Didrik Rokhaug

Rust applications for Zephyr

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth, Sebastian Bøe
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Didrik Rokhaug

Rust applications for Zephyr

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth, Sebastian Bøe
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Preface

I would like to thank my supervisors, Sverre Hendseth at NTNU and Sebastian Bøe at Nordic Semiconductor for their help with this project. Sverre for asking hard questions, and Sebastian for help navigating Zephyr's source code and the world of compiler options.

Didrik Rokhaug

Problem Statement

Rust is a new system programming language, that can be used in embedded systems. A requested feature that would help improve Rust's usability in embedded systems is the ability to use a mature embedded operating system. One such operating system is Zephyr, a new open source real-time operating system project hosted by the Linux Foundation.

There are several challenges associated with making a Rust application for Zephyr. Bindings need to be created so that Rust knows how to call Zephyr's APIs and the Rust application need to be linked together with Zephyr.

The goal of this work is to explore the possibility of writing Rust applications using the Zephyr embedded operating system, and if possible make a proof of concept application showing how a Rust application for Zephyr can be made.

The student shall:

- Document relevant parts of Rust and Zephyr's toolchain and build process with regards to a Rust application on Zephyr
- Find challenges with making a high quality interface to Zephyr in Rust
- Make a proof of concept application in Rust that uses Zephyr

Abstract

Rust is a new systems programming language with features that make it suitable for embedded development. However, it is currently not possible to use it together with embedded operating systems. One such operating system that is well suited for the same applications as Rust is Zephyr. In this report we will present a proof of concept application that shows that Rust and Zephyr can be used together. We will also outline what work that need to be done in order for Rust application development for Zephyr to be a viable option.

Sammendrag

Rust er ett nytt systemprogrammeringsspråk med egenskaper som gjør det godt egnet for utvikling av applikasjoner for innevevde datasystemer. En ting som mangler for at dette skal bli lettere er muligheten til å bruke sanntids operativsystemer. Ett slikt operativsystem er Zephyr. I denne rapported vil vi presentere en proof of concept applikasjon som viser at Rust og Zephyr kan brukes sammen. Vi vil også legge frem hvilket arbeid som må gjøres for at applikasjonsutvikling i Rust for Zephyr skal være et levedyktig alternativ.

Contents

Chapter 1

Introduction

Rust is a new systems programming language that provides a lot of low-level control, while also maintaining a high abstraction layer and a focus on safety. It can be used on embedded devices, but its ecosystem is still young. One requested feature is the possibility to use an embedded operating system together with Rust[7]. One possible embedded OS that has been mentioned in these discussions is Zephyr. While Zephyr itself is a new project itself, it is supported by several big corporations in the industry and is based on older projects, which means that a lot of its source code is already mature. Zephyr does also support a wide range of features and many different platforms. This makes Zephyr an attractive operating system to use together with Rust.

In addition to being of great help for the Rust embedded ecosystem, making a Rust application that uses Zephyr is interesting for other reasons. As Zephyr is written in C and not in Rust, ensuring that the two languages interact properly will be necessary. Considering that Zephyr is a complicated system which uses several advanced C and compiler features, the interactions between Rust and Zephyr become more complicated. Zephyr's build process is also complicated, and incorporating Rust's build system will therefore also be a challenge.

Another reason that making Rust applications is an interesting challenge is that even though a solution to many of the problems that may arise might not be very complicated, the knowledge needed to find the solutions are not easily accessible. The reason for this is that most developers do not need to know all the details about how the tools they use work, and these tools do not have much scientific significance. The knowledge is therefore passed from generation to generation of a small group of developers that is responsible for maintaining and improving these tools.

In this report we will take a look at how Zephyr and Rust might be able to work together, make a proof of concept application and outline the work needed to make Rust applications on Zephyr a viable option.

Chapter 2

Background

2.1 Rust

Rust is a new programming language sponsored by Mozilla. Rust 1.0, the first stable release, was released May 15. 2015[15]. It focuses on safety, concurrency and speed, while also providing low-level control. This makes it suitable for embedded development. Rust has many of similarities with C, but it also takes inspiration from functional languages such as Haskell. In addition to this, it also has many of tools that are inspired by the tools of dynamic languages like Ruby, Python and JavaScript[15].

Rust aims to provide memory safety without garbage collection. This safety is achieved using two concepts called ownership and lifetimes. When a new variable is created, it is bound to a name. We then say that that name "owns" the variable. Later, the variable can be bound to a new name, and ownership is then passed over. Tracking ownership allows Rust to know when a variable is no longer needed. This is because when the owner goes out of scope, the variable is no longer bound to any name, and can therefore not be accessed by the code any longer. The variable can therefore safely be dropped, and its memory freed. If we want to let someone else use the variable for a while, e.g. a function, we can let the function borrow the variable by giving it a reference to the variable. However, in order to prevent race conditions, we can either hand out multiple references that cannot mutate the variable, or we can only hand out one single mutable reference. This way, Rust is able to prevent simultaneous reads and writes to a variable, ensuring that it always is in a consistent state. In addition to this, by tracking the lifetime of the bindings and borrows, that is, the time a variable is borrowed, or the scope in which the binding or borrow exists, Rust can ensure that there are no bindings to a variable that is dropped.

As Rust is supposed to be usable everywhere C is used today; it also has built-in mechanisms for working with libraries written in C. In order to call a function written in C from Rust, there are a couple of things that need to be done: the function must be declared as an external function using the C ABI, and the library where the function is defined must be linked in. If the function uses

```c
// File: library.c

struct c_struct {
    int a,
    int b
}

// We must tell the C compiler what the Rust function looks like
extern int rust_function(int a);

int c_function(struct c_struct arg) {
    return rust_function(arg.a);
}
```

Listing 2.1: An example of using a Rust function, and exporting a function to Rust

any composite types such as structs or enums, these types must also be defined in Rust. Also, as the Rust compiler cannot check that the C function upholds Rust's safety guarantees, when we call the function it must be inside a `unsafe` block. When making a Rust function to call from C, we also need to mark that the function must use the C ABI. In addition, we need to tell the Rust compiler not to mangle the name of the function. An example of using the foreign function interface is shown in listing 2.1 and listing 2.2.

Sometimes, Rust's safety guarantees are a bit too strict, or Rust might be unable to prove that something is safe, and therefore will not allow it, even though the programmer may know that the operation is safe. In these cases, Rust provides the `unsafe` keyword. Inside a block of code marked as unsafe, Rust allows us to do a few things normally not allowed[24]:

- Dereference a raw pointer
- Access or modify a mutable static variable
- Call an unsafe function or method
- Implement an unsafe trait

Calling foreign functions falls under the "Call an unsafe function or method" point. Static variables in Rust are variables which have a static lifetime, i.e., they last through the whole program. In addition to this, their scope is global (to the module where they are defined). This means that they, in theory, can be accessed by multiple threads. Therefore, if the static variable is mutable, it is unsafe to access it, as another thread might be modifying it at the same time. As a part of Rust's lifetime analysis, Rust is able to prevent reading of uninitialized memory. Such memory access is considered unsafe, as the memory can contain any value. The reason for this being a problem becomes clear if we consider an uninitialized variable of type `bool`. The memory the `bool` is made of can only have two valid values: 0 and 1. However, the uninitialized piece of memory can have any value, which leads to a problem if we try to read the value of the `bool`. In safe code (code that is not inside an `unsafe` block)

```rust
// File: main.rs

// We need to tell the Rust compiler to pack the C
// struct in the same way that the C compiler would.
#[repr(C)]
struct c_struct {
    a: i32,
    b: i32,
}

#[no_mangle]
pub extern "C" fn rust_function(a: i32) -> i32 {
    a + 1
}

// Inside this block we can define external functions
// that must be called using the C ABI.
extern "C" {
    fn c_function(arg: c_struct) -> i32;
}

fn main() {
    let foo = c_struct{a: 1, b: 2};
    unsafe {
        println!("foo.a + 1: {}", c_function(foo));
    }
}
```

Listing 2.2: An example of usage of Rust's foreign function interface

this cannot happen, as Rust can detect that the variable is uninitialized, and it is impossible to get references to uninitialized variables. However, Rust also supports raw pointers (standard C pointers). As these pointers can be set to point to any address, Rust is unable to guarantee that they always point to valid initialized memory. Therefore, dereferencing a raw pointer is considered unsafe, and must be done inside an unsafe block[21].

2.1.1 Rust's toolchain and other useful tools

Rust's compiler "rustc" is based on the LLVM compiler backend. The LLVM project consists of several subprojects; the most relevant are the LLVM Core, Clang, compiler-rt, and LLD. LLVM Core is a set of libraries that provide a source and target-independent optimizer and code generation support for many different CPUs. Clang is a C compiler that uses LLVM as a backend. Clang does this by producing an LLVM intermediate representation (LLVM IR), that is given to LLVM Core, which then generates the final binary code. Clang can also be used as a platform to make source-level tools. The LLVM IR contains some high-level operations that are not supported on all platforms. For these platforms, compiler-rt provides implements routines for these operations[4]. LLD is a new linker, that can be used as a replacement of system linkers. It has been used as the default linker on ARM Cortex-M targets since August 28. 2018[17].

To help with calling the compiler, and handling dependencies Rust provides Cargo. Cargo is primarily a package manager that handles the dependencies of a Rust project but can also be used to specify how the Rust project is to be built. This includes building and linking to C libraries. To provide this functionality Cargo uses a couple of configuration files and an optional build script. The first configuration file, called the manifest file, contains information about the project. What is the name of the project, and what version is it? Who wrote it? It also has information about what (Rust) dependencies is used, and what feature flags they use; what other libraries are to be linked, and where to find them. It also specifies how the project itself is built. This includes options such as what type of library it should be compiled to (static or dynamic), what optimization level should be used, and how should panics be handled.

The other configuration file ".cargo/config" and is used to configure Cargo itself. It can also be used to set certain flags when compiling for a specific target. The options in this file can be spread across several files, upwards in the file hierarchy. This way, we can specify options both globally and per project. The final way of influencing the build is a build script. These are usually called "build.rs", but any file can be specified in the manifest file. The build file contains a Rust program that is run before the project is built. The most common use cases for these build scripts are to generate code at compile time, build code written in another language, and to specify how to link to native languages. The build script can pass commands and options to Cargo using its standard output[9].

Rust uses a hierarchical module structure, meaning that the module structure represents a tree, with a top-level module that contains children modules and so on. Any function, type, trait or submodule (an item) defined in a module is private by default but can be made public by the keyword `pub`. The exception

is the top level of the library (crate in rust parlance), who exports nothing by default. An item is always public to its immediate parent module, and the parent's children module[24]. A module can be written in the same file as its parent, but they are often given their own files. In order to simplify the search for these files, the name of the file is decided by the Rust compiler. A file containing a module must either have the same name as the module or be called mod.rs and exist in a folder with the same name as the module. E.g., a module called foo can either be written in the file of the parent module, in foo.rs or mod/foo.rs. Similarly, Cargo expects the top level file in a library to be called lib.rs, and in an executable main.rs.

As rustc targets LLVM rather than an actual architecture, Rust can be compiled for most platforms that LLVM can generate code for. This also means that rustc is a cross compiler by default. In order to compile for a different target, all we need to do is to get the correct version of the standard library and specify which target we want to compile for. However, if we want to use the standard library (or parts of it) on a target that Rust does not support, or we want to change how the standard library is built, we have to build the standard library ourselves. This can be done with the tool Xargo, which imitates the behavior of Cargo but also builds the standard library. Xargo will compile the standard library and call the compiler with the right flags so that the new standard library is linked in. Xargo will by default only compile the platform-independent core of the standard library, but if other parts of the library are needed, they can also be compiled[12].

To know how to compile for a specific target rustc uses a target specification file. This approach was first suggested in RFC 131[14]. Having the target specifications written in its own configuration file allows users to edit targets easily, and thus easily port Rust to new platforms. The configuration file consists of a JSON object which specifies what LLVM target is to be used, along with the endianness, pointer and integer width and data layout of the target. In addition to this, there are fields for specifying the operating system, environment (which libc), vendor, and architecture. These options are used for conditional compilation by Rust. Finally, there are a lot of optional settings that can be set. Among these are what linker to use and what arguments the linker should be used with, what output formats are available and some settings that control what optimizations are applied. A full description of the target specification can be found in [8].

While the target specifications can be written in JSON, the supported targets are built in directly in rustc. There are two reasons for this:

1. It eases distribution by avoiding the need for configuration files.
2. The specifications become immutable, and can, therefore, be trusted to be correct.

However, rustc can print out the specifications in JSON format. The specification of the target thumbv7em-none-eabi, which is the target for a bare metal Cortex M4 without a floating point unit can be seen in listing 2.3.

Another helpful tool for Rust developers is (Rust-)Bindgen. As much of the work with calling C functions comes from providing the C prototypes in Rust, Bindgen

```
{
  "abi-blacklist": [
    "stdcall",
    "fastcall",
    "vectorcall",
    "thiscall",
    "win64",
    "sysv64"
  ],
  "arch": "arm",
  "data-layout": "e-m:e-p:32:32-i64:64-v128:64:128-a:0:32-n32-S64",
  "emit-debug-gdb-scripts": false,
  "env": "",
  "executables": true,
  "is-builtin": true,
  "linker": "rust-lld",
  "linker-flavor": "ld.lld",
  "llvm-target": "thumbv7em-none-eabi",
  "max-atomic-width": 32,
  "os": "none",
  "panic-strategy": "abort",
  "relocation-model": "static",
  "target-c-int-width": "32",
  "target-endian": "little",
  "target-pointer-width": "32",
  "vendor": ""
}
```

Listing 2.3: The target specification file for the thumbv7em-none-eabi target.

was made to generate these bindings automatically. Bindgen uses libclang, a library interface to Clang. This way, Bindgen can generate an abstract syntax tree (AST) of the code, and find all the function and type definitions that are used[1]. By using libclang Bindgen also gets the benefit of running the preprocessor on the code, so if there are several versions of a function based on some configuration options, Bindgen will see the correct one.

2.2 Zephyr

Zephyr is an open source real-time operating system (RTOS) based on Wind River's VxWorks Microkernel profile for VxWorks[11]. The project is hosted by the Linux Foundation, which provides neutral governing, ensuring that all stakeholders are represented. Among the supporters of the project are Intel, Linaro, NXP, and Nordic Semiconductor[13]. While the project itself is only from 2016[16], the history of the kernel goes back to 2001 when Wind River acquired Eonic Systems' digital signal processor RTOS Virtuoso[28]. Zephyr uses the Apache 2.0 license, which makes it possible to use it in commercial solutions, and to incorporate proprietary IP[11].

Zephyr focuses on Internet of Things (IoT) applications and therefore supports several protocols and standards such as Bluetooth Low Energy, 802.15.4, Wi-Fi, CoAP and MQTT[11]. Another important area of focus is IoT is security. As a part of this focus, the code is carefully reviewed and tested, including by static analysis. As many IoT devices have very little memory, Zephyr is made to be modular, so that we do not have to pay for something we do not use.

2.2.1 Configuration

Zephyr supports many different platforms. These platforms support different features, have a different amount of memory, different sets of peripherals, and everything has different addresses. Zephyr does also support many features that we might not want to include in our project due to memory, power, or complexity constraints. In order to support this high degree of options for the users, Zephyr uses two different systems to provide configurability. The first is device trees, which are used to describe the hardware Zephyr runs on. The other is Kconfig which is used for actual configuration of the kernel.

The device tree is not used to configure Zephyr directly but informs Zephyr about what peripherals are available on the given target, what their addresses are, what interrupts they have, etc[23]. Linux also uses device trees for similar purposes. A difference between Linux's and Zephyr's usage of the device trees, however, is that Linux compiles the device trees using a device tree compiler (DTC) into a binary blob that is loaded into memory by the bootloader. When the Linux kernel boots up, it will read this binary blob, and use the information to set up its drives correctly. Zephyr, on the other hand, uses the device tree compiler only to combine all the device tree files into a new device tree file. This file is then parsed together with a set of YAML files by a python script to generate C preprocessor `#define`s used by the drivers. Part of the device

```
leds {
    compatible = "gpio-leds";
    led0: led_0 {
        gpios = <&gpio0 17 GPIO_INT_ACTIVE_LOW>;
        label = "Green LED 0";
    };
    led1: led_1 {
        gpios = <&gpio0 18 GPIO_INT_ACTIVE_LOW>;
        label = "Green LED 1";
    };
};
```

Listing 2.4: A piece of the device tree for the nrf52_pca10040 board.

```
/* led_0 */
#define DT_GPIO_LEDS_LED_0_GPIO_CONTROLLER "GPIO_0"
#define DT_GPIO_LEDS_LED_0_GPIO_FLAGS 0
#define DT_GPIO_LEDS_LED_0_GPIO_PIN 17
#define DT_GPIO_LEDS_LED_0_LABEL "Green LED 0"
#define DT_GPIO_LEDS_LED0_GPIO_CONTROLLER
↪  DT_GPIO_LEDS_LED_0_GPIO_CONTROLLER
#define DT_GPIO_LEDS_LED0_GPIO_FLAGS
↪  DT_GPIO_LEDS_LED_0_GPIO_FLAGS
#define DT_GPIO_LEDS_LED0_GPIO_PIN DT_GPIO_LEDS_LED_0_GPIO_PIN
#define DT_GPIO_LEDS_LED0_LABEL DT_GPIO_LEDS_LED_0_LABEL
#define LED0_GPIO_CONTROLLER DT_GPIO_LEDS_LED_0_GPIO_CONTROLLER
#define LED0_GPIO_FLAGS DT_GPIO_LEDS_LED_0_GPIO_FLAGS
#define LED0_GPIO_PIN DT_GPIO_LEDS_LED_0_GPIO_PIN
#define LED0_LABEL DT_GPIO_LEDS_LED_0_LABEL
```

Listing 2.5: Part of the output generated from nrf52_pca10040's device tree.

tree for nrf52_pca10040 is shown in listing 2.4 and part of the output is shown in listing 2.5. The biggest reason for this different use is that the binary blob generated by the DTC is huge. This becomes a problem when used on memory constrained devices.

Kconfig is also used by Linux to configure the kernel. Kconfig is a language used to describe configuration options and how they relate to each other. Unlike the device trees, we need to specify what options are available. When specifying the option, we can specify the type, e.g., number, bool, string; set a default value, and specify relations to other configuration options. The graphical tool Makeconfig can be used to select the configurations, or we can write it in a file. A Zephyr application contains a file called prj.conf which contains the configuration for the project. In addition to prj.conf, the configuration can be spread across multiple files. This is used by Zephyr to set options based on the selected board automatically. The output from the configuration files is the header file autoconf.h, a part of the autoconf.h generated for Zephyr's "Blinky" sample for nrf52_pca10040 is shown in listing 2.6.

```
/* Generated by Kconfiglib (https://github.com/ulfalizer/Kconfiglib) */
#define CONFIG_BOARD "nrf52_pca10040"
#define CONFIG_SOC "nRF52832_QFAA"
#define CONFIG_SOC_SERIES "nrf52"
#define CONFIG_NUM_IRQS 39
// ...
#define CONFIG_GPIO_NRFX 1
#define CONFIG_GPIO_NRF_INIT_PRIORITY 40
#define CONFIG_GPIO_NRF_P0 1
```

Listing 2.6: Parts of the autoconf.h generated when building the "Blinky" sample for the nrf52_pca10040 board.

2.2.2 Zephyr's toolchain

Building Zephyr can be divided into several steps. First, Zephyr uses CMake to check that the build environment is OK and to generate instructions on how to build Zephyr itself. CMake also generates the macros and defines Zephyr uses to configure itself. Once this is done, Zephyr uses Make or Ninja to build the source code itself. The work that CMake does is called the configuration phase. The next phase is called the build phase. This phase is driven by either Make or Ninja, and can also be divided into several steps: the generation step, the build step, the link step, and the post-build step. This process is illustrated in fig. 2.1.

During the generation step, Zephyr generates more C code. This generated code includes parts of Zephyr's system call dispatch infrastructure and some files used to keep track of kernel objects. During the build step, Zephyr gets compiled into a series of statically linked archives. The two most interesting archives are libzephyr.a and libkernel.a. These libraries are where most of the operating system itself is defined. The rest are the various subsystems and drivers that are used by the application or this particular configuration of the operating system. In addition to this, there are a few hardware dependent libraries. The application is also built at this stage as a static library called libapp.a.

During the link step, all these libraries are linked together. When the application and the rest of Zephyr are linked together into an executable, this executable is analyzed to extract the interrupt service routines, and generate a new C file containing an interrupt table. All the libraries are then linked together again, to create the final executable. In the post-build step, a script runs to check that the executable looks correct, the executable is converted to formats better suited for flashing the hardware, and some debug output is generated. A full list of the commands run by the build script is shown in appendix A and in make.log in the accompanying zip file.

2.2.3 System calls

This section is taken from my project thesis "An alternative approach to Zephyr's system call dispatch, using Rust"[26].

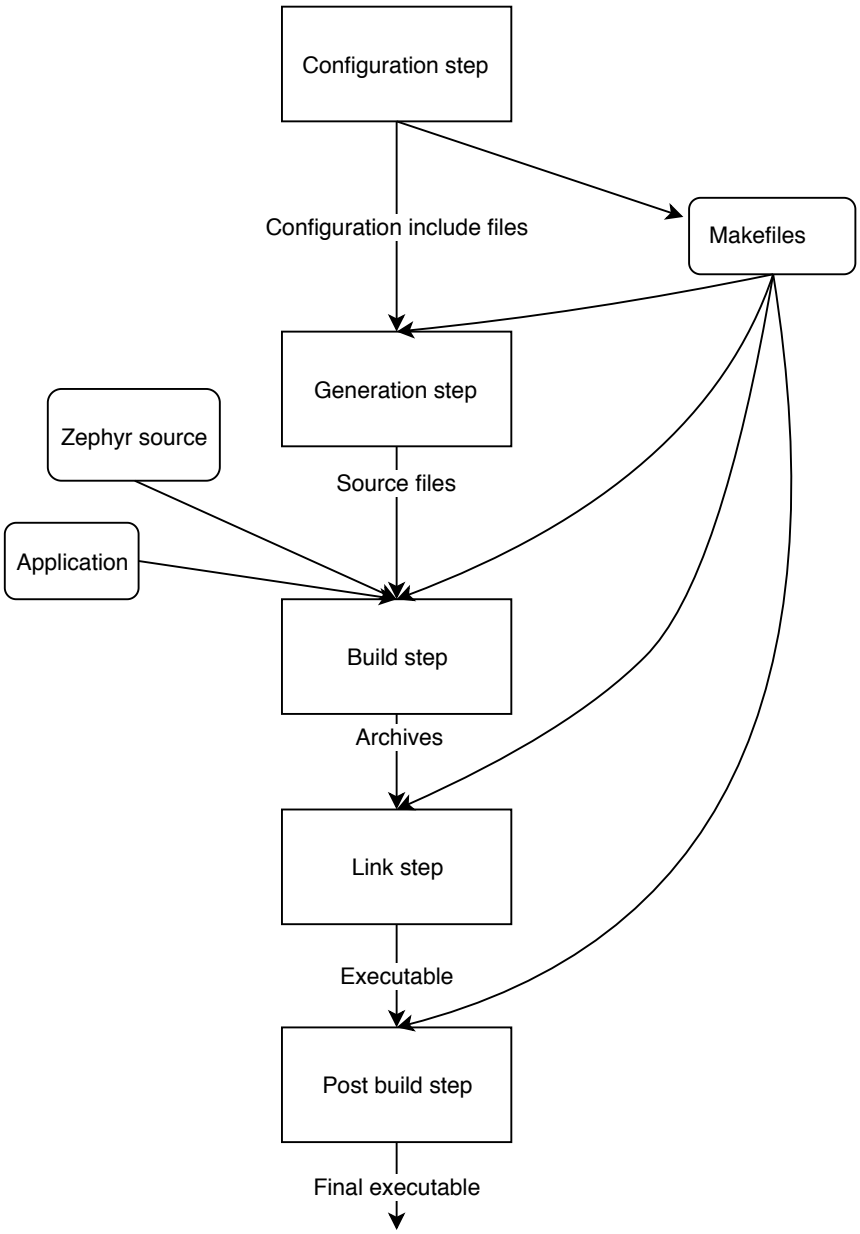


Figure 2.1: Visualization of Zephyr's build process.

As user mode threads have limited privileges, there are several things they are not allowed to do. User-mode threads run in a non-privileged CPU state[19]. This means that there are certain instructions and operations they cannot execute. In addition to this, they do not have access to the entire memory, and they cannot modify kernel objects[18]. Kernel objects are split into three classes[18]:

- A core kernel object
- A thread stack
- A device driver instance

Core kernel objects are semaphores, threads, pipes, etc. Device driver instances are structs that contain a set of function pointers that are used by the specific driver.

In order for a user mode thread to access any of these objects, or instructions, they have to use system calls. These are special functions defined in the Zephyr kernel that can be called from user mode but run in kernel mode. To make it easier to write system calls, Zephyr generates the code for switching to kernel mode and back automatically.

A Zephyr system call consists of three components: a C prototype, a handler function, and an implementation function. The C prototype is listed in a header that is included by the application and is prefixed with `__syscall`. This is the prototype of the system call that is called by the application, but it is not manually implemented. Instead, code is generated that does the privilege escalation, checks that the given parameters are valid using the handler function, and then runs the implementation function, where the actual code of the system call is. The handler function is declared using a macro that expands to the correct function declaration. Inside the handler function, a set of macros is used to verify that the input arguments are valid. When the inputs are validated, the handler function calls the implementation function and returns the result of the system call. The implementation function is, as the name implies, where the system call is implemented. Thanks to the handler function, the implementation function can assume that the inputs are valid and that the calling thread has access to the service that the system call provides.

While the `__syscall` marker is defined as a C preprocessor macro, it does not expand to anything useful. Instead, several Python scripts search all the header files in Zephyr's `include` directory. When the scripts find the `__syscall` marker, they store the name, return type, argument type, and names and the name of the header where file the system call was declared in a JSON file. Then another Python script parses the JSON file and outputs the C code and macros used to dispatch the system call. This includes putting a pointer to the handler function in a table of all system calls and writing a macro that declares the system call.

The macro expands to the function that the user mode thread calls when it wants to perform the relevant system call. The function checks if it was called from user mode or not, and then either calls an architecture specific function for checking that the system call is a valid system call and running the system call in kernel mode, or runs the system call directly. On the other hand, if

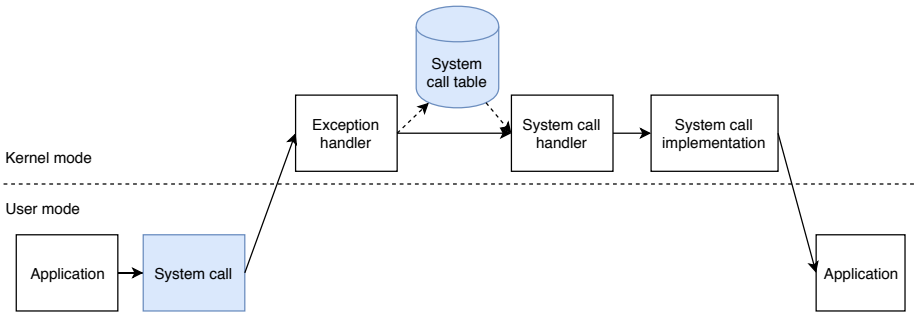


Figure 2.2: A diagram showing the control flow of Zephyr's system call dispatch. The shaded boxes are generated by Python scripts.

the calling thread already runs in kernel mode, the implementation function is called directly.

If we are in user mode, we need to switch to kernel mode. This includes setting the CPU in a privileged state. In order to go from an unprivileged state to a privileged state, we need to go via an exception[20]. This is because you cannot change to privileged mode from unprivileged mode, but exceptions run in privileged mode, and can, therefore, be used to execute privileged code. However, before the code can be executed, it needs to be validated to ensure that the user cannot execute arbitrary code in privileged mode. Zephyr does this by not passing a pointer to the system call function to the exception handler, but instead, it passes an index to a table that contains all the system call handler functions. The exception handler then checks whether the index is within the boundaries of the system call table or not. If the index is fine, then the exception handler changes to privileged mode and sets up the stack and registers needed to do the system call. When the system call returns, the CPU is set back to an unprivileged state, before the function returns. A diagram of the control flow is given in fig. 2.2.

2.3 Object files and linking

When compiling a source file, the output can be given in many different formats depending on what we want to achieve. Common for most of these forms, however, is that they are so-called object files, or files containing machine code, even though it may not be executable, as it is not able to tell where to jump to when it is supposed to jump. While these files are meant to be read by a computer, and not a human, sometimes we have to read them in order to understand what is wrong with our program. In this section, we will first take a look at the formats produced by Zephyr's build system: ELF object files, and archives or statically linked libraries. Then we will take a look at some tools that lets us extract useful information and interact with these files.

There are a lot of different formats for object files, depending on what platform and toolchain are used and what the object files are used for. On UNIX like systems, the executable and linkable format (ELF) is used both for relocatable

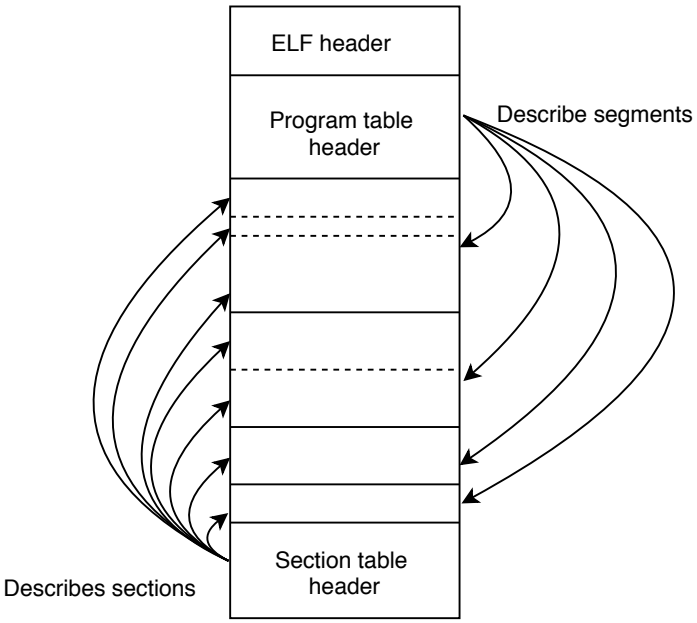


Figure 2.3: Layout of an ELF file.

linkable files, loadable executables, and loadable shared libraries[25]. This is done by having two ways of reading the ELF file, as shown in fig. 2.3. The file starts with an ELF header, which contains information about the file itself, i.e., what kind of file it is, how to read it (the format is cross-platform, and therefore contain information about byte order and word size), and where to find the program and section header tables. It is these two tables that make the ELF format so versatile. After the ELF header follows the program header table, which contains descriptions of memory segments that are contained in the file. These segments can be loaded into memory in order to run the ELF as an executable. The section header table, on the other hand, describes the sections which exist in the file. The linker uses the sections in order to link together several ELF files. This duality means that an ELF file might be linkable and loadable at the same time.

When the linker links together two object files, it takes the sections in the files as input and places them in segments in the output file. In addition to this, it also resolves any symbol references. The code in a section might refer to code or data in a different section. An ELF section contains different information depending on its type[25]:

- PROGBITS: Program contents, including code, data, and linking debugger information.
- NOBITS: Similar to PROGBITS, but no data is stored in the file itself. Used for zero allocated data.
- SYMTAB and DYNSYM: Symbol tables. SYMTAB sections hold symbols used for the regular linker. DYNSYM holds symbols used for dynamic linking.

```
typedef struct {
    Elf32_Word st_name;      // Index into the string table.
    Elf32_Addr st_value;     // Value of the symbol. Section relative in a
                             // relocatable file, absolute in an executable.
    Elf32_Word st_size;      // Object or function size.
    unsigned char st_info;   // The 4 least significant bits of this value is
                             // the symbols type, i.e. if it is a data object
                             // function or something else. The 4 most
                             // significant is the binding: Local, global or weak
    unsigned char st_other; // Unused.
    Elf32_Half st_shndx;     // Section number, ABS, COMMON or UNDEF.
} Elf32_Sym;
```

Listing 2.7: Definition of an ELF symbol table entry[27]. `Elf32_Word` and `Elf32_Addr` are 4 bytes long, while `Elf32_Half` is 2 bytes.

- STRTAB: A string table. Rather than storing the name of a section or symbol in their corresponding tables, they instead have pointers to these tables. This way, all entries in other tables have a constant length.
- REL and RELA: Relocation information. REL contains entries that add the relocation information to a base value stored in the code or data, while RELA entries include the base values themselves.
- DYNAMIC and HASH: Dynamic linking information and run-time symbol hash table.

A symbol table entry contains information needed by the linker to allocate storage for the symbol and to do symbol resolution. A symbol table entry is shown in listing 2.7. A symbols binding can be local, global, or weak. A local binding means that it is only accessible from within the same object file (module). This means that symbols in other modules cannot refer to this symbol. A global symbol is visible everywhere. A weak symbol is in many ways the same as a global symbol, but if there are any global symbols with the same name, the global symbol is used instead of the weak symbol. If a weak symbol is undefined, it defaults to zero. If the linker encounters multiple weak definitions of a weak symbol, it is free to choose any of the definitions.

Most programs include some common functionality. While this functionality could be included in the form of object files, this quickly leads to several downsides. If we have a large object file that defines a lot of functionality, say a math library, and we only use a small part of it, our executable will still include the entire math object file. If on the other hand, we split the math object file into smaller ones, we would have to tell the linker to include every single one that we need, and if any of the smaller object files depend on another file that file would also have to be included.

A solution to this problem is static libraries. On UNIX systems, these are simply archives of object files. This means that the static library file can be seen as a directory, containing separate files[25]. In addition to this, the library contains some added directory information used to speed up searches in the library. As the added information is itself appearing in the archive as a separate file, the

archive can, in theory, contain any set of files, but in practice, this format is only used as archives of object files.

When the linker links a set of object files and libraries the order they appear on the command line matters as the linker will process them in order[25]. The linker will include the entirety of object files, and it will keep a list of encountered symbols, both defined and undefined. When it encounters a library, it will look through the library for symbols matching its still undefined symbols. When the linker finds a symbol it needs, it will include the entire module (object file inside an archive) that contains the definition. As this module might contain new undefined symbols, the linker will have to look through the library again. What is important to note is that the linker will not go back and look at earlier libraries. This can cause problems if we include two libraries that contain a circular dependency.

In addition to regular undefined external symbols, a library can also contain weak external symbols. These are symbols that will resolve to external definitions if a definition exists, but it will not cause a new module to be included on its behalf. This is useful when, for instance, you have a library that contains routines for handling floating point numbers. As the floating point routines take a lot of space, we do not want to include them in our executable if we do not need them. If we have a function that might need to handle floating point numbers, this function can use weak references to the floating point routines in order to only include them if we use the floating point library in another way (e.g., referencing a symbol in the floating point library to ensure that it is included).

A library can also contain weak definitions. These define a global symbol similar to regular global definitions, but if the symbol is defined elsewhere, the weak definition will be ignored[25]. Weak definitions are not much used but can be used to define default behavior for functions that the user of a library should normally implement.

There are various tools for creating and working with static libraries and object files. On UNIX systems “ar” is the commonly used tool for creating archives. Earlier it did not add the directory containing the symbol table; this was instead done by “ranlib”. In order to read the symbol table in an archive or object file, we can use “nm”. nm will also list the type of the symbol and the symbol value. If we want more information about an object file we can use “readelf” or “objdump”. These two programs do most of the same things, but objdump relies on the “Binary File Descriptor” (BFD) library, while readelf does not. As the GNU linker and compiler also depends on the BFD library, if there were a bug in it, it would be hard to discover. readelf is, therefore implemented separately. If we need to copy, translate or make changes to object files, we can use “objcopy”.

Chapter 3

Generation of bindings to Zephyr's API

In order to be able to use Zephyr from Rust, we need to have bindings to Zephyr's API so that the Rust compiler knows which functions exist and how to call them. In this chapter, we will take a look at how bindings can be made to Zephyr and what challenges there are.

Usually, when making bindings to a C library, people create two Rust libraries, or crates as they are called in Rust parlance, one with the same name as the C library and the other with the same name but ending with -sys. The reason for this is that calling C functions are considered unsafe, an so using the bindings directly would lead to many `unsafe` blocks. To avoid that, the unsafe direct bindings are usually wrapped in a safe Rust layer that ensures that the C API is used correctly. This also gives the opportunity to create a more Rust like API. The crate with the same name as the library is the one that is included by the users. This library depends on the -sys crate, which contains the C bindings. While the top crate usually is written by hand, the -sys crate is usually created by Bindgen.

The typical way of using Bindgen to make a -sys crate is to use bindgen as a library in the build.rs script. This way, if the library contains any platform specific things, these things will be configured correctly, without the crate owner having to build a different crate for all possible targets. A simple build.rs file that uses bindgen can be seen in listing 3.1. First, a Bindgen `Builder` is created. Builders are a common pattern for ensuring that a struct is created correctly, especially when they have optional fields. Bindgen uses a C header file in order to know which bindings to create. If we want to create bindings to more header files, we can make a wrapper that includes all the header files that we want to generate bindings to. While the build script is run in the crate root, it is considered bad form to generate files in the root or source directory, therefore, provides a directory in the build folder. We get the path to this output directory as an environment variable. Finally, we write the bindings into the file bindings.rs. This file can then be included in the crates lib.rs using Rust's `include!` macro.

```rust
// File: build.rs

fn main() {
    let bindings = bindgen::Builder::default()
        .header("header.h")
        .generate()
        .expect("Unable to generate bindings"); // Simple error handling

    let out_path = std::path::PathBuf::from(env::var("OUT_DIR").unwrap());
    bindings.write_to_file(out_path.join("bindings.rs"))
        .unwrap();
}
```

Listing 3.1: A simple build.rs that uses Bindgen.

As Bindgen only takes one header file as input, while Zephyr's API is spread across multiple, we have to make a single header file that contains all the symbols we want to bind to. This can be done by making a header file that includes all the header files that we want to create bindings to. This works as Bindgen is using libclang internally, thus having most of the functionality of an actual C compiler. The easiest way of doing this is to read Zephyr's "include" directory, and include all header files we find in our wrapper. This functionality is easily added to the build script. In order to compile Zephyr, one has to run (on Windows) or source (on Mac/Linux) a script that sets up a few environment variables that Zephyr needs. One of these is `$ZEPHYR_BASE`, which is the path to Zephyr's root directory. Using this environment variable, we can easily find the path to Zephyr's include directory.

Attempting to generate bindings with this build script fails, as Bindgen is unable to find the included header files. While this can be fixed by writing the full path of the header files we are including, this only delays our problem, as the header files themselves include header files which Bindgen is unable to find. As these errors come from libclang as it is trying to parse the header files, we need to tell libclang where to find the files. This can be done by adding include arguments to Bindgen's invocation of libclang. Bindgen provides a method called `clang_arg` to its `Builder` struct that does this. In order to find what arguments to add, we can read Zephyr's compiler invocations. When CMake creates Makefiles, it adds the possibility of adding a "verbose" flag to our Make invocations. This tells Make that we want to see all the commands that Make runs, so Make prints them to the standard output. Piping this output into a file, we can get a log of all the commands that get run in order to compile Zephyr. In appendix A the logfile gotten from compiling Zephyr's "Blinky" sample for the nRF52 development kit.

Adding the arguments used to compile the Zephyr application to Bindgen requires converting the GCC arguments that Zephyr uses, to the corresponding Clang arguments that Bindgen and libclang uses. Also, as we are not doing a whole compilation, but only parse the code, libclang does not use all the arguments. Most of the arguments are easily translated, but the arguments that specify the architecture were either unused or unknown.

Simply setting up the necessary compiler arguments is not enough. There are two aspects of Zephyr's API that makes it hard to generate bindings automatically. The first is the configurability of Zephyr. Some libraries have multiple implementations, and others only make sense for certain targets. This creates problems when we try to compile all the headers at the same time. Another challenge is that Zephyr uses a lot of `static inline` functions in its API. As these functions do not exist after Zephyr has been compiled, we are unable to call these functions from Rust.

3.1 Configurations

One possible way of solving this problem is to create a "super configuration" which enables all the options so that all the headers compile. While this approach looks viable at first glance, it is not always easy to know what configuration to set to satisfy a given compiler error, several of the options might be mutually exclusive, and types might change based on what options are set. One typical example of types that change depending on configurations are structs that gain more fields if a configuration option is set, one such struct is shown in listing 3.2. This will lead to Rust and C having a different layout of the same struct, which can cause unexpected and hard to find bugs, assuming it compiles at all. In order to ensure that the Rust application and Zephyr have the same struct layouts, the "super configuration" would have to be used when compiling Zephyr as well, leading to bloated executables at best, and a configuration that does not work for the target at worst.

```
1  /**
2   * @ingroup thread_apis
3   * Thread Structure
4   */
5  struct k_thread {
6
7          struct _thread_base base;
8
9          /** defined by the architecture, but all archs need these */
10         struct _caller_saved caller_saved;
11         /** defined by the architecture, but all archs need these */
12         struct _callee_saved callee_saved;
13
14         /** static thread init data */
15         void *init_data;
16
17         /**
18          * abort function
19          * @req K-THREAD-002
20          * */
21         void (*fn_abort)(void);
22
23 #if defined(CONFIG_THREAD_MONITOR)
24         /** thread entry and parameters description */
```

```
25          struct __thread_entry entry;
26
27          /** next item in list of all threads */
28          struct k_thread *next_thread;
29  #endif
30
31  #if defined(CONFIG_THREAD_NAME)
32          /* Thread name */
33          const char *name;
34  #endif
35
36  #ifdef CONFIG_THREAD_CUSTOM_DATA
37          /** crude thread-local storage */
38          void *custom_data;
39  #endif
40
41  #ifdef CONFIG_THREAD_USERSPACE_LOCAL_DATA
42          struct _thread_userspace_local_data *userspace_local_data;
43  #endif
44
45  #ifdef CONFIG_ERRNO
46  #ifndef CONFIG_USERSPACE
47          /** per-thread errno variable */
48          int errno_var;
49  #endif
50  #endif
51
52  #if defined(CONFIG_THREAD_STACK_INFO)
53          /** Stack Info */
54          struct _thread_stack_info stack_info;
55  #endif /* CONFIG_THREAD_STACK_INFO */
56
57  #if defined(CONFIG_USERSPACE)
58          /** memory domain info of the thread */
59          struct _mem_domain_info mem_domain_info;
60          /** Base address of thread stack */
61          k_thread_stack_t *stack_obj;
62  #endif /* CONFIG_USERSPACE */
63
64  #if defined(CONFIG_USE_SWITCH)
65          /* When using __switch() a few previously arch-specific items
66           * become part of the core OS
67           */
68
69          /** _Swap() return value */
70          int swap_retval;
71
72          /** Context handle returned via _arch_switch() */
73          void *switch_handle;
74  #endif
```

```
75          /** resource pool */
76          struct k_mem_pool *resource_pool;
77
78          /** arch-specifics: must always be at the end */
79          struct _thread_arch arch;
80  };
```

Listing 3.2: A struct whos definition changes based on the configuration of the project.

As crates are distributed as source files and compiled together with the application that uses them, rather than being distributed as compiled object files, we could make a way of reading the configuration used for the project, and use that configuration to make the bindings. This would require a way of knowing which header files are necessarily based on the configuration. This also complicates the interaction with Zephyr's build system. Either, we have to call Kconfig and DTS ourselves, to generate our own configuration files, or Zephyr's build system would have to know that the application is written in Rust and that bindings need to be created. Another downside with this approach is that it would be hard to document the bindings or make them discoverable using auto-completion features of IDEs and text editors as the bindings themselves does not exist until the application is compiled. This would make it harder for the application developer to know how to use the Zephyr APIs. This problem might be solved by a higher level Rust crate, that creates a safe Rust API on top of the raw bindings. For this to work would require either Rust to know what the configuration options mean so that Rust can use its conditional compilation features to not include unused parts of Zephyr's API, or we would require the linker and application programmer not to call functions that are not supported by the given configuration.

3.2 `static inline` functions

Another challenge with creating bindings to Zephyr's API is that a lot of the functions are declared as `static inline`, and implemented directly in the header files. This means that when a C module includes any of these functions, the function body gets pasted at the place of the call, instead of an actual function call happening. The `static` keyword means that the function is not exported outside of the module. This is necessary as the function is defined in the header file and therefore defined in any module that includes the header file. If the `static` keyword is not used, the linking will fail, as the linker will not know what to do with all the definitions of the function. While this inlining of functions has a speed benefit (at the cost of code size), it also means that the linker never sees these functions at all. This becomes a problem later, as we try to call these functions from Rust. As the Rust compiler does not know C (and vice versa), any interaction with foreign functions must happen via a function call. However, as these functions are inlined where used, and then discarded by the compiler, there are no functions for the linker to link to.

There are a couple of possible solutions to this problem. There are a couple of

compiler options called `-fkeep-static-functions` and `-fkeep-inline-functions` these options include the static and inline functions in the object files, instead of being discarded by the compiler. These flags do not solve our problem; however, as the functions are still marked as having internal linkage, meaning that they cannot be referenced from outside the module.

Another approach is to make wrapper functions that are not marked as `static inline`. This approach would work for obvious reasons, but it is not without problems. First, we need to create all the wrapper functions. This might be done automatically, but some of the functions, especially in driver APIs, are just convenience layers on top of a struct with function pointers. This means that the in some situations the entire API is defined in a header file that does not have a corresponding implementation file, but instead is used to abstract several hardware dependent implementations. This leads to a problem about where to put the wrapper files. We still need to edit Zephyr's source code. While Zephyr is open source, so that editing the source code is possible, we would require the Zephyr developers to support adding these wrapper functions. From the Zephyr projects point of view, this might not be something they are willing to do, as it adds more code to maintain, and it adds an expectation that they support Rust applications. Another way of adding wrapper functions is to fork Zephyr's source code so that we have our own version. This would put the maintenance burden on us, instead of Zephyr's developers. Another problem with this approach is that the wrapper functions would have to have different names than the function they wrap unless we can do something smart with the linker. Having different names would make it harder to use our bindings, as the functions the user would have to call would be different from the functions the user want to call.

Another option for solving the `static inline` problem is to handle it from the Rust side. As actually calling a `static inline` function is impossible, what we have to do instead is to implement it in Rust. For the moment, this has to be done manually, but a way of calling `static inline` functions has been a requested feature for a long time, and there has been some progress on the issue[2]. There is work done on tools both for inlining C code in Rust, to be compiled by a C compiler; and to generate Rust code from C. If these tools become powerful enough and integrated into Bindgen it would be possible to use the Rust version of the inlined functions in the Rust application. This way, it might also be possible to keep the inlining, thus not losing much of the performance.

As we need a way to generate bindings to `static inline` functions and a way to handle different configurations in order to generate bindings, This is not possible to do automatically at the moment. In a more limited case, it is possible to know enough about the configurations to generate bindings using Bindgen, but the `static inline` functions is a real roadblock for this project.

Chapter 4

A Rust application on Zephyr

When first starting programming in a new language, or with new tools, it is customary to first write a "Hello, World!" program. In the embedded world, the equivalent is "Blinky", a simple program that blinks an LED by toggling an output pin in a loop. Therefore, as a first step of showing that one can make Rust applications for Zephyr, porting Zephyr's "Blinky" sample is a natural step.

Creating a Rust application for Zephyr, in theory, is as simple as creating bindings to Zephyr's API, writing an application that uses these bindings, building it and then linking in Zephyr. The Blinky application is quite simple, consisting only of a couple of `#defines` that gives better names to peripherals defined in the device tree, some initial configuration and a loop which toggles the LED and then sleeps. The C source code is given in listing 4.1.

4.1 The bindings

While I was not able to generate bindings to Zephyr as a whole, Bindgen can easily handle the output of DTC and Kconfig, as these files only contain simple C `#define`s and do not depend on any other files. In addition to the defines, four functions are being called:

1. `device_get_binding`
2. `gpio_pin_configure`
3. `gpio_pin_write`
4. `k_sleep`

Of these four functions, `device_get_binding` and `k_sleep` are system calls and the two `gpio_pin_*` functions are `static inline`. The system call dispatch mechanism in Zephyr is described in section 2.2.3. What this means for in this situation is that the functions the users call (i.e., `device_get_binding` and `k_sleep`) does not really exist as anything other than a `static inline` function that checks if we are in user mode or not, and then either calls a handler function before the system call is invoked, or calls the implementation function directly.

```c
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr.h>
#include <device.h>
#include <gpio.h>

#define LED_PORT LED0_GPIO_CONTROLLER
#define LED	 LED0_GPIO_PIN

/* 1000 msec = 1 sec */
#define SLEEP_TIME	 1000

void main(void)
{
	int cnt = 0;
	struct device *dev;

	dev = device_get_binding(LED_PORT);
	/* Set LED pin as output */
	gpio_pin_configure(dev, LED, GPIO_DIR_OUT);

	while (1) {
		/* Set pin to HIGH/LOW every 1 second */
		gpio_pin_write(dev, LED, cnt % 2);
		cnt++;
		k_sleep(SLEEP_TIME);
	}
}
```

Listing 4.1: Zephyr's Blinky sample.

```c
// file: /zephyr/include/gpio.h

__syscall int gpio_config(struct device *port, int access_op, u32_t pin,
              int flags);

static inline int _impl_gpio_config(struct device *port, int access_op,
                    u32_t pin, int flags)
{
    const struct gpio_driver_api *api =
        (const struct gpio_driver_api *)port->driver_api;

    return api->config(port, access_op, pin, flags);
}
```

Listing 4.2: Definition of `gpio_config`.

In our case, we run the entire application in kernel mode, and we can therefore for simplicity replace our calls to `device_get_binding` and `k_sleep` with calls to `_impl_device_get_ginding` and `_impl_k_sleep` without a problem. If we had a good way to interface with `static inline` functions that would obviously be better.

The two `gpio_pin_*` functions are a simple abstraction over more general `gpio_*` functions that can work with either a pin or a port. These functions are also system calls, but in this case, the implementation functions are also declared as `static inline`. The declaration of these functions can be seen in listing 4.2. The device returned from `device_get_binding` contains a pointer to a struct containing a set of function pointers. These function pointers are the actual driver API. As the implementation function also is `static inline` we cannot simply call the implementation function as we did with `device_get_binding` and `k_sleep`. As there are no gpio.c file so there was no obvious place to write a wrapper function, and the functions are relatively simple, these functions were implemented in Rust. In addition to the functions, we must also tell rustc what the data types look like.

```rust
mod zephyr_sys {
    // Module containing Rust definitions of common C types
    use super::cty;

    // Module containing the output of Kconfig and DTC
    pub mod config;

    pub const GPIO_DIR_OUT: i32 = 1 << 0;

    #[repr(C)]
    pub struct device_config {
        name: *const cty::c_char,
        init: Option<unsafe extern "C" fn (foo: *mut device) -> cty::c_int>,
        config_info: *const cty::c_void,
    }

```

```
17      #[repr(C)]
18      pub struct device {
19          pub config: *mut device_config,
20          pub driver_api: *mut cty::c_void,
21          pub driver_data: *mut cty::c_void,
22      }
23
24      // A C #define used inside Zephyr's gpio library
25      const GPIO_ACCESS_BY_PIN: cty::c_int = 0;
26
27      struct _snode {
28          next: *mut _snode,
29      }
30
31      type sys_snode_t = _snode;
32
33      type gpio_callback_handler_t =
34          Option<unsafe extern "C" fn(port: *mut device, cb: *mut gpio_callback,
35                                      pins: u32)>;
36
37      // This union is defined inside the struct gpio_callback
38      // and not given a name in Zephyr.
39      #[repr(C)]
40      union gpio_callback_anon_union {
41          pin_mask: u32,
42          pin: u32,
43      }
44
45      #[repr(C)]
46      struct gpio_callback {
47          node: sys_snode_t,
48          handler: gpio_callback_handler_t,
49          my_field: gpio_callback_anon_union,
50      }
51
52      type gpio_config_t =
53          Option<extern "C" fn(port: *mut device, access_op: cty::c_int, pin: u32,
54                               flags: cty::c_int) -> cty::c_int>;
55      type gpio_write_t =
56          Option<extern "C" fn(port: *mut device, access_op: cty::c_int, pin: u32,
57                               value: u32) -> cty::c_int>;
58      type gpio_read_t =
59          Option<extern "C" fn(port: *mut device, access_op: cty::c_int, pin: u32,
60                               value: *mut u32) -> cty::c_int>;
61      type gpio_manage_callback_t =
62          Option<extern "C" fn(port: *mut device, callback: *mut gpio_callback,
63                               set: bool) -> cty::c_int>;
64      type gpio_enable_callback_t =
65          Option<extern "C" fn(port: *mut device, access_op: cty::c_int, pin: u32)
66                               -> cty::c_int>;
```

```
67      type gpio_disable_callback_t =
68          Option<extern "C" fn(port: *mut device, access_op: cty::c_int, pin: u32)
69                               -> cty::c_int>;
70      type gpio_api_get_pending_int = Option<extern "C" fn(dev: *mut device) -> u32>;
71
72      #[repr(C)]
73      struct gpio_driver_api {
74          config: gpio_config_t,
75          write: gpio_write_t,
76          read: gpio_read_t,
77          manage_callback: gpio_manage_callback_t,
78          disable_callback: gpio_disable_callback_t,
79          get_pending_int: gpio_api_get_pending_int,
80      }
81
82      fn _impl_gpio_config(port: *mut device, access_op: cty::c_int, pin: u32,
83                           flags: cty::c_int) -> cty::c_int
84      {
85          unsafe {
86              let api = (*port).driver_api as *const gpio_driver_api;
87              return ((*api).config)(port, access_op, pin, flags);
88          }
89      }
90
91      // This function is static inline in C, so we define it here
92      #[no_mangle]
93      pub fn gpio_pin_configure(port: *mut device, pin: u32, flags: cty::c_int)
94                                -> cty::c_int
95      {
96          return _impl_gpio_config(port, GPIO_ACCESS_BY_PIN, pin, flags);
97      }
98
99      fn _impl_gpio_write(port: *mut device, access_op: cty::c_int, pin: u32, value: u32)
100                          -> cty::c_int
101     {
102         unsafe {
103             let api = (*port).driver_api as *const gpio_driver_api;
104             return ((*api).write)(port, access_op, pin, value);
105         }
106     }
107
108     // This function is static inline in C, so we define it here
109     #[no_mangle]
110     pub fn gpio_pin_write(port: *mut device, pin: u32, value: u32) -> cty::c_int
111     {
112         return _impl_gpio_write(port, GPIO_ACCESS_BY_PIN, pin, value);
113     }
114
115     // Bindings to external C functions
116     extern "C" {
```

```
117             pub fn _impl_device_get_binding(name: *const cty::c_char) -> *mut device;
118
119             pub fn _impl_k_sleep(duration: i32);
120         }
121     }
```

Listing 4.3: Rust bindings needed for a Rust port of Zephyr's Blinky sample.

By putting the bindings in their own module, we get the module shown in listing 4.3. The function pointers wrapped by an `Option` uses what is called the "null-pointer" optimization[6]. In Rust, very few types and no references are allowed to be NULL. This sometimes creates an opportunity for the compiler to do an optimization that saves some space. If we have an enum that only has two variants, where one of the variants are zero-sized (i.e., holds no data), and the type of the other variant cannot be NULL, the compiler can drop the tag used to identify the variant, and instead, check if the value is NULL. In our case, if we were to receive a function pointer from C and it was wrapped in an `Option` we would be forced to check if we had gotten a valid pointer (`Some()`) or a NULL pointer (`None`). In this case, it does not matter, but in other cases, this is a way Rust can provide some safety across an FFI boundary if the bindings are created in a smart way.

4.2 The application

The "Blinky" application is quite simple. The C application starts by redefining a couple of symbols created based on the device tree. While Rust does not have a way of textually replacing symbols similar to the C preprocessor, Rust's `const` can be used in much the same way. The biggest difference is that we need to declare the type of the `const`. Following the constants is the main function. This function is almost identical to its C counterpart except for a couple of details. As we are calling foreign functions, we need to be inside an `unsafe` block. Technically we only need to be inside one when actually calling the functions, but for simplicity and readability we have put all the code inside the `unsafe` block in this case. Ideally, we would have a safe wrapper around all the foreign functions, and thus not need `unsafe` in the application at all, but for this simple prototype, this approach is fine.

The other noteworthy difference is the loop. In C, they use a `while` loop that never stops, and inside it, they have a variable that counts upwards. If the variable is odd, they set the pin high, if it is even they set it low. While we can do the same in Rust, there are some differences to be aware of. While C has two types of loops: `while` and `for`, Rust has three: `while`, `for` and `loop`. The first two are very close to their C counterpart, but the `loop` is used when we have infinite loops. This gives the compiler more information about the intent of our code, helping us catch errors at compile time.

While the `loop` is the closest thing to C's `while (1)`, it is not very idiomatic Rust to have a mutable state variable. Instead, we can use a `for` loop with an infinite range. This solution might cause a problem, if we compile Rust code using the built-in "debug" profile as arithmetic operations then are checked for

```rust
const LED_PORT: *const cty::c_char = zephyr_sys::config::LED0_GPIO)CONTROLLER;
const LED: u32 = zephyr_sys::config::LED0_GPIO_PIN;

const SLEEP_TIME: i32 = 1000;

#[no_mangle]
extern "C" fn main() {
    unsafe {
        let mut cnt = 0;
        let dev = zephyr_sys::_impl_device_get_binding(LED_PORT);

        zephyr_sys::gpio_pin_configure(dev, LED, zephyr_sys::GPIO_DIR_OUT);

        loop {
            zephyr_sys::gpio_pin_write(dev, LED, cnt % 2);
            cnt = cnt.wrapping_add(1);
            zephyr_sys::_impl_k_sleep(SLEEP_TIME);
        }
    }
}
```

Listing 4.4: Rust port of Zephyr's Blinky sample.

overflow. As a 32-bit number cannot hold infinite large numbers, at some point, we will try to store a number in our counting variable that is larger than our counting variable can hold. This overflow will lead to our Rust code crashing, while the C code will simply wrap around, and go from the largest possible number to the smallest possible number. This problem is not limited to ranges and would indeed also happen with the `loop`. The easiest way to solve the problem is to use the method `wrapping_add()`, which behaves similar to normal addition in C. The "problem" also goes away if we compile the application with optimization, as the checks for overflows then are removed. In total, we get the application shown in listing 4.4.

4.3 Building the application

As we are running the application on an ARM Cortex M4 CPU, we need to tell Cargo to cross-compile our application. This is done by specifying the target thumbv7em-none-eabi in the command line. The Rust standard library does not support this target. We, therefore, have to add an attribute to our application, so the standard library is not included. In addition to this, we need to specify what should happen if our code panics. We can do that by adding a function marked with a special attribute. In our case, we put a `loop` inside our function so that nothing worse happens if we panic. The panic handler is listed in listing 4.5.

In addition, we want our application to be compiled as a static library. This is done by adding a line specifying the crate-type in Cargo's manifest file. It is not

```
1 loop
```

Listing 4.5: A simple panic handler.

necessary to create a new target, but Xargo must be used regardless. In order to compile the Rust application closer to the C application, we can add a line to the specification of thumbv7em-none-eabi that specifies that the relocation model is static. This tells the compiler that it does not have to produce position independent code (pic), which means that the code works correctly regardless of where it is loaded into memory[25]. This feature is useful for shared libraries, but as we only are running one application, we have full control over the memory layout. We also have the option of changing the default archiver to the one that Zephyr uses.

4.4 Linking to Zephyr

The final step in making our proof of concept Rust application is to link it to Zephyr. Two approaches can be taken here. The first is to let Rust drive the linking, by telling it which libraries to link, and where to find them, the other is to replace Zephyr's libapp.a with our own, and then re-run the necessary commands, thus simulating a Zephyr driven process. Letting Rust drive the linking does present some challenges. As Zephyr uses the resulting executable from the linking to generate more code and then doing another pass of linking, it is important that the correct information exists in the executable. When building a Rust application, we do not have direct control over the linker command. Instead, we can set various configuration options, especially in the target specification to control the linker. While we can include a linker script (which Zephyr does use), we are not able to control the linking sufficiently without considerable effort. The biggest problem is that the .intList sections from the input libraries are discarded. These are the sections later used to generate the interrupt table, and therefore, while the Rust driven compilation succeeds, later stages fail.

If we instead use the Zephyr driven approach, we encounter other problems. The Rust static library contains all the dependencies the Rust application need to run (except for Zephyr's libraries which we have not linked in yet). These dependencies include some parts of the Rust standard library and their dependencies. Among these we find a small part of the C standard library which defines four functions:

- memcpy
- memmove
- memset
- memcmp

As Zephyr also includes the C standard library (libc), we get a linker error complaining about multiple definitions of these functions. Usually, if a linker encounters two definitions of a symbol from two different libraries, it will simply

keep the first, and ignore the second definition, as the symbol reference is already satisfied[25]. However, in this case, libapp.a and libc.a are both included after the `-whole-archive` linker flag, which means that all the symbols in the archive should be kept. There are three ways to solve this problem. The first is to find a way of not exposing the Rust version of libc, the second is to not expose Zephyr's version of libc, and the third is to change the linker command so that the linker can resolve the symbols.

Rust can be compiled to use different versions of libc. On Linux, we can compile rust to both dynamically link in glibc, but also to statically link in musl[10]. There are however next to none documentation on how this is done, and adding support for a new libc implementation would probably require changes in rustc. We also rather want to keep Zephyr's libc implementation as Zephyr probably makes more assumptions about it than Rust does. It is also no good way of making Rust not expose its libc dependency. Finally, as Rust only use the four functions mentioned above from libc, if we remove Zephyr's libc implementation, there are a lot of other functions in libc that now are undefined. We, therefore, need to make the linker resolve this conflict, rather than remove it.

The first step we need to take in order to let the linker resolve our conflict is to move either libapp.a or Zephyr's libc implementation from the `-whole-archive` section of the linker command. If we move Zephyr's libc then the linker will use Rust's definitions of the four mem- functions. However, in the same module as Zephyr's definitions of the four mem- functions there are also other functions defined. As these functions are not defined by Rust, these symbols might be included in order to satisfy an external reference in another Zephyr library. When this happens the linker will include the whole module, and we are back to square one wiht multiple definitions of the four mem- functions.

The solution is, therefore, to move our libapp.a out from the `-whole-archive` section of the linker command. We do, however, need to be careful about where we are putting the libapp.a. If we put it before the `-whole-archive` we get the same problems as if it were inside, as any Rust reference to the mem- functions will include the Rust definitions. If we put it last, only modules containing still undefined symbols will be included. As Zephyr defines a main function in case the application does not, our main function will not be included the whole application will be discarded. If we remove Zephyr's fallback main implementation, our application will be included, but any functions used by the application defined in libkernel.a as these functions have already been deemed unnecessary by the linker. This happens because when the linker includes symbols from libraries it will only include symbols which are yet undefined, and it will not go back to check old libraries if they contain any symbols that are needed by later included modules.

If we instead put libapp.a before libkernel.a we also get the problem with the missing main. Our main function is used by libkernel.a, so if we want it included, we must list libkernel.a first. However, this leads to the linker using libkernel.a's fallback main function and our main is not included. To solve this circular dependency, we can add a flag to linker command that adds main to the list of undefined symbols, and list libapp.a before libkernel.a. What will then happen is that the linker will look for a main function in all the libraries listed. It will not find any definition of it before it encounters out libapp.a. While including

our main function in order to resolve its undefined main reference, it will also get references to the functions we need in libkernel.a. When it later encounters libkernel.a's main function it will not use it as it already has a definition of main. This gives us the linker command shown in listing 4.6. Using this linker command, we are able to successfully link our Rust application to Zephyr. In order to get a working executable, we have to run the last commands in the log file, changing the second linker command in a similar fashion.

```
ccache /opt/zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/
    bin/arm-zephyr-eabi/arm-zephyr-eabi-gcc CMakeFiles/
    zephyr_prebuilt.dir/misc/empty_file.c.obj -o
    zephyr_prebuilt.elf -T linker.cmd -Wl,-Map=/zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    zephyr.map -u_OffsetAbsSyms -u_ConfigAbsSyms -umain -
    Wl,--whole-archive ../app/libapp.a libzephyr.a arch/arm/
    core/libarch__arm__core.a arch/arm/core/cortex_m/
    libarch__arm__core__cortex_m.a arch/arm/core/cortex_m/
    mpu/libarch__arm__core__cortex_m__mpu.a lib/libc/
    minimal/liblib__libc__minimal.a subsys/bluetooth/
    common/libsubsys__bluetooth__common.a subsys/bluetooth
    /host/libsubsys__bluetooth__host.a subsys/bluetooth/
    controller/libsubsys__bluetooth__controller.a subsys/
    net/libsubsys__net.a drivers/gpio/libdrivers__gpio.a
    drivers/entropy/libdrivers__entropy.a -Wl,--no-whole-
    archive ../app/libapp.a kernel/libkernel.a CMakeFiles/
    offsets.dir/arch/arm/core/offsets/offsets.c.obj -L"/
    opt/zephyr-sdk/sysroots/armv5-zephyr-eabi/usr/lib/arm-
    zephyr-eabi/6.2.0/armv7e-m" -L/zephyr/samples/basic/
    blinky/build/nrf52_pca10040/zephyr -lgcc -Wl,--print-
    memory-usage -mthumb -nostdlib -static -no-pie -Wl,-X
    -Wl,-N -Wl,--gc-sections -Wl,--build-id=none -Wl,--
    orphan-handling=warn -mabi=aapcs
```

Listing 4.6: The modified linker command used to link the Rust application to Zephyr. Added options are **bold**, while removed options are ~~stricken through~~.

Chapter 5

Discussion

As shown in chapter 4, it is possible to run a simple application written in Rust that uses Zephyr. There are, however, several open questions. First, in order to actually make Zephyr usable from Rust, we need bindings. These should ideally be made automatically, both to reduce the necessary maintenance work, but also to reduce the chance of any errors in the bindings.

Another open question is why the application must be built by Xargo. Xargo should, in theory, behave similar to Cargo, except that Xargo will build a new sysroot if necessary. However, when building a new sysroot should not be necessary, and even though Xargo builds a new one regardless, it should not be different from the one shipped with rustc. Having to use Xargo is also a bit problematic as Xargo was only meant as a temporary solution to be able to build applications for embedded targets. As Cargo now supports Cortex M devices, active development of Xargo has stopped[5]. There is work on tools to replace Xargo, and the Cargo team want to expand Cargo to be able to build new sysroots, but this work might still take years to complete[22].

We also need a simple way to build our application and link it with Zephyr if we want to make it viable to write Zephyr application in Rust. While a Rust driven approach might work, it is much easier to let Zephyr drive the process. While it is difficult to make Zephyr build our Rust application instead of a C application, Zephyr's build system lets us add third-party libraries as dependencies, including specifying how they are built. We can, therefore, create a C application that calls the entry point of our Rust application, which is added as a third party library. The only problem with this approach is that we lose control over how our library is linked in, leading to a conflict between Rust's and Zephyr's definition of the four mem- functions.

While we solved the multiple definitions problem by manipulating the linker command, there is another solution that was not discussed. Instead of finding a way of solving the problem, we could remove it entirely. By using objcopy, we can make changes to our archive. While we could remove the definitions entirely, making them weak will also solve our problem. As the linker then will choose the strong symbol over the weak instead of giving us an error. The resulting executables from these two approaches should not be very different.

While the four mem- functions give us a problem, we also have a collision between libgcc and compiler-rt. Both these libraries implement arithmetic operations that the processor cannot handle. The difference between them is that libgcc is implemented as part of the GNU Compiler Collection[3], while compiler-rt is part of the LLVM project. The reason for this collision not causing any linker error is that the linker would have satisfied any undefined references by the first implementation it encountered, and ignore the second as it is in a library. However, if we found a way of not exporting the mem- functions, we could also do the same for compiler-rt in order to ensure that we do not run into any compatibility issues, albeit at the cost of increased size of our executable.

The application presented in this report shows that it is possible to create a simple application in Rust that uses Zephyr. However, Zephyr does have more features than only blinking an LED. Zephyr uses interrupts and a multithreaded environment. While `k_sleep` is implemented as a switch to an "idle thread" that does power management, we have not really seen how Rust and Zephyr interact when there are a lot of interrupts and thread changes. This might cause trouble due to different usage of the stack between Rust and Zephyr.

Zephyr does also support compile-time initialization of many types of kernel objects. From the application's viewpoint, there is a C preprocessor macro that defines the kernel objects, but taking a look at the definition of some of these macros there also appears to be some more compiler and linker magic going on. As Rust does not have a preprocessor similar to C, this might not work from Rust at all. However, one possible solution is that if the Rust application is compiled as a third party library by Zephyr, and the Rust entry point is called from a Zephyr application written in C, then this C application could also define any necessary kernel objects that should be initialized at compile time. In order to be able to access these object from the Rust application, we would need bindings to them, and these bindings would probably have to be written by hand (possibly with the help of a Rust macro). The C application could also possibly be written by a Cargo build script. This requires that we can compile the Rust application before Zephyr tries to compile the application, but if this turns out to be possible, the application programmer would only have to care about Rust. While this would be an inconvenience for the application developer, we would gain the possibility of using the macros defined by Zephyr.

In addition to the C preprocessor, there are other things about C that translates poorly to Rust. In Zephyr and C, in general, it is usual to have a library that defines some context struct to hold internal data. The user of the library declares a variable to hold such a struct and then passes a pointer to a library function that initializes the struct. The other functions in the library then take a pointer to the struct in order to keep track of the state of the library. This approach is similar to objects in object-oriented languages, except that the data and behavior are separated and the user has to handle the memory of the data. This is, however, very unidiomatic Rust. Rust, in general, dissuades the use of raw pointers, and while this can be handled by a Rust wrapper over the C API Rust is no fan of uninitialized memory. In safe Rust, it is not possible at all to get a reference to uninitialized memory. This means that in safe Rust, we cannot declare a struct, and have a function initialize it for us. This leads to a problem when interfacing with C code that expects a pointer to a piece

of memory where the function can initialize a struct. We are also unable to initialize these structs ourselves, as the struct might be very complicated, and are internal details of the C module.

There might be ways to hide these details from the user of the Zephyr bindings with a well crafted Rust wrapper layer. There are (unsafe) functions that can give you a piece of uninitialized memory, although one of them was recently deprecated as there was no way of using it safely at all[21]. The usage of uninitialized structs become especially problematic if they also are supposed to be `static`, that is global with a lifetime equal to that of the program. This means that the value of a `static` variable must be known at compile time. As this is not the case when we rely on a C function to initialize it for us, we have a problem. There exists a Rust macro that lets us postpone the initialization of a `static` variable that might be able to solve our problem, but it is primarily intended to initialize data structures that live on the heap, but we want to have as a `static` variable. Other than the use of `static` variables in Zephyr's APIs, we must be cautious in general when using unsafe. This means that a wrapper of Zephyr's API must be very carefully written in general.

Another challenge that must be solved in order for Rust to be usable with Zephyr is the configurability of Zephyr. Zephyr's configurations changes what bindings must (and can) be made and what they look like. Making bindings is not very hard if we know all the necessary header files, and the project's configuration has been converted into header files containing C macros has been generated by DTC and Kconfig. The problem is that we do not know the configuration of the project until an application developer starts the project. It is infeasible to create a set of bindings for every combination of configurations and boards. While we can use a build script to generate the configuration information we need to generate the bindings, this still means that the developer will not be able to know the full type of the bindings ahead of time.

Another thing to consider is how this interacts with a higher level safe Rust wrapper of Zephyr's API. On the one hand, this might mean that the application developer does not need to know what the bindings look like, but care must be taken to avoid that the application developer uses functions that are not supported by the current configuration. Rust does have mechanisms for conditional compilation, but this would mean that rustc somehow must be aware of the configuration of the project.

A final interesting question is whether it is possible to implement Rust's standard library on top of Zephyr. Currently, the Rust application is built without the Rust standard library. It only uses the system independent core library. A big reason for not supporting the standard library is that we currently do not support heap allocation. However, Zephyr does support memory allocation, and Rust does have a mechanism of specifying which allocator to use. This means that it should be possible to tell Rust to use Zephyr's memory allocator to allocate memory. This would give us access to a larger part of the standard library, including most of the collection data types. Rust is also relying on the operating system for threads and networking. It might also be possible to implement these parts on top of Zephyr. This is helped by Zephyr supporting a POSIX interface for some of these things.

Chapter 6

Conclusion

In this report, we have shown that it is possible to make a simple Zephyr application in Rust. We have also found several challenges that must be overcome in order to make this viable for larger projects. The most immediate challenge is to generate bindings. First, we need to know what header files to create bindings from. Next, we would have to know the configuration. Finally, we need to generate bindings to `static inline` functions.

When the bindings are created, we need a good way to build our Rust application and link it to Zephyr. These two problems are also linked, as the configuration we need to generate bindings are created when Zephyr is built. When the application is built, we need to link it to Zephyr. Here the biggest challenge is that Zephyr and Rust use two different toolchains that both define some common symbols.

When the application is built and linked, we still need to ensure that the Rust application actually is compatible with Zephyr and C. The most interesting question here is how interrupts affect the stack, and how Rust uses it.

When all these problems are solved, we will have a way of writing Rust applications for Zephyr, but this would have to be maintained, documentation must be written, and examples of how to use the Rust bindings and wrapper must be made. This means that there is still much work to be done before writing Rust applications in Zephyr becomes a viable option.

Most of these problems should not be too hard to solve, although some thought must be given on how the user experience can become as pleasant as possible. The biggest challenge is to create bindings to `static inline` functions, as the very definitions of `static` and `inline` means that this should be impossible. There are, however, some ways around the problem that are being worked on. This problem is not limited to Zephyr, and a solution would help create bindings to other libraries as well. A stopgap solution to the problem would be to create a fork of Zephyr that adds wrapper functions around the `static inline` functions. This way, we would have functions to bind to, without requiring significant changes in Zephyr or any Rust tools.

In order for the Rust bindings to really be useful, we also need to wrap them in a

safe Rust layer. This layer might be fairly thin, or it could present a much more idiomatic Rust interface. Another and more exciting possibility is that this layer might be the Rust standard library. There is much work that would have to happen for this to be possible, as the standard library holds quite strict stability and safety guarantees, but it is definitely worth exploring the possibility more in-depth if the other problems can be solved.

Despite a large number of open questions and much work that must be done to make it work, writing a Rust application in Zephyr should be possible. Most of the open problems should be solvable with enough work on the tools needed. Also, most of the problems can be solved without making changes to either Zephyr or Rust. This means that there are a lot fewer stakeholders that must be involved, and we need support form in order to be able to write Zephyr applications in Rust.

Bibliography

[1] Bindgen contribution guide. https://github.com/rust-lang/rust-bindgen/blob/master/CONTRIBUTING.md#code-overview. Accessed 2019-05-16.

[2] Generate C code to export static inline functions. https://github.com/rust-lang/rust-bindgen/issues/1090. Accessed 2019-06-06.

[3] libgcc documentation. gcc.gnu.org/onlinedocs/gccint/Libgcc.html. Accessed 2019-06-04.

[4] LLVM project home page. https://www.llvm.org. Accessed 2019-05-13.

[5] PSA: Xargo is now in maintenance mode. https://github.com/japaric/xargo/issues/193. Accessed 2019-05-14.

[6] *The Rustonomicon*. Accessed 2019-05-31.

[7] Survey: 2019 wishlist. https://github.com/rust-embedded/wg/issues/256. Accessed 2019-06-06.

[8] Target specification definition. https://github.com/rust-lang/rust/blob/master/src/librustc_target/spec/mod.rs. Accessed 2019-05-30.

[9] The Cargo Book. https://doc.rust-lang.org/cargo/index.html. Accessed 2019-05-16.

[10] The Rust Reference. https://doc.rust-lang.org/reference/introduction.html. Accessed 2019-06-03.

[11] What is the Zephyr Project? https://www.zephyrproject.org/what-is-zephyr/. Accessed 2018-12-01.

[12] Xargo. https://github.com/japaric/xargo. Accessed 2019-05-14.

[13] Zephyr project home page. https://www.zephyrproject.org. Accessed 2018-12-01.

[14] Flexible target specification. RFC 131, June 2014.

[15] The history of rust. In *Applicative 2016*, Applicative 2016, pages –, New York, NY, USA, 2016. ACM. Speaker-Klabnik, Steve.

[16] The Linux Foundation Announces Project to Build Real-Time Operating System for Internet of Things Devices. https://www.zephyrproject.org/linux-foundation-announces-porject-

build-real-time-operating-system-internet-things-devices/, 2016. Accessed 2019-05-20.

[17] PSA: Cortex-M Breakage (LLD as the default linker). PSA, August 2018.

[18] Zephyr Kernel Primer, Kernel Objects. https://docs.zephyrproject.org/latest/kernel/usermode/kernelobjects.html, 2018. Accessed 2018-12-06.

[19] Zephyr Kernel Primer, User Mode. https://docs.zephyrproject.org/latest/kernel/usermode/usermode.html, 2018. Accessed 2018-12-04.

[20] ARM. *Cortex-M4 Devices, Generic User Guide*, 2010.

[21] Alexis Beingessner. Here's my type, so initialize me maybe (mem::uninitialized is deprecated). https://gankro.github.io/blah/initialize-me-maybe/, May 2019. Accessed 2019-06-05.

[22] Nick Cameron. Cargo's next few years. https://www.ncameron.org/blog/cargos-next-few-years/, February 2019. Accessed 2019-06-06.

[23] Andy Gross. Device tree in Zephyr project. Presented at Embedded Linux Conference, 2017.

[24] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2. edition, 2018.

[25] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 1999.

[26] Didrik Rokhaug. An alternative approach to zephyr's system call dispatch, using rust. Project thesis, Norwegian University of Science and Technology, 2018.

[27] TIS Comittee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, 1.2 edition, May 1995.

[28] Jim Turley. Wind River Sets Rocket RTOS On Free Trajectory. https://www.eejournal.com/article/20151125-windriver, 2015. Accessed 2018-12-03.

Appendix A

Build log for Zephyr's "Blinky" sample

Here are parts of the logfile created by building Zephyr's "Blinky" sample for the nRF52 development kit with the verbose option. The paths have been shortened, and parts of the file have been removed due to the similarity with the excerpts listed. A summary of the omitted parts are given in their place. The full file (with shortened paths) can be found in the accompanying zip file.

The commands used to produce the logfile were:

- `$ cd /zephyr/samples/basic/blinky`
- `$ mkdir build/nrf52_pca10040 && cd build/nrf52_pca10040`
- `$ cmake -DBOARD=nrf52_pca10040 ../..`
- `$ make VERBOSE=1 > make.log`

A.1 The generation step

```
/usr/bin/cmake -S/zephyr/samples/basic/blinky -B/zephyr/
    samples/basic/blinky/build/nrf52_pca10040 --check-
    build-system CMakeFiles/Makefile.cmake 0
/usr/bin/cmake -E cmake_progress_start /zephyr/samples/
    basic/blinky/build/nrf52_pca10040/CMakeFiles /zephyr/
    samples/basic/blinky/build/nrf52_pca10040/CMakeFiles/
    progress.marks
make -f CMakeFiles/Makefile2 all
make[1]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
make -f zephyr/CMakeFiles/kobj_types_h_target.dir/build.
    make zephyr/CMakeFiles/kobj_types_h_target.dir/depend
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
```

```
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040 && /
    usr/bin/cmake -E cmake_depends "Unix Makefiles" /
    zephyr/samples/basic/blinky /zephyr /zephyr/samples/
    basic/blinky/build/nrf52_pca10040 /zephyr/samples/
    basic/blinky/build/nrf52_pca10040/zephyr /zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    CMakeFiles/kobj_types_h_target.dir/DependInfo.cmake --
    color=
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/kobj_types_h_target.
    dir/DependInfo.cmake" is newer than depender "/zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    CMakeFiles/kobj_types_h_target.dir/depend.internal".
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    CMakeDirectoryInformation.cmake" is newer than
    depender "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/kobj_types_h_target.
    dir/depend.internal".
Scanning dependencies of target kobj_types_h_target
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
make -f zephyr/CMakeFiles/kobj_types_h_target.dir/build.
    make zephyr/CMakeFiles/kobj_types_h_target.dir/build
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
[  1%] Generating include/generated/kobj-types-enum.h,
    include/generated/otype-to-str.h
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /usr/bin/python /zephyr/scripts/
    gen_kobject_list.py --kobj-types-output /zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    include/generated/kobj-types-enum.h --kobj-otype-
    output /zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/include/generated/otype-to-str.h
     --kobj-size-output /zephyr/samples/basic/blinky/build
    /nrf52_pca10040/zephyr/include/generated/otype-to-size
    .h
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[  1%] Built target kobj_types_h_target
make -f zephyr/CMakeFiles/syscall_macros_h_target.dir/
    build.make zephyr/CMakeFiles/syscall_macros_h_target.
    dir/depend
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040 && /
    usr/bin/cmake -E cmake_depends "Unix Makefiles" /
    zephyr/samples/basic/blinky /zephyr /zephyr/samples/
    basic/blinky/build/nrf52_pca10040 /zephyr/samples/
```

```
    basic/blinky/build/nrf52_pca10040/zephyr /zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    CMakeFiles/syscall_macros_h_target.dir/DependInfo.
    cmake --color=
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    syscall_macros_h_target.dir/DependInfo.cmake" is newer
     than depender "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    syscall_macros_h_target.dir/depend.internal".
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    CMakeDirectoryInformation.cmake" is newer than
    depender "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    syscall_macros_h_target.dir/depend.internal".
Scanning dependencies of target syscall_macros_h_target
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
make -f zephyr/CMakeFiles/syscall_macros_h_target.dir/
    build.make zephyr/CMakeFiles/syscall_macros_h_target.
    dir/build
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
[  2%] Generating include/generated/syscall_macros.h
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /usr/bin/python /zephyr/scripts/
    gen_syscall_header.py > /zephyr/samples/basic/blinky/
    build/nrf52_pca10040/zephyr/include/generated/
    syscall_macros.h
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[  2%] Built target syscall_macros_h_target
make -f zephyr/CMakeFiles/syscall_list_h_target.dir/build
    .make zephyr/CMakeFiles/syscall_list_h_target.dir/
    depend
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040 && /
    usr/bin/cmake -E cmake_depends "Unix Makefiles" /
    zephyr/samples/basic/blinky /zephyr /zephyr/samples/
    basic/blinky/build/nrf52_pca10040 /zephyr/samples/
    basic/blinky/build/nrf52_pca10040/zephyr /zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    CMakeFiles/syscall_list_h_target.dir/DependInfo.cmake
    --color=
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/syscall_list_h_target
    .dir/DependInfo.cmake" is newer than depender "/zephyr
    /samples/basic/blinky/build/nrf52_pca10040/zephyr/
```

```
    CMakeFiles/syscall_list_h_target.dir/depend.internal".
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    CMakeDirectoryInformation.cmake" is newer than
    depender "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/syscall_list_h_target
    .dir/depend.internal".
Scanning dependencies of target syscall_list_h_target
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
make -f zephyr/CMakeFiles/syscall_list_h_target.dir/build
    .make zephyr/CMakeFiles/syscall_list_h_target.dir/
    build
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
[  3%] Generating misc/generated/syscalls.json
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /usr/bin/python /zephyr/scripts/
    parse_syscalls.py --include /zephyr/include --json-
    file /zephyr/samples/basic/blinky/build/nrf52_pca10040
    /zephyr/misc/generated/syscalls.json
[  4%] Generating include/generated/syscall_dispatch.c,
    include/generated/syscall_list.h
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /usr/bin/python /zephyr/scripts/gen_syscalls
    .py --json-file /zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/misc/generated/syscalls.json --
    base-output include/generated/syscalls --syscall-
    dispatch include/generated/syscall_dispatch.c --
    syscall-list /zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/include/generated/syscall_list.h
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[  5%] Built target syscall_list_h_target
make -f zephyr/CMakeFiles/driver_validation_h_target.dir/
    build.make zephyr/CMakeFiles/
    driver_validation_h_target.dir/depend
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040 && /
    usr/bin/cmake -E cmake_depends "Unix Makefiles" /
    zephyr/samples/basic/blinky /zephyr /zephyr/samples/
    basic/blinky/build/nrf52_pca10040 /zephyr/samples/
    basic/blinky/build/nrf52_pca10040/zephyr /zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    CMakeFiles/driver_validation_h_target.dir/DependInfo.
    cmake --color=
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    driver_validation_h_target.dir/DependInfo.cmake" is
```

```
    newer than depender "/zephyr/samples/basic/blinky/
    build/nrf52_pca10040/zephyr/CMakeFiles/
    driver_validation_h_target.dir/depend.internal".
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    CMakeDirectoryInformation.cmake" is newer than
    depender "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    driver_validation_h_target.dir/depend.internal".
Scanning dependencies of target
    driver_validation_h_target
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
make -f zephyr/CMakeFiles/driver_validation_h_target.dir/
    build.make zephyr/CMakeFiles/
    driver_validation_h_target.dir/build
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
[  5%] Generating include/generated/driver-validation.h
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /usr/bin/python /zephyr/scripts/
    gen_kobject_list.py --validation-output /zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    include/generated/driver-validation.h
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[  5%] Built target driver_validation_h_target
make -f zephyr/CMakeFiles/offsets.dir/build.make zephyr/
    CMakeFiles/offsets.dir/depend
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040 && /
    usr/bin/cmake -E cmake_depends "Unix Makefiles" /
    zephyr/samples/basic/blinky /zephyr /zephyr/samples/
    basic/blinky/build/nrf52_pca10040 /zephyr/samples/
    basic/blinky/build/nrf52_pca10040/zephyr /zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    CMakeFiles/offsets.dir/DependInfo.cmake --color=
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/offsets.dir/
    DependInfo.cmake" is newer than depender "/zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    CMakeFiles/offsets.dir/depend.internal".
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    CMakeDirectoryInformation.cmake" is newer than
    depender "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/offsets.dir/depend.
    internal".
Scanning dependencies of target offsets
```

```
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
make -f zephyr/CMakeFiles/offsets.dir/build.make zephyr/
    CMakeFiles/offsets.dir/build
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
[  5%] Building C object zephyr/CMakeFiles/offsets.dir/
    arch/arm/core/offsets/offsets.c.obj
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && ccache /opt/zephyr-sdk/sysroots/x86_64-
    pokysdk-linux/usr/bin/arm-zephyr-eabi/arm-zephyr-eabi-
    gcc -DBUILD_VERSION=zephyr-v1.13.0-3719-g01592071f1 -
    DKERNEL -DNRF52832_XXAA -D_FORTIFY_SOURCE=2 -
    D__ZEPHYR__=1 -I/zephyr/kernel/include -I/zephyr/arch/
    arm/include -I/zephyr/soc/arm/nordic_nrf/nrf52 -I/
    zephyr/soc/arm/nordic_nrf/nrf52/include -I/zephyr/soc/
    arm/nordic_nrf/include -I/zephyr/include -I/zephyr/
    include/drivers -I/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/include/generated -I/zephyr/lib/
    libc/minimal/include -I/zephyr/ext/hal/cmsis/Include -
    I/zephyr/ext/hal/nordic/nrfx -I/zephyr/ext/hal/nordic/
    nrfx/drivers/include -I/zephyr/ext/hal/nordic/nrfx/hal
     -I/zephyr/ext/hal/nordic/nrfx/mdk -I/zephyr/ext/hal/
    nordic/. -I/zephyr/subsys/bluetooth -isystem /opt/
    zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/lib/arm-
    zephyr-eabi/gcc/arm-zephyr-eabi/6.2.0/include -isystem
     /opt/zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/lib
    /arm-zephyr-eabi/gcc/arm-zephyr-eabi/6.2.0/include-
    fixed  -Os -g -Wall -Wformat -Wformat-security -Wno-
    format-zero-length -imacros /zephyr/samples/basic/
    blinky/build/nrf52_pca10040/zephyr/include/generated/
    autoconf.h -ffreestanding -Wno-main -fno-common --
    sysroot /opt/zephyr-sdk/sysroots/armv5-zephyr-eabi/usr
     -mthumb -mcpu=cortex-m4 -fno-asynchronous-unwind-
    tables -fno-pie -fno-pic -fno-strict-overflow -Wno-
    pointer-sign -Wno-unused-but-set-variable -fno-reorder
    -functions -fno-defer-pop -Werror=implicit-int -
    Wpointer-arith -ffunction-sections -fdata-sections -
    mabi=aapcs -march=armv7e-m -std=c99 -o CMakeFiles/
    offsets.dir/arch/arm/core/offsets/offsets.c.obj   -c /
    zephyr/arch/arm/core/offsets/offsets.c
[  6%] Linking C static library liboffsets.a
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /usr/bin/cmake -P CMakeFiles/offsets.dir/
    cmake_clean_target.cmake
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /usr/bin/cmake -E cmake_link_script
    CMakeFiles/offsets.dir/link.txt --verbose=1
ccache /opt/zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/
    bin/arm-zephyr-eabi/arm-zephyr-eabi-ar qc liboffsets.a
```

```
      CMakeFiles/offsets.dir/arch/arm/core/offsets/offsets
    .c.obj
ccache /opt/zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/
    bin/arm-zephyr-eabi/arm-zephyr-eabi-ranlib liboffsets.
    a
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[  6%] Built target offsets
make -f zephyr/CMakeFiles/offsets_h.dir/build.make zephyr
    /CMakeFiles/offsets_h.dir/depend
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040 && /
    usr/bin/cmake -E cmake_depends "Unix Makefiles" /
    zephyr/samples/basic/blinky /zephyr /zephyr/samples/
    basic/blinky/build/nrf52_pca10040 /zephyr/samples/
    basic/blinky/build/nrf52_pca10040/zephyr /zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    CMakeFiles/offsets_h.dir/DependInfo.cmake --color=
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/offsets_h.dir/
    DependInfo.cmake" is newer than depender "/zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    CMakeFiles/offsets_h.dir/depend.internal".
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    CMakeDirectoryInformation.cmake" is newer than
    depender "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/offsets_h.dir/depend.
    internal".
Scanning dependencies of target offsets_h
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
make -f zephyr/CMakeFiles/offsets_h.dir/build.make zephyr
    /CMakeFiles/offsets_h.dir/build
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
[  7%] Generating include/generated/offsets.h
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /usr/bin/python /zephyr/scripts/
    gen_offset_header.py -i /zephyr/samples/basic/blinky/
    build/nrf52_pca10040/zephyr/CMakeFiles/offsets.dir/
    arch/arm/core/offsets/offsets.c.obj -o /zephyr/samples
    /basic/blinky/build/nrf52_pca10040/zephyr/include/
    generated/offsets.h
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[  7%] Built target offsets_h
```

A.2 The build step

A.2.1 The building of libapp.a

```
make -f CMakeFiles/app.dir/build.make CMakeFiles/app.dir/
    depend
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040 && /
    usr/bin/cmake -E cmake_depends "Unix Makefiles" /
    zephyr/samples/basic/blinky /zephyr/samples/basic/
    blinky /zephyr/samples/basic/blinky/build/
    nrf52_pca10040 /zephyr/samples/basic/blinky/build/
    nrf52_pca10040 /zephyr/samples/basic/blinky/build/
    nrf52_pca10040/CMakeFiles/app.dir/DependInfo.cmake --
    color=
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/CMakeFiles/app.dir/DependInfo.cmake" is
     newer than depender "/zephyr/samples/basic/blinky/
    build/nrf52_pca10040/CMakeFiles/app.dir/depend.
    internal".
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/CMakeFiles/CMakeDirectoryInformation.
    cmake" is newer than depender "/zephyr/samples/basic/
    blinky/build/nrf52_pca10040/CMakeFiles/app.dir/depend.
    internal".
Scanning dependencies of target app
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
make -f CMakeFiles/app.dir/build.make CMakeFiles/app.dir/
    build
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
[  7%] Building C object CMakeFiles/app.dir/src/main.c.
    obj
ccache /opt/zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/
    bin/arm-zephyr-eabi/arm-zephyr-eabi-gcc -
    DBUILD_VERSION=zephyr-v1.13.0-3719-g01592071f1 -
    DKERNEL -DNRF52832_XXAA -D_FORTIFY_SOURCE=2 -
    D__ZEPHYR__=1 -I/zephyr/kernel/include -I/zephyr/arch/
    arm/include -I/zephyr/soc/arm/nordic_nrf/nrf52 -I/
    zephyr/soc/arm/nordic_nrf/nrf52/include -I/zephyr/soc/
    arm/nordic_nrf/include -I/zephyr/include -I/zephyr/
    include/drivers -I/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/include/generated -I/zephyr/lib/
    libc/minimal/include -I/zephyr/ext/hal/cmsis/Include -
    I/zephyr/ext/hal/nordic/nrfx -I/zephyr/ext/hal/nordic/
    nrfx/drivers/include -I/zephyr/ext/hal/nordic/nrfx/hal
     -I/zephyr/ext/hal/nordic/nrfx/mdk -I/zephyr/ext/hal/
```

```
    nordic/. -I/zephyr/subsys/bluetooth -isystem /opt/
    zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/lib/arm-
    zephyr-eabi/gcc/arm-zephyr-eabi/6.2.0/include -isystem
     /opt/zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/lib
    /arm-zephyr-eabi/gcc/arm-zephyr-eabi/6.2.0/include-
    fixed  -Os -g -Wall -Wformat -Wformat-security -Wno-
    format-zero-length -imacros /zephyr/samples/basic/
    blinky/build/nrf52_pca10040/zephyr/include/generated/
    autoconf.h -ffreestanding -Wno-main -fno-common --
    sysroot /opt/zephyr-sdk/sysroots/armv5-zephyr-eabi/usr
     -mthumb -mcpu=cortex-m4 -fno-asynchronous-unwind-
    tables -fno-pie -fno-pic -fno-strict-overflow -Wno-
    pointer-sign -Wno-unused-but-set-variable -fno-reorder
    -functions -fno-defer-pop -Werror=implicit-int -
    Wpointer-arith -ffunction-sections -fdata-sections -
    mabi=aapcs -march=armv7e-m -std=c99 -o CMakeFiles/app.
    dir/src/main.c.obj   -c /zephyr/samples/basic/blinky/
    src/main.c
[  8%] Linking C static library app/libapp.a
/usr/bin/cmake -P CMakeFiles/app.dir/cmake_clean_target.
    cmake
/usr/bin/cmake -E cmake_link_script CMakeFiles/app.dir/
    link.txt --verbose=1
ccache /opt/zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/
    bin/arm-zephyr-eabi/arm-zephyr-eabi-ar qc app/libapp.a
      CMakeFiles/app.dir/src/main.c.obj
ccache /opt/zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/
    bin/arm-zephyr-eabi/arm-zephyr-eabi-ranlib app/libapp.
    a
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[  8%] Built target app
```

A.2.2 The rest of the build step

In the build step there are a lot of other archives being generated, all containging a varying number of object files. As all the object files and archives are generated in a similar fashion to main.c.obj and libapp, we include a simple list of archives and their object files. Each object file is based on a source file with the same name (without the .obj suffix). When a folder is given in parenthesis after the name of the archive, all the source files used can be found in that folder. Otherwise the file used to create the object file is found in the folder given in parenthesis after the name of the object file. Any common prefix to the path is written after the name of the archive. The full log file can be found in the accompanying zip file.

- libkernel.a (/zephyr/kernel/)
  - device.c.obj
  - errno.c.obj

  - idle.c.obj
  - init.c.obj
  - mailbox.c.obj
  - mem_slab.c.obj
  - mempool.c.obj
  - msg_q.c.obj
  - mutex.c.obj
  - pipes.c.obj
  - queue.c.obj
  - sched.c.obj
  - sem.c.obj
  - stack.c.obj
  - system_work_q.c.obj
  - thread.c.obj
  - thread_abort.c.obj
  - version.c.obj
  - work_q.c.obj
  - smp.c.obj
  - timeout.c.obj
  - timer.c.obj
  - poll.c.obj
- libzephyr.a (/zephyr/)
  - isr_tables.c.obj (arch/common/)
  - sw_isr_common.c.obj (arch/common/)
  - crc32_sw.c.obj (lib/os/)
  - crc16_sw.c.obj (lib/os/)
  - crc8_sw.c.obj (lib/os/)
  - crc7_sw.c.obj (lib/os/)
  - fdtable.c.obj (lib/os/)
  - mempool.c.obj (lib/os/)
  - rb.c.obj (lib/os/)
  - thread_entry.c.obj (lib/os/)
  - work_q.c.obj (lib/os/)

  - printk.c.ob (lib/os/)
  - configs.c.obj (samples/basic/blinky/build/nrf52_pca10040/zephyr/misc/generated/)
  - power.c.obj (soc/arm/nordic_nrf/nrf52/)
  - soc.c.obj (soc/arm/nordic_nrf/nrf52/)
  - mpu_regions.c.obj (soc/arm/nordic_nrf/nrf52/)
  - system_nrf52.c.obj (ext/hal/nordic/nrfx/mdk/)
  - nrfx_glue.c.obj (ext/hal/nordic/)
  - rand32_entropy_device.c.obj (subsys/random/)
  - nrf_power_clock.c.obj (drivers/clock_control/)
  - sys_clock_init.c.obj (drivers/timer/)
  - nrf_rtc_timer.c.obj (drivers/timer/)
- libarch__arm__core.a (/zephyr/arch/arm/core/)
  - exc_exit.S.obj
  - irq_init.c.obj
  - swap.c.obj
  - swap_helper.S.obj
  - fault.c.obj
  - irq_manage.c.obj
  - thread.c.obj
  - cpu_idle.S.obj
  - fault_s.S.obj
  - fatal.c.obj
  - sys_fatal_error_handler.c.obj
  - thread_abort.c.obj
  - isr_wrapper.S.obj
- libarch__arm__core__cortex_m.a (/zephyr/arch/arm/core/cortex_m/)
  - vector_table.S.obj
  - reset.S.obj
  - nmi_on_reset.S.obj
  - prep_c.c.obj
  - scb.c.obj
  - nmi.c.obj
  - exc_manage.c.obj

- libarch__arm__core__cortex_m__mpu.a (/zephyr/arch/arm/core/cortex_m/mpu/)
  - arm_core_mpu.c.obj
  - arm_mpu.c.obj
- liblib__libc__minimal.a (/zephyr/lib/libc/minimal/source/)
  - atoi.c.obj (stdlib/)
  - strtol.c.obj (stdlib/)
  - strtoul.c.obj (stdlib/)
  - malloc.c.obj (stdlib/)
  - strncasecmp.c.obj (string/)
  - strstr.c.obj (string/)
  - string.c.obj (string/)
  - prf.c.obj (stdout/)
  - stdout_console.c.obj (stdout/)
  - sprintf.c.obj (stdout/)
  - fprintf.c.obj (stdout/)
- libsubsys__bluetooth__common.a (/zephyr/subsys/bluetooth/common/)
  - dummy.c.obj
  - log.c.obj
- libsubsys__bluetooth__host.a (/zephyr/subsys/bluetooth/host)
  - uuid.c.obj
  - hci_core.c.obj
- libsubsys__bluetooth__controller.a (/zephyr/subsys/bluetooth/controller/)
  - mem.c.obj (util/)
  - memq.c.obj (util/)
  - mayfly.c.obj (util/)
  - util.c.obj (util/)
  - ticker.c.obj (ticker/)
  - ll_addr.c.obj (ll_sw/)
  - ll_tx_pwr.c.obj (ll_sw/)
  - hci_driver.c.obj (hci/)
  - hci.c.obj (hci/)
  - crypto.c.obj (crypto/)
  - ctrl.c.obj (ll_sw)

  - ll.c.obj (ll_sw)
  - ll_adc.c.obj (ll_sw)
  - ll_filter.c.obj (ll_sw)
  - ll_adv_aux.c.obj (ll_sw)
  - cntr.c.obj (ll_sw/nordic/hal/nrf5/)
  - ecb.c.obj (ll_sw/nordic/hal/nrf5/)
  - radio.c.obj (ll_sw/nordic/hal/nrf5/radio)
  - mayfly.c.obj (ll_sw/nordic/hal/nrf5/)
  - ticker.c.obj (ll_sw/nordic/hal/nrf5/)
- libsubsys__net.a (/zephyr/subsys/net/)
  - buf.c.obj
- libdrivers__gpio.a (/zephyr/drivers/gpio/)
  - gpio_nrfx.c.obj
- libdrivers__entropy.a (/zephyr/drivers/entropy/)
  - entropy_nrf5.c.obj

A.3 The link step

```
make -f zephyr/CMakeFiles/zephyr_prebuilt.dir/build.make
    zephyr/CMakeFiles/zephyr_prebuilt.dir/depend
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040 && /
    usr/bin/cmake -E cmake_depends "Unix Makefiles" /
    zephyr/samples/basic/blinky /zephyr /zephyr/samples/
    basic/blinky/build/nrf52_pca10040 /zephyr/samples/
    basic/blinky/build/nrf52_pca10040/zephyr /zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    CMakeFiles/zephyr_prebuilt.dir/DependInfo.cmake --
    color=
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/zephyr_prebuilt.dir/
    DependInfo.cmake" is newer than depender "/zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    CMakeFiles/zephyr_prebuilt.dir/depend.internal".
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    CMakeDirectoryInformation.cmake" is newer than
    depender "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/zephyr_prebuilt.dir/
    depend.internal".
```

```
Scanning dependencies of target zephyr_prebuilt
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
make -f zephyr/CMakeFiles/zephyr_prebuilt.dir/build.make
    zephyr/CMakeFiles/zephyr_prebuilt.dir/build
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
[ 94%] Building C object zephyr/CMakeFiles/
    zephyr_prebuilt.dir/misc/empty_file.c.obj
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && ccache /opt/zephyr-sdk/sysroots/x86_64-
    pokysdk-linux/usr/bin/arm-zephyr-eabi/arm-zephyr-eabi-
    gcc -DBUILD_VERSION=zephyr-v1.13.0-3719-g01592071f1 -
    DKERNEL -DNRF52832_XXAA -D_FORTIFY_SOURCE=2 -
    D__ZEPHYR__=1 -I/zephyr/kernel/include -I/zephyr/arch/
    arm/include -I/zephyr/soc/arm/nordic_nrf/nrf52 -I/
    zephyr/soc/arm/nordic_nrf/nrf52/include -I/zephyr/soc/
    arm/nordic_nrf/include -I/zephyr/include -I/zephyr/
    include/drivers -I/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/include/generated -I/zephyr/lib/
    libc/minimal/include -I/zephyr/ext/hal/cmsis/Include -
    I/zephyr/ext/hal/nordic/nrfx -I/zephyr/ext/hal/nordic/
    nrfx/drivers/include -I/zephyr/ext/hal/nordic/nrfx/hal
     -I/zephyr/ext/hal/nordic/nrfx/mdk -I/zephyr/ext/hal/
    nordic/. -I/zephyr/subsys/bluetooth -isystem /opt/
    zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/lib/arm-
    zephyr-eabi/gcc/arm-zephyr-eabi/6.2.0/include -isystem
     /opt/zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/lib
    /arm-zephyr-eabi/gcc/arm-zephyr-eabi/6.2.0/include-
    fixed -Os -g -Wall -Wformat -Wformat-security -Wno-
    format-zero-length -imacros /zephyr/samples/basic/
    blinky/build/nrf52_pca10040/zephyr/include/generated/
    autoconf.h -ffreestanding -Wno-main -fno-common --
    sysroot /opt/zephyr-sdk/sysroots/armv5-zephyr-eabi/usr
     -mthumb -mcpu=cortex-m4 -fno-asynchronous-unwind-
    tables -fno-pie -fno-pic -fno-strict-overflow -Wno-
    pointer-sign -Wno-unused-but-set-variable -fno-reorder
    -functions -fno-defer-pop -Werror=implicit-int -
    Wpointer-arith -ffunction-sections -fdata-sections -
    mabi=aapcs -march=armv7e-m -std=c99 -o CMakeFiles/
    zephyr_prebuilt.dir/misc/empty_file.c.obj -c /zephyr
    /misc/empty_file.c
[ 95%] Linking C executable zephyr_prebuilt.elf
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /usr/bin/cmake -E cmake_link_script
    CMakeFiles/zephyr_prebuilt.dir/link.txt --verbose=1
ccache /opt/zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/
    bin/arm-zephyr-eabi/arm-zephyr-eabi-gcc CMakeFiles/
    zephyr_prebuilt.dir/misc/empty_file.c.obj -o
    zephyr_prebuilt.elf -T linker.cmd -Wl,-Map=/zephyr/
```

```
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    zephyr.map -u_OffsetAbsSyms -u_ConfigAbsSyms -Wl,--
    whole-archive ../app/libapp.a libzephyr.a arch/arm/
    core/libarch__arm__core.a arch/arm/core/cortex_m/
    libarch__arm__core__cortex_m.a arch/arm/core/cortex_m/
    mpu/libarch__arm__core__cortex_m__mpu.a lib/libc/
    minimal/liblib__libc__minimal.a subsys/bluetooth/
    common/libsubsys__bluetooth__common.a subsys/bluetooth
    /host/libsubsys__bluetooth__host.a subsys/bluetooth/
    controller/libsubsys__bluetooth__controller.a subsys/
    net/libsubsys__net.a drivers/gpio/libdrivers__gpio.a
    drivers/entropy/libdrivers__entropy.a -Wl,--no-whole-
    archive kernel/libkernel.a CMakeFiles/offsets.dir/arch
    /arm/core/offsets/offsets.c.obj -L"/opt/zephyr-sdk/
    sysroots/armv5-zephyr-eabi/usr/lib/arm-zephyr-eabi
    /6.2.0/armv7e-m" -L/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr -lgcc -Wl,--print-memory-usage -
    mthumb -nostdlib -static -no-pie -Wl,-X -Wl,-N -Wl,--
    gc-sections -Wl,--build-id=none -Wl,--orphan-handling=
    warn -mabi=aapcs
Memory region         Used Size  Region Size  %age Used
           FLASH:       36408 B       512 KB      6.94%
            SRAM:       10868 B        64 KB     16.58%
        IDT_LIST:         120 B         2 KB      5.86%
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[ 95%] Built target zephyr_prebuilt
make -f zephyr/CMakeFiles/linker_pass_final_script_target
    .dir/build.make zephyr/CMakeFiles/
    linker_pass_final_script_target.dir/depend
make[2]: Entering directory '/zephyr/samples/basic/blinky
    /build/nrf52_pca10040'
cd /zephyr/samples/basic/blinky/build/nrf52_pca10040 && /
    usr/bin/cmake -E cmake_depends "Unix Makefiles" /
    zephyr/samples/basic/blinky /zephyr /zephyr/samples/
    basic/blinky/build/nrf52_pca10040 /zephyr/samples/
    basic/blinky/build/nrf52_pca10040/zephyr /zephyr/
    samples/basic/blinky/build/nrf52_pca10040/zephyr/
    CMakeFiles/linker_pass_final_script_target.dir/
    DependInfo.cmake --color=
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    linker_pass_final_script_target.dir/DependInfo.cmake"
    is newer than depender "/zephyr/samples/basic/blinky/
    build/nrf52_pca10040/zephyr/CMakeFiles/
    linker_pass_final_script_target.dir/depend.internal".
Dependee "/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    CMakeDirectoryInformation.cmake" is newer than
    depender "/zephyr/samples/basic/blinky/build/
```

```
    nrf52_pca10040/zephyr/CMakeFiles/
    linker_pass_final_script_target.dir/depend.internal".
Scanning dependencies of target
    linker_pass_final_script_target
make[2]: Leaving directory '/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
make -f zephyr/CMakeFiles/linker_pass_final_script_target
    .dir/build.make zephyr/CMakeFiles/
    linker_pass_final_script_target.dir/build
make[2]: Entering directory '/home/didrik/Dropbox/skole/
    ntnu/V2019/master/zephyr/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[ 96%] Generating linker_pass_final.cmd
cd /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /opt/zephyr-sdk/sysroots/x86_64-pokysdk-
    linux/usr/bin/arm-zephyr-eabi/arm-zephyr-eabi-gcc -x
    assembler-with-cpp -nostdinc -undef -MD -MF
    linker_pass_final.cmd.dep -MT zephyr/linker_pass_final
    .cmd -I/home/didrik/Dropbox/skole/ntnu/V2019/master/
    zephyr/zephyr/kernel/include -I/home/didrik/Dropbox/
    skole/ntnu/V2019/master/zephyr/zephyr/arch/arm/include
     -I/home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr
    /zephyr/soc/arm/nordic_nrf/nrf52 -I/home/didrik/
    Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/soc/arm/
    nordic_nrf/nrf52/include -I/home/didrik/Dropbox/skole/
    ntnu/V2019/master/zephyr/zephyr/soc/arm/nordic_nrf/
    include -I/home/didrik/Dropbox/skole/ntnu/V2019/master
    /zephyr/zephyr/include -I/home/didrik/Dropbox/skole/
    ntnu/V2019/master/zephyr/zephyr/include/drivers -I/
    home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr/include/generated -I/opt/zephyr-sdk/sysroots/
    x86_64-pokysdk-linux/usr/lib/arm-zephyr-eabi/gcc/arm-
    zephyr-eabi/6.2.0/include -I/opt/zephyr-sdk/sysroots/
    x86_64-pokysdk-linux/usr/lib/arm-zephyr-eabi/gcc/arm-
    zephyr-eabi/6.2.0/include-fixed -I/home/didrik/Dropbox
    /skole/ntnu/V2019/master/zephyr/zephyr/lib/libc/
    minimal/include -I/home/didrik/Dropbox/skole/ntnu/
    V2019/master/zephyr/zephyr/soc/arm/nordic_nrf/include
    -I/home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/ext/hal/cmsis/Include -I/home/didrik/Dropbox/
    skole/ntnu/V2019/master/zephyr/zephyr/ext/hal/nordic/
    nrfx -I/home/didrik/Dropbox/skole/ntnu/V2019/master/
    zephyr/zephyr/ext/hal/nordic/nrfx/drivers/include -I/
    home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/ext/hal/nordic/nrfx/hal -I/home/didrik/Dropbox/
    skole/ntnu/V2019/master/zephyr/zephyr/ext/hal/nordic/
    nrfx/mdk -I/home/didrik/Dropbox/skole/ntnu/V2019/
    master/zephyr/zephyr/ext/hal/nordic/. -I/home/didrik/
```

```
    Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/subsys/
    bluetooth -D__GCC_LINKER_CMD__ -DLINKER_PASS2 -E /home
    /didrik/Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/
    soc/arm/nordic_nrf/nrf52/linker.ld -P -o
    linker_pass_final.cmd
make[2]: Leaving directory '/home/didrik/Dropbox/skole/
    ntnu/V2019/master/zephyr/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[ 96%] Built target linker_pass_final_script_target
make -f zephyr/CMakeFiles/kernel_elf.dir/build.make
    zephyr/CMakeFiles/kernel_elf.dir/depend
make[2]: Entering directory '/home/didrik/Dropbox/skole/
    ntnu/V2019/master/zephyr/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[ 97%] Generating isr_tables.c
cd /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /opt/zephyr-sdk/sysroots/x86_64-pokysdk-
    linux/usr/bin/arm-zephyr-eabi/arm-zephyr-eabi-objcopy
    -I elf32-littlearm -O binary --only-section=.intList /
    home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr/zephyr_prebuilt.elf isrList.bin
cd /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /usr/bin/python /home/didrik/Dropbox/skole/
    ntnu/V2019/master/zephyr/zephyr/arch/common/
    gen_isr_tables.py --output-source isr_tables.c --
    kernel /home/didrik/Dropbox/skole/ntnu/V2019/master/
    zephyr/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/zephyr_prebuilt.elf --intlist
    isrList.bin    --sw-isr-table --vector-table
cd /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040 && /
    usr/bin/cmake -E cmake_depends "Unix Makefiles" /home/
    didrik/Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/
    samples/basic/blinky /home/didrik/Dropbox/skole/ntnu/
    V2019/master/zephyr/zephyr /home/didrik/Dropbox/skole/
    ntnu/V2019/master/zephyr/zephyr/samples/basic/blinky/
    build/nrf52_pca10040 /home/didrik/Dropbox/skole/ntnu/
    V2019/master/zephyr/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr /home/didrik/Dropbox/skole/ntnu/
    V2019/master/zephyr/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/kernel_elf.dir/
    DependInfo.cmake --color=
Dependee "/home/didrik/Dropbox/skole/ntnu/V2019/master/
    zephyr/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/kernel_elf.dir/
    DependInfo.cmake" is newer than depender "/home/didrik
    /Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/samples
```

```
    /basic/blinky/build/nrf52_pca10040/zephyr/CMakeFiles/
    kernel_elf.dir/depend.internal".
Dependee "/home/didrik/Dropbox/skole/ntnu/V2019/master/
    zephyr/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/
    CMakeDirectoryInformation.cmake" is newer than
    depender "/home/didrik/Dropbox/skole/ntnu/V2019/master
    /zephyr/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/CMakeFiles/kernel_elf.dir/depend
    .internal".
Scanning dependencies of target kernel_elf
make[2]: Leaving directory '/home/didrik/Dropbox/skole/
    ntnu/V2019/master/zephyr/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
make -f zephyr/CMakeFiles/kernel_elf.dir/build.make
    zephyr/CMakeFiles/kernel_elf.dir/build
make[2]: Entering directory '/home/didrik/Dropbox/skole/
    ntnu/V2019/master/zephyr/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[ 97%] Building C object zephyr/CMakeFiles/kernel_elf.dir
    /misc/empty_file.c.obj
cd /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && ccache /opt/zephyr-sdk/sysroots/x86_64-
    pokysdk-linux/usr/bin/arm-zephyr-eabi/arm-zephyr-eabi-
    gcc -DBUILD_VERSION=zephyr-v1.13.0-3719-g01592071f1 -
    DKERNEL -DNRF52832_XXAA -D_FORTIFY_SOURCE=2 -
    D__ZEPHYR__=1 -I/home/didrik/Dropbox/skole/ntnu/V2019/
    master/zephyr/zephyr/kernel/include -I/home/didrik/
    Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/arch/arm
    /include -I/home/didrik/Dropbox/skole/ntnu/V2019/
    master/zephyr/zephyr/soc/arm/nordic_nrf/nrf52 -I/home/
    didrik/Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/
    soc/arm/nordic_nrf/nrf52/include -I/home/didrik/
    Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/soc/arm/
    nordic_nrf/include -I/home/didrik/Dropbox/skole/ntnu/
    V2019/master/zephyr/zephyr/include -I/home/didrik/
    Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/include/
    drivers -I/home/didrik/Dropbox/skole/ntnu/V2019/master
    /zephyr/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/include/generated -I/home/didrik
    /Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/lib/
    libc/minimal/include -I/home/didrik/Dropbox/skole/ntnu
    /V2019/master/zephyr/zephyr/ext/hal/cmsis/Include -I/
    home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/ext/hal/nordic/nrfx -I/home/didrik/Dropbox/
    skole/ntnu/V2019/master/zephyr/zephyr/ext/hal/nordic/
    nrfx/drivers/include -I/home/didrik/Dropbox/skole/ntnu
    /V2019/master/zephyr/zephyr/ext/hal/nordic/nrfx/hal -I
    /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
```

```
zephyr/ext/hal/nordic/nrfx/mdk -I/home/didrik/Dropbox/
skole/ntnu/V2019/master/zephyr/zephyr/ext/hal/nordic/.
-I/home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr
/zephyr/subsys/bluetooth -isystem /opt/zephyr-sdk/
sysroots/x86_64-pokysdk-linux/usr/lib/arm-zephyr-eabi/
gcc/arm-zephyr-eabi/6.2.0/include -isystem /opt/zephyr
-sdk/sysroots/x86_64-pokysdk-linux/usr/lib/arm-zephyr-
eabi/gcc/arm-zephyr-eabi/6.2.0/include-fixed -Os -g -
Wall -Wformat -Wformat-security -Wno-format-zero-
length -imacros /home/didrik/Dropbox/skole/ntnu/V2019/
master/zephyr/zephyr/samples/basic/blinky/build/
nrf52_pca10040/zephyr/include/generated/autoconf.h -
ffreestanding -Wno-main -fno-common --sysroot /opt/
zephyr-sdk/sysroots/armv5-zephyr-eabi/usr -mthumb -
mcpu=cortex-m4 -fno-asynchronous-unwind-tables -fno-
pie -fno-pic -fno-strict-overflow -Wno-pointer-sign -
Wno-unused-but-set-variable -fno-reorder-functions -
fno-defer-pop -Werror=implicit-int -Wpointer-arith -
ffunction-sections -fdata-sections -mabi=aapcs -march=
armv7e-m -std=c99 -o CMakeFiles/kernel_elf.dir/misc/
empty_file.c.obj -c /home/didrik/Dropbox/skole/ntnu/
V2019/master/zephyr/zephyr/misc/empty_file.c
[ 98%] Building C object zephyr/CMakeFiles/kernel_elf.dir
/isr_tables.c.obj
cd /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
zephyr/samples/basic/blinky/build/nrf52_pca10040/
zephyr && ccache /opt/zephyr-sdk/sysroots/x86_64-
pokysdk-linux/usr/bin/arm-zephyr-eabi/arm-zephyr-eabi-
gcc -DBUILD_VERSION=zephyr-v1.13.0-3719-g01592071f1 -
DKERNEL -DNRF52832_XXAA -D_FORTIFY_SOURCE=2 -
D__ZEPHYR__=1 -I/home/didrik/Dropbox/skole/ntnu/V2019/
master/zephyr/zephyr/kernel/include -I/home/didrik/
Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/arch/arm
/include -I/home/didrik/Dropbox/skole/ntnu/V2019/
master/zephyr/zephyr/soc/arm/nordic_nrf/nrf52 -I/home/
didrik/Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/
soc/arm/nordic_nrf/nrf52/include -I/home/didrik/
Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/soc/arm/
nordic_nrf/include -I/home/didrik/Dropbox/skole/ntnu/
V2019/master/zephyr/zephyr/include -I/home/didrik/
Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/include/
drivers -I/home/didrik/Dropbox/skole/ntnu/V2019/master
/zephyr/zephyr/samples/basic/blinky/build/
nrf52_pca10040/zephyr/include/generated -I/home/didrik
/Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/lib/
libc/minimal/include -I/home/didrik/Dropbox/skole/ntnu
/V2019/master/zephyr/zephyr/ext/hal/cmsis/Include -I/
home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
zephyr/ext/hal/nordic/nrfx -I/home/didrik/Dropbox/
skole/ntnu/V2019/master/zephyr/zephyr/ext/hal/nordic/
```

```
    nrfx/drivers/include -I/home/didrik/Dropbox/skole/ntnu
    /V2019/master/zephyr/zephyr/ext/hal/nordic/nrfx/hal -I
    /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/ext/hal/nordic/nrfx/mdk -I/home/didrik/Dropbox/
    skole/ntnu/V2019/master/zephyr/zephyr/ext/hal/nordic/.
     -I/home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr
    /zephyr/subsys/bluetooth -isystem /opt/zephyr-sdk/
    sysroots/x86_64-pokysdk-linux/usr/lib/arm-zephyr-eabi/
    gcc/arm-zephyr-eabi/6.2.0/include -isystem /opt/zephyr
    -sdk/sysroots/x86_64-pokysdk-linux/usr/lib/arm-zephyr-
    eabi/gcc/arm-zephyr-eabi/6.2.0/include-fixed -Os -g -
    Wall -Wformat -Wformat-security -Wno-format-zero-
    length -imacros /home/didrik/Dropbox/skole/ntnu/V2019/
    master/zephyr/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/include/generated/autoconf.h -
    ffreestanding -Wno-main -fno-common --sysroot /opt/
    zephyr-sdk/sysroots/armv5-zephyr-eabi/usr -mthumb -
    mcpu=cortex-m4 -fno-asynchronous-unwind-tables -fno-
    pie -fno-pic -fno-strict-overflow -Wno-pointer-sign -
    Wno-unused-but-set-variable -fno-reorder-functions -
    fno-defer-pop -Werror=implicit-int -Wpointer-arith -
    ffunction-sections -fdata-sections -mabi=aapcs -march=
    armv7e-m -std=c99 -o CMakeFiles/kernel_elf.dir/
    isr_tables.c.obj -c /home/didrik/Dropbox/skole/ntnu/
    V2019/master/zephyr/zephyr/samples/basic/blinky/build/
    nrf52_pca10040/zephyr/isr_tables.c
[100%] Linking C executable zephyr.elf
cd /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /usr/bin/cmake -E cmake_link_script
    CMakeFiles/kernel_elf.dir/link.txt --verbose=1
ccache /opt/zephyr-sdk/sysroots/x86_64-pokysdk-linux/usr/
    bin/arm-zephyr-eabi/arm-zephyr-eabi-gcc CMakeFiles/
    kernel_elf.dir/misc/empty_file.c.obj CMakeFiles/
    kernel_elf.dir/isr_tables.c.obj -o zephyr.elf -T
    linker_pass_final.cmd -Wl,-Map=/home/didrik/Dropbox/
    skole/ntnu/V2019/master/zephyr/zephyr/samples/basic/
    blinky/build/nrf52_pca10040/zephyr/zephyr.map -
    u_OffsetAbsSyms -u_ConfigAbsSyms -Wl,--whole-archive
    ../app/libapp.a libzephyr.a arch/arm/core/
    libarch__arm__core.a arch/arm/core/cortex_m/
    libarch__arm__core__cortex_m.a arch/arm/core/cortex_m/
    mpu/libarch__arm__core__cortex_m__mpu.a lib/libc/
    minimal/liblib__libc__minimal.a subsys/bluetooth/
    common/libsubsys__bluetooth__common.a subsys/bluetooth
    /host/libsubsys__bluetooth__host.a subsys/bluetooth/
    controller/libsubsys__bluetooth__controller.a subsys/
    net/libsubsys__net.a drivers/gpio/libdrivers__gpio.a
    drivers/entropy/libdrivers__entropy.a -Wl,--no-whole-
    archive kernel/libkernel.a CMakeFiles/offsets.dir/arch
```

```
/arm/core/offsets/offsets.c.obj -L"/opt/zephyr-sdk/
sysroots/armv5-zephyr-eabi/usr/lib/arm-zephyr-eabi
/6.2.0/armv7e-m" -L/home/didrik/Dropbox/skole/ntnu/
V2019/master/zephyr/zephyr/samples/basic/blinky/build/
nrf52_pca10040/zephyr -lgcc -mthumb -nostdlib -static
-no-pie -Wl,-X -Wl,-N -Wl,--gc-sections -Wl,--build-id
=none -Wl,--orphan-handling=warn -mabi=aapcs
```

A.4 The post build stage

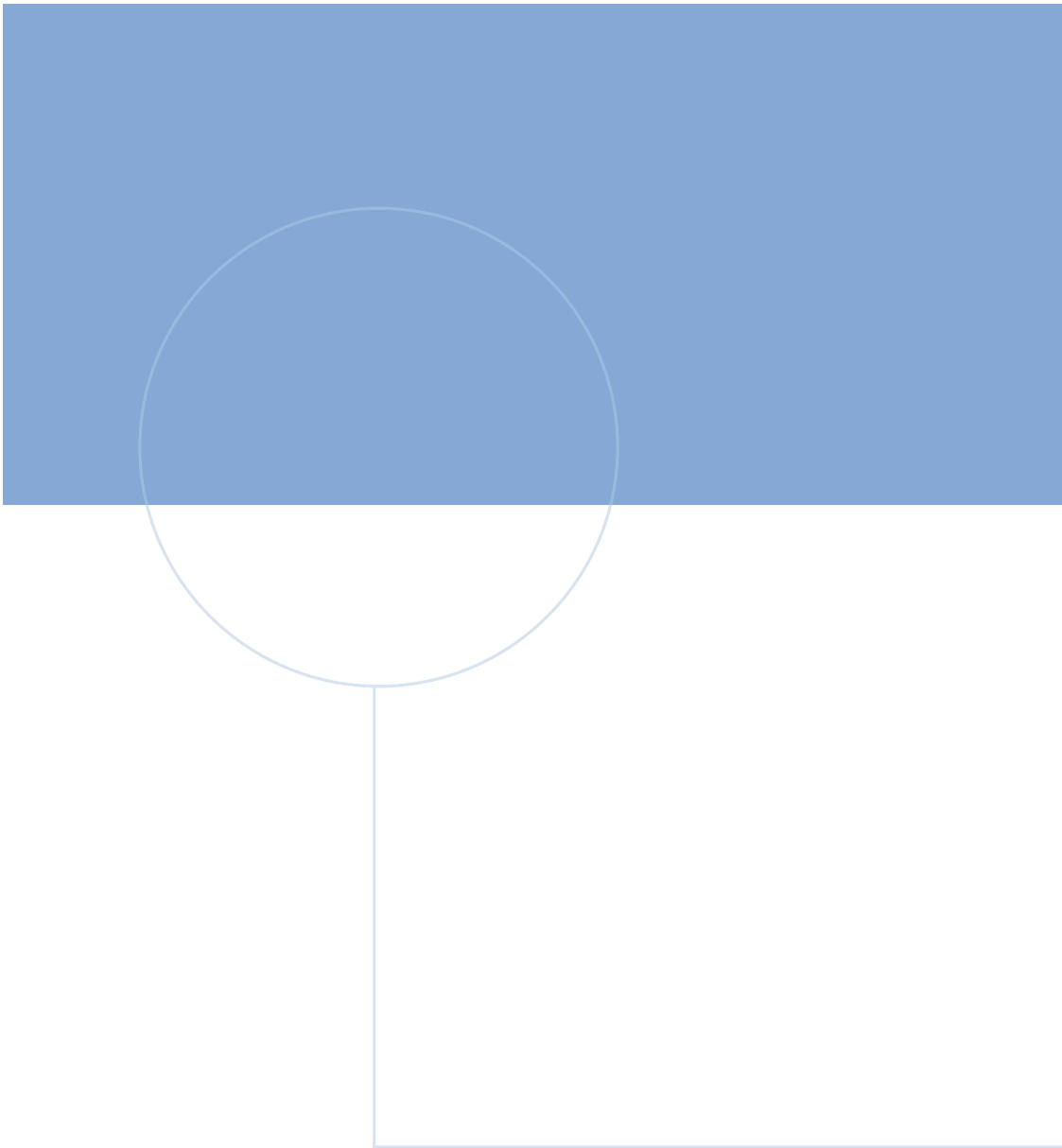
```
Generating files from zephyr.elf for board:
    nrf52_pca10040
cd /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /usr/bin/python /home/didrik/Dropbox/skole/
    ntnu/V2019/master/zephyr/zephyr/scripts/check_link_map
    .py zephyr.map
cd /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /opt/zephyr-sdk/sysroots/x86_64-pokysdk-
    linux/usr/bin/arm-zephyr-eabi/arm-zephyr-eabi-objcopy
    -S -Oihex --gap-fill 0xff -R .comment -R COMMON -R .
    eh_frame zephyr.elf zephyr.hex
cd /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /opt/zephyr-sdk/sysroots/x86_64-pokysdk-
    linux/usr/bin/arm-zephyr-eabi/arm-zephyr-eabi-objcopy
    -S -Obinary --gap-fill 0xff -R .comment -R COMMON -R .
    eh_frame zephyr.elf zephyr.bin
cd /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /opt/zephyr-sdk/sysroots/x86_64-pokysdk-
    linux/usr/bin/arm-zephyr-eabi/arm-zephyr-eabi-objdump
    -S zephyr.elf > zephyr.lst
cd /home/didrik/Dropbox/skole/ntnu/V2019/master/zephyr/
    zephyr/samples/basic/blinky/build/nrf52_pca10040/
    zephyr && /opt/zephyr-sdk/sysroots/x86_64-pokysdk-
    linux/usr/bin/arm-zephyr-eabi/arm-zephyr-eabi-readelf
    -e zephyr.elf > zephyr.stat
make[2]: Leaving directory '/home/didrik/Dropbox/skole/
    ntnu/V2019/master/zephyr/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
[100%] Built target kernel_elf
make[1]: Leaving directory '/home/didrik/Dropbox/skole/
    ntnu/V2019/master/zephyr/zephyr/samples/basic/blinky/
    build/nrf52_pca10040'
/usr/bin/cmake -E cmake_progress_start /home/didrik/
    Dropbox/skole/ntnu/V2019/master/zephyr/zephyr/samples/
    basic/blinky/build/nrf52_pca10040/CMakeFiles 0
```

Appendix B

Contents of accompanying zip file

In the zip file som følger med this report there are the following files and folders:

- make.log The logfile generated by make when building Zephyr's "Blinky" sample.
- zephyr-blinky-minimal/ Folder containing the source code of the Rust port of Zephyr's "Blinky" sample.



NTNU
Norwegian University of
Science and Technology