasing the required implementation and maintenance work.

The other disadvantage is that the enqueue and dequeue operations on these queues are *not* constant time but instead contain a `loop` that is retried when the operation is *interrupted* – regardless of whether the interrupt handler tries to use the queue or not. This makes WCET analysis more complicated as the existence of higher priority interrupts needs to be factored into the analysis.

Finally, even the best case (single iteration) execution time of a enqueue (dequeue) operation on an MPMC queue takes at least twice the execution time of a `lock` plus a enqueue (dequeue) operation on an SPSC queue.

4.7.2 Lock-free memory pool

A lock-free memory pool could have been used instead of the free queues. The advantage of using a pool is that a `lock` would not be required to get a slot to write the payload into.

A lock-free memory pool is implemented using a CAS loop so this alternative implementation runs into similar problems as the MPMC queue.

4.8 Users

RTFM has been used both by hobbyists fig. 4.5 and in production environments fig. 4.6. Statistics on crates.io [34] show around 2,000 monthly downloads over the past few months.

(a) rusty-clock

(b) keyberon

Figure 4.5: Hobby projects

(a) Grepit AB

(b) WideFind AB

Figure 4.6: Production users

Among the most notorious hobby projects using RTFM we have `rusty-clock` [35], an alarm clock that also displays pressure, temperature and humidity; and `keyberon` [36], a USB mechanical keyboard. Among the production users we have Grepit AB (https://grepit.se/) and WideFind AB (https://www.widefind.se/).

Chapter 5

Multi-core extension

As covered in sec. 2.3 the approach to support multi-core devices in RTFM is to treat each core as if it was running its own, separate RTFM application, that is each core will have its own set of tasks and resources and these sets will not be allowed to overlap.

With this approach each application will retain all the real time and memory safety properties that were explored in sec. 4. Multi-core applications need some form of inter-core communication; the model allows message passing between cores to fulfill this need. Cross-core message passing works in the same way as single-core message passing: one core can spawn a task on a different core and pass a message along; the message will become the input of the task.

As most of the single-core functionality is present in the multi-core version the overall memory safety and real-time suitability of the multi-core version depends on the implementation of the cross-core message passing functionality.

API wise the multi-core version of RTFM was kept as close as possible in syntax to the single-core version. This eases the transition path of a user of the single-core version to the multi-core version; it also reduces the amount of documentation that needs to be written for the latter.

The multi-core API is the single-core API with one additional requirement, one additional restriction and some API extensions. The additional requirement is that each tasks must be assigned to a core. The additional restriction is that resources can only be shared between tasks that run on the same core. The API extensions will be covered in sec. 5.2.

5.1 Overview example

It takes few changes to transform the single core network example from sec. 4.1 into a dual-core application.

The dual-core version of lst. 4.1 is shown in lst. 5.1. In this version the work has been divided between the cores. The second core takes care of reading packets from the radio interface in the `on_new_packet` task and the first core processes those packets in the `process_packet` task.

The advantage of this division of work is that now both tasks can run in parallel, increasing throughput.

5.2 API

The multi-core API is the single-core API plus a few additions to the existing attributes in the form of new arguments.

Listing 5.1 dual core version of lst. 4.1

```
#[rtfm::app(cores = 2, /* .. */)]
const APP: () = {
    struct Resources { radio: Radio }

    #[init(core = 0)]
    fn init(cx: init::Context) -> init::LateResources {
        // ..
        init::LateResources { RADIO: radio }
    }

    #[task(core = 1, binds = EXTI0, resources = [radio], spawn = [process_packet])]
    fn on_new_packet(cx: on_new_packet::Context) {
        let new_packet = cx.resources.radio.next_packet();
        if let Some(buffer) = P::alloc() {
            let packet = new_packet.read_into(buffer);
            cx.spawn.process_packet(packet);
        } else {
            new_packet.discard();
        }
    }

    #[task(core = 0, capacity = 4)]
    fn process_packet(cx: process_packet::Context, packet: Box<P>) { /* .. */ }
};
```

5.2.1 `cores`

The `cores` argument in the `#[app]` attribute (line 1 of the overview example) is used to indicate the DSL that the application is a multi-core application. `cores` takes an integer equal or greater than 2. Omitting the `cores` argument indicates that the application is a single-core application.

5.2.2 `core`

In multi-core mode the `#[task]`, `#[init]` and `#[idle]` attributes gain a new, mandatory `core` argument. This argument takes an integer in the half-open range `[0, {cores})` that indicates on which core the function will run.

Resources do not need a `core` annotation because resource ownership is inferred from the `resources` lists. In the overview example, the `radio` resource is owned by the second core even though it is initialized by the first core.

5.2.3 `late`

Initialization of late resources is more flexible in multi-core mode. In the overview example the first core initialized a resource owned by the second core but it is possible to split the initialization of late resources across many cores. To disambiguate which core initializes which resources the `late` argument may be used. This argument is part of the `#[init]` attribute and takes a list of resources that the `init` function will initialize.

It is not necessary to use the `late` on all the `#[init]` attributes. The DSL assumes that the non `late` annotated `#[init]` initializes the remaining resources. An example of this kind of inference is presented in lst. 5.2.

Listing 5.2 Split initialization of resources

```
struct Resources { x: u64, y: u64 }

#[init(core = 0, late = [x])]
fn init0(cx: init::Context) -> init0::LateResources {
    init0::LateResources { x: 1 }
}

#[init(core = 1)]
fn init1(cx: init::Context) -> init1::LateResources {
    init0::LateResources { y: 2 }
}
```

5.2.4 #[shared]

Listing 5.3 placing a static variable in `#[shared]` memory

```
#[idle(core = 0, spawn = [a])]
fn idle(cx: idle::Context) -> ! {
    #[shared]
    static mut X: u64 = 0;

    let x: &'static mut u64 = X;
    cx.spawn.a(x);

    loop {
        // .. use stack allocated variables ..
    }
}

#[task(core = 1)]
fn a(cx: a::Context, x: &'static mut u64) {
    // .. use `x` ..
}
```

Multi-core devices usually have several memory regions to minimize memory contention. RTFM has the concept of core-local and shared memory; it places code and data in different linker sections that then the user can map to the different memory regions that their target device has.

RTFM defaults to core-local memory for anything that does not *need* to be in shared memory but this default can be overridden using the `#[shared]` attribute. The attribute can be placed on any static variable and resources; its effect is forcing the variable to be placed in shared memory.

This attribute can reduce memory contention when using memory managed by a memory pool or a general purpose memory allocator and owning pointers are exchanged between the cores. The program in lst. 5.3 illustrates the problem that may arise when moving pointers between the cores.

Without the `#[shared]` attribute the variable `x` would point into core #0's local memory. Accessing this memory from core #1 would result in contention because the call stack of core #0 is also located in local memory so all accesses to stack allocated variables and function calls that push registers onto the stack would compete with task `a` for access to core #0's local memory.

5.3 Implementation

In this section the source code transformation performed by the DSL in multi-core mode is presented and further memory safety requirements are discussed.

5.3.1 Base binary interface

Listing 5.4 Symbols used to interface the runtime crate

```
extern "C" fn main_0() -> ! { /* .. */ }

extern "C" fn main_1() -> ! { /* .. */ }

extern "C" fn SomeInterruptHandler_0() { /* .. */ }

extern "C" fn SomeInterruptHandler_1() { /* .. */ }
```

In multi-core mode the DSL still relies on a runtime crate but the binary contract between them is different. Instead of a single `main` function for the entry point, there is one `main` function for each core that serves as the program entry point for that core. As symbols must have unique names each `main` function is suffixed with a *core identifier*, a number in the open range `[0 {cores})` that identifies each core on the system. Similarly, each core has its own interrupt handler for any given interrupt; to distinguish these handlers they are also suffixed with a core identifier.

Lst. 5.4 is an example of the symbols used to interface a runtime crate in a dual-core system. Functions `main_0` and `SomeInterruptHandler_0` run on the first core and the other two functions run on the second core.

The DSL does not impose any requirement on which core reaches its `main` function first but requires that *all* static variables are initialized before *any* of the cores reaches its `main` function.

5.3.2 `lock`

The implementation of the `lock` API does not change in multi-core mode.

5.3.3 `xpend`

The ARM Cortex-M ISA does not specify a standard mechanism for signaling interrupts *between* cores so microcontroller vendors provide their own implementation of this feature, usually in the form of some peripheral. To cope with this device specific implementation the DSL expects the `device` module to provide an `xpend` function to signal interrupts between cores. The runtime will call this function when sending a message to a different core. The signature of this function is `fn(u8, impl Nr)` where the first argument is the core that will receive the signal and the second argument is the interrupt being signaled (usually an enumeration).

The implementation of this function for the dual-core LPC55S69 microcontroller is shown in lst. 5.5. That microcontroller contains a `MAILBOX` peripheral that can send a *single* interrupt signal to the other core and also provides registers that can be used to pass a word (32-bit) of information along with the signal. Each bit of this word is used to encode 32 different device interrupts.

On the receiver side the `MAILBOX` interrupt handler, shown in lst. 5.6, is used to forward the interrupts encoded in the 32-bit word to the local `NVIC` peripheral (line 4).

5.3.4 Message passing

The multi-core implementation of message passing is very similar to its single-core counterpart with the caveat that in multi-core there are separate buffers, free queues and ready queues for *each* sender-receiver

Listing 5.5 Implementation of the `xpend` function for the LPC55S69

```
const MAILBOX_IRQ0SET: *mut u32 = (0x5008_B000 + 0x04) as *mut u32;
const MAILBOX_IRQ1SET: *mut u32 = (0x5008_B000 + 0x14) as *mut u32;

pub fn xpend(core: u8, interrupt: impl Nr) {
    let mask = 1 << interrupt.nr();
    if core == 0 {
        unsafe { MAILBOX_IRQ1SET.write_volatile(mask) }
    } else if core == 1 {
        unsafe { MAILBOX_IRQ0SET.write_volatile(mask) }
    }
}
```

Listing 5.6 `MAILBOX` interrupt handler

```
#[no_mangle]
unsafe extern "C" fn MAILBOX_0() {
    let mask = MAILBOX_IRQ1.read_volatile();
    NVIC_ISPR.write_volatile(mask);
    MAILBOX_IRQ1CLR.write_volatile(mask);
}
```

pair. The reason for this is that all the queues are single-producer single-consumer so the same queue cannot be used by more than one core.

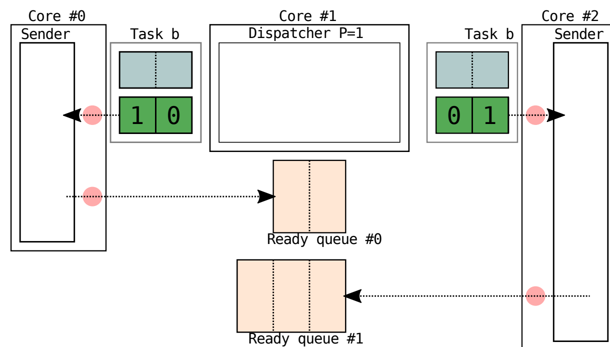


Figure 5.1: Message passing in a multi-core system

Consider the program in lst. 5.7 and the relevant parts of its expansion in lst. 5.8. As both core #0 and core #1 can spawn task `b` which runs on core #1 two message channels are needed: one from core #0 to core #2 and another one from core #1 to core #2. The data structures associated to the first channel are in lines 1-5; the data structures associated to the second channel are in lines 7-11. Fig. 5.1 shows the data structures used in lst. 5.8.

5.3.4.1 `spawn`

The implementation of the `spawn.b` call is just like its single-core version but in this example there are two of versions of it: a cross-core version (lines 13-15) and a core-local version (lines 17-19). The only difference between the two is the static variables they use.

5.3.4.2 Task dispatcher

There is still one task dispatcher per priority level but in multi-core mode the dispatcher must pull messages from several ready queues, one per sender core. Lst. 5.9 shows the core #1 task dispatcher that dispatches

Listing 5.7 Minimal application that features cross-core and core-local message passing

```
1  #[task(core = 0, spawn = [b])]
2  fn a0(cx: a0::Context) { /* .. */ }
3
4  #[task(core = 1, spawn = [b, c])]
5  fn a1(cx: a1::Context) { /* .. */ }
6
7  #[task(core = 2, capacity = 2)]
8  fn b(cx: b::Context, input: i64) { /* .. */ }
9
10 #[task(core = 2)]
11 fn c(cx: c::Context) { /* .. */ }
```

tasks at priority `1`. This dispatcher pulls messages from the cross-core channel (line 3) and the core-local channel (line 9).

5.3.4.3 SPSC

Note that only core #0 dequeues empty slots from the free queue `b1_FQ_S0` and that only core #1 enqueues freed slots back into it. Similarly, only core #0 enqueues into the ready queue `R1_RQ1_S0` and only core #1 dequeues from it. Only one core uses each side of these queues so the Single Producer Single Consumer (SPSC) requirement is respected. More than one task on the same core may enqueue or dequeue into the same queue, but access is mediated within the core using the `lock` API as it was done in the single-core implementation.

5.3.5 Timer queue

Each core has a private timer queue that the other cores will not access. The implementation of the `schedule` API and the `SysTick` handler are just like the single-core version. The only observable difference between the multi-core version of the timer queue and the single-core one is the priority of the `SysTick` handler. When the timer queue may send messages to other cores the priority of the handler is set to be the highest priority *on that core*. The reason for this is parallel execution; the receiver core may be able to execute the task immediately regardless of what tasks may be pending to execute on the sender. Choosing any lower priority can cause a message to be sent to the core later than it could have been dispatched.

5.3.6 Synchronization barriers

During the initialization phase the runtime may insert synchronization barriers at various places to ensure the correct behavior of the system and to avoid breaking Rust memory safety rules.

5.3.6.1 `Barrier`

Lst. 5.10 shows the implementation of the synchronization `Barrier` used by the runtime. It is a one-shot semaphore meant to be used by two cores where one of the core waits until the other core tells it to continue. The abstraction only has two methods: `wait` and `release`. The `wait` method forces the caller to busy wait until the `release` method is called at least once. These methods are meant to be called from different cores to avoid deadlocks.

5.3.6.2 `spawn`

To be able to successfully spawn a task on some other core the other core must have already left the reset state. If cross-core message passing is used in the application the sender core will busy wait until the receiver core has left the reset state. The sender core may perform this wait before `init` starts or before leaving the initialization phase. Where the wait occurs depends on where message passing may occur.

Listing 5.8 Expansion of lst. 5.7

```
enum R2_T1_S0 { b }

static mut b_INPUTS_S0: [MaybeUninit<i64>; 2] = [MaybeUninit::uninit(); 1];
static mut b_FQ_S0: Queue<u8, 2> = Queue::new();
static mut R2_RQ1_S0: Queue<(R2_T1_S0, u8), 2> = Queue::new();

enum R2_T1_S1 { a1, b }

static mut b_INPUTS_S1: [MaybeUninit<i64>; 2] = [MaybeUninit::uninit(); 1];
static mut b_FQ_S1: Queue<u8, 2> = Queue::new();
static mut R2_RQ1_S1: Queue<(R2_T1_S1, u8), 3> = Queue::new();

unsafe fn spawn_b_s0(priority: &Cell<u8>, payload: i64) -> Result<(), i64> {
    // uses `b_INPUTS_S0`, `b_FQ_S0` and `R2_RQ1_S0`
}

unsafe fn spawn_b_s1(priority: &Cell<u8>, payload: i64) -> Result<(), i64> {
    // uses `b_INPUTS_S1`, `b_FQ_S1` and `R2_RQ1_S1`
}

impl a0::Spawn<'_> {
    fn b(&self, payload: i64) -> Result<(), i64> {
        unsafe { spawn_b_s0(self.priority(), payload) }
    }
}

impl a1::Spawn<'_> {
    fn b(&self, payload: i64) -> Result<(), i64> {
        unsafe { spawn_b_s1(self.priority(), payload) }
    }
}
```

If cross-core message passing occurs in `init` then the sender will wait before `init`. If cross-core message passing only occurs in a task then the sender will busy wait after returning from `init`.

The example in lst. 5.11 showcases these two possible scenarios. The expansion of lst. 5.11 is shown in lst. 5.12. In this tri-core application the first core waits for the second core (line 6) before calling the user defined `init` function and then waits for the third core (line 9) before re-enabling interrupts (line 10). Cores #1 and #2 release the first core after they have configured their interrupts and before calling their `init` function in lines 16 and 22, respectively.

Omitting these synchronization barriers does not compromise memory safety. It does result in messages being lost but at worst this only leaks memory: consider the scenario where one core sends a pointer into memory managed by a memory pool; if the message is never delivered then the memory is never returned to the pool and the slot in the `INPUTS` and `INSTANTS` buffers are also lost.

5.3.6.3 Cross-core resource initialization

In applications where cross-core resource initialization is used synchronization barriers are needed to prevent one core observing a late resource in an uninitialized state.

Consider the program in lst. 5.13 and its expansion in lst. 5.14. Task `a` should only start *after* the resource `x` has been initialized so a synchronization barrier is used (line 2). The first core releases the barrier after it has initialized the static variable `x`. The second core waits for the barrier to be released before it re-enables

Listing 5.9 One of the task disptachers of lst. 5.7

```
1  #[no_mangle]
2  unsafe extern "C" fn EXTI0_1() {
3      while let Some((task, index)) = R1_RQ_S0.dequeue() {
4          match task {
5              R1_T1_S0::b1 => { /* .. */ }
6          }
7      }
8
9      while let Some((task, index)) = R1_RQ_S1.dequeue() {
10         match task {
11             R1_T1_S1::a1 => { /* .. */ }
12             R1_T1_S1::b1 => { /* .. */ }
13         }
14     }
15 }
```

Listing 5.10 Implementation of the `Barrier` abstraction

```
1  pub struct Barrier { inner: AtomicBool }
2
3  impl Barrier {
4      pub fn wait(&self) {
5          while !self.inner.load(Ordering::Acquire) {}
6      }
7
8      pub fn release(&self) {
9          self.inner.store(true, Ordering::Release);
10     }
11 }
```

its interrupts. Task `a` can only start after checkpoint (I).

The `wait` and `release` calls contain a memory barrier. In the general case, a memory barrier is needed so that the write to the static variable `x` propagates to the other core before it returns from the call to `wait`.

5.3.6.4 Time zero

When the `schedule` API is used all cores need to leave the initialization phase at about the same time and the `monotonic` timer needs to be reset before any task can start. This is so all tasks observe a time greater than "time zero" when polling the `monotonic` timer.

Using the tri-core application in lst. 5.11 as an example. The synchronization is implemented as shown in lst. 5.15 using 3 `Barrier`s and core #0 is tasked with resetting the monotonic timer. The first core waits until the other two cores are ready to exit the initialization phase (lines 7-8), resets the monotonic timer (line 9) and then unblocks the other two cores (lines 10-11). The other two cores first notify core #0 that they are ready to exit the `init` phase (lines 17 and 25) and then wait until core #0 has reset the monotonic timer (lines 18 and 26).

5.3.7 Code and data placement

Memory contention can degrade the performance of multi-core applications and should be minimized. To this end the framework defaults to placing code and data in core-local memory and only places data in shared memory when required. This section covers how code and data placement is implemented and exposed to the end user.

Listing 5.11 Minimal application that needs barriers due to its use of the `spawn` API

```
#[init(core = 0, spawn = [b])]
fn init(cx: c::Context) { /* .. */ }

#[task(core = 0, spawn = [c])]
fn a(cx: a::Context) { /* .. */ }

#[task(core = 1)]
fn b(cx: b::Context) { /* .. */ }

#[task(core = 2)]
fn c(cx: c::Context) { /* .. */ }
```

The Rust compiler produces ELF binaries when compiling for the ARM Cortex-M architecture. The memory layout of an ELF binary is ultimately decided by the linker but application authors have coarse grained control over the memory layout through *linker sections*. There are a few standard linker sections that the Rust compiler generates:

- `.text`, functions are placed here
- `.rodata`, constants like strings are placed here
- `.bss`, static variables that must be initialized to zero are placed here
- `.data`, the rest of static variables are placed here

To introduce the concept of core-local memory the framework introduces these custom (non-standard) linker sections: `.text_{i}`, `.bss_{i}` and `.data_{i}`. There is one set of these sections per core used by the application. The four default linker sections are treated as *shared* memory.

The framework uses the `#[linker_section]` attribute on the items that must go in core-local memory to place them in one of the custom sections. This attribute is *not* used on items that must go in shared memory; that way they will be in one of the default linker sections.

Consider the example in lst. 5.16 and its expansion in lst. 5.17. The expansion indicates where each variable and function is placed. The placement rules are as follows:

- All user code and interrupt handlers go into the `.text_{i}` section. (Lines 2, 5, 24 and 28)
- All `static mut` resources, core-local `static` resources and task local variables go into the `.data_{i}` section. (Line 9)
- `static` resources shared between cores are not annotated; the compiler automatically places them in the shared `.bss` or `.data` section. (Lines 11-13)
- Buffers, free queues and ready queues used for core-local message passing are placed in the `.bss_{i}` section; the ones used for cross-core message passing are not annotated. (Lines 16-20)
- The timer queue is placed in the `.bss_{i}` section. (Not present in the example)

The data structures shown in lst. 5.17 are depicted in fig. 5.2. Note that unlike fig. 5.1 some data structures in fig. 5.2 – the ones used for core-local message passing – are in core-local memory (core #0 box), whereas all the data structures in fig. 5.1 are in shared memory (outside any core #? box).

5.3.8 Example memory layout

With these custom linker sections the end user can write a linker script to locate the sections in memory regions in a way that minimizes contention. To give an example of how this is accomplished consider the LPC55S69 microcontroller, a dual-core device with 5 RAM regions and one Flash bank.

As there is only one Flash bank, one core must run code off RAM to avoid contention on Flash memory; the other core will run code from Flash. The first two RAM regions will be used for memory local to cores #0

Listing 5.12 Expansion of lst. 5.11

```
static SB1: Barrier = Barrier::new();
static SB2: Barrier = Barrier::new();

unsafe extern "C" fn main_0() -> ! {
    // ..
    SB1.wait();
    let late = crate::init(/* .. */);
    // .. initialize late resources ..
    SB2.wait();
    PRIMASK::clear();
    // ..
}

unsafe extern "C" fn main_1() -> ! {
    // .. configure interrupts ..
    SB1.release();
    // .. call init, etc. ..
}

unsafe extern "C" fn main_2() -> ! {
    // .. configure interrupts ..
    SB2.release();
    // .. call init, etc. ..
}
```

Listing 5.13 Minimal application that features cross-core initialization of resources

```
struct Resources { x: u64 }

#[init(core = 0)]
fn init(ctxt: init::Context) -> init::LateResources {
    init::LateResources { x: 0 }
}

#[task(core = 1, resources = [x])]
fn a(ctxt: a::Context) { /* .. */ }
```

and #1 and a third RAM region will be used as shared memory.

Table 5.1: Mapping linker sections to memory regions

Region	Start address	Size	Sections
Flash	0x00000000	630 KB	`.text`, `.text_0`, `.rodata`
SRAM0	0x20000000	64 KB	`.bss_0`, `.data_0`, `.text_1`
SRAM1	0x20010000	64 KB	`.bss_1`, `.data_1`
SRAM2	0x20020000	64 KB	`.bss`, `.data`

Listing 5.14 Expansion of lst. 5.13

```
const APP: () = {
    static IB0: Barrier = Barrier::new();
    static mut x: MaybeUninit<u64> = MaybeUninit::new();

    unsafe extern "C" fn main_0() -> ! {
        // ..
        let late = crate::init(init::Context { /* .. */ });
        x.as_mut_ptr().write(late.x);
        IB0.release();
        // ..
    }

    unsafe extern "C" fn main_1() -> ! {
        // ..
        IB0.wait();
        PRIMASK::clear();
        // .. (I)
    }
};
```

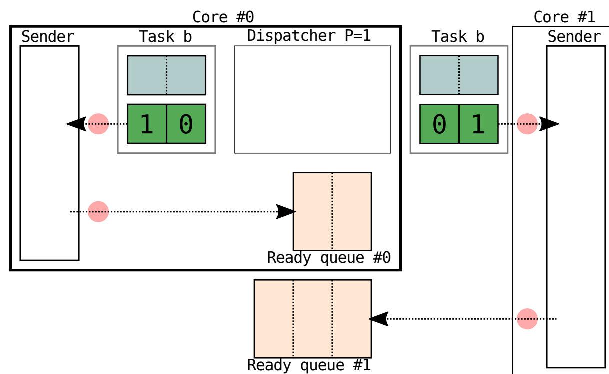


Figure 5.2: Placement of message passing data structures

Listing 5.15 Synchronized exit from the initialization phase

```
static TQB0: Barrier = Barrier::new();
static TQB1: Barrier = Barrier::new();
static TQB2: Barrier = Barrier::new();

unsafe extern "C" fn main_0() -> ! {
    // ..
    TQB1.wait();
    TQB2.wait();
    Monotonic::reset();
    TQB0.release();
    PRIMASK::clear();
    // ..
}

unsafe extern "C" fn main_1() -> ! {
    // ..
    TQB1.release();
    TQB0.wait();
    PRIMASK::clear();
    // ..
}

unsafe extern "C" fn main_2() -> ! {
    // ..
    TQB2.release();
    TQB0.wait();
    PRIMASK::clear();
    // ..
}
```

Listing 5.16 A small application to demo placement of code and data

```
struct Resources { x: u64 }

#[task(core = 0, resources = [x], spawn = [b])]
fn a0(cx: a0::Context) { /* .. */ }

#[task(core = 1, spawn = [b])]
fn a1(cx: a1::Context) { /* .. */ }

#[task(core = 1)]
fn b(cx: b::Context, payload: i64) { /* .. */ }
```

Listing 5.17 Expansion of lst. 5.16

```
#[linker_section = ".text_0"]
fn a(cx: a::Context) { /* .. */ }

#[linker_section = ".text_1"]
fn b(cx: b::Context) { /* .. */ }

const APP: () = {
    #[linker_section = ".data_0"]
    static mut X: u64 = 0;

    static mut b_INPUTS_S0: [MaybeUninit<i64>; 1] = [MaybeUninit::uninit; 1];
    static mut b_FQ_S0: Queue<u8, 1> = Queue::new();
    static mut R1_RQ1_S0: Queue<R1_T1_S0, 1> = Queue::new();

    #[linker_section = ".bss_0"]
    static mut b_INPUTS_S1: [MaybeUninit<i64>; 1] = [MaybeUninit::uninit; 1];
    #[linker_section = ".bss_0"]
    static mut b_FQ_S1: Queue<u8, 1> = Queue::new();
    #[linker_section = ".bss_0"]
    static mut R1_RQ1_S1: Queue<R1_T1_S1, 1> = Queue::new();

    #[linker_section = ".text_0"]
    #[no_mangle]
    unsafe extern "C" fn EXTI0_0() { /* .. */ }

    #[linker_section = ".text_1"]
    #[no_mangle]
    unsafe extern "C" fn EXTI0_1() { /* .. */ }
};
```

5.4 WCET

In this section WCET analysis is performed on all the cross-core API. Asymptotically the WCET of cross-core operations is no different than of core-local operations. However, extra overhead can arise due to contention on shared memory. The focus of this section is characterizing the overhead due to contention.

All measurements were done on a LPC55S69 microcontroller, a device with two ARM Cortex-M33 cores, one Flash bank and 5 RAM regions. Both cores were clocked at 8 MHz. As recommended by the vendor the second core was configured to run code off RAM whereas the first core was configured to run code off Flash.

When memory contention occurs one core is given higher priority to resolve the contention; this results in an observable access delay on the other core. Measurements were mainly done on the first core so higher access priority was given to the second core to include the contention overhead in the measurements.

It should be noted that these measurements were done on an unloaded system and at a clock frequency that results in zero Flash access wait cycles. In a real application, the effect of clock frequency on Flash access times as well as the effect of background DMA transfers need to be taken into account. These effects are application dependent so redoing the measurements under actual load conditions is advised.

5.4.1 Blocking exchange

Listing 5.18 Program used to measure the overhead of a blocking exchange

```
1  static mut X: MaybeUninit<[i32; N]> = MaybeUninit::uninit();
2  static B: Barrier = Barrier::new();
3
4  #[idle(core = 0)]
5  fn idle0(cx: idle0::Context) -> ! {
6      B.wait();
7      let x = unsafe { X.as_ptr().read() };
8      let after = CTIMER0::now();
9      loop {}
10 }
11
12 #[idle(core = 1)]
13 fn idle1(cx: idle1::Context) -> ! {
14     let before = CTIMER0::now();
15     unsafe { X.as_mut_ptr().write([0; N]) }
16     B.release();
17     loop {}
18 }
```

To have a reference point to which compare the overhead of message passing as implemented in RTFM in this section the overhead of a blocking exchange is measured.

This blocking exchange consists of writing a message to a shared memory location on one core, the *sender*, and then reading it out on the other core, the *receiver*. Access to the shared memory location is synchronized using a semaphore: the receiver waits on the semaphore which is set by the sender after the message has been written to the shared memory location.

The program in lst. 5.18 depicts the measurement setup. The `Barrier` abstraction introduced in sec. 5.3.6 is used as a semaphore and a device timer, `CTIMER0`, is configured as a monotonic timer and used to time the operation.

Tbl. 5.2 presents the overhead, computed as `after - before`, for different payload sizes.

Table 5.2: Overhead of blocking exchange for different payload sizes

Payload size (bytes)	Elapsed
0	18
4	20 - 21
8	22 - 25
16	29 - 32
32	46 - 47

Although this blocking exchange would be a poor choice for most applications due to its blocking nature, it is the fastest way to exchange data between two cores and serves as a baseline for all the other measurements presented in this section. The operations performed in the exchange are as follows:

- Sender
  - `write` operation on `X`. `O(sizeof(X))` overhead
  - `write` operation on `B`. `O(1)` overhead
  - memory barrier . `O(1)` overhead
- Receiver
  - memory barrier . `O(1)` overhead
  - `read` operation on `B`. `O(1)` overhead
  - compare and branch. `O(1)` overhead
  - `read` operation on `X`. `O(sizeof(X))` overhead

Overall the overhead is `O(sizeof(X))`, proportional to the payload size.

5.4.2 `xpend`

Listing 5.19 Program used to measure the overhead of cross-core interrupt signaling

```
#[idle(core = 1)]
fn idle(cx: idle::Context) -> ! {
    let before = Instant::now();
    xpend(0, Interrupt::CTIMER0);
    loop {}
}

#[task(core = 0, binds = CTIMER0)]
fn ctimer0(cx: ctimer0::Context) {
    let after: Instant = cx.start;
}
```

In this section the overhead of cross-core interrupt signaling is measured and compared to the overhead of core-local interrupt signaling. Cross-core interrupt signaling is not standardized in the ARM Cortex-M ISA and its implementation varies from device to device; this section analyzes the implementation for the LPC55S69, which was covered in sec. 5.3.3.

The program in lst. 5.19 was used to measure the time between a call to `xpend` on one core (line 3) and the start of the corresponding interrupt on the target core (line 10).

The overhead (`after - before`) was found to be 41 CPU cycles. The steps involved in this operation are:

- core 0: `xpend`
- core 1: `MAILBOX` preempts the current context ~ 12 cycles
- core 1: run `MAILBOX` handler to completion
- core 1: tail chain into the `CTIMER0` handler ~ 6 cycles

The preemption operation could take 17 additional CPU cycles if FP registers were being used in `idle`, according to tbl. 4.5.

5.4.2.1 vs `pend`

Listing 5.20 Program used to measure the overhead of core-local interrupt signaling

```
#[idle(core = 0)]
fn idle(cx: idle::Context) -> ! {
    NVIC::pend(Interrupt::CTIMER0);
    loop {}
}

#[task(core = 0, binds = CTIMER0)]
fn ctimer0(cx: ctimer0::Context) {
    asm::bkpt();
}
```

The program in lst. 5.20 was used to measure the time between a call to `pend` on one core (line 3) and the start of the corresponding interrupt on the *same* core (line 9).

The overhead was found to be 20 cycles. The steps involved in this operation are:

- `NVIC::pend`
- `CTIMER0` preempts the current context ~ 12 cycles

The preemption operation could take 17 additional cycles if FP registers were being used in `idle`.

In a device with dedicated cross-core interrupt signaling the gap between the overhead of `xpend` and the overhead of `pend` would be very close to zero.

5.4.3 Message passing

In this section the overhead of cross-core message passing is analyzed under conditions of memory contention.

5.4.3.1 `spawn`

Listing 5.21 Program used to measure the overhead of cross-core message passing

```
struct Resources {
    #[shared]
    x: AtomicU32,
}

#[task(core = 0, spawn = [a1])]
fn a0(cx: a0::Context) {
    asm::bkpt(); // before
    cx.spawn.a1([0; N]);
    asm::bkpt(); // after
}

#[idle(core = 1, resources = [x])]
fn idle(cx: idle::Context) -> ! {
    loop {
        cx.resources.x.store(0, Ordering::Relaxed);
    }
}
```

The operations involved in a cross-core `spawn` calls are the same as the operations required to do a core-local `spawn` call except that they are performed on shared memory rather than core-local memory. The cross-core version is also a constant time operation whose overhead is proportional to the size of the message payload.

The program in lst. 5.21 was used to measure the overhead of a `spawn` call (lines 8-10). To add memory contention to the measurement the second core is set to continuously perform writes to an atomic variable located in shared memory (line 16).

Tbl. 5.3 presents the overhead of the `spawn` calls for different sized payloads. The table also includes the overhead under conditions of no contention where the variable `X` was moved to core-local memory.

Table 5.3: WCET of `spawn` under (no) contention

Payload size (bytes) / contention	no	yes
0	35	36
4	38	39
8	40	43
16	56	57
32	68	71

The measurements presented correspond to the worst case path where both `lock`s are taken in the `spawn` call.

5.4.3.2 Task dispatcher

Listing 5.22 Program used to measure the overhead of the task dispatcher

```
1  struct Resources {
2      #[shared]
3      x: AtomicU32,
4  }
5
6  #[task(core = 0, spawn = [a0])]
7  fn a0(cx: a0::Context, input: [i32; N]) {
8      asm::bkpt();
9  }
10
11 #[init(core = 1, spawn = [a0])]
12 fn init(cx: init::Context) {
13     cx.spawn.a0([0; N]);
14 }
15
16 #[idle(core = 1, resources = [x])]
17 fn idle(cx: idle::Context) -> ! {
18     loop {
19         cx.resources.x.store(0, Ordering::Relaxed);
20     }
21 }
```

The operations involved in dispatching a software task in multi-core mode are similar to the operations performed in single-core mode with the small difference that in multi-core mode multiple ready queues, as many as the number of cores the application uses, are dequeued from the task dispatcher. In the worst case, this adds up to `{cores} - 1` failed dequeue operations on empty ready queues to the overhead of dispatching a software task. The WCET remains constant time because those dequeue operations are constant time and the number of cores the application uses is fixed and known at compile time.

The program in lst. 5.22 was used to measure the overhead of dispatching a software task, measured as the time it takes to go from the entry to the interrupt that dispatches the tasks to the first statement in the next task to be dispatched (line 8).

Artificial contention was added by having the second core continuously perform write operations on an atomic variable that resides in shared memory (line 19).

The following table presents the task dispatching overhead for payloads of different sizes. The table also includes the overhead in the scenario where the second core causes no memory contention.

Table 5.4: Overhead of the task dispatcher under (no) contention

Payload size (bytes) / contention	no	yes
0	63	65
4	70	74
8	71	73
16	74	79
32	86	91

The numbers presented correspond to the worst case path where the dispatcher first tries to dequeue an empty core-local queue and then moves onto the cross-core queue.

5.4.3.3 End to end

Listing 5.23 Program used to measure the end to end overhead of message passing

```
1  struct Resources {
2      #[shared]
3      x: AtomicU32,
4  }
5
6  #[task(core = 0)]
7  fn a0(cx: a0::Context, payload: [i32; N]) {
8      let after = Instant::now();
9  }
10
11 #[task(core = 1, spawn = [a0])]
12 fn a1(cx: a1::Context) {
13     let before = Instant::now();
14     cx.spawn.a0([0; N]);
15 }
16
17 #[idle(core = 1, resources = [X])]
18 fn idle(cx: idle::Context) -> ! {
19     loop {
20         cx.resources.x.store(1, Ordering::Relaxed);
21     }
22 }
```

This section presents measurements of the time it takes for a message to be dispatched from the moment it is sent, or the end to end time.

The program in lst. 5.23 was used to perform the measurement. The effect of memory contention is included in the measurement by having the second core continuously write to an atomic variable located in shared memory (line 20).

Tbl. 5.5 presents the end to end overhead for different payload sizes. The table also includes measurements under conditions of no contention where the location of the variable `X` was changed to core-local memory.

Table 5.5: End to end overhead of message passing under (no) contention

Payload size (bytes) / contention	no	yes
0	125	128
4	136	140
8	138	142
16	149	155
32	174	184

The number presented correspond to the WCET paths for both the `spawn` call and the task dispatcher code.

5.4.3.4 Contention effect

Overall it was found that the effect of contention on the WCET is proportional to the number of operations performed on shared memory. In the case of the `spawn` API the operations on shared memory are proportional to the size of the payload so the additional overhead of memory contention can be folded into the `O(sizeof(T))` component of the WCET as a $(1 + \alpha)$ factor.

5.4.4 Timer queue

In single-core mode the biggest contributor to the WCET of the `schedule` API and the timer queue handler are the enqueue / dequeue operations on the timer queue. These operations have a WCET of `O(log2(N))` where `N` is the capacity of the queue.

In multi-core mode the timer queue remains in core-local memory so the enqueue / dequeue operations are not subject to additional overhead due to memory contention. The only operations subject to memory contention overhead are those performed on shared memory and they are dominated by the read / write operation to the `INPUTS` buffer which has a WCET of `O(sizeof(T))`.

5.5 Code sharing

Listing 5.24 Memory contention due to code sharing

```
#[rtfm::app(cores = 2, /* .. */)]
const APP: () = {
    #[task(core = 0)]
    fn a(cx: a::Context) {
        foo();
    }

    #[task(core = 1)]
    fn b(cx: b::Context) {
        foo();
    }
};

fn foo() { /* .. */ }
```

As covered in sec. 5.3.7 RTFM defaults to placing all code and data in core-local memory. However, this only applies to items managed by the DSL. All the code and data that is defined outside the `#[app]` module,

including external dependencies, is placed in shared memory by default. This shared memory default can lead to code sharing between the cores and this can cause memory contention on instruction fetch which greatly degrades performance.

The program in lst. 5.24 showcases the problem. If the function `foo` is *not* inlined in both tasks `a` and `b` then both cores will run `foo` off shared memory leading to memory contention. To avoid the problem shared code, the `.text` section, can be cached on each core. Then both cores will execute `foo` off their own cache avoiding memory contention. Instruction caches may not be available on all devices so this is not always an option.

Chapter 6

Heterogeneous devices

When used in multi-core mode RTFM provides two codegen backends: the `homogeneous` backend and the `heterogeneous` backend. The `homogeneous` backend is for targeting homogeneous multi-core devices like the LPC55S69, and was covered in sec. 5. The `heterogeneous` backend uses the μAMP framework and tools to support heterogeneous multi-core devices. This chapter describes how the `heterogeneous` backend differs from the `homogeneous` one.

6.1 μAMP

μAMP (micro Asymmetric Multi-Processing) is a framework for building applications for heterogeneous multi-core devices. The framework consists of two parts: the `cargo-microamp` subcommand and the `#[microamp::shared]` attribute.

The `cargo-microamp` subcommand builds an application once for each core in the target device. Each compilation process can use a different compilation target; this is how heterogeneous multi-core devices are supported. At the end of the process several ELF images are produced; each image can use an instruction set that is incompatible with the instruction set used in the other images.

Apart from using a different compilation target `cargo-microamp` passes a `--cfg core={i}` flag to the compiler when building the crate. This lets the author use the conditional compilation feature built into the language to customize the behavior of the program for each core on the device.

Listing 6.1 A minimal dual-core application

```
1  #[cfg(core = "0")]
2  unsafe extern "C" fn main() -> ! { /* .. */ }
3
4  #[cfg(core = "1")]
5  unsafe extern "C" fn main() -> ! { /* .. */ }
```

For example, a dual-core application may look like lst. 6.1. The function `main`, the entry point, appears twice but only one of them will be included in the source file when `cargo-microamp` builds the crate for cores #0 and #1.

Using the LPC54114 microcontroller, a dual core device with one ARM Cortex-M4F core and one ARM Cortex-M0+ core, as an example the previous application would be compiled for the targets `thumbv7em-none-eabihf` (ARM Cortex-M4F) and `thumbv6m-none-eabi` (ARM Cortex-M0+) as shown in lst. 6.2. The subcommand produces two ELF images that use different instruction sets as confirmed by the `readelf` command in lst. 6.3.

Listing 6.2 `cargo-microamp` command to build an application for the LPC54114

```
$ cargo microamp \
      --bin app \
      --target thumbv7em-none-eabihf,thumbv6m-none-eabi
```

Listing 6.3 Details about the ELF images produced in lst. 6.2

```
$ readelf -A app-0
Attribute Section: aeabi
File Attributes
  Tag_CPU_arch: v7E-M
  Tag_CPU_arch_profile: Microcontroller
  Tag_THUMB_ISA_use: Thumb-2
  Tag_FP_arch: VFPv4-D16
  Tag_ABI_HardFP_use: SP only
  Tag_ABI_VFP_args: VFP registers
  Tag_CPU_unaligned_access: v6
  Tag_FP_HP_extension: Allowed

$ readelf -A app-1
Attribute Section: aeabi
File Attributes
  Tag_CPU_arch: v6S-M
  Tag_CPU_arch_profile: Microcontroller
  Tag_THUMB_ISA_use: Thumb-1
  Tag_CPU_unaligned_access: None
```

6.1.1 `#[shared]`

By default, all the code and data in a μAMP application, including the one that comes from external dependencies, is placed in core-local memory. A `static` variable, for example, that appears in the top crate will appear in both images but have different addresses in each image: each image gets a *copy* of the `static` variable.

To revert this default the `#[microamp::shared]` attribute is provided by the framework. This attribute forces a static variable to be placed in shared memory so that the variable in *all* the images refers to the *same* memory location. In the example in lst. 6.4 both cores increase the same atomic variable.

This can be confirmed in the symbol list of the ELF images, shown in lst. 6.5. The variable `X` refers to the same memory location on both images.

Without the `#[microamp::shared]` attribute each core accesses its own copy of the variable `X` and no memory is shared between the cores. Lst. 6.6 shows the list of symbols for that alternative scenario. The `X` variable in each image refers to a different memory location.

6.2 `heterogeneous`

In `heterogeneous` mode code generation is largely the same except for how items are laid out in memory. In `heterogeneous` mode custom linker sections are not used because μAMP's default of placing items in core-local memory matches RTFM's placement policy. Instead of custom linker sections, to place an item in one core or another conditional compilation through the `#[cfg(core = "{i}")]` attribute is used. Items that need to be placed in shared memory are marked with the `#[microamp::shared]` attribute.

Lst. 6.7 shows the different code generated for the sample program shown in sec. 5.3.7.

Listing 6.4 Inter-processor communication using a `#[shared]` variable

```
1  #[microamp::shared]
2  static X: AtomicU32 = AtomicU32::new(0);
3
4  #[cfg(core = "0")]
5  unsafe extern "C" fn main() -> ! {
6      loop {
7          X.fetch_add(1, Ordering::AcqRel);
8      }
9  }
10
11 #[cfg(core = "1")]
12 unsafe extern "C" fn main() -> ! {
13     loop {
14         if X.load(Ordering::Acquire) % 2 == 0 {
15             // ..
16         }
17     }
18 }
```

Listing 6.5 Memory location of a `#[shared]` variable

```
$ arm-none-eabi-CSn amp-0 | tail -n1
20020000 00000004 D X

$ arm-none-eabi-CSn amp-1 | tail -n1
20020000 00000004 D X
```

6.3 Additional restrictions

The full multi-core API can be used with the `heterogeneous` backend but this backend imposes an additional restriction on the message passing API: it is not permitted to send *code*, like function pointers and trait objects, between cores; only data can be exchanged between the cores.

The reason for this is that due to the heterogeneous nature of the target device a function that works on one core may not work another because the cores could have different instruction sets. Trying to execute a function compiled for one core on a different core may produce an "illegal instruction" exception in the best case but it could also result in undefined behavior.

6.4 No code sharing

The `heterogeneous` backend defaults to not sharing code between the cores. Compiling the example from sec. 5.5 using the `heterogeneous` backend produces a copy of the function `foo` in each image regardless of whether inlining occurs or not. In addition to this it is not possible to send a pointer into core #0's copy of function `foo` to core #1 because this is rejected by the DSL. In conclusion, it is not possible for cores to share code within the safe subset of Rust.

For this reason the `heterogeneous` backend may also be suitable for homogeneous devices that do not have an instruction cache. No code sharing means that no memory contention can occur when fetching code from memory.

Listing 6.6 Memory location of a core-local static variable

```
$ arm-none-eabi-CSn amp-0 | tail -n1
20000000 00000004 B X

$ arm-none-eabi-CSn amp-0 | tail -n1
2001011c 00000004 B X
```

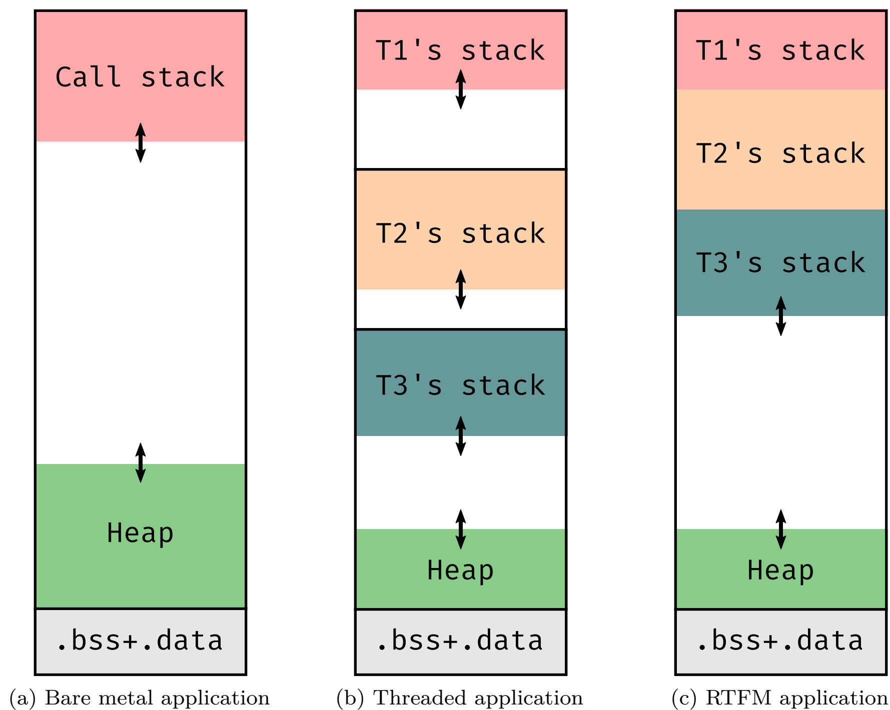
Listing 6.7 Expansion of lst. 5.16 using the `heterogeneous` backend

```
#[cfg(core = "0")]
fn a(cx: a::Context) { /* .. */ }

#[cfg(core = "1")]
fn b(cx: b::Context) { /* .. */ }

const APP: () = {
    #[cfg(core = "0")]
    static mut X: u64 = 0;

    #[microamp::shared]
    static mut b_INPUTS_S0: [MaybeUninit<i64>; 1] = [MaybeUninit::uninit; 1];
    #[microamp::shared]
    static mut b_FQ_S0: Queue<u8, 1> = Queue::new();
    #[microamp::shared]
    static mut R1_RQ1_S0: Queue<R1_T1_S0, 1> = Queue::new();

    #[cfg(core = "1")]
    static mut b_INPUTS_S1: [MaybeUninit<i64>; 1] = [MaybeUninit::uninit; 1];
    #[cfg(core = "1")]
    static mut b_FQ_S1: Queue<u8, 1> = Queue::new();
    #[cfg(core = "1")]
    static mut R1_RQ1_S1: Queue<R1_T1_S1, 1> = Queue::new();

    #[cfg(core = "0")]
    #[no_mangle]
    unsafe extern "C" fn EXTI0() { /* .. */ }

    #[cfg(core = "1")]
    #[no_mangle]
    unsafe extern "C" fn EXTI0() { /* .. */ }
};
```

Chapter 7

Stack usage analysis

7.1 Stack overflows



(a) Bare metal application (b) Threaded application (c) RTFM application

Figure 7.1: Common program memory layouts

Historically, programs are laid out in memory as shown in fig. 7.1a, with static variables, shown as `.bss+.data` in the graph, in some fixed location and the (call) stack and the heap growing towards each other. In the ARM architecture the stack grows downwards, towards smaller addresses, by default.

The problem that can arise with this layout is that the stack can grow too large and collide into the heap. This situation is called a *stack overflow*, or *stack overrun*. The result of this collision is that the stack overwrites parts of the heap corrupting it and the values allocated in it. Stack overflows also have negative effects on systems that do not use a heap because it can still overwrite and corrupt static variables.

In threaded systems each thread has its own stack and these stacks are usually laid next to each other in

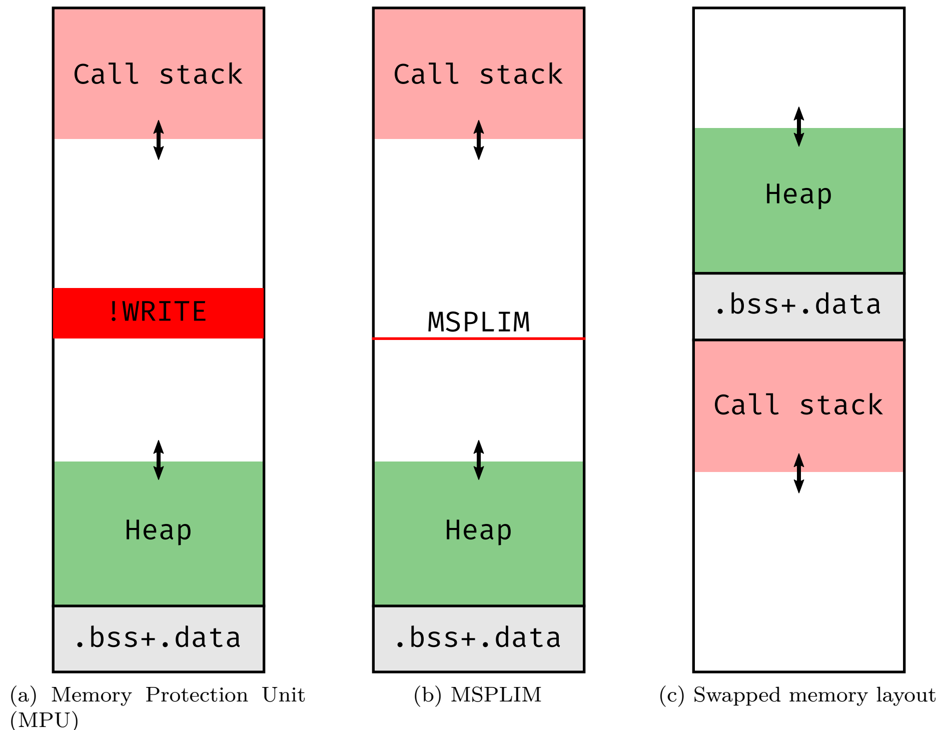
memory, see fig. 7.1b. If the stack of one thread grows too large it can overrun the stack of another thread corrupting it. The chance of stack overflows increases as the number of threads increases because the size of the stacks decreases (the system has limited memory) and the number of inter-thread boundaries increases.

Selecting the stack size of threads in embedded applications is a hard problem. The maximum number of threads is usually known at compile time so one approach is to split the limited memory evenly among all the threads but this can lead to the allocated memory being underutilized by some threads and other threads to not have enough memory. Statically analyzing the stack usage of each thread can provide useful information that can guide the process of choosing a stack size for each thread.

The chance of stack overflows is lesser in RTFM applications, in comparison to threaded applications, because the stacks of tasks sit next to each other with no gaps, see fig. 7.1c. As there are no empty spaces memory is maximally utilized.

Still, it is useful to statically analyze the stack usage of RTFM applications for this is a requirement for certifying safety critical applications.

7.2 Stack overflow protection



(a) Memory Protection Unit (MPU)

(b) MSPLIM

(c) Swapped memory layout

Figure 7.2: Stack overflow protection mechanisms

Regardless of whether stack usage analysis is performed or not one must put protection against stack overflows in place for the stack can grow indefinitely in safe Rust and safe Rust should never corrupt memory. This section describes some stack overflow mechanisms one can use in ARM Cortex-M applications.

7.2.1 Memory Protection Unit (MPU)

The Memory Protection Unit (MPU) is an ARM Cortex-M peripheral used to control read / write access to some regions of memory. Although meant to be used to implement process isolation in multi-process systems this peripheral can also be used as a stack overflow protection mechanism.

To protect against stack overflows a guard page with disabled write access can be placed between the stack and the heap, see fig. 7.2a. This creates a boundary between the two; if either tries to grow beyond the boundary it causes the MPU to raise a `MemManage` exception averting memory corruption.

This approach can also be used in multi-threaded applications. In that scenario the guard page can be set between the stack of the currently active thread and the stack of the neighbor thread. The guard page needs to be changed every time there is a context switch between threads.

Although the MPU peripheral is specified by ARM, it is an optional component on most Cortex-M devices but not available on Cortex-M0 devices.

Integration of the MPU in the RTFM model was explored in rtfm-lang in [37].

7.2.2 MSPLIM

The latest iteration of the Cortex-M sub-architecture, ARMv8-M, provides built-in stack overflow protection through the special stack pointer limit registers: MSPLIM and PSPLIM. If the stack pointer decreases beyond the value contained in one of these registers then an exception is raised. These registers can be used to create a boundary between the stack and the heap, see fig. 7.2b.

This approach can also be used in multi-threaded applications. In those applications each thread has a different stack limit so the value of the limit register needs to be updated in each context switch.

These registers are only available on ARMv8-M devices where the Security extension is implemented.

7.2.3 Swapped memory layout

Single-core Cortex-M devices usually have a single region of RAM surrounded by reserved address space. Trying to write into this reserved address space raises a `HardFault` exception. This fact can be used to implement stack overflow protection.

If the standard memory layout is changed as follows: the stack is placed at the bottom, the static variables at the middle and the heap at the top then stack overflows would result in a write to the reserved address space, which raises a `HardFault` exception instead of corrupting memory, see fig. 7.2c.

This approach can be used for RTFM applications but not for multi-threaded applications, which have multiple stacks.

7.3 `cargo-call-stack`

`cargo-call-stack` is a Cargo subcommand that performs whole program stack usage analysis of a Rust application and produces a DOT file that contains the call graph of the program where each node in the call graph is annotated with its individual stack usage and cumulative (worst-case) stack usage.

Lst. 7.1 shows an RTFM application that uses 3 software tasks and 2 different priority levels. The tasks in this application invoke the functions `foo`, `bar` and `baz` and the function `foo` invokes function `bar`. Lst. 7.2 shows how `cargo-call-stack` is invoked to produce the call graph for the application in lst. 7.1. fig. 7.3 shows the call graph produced by `cargo-call-stack`.

7.3.1 Capabilities

This section goes into more detail about the capabilities of `cargo-call-stack`.

7.3.1.1 Filtering

`cargo-call-stack` produces a whole program call graph by default but it can also filter the call graph to only include nodes reachable from a *starting point* specified in the command line.

Listing 7.1 3 tasks and 2 priorities

```
#[task(priority = 1)]
fn a(cx: a::Context) {
    foo();
}

#[task(priority = 1)]
fn b(cx: b::Context) {
    bar();
}

#[task(priority = 2)]
fn c(cx: c::Context) {
    baz();
}
```

Listing 7.2 Building a call graph with `cargo-call-stack`

```
$ cargo call-stack --bin app > cg.dot
$ dot -Tsvg cg.dot > cg.svg
```

Consider the program in lst. 7.3 . By default `cargo-call-stack` generates the call graph in fig. 7.4a, which includes `idle` and the hardware tasks `exti0` and `exti1`, but one can filter the graph to focus on `idle` using the command in lst. 7.4 to obtain call graph in fig. 7.4b.

7.3.1.2 Cycles

`cargo-call-stack` can compute the max stack usage of programs that contain cycles if all the nodes in each cycles use zero stack usage. An example is presented in lst. 7.5; its call graph is shown in fig. 7.5. The program contains a cycle formed by functions `foo`, `bar` and `baz`. All the nodes that are part of the cycle are grouped in a cluster named `SCC0` (SCC stands for Strongly Connected Component).

7.3.1.3 Function pointers

`cargo-call-stack` can reason about indirect function calls through function pointers. When a function pointer call is encountered a fictitious node representing the call is inserted in the graph and edges between that node and all the potential callees are added. Only functions that match the signature of the function pointer and are coerced into a function pointer are considered as potential callees.

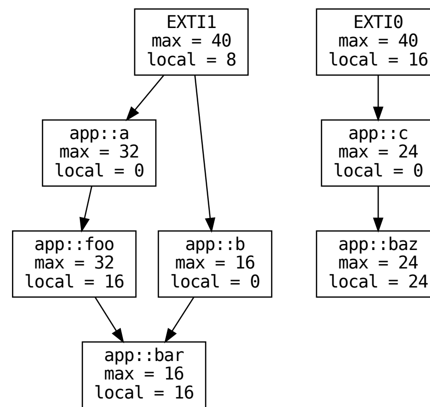


Figure 7.3: Three tasks; two priority levels

Listing 7.3 2 tasks and idle

```
#[idle]
fn idle(cx: idle::Context) -> ! {
    foo();
    bar();
    loop {}
}

#[task(binds = EXTI0)]
fn exti0(cx: exti0::Context) {
    foo();
}

#[task(binds = EXTI1)]
fn exti1(cx: exti1::Context) {
    bar();
}
```

Listing 7.4 command to filter the call graph

```
$ # starting point = 'app::idle'
$ cargo call-stack --bin app 'app::idle' cg.dot
```

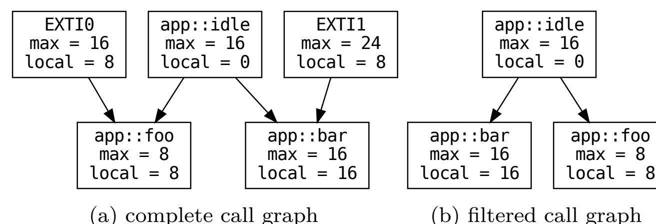
In the example program in lst. 7.6 there are three functions with signature `fn() -> u32` but only two of them are converted into function pointers (lines 3 and 5). The function pointer call (line 7) cannot invoke the function `baz`, which is never coerced into a function pointer, so it is not included in the list of potential callees when computing the call graph.

The call graph for this program is shown in fig. 7.6. Node `fn() -> u32` is the fictitious node that represents the call of a function pointer with signature `fn() -> u32`. Only functions `foo` and `bar`, which are the potential callees, are connected to this fictitious node.

7.3.1.4 Trait objects

7.3.1.4.1 Dynamic dispatch Method calls on trait objects are dynamically dispatched; the operation results in the trait method invocation of the concrete type behind the trait object. In the call graph this form of dynamic dispatch is also depicted as a fictitious node that connects the call site to the potential callees.

In the program in lst. 7.7 there are three zero sized types that implement the `Foo` trait: `Bar`, `Baz` and `Quux`. Only the two first are converted into trait objects (lines 3 and 5) and potential callees for the dynamic dispatch that occurs in line 7. In line 8 trait method `foo` is called on type `Quux` but this a direct function call.



(a) complete call graph (b) filtered call graph

Figure 7.4: Filtering a call graph

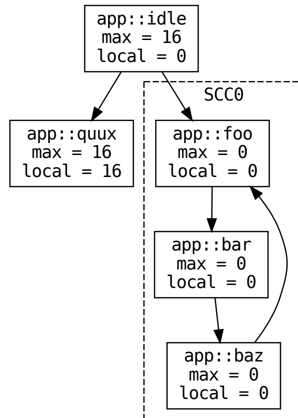


Figure 7.5: Cycle with bounded worst-case stack usage

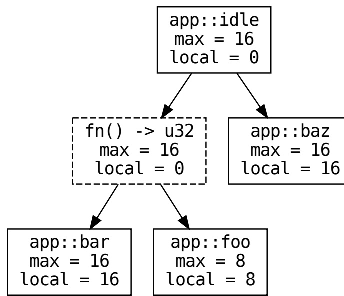


Figure 7.6: Call graph that contains a function pointer call

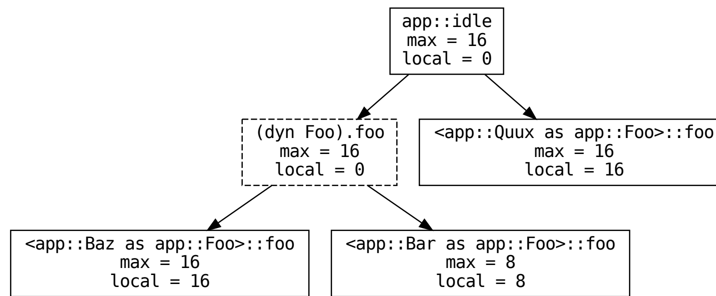


Figure 7.7: Call graph that contains dynamic dispatch

Listing 7.5 A cyclic program

```
#[idle]
fn idle(cx: idle::Context) -> ! {
    foo();
    quux();
    loop {}
}

fn foo() {
    if X.load(Ordering::Relaxed) { bar() }
}

fn bar() {
    if X.load(Ordering::Relaxed) { baz() }
}

fn baz() {
    if X.load(Ordering::Relaxed) { foo() }
}
```

Listing 7.6 A function pointer call

```
#[idle]
fn idle(cx: idle::Context) -> ! {
    let mut x: fn() -> u32 = foo;
    if X.load(Ordering::Relaxed) {
        x = bar;
    }
    x();
    baz();
    loop {}
}

fn baz() -> u32 { /* .. */ }
```

The call graph for this program is shown in fig. 7.7. Node `(dyn Foo).foo` is the fictitious node that represents the invocation of method `foo` on the trait object `dyn Foo`. Two edges connect this node to the implementations of method `foo` for types `Bar` and `Baz`.

7.3.1.4.2 Destructors Freeing (`drop`-ing) a trait object runs the *drop glue* of the type behind the trait object. The drop glue is code generated by the compiler to free a type; it includes calls to the destructors of all the fields of the type plus a call to its user defined destructor, `Drop` trait implementation, if it has one.

This fact is reflected in the call graph using a fictitious node that connects the trait object destructor to all the drop glues that could be called by it.

In the program in lst. 7.8 a trait object is allocated on the heap (line 3). This *boxed* trait object can refer to either `Bar` or `Baz` depending on the state of the atomic boolean `X` before it is destroyed in line 7 types `Bar` and `Baz` have custom destructors (lines 14 and 16) and either could be invoked by `dyn Foo`'s destructor.

The call graph for this program is shown in fig. 7.8. Node `drop(dyn Foo)` is a fictitious node that represents the invocation of the trait object destructor. This node is connected to the drop glue functions (`real_drop_in_place`) of both `Bar` and `Baz`. Each drop glue function respectively calls the user defined destructor (`Drop`) of the type is freeing.

Listing 7.7 A program that uses a trait object

```
#[idle]
fn idle(cx: idle::Context) -> ! {
    let mut to: &dyn Foo = &Bar;
    if X.load(Ordering::Acquire) {
        to = &Baz;
    }
    to.foo();
    Quux.foo();
    loop {}

}

trait Foo {
    fn foo(&self) -> bool;
}

impl Foo for Bar { /* .. */ }
impl Foo for Baz { /* .. */ }
impl Foo for Quux { /* .. */ }
```

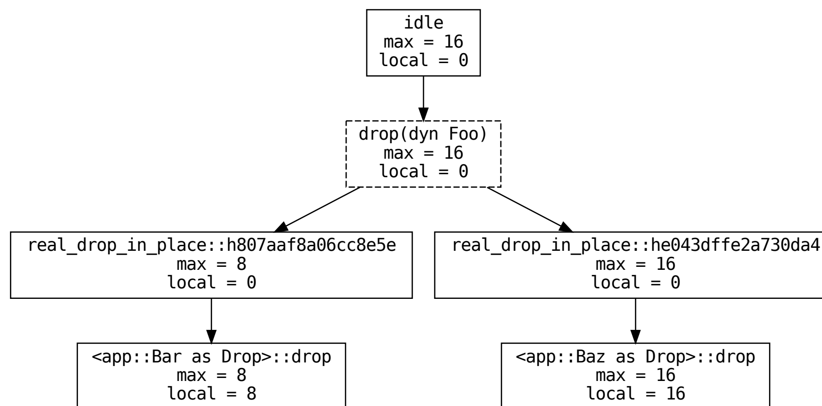


Figure 7.8: Call graph that contains a trait object destructor

Listing 7.8 Freeing a trait object

```
#[idle]
fn idle(cx: idle::Context) -> ! {
    let mut x: Box<dyn Foo> = Box::new(Bar);
    if X.load(Ordering::Relaxed) {
        unsafe { core::ptr::write(&mut x, Box::new(Baz)) }
    }
    drop(x);
    loop {}
}

trait Foo {}

impl Foo for Bar {}
impl Drop for Bar { /* .. */ }
impl Foo for Baz {}
impl Drop for Baz { /* .. */ }
```

7.3.2 Implementation

Computing the worst case stack usage consists of three steps: computing the stack usage of each subroutine in the program, finding the call graph of the whole program and then traversing the graph to add up all individual stack usages.

7.3.2.1 Per function stack usage

7.3.2.1.1 `emit-stack-sizes` The official Rust compiler, `rustc`, uses LLVM, the Low Level Virtual Machine, as its code generation backend. LLVM has a feature called `emit-stack-sizes` [38] that adds stack usage information to every function it lowers from LLVM IR (Intermediate Representation) to machine code. This information is exposed as a linker section named `.stack_sizes` when the output format is ELF.

This feature is exposed behind the unstable `rustc` flag `-Z emit-stack-sizes`. `cargo-call-stack` uses this flag to add stack usage information to the program it builds and then parses the output ELF file to extract this information.

The stack usage information produced by LLVM is accurate except in the case of functions that contain inline assembly (`asm!` macro in Rust code). LLVM does not parse the contents of this inline assembly and simply assumes that it does not make use of the stack. This is a reasonable assumption for most practical uses of inline assembly so `cargo-call-stack` makes the same assumption and additionally informs the user about these assumption by printing warnings about them.

Lastly, LLVM does not produce stack usage information for subroutines defined in module level assembly (`global_asm!` macro in Rust code).

7.3.2.1.2 Machine code analysis Not all the machine code in the final artifact comes from Rust code. Embedded applications occasionally use assembly to implement subroutines that have non-standard calling conventions. LLVM does not provide stack usage information for subroutines written in assembly so `cargo-call-stack` has an analysis pass that extracts stack usage information from ARM Cortex-M machine code.

The analysis pass scans each subroutine in the ELF, locates the instructions that decrease the stack pointer (`sp`) register (listed below) and adds their individual stack usage to compute the maximum stack usage. The pass does not provide a stack usage for subroutines that include branching within the subroutine – these are the product of loops and conditionals. However, this is sufficient to analyze all the subroutines written in pure assembly that appear in practice.

- `push {r7, lr}`, stores some registers onto the stack
- `stmdb sp!, {r4, r5, r6, r7, r8, lr}`, stores some registers onto the stack
- `sub sp, #4`, decrements the stack pointer (16-bit encoding)
- `sub.w sp, sp, #520`, decrements the stack pointer (32-bit encoding)
- `vpush {d8}`, stores some FPU registers onto the the stack

7.3.2.2 Call graph

The bulk of the call graph analysis is performed on the LLVM IR produced by the Rust compiler. LLVM IR is used instead of Rust source code because the IR produced by the compiler has already been optimized and it very closely matches the machine code in the final artifact. By using LLVM IR the effect of inlining is already included in the artifact and does not need to be accounted for during the analysis.

Analyzing the machine code has similar benefits but LLVM IR was chosen instead to reduce implementation effort. LLVM IR syntax is architecture independent so a single parser can deal with all kinds of compilation targets. Machine code on the other hand requires a parser (decoder) for every architecture that one intends to support.

7.3.2.2.1 LLVM IR

The Rust compiler emits LLVM IR in human readable text format. `cargo-call-stack` parses this output to extract information about function definitions and function calls.

Function definitions appears as `define` items in the IR and function calls appear as the `call` and `invoke` instructions. In the case of direct function calls the callee that appears in the `call` or `invoke` instruction is an *identifier* (syntax: `@identifier`) that matches the name of function being called. In indirect function calls the callee is a *value* (syntax: `%value`) previously defined in the function body and that corresponds to a function pointer.

Lst. 7.9 shows the IR of the task `a` from the program in lst. 7.1 which directly calls function `foo`.

Listing 7.9 LLVM IR of a direct function call

```
define void @a() unnamed_addr #0 {
start:
  tail call void @foo()
  ret void
}
```

Lst. 7.11 shows the IR of the function `foo` shown in lst. 7.10 which performs a function pointer call. In the IR, the callee of the `call` instruction is the value `%0`, the argument of the function.

Listing 7.10 Function pointer call

```
fn foo(f: fn() -> u32) -> u32 {
    f()
}
```

Listing 7.11 LLVM IR of an indirect function call

```
define i32 @foo(i32 ()* nocapture nonnull) unnamed_addr #0 {
  %1 = tail call i32 %0() #8
  ret i32 %1
}
```

A node is added to the call graph for every function definition in the IR and for every direct function call in the IR an edge is added between the caller an the callee. Handling of indirect function calls is covered in sec. 7.3.2.2.3.

7.3.2.2.2 Machine code

Additional information is procured from the output machine code when the target is the ARM Cortex-M architecture. This information is required to draw an accurate call graph when the IR contains *LLVM intrinsics* and instructions that are subject to architecture dependent Instruction Selection (ISel) logic.

LLVM intrinsics look like function calls in the LLVM IR but may be lowered directly to instructions by the LLVM backend depending on the compilation target and the inputs passed to the intrinsic. One example is the family of `@llvm.memset` intrinsics which are used to initialize arrays (blocks of memory); LLVM lowers these intrinsics directly to machine code when the arrays are small but produces a call to the `__aebi_memcpy` or `__aeabi_memcpy4` function when the arrays are large. `__aeabi_memcpy4` is used when the alignment of the array is known to be a multiple of 4; `__aeabi_memcpy` is used in the rest of cases.

Some IR instructions have architecture dependent ISel logic and may lower to function calls when the target instruction set does not contain a dedicated instruction for the operation. One example is the `fadd` instruction which represents the addition of two single-precision floating point numbers. This intrinsic lowers to the `vadd.f32` instruction when the compilation target is `thumbv7em-none-eabihf`, which contains FPU instructions in its instruction set, but lowers to a function call to the function `__aeabi_fadd` for other Cortex-M compilation targets.

Instead of duplicating the LLVM's ISel logic in `cargo-call-stack` to deal with these corner cases the final result of that logic is extracted from the machine code. That is `cargo-call-stack` ignores the calls to intrinsics and IR instructions other than `call` and `invoke` and instead extracts additional function call information from the machine code.

Only information about *direct* function calls is extracted from the machine code. In the ARM Cortex-M architecture these appear as instructions `b $label` and `bl $label` in the machine code. Additional edges are added to the graph from this information.

7.3.2.2.3 Compiler support To reason about indirect function calls the Rust compiler was patched to produce additional LLVM IR level metadata (syntax: `!metadata !0`). This experimental feature is enabled using the unstable `-Z call-metadata` `rustc` flag.

7.3.2.2.3.1 `!"fn"` When the `call-metadata` feature is enabled the compiler adds `!"fn"` type metadata to all the functions definitions of functions that are converted into function pointers and to all the function pointer call sites, `call` / `invoke` instructions. Each metadata node contains the signature of the function being defined / called. This lets `cargo-call-stack` match call sites to the definitions of the potential callees.

Lst. 7.12 shows the IR of lst. 7.6. Only functions `foo` and `bar`, which are coerced into pointers, are given `!"fn"` metadata. The function pointer call site is also given the same `!"fn"` metadata. The metadata node contains the function signature encoded as a string: `"fn() -> u32"`.

Listing 7.12 `!"fn"` metadata in the LLVM IR of lst. 7.6

```
define void @idle() unnamed_addr #1 {
start:
  ; ..
  %2 = tail call i32 %spec.select() #8, !rust !93 ; x()
  ; ..
}

define i32 @foo() unnamed_addr #0 !rust !93 {
  ; ..
}

define i32 @bar() unnamed_addr #0 !rust !93 {
  ; ..
}

define i32 @baz() unnamed_addr #0 {
  ; ..
}

; ..
!93 = !{!"fn", !"fn() -> u32"}
```

As seen in fig. 7.6 `cargo-call-stack` creates one fictitious node for each `!"fn"` metadata nodes and adds incoming edges between the node and the calls sites that have the same metadata and outgoing edges between the node and the potential callees.

7.3.2.2.3.2 `!"dyn"` The `call-metadata` feature also adds `!"dyn"` metadata to each trait method implementation that belongs to a type that is converted into a trait object. This metadata is also added to call sites where dynamic dispatch of a trait method occurs. This metadata node contains the name of the trait and the name of the method being dispatched / defined.

Lst. 7.13 shows the IR of lst. 7.7. Only the `Foo` implementations for `Bar` and `Baz`, which are coerced into `Foo` trait objects, are given `!"dyn"` metadata. The dynamic dispatch site is also given the same `!"dyn"` metadata. The metadata node contains the name of the trait and the trait method, `"Foo"` and `"foo"`, being dispatched.

Listing 7.13 `!"dyn"` metadata in the LLVM IR of lst. 7.7

```
define void @idle() unnamed_addr #2 {
; ..
  ; `to.foo()`
  %4 = tail call zeroext i1 %3({}* %2) #8, !rust !164
; call <app::Quux as app::Foo>::foo
  tail call fastcc void @"(..)"()
; ..
}

; <app::Bar as app::Foo>::foo
define internal zeroext i1 @"(..)"(%Bar*) !rust !164 {
  ; ..
}

; <app::Baz as app::Foo>::foo
define internal zeroext i1 @"(..)"(%Baz*) !rust !164 {
  ; ..
}

; <app::Quux as app::Foo>::foo
define internal fastcc void @"(..)"() {
  ; ..
}

!rust !164 = !{!"dyn", !"Foo", !"foo"}
```

As seen in fig. 7.7 `cargo-call-stack` creates a fictitious node for each `!"dyn"` metadata node and adds incoming edges between the node and the call sites that have the same metadata and outgoing edges between the node and the potential callees.

7.3.2.2.3.3 `!"drop"` Finally, the `call-metadata` feature also adds `!"drop"` metadata information to the LLVM IR. This metadata is attached to the drop glue functions of types that implement traits that are used as trait objects and to the call sites where the drop glue of a trait object is invoked. Each node contains the name of the associated trait.

Lst. 7.14 shows the IR of lst. 7.8. The `drop(x)` call site is annotated with `!"drop"` metadata and so are the drop glue functions of types `Bar` and `Baz`. The metadata is the same in the three places and contains the name of the associated trait: `Foo`.

As seen in fig. 7.8 `cargo-call-stack` creates a fictitious node for each `!"drop"` metadata node and adds incoming edges between the node and the call sites that have the same metadata and outgoing edges between the node and the potential callees.

7.3.2.3 Graph traversal

To compute the worst case stack usage the graph is traversed adding up the stack usage of each individual function.

If the graph is acyclic then the graph is walked node by node in reverse topological order. The worst case stack usage of each node is computed as its local stack usage plus the worst case stack usage of all the

Listing 7.14 `!"drop"` metadata in the LLVM IR of lst. 7.8

```
; Function Attrs: noinline noreturn nounwind
define void @idle() unnamed_addr #1 !dbg !1768 {
; ..
  ; `drop(x)`
  tail call void %2({}* inttoptr (i32 1 to {}*)) #8, !rust !131
; ..
}

; real_drop_in_place::h807aaf8a06cc8e5e
define internal void @"(..)"(%"Bar"*) !rust !131 {
start:
; call <app::Bar as Drop>::drop
  tail call fastcc void @"(..)"()
  ret void
}

; real_drop_in_place::he043dffe2a730da4
define internal void @"(..)"(%"Baz"*) !rust !131 {
start:
; call <app::Baz as Drop>::drop
  tail call fastcc void @"(..)"()
  ret void
}

!131 = !{!"drop", !"Foo"}
```

functions it calls (outgoing neighbors). If any of the nodes has unknown stack usage then the worst case stack usage is still computed as before but reported as a lower bound.

If the graph contains cycles then the Kosaraju algorithm is used to find the strongly connected components (SCC) in the graph. The SCCs are then traversed in reverse chronological order. If the SCC has a single node then the worst case stack usage is computed as in the acyclic case. If the SCC has more than one node and all the nodes in a SCC have zero stack usage then the worst case stack usage of all its members is computed as the sum of the worst case stack usage of all the outgoing SCC neighbors. If the SCC has more than one node and at least one node has a non-zero stack usage then the worst case stack usage of all its members is computed as the sum of the highest local stack usage plus the sum of the worst case stack usages of all its outgoing SCC neighbors but reported as a lower bound.

7.3.3 `core::fmt`

The `core` library provides formatting functionality in its `fmt` module that is widely used in the ecosystem and the standard library itself. This formatting functionality uses several formatting traits, like `Debug` and `Display`, and the built-in `write!` and `print!` families of macros.

Lst. 7.15 depicts the normal usage of the `core::fmt` API. The `{}` specifier in the `print!` macro calls the `Display` implementation on the type `Foo` whereas the `{:?}` specifier calls `Debug` implementation.

Underneath, these macros implement a form of dynamic dispatch that uses neither enumerations or trait objects. The implementation, shown in lst. 7.16, erases the type of a value that implements one of the formatting traits using the unsafe `transmute` operation (line 10) and stores the value and a pointer to its `fmt` method in an `ArgumentV1` value. With the `ArgumentV1` type the implementation can store values that implement *some* formatting trait in an array for further processing. Eventually the implementation invokes the formatting method on the erased type (line 17).

Listing 7.15 Formatting values with `core::fmt`

```rust
#[derive(Debug)]
struct Foo {
    x: i32,
}

impl core::fmt::Display for Foo {
    fn fmt(&self, f: &mut core::fmt::Formatter) -> core::fmt::Result {
        write!(f, "x={}", self.x)
    }
}

fn print() {
    let foo = Foo { x: 42 };
    println!("{}", foo); //   output: "x=42"
    println!("{:?}", foo); // output: "Foo { x: 42 }"
}
```

This type erasure technique bypasses the type system and makes the LLVM IR analysis presented in sec. 7.3.2.2 inaccurate. The analysis sees a function pointer call with `!"fn"` metadata that indicates a signature `fn(&Void, &mut Formatter) -> Result` and this results in an empty list of potential callees. In lst. 7.15, this function pointer call could dispatch one of these two methods `<Foo as Debug>::fmt` or `<Foo as Display>::fmt`.

As this functionality is core to the standard library and widely used `cargo-call-stack` specially handles function pointer calls of the type `fn(&Void, &mut Formatter) -> Result` and connects them to implementations of the formatting traits to produce accurate call graphs. fig. 7.9 shows the call graph produced by `cargo-call-stack` for a program that only calls the expression `printl!("{}", 42u32)` to print a 32-bit integer.

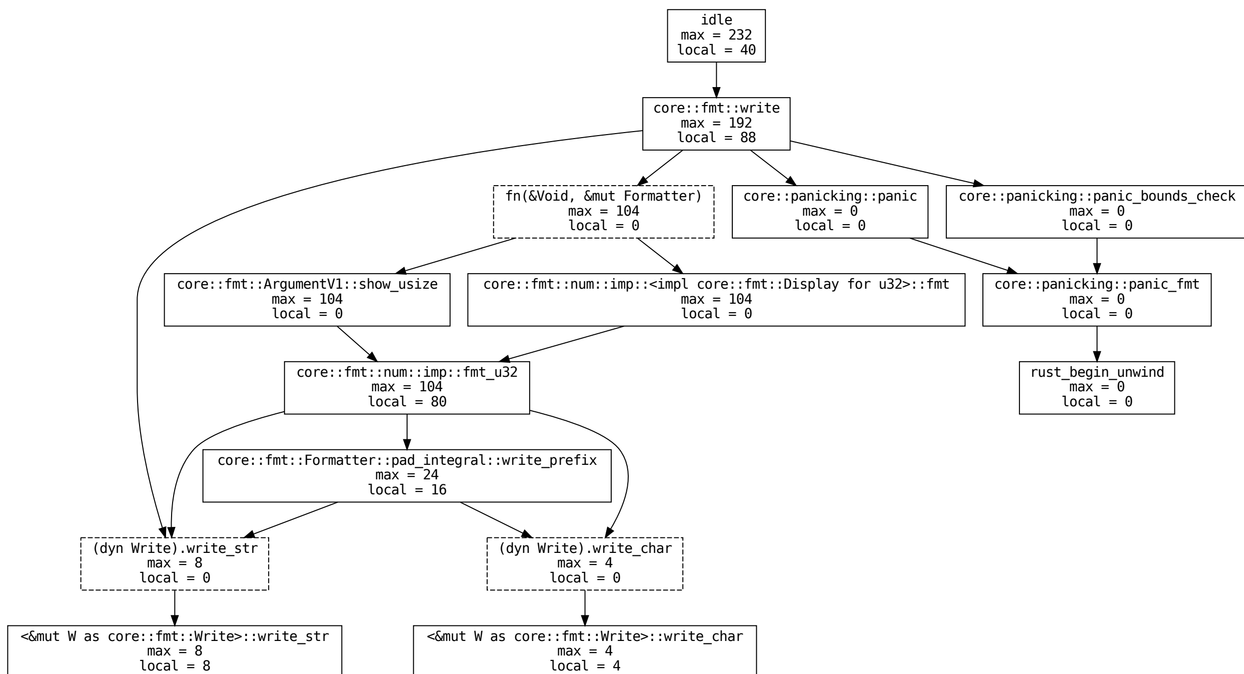


Figure 7.9: Formatting a 32-bit integer

Listing 7.16 implementation details of `core::fmt`

```
struct Void { /* .. */ }

pub struct ArgumentV1<'a> {
    value: &'a Void,
    formatter: fn(&Void, &mut Formatter<'_>) -> Result,
}

impl<'a> ArgumentV1<'a> {
    pub fn new<'b, T>(x: &'a T, f: fn(&T, &mut Formatter<'_>) -> Result) -> Self {
        unsafe { Self { formatter: mem::transmute(f), value: mem::transmute(x) } }
    }
}

pub fn write(args: &[ArgumentV1]) -> Result {
    for arg in args {
        // ..
        (arg.formatter)(arg.value, &mut formatter)?;
    }
    // ..
}
```

7.3.3.1 ufmt

Although `cargo-call-stack` can handle uses of the `core::fmt` machinery, the current implementation of `core::fmt` easily results in cyclic call graphs due to the internal use of recursion, trait objects and presence of panicking branches in the implementation – panic handlers themselves usually format panic messages leading to more recursion.

The result is programs whose worst case stack usage cannot be determined by `cargo-call-stack`. As part of this thesis project an alternative implementation of the `core::fmt` API was developed: `ufmt`, the "micro formatting" library [39]. This alternative implementation does not use trait objects or any other form of dynamic dispatch; it also avoids recursion and have been carefully written to not contain any panicking branch when optimized. Using this alternative library leads to programs whose stack usage can be bounded and also results in faster and smaller binaries.

Lst. 7.17 shows a complex example that uses the `ufmt` library to format two different structures that use three different integer types. Fig. 7.10 shows the call graph of the example; the result is much simpler than the call graph for formatting a single integer using `core::fmt` (shown in fig. 7.9) and only contains direct function calls.

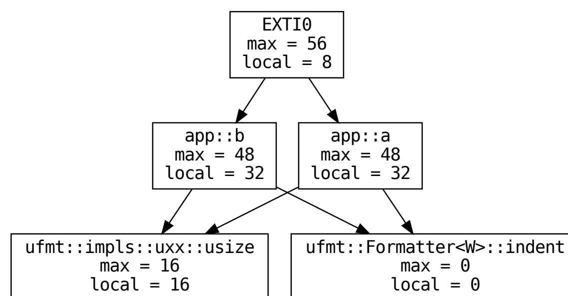


Figure 7.10: Call graph of lst. 7.17

The API in this library does not map one to one to the `core::fmt` API – it uses a different set of formatting traits (`uWrite`, `uDebug`, etc) and different macros (`uwrite!`, `uwriteln!`) – thus non-trivial changes are

Listing 7.17 Formatting structures with `ufmt`

```
#[derive(uDebug)]
struct Foo {
    x: u8,
    y: u16,
}

#[derive(uDebug)]
struct Bar(u32);

#[task]
fn a(cx: a::Context) {
    let x = X.load(Ordering::Relaxed);
    let y = Y.load(Ordering::Relaxed);

    uwriteln!(&mut W, "{:#?}", Foo { x, y });
}

#[task]
fn b(cx: b::Context) {
    uwriteln!(&mut W, "{:#?}", Bar(Z.load(Ordering::Relaxed)));
}
```

required to swap to it but the advantages are clear.

Chapter 8

Conclusions and future work

In this thesis a port of Real Time For the Masses was presented. This alternative to time sliced threads provides several advantages like reduced overhead, efficient memory utilization, predictable performance (constant time overhead), deadlock freedom and suitability for static analysis.

The port was analyzed from the point of view of memory safety to see if the Rust aliasing rule was respected in many corner cases. Additionally, the overhead of the abstractions provided by the framework were studied and confirmed to have bounded constant time overhead, making the implementation suitable for hard real time applications.

In addition to the Rust port of single core RTFM; the API was extended to support multi-core devices and the implementation was tested on both a homogeneous multi-core device and a heterogeneous multi-core device. Likewise, the abstractions in this multi-core implementation were also characterized confirming that the abstractions still have constant time overhead in multi-core setting even in the presence of memory contention between the cores.

Both RTFM APIs have been analyzed for memory safety and confirmed to adhere to Rust aliasing rule, a requirement for machine checked memory safety. Furthermore, RTFM has been successfully validated by several users, both hobbyists and production users.

Finally, `cargo-call-stack`, a tool for whole program static stack usage analysis, was presented. Its different features were explored and it was shown that the tool can deal with the most common embedded Rust programs. Implementation details were also covered, including changes required in the compiler to get more accurate call graphs.

8.1 Future work

Although widely used today, with thousands of downloads in the past 90 days, there is still room for improvement in the Rust port of RTFM.

8.1.1 Shared-exclusive locks

In the current implementation, contexts can only request unique access to a resource. For this kind of access the runtime hands out a unique reference (`&mut-`) to the context either directly or behind a resource proxy.

This enhancement consists of extending the kind of resource access a task can request: shared access and unique access. For this new kind of access, the shared access kind, the runtime would hand out a shared reference (`&-`) to the context.

Having more control over the kind of reference a context can receive would let the user reduce the number of locks required to access resources. Lst. 8.1 is an hypothetical example where using shared access elides

Listing 8.1 Using shared access to avoid a critical section

```
struct Resources { x: i64 }

#[task(priority = 1, resources = [x])]
fn a(cx: a::Context) {
    cx.resources.x.lock(|x: &mut i64| { /* .. */ });
}

#[task(priority = 2, resources = [&x])]
fn b(cx: b::Context) {
    let x: &i64 = cx.resources.x;
}

#[task(priority = 3, resources = [&x])]
fn c(cx: c::Context) {
    let x: &i64 = cx.resources.x;
}
```

Listing 8.2 `lock_shared` API

```
struct Resources { x: i64 }

#[task(priority = 1, resources = [&x])]
fn a(cx: a::Context) {
    cx.resources.x.lock_shared(|x: &i64| { /* .. */ });
}

#[task(priority = 2, resources = [x])]
fn b(cx: b::Context) {
    let x: &mut i64 = cx.resources.x;
}

trait LockShared {
    type T;
    fn lock_shared<R>(&self, impl FnOnce(&Self::T) -> R) -> R;
}
```

a critical section in the mid-priority task (line 10). In this made up example the `&x` syntax (line 8) in the `resources` list is used to request shared access; the `x` syntax (line 3) continues to refer to a request for unique access. The critical section is not needed in `b` because the other task that can access the resource concurrently, `c`, also accesses the resource through a shared reference (`&-`). `c` preempting the task `b` results in two shared references to `x` existing at the same time – this is allowed by the Rust aliasing rule. Task `a`, on the other hand, requires unique access (`&mut-`) to the resource so it needs to block both `b` and `c` to maintain the Rust aliasing rule.

Supporting shared access to resources would require adding a second kind of critical section API. This new API would resemble the `lock` API but would hand over a shared reference (`&-`) instead of a unique reference (`&mut-`). Lst. 8.2 sketches out the potential API.

More studies are required to determine the exact set of `Sync` and `Send` requirements that resources accessed through shared-exclusive locks would need to fulfill to avoid memory unsafety.

8.1.2 Canceling and rescheduling tasks

Several use cases would benefit from having a dedicated API to cancel previously scheduled tasks and

Listing 8.3 Task cancellation API

```
#[task(schedule = [b])]
fn a(cx: a::Context) {
    let handle = a.schedule.b(instant1, input1).unwrap();
    a.schedule.b(instant2, input2).unwrap();
    // ..
    if some_condition {
        handle.cancel();
    } else if other_condition {
        handle.reschedule(instant3);
    }
}
```

reschedule them. Lst. 8.3 sketches out a potential API. Under this proposal the current `schedule` API would return a message handle (line 3) that provides a `cancel` method that cancels *that* particular message (line 7) and a `reschedule` method (line 9) that changes its scheduled execution time. Note that the message scheduled in line 4 is impossible to cancel or reschedule because its handle is immediately discarded. An implementation of this API would need to be careful about calls to `cancel` or `reschedule` on messages that have already been dispatched as to not cancel or reschedule a different message.

8.1.3 Automatic capacity selection

Listing 8.4 Application with timing information and automatic capacity selection

```
#[task(binds = A, priority = 3, spawn = [c], inter-arrival = 11, wcet = 1)]
fn a(cx: a::Context) {
    let c: spawn::c = cx.spawn.c;
    c.send(0);
    // c.send(1); //~ error: use of moved value `c`
}

#[task(binds = B, priority = 2, spawn = [c], inter-arrival = 29, wcet = 1)]
fn b(cx: b::Context) {
    cx.spawn.c.send(2);
}

#[task(priority = 1, wcet = 5)] // capacity = 2
fn c(cx: c::Context, input: i32) {
    // ..
}
```

Both the `spawn` and `schedule` API can fail if there is not sufficient space in the spawnee / schedulee message buffer. This situation can arise in practice if the user does not select a sufficiently large message buffer capacity.

Timing analysis could remove the need for manually selecting task capacities and the runtime checks in the `spawn` and `schedule` APIs. The analysis would require user input like the worst case inter-arrival time of hardware tasks and the worst case execution time (WCET) of tasks (see [40] for a fully automated approach to WCET estimation); though the latter could potentially be computed at compile time in some scenarios.

In addition to this analysis it would be necessary to limit how many messages a context can post. This can be achieved using a proxy with move semantics: instead of methods on a `spawn struct` each spawn-able task would be a separate `struct` with a single `fn send(self)` method that consumes the `struct` when used.

Lst. 8.4 sketches out potential syntax for providing timing metadata. In this example the `spawn` API has

been modified to use move semantics (line 4) limiting the number of messages each context can post to a single one (line 5). The updated `spawn` API is infallible: it does not return a `Result` or contain runtime checks; this is possible because the provided timing information was used to select the capacity of the task `c` message buffer – selected to be `2` in this example – and the system was deemed schedulable.

8.1.4 Schedulability and security

Real-time semantics for the IEC 61499 standard, an industrial automation standard for specifying distributed control systems, were developed in [41] by mapping the function block architecture defined in the standard to RTFM tasks and resources. As part of that work, rtfm-lang gained timing semantics: `after` and `before` timing information that define a permissible execution window. Those semantics have yet to be implemented in the Rust port of RTFM but are required to reason about the schedulability of RTFM applications.

The topic of encoding security in RTFM Rust was explored in [42]. In that work an alternative implementation of RTFM based on reactive objects (not presented in this document) was augmented with *secure containers*, data subject to strict access control, and encrypted containers. This alternative implementation was not pursued due to its high implementation and public API complexity, which made difficult to analyze its memory safety. However, the idea of encoding security or assisting the implementation of secure applications should be revisited in the current implementation of RTFM Rust.

8.1.5 Stack analysis of RTFM applications

Although it is possible today to use `cargo-call-stack` to analyze the stack usage of a RTFM application the output does not provide a complete picture: the call graph reported by `cargo-call-stack` contains many disconnected subgraphs, one for each hardware task and each task dispatcher. What the graph does not show is that these tasks can preempt each other.

To compute the overall maximum stack usage of an RTFM application one would need to manually add the worst case stack usage of each priority level. To automate this process `cargo-call-stack` would need to cooperate with RTFM to identify tasks and their priorities; with this information the tool would be able to connect the subgraphs of hardware tasks and task dispatchers and more readily provide the maximum stack usage of the application.

8.1.6 A RTFM organization

We'd like to create an organization around the Real Time For the Masses framework to ensure the continuation of its development. The organization would let other software developers contribute with code and let us receive more structured feedback from users (stakeholders). These two would let us establish a roadmap where new features can be prioritized according to user feedback.

References

[1] Ericsson, "Internet of things forecast." https://www.ericsson.com/en/mobility-report/internet-of-things-forecast.

[2] P. Lindgren, "The original real-time for the masses." http://www.rtfm-lang.org/.

[3] J. Eriksson, F. Häggström, S. Aittamaa, A. Kruglyak, and P. Lindgren, "Real-time for the masses, step 1: Programming api and static priority srp kernel primitives," in *2013 8th ieee international symposium on industrial embedded systems (sies)*, 2013, pp. 110–113.

[4] P. Lindgren, M. Lindner, A. Lindner, D. Pereira, and L. M. Pinho, "RTFM-core: Language and implementation," in *2015 ieee 10th conference on industrial electronics and applications (iciea)*, 2015, pp. 990–995.

[5] P. Lindgren, E. Fresk, M. Lindner, A. Lindner, D. Pereira, and L. M. Pinho, "Abstract timers and their implementation onto the arm cortex-m family of mcus," *ACM SIGBED Review*, vol. 13, no. 1, pp. 48–53, 2016.

[6] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamari, and L. Ryzhyk, "System programming in rust: Beyond safety," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 1, pp. 94–99, 2017.

[7] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[8] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Proceedings. Real-time systems symposium*, 1988, pp. 259–269.

[9] P. Gai, G. Lipari, and M. Di Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Proceedings 22nd ieee real-time systems symposium (rtss 2001)(Cat. No. 01PR1420)*, 2001, pp. 73–83.

[10] A. Burns and A. J. Wellings, "A schedulability compatible multiprocessor resource sharing protocol–mrsp," in *2013 25th euromicro conference on real-time systems*, 2013, pp. 282–291.

[11] P. Lindgren, D. Pereira, J. Eriksson, M. Lindner, and L. Miguel, "Rtfm-lang static semantics for systems with mixed criticality," in *Ada user journal, proc of workshop on mixed criticality for industrial systems (wmcis'2014)*, 2014, vol. 35, pp. 128–132.

[12] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in cyclone," in *ACM sigplan notices*, 2002, vol. 37, pp. 282–293.

[13] The Rust developers, "Rust's send trait." https://doc.rust-lang.org/1.37.0/std/marker/trait.Send.html.

[14] The Rust developers, "Rust's sync trait." https://doc.rust-lang.org/1.37.0/std/marker/trait.Sync.html.

[15] The Rust developers, "Rust's arc, atomically reference counted, type." https://doc.rust-lang.org/1.37.0/std/sync/struct.Arc.html.

[16] The Rust developers, "Rust's thread spawning api." https://doc.rust-lang.org/1.37.0/std/thread/fn.spawn.html.

[17] The Rust developers, "Rust's rc, reference counted, type." https://doc.rust-lang.org/1.37.0/std/rc/struct.Rc.html.

[18] J. Aparicio, "A memory pool abstraction." https://docs.rs/heapless/0.5.0/heapless/pool/index.html.

[19] The Rust developers, "Rust's mutex type." https://doc.rust-lang.org/1.37.0/std/sync/struct.Mutex.html.

[20] J. Aparicio, "The rtfm book." https://rtfm.rs/0.5.

[21] J. Aparicio, "The rtfm api reference." https://rtfm.rs/0.5/api.

[22] J. Aparicio, "Rust port of real time for the masses for arm cortex-m microcontrollers." https://github.com/rtfm-rs/cortex-m-rtfm.

[23] J. Aparicio, "Rust port of real time for the masses for the lpc55s69 (2x cortex-m33)." https://github.com/japaric/lpcxpresso55S69.

[24] J. Aparicio, "Rust port of real time for the masses for the lpc54114 (cortex-m4 + cortex-m0+)." https://github.com/japaric/lpcxpresso54114.

[25] J. Aparicio, "Rust port of real time for the masses for the zynq ultrascale+ (2x arm cortex-r cores)." https://github.com/japaric/ultrascale-plus.

[26] J. Aparicio, "Rust port of real time for the masses for the hifive1 (riscv)." https://github.com/japaric/hifive1.

[27] J. Aparicio, "Rust port of real time for the masses for linux." https://github.com/japaric/linux-rtfm.

[28] A. Lindner, M. Lindner, and P. Lindgren, "RTFM-rt: A threaded runtime for rtfm-core-towards execution of iec 61499," in *2015 ieee 20th conference on emerging technologies & factory automation (etfa)*, 2015, pp. 1–8.

[29] The Embedded Rust Working Group, "A minimal runtime for arm cortex-m microcontrollers." https://crates.io/crates/cortex-m-rt.

[30] ARM, "ARM cortex-m programming guide to memory barrier instructions." 2012.

[31] J. A. Rivera, M. Lindner, and P. Lindgren, "Heapless: Dynamic data structures without dynamic heap allocator for rust," in *2018 ieee 16th international conference on industrial informatics (indin)*, 2018, pp. 87–94.

[32] ARM, "What is the true interrupt latency of cortex-m3 and cortex-m4 for interrupt entry and exit?" http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka16366.html.

[33] D. Vyukov, "Bounded mpmc queue." http://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue.

[34] J. Aparicio, "Rust port of real time for the masses for arm cortex-m microcontrollers." https://crates.io/crates/cortex-m-rtfm.

[35] G. P., "An alarm clock that also displays pressure, temperature and humidity." https://github.com/TeXitoi/rusty-clock.

[36] G. P., "A hand wired ortholinear mechanical keyboard with firmware in rust." https://github.com/TeXitoi/keyberon.

[37] M. Lindner, A. Lindner, and P. Lindgren, "Safe tasks: Run time verification of the rtfm-lang model of computation," in *2016 ieee 21st international conference on emerging technologies and factory automation (etfa)*, 2016, pp. 1–8.

[38] The LLVM developers, "LLVM: Emitting function stack size information." https://llvm.org/docs/CodeGenerator.html#emitting-function-stack-size-information.

[39] J. Aparicio, "Ufmt, the micro formatting library." https://crates.io/crates/ufmt.

[40] M. Lindner, J. Aparicio, H. Tjader, P. Lindgren, and J. Eriksson, "Hardware-in-the-loop based wcet analysis with klee," in *2018 ieee 23rd international conference on emerging technologies and factory automation (etfa)*, 2018, vol. 1, pp. 345–352.

[41] P. Lindgren, M. Lindner, A. Lindner, V. Vyatkin, D. Pereira, and L. M. Pinho, "A real-time semantics for the iec 61499 standard," in *2015 ieee 20th conference on emerging technologies & factory automation (etfa)*, 2015, pp. 1–6.

[42] M. Lindner, J. Aparicio, and P. Lindgren, "Concurrent reactive objects in rust secure by construction," *Ada User Journal*, 2018.