MARC BRINKMANN

SPECS OF RUST

SPECS OF RUST

MARC BRINKMANN

Karlsruhe Institute of Technology

Safe embedded software development through compile-time verification of hardware resource access using device trees and ownership semantics

Institute for Applied Computer Science
Faculty of Informatics
KIT

July 2016

ABSTRACT

With microcontroller units becoming more and more prevalent in all sorts of systems and an increasing number of small and affordable devices being connected to the internet, embedded development is becoming even more prominent every day. However, its performance-driven dependency on unsafe but popular languages like C and C++ has resulted in security and safety challenges that require practical solutions that are usable today.

The Rust programming language challenges the notion that safety, powerful abstractions and automatic memory management prohibit excellent performance in environments where overhead is expensive. This thesis demonstrates that development without C or assembly code is possible without sacrificing characteristic performance and develops a safe hardware abstraction that utilizes Rust's type system to prevent accidental misuse of resources and access conflicts. Additionally it demonstrates that these abstractions can be automatically generated from already available hardware specifications in the form of Device Trees.

ACKNOWLEDGMENTS

The Rust community has a reputation for being friendly and helpful; it is well deserved.

The community and core team can usually be found in `#rust` *on* `irc.mozilla.org`*.*

Thank you Alex K. and Alex M. for supplying all the electrical engineering knowledge that I lacked and the gentle ridicule when appropriate — I promise to pay you back in kind.

A big thanks to Dipl. inf. Oliver Schneider for being a motivated, patient and all-around great advisor and Prof. Dr. Veit Hagenmeyer at the Institute for Applied Computer Science for letting me work on something so "exotic" in the first place.

Finally, thank you, Nicole — well beyond these humble pages.

CONTENTS

LIST OF FIGURES

ACRONYMS

API	application programming interface
ABI	application binary interface
AHB	Advanced High-performance Bus
ARM	Advanced RISC Machine
libcore	Rust core library
CMSIS	Cortex microcontroller software interface standard
CPU	central processing unit
CRC32	cycling redundancy check
DOD	Department of Defense
DT	Device Tree
dtc	Device Tree compiler
dtb	flattened Device Tree
dts	Device Tree source file
ELF	Executable and Linkable Format
FFI	foreign function interface
GCC	GNU Compiler Collection
GPIO	general purpose input/output
IC	integrated circuit
IDE	integrated development environment
IoT	Internet of Things
ISA	instruction set architecture
ISR	interrupt service routine
IVT	interrupt vector table
JSON	Javascript Object Notation
JTAG	Joint Test Action Group
KSLOC	thousand sources lines of code

LED	light-emitting diode
LLVM	the LLVM project
LR	link register
MachO	Mach object
MB	megabytes
MCU	microcontroller unit
MHz	megahertz
ODR	output data register
OF	Open Firmware
OS	Operating system
PE	Portable Executable
PC	program counter
PCB	printed circuit board
PID	proportional-integral-derivative
POD	plain old data
PLC	programmable logic controller
PPC	PowerPC microarchitecture
RCC	reset and clock control
ROM	read-only memory
stdlib	Rust standard library
SP	stack pointer
SVD	system view description
SWD	serial-wire debug
TCP/IP	Transmission Control Protocol / Internet Protocol
TFT	thin-film transistor
USB	Universal Serial Bus

1 INTRODUCTION

Occasionally a step forward can result in unintended side-effects when legacy systems are unable to keep up with the changing environment. An interesting area to witness this effect in is the rapidly changing embedded development landscape: Firmware or other software, often placed in read-only memory (ROM) of devices with comparably limited yet significant processing power is — sometimes inadvertently — exposed to the world. Before the advent of ubiquitous connectivity the programs running on these devices could reasonably expected to be left alone during their lifetime, with no exposed network or internet connections and special hardware needed to access their non-volatile memory.

During the recent years, this has changed: Small computing devices have gotten more powerful whilst getting cheaper and components come with commonly available access ports for flashing firmware updates, increasing the pace and availability of development. Devices are connected to networks and through them, the internet as well. All this has increased software complexity: In 1969, about 30,000 lines of pure assembly were enough to get a manned spacecraft to the moon — and back [16] — while the over 300,000 lines of C tasked with encrypting a Transmission Control Protocol / Internet Protocol (TCP/IP) connection [23] contained a bug that made an estimated 25%-50% of public websites susceptible to abuse [14].

The Apollo 11 source code has recently been transcribed and published electronically [16].

In the not too distant field of automotive technology, internet connected cars have become a reality and already are being successfully attacked, becoming dangerous to their owners [20]. The range of attackable devices broadens, from the ill-conceived Internet of Things (IoT) light bulb [15] that routes its on-off-function through a server in China to the highly sophisticated "Stuxnet" attack on Iran's nuclear program. The former demonstrates that there is no technical solution to secure a design that is flawed from the start, the latter leveraged four different security holes on Microsoft Windows systems. These included obvious design flaws like using cycling redundancy check (CRC32) where a cryptographically secure hash function was required as well as "classic" memory management failures [21]. Legacy systems are even more at risk because today's threat-model did not exist when these devices were developed: Brüggemann and Spenneberg [8] and Brüggemann [7] demonstrated this on wide-spread Siemens programmable logic controllers (PLCs) found in production use by creating a self-propagating worm abusing the limited "engineer-friendly" programming language available on these devices.

Solving the problem with Rust

The increased exposure of firmwares to potentially hostile environments requires both a new mentality and new tools to tackle this issue. One issue that cannot be solved is a simple lack of understanding of good security practices, a broken design on a higher level cannot be fixed by the implementation further down. On small computing devices that often run without any operating system at all, traditionally memory-unsafe languages like C and C++ have played a dominant role in implementations [10]. Larger programs using these languages almost inevitably end up with bugs that, due to the manual memory management model employed, can often be escalated into security issues.

To alleviate these issues a new language called *Rust* was developed by Mozilla Research [22]. Rust solves the security issues caused by memory management issues by design — unless explicitly desired through a keyword aptly named `unsafe`, it is not possible to create bugs of this class that so often plague implementations in the other languages mentioned.

Modern C++ 11/14 supports some of these constructs but cannot enforce their use.

Eliminating these issues is hardly an exclusive feature, any language that offers automatic memory management usually avoids them. What sets Rust apart is that these guarantees are given at compile time, without any overhead, through its ownership model of memory management. C and C++ are commonly chosen for being "close to the machine", that is their ability to write programs with no overhead and full control, an important feature when running on very limited hardware. Rust shares this characteristic while still living up to its high expectations of memory safety, making it a good choice for embedded development.

1.0.1 *More than security: Safety*

Convincing programmers and project leads to change the core implementation language away from "tried-and-true" C is no easy task, especially when security concerns are often given only afterthoughts, as they do not make out marketable features. In addition to being more secure by being memory safe, the abstractions in Rust lend themselves to better safety overall; Rust possesses excellent capabilities to avoid concurrent memory access failures and some race conditions in parallel programming, which is explored in-depth in [4].

With a more powerful type system than C++ and especially C, better abstractions can be built and more programming errors can be caught at compile time. Surprisingly, once the abstractions are built this can result in a notable increase in productivity: After successful compilation many programs run remarkably error free. This thesis explores how these abstractions can be used to model restrictions imposed by the embedded hardware used, utilizing Rust's memory ownership and type system to improve safety as well as ergonomics while programming. Furthermore it will show that, while the learning curve for Rust is steeper, the end result will not have any drawbacks in performance compared to an equivalent C program.

1.0.2 *About this thesis*

Besides fulfilling an obvious academic requirement, this thesis is intended to be of use for two groups: Engineers with a background in electrical engineering that develop firmwares in C and C++, who may not have the same amount of software engineering experience as their software-developing counterparts and the latter, who may up until this point not have touched a platform outside their own range of desktop- or server operating systems. For this reason introductions from both viewpoints are included.

Chapter 2 will introduce the Rust programming language and its core mechanisms that are relevant for the systems developed in this thesis.

Chapter 3 contains a brief overview of the basic differences when developing on embedded devices and introduce one of these platforms.

Chapter 4 demonstrates how the problem at hand is usually solved when using C and how the process translates into Rust.

Chapter 5 develops a more idiomatic approach for these solutions, gaining additionally safety and convenience through Rust's language features.

Chapter 6 is about a practical method of implementing the previously given system automatically and shifting the burden of correct hardware description back to the hardware vendor.

2

THE RUST PROGRAMMING LANGUAGE

Rust [32] is a modern programming language intended as a systems programming language, a task that currently is overwhelmingly handled by languages like C and C++ [9, 10].

2.1 SYSTEMS PROGRAMMING LANGUAGE

While the definition of "systems programming language" varies, in this thesis it will be roughly defined as a language that is high-level but produces code that is almost as fast and small as an equivalent hand-written assembly program would be.

Defining systems programming in this way yields a set of practical implications. First, requiring a runtime is usually prohibitive, as it increases the resulting binary dramatically or must be shipped separately. When dealing with embedded systems, it is not uncommon to find program memory measured in kilobytes or bytes instead of mega- or gigabytes; being able to to create the smallest program possible is an actual requirement.

The word "runtime" is used here somewhat loosely. Arguably even C requires a tiny runtime that initializes a section of memory prior to execution.

Not being able to ship a large runtime often precludes interpreted languages or efficient garbage collectors as well as virtual machines whose bytecode must be compiled or interpreted. While there are plenty of examples of Java or other languages running on embedded devices, the smaller the device gets in terms of processing power or memory, the more likely it is to find that the only currently available option presents itself in the form of a C compiler.

While not the focus of this work, it should be noted that on the more powerful end of the spectrum, systems programming languages are also in demand for kernels, high-performance computing or video games, where short latencies, real-time support or low overhead hardware access may be desirable.

2.1.1 *A note about Ada*

Specifically in the field of embedded and real-time development [9], the Ada language has some traction, being at the center of the United States Department of Defense (DOD) strategy since the mid-1970s [11] and should not be omitted here. Military and aerospace programming puts a large emphasis on being correct and free of errors compared to end-user or other software and Ada has been named a "competitive advantage" [11] over C/C++ and Java for the DOD, for example eliminating 85% of defects per thousand sources lines of code (KSLOC) (Rust did not exist when the report was written).

While Ada did provide some of the safety features that are the core goals of Rust, the Committee on the Past and Present Contexts for the Use of Ada in the Department of Defense and Computer Science and Telecommunications Board and National Research Council [11] notes that there are issues with adoption, due to low commercial awareness, a lack of academic instruction in Ada, a limited ecosystem of tools and few commercial, non-military career opportunities.

While these issues have been a problem for Ada for at least 20 years, they are shared by Rust by virtue of being a language that has reached a stable 1.0 version little over a year ago [4]. The large difference in syntax [9] of Ada from the C language family and Java is a contributing factor as well. In contrast, Rust is deliberately designed to be familiar, being described as "not a particularly original language" [33].

Rust improves on some of Ada's other shortcomings: A runtime that at worst assumes complete control over the hardware [11] is not a challenge that Rust needs to overcome. Both Ada and Rust possess excellent capabilities for calling C code as well as being called from C itself [33].

2.2 A BRIEF INTRODUCTION

The example code has been taken from a real PID controller implementation that, among other things, drives 130 kW electrical motors [5].

Listing 1 shows a short function from an implementation of a PID controller. The update function is responsible for updating a `struct`, which is similar to a C structure, that holds the internal state of the controller. Periodically, update is called with current process values and time difference; the full implementation is available in Brinkmann [5]. More background information can be found inside Åström and Hägglund [2], chapter 3 and is outside the scope of this thesis.

Rust syntax in general is recognizable to those that have experience with the C language family, including the familiar curly braces, using lowercase identifiers, sharing many keywords and operators, C++-style comments and other features. Some key differences include the different order in function signatures and variable declarations (name–type, instead of type–name) and the omittance of parenthesis and semi-colons in some places.

Binaries of compiled Rust code run without a runtime (although by default, a memory allocator is statically linked into each binary; this can be changed for individual programs) and the toolchain supports common output binary formats such as Executable and Linkable Format (ELF), Portable Executable (PE) or Mach object (MachO).

For programs running directly on hardware (without an Operating system (OS), often called "bare metal") the binary program can be extracted from the ELF binary and flashed or run with a hardware-specific tool. The process after binary compilation is indistinguishable from one for an executable compiled from C code.

The following section gives a short overview of the syntactic language features relevant to the example code listing 1 and what is of interest in this thesis. A more complete primer can be found in Beingessner [4], a complete course is available through the official documentation [29].

2.2.1 *Syntax*

Basic control structures

Rust shares control structures `if`, `else`, `while`, `return` with C [4], a notable difference is that no parenthesis are required around the condition. Arithmetic, comparison and assignment operators behave the same as well.

C-style `for`-loops containing an initialization statement, condition expression and increment statement are not present in the language, eliminating many off-by-one errors by design. Instead, for-each style loops are used instead:

```rust
for item in iterator {
    // ...
}
```

Ranges are supported using the `..` operator:

```rust
for n in 1..100 {
    // n will range from 1 to 100, excluding 100
}
```

Commonly used "endless" loops (`for(;;)` or `while(1)` in C) are directly expressed by `loop`:

```rust
loop {
    // will loop until explicit break
}
```

Many control structures in Rust are expressions, each block has a value equal to the last expression it contains. For example the inline `if` expression

Figure 1: Real-world example: Update function of a software PID controller

```
fn update(&mut self, value: f64, delta_t: f64) -> f64 {
    let error = self.target - value;

    // PROPORTIONAL
    let p_term = self.p_gain * error;

    // INTEGRAL
    self.err_sum = util::limit_range(
        self.i_min, self.i_max,
        self.err_sum + self.i_gain * error * delta_t
    );
    let i_term = self.err_sum;

    // DIFFERENTIAL
    let d_term = if self.prev_value.is_nan()
                 || self.prev_error.is_nan() {
        // no previous values, skip the derivative calculation
        0.0
    } else {
        match self.d_mode {
            DerivativeMode::OnMeasurement => {
                // we use -delta_v instead of delta_error
                // to reduce "derivative kick",
                self.d_gain * (self.prev_value - value) / delta_t
            },
            DerivativeMode::OnError => {
                self.d_gain * (error - self.prev_error) / delta_t
            }
        }
    };

    // store previous values
    self.prev_value = value;
    self.prev_error = error;

    util::limit_range(
        self.out_min, self.out_max,
        p_term + d_term + i_term
    )
}
```

```
let result = if some_condition { 0.0 }
             else { 1.0 };
```

is equivalent to

```
let result;

if some_condition {
  result = 0.0;
} else {
  result = 1.0;
}
```

Variable declarations

Variable declaration is notably different, requiring a `let` statement and being immutable by default, unless a `mut` qualifier is added:

```
let error = self.target - value;
```

Here, `error` is the immutable result of the subtraction of `value` from `self.target`. Of note here is that Rust performs type inference: Since `self.target` and `value` are both 64-bit floating point numbers (`f64`) the resulting `error` is also of type `f64`.

Optionally, variables can be annotated with the type, by prefixing the variable name with a colon:

```
let error: f64 = self.target - value;
```

Constants

Constants are declared using `const` instead of `let`, the type annotation is not optional but mandatory and the identifier must be uppercase:

Constants are not to be confused with immutable variables - the latter cannot exist the global scope.

```
const MAGIC_NUMBER: u32 = 7;
```

Primitive types

Rust primitive types include integer types `i8`, `i16`, `i32`, `i64` as well as their unsigned counterparts `u8`, `u16`, `u32`, `u64`. Booleans use the `bool` type, while 32-bit and 64-bit floating point numbers are represented by the types `f32` and `f64`. Additionally, two types exist to represent an integer that is the "native" size of the target architecture: `isize` and `usize` [29].

Static typing is very strict and less lenient than the typical C compiler. The following snippet will cause the compiler not to report a warning but an actual error:

```
let some_condition: bool = 1;  // error!
```

Acceptable values for a `bool` are only `true` and `false`.

Compound data types

Rust knows arrays and tuples, the former being continuous regions of memory with a fixed length of a single type, while tuples are fixed-size constructs of different, but also non-varying types. Array syntax deviates a bit from that of C:

```
let three_fifties: [u32; 3] = [50; 3];
```

Arrays do not need to be terminated as the arrays length is encoded in its type.

The example above declares an array of three unsigned 32-bit integers, then assigns it a value of `50, 50, 50`.

Similar to C's `struct` is Rust's own `struct` type:

```
struct Point2D {
  x: f32,
  y: f32,
}
```

Struct members can be accessed using the familiar dot-notation: If `p` is an instance of `Point2D` then `p.x` accesses its field `x`.

`enum`s provide Rust with algebraic data types. The name is slightly misleading due to the closeness to C's `enum` — the former are tagged unions [4], allowing the creation of new types by specifying a type that allows values from a set of *variants*:

```
enum Shape {
    Point(Point2D),
    Circle {
        position: Point2D,
        radius: f32,
    },
    Rectangle(Point2D, f32, f32),
}
```

The example defines three *variants*: `Point`, which contains a single struct `Point2D` defined above, `Circle`, containing an anonymous struct with two fields for the circle's position and radius and `Rectangle`, containing a three-tuple, presumably for the rectangle's position, width and height.

These variants are the only permissible values for variables of type `Shape`:

```
let p = Point2D {
    x: 1.5,
    y: 2,
};

let p2 = Point2D {
    x: 2.5,
    y: 3,
}

let mut current_shape = Shape::Point(p1);

// note that current_shape can be assigned a new value because
// it was declared mut
current_shape = Shape::Circle {
  position: p2,
  radius: 4.5,
};
```

Enums can also be used like their C namesakes, as a way of specifying names for constants:

```
enum Color {
  Red,
  Green,
  Blue,
}
```

In this simpler case the code produced by the compiler will not contain any tagging overhead; it will be as efficient as its C equivalent with the added benefit of stricter type checking.

Pattern matching

Pattern matching in Rust allows matching on values or variants and can be performed by a `match` expression:

`let` statements and some other expressions also allow pattern matching.

```
let area = match current_shape {
    Shape::Point(_) => 0,
    Shape::Circle {radius, ..} => pi * radius * radius,
    Shape::Rectangle(_, w, h) => w * h,
};
```

Fields can either be ignored using an underscore (_) or moved out (see section 2.3) of the structure. A `match` block requires that all possible variants of an enum are handled, otherwise the compiler will report an error: It would be impossible to omit a match handling `Shape::Rectangle` in the example above.

Functions

Functions in Rust are declared using the `fn` keyword:

```
fn add(a: u32, b: u32) -> u32 {
    a + b
}
```

Arguments are declared using the same syntax as `let` statements, with type annotations following being mandatory. The return type of a function follows after the arrow (`->`).

While `return` can be used to exit from a function, it is not necessary to actually return a value: The last expression inside the function block is also its value.

A function can be declared as *divergent*, indicating that it will not ever return, by setting the return type to `!`. An example of a divergent function is one containing an infinite loop.

References

References are denoted using an ampersand (&), which is also the symbol used for the address-of operator [4]. The familiar * operator is used to dereference values again:

```
let x: u32   = 5;   // x is an unsigned 32-bit integer
let y: &u32  = &x;  // y is a reference to x
let z: u32   = *y;  // z now contains a copy of the value of x
```

Like variables, references are immutable by default, but can explicitly be created mutable:

```
let mut x: u32      = 5;
let     y: &mut u32 = &mut x;
*y = 6;

// x is now 6.
```

Noteworthy is that, while y is commonly called a “mutable reference”, y itself is not mutable — while it is possible to mutate whatever it is that y points to, one cannot change where y points.

References are similar to pointers but differ in key aspects. This topic will be treated with more detail in sections 2.3 and 2.4.1.

Implementations

Rust allows “attaching” functions to a specific struct using an `impl` block; somewhat similar but not exactly like object-orientation found in C++ and Java:

```
struct PIDController {
    // ...
}

impl PIDController {
    fn update(&mut self, value: f64, delta_t: f64) -> f64 {
        // ...
    }
}
```

An `impl` block contains any number of functions associated with a specific type. This by itself does not change the semantics in any way [4], but provides organizational convenience. The first argument differs in that it may optionally be an explicit self-reference given in a slightly different syntax. A dot-expression `.` can be used to call implementation functions, causing a reference to be passed automatically:

```
// ‘self‘ will be a mutable reference to pid_ctrl inside update
let result = pid_ctrl.update(val, dt);
```

A dot-expression also automatically dereferences, inside `update`, `self.p_gain` will first dereference `self` before looking up the field `p_gain`.

Using `.`, an offset from the structure p to access the field will be calculated at compile time.

Traits

Traits are Rust’s way of implementing interfaces [4]. A trait defines a set of functions and an `impl` block is used to define these for a specific type:

```
trait HasArea {
    fn area(&self) -> f64;
}
```

```
// The ‘HasArea‘ trait can now be implemented by the ‘Shape‘ type:
impl HasArea for Shape {
    fn area(&self) -> f64 {
        match *self {
            // ...
        }
    }
}
```

Generics

impl, fn, struct, enum and trait may be generic over arbitrary types [4], implementing a subset of the functionality of C++ templates while being a little more well-formed: A generic item is type-checked at its definition site, not only on instantiation.

```
// Defines a new pair type of exactly two values of the same type.
struct<T> Pair {
    fst: T,
    snd: T,
}
```

Namespaces and modules

The :: operator is used to access different namespaces. A type's variants can be accessed using their fully qualified name TypeName::VariantName, e.g. Shape::Circle.

Rust also supports modules, which need to be declared using the mod keyword in the top level source file. Modules are mapped 1:1 to files, naming is enforced — a module named foo must be implemented in a file called foo.rs.

The use keyword brings a module from a different crate module back into scope and can optionally import structures or functions:

```
use foo::Foo;

// can now use the Foo struct from the module foo, implemented in foo.rs
```

Panics

While a later addition made panics recoverable they are not intended to be used like exceptions [13].

While Rust does not support exceptions, it implements a similar mechanism called *panics*. A key difference between the two is that it is not uncommon to catch an exception but recovering from a panic is reserved for rare corner cases. Panics can be incited by the panic! macro (see 2.6) or happen “naturally” on arithmetic overflows in debug mode, division by zero or violations caught by bounds-checking.

A panic is a controlled abort of a thread that tries to clean up resources by calling destructors before exiting [29].

The foreign function interface

The Rust foreign function interface (FFI) allows calling C functions and being called from C functions as well [29].

Any language that allows C extensions can interface with Rust.

The `extern` keyword marks a function or global variable as external, to be linked by the linker after compilation. Another important aspect are the `#[no_mangle]` and `#[repr(C)]` attributes (see section 2.4.2) that disable name mangling and enforce C-compatible memory layout.

With name mangling enabled, Rust will extend the final function name to avoid naming conflicts.

2.3 OWNERSHIP

At the core of Rust is the concept of *ownership* [29]. Any value has exactly one owner at any time. Passing or assigning the value transfers ownership to a new owner. By default, Rust has *move semantics*: Instead of copying a value by default, it is moved instead.

```rust
let hello = "Hello, world".to_owned();
let a = hello;

// The following line results in a compilation error; the heap-allocated
// string in ‘hello‘ has been moved into ‘a‘ and is no longer available
let b = hello;
```

The example does not compile because if it would, the heap-allocated `String` would have two owners that could access it concurrently. Small types like integers are marked with a special trait called `Copy`, causing them to be copied instead of moved. The following code will compile:

```rust
let one: u32 = 1;
let a = one;
let b = one;

// one, a and b are all equal to 1
```

2.3.1 *Borrowing*

The second core aspect of the ownership system is borrowing: At any time, there are zero or more read-only references or a single mutable reference to any single value. If any reference exists, the value is said to be *borrowed* until it is returned, that is all references to it cease to exist [29].

Revisiting the earlier example:

```rust
let hello = "Hello, world".to_owned();
let a = &hello;
let b = &hello;
// both a and b are now references to hello

// the following line is an error again: a and b currently borrow hello.
// only after the borrows have ended, that is a and b go out of scope,
// can hello be moved again
let hello2 = hello;
```

If `let a = &mut hello` were used to create a mutable reference, the following line borrowing `hello` again would fail to compile — only one mutable reference to `hello` can exist at any one time, preventing additional immutable references of it.

Borrowing is the most common way to pass arguments to functions that are read-only and not copied; passing in a reference at the function call site creates a borrow and the reference ceases to exist after the call:

```rust
let name = "Alice".to_owned();

// call a function, passing in a reference to name
send_greeting(&name);

// at this point, the reference ‘&name‘ is out of scope, name is no
// longer borrowed
```

2.3.2 *Lifetimes*

To complete the ownership model, lifetimes are used to track how long a value is valid to prevent unexpected invalidation [29]. The lifetimes themselves are generic type parameters:

```rust
struct RingBufIterator<'a> {
    buf: &'a RingBuf,
    pos: usize,
}
```

Iterator invalidation occurs when the structure over which is iterated is modified during iteration and is not discussed here.

This example code above is an iterator over a fictional ring-buffer, it keeps a reference to the ring buffer itself and a current position. Two important invariants are upheld here: Since the iterator structure contains an immutable reference to the ring buffer, the buffer cannot be modified as long as the iterator exists, avoiding iterator invalidation errors. Similarly, the lifetime of the buffer is required to be at least as long as the iterators — in other words, while there is still an iterator pointing to it, the `RingBuf` cannot be freed [29].

static lifetime

A special `'static` lifetime exists and lasts for the entirety of the program. A typical example for these are string constants stored in the binary itself. Rust's string reference type is called `&str`:

In rough C++ terms, a `&'static str` can be thought of as a `const char*`, while a `String` is a `std::string`.

```
let greeting: &'static str = "Hello, world.";
```

2.4 UNSAFE CODE

`unsafe` code blocks provide a way to temporarily break some invariants upheld by the compiler, either to improve performance of a specific low-level section or to implement functionality that clashes with these safety checks. In latter case, it is up to the programmer to manually verify that invariants are upheld. `unsafe` is far from a *carte blanche* though as type checking and a lot of other core requirements are still enforced.

Functions can be declared unsafe as well, making it impossible to call them outside of an `unsafe` block.

The extra capabilities gained in an unsafe block are remarkably few [29]:

1. `static mut` variables may be accessed or updated
2. Raw pointers (2.4.1) can be dereferenced
3. Functions marked `unsafe` may be called

Most often, unsafe blocks are used sparingly to implement low-level primitives, for example a function filling a region of a framebuffer with a certain color might check if the area is inside the bounds once, then use an unsafe method to directly access the underlying memory unchecked instead of bounds-checking each individual pixel.

`&'static mut` are a special case in that they are references to globally shared, mutable memory regions. Rust prevents data races by forbidding simultaneous modification and sharing of values, thus creating references to globally shared mutable values is unsafe.

2.4.1 *Raw pointers*

Raw pointers are very close to their C-equivalents and use almost the same syntax:

Actual pointer math is rare even in embedded Rust code.

```
let val: i32 = 15;
let val_ptr = &val as *const i32;
// val_ptr now contains the memory address of val
```

```rust
// we can increase the address in val by 4 bytes by casting it into a
// pointer-sized int, adding 4 and casting the result back
let ptr_plus_four = ((val_ptr as usize) + 4) as *const i32;

// to access the memory region, we need an unsafe block
let val_offset_four: &i32 = unsafe {
    &*ptr_plus_four
};
```

A reference can be cast into a pointer, which will in turn be convertible into a `usize`, representing a memory address. This operation and any modification of the pointer is safe, since these are just numbers being modified.

The value of an unsafe block is its last expression.

To dereference a pointer, a unsafe block is required. Pointer dereferencing can be used to circumvent all of Rust's memory protection; this is best demonstrated by the fact that an arbitrary location can be accessed directly by casting an address to a pointer type:

```rust
let arbitrary_address: usize = 0x012345;

let ptr = arbitrary_address as *const u32;

// access 32-bit integer at memory location 0x012345
let value: u32 = unsafe {
    *ptr
};
```

2.4.2 *Attributes*

"inlining" is an optimization where a function is not called, but its body placed at every call site instead to save function calling overhead.

Declarations support optional *attributes* that influence behavior during compilation, generate code or do other things a level apart from source compilation [29]:

```rust
#[inline(always)]
fn a_small_function() {
    // this function will always be inlined
}

#[repr(C)]
struct SomeStruct {
    // due to the '#[repr(C)]', SomeStruct is guaranteed to be laid out
    // in memory exactly like an equivalent C struct would
}
```

The `#[repr(C)]` attribute is especially useful when dealing with C-libraries, as it allows types to be used without alteration in C and Rust.

2.5 IDIOMATIC ERROR HANDLING

One big draw of Rust's type system is its ability to express errors and invalid values without exceptions. Central to this effort are the `Option<T>` and `Result<T, E>` types [29].

`Option` is very similar to Haskell's `Maybe` monad.

An `Option<T>` is a parametric type that has two variants: `Some<T>` and `None`. Its use can be illustrated by this safe function for integer division:

```
fn safeDivide(dividend: u32, divisor: u32) -> Option<u32> {
    if divisor == 0 {
        // invalid: cannot divide by 0
        None
    } else {
        // perform regular integer division
        let result = dividend/divisor;
        Some(result)
    }
}
```

The `safeDivide` function can never panic due to a divide-by-zero error and its return type is `Option<u32>` instead of `u32`. Any use of the return value must be explicit about how the error case should be handled; this can be with a `match` block, explicitly accepting that the code may crash or some other way.

The `Result<T, E>` type behaves similarly with an `Ok<T>` variant for successful results and `Err<E>` for failures. The added error type `E` allows returning a cause for an operation failure; instead of just stating that something failed (as with `None`) it allows attaching a reason.

`Option<T>` is not solely used for success-or-failure error handling: An `Option<T>` can be used to indicate a "nullable" value, e.g. a speed limit may be set or absent, a callback may be enabled or not, etc.

2.6 MACROS

Macros are expanded at compile time into Rust code. Rust uses a "hygienic" macro system that prevents some issues that can occur when using C-family macros: Outside scope is carefully guarded in a macro and macros may only emit valid tokens. As a result some pitfalls that are to be looked out for when writing C/C++ macros like forgetting to include parenthesis around a macro definition resulting in unde-

As macros are fairly complex, they are only touched upon briefly here. See the Rust book [29] for details.

sired results or a macro temporary variable shadowing another are completely avoided by design.

Macros are defined through `macro_rules!` and called using an exclamation mark (`!`) as well. A commonly called macro is the `try!` macro that "forwards" a result [29]:

```
let mut file = try!(File::open("some_file"));
```

The actual `try!` macros works slightly different but has been simplified here for the sake of clarity.

The function `File::open` returns a `Result<File, io::Error>`, e.g. returning an `Err<io::Error>` if a file could not be found. Even if the open succeeded, the returned variant would be `Ok<File>`. The `try!` macro inserts the rough equivalent of the following code:

```
let mut file = match File::open("some_file") {
    Ok(f) => f,
    Err(e) => return Err(e)
}
```

After the `try!` expansion, `file` will either be a valid `File` value or the function will have returned with an error.

2.7 BACKWARDS COMPATIBILITY

Rust follows a strict release cycle of six weeks between versions tagged *stable*, with increasing minor version numbers. In parallel, *beta* and *nightly* channels are used to implement new features before they make it back into the *stable* channels [35]. The conservative *stable* versions also guarantee complete backwards compatibility of any 1.x version all the way back to 1.0.

As a result there is a high bar for stabilization of a feature; any change in or removal of behavior is impossible without a major version increase. *nightly* builds are not full of open implementation bugs but rather more likely to contain completed features that are still awaiting appraisal of their long-term impact on the way to stabilization.

Inline assembly is a feature that is not going away, but neither syntactically stable yet [34].

Some implementations in this thesis utilize features that have not made it all the way to stable at the time of writing. None of the utilized features are "controversial" in the sense that they are in high danger of being removed altogether but rather not stabilized due to a lack of consent about finer details.

2.8 RUST TOOLING

LLVM was formerly "Low Level Virtual Machine", but the project has outgrown its acronym.

In addition to the compiler a suite of tools is required to produce actual working software. Rust is based on the LLVM project (LLVM) and uses it as a backend for code-generation but still links with the GCC linker. Dependencies are managed by the package manager Cargo, which provides versioning, a flexible build system and an online package repository [28].

3 BARE METAL DEVELOPMENT

Running a program on *bare metal*, that is in the absence of an OS, is the simplest conceptually, but often a process full of small details that are easily forgotten otherwise because one has gotten used to their presence while writing applications on larger OS managed systems. Yet it is often the most sensible choice when presented with requirements for low power usage or cheap manufacturing cost, strict absolute timing needs or embarrassingly simple functionality goals.

A plethora of architectures and chips exist today, many cheaply at the disposal of almost every developer with slightly different or even fundamentally unique designs. This thesis will demonstrate programs with the STM32F29I-Discovery [26] microcontroller unit (MCU), because it uses the widely available Advanced RISC Machine (ARM) Cortex M4 processor that shares core architectural traits with a large family of similar designs [36]. The STM32F429I was chosen because it there is some Linux kernel — and Device Tree (DT) — support available for it at the time of this writing, which is more detailed in Chapter 6.

Embedded firmware development is often done using tools and integrated development environments (IDEs) supplied by the hardware vendor in an effort to shield the developer from the need of understanding the details of the underlying hardware. These tools vary in quality and while enabling a programmer to get things done fast, without the insights that come from a deeper knowledge of the toolchain often with avoidable flaws. Furthermore platform support outside of Windows is often lackluster, though sometimes improved by the existence of numerous open source projects implementing the necessary applications to allow development.

Fortunately `arm-gcc` is receiving contributions from ARM itself.

For this thesis, the decision between vendor supplied tools and libraries and cross-platform open source applications is a non issue — at the time of this writing there was no official support for Rust on any device considered, leaving no choice but to build things from the ground up.

3.1 A HIGH LEVEL OVERVIEW

On a fundamental level every computer operates using a simple principle: A list of instructions that form the program are read one by one and executed. Internally, the central processing unit (CPU) that executes these has a small set of memory called registers, which are directly accessible. Among those registers is a program counter (PC) register containing the memory address of the next instruction to be fetched and executed: During normal operation this value is automatically increased after each instruction but writing a new address into the PC register allows the CPU to perform a *jump*: Execution is continued at the new address, allowing different code paths to be taken.

A jump is often also called a branch.

3.1.1 *Memory mapped peripheral access*

For any meaningful application, some sort of memory is required to be attached to the CPU to store the program and any data it might operate on initially. Every CPU has a native *word size*, the size of the largest integer it can process in most operations. It determines the size of a pointer and in doing so the largest amount of memory that can directly be addressed.

Rust avoids a common mistake by using machine-independent integers like `i32` in the idiomatic style instead of `usize`.

A 32-bit CPU can address 4096 megabytes (MB) of memory but most MCUs have between less than 1 and 128 MB memory attached. The extra addressing space is used for what is known as *memory mapping* peripherals [36].

All ARM Cortex-M series CPUs as of this writing are 32-bit processors.

When a hardware component is memory mapped into a region of memory, any writes to that memory region by the CPU will instead be written to the internal registers of the memory mapped hardware component, allowing the CPU to control its operation. For example, to toggle an output pin writing a value of 0 or 1 to a specific memory address may be required; similarly the frame buffer of a thin-film transistor (TFT) screen may be written to using a fixed memory region. Input can be handled as well; the press of a button may toggle a memory address — and/or toggle an interrupt (see below).

Volatile access must be taken into account and is discussed in more detail in chapter 4.

3.1.2 *Program flow and interrupts*

Another concern is general program flow: Most embedded systems are single-threaded, executing instructions one-by-one. If a reaction to outside events such as a button press or network message arriving is desired, *polling* is one way of handling these events: Whenever an event occurs, the affected component stores information about the event inside one or more registers, which is periodically read (via *memory mapping*, see above) by the CPU to check if new information is available. The running program can act accordingly and afterwards continue the polling cycle if desired [36].

A drawback of polling is that is often prevents use of energy-saving modes and wastes CPU cycles.

A better approach is interrupt handling: Instead of waiting to be polled, components are wired to the CPU interrupt controller. When an interrupt is raised, the CPU suspends normal execution by saving all register contents, then jumping directly into an interrupt service routine (ISR). After the ISR finishes, normal execution is resumed by restoring the original register contents, including the PC [36].

Interrupt handling can be arbitrary complex with hierarchies of interruptable interrupts built on the same underlying principle.

3.2 THE ARM CORTEX M4

The Cortex M4 processor is a 32-Bit processor featuring an interrupt controller and 16 registers including stack pointer (SP), link register (LR) and PC. The SP can be set freely during initialization and is used by the processor as space to store register values during interrupts, while LR is used to store return addresses. Any device peripherals attached to the CPU must be connected through memory mapping [36].

3.2.1 *Interrupt Vectors*

An interrupt vector table (IVT) contains addresses for any number of interrupts that can be handled, the first entry being the initial value of the SP [36].

The virtual address `0x0000 0000` initially get redirected to the start of the flash memory.

The IVT is located at memory address `0x0000 0000`, but can be relocated during runtime.

3.2.2 *Boot sequence*

The boot sequence of the Cortex M4 is very simple: After a reset — and in case of power on, reaching operating voltage — all registers are initialized to known default values. The SP is initialized from the IVT at this point.

The CPU is set to an initial clock rate of 8 megahertz (MHz) and immediately begins fetching and executing instructions. Many peripherals at this point are not powered on [36].

3.2.3 *Additional features*

The Cortex M4 has a host of other features, such as multiple stack pointers for kernel and processes, memory protection units and other functionality, targeted at the development of an actual operating system on the hardware. As these are not necessary for single-threaded, single-task applications, they are not further mentioned here.

3.3 THE STM32F29I-DISCOVERY

The STM32F29I-Discovery (see figure 3) is a development board housing an MCU and peripherals. Hardware includes headers for the CPU

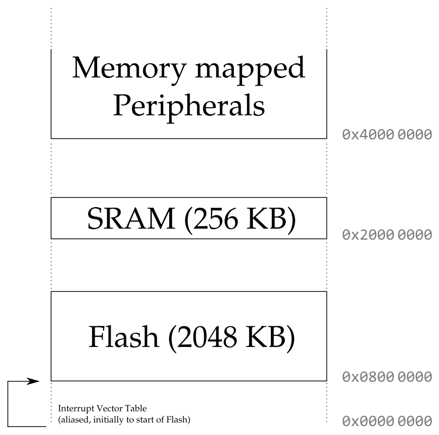
Memory mapped Peripherals
0x4000 0000
SRAM (256 KB)
0x2000 0000
Flash (2048 KB)
0x0800 0000
Interrupt Vector Table (aliased, initially to start of Flash)
0x0000 0000

Figure 2: A simplified view of the memory map of the STM32F429I-Discovery

general purpose input/output (GPIO) and other pins, a TFT screen, a button and an LED connected to GPIO pins, as well as additional integrated circuits (ICs) for a variety of protocols to connect devices or networks. The board's core is an ARM Cortex-M4 CPU, additionally an ST-Link debugging tool is attached to upload firmware and provide debugging facilities. The system can simultaneously be connected and powered via Universal Serial Bus (USB).

The memory layout holds few surprises; attached components are mapped into the address space at a fixed offset, the same holds true for SRAM and Flash memory. No alterations to the boot process are made, the CPU starts up almost immediately.

3.3.1 *ST-Link*

To execute any program, it must be transferred to the MCU before it can be run. Different interfaces exist for this purpose, one standardized solution available being the Joint Test Action Group (JTAG) debugging interface. Through this, a CPU can be started, stopped, halted and single-stepped and other peripherals directly controlled, allowing the upload of firmware directly into the flash memory. To use the JTAG interface, a special hardware tool is usually required and the necessary pins of the CPU need to be exposed.

The STM32F29I-Discovery, being a *development* board, comes with a debugger already attached; through a USB connection the ST-Link debugger can be controlled to influence execution, read and write memory directly or upload firmware.

A blob here is a single continuous piece of binary data.

On STM32F429I-Discovery, flash memory is mapped starting at `0x0800 0000`.

Instead of JTAG, the ST-Link debugger uses the serial-wire debug (SWD) protocol, a part of the ARM specification. In the absence of official software to utilize the debugger, an open source implementation exists and is used instead [30]. Through the ST-LINK/V2 protocol a

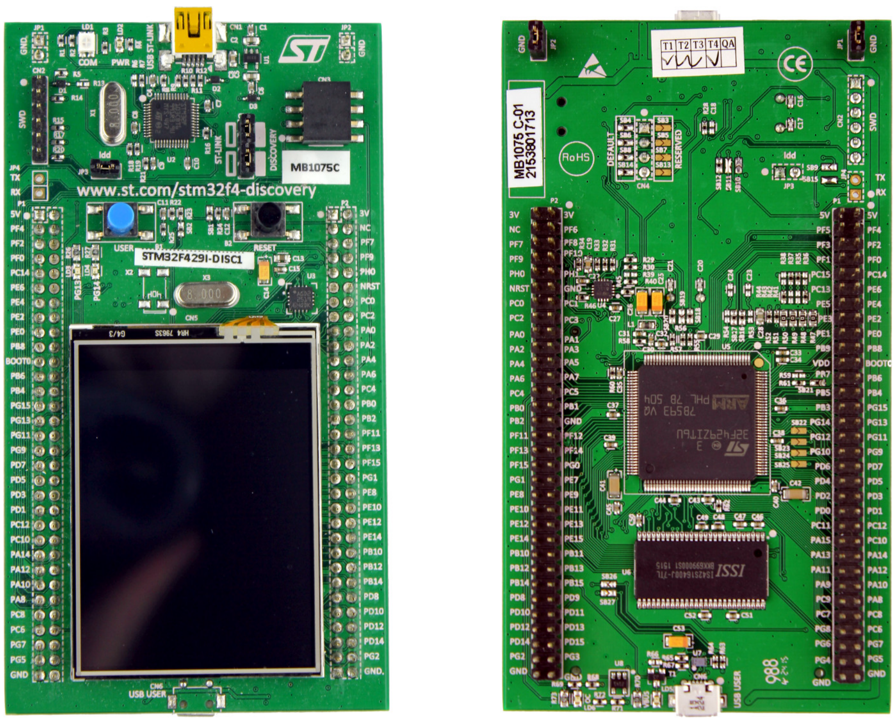

Figure 3: The STM32F429I-Discovery, front and back.

blob of binary data can be placed at any memory location directly. Since the system's flash memory is mapped to a fixed continuous address, writing a new firmware to the board is the same as writing it to a specific memory region.

3.4 CREATING A FIRMWARE IMAGE

With the tools to upload an image in place, the last requirement to store and run a program is the creation of a suitable binary that fulfills the following requirements:

- Starts with an IVT at offset 0 that contains initial values for the SP to SRAM and PC to the initially called function.
- Uses fixed addresses for all function and data, calculated using offset + flash memory address.
- Uses all hardware correctly using the memory-mapped addresses.

E.g. a function place at `0x0000 1234` in the binary must be called at `0x0800 1234`.

This is usually achieved by the linker that is supplied with a suitable linker script describing the memory layout of the target architecture. C compilers as well as Rust produce an ELF binary using the GNU Compiler Collection (GCC) linker, the same linker script can be used for C as well as Rust programs.

4

PRACTICAL EMBEDDED DEVELOPMENT WITH RUST

The first step in any embedded development process is obtaining a toolchain. A Rust compiler and Cargo can be obtained by the `rustup` cross-platform installer [31] that will download any Rust channel or version for any supported target platform.

Given a compiler for the host and a package manager, additional tools can be downloaded, compiled and installed through Cargo.

4.1 COMPILING AND LINKING FOR THE STM32F429I-DISCOVERY

Instead of being recompiled for every target platform, the Rust compiler can be configured at runtime using a JSON file to add new target specifications. Figure 4 shows a suitable configuration file for the STM32F429I-Discovery; the `thumbv7em` instruction set architecture (ISA) is an ARM ISA optimized for embedded systems.

Cargo configuration is not demonstrated here, but can be found at [28].

More precisely, `thumbv7em` is the ARMv7 architecture with the Thumb-2 instruction set.

With a compiler in place, a linker script (see section 3.4) is needed for linking. A minimal layout file is shown in figure 5. The linker script is not Rust-specific and can be shared with C/C++ programs as well.

The core libraries

After compiler and linker are set up, the core libraries are the last missing piece to compile a working firmware.

At the time of this writing, 33 targets were officially supported by rustup, but not `thumbv7em-none-eabi`.

The Rust standard library (stdlib) is feature rich and supports heap-allocated strings, multi-threading, synchronization primitives like mutexes, collections, Unicode, filesystem access and lots of other features. Many of these function depend on OS primitives, e.g. the threading is dependent on OS threads and without an actual filesystem there can be no file operations.

Underneath the stdlib is the Rust core library (libcore), a lower level library that has no dependencies and implements all features that do not have any requirements outside the library. Numeric operations, iterators, formatted string writing and idiomatic error handling are found here.

One could roughly compare the stdlib to the C++ template library and the libcore to C's libc.

Figure 4: STM32F429I-Discovery compatible JSON target specification.

```
{
  "arch": "arm",
  "cpu": "cortex-m4",
  "data-layout": "e-m:e-p:32:32-i64:64-v128:64:128-a:0:32-n32-S64",
  "disable-redzone": true,
  "executables": true,
  "llvm-target": "thumbv7em-none-eabi",
  "morestack": false,
  "os": "none",
  "relocation-model": "static",
  "target-endian": "little",
  "target-pointer-width": "32",
  "no-compiler-rt": true,
  "pre-link-args": [
    "-Tlayout.ld",
    "-Wl,--build-id=none",
    "-Wl,--gc-sections",
    "-mcpu=cortex-m4",
    "-mthumb",
    "-nostartfiles"
  ]
}
```

Figure 5: A minimal linker script for the STM32F429I-Discovery.

```
MEMORY
{
  /* there’s a small hole after sram0 ends, but sram1 and 2 are continuous */
  sram0 (rwx)  : ORIGIN = 0x20000000, LENGTH = 112K
  sram1 (rwx)  : ORIGIN = 0x2001C000, LENGTH = 16K
  sram2 (rwx)  : ORIGIN = 0x20020000, LENGTH = 64K

  /* 2 megs of flash memory */
  flash (rwx) : ORIGIN = 0x08000000, LENGTH = 2M
}

SECTIONS
{
    .text 0x08000000:
    {
        _VT_BEGIN = .;
        KEEP(*(vectors))    /* Vector table */
        _VT_END = .;
        . = 0x10000;
        _TEXT_BEGIN = .;
        *(.text)            /* Program code */
        _TEXT_END = .;
        _RODATA_BEGIN = .;
        *(.rodata)          /* Read only data */
        _RODATA_END = .;
    } > flash

    /* "The .data segment contains any global or static variables which
       have a pre-defined value and can be modified. [...]" */
    .data :
    {
        *(.data)
    } > sram0

    /* "The BSS segment contains all global variables and static variables
       that are initialized to zero or lack explicit initialization [...]" */
    .bss :
    {
      _BSS_BEGIN = .;
        *(.bss)
      _BSS_END = .;
    } > sram0

    /* Stack: 64K of sram2. _STACK_TOP must be at the end of this area. */
    .stack 0x2002FFFC: {
      _STACK_TOP = .;
    }
}
```

At the very least a working libcore is required to compile any Rust program, officially supported targets ship with compiled binaries of these. Since `thumbv7em-none-eabi` is not officially supported yet, the libcore must be built first.

A shortcut exists at this point: The xargo [1] crate simplifies this process by wrapping Cargo, downloading and compiling the libcore's sources when no binaries are found for the target platform.

It is conceivable that xargo's functionality may be subsumed by rustup longterm.

4.2 A FIRST PROGRAM

With libcore in place it becomes possible write the first embedded Rust program:

```
#![feature(lang_items)]
#![no_std]

pub mod stubs;

// _STACK_TOP is filled in by the linker.
extern {
    static _STACK_TOP: ();
}

// the linker expects a C-compatible start function.
extern "C" fn _start() {
    main()
}

// the vector table, at minimum, requires the stack pointer's
// initial value and the reset vector
#[repr(C)]
pub struct VectorTable {
    pub stack_pointer_initial: &'static (),
    pub reset_vector: Option<extern "C" fn()>,
}

#[link_section="vectors"]
#[no_mangle]
pub static VECTOR_TABLE: VectorTable = VectorTable {
    stack_pointer_initial: &_STACK_TOP,
    reset_vector: Some(_start),
};

fn main() {
    loop {
    }
}
```

Its main function is fairly unexciting, a simple endless loop. The setup around it requires a little more care:

4.2.1 *Attributes*

Crate attributes are attributes prefixed with an exclamation mark:

```
#![feature(lang_items)]
#![no_std]
```

The `#!` prefix applies the attribute to the declaration it is in instead of the declaration that follows. In the example the target is the top-level module or crate.

Language items, enabled by the `#![feature(lang_items)]` attribute, mark specific functions as implementations for core functionality to allow them to be implemented in libraries [29], e.g. a custom memory allocator would need to provide the equivalent of `malloc` and `free`. Section 4.2.2 utilizes these.

`malloc` is the typical name for a C function that allocates memory, which can later on be freed with `free`.

Avoiding the use of stdlib, `#![no_std]` disables it.

4.2.2 *Stubs*

The stubs module, whose contents are listed in figure 6, implements a few necessary function stubs that are expected to be present by Rust but can be simply empty functions in many embedded development use cases. The first five stubs all deal with the handling of "exceptions", which are panics. In the example program there is almost no reasonable recovery or handling of panics and as a result all the stubs are empty. A more sophisticated panic handler could halt the CPU using inline assembly to allow post-mortem debugging or even try to display things on the built-in TFT display if still possible.

A method stub or stub is a stand-in for missing functionality.

The `start` function is looking a little out of place — its signature is somewhat similar to the regular `main` function of a UNIX program. It is needed because the Rust compiler will produce an ELF binary and expect a regular start function to be present. If one were to call the resulting binary, a simple error code of `-1` would be produced. Since the MCU will start from the offset given in the IVT, it will simply be ignored.

4.2.3 *Reset vector and stack pointer*

As discussed earlier in section 3.4, the IVT must be initialized with a value for the stack pointer and a reset vector that points to be first instruction that should be executed. Since the stack's location is defined in the linker script, `static _STACK_TOP` is defined to be `extern`

Figure 6: The method stubs required to compile the example program

```
use core::fmt;

// the following two method stubs are needed to compile
#[lang = "eh_personality"]
extern fn eh_personality() {}
#[lang = "panic_fmt"]
extern fn panic_impl(_: fmt::Arguments, _: &'static str, _: u32) -> ! {
    loop {}
}
#[no_mangle]
pub extern fn __exidx_end() {}
#[no_mangle]
pub extern fn __exidx_start() {}
#[no_mangle]
pub extern fn __aeabi_unwind_cpp_pr0() {}

#[lang = "start"]
fn start(_: *const u8, _: isize, _: *const *const u8) -> isize {
    -1
}
```

and will be supplied by the linker at link time. Its type is of no consequence to the program and uses the *unit type*, since only its address is needed.

The unit type is a type with a single value that is zero bytes in size [29]. In C, `void` could be substituted.

The `extern "C" fn _start()` is the entry point to the program; all interrupt handlers use C application binary interfaces (ABIs). For convenience, our start function calls directly into the Rust function `main`.

Tying all these together is the `VectorTable` type which represents an IVT with just two entries, one of which is the stack pointer, the other being the reset vector. The `#[link_section="vectors"]` attribute places it at the location for `vectors` in the linker script (see figure 5), which will be placed at the beginning of the flash memory.

If this solution seems brittle, creating a raw pointer directly or through a trait can be used instead.

One slightly more obscure fact here is the use of `Option<extern "C" fn()>` as a pointer: Whenever a reference type is placed inside an `Option`, since it is non-nullable, an optimization takes place that represents `None` as the value 0 and `Some(address)` as just `address`. For this reason, an `Option<&T>` is the same size as a `&T`.

4.3 A MORE COMPLICATED PROGRAM

An infinite loop that does nothing is not the most exciting example, for this reason the "Hello world" of embedded programming is a blinking light-emitting diode (LED). The STM32F429I-Discovery comes with two LEDs soldered on that are easily accessible as they are wired to GPIO pins PG13 and PG14 [26]. Turning on an LED and blinking it requires several steps, all of which involve memory-mapped peripheral access:

1. The GPIO pins themselves must be "powered on" by enabling the clock signal.
2. The direction of the GPIO pin must be set to *output*.
3. The pin's output value must be periodically toggled between 0 and 1.

4.3.1 *Volatile memory access*

When dealing with memory-mapped devices, an optimizing compiler can become a problem, as it will make deductions solely based on the code to be compiled and no outside influences.

Assuming a specific location in memory must be read in a loop until it turns from 0 to 1 — e.g. waiting for a button press — an optimizing compiler that sees the following code

```
// 'button_pressed' contains the state of an external button
let button_pressed = ...;

loop {
    if button_pressed {
        break
    }
}
```

might deduce that since `button_pressed` is not modified inside the loop, the code may be refactored into the following equivalent:

```
let button_pressed = // ...

if ! button_pressed {
    loop {

    }
}
```

This is, of course, incorrect — even though from the compiler's point of view it is the same and much faster even! The same issue exists for writes; if the output pin connected to an LED is to be toggled multiple times, it would not be good for all writes to be optimized into a single one.

The solution is declaring a read or write *volatile* [3]. Volatility means that the read or write cannot be reordered or optimized away because in between, the value may be modified by outside actors.

The volatile qualifier in C and Rust

`const` is another qualifier in C.

In C, `volatile` is a qualifier [19] indicating that a variable is volatile:

```c
uint8_t volatile *reg = (uint8_t volatile *) 0x00113344;

*reg = 1;
```

In C, `uint8_t` is a common `typedef` for an unsigned byte.

The code above defines `reg` as a pointer to a `volatile` unsigned byte and changes its value to one. Since the target of `reg` is defined as volatile, the write will not be optimized away.

A key difference is that Rust does not have a volatile qualifier or any other way of indicating that a variable is volatile. Instead, libcore functions need to be used to emit the correct instructions. The C example above is written as follows in Rust:

```rust
use core::ptr;

let reg_ptr = 0x00113344 as *mut u8;

// direct memory access requires an unsafe block
unsafe {
    ptr::write_volatile(reg_ptr, 1);
}
```

Volatile memory access is performed by regular libcore functions.

4.3.2 *A blinking LED*

From this point on, code listings will not be complete listings due to length constraints.

Armed with volatile memory access, the desired program can now be implemented. First, a few addresses need to be defined:

```rust
// Register addresses
const AHB1_GPIOG_BASE: u32 = 0x4002_1800;

const AHB1_RCC_BASE: u32 = 0x4002_3800;
const RCC_AHB1ENR: u32 = AHB1_RCC_BASE + 0x30;

const AHB1_GPIOG_MODER: u32 = AHB1_GPIOG_BASE + 0x00;
const AHB1_GPIOG_ODR: u32 = AHB1_GPIOG_BASE + 0x14;

// Bitmask to enable GPIO G
const RCC_AHB1ENR_GPIOG_EN: u32 = 0b0100_0000;

// Bitmask to set pin 13's mode to output
const AHB1_GPIOG_MODER13_OUT: u32 = 0x01 << 26;

// Bitmask to enable pin 13 on GPIO G
const AHB1_GPIOG_ENABLE_13: u32 = 1 << 13;
```

All peripherals needed are on the first Advanced High-performance Bus (AHB) [27]. The base address of GPIO G — the LED in question is labeled as "**PG13**" on the printed circuit board (PCB) itself — is stored in a constant named `AHB1_GPIOG_BASE`. The naming scheme is almost identical to the Cortex microcontroller software interface standard (CMSIS) library (see section 4.4 for details), detailed information about the layout of the GPIO register can be found in section 5.2.

The exact bus devices are on is of little consequence here, except that all share one continuous address space.

The reset and clock control (RCC) register enables and disables peripherals by controlling whether or not they receive a clock signal. The sub-address `RCC_AHB1ENR` allows toggling some peripherals on the first AHB; the 6th bit toggles GPIO G specifically. Other constants contain the bitmask and addresses to set output direction and value [26, 27, 36].

After defining constants for all required addresses, the main logic can be implemented:

```
use core::ptr;

fn main() {
    // 1. enable the clock
    unsafe {
        let prev = ptr::read_volatile(RCC_AHB1ENR as *const u32);
        ptr::write_volatile(
            RCC_AHB1ENR as *mut u32,
            prev | RCC_AHB1ENR_GPIOG_EN
        )
    };

    // 2. set direction of pin 13 (PG13) to output
    unsafe {
        let prev = ptr::read_volatile(AHB1_GPIOG_MODER as *const u32);
        ptr::write_volatile(
            AHB1_GPIOG_MODER as *mut u32,
            prev | AHB1_GPIOG_MODER13_OUT
        )
    };
    loop {
        // 3a. set GPIOs to output to enable just LED 13
        unsafe {
            ptr::write_volatile(
                AHB1_GPIOG_ODR as *mut u32,
                AHB1_GPIOG_ENABLE_13
            )
        };

        for i in 0..50000 {
            // waste a few cycles
        }

        // 3b. turn off all GPIOs
        unsafe {
            ptr::write_volatile(
                AHB1_GPIOG_ODR as *mut u32,
                0
            )
        };

        for i in 0..50000 {
            // waste a few cycles
        }
    }
}
```

A better delay is implemented either using `NOP` instructions or a timer interrupt; `for` was chosen for simplicity here

Again, `unsafe` blocks are needed for accessing memory directly through pointers. The `for` loops emulate a delay as long as compiler optimizations are turned off — otherwise the loops will be optimized away and the LED will light up at 50% intensity instead, being toggled on and off very fast.

The example also has a flaw: It will turn off all other LEDs/GPIOs while toggling PG13; as the output data register (ODR) is written 32-bits at a time, while each bit corresponds to a particular output pin. A better long-term solution will at the very least need to read the register contents before rewriting it — and possibly introduce race conditions while doing so — to keep the state of other pins intact.

Overall, the implementation above is a poor one: It is an almost direct translation from C, without any abstractions introduced resulting in fairly unwieldy code. In the following chapter the design will be improved upon.

4.4 CMSIS

A standard peripherals library contains additional functionality beyond hardware access, but the majority of code deals with the latter.

CMSIS is an effort by ARM to create a portable library for peripheral access on any ARM MCU. While attached hardware may differ across PCBs, since the Cortex MCU already contains a lot of ICs for device access, a large amount of functionality can be carried across from different implementations [36]. Manufacturers of ARM-based MCUs are required to provide a CMSIS-compatible library that includes header files and definitions for additional hardware contained on the respective board.

CMSIS is written in C and models the memory layout directly using C structures. An example from the CMSIS compliant peripherals library provided by ST Microelectronics for the STM32F429I-Discovery contains the following definition for a GPIO register (some comments have been removed) [25]:

```c
typedef struct
{
  __IO uint32_t MODER;    /* GPIO port mode register */
  __IO uint32_t OTYPER;   /* GPIO port output type register */
  __IO uint32_t OSPEEDR;  /* GPIO port output speed register */
  __IO uint32_t PUPDR;    /* GPIO port pull-up/pull-down register */
  __IO uint32_t IDR;      /* GPIO port input data register */
  __IO uint32_t ODR;      /* GPIO port output data register */
  __IO uint16_t BSRRL;    /* GPIO port bit set/reset low register */
  __IO uint16_t BSRRH;    /* GPIO port bit set/reset high register */
  __IO uint32_t LCKR;     /* GPIO port configuration lock register */
  __IO uint32_t AFR[2];   /* GPIO alternate function registers */
} GPIO_TypeDef;
```

`__IO` is a macro defined as an alias for `volatile`. A pointer with the correct memory address for the GPIO port is globally available (some lines have been omitted for clarity) [25]:

```c
#define PERIPH_BASE       ((uint32_t)0x40000000)
#define AHB1PERIPH_BASE     (PERIPH_BASE + 0x00020000)
#define GPIOG_BASE         (AHB1PERIPH_BASE + 0x1800)
#define GPIOG            ((GPIO_TypeDef *) GPIOG_BASE)
```

After including the header, the globally available `GPIOG` can be accessed from anywhere in the source and is expanded at compile time to a direct pointer-cast access on a volatile struct:

```c
// set pin 13 to output
GPIOG->MODER = 0x01 << 26;
```

This is a convenient and fast way to access these registers — but comes with all the perils of unguarded access to memory, lacking any safe-guard. Improving the ergonomics of the Rust-based, safe approach is the first step towards a better and safer way of accessing these registers and will be discussed in the following chapter.

5 USING OWNERSHIP ON HARDWARE RESOURCES

In everyday Rust, using an `unsafe` block or function is a rare occurrence, as they are intended to be hidden behind an abstraction layer of well-checked code. The previous chapters made fairly liberal use of the `unsafe` keyword, which is an issue that needs to be addressed with proper abstractions.

5.1 GUARDING A POINTER

Raw pointers can be safely created, compared, modified or otherwise manipulated, as they are merely word-sized integers representing a memory address. Their hazardous potential is only visible when they are dereferenced and the manipulations take effect. Mentally substituting a 32-bit integer for the a pointer type illustrates this: Arithmetic operations on it are safe; only when the integer is interpreted as a memory address which is looked up can it negatively affect the system.

To guarantee that a pointer dereference is safe, it is up to the programmer to ensure that it points to valid memory and is not *aliased* — no other mutable pointer in a parallel task is referencing the memory region or parts of it at the same time.

Parallel tasks do not necessarily require parallel execution.

For safe access, a pointer that does not change and is guaranteed to point to a valid region requires only that no additional pointers are created that alias its memory region. By being careful with their creation it is possible to construct pointers or pointer-wrapping structures that allow safe dereferencing.

Essentially this reverses the burden of ensuring safety: Creation of pointers becomes unsafe while their use is made safe.

We implement this in a first attempt at a new register-type and even encode the volatile properties at the same time:

A static `new` function is the idiomatic way to create constructor-like methods on structs.

```
use core::ptr;

struct HardwareRegister<T> {
    addr: *mut T
}
```

```
impl<T> HardwareRegister<T> {
    pub unsafe fn new(addr: *mut T) -> HardwareRegister<T> {
        HardwareRegister {
            addr: addr,
        }
    }

    pub fn read(&self) -> T {
        unsafe { ptr::read_volatile(self.addr) }
    }

    pub fn write(&mut self, value: T) {
        unsafe { ptr::write_volatile(self.addr, value) }
    }
}
```

The `HardwareRegister` `struct` wraps a pointer to a type `T` and marks the creation as unsafe — it is up to the user to ensure its uniqueness and that it points to the correct address.

Any struct is non-cloneable, unless it implements the `Clone` trait. `Copy` indicates a plain old data (POD) type.

Every `HardwareRegister` is also protected by ownership rules; since it implements neither the `Clone` nor the `Copy` trait, no duplicates can be made. `read` requires a simple reference, while `write` requires a mutable one, preventing any write access while the register is still borrowed for reading purposes. For the same reason, only a single task can write to the register at any one time.

Both read and write enforce volatile access to the memory region, but more importantly they are not unsafe — the `unsafe` block placed inside the safe functions `read` and `write`. `HardwareRegister` assumes that all necessary properties have been checked on creation, indicated by the fact that `new` is unsafe.

5.2 THE GPIO REGISTER

The GPIO register data structure can be expressed almost the same way in Rust as it can be in C (compare section 4.4):

```
#[repr(C)]
pub struct GPIO {
    // 0x00 MODER
    mode_reg: u32,
    // 0x04 OTYPER
    type_reg: u32,
    // 0x08 SPEEDR
    speed_reg: u32,
    // 0x0C PUPDR
    pull_up_down_reg: u32,
    // 0x10 IDR
    input_reg: u32,
    // 0x14 ODR
    output_reg: u32,
    // 0x18 BSSR
    bssr_reg: u32,
    // 0x1C LCKR
    lock_reg: u32,
    // 0x20 AFRL
    alt_func_reg_low: u32,
    // 0x24 AFRH
    alt_func_reg_high: u32,
}
```

Some small alterations have been made for simplicity, such as splitting the array of 32-bit integers of `AFR` into two separate fields. The `#[repr(C)]` attribute ensures that the memory layout is the same as the corresponding C structure.

Shortfalls of the wrapped pointer approach

Attempting to use the resulting `GPIO` structure with the previous wrapped pointer type quickly reveals its shortcomings:

```
let gpio_reg = unsafe {
    HardwareRegister::new(0x0011223344 as *mut GPIO)
};

// GPIO can only be read as a whole!
let kitchen_sink = gpio_reg.read();
```

Here, `gpio_reg` can only be read and written as a whole. With direct access to fields being indispensable, a possible fix could naively be implemented by wrapping pointers for each individual field:

```
pub struct GPIOPtrs {
    mode_reg: HardwareRegister<u32>,
    type_reg: HardwareRegister<u32>,
    // ...
}
```

Now individual access is possible but at an even greater cost: The `GPIOPtrs` struct has become complicated to create as it is now a collection of pointers, each of which needs to be initialized individually.

Instead of an overhead of a single pointer, now an extra pointer needs to be stored for each field, even risking overhead through the indirection on access.

5.3 A BETTER SOLUTION

While the guarded pointer protects the integrity well and the individual field based approach allows for fine grained access, both abstractions lack either in flexibility or efficiency. To reach par with the C implementation, the compiler's knowledge of the data structure and the offsets of its fields must be leveraged for gains in space and speed efficiency.

`Volatile<T>` is a type that wraps a single `T`. The value is accessible as `self.0`.

The first step is redesigning the field type: To ensure volatility on fields, a type wrapping not a pointer but an actual type is introduced:

```rust
#[repr(C)]
pub struct Volatile<T>(T);

impl<T> Volatile<T> {
    pub fn read(&self) -> T {
        unsafe {
            ptr::read_volatile(&self.0)
        }
    }

    pub fn write(&mut self, src: T) {
        unsafe {
            ptr::write_volatile(&mut self.0, src)
        }
    }
}
```

`Volatile` itself does not store a pointer, but wraps an actual type. The read and write method create a reference to `self.0`, which is the value itself, before passing it to the volatile access functions.

Now fields can be expressed in an improved `struct`:

```rust
#[repr(C)]
pub struct GPIO {
    // 0x00 MODER
    mode_reg: Volatile<u32>,
    // 0x04 OTYPER
    type_reg: Volatile<u32>,
    // 0x08 SPEEDR
    speed_reg: Volatile<u32>,
    // 0x0C PUPDR
    pull_up_down_reg: Volatile<u32>,
    // 0x10 IDR
    input_reg: Volatile<u32>,
    // 0x14 ODR
    output_reg: Volatile<u32>,
    // 0x18 BSSR
    bssr_reg: Volatile<u32>,
    // 0x1C LCKR
    lock_reg: Volatile<u32>,
    // 0x20 AFRL
    alt_func_reg_low: Volatile<u32>,
    // 0x24 AFRH
    alt_func_reg_high: Volatile<u32>,
}
```

After defining the new `GPIO` structure, instances can be created. However, an instance would imply an owned section of memory that can be moved — there's no guarantee where it will be on creation.

For this reason, new instances should not be created normally on the stack or heap, but solely by casting a pointer to an existing memory mapped hardware peripheral into a reference.

An example, to create a GPIO structure for GPIO G as seen in section 4.3.2, the following code can be used:

```
let gpio_g: &'static mut GPIO = unsafe { &mut *(0x4002_1800 as *mut GPIO) };
```

The code looks a bit cryptic but is straightforward: The integer literal `0x4002_1800` is cast into a mutable pointer to a `GPIO` at the specified memory location. The resulting pointer is immediately dereferenced — hence the `unsafe` block — and re-referenced to be turned into a mutable reference with a static lifetime.

Any number of underscores can be inserted into integer literals to group the digits for legibility.

To ease instantiation later on, this process can be expressed in a trait with a default implementation:

A trait can provide a default implementation for any of its functions

```
pub trait VolatileStruct : Sized {
    unsafe fn from_ptr(addr: *mut Self) -> &'static mut Self {
        &mut *addr
    }

    unsafe fn from_addr(addr: usize) -> &'static mut Self {
        // the Sized bound is used here, otherwise the cast doesn't work
        Self::from_ptr(addr as *mut Self)
    }
}
```

The `: Sized` requires that types implementing the trait must have a known size, otherwise the necessary cast in `from_addr` is not possible.

The GPIO G example can now be rewritten as:

```
impl VolatileStruct for GPIO {
   // empty impl block, the default implementation is sufficient
}

// ...

let gpio_g = unsafe { GPIO::from_addr(0x4002_1800) };
```

The full for `Volatile<T>` and `VolatileStruct` can be found in figure 7.

As it is up to the programmer to ensure the address is correct and only used once, the explicit `unsafe` is kept on both the `from_ptr` and `from_addr` trait functions. Otherwise no casts are necessary and any new type that implements the `VolatileStruct` trait can be instantiated this way.

The proposed implementation successfully keeps the "unsafe construction, safe dereferencing" from the first approach and the volatile single-field access from the second, all without runtime overhead. Unsafe code sections are limited to instantiation only.

5.4 ACTUAL OWNERSHIP VS STATIC LIFETIMES

Using a `&'static mut T` reference instead of an actual instance requires re-evaluation of whether or not the original objective can still be achieved: Extending ownership semantics to actual hardware. While mutability and immutability work very much the same, as any number of immutable references can be created from any mutable reference, moving the value works differently, which is easily illustrated:

```
let x = ...;  // x is some non-cloneable value

// passing a mutable reference into f
f(&mut x);
// the borrow of a ends here, can be borrowed again
f2(&mut x);

// passing a by value
g(x);

// x has been moved into g and not returned
g2(x)    // error!
```

Unless it is, of course, returned.

The main difference of passing x by reference instead of by value is that it is impossible to get it back out of the function. When passing by reference, if the function does not have an explicit lifetime requirement, a new reference is created with a lifetime that begins when the function is executed and ends once it returns.

While a `&'static mut` is also a reference, it has the longest possible lifetime: `'static`. The lifetime does not end after the function ends and since it is the longest possible lifetime, the borrow does not end because the function could have stored the passed-in reference in another data structure. Passing the reference of the memory-mapped hardware creates an infinitely long borrow instead of moving it — which prevents reuse further down as effectively as moving out of an object.

5.5 SYNTACTIC SUGAR AND PERFORMANCE

Rust supports operator overloading through traits which can be used to increase the convenience of or'ing register contents. This allows turning the previous version

```rust
// assuming rcc_ahb1enr is a suitable struct representing the
// reset and clock control register:

let tmp = rcc_ahb1enr.reg.read();
rcc_ahb1enr.reg.write(tmp | RCC_AHB1ENR_GPIOG_EN);
```

into the following:

```rust
use core::ops;

impl<T> ops::BitOrAssign<T> for Volatile<T>
where T: ops::BitOr<T, Output=T>
{
    fn bitor_assign(&mut self, val: T) {
        let tmp = self.read();
        let new_val = tmp | val;
        self.write(new_val);
    }
}

// now simpler:
rcc_ahb1enr.reg |= RCC_AHB1ENR_GPIOG_EN;
```

At the end, the Rust code looks as simple as the equivalent C code — with all the added benefits of being written in Rust.

After finally achieving the same simplicity as the original C code, a quick glance at the resulting machine code can confirm that no additional overhead has been introduced. Turning an LED on and off repeatedly results in a single instruction if no data is preserved in between (i.e. nothing is OR'ed), otherwise the addresses are allocated inside the registers once and reused.

Figure 7: Implementation for volatile structs, including the or-assign operator.

```
use core::{ops, ptr};

#[repr(C)]
pub struct Volatile<T>(T);

impl<T> Volatile<T> {
    pub fn read(&self) -> T {
        unsafe {
            ptr::read_volatile(&self.0 as *const T)
        }
    }

    pub fn write(&mut self, src: T) {
        unsafe {
            ptr::write_volatile(&mut self.0, src)
        }
    }
}

impl<T> ops::BitOrAssign<T> for Volatile<T>
where T: ops::BitOr<T, Output=T>
{
    fn bitor_assign(&mut self, val: T) {
        let tmp = self.read();
        let new_val = tmp | val;
        self.write(new_val);
    }
}

pub trait VolatileStruct : Sized {
    unsafe fn from_ptr(addr: *mut Self) -> &'static mut Self {
        let item: &'static mut Self = &mut *addr;
        item
    }

    unsafe fn from_addr(addr: usize) -> &'static mut Self {
        // we need the Sized trait here, otherwise the cast doesn't work
        Self::from_ptr(addr as *mut Self)
    }
}
```

6 HARDWARE DESCRIPTION FROM DEVICE TREES

The previous chapter demonstrated an approach of using memory-mapped structures for safe access to hardware peripherals. Assuming those implementations are verified for correctness the largest section for potentially hazardous code remaining is the instantiation: A typo in a memory addresses will go unnoticed during compilation but result in unpredictable behavior at runtime.

Information about memory layouts is available from reference manuals but often in an easily parseable form as well. Generating all addresses and memory-mapped structure instantiations automatically from a reference file shifts the burden of providing a correct list of memory regions toward the provider of the hardware specification. At best, these can be provided and verified by the manufacturer of the hardware itself. The following shows a safe and verified hardware layer in Rust based on the correctness of the specification provided.

6.1 DEVICE TREES

One specification format that is available for some ARM MCUs capable of running Linux are DTs.

DTs are data structures containing hardware descriptions [24] and are used to boot Linux and other operating systems on PowerPC microarchitecture (PPC) platforms, having displaced Open Firmware (OF) about a decade ago [17]. For any new MCU running Linux, DT support is mandatory [24].

Unfortunately not every ARM MCU is announced with Linux support.

A central goal of having DTs is to allow reconfiguring a kernel at boot time, without having to recompile it for different machines that share an instruction set architecture and a large amount of connected hardware [17].

To achieve this, a DT is first described using a Device Tree source file (dts) which contains a human readable description of at least the CPU and memory configuration. Figure 8 contains a short fragment of a dts to illustrate the format.

After compilation a flattened Device Tree (dtb) is then included in the firmware, containing all information in a compact and traversable binary format.

The dts file for the STM32F429I-Discovery can be found in the official kernel sources [12], it is fairly short and includes multiple other files, as it shares component code with other MCUs. The Linux kernel's dts files uncommonly use the C-preprocessor to include dependencies, requiring an extra step before building with the Device Tree compiler (dtc).

The kernel dts files can also be built using the kernel `Makefile` using the `dts` target.

Figure 8: A simple, minimal example of a device tree source file

```
/dts-v1/;

/ {
  #address-cells = <1>;
  #size-cells = <1>;
  model = "fsl,mpc8572ds";
  compatible = "fsl,mpc8572ds";

  cpus {
    #address-cells = <1>;
    #size-cells = <0>;

    cpu@0 {
      device_type = "cpu";
      reg = <0>;
      timebase-frequency = <825000000>;
      clock-frequency = <825000000>;
    };
  };

  memory {
    device_type = "memory";
    reg = <0x00000000 0x20000000>;
  };
};
```

6.1.1 *Format*

A dtb consists of a header containing version information, which CPU is in charge of the bootprocess and offsets of string table, root node and others.

Each tree node has a unique path that is also its name, as well as a list of properties mapping strings to arbitrary values. Properties and nodes are stored in blocks with a start and end marker, nesting is used to store the actual tree structure [18].

Start and end markers are 32-bit integers, fields that are not multiples of 32-bit words are padded to force alignment on word-boundaries for following fields, markers or nodes. Variable-length data is length-prefixed, strings for property names are stored in a string table for deduplication purposes [18].

6.2 DEVICE TREE PARSER

To generate hardware descriptions from DTs, a parser for dtb files was written. The flattened representation was chosen instead of the source because it was simpler to process, contained only relevant information and some vendors ship only dtbs without sources.

While the implementation for this thesis did not require particularly fast or small code, some thought was still given to reusability in later work, possibly on embedded devices. To accommodate this (albeit weak) requirement the library does not use the stdlib, only the libcore. The source code for the DT parser is available at [6].

The parser itself reads the whole tree into a new data structure, instead of traversing the dtb blob itself. Versions that attempted to reduce memory usage by not copying any information from said blob turned out to be impractical due to the fact that there are no backreferences to allow fast upwards traversal of the tree without parsing it once or keeping complex state. This appears to be a design decision, making the tree easier to modify and more compact when stored, but harder to look up the path starting from a leaf node.

6.3 CODE GENERATION

After parsing the tree, it can be printed for debugging purposes, producing output similar to this:

Since the output is fairly long, only small parts are printed here.

```
+
+--- #address-cells = 0x00000001
+--- #size-cells = 0x00000001
+--- model = "STMicroelectronics STM32F429i-DISCO board"
+--- compatible = "st,stm32f429i-discost,stm32f429"
+-+- chosen
| +--- bootargs = "root=/dev/ram rdinit=/linuxrc"
| \--- stdout-path = "serial0:115200n8"
+-+- aliases
| \--- serial0 = "/soc/serial@40011000"
+-+- memory
| +--- device_type = "memory"
| \--- reg = 0x9000000000800000

...
```

```
+-+- soc
| +--- #address-cells = 0x00000001
| +--- #size-cells = 0x00000001
| +--- compatible = "simple-bus"
| +--- interrupt-parent = 0x00000002
| +--- ranges = ""
| +--- dma-ranges = [192, 0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 0]

...

| +-+- pin-controller
| | +--- #address-cells = 0x00000001
| | +--- #size-cells = 0x00000001
| | +--- compatible = "st,stm32f429-pinctrl"
| | +--- ranges = [0, 0, 0, 0, 64, 2, 0, 0, 0, 0, 48, 0]
| | +--- pins-are-numbered = ""
| | +-+- gpio@40020000
| | | +--- gpio-controller = ""
| | | +--- #gpio-cells = 0x00000002
| | | +--- reg = 0x0000000000000400
| | | +--- clocks = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
| | | \--- st,bank-name = "GPIOA"

...
```

Each `compatible` property indicates the kind of peripheral the node represents. Subnodes in general use addressing relative to their parent node.

The information gained can now be used to generate code and memory addresses for a hardware structure:

```rust
// hardware.rs
// ...

struct Hardware {
    // ...

    // PinController _pin_controller:
    gpio_a: gpio::GPIOBank,
    gpio_b: gpio::GPIOBank,
    gpio_c: gpio::GPIOBank,
    gpio_d: gpio::GPIOBank,
    gpio_e: gpio::GPIOBank,
    gpio_f: gpio::GPIOBank,
    gpio_g: gpio::GPIOBank,
    gpio_h: gpio::GPIOBank,
    gpio_i: gpio::GPIOBank,
    gpio_j: gpio::GPIOBank,
    gpio_k: gpio::GPIOBank,

    // ...
}

#[inline(always)]
pub unsafe fn hw() -> Hardware {
    Hardware {
        // ...
        // [compatible: st,stm32f429-pinctrl] /soc/pin-controller
        gpio_a: gpio::GPIOBank::from_ptr(0x40020000 as *mut gpio::GPIOBank),
        gpio_b: gpio::GPIOBank::from_ptr(0x40020400 as *mut gpio::GPIOBank),
        gpio_c: gpio::GPIOBank::from_ptr(0x40020800 as *mut gpio::GPIOBank),
        gpio_d: gpio::GPIOBank::from_ptr(0x40020C00 as *mut gpio::GPIOBank),
        gpio_e: gpio::GPIOBank::from_ptr(0x40021000 as *mut gpio::GPIOBank),
        gpio_f: gpio::GPIOBank::from_ptr(0x40021400 as *mut gpio::GPIOBank),
        gpio_g: gpio::GPIOBank::from_ptr(0x40021800 as *mut gpio::GPIOBank),
        gpio_h: gpio::GPIOBank::from_ptr(0x40021C00 as *mut gpio::GPIOBank),
        gpio_i: gpio::GPIOBank::from_ptr(0x40022000 as *mut gpio::GPIOBank),
        gpio_j: gpio::GPIOBank::from_ptr(0x40022400 as *mut gpio::GPIOBank),
        gpio_k: gpio::GPIOBank::from_ptr(0x40022800 as *mut gpio::GPIOBank),

        // [compatible: st,stm32f42xx-rccst,stm32-rcc] /soc/rcc@40023810\
        // ...
    }
}
// end: hardware.rs
```

Here, in a module `hardware`, a single struct holding all hardware structs is created automatically from the device trees, condensing unsafe code by the client application into a single call to the `hw` function.

6.3.1 *Destructuring hardware*

A good way of using this structure is calling it from the `start` function and passing the return value on to the `main` function:

```
use hardware;

// ...

extern "C" fn _start() {
    main(hardware::hw())
}
```

The client applications main function changes its signature being passed a hardware instance. Destructuring can be used to extract the desired hardware peripherals, discarding the unused:

```
use hardware::Hardware;

fn main(hw: Hardware) {
    let Hardware {
      gpio_a,
      gpio_g,
      ..
    } = hw;
}
```

After the first line of `main` executes, `hw` is no longer usable, as it has been destructured. Its fields `gpio_a` and `gpio_g` are bound to variables of the same name, while all other fields are inaccessible.

Through this construction, all unsafe code is library code or generated automatically, while any number of client apps are simply passed usable abstractions for the existing hardware and can pick and use any number of peripherals without any unsafe blocks or calls.

7 GOING FURTHER

Embedded development is a vast area and with a language as young as Rust a lot of new possibilities have not been fully realized yet. Due to its language features it becomes possible to transfer concepts from more higher-level languages into the constrained space of MCU firmware.

Through its type system, Rust enables techniques in an embedded environment that were not necessarily present before, especially on small machines. State-machines are very important in small-scale applications and can be directly encoded into types, eliminating a class of errors at compile time – without overhead.

The abstract device model shown can be augmented with more layers; the ownership system will prevent reuse of a "scarce" resource by two competing drivers.

The design from mutexes and other guards from the Rust standard library can be transferred to interrupt-based concurrency, resulting in familiar application programming interfaces (APIs).

Additionally there are other formats that could possibly be used to generate the underlying code structures such as CMSIS system view description (SVD), an XML format intended to be used in debuggers in existing IDEs.

7.1 CONCLUDING REMARKS

This thesis demonstrated that Rust embedded development need not be harder from a toolchain perspective than its popular C/C++ counterparts. Another argument can be made in favor of Rust because C and C++ lack a centralized repository for libraries and the unified build system that Rust already provides today.

Some of the tools like `xargo` or `rustup` were greatly improved during or shortly before work on this thesis began, if the progress made by the Rust ecosystem in the embedded field continues at the current rate, Rust may surpass C/C++ in usability in this particular area.

Having established that development is not unnecessarily hard, some potential gains of using Rust to improve the quality of the resulting software were demonstrated on memory-mapped device access, without degradation in performance or ease of use when programming.

Rust has a higher learning curve than many other languages, but it should be seen as an investment to gain access to the powerful abstractions that allowed building the volatile access layer in chapter 5. In the follow-up chapter it was shown that this enables the creation of new tools that can leverage already-present information for a tangible gain in comfort and safety.

Rust embedded development without C is real today. The author sincerely hopes that it will thrive and lead to an improvement in software quality in the future.

BIBLIOGRAPHY

[1] Jorge Aparicio. *xargo: Effortless cross compilation to custom bare-metal targets like ARM Cortex-M*. URL: `https://crates.io/crates/xargo` (visited on 2016-07-18).

[2] Karl J. Åström and Tore Hägglund. *PID Controllers: Theory, Design, and Tuning*. ISA: The Instrumentation, Systems, and Automation Society, 1995. ISBN: 1556175167.

[3] Wikipedia authors. *volatile (computer programming)*. URL: `https://en.wikipedia.org/wiki/Volatile_(computer_programming)` (visited on 2016-07-19).

[4] Alexis Beingessner. "You can't spell trust without Rust." master thesis. Carleton University, 2015.

[5] Marc Brinkmann. *PID controller*. 2016-06-19. URL: `https://github.com/mbr/pid_control-rs`.

[6] Marc Brinkmann. *Reads and parses Linux device tree images*. 2016-04-12. URL: `https://github.com/mbr/device_tree-rs`.

[7] Maik Brüggemann. "Untersuchung der IT-Sicherheit moderner ICS-Systeme am Beispiel der Siemens SIMATIC S7-1200." bachelor thesis. Fachhochschule Münster, 2013.

[8] Maik Brüggemann and Ralf Spenneberg. *PLC Blaster: Ein Computerwurm für PLCs*. 2015. URL: `https://media.ccc.de/v/32c3-7229-plc-blaster`.

[9] Alan Burns and Andrew J. Wellings. *Real-time systems and programming languages : Ada 95, real-time Java and real-time POSIX*. 3. ed. International computer science series. Includes bibliographical references and index. - Previous ed.: 1997; : No price : Formerly CIP. Harlow: Addison-Wesley, 2001. ISBN: 0-201-72988-1; 978-0-201-72988-7.

[10] Stephen Cass. *The 2015 Top Ten Programming Languages*. 2015-07-20. URL: `http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages`.

[11] Committee on the Past and Present Contexts for the Use of Ada in the Department of Defense and Computer Science and Telecommunications Board and National Research Council. *Ada and Beyond: Software Policies for the Department of Defense*. National Academies Press, 1997. ISBN: 0309055970.

[12] Maxime Coquelin. *stm32f429-disco.dts*. 2015. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/arch/arm/boot/dts/stm32f429-disco.dts (visited on 2016-07-23).

[13] Alex Crichton. *Stabilize catch_panic*. URL: https://github.com/alexcrichton/rfcs/blob/stabilize-catch-panic/text/0000-stabilize-catch-panic.md (visited on 2016-07-19).

[14] Zakir Durumeric et al. "The Matter of Heartbleed." In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC '14. Vancouver, BC, Canada: ACM, 2014, pp. 475–488. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663755. URL: http://doi.acm.org/10.1145/2663716.2663755.

[15] Mathew Garret. *I bought some awful light bulbs so you don't have to*. 2016-02-24. URL: https://mjg59.dreamwidth.org/40397.html.

[16] Chris Garry. *Original Apollo 11 Guidance Computer (AGC) source code for the command and lunar modules*. 2016-07-10. URL: https://github.com/chrislgarry/Apollo-11.

[17] David Gibson and Ben Herrenschmidt. *An overview of the concept of the device tree and device tree compiler*. Tech. rep. URL: http://www.ozlabs.org/~dgibson/papers/dtc-paper.pdf.

[18] Benjamin Herrenschmidt and Becky Bruce. *Booting the Linux/ppc kernel without Open Firmware*. 2006. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/plain/Documentation/devicetree/%20booting-without-of.txt?h=v4.5&id=b562e44f507e863c6792946e4e1b1449fbbac85d.

[19] Brian W. Kernigham. *The C Programming Language*. 1988. ISBN: 0131103628.

[20] Karl Koscher et al. "Experimental Security Analysis of a Modern Automobile." In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 447–462. ISBN: 978-0-7695-4035-1. DOI: 10.1109/SP.2010.34. URL: http://dx.doi.org/10.1109/SP.2010.34.

[21] Aleksandr Matrosov, Eugene Rodionov, David Harley, and Juraj Malcho. "Stuxnet under the microscope." In: *ESET LLC (September 2010)* (2010).

[22] Nicholas D Matsakis and Felix S Klock II. "The rust language." In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM. 2014, pp. 103–104.

[23] OpenSSL. *TLS/SSL and crypto library*. 2016-07-11. URL: https://github.com/openssl/openssl.

[24] Thomas Petazzoni. *Your new ARM SoC Linux support check-list!* 2012. URL: http://www.elinux.org/images/a/ad/Arm-soc-checklist.pdf.

[25] ST Microelectronics. *STM32F4 DSP and standard peripherals library*. 2014. URL: http://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32-standard-peripheral-libraries/stsw-stm32065.html.

[26] ST Microelectronics. *RM0090 Reference manual. STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM ® -based 32-bit MCUs*. 2015. URL: http://www.st.com/resource/en/reference_manual/dm00031020.pdf.

[27] ST Microelectronics. *STM32F427xx, STM32F429xx*. 2016. URL: http://www.st.com/content/ccc/resource/technical/document/datasheet/03/b4/b2/36/4c/72/49/29/DM00071990.pdf/files/DM00071990.pdf/jcr:content/translations/en.DM00071990.pdf.

[28] The Rust Team. *Cargo*. URL: https://crates.io/ (visited on 2016-07-13).

[29] The Rust Team. *The Rust Programming Language*. 2016. URL: https://doc.rust-lang.org/book/.

[30] texane. *stm32 discovery line linux programmer*. 2016-06-25. URL: https://github.com/texane/stlink.

[31] The Rust Team. *rustup.rs - The Rust toolchain installer*. URL: https://www.rustup.rs/ (visited on 2016-07-18).

[32] The Rust Team. *The Rust programming language*. 2016. URL: https://www.rust-lang.org.

[33] The Rust Team. *The Rust Reference*. 2016. URL: http://doc.rust-lang.org/reference.html.

[34] Aaron Turon. *Tracking issue for 'asm' (inline assembly)*. URL: https://github.com/rust-lang/rust/issues/29722 (visited on 2016-07-13).

[35] Aaron Turon and Niko Matsakis. *Stability as a Deliverable*. 2014-10-30. URL: https://blog.rust-lang.org/2014/10/30/Stability.html.

[36] Joseph Yiu. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, Third Edition*. Newnes, 2013. ISBN: 0124080820.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LYX:

`https://bitbucket.org/amiede/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

`http://postcards.miede.de/`

Final Version as of July 29, 2016 (`classicthesis`).