FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Swift for Embedded Systems
Student:	Bc. Alan Dragomirecký
Supervisor:	Ing. Petr Máj
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2019/20

Instructions

Familiarize yourself with the Swift language, its implementation and with the LLVM compiler system. Research the limitations embedded systems create for used programming languages and devise and implement changes to the Swift language so it can be used in embedded systems. Design Swift-based interface to the low-level peripherals and determine necessary compiler adjustments for this communication. As a proof of concept, implement a Swift library and LLVM extensions for embedded programming on the Cortex-M device family. Finally, compare your solution with other programming languages and peripheral control frameworks used in embedded systems.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague June 20, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Swift for Embedded Systems

Bc. Alan Dragomirecký

Department of Computer Science
Supervisor: Ing. Petr Máj

May 9, 2019

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 9, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Dragomirecký, Alan. *Swift for Embedded Systems.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Acknowledgements

I would like to thank my supervisor Ing. Petr Máj, for always providing me with valuable suggestions and feedback during my work on this thesis.

Special thanks go to my family and my friends for providing me with unfailing support throughout my years of study and through the process of writing this thesis.

Thank you.

Abstrakt

Po svém zveřejnění v roce 2014 se Swift stal okamžitě jedním z jazyků s nejrychleji rostoucí popularitou. Jeho hlavním zaměřením je vývoj uživatelských aplikací, brzy si ale našel své místo i v serverových aplikacích a nově i v datových vědách. Do této chvíle nebyla nicméně zveřejněná žádná práce zabývající se použitím Swiftu v těch nejmenších počítačích se značně omezenými výpočetními prostředky – ve vestavěných systémech. Tato práce si klade za cíl situaci změnit a být prvním krokem na cestě Swiftu k těmto zařízením. V práci je popsán proces přidání nové bare-metal platformy do kompilátoru Swiftu a nástrojů s ním spojených. V závěru je představen Swift jako možná alternativa k již existujícím řešením na poli vývoje pro vestavěné systémy.

Klíčová slova Swift, vestavěné systémy, kompilátor, internet věcí

Abstract

Released in 2014, Swift quickly become one of the fastest growing programming languages, and, in addition to its original field of use in application development, is finding its place on servers and recently also in data science. However, no work has been published regarding the use of Swift on the smallest computers with highly-constrained resources – embedded systems. This thesis aims to be the first step in extending Swift's possibilities towards this segment. It describes the process of adding support for a new bare-metal platform to the Swift compiler and its related tools. As a result, Swift is presented as a viable alternative to existing embedded platforms.

Keywords Swift language, embedded systems, compiler, Internet of Things

Contents

Introduction **1**

1 Background **3**
1.1 Swift . . . 3
1.2 Swift Runtime . . . 8
1.3 Swift Standard Library . . . 8
1.4 Swift Compiler . . . 9

2 Realization **11**
2.1 Initial Compiler Adjustments . . . 12
2.2 Building Swift Runtime and Standard Library . . . 15
2.3 Running in Emulator . . . 18
2.4 Testing . . . 22
2.5 Code Size Reduction . . . 26
2.6 Swift Package Manager . . . 38
2.7 Running on Hardware . . . 40

3 Evaluation **47**
3.1 Performance . . . 48
3.2 Code Size . . . 53
3.3 Using Swift for Embedded Systems . . . 54

Conclusion **59**

Bibliography **61**

A Contents of enclosed SD card **65**

List of Figures

11 Swift compilation process. 10

21 Visualization of sections assembling the executable's `.text` segment for initial version without code-size optimizations. (`libswiftCore.a` (97 %), `libstdc++.a` (0.02 %), `libc.a` (0.01 %), ...) 27

22 Code-size comparison for the full Swift standard library when built with different optimization settings. 27

23 An example of a dependency graph generated by our tool for analysis of the linking process. It shows the HelloWorld application. Nodes represent sections and edges dependencies between them. Size of a node represents the relative size of the section. 33

24 Dependency graph visualization of the HelloWorld application with split sections. Green sections are the ones discarded by the linker thanks to the implemented section splitting. 36

25 Comparison of the program's code size with the optimizations applied. 38

26 Directory structure of the packages used for the HelloWorld application (simplified). 41

31 Measured data from an oscilloscope – the microcontroller responding to an input signal (interrupt) by toggling another pin's polarity. 50

32 Results of the Fannkuch Benchmark. 52

33 Program-memory requirements comparison of programs running the Fannkuch benchmark. 53

34 Code size of different Swift applications. The code size does change dramatically when additional features of the Swift standard library are used. 55

List of Listings

1 Example of constrained protocol conformance in an extension. . 4
2 Demonstration of a violation of the *Law of Exclusivity*. [1] . . . 5
3 Example of error handling. 7
4 An example of the new spelling for compile-time conditions supporting the bare-metal environment. 12
5 Content of `main.swift` file of the "HelloWorld" application. . . 20
6 Part of a linker script handling one section of the runtime's metadata. 21
7 A command from `Makefile` of our minimal application responsible for compiling Swift source code and linking the application. 22
8 A trivial test from the Swift project located at `test/Interpreter/hello_func.swift`. 23
9 An example of a test using the StdlibUnittest library. 24
10 Part of a dominator tree of the graph depicted in Figure 23. Each line represents a single node, the root node is on the third line. Each line shows: *size including its dependencies*, *section name (object file name)*, and *(section size, section identifier)*. . 34
11 The manifest file of the HelloWorld package targeting STM32F439ZI. 42
12 Universal GPIO interface as defined in the `Hardware` library in `projects/Hardware/Sources/gpio.swift`. 45
13 A demo application using the STM32F4 library to interact with hardware. 46
14 Swift source code for the bit-banging performance test. 49
15 C source code using the Arduino platform for the bit-banging test. 49
16 MicroPython version of the source code for the bit-banging test. 49
17 Swift source code testing response time to interrupts. 51
18 C source code using the Arduino platform testing response time to interrupts. 51

19 MicroPython version of the source code testing response time to interrupts. 51

Introduction

An embedded system is a combination of hardware and software designed to perform one particular task within a larger system[2, 3]. Such a definition and a wide variety of other available definitions include everything from 8-bit microcontrollers with a few kilobytes of program memory on one side to powerful computers like our phones or smartwatches with memory sizes starting in gigabytes on the other. Those high-performance devices usually run a full Operating System (OS) and do not pose many challenges to the programs running inside them. However, as we move on the spectrum to the low-performance devices, a number of challenges start to appear. A traditional operating system is soon replaced with often a more lightweight Real-Time Operating System (RTOS), or we eliminate the OS entirely and run our program on so-called bare metal. Among the many reasons why this is being done, the most common one is the limited resources of the device.

This document focuses on low-performance embedded devices. Therefore, for the remainder of this thesis, "embedded systems" without further qualification are considered to be very small devices with a handful of megabytes of memory and traditionally a single processor core.

In attempting to optimally utilize the resources of an embedded system, the choice of a programming language is crucial. Languages with a high level of abstraction often do not provide convenient interaction with low-level fundamentals such as raw memory access, and the code-size and performance cost for their level of abstraction is usually too high. This leads to the number of languages being widely used on embedded systems to be much lower in comparison to other fields. The ruling languages for this segment are still C and C++ (where usually only a subset of C++ features is used). Of course, other languages are being used in some specific fields (such as Assembly, Lua, Ada), but their share in embedded systems in comparison to C/C++ is minimal.

Recently, many new languages have found their way into embedded systems. We can see Rust getting significant attention, but even some high-level interpreted languages are on the rise. With the boom of the Internet of Things,

communities around languages such as Python or Javascript have built implementations of those languages that can run on devices with as little as a few hundred kilobytes of program memory. It is hard to imagine those interpreters replacing C/C++ in the main segment of embedded systems, but it is an attractive alternative for makers and hobbyists.

Swift is the language which is the focus of this thesis. After being announced by Apple in 2014, it quickly became one of the fastest growing languages. Currently, seven percent[4] of developers name Swift as one of their most favored technologies. Its dominating field is application development for iOS and macOS. However, since Swift had been open sourced, the community made it possible to use it on Linux and Windows, with an official Server Work Group promoting the use of Swift for server-side development.

As Swift is expanding beyond iOS and macOS and is becoming a more mature programming language, it is tempting to explore its possibilities in the field of embedded development. If one allows the broader usage of the term "embedded systems," Swift is already heavily used there. An example would be application development for watchOS. Also, there are projects focusing on Swift on RaspberryPi and other ARM-based boards running Linux, including libraries to control their hardware peripherals. However, in our research, to date we have not found any indications of any published work in terms of running Swift on bare-metal and devices with highly limited computational resources. Nonetheless, this is no surprise, as the language is still young and under rapid development.

This thesis does not intend to propose Swift as being the replacement for C++ in embedded systems. Instead, it intends to be the first step on the way to broadening Swift's possibilities towards the Internet of Things, while researching the current limitations of the language and its implementation. There is the possibility of making Swift an alternative in the segment currently occupied by platforms such as Arduino or Micropython.

Chapter 1

Background

In this chapter, Swift is introduced, with a description of some of its key features with respect to embedded systems. Its compiler and runtime work are also outlined.

1.1 Swift

Swift is a general-purpose and multi-paradigm programming language. It was designed as a compiled, full-stack programming language with the intent to support high-level language constructs, while at the same time giving the compiler enough information to produce highly optimized machine code. The language has support for both value and reference types with an expressive and safe type system. It guarantees memory safety, implements both static and dynamic dispatch, and a significant amount of attention has been given to the prevention of common programming errors (Optional type instead of "nullptr," raw pointers are not exposed by default, there is no C-style for loop, etc.).

1.1.1 Protocol-Oriented Programming

From its very origins, Swift was designed to be able to interact with Objective-C, as that was the primary language used on Apple platforms at the time. Implementing all Objective-C's features required for seamless cooperation between the two makes Swift as capable in Object-Oriented Programming (OOP) as Objective-C is. Also, Swift offers immutable data types, first-class functions and other aspects making it viable for Functional Programming. Nevertheless, in the very heart of Swift, we would not find any of those programming paradigms. It would rather be a paradigm that Swift's authors call Protocol-Oriented Programming.

Protocol-oriented programming is an evolution of OOP, aiming to solve some of its problems. Instead of defining a shared object's functionality in

```swift
protocol WordCountable {
    func numberOfWords() -> Int
}

extension String : WordCountable {
    func numberOfWords() -> Int {
        return self.split(separator: " ").count
    }
}

extension Array : WordCountable where Element : WordCountable {
    func numberOfWords() -> Int {
        return self.reduce(0, { $0 + $1.numberOfWords() })
    }
}

["an array", "of strings"] is WordCountable // true
["an array", "of strings"].numberOfWords()  // 4
["an array", "with an integer", 2] is WordCountable // false
["an array", "with an integer", 2].numberOfWords()
// compiler error
```

Listing 1: Example of constrained protocol conformance in an extension.

terms of class inheritance, Swift prefers to express this in terms of the protocols they implement. This itself can be found in many other object-oriented languages (interfaces in C#, traits in Scala), but Swift adds multiple features, making this approach much more powerful.

One of these is *extensions*. It comes from Objective-C, where it is called "categories" – ability, to extend an existing class at runtime with a new method. Swift enhances this concept with the possibility to extend structs and enums. Furthermore, Swift allows making the extension of a type conditional (for example, we can add a method to the generic Array type for all its instances, where the elements of the array implement some specific functionality – see Listing 1).

Extensions together with protocols are the key for Protocol-Oriented Programming. It is not merely that protocol can be described as a composition of other protocols, or that a default implementation can be provided for some of its requirements. Existing types can easily be extended with conformance to new protocols or even better, those types can automatically gain functionality based on some fundamental protocol they implement.

1.1.2 Memory Management

Swift's choice of memory management is Automatic Reference Counting (ARC). ARC is a concept whereby each instance of a reference type (class) stores the number of references to it (next to the instance data). A compiler automatically inserts instructions incrementing (*retain*) and decrementing (*release*) it. At runtime, when the reference count reaches zero, the object gets deallocated.

We can compare the concept of ARC in Swift to the usage of smart pointers in C++. Their proper usage should generate the same behavior and comparable performance to Swift's ARC. However, Swift's ARC is a language-level feature enabling the compiler to do further optimizations.

The choice of ARC versus the traditional Tracing Garbage Collection (GC) also has a disadvantage – a programmer must be aware of potential cyclic dependencies between instances and resolve them manually (by using the `weak` reference type). GC systems are usually able to detect cyclic dependencies and resolve them. On the other hand, they introduce further problems – latency issues because of GC pauses.

1.1.3 Memory Safety

A long-term goal of Swift is to achieve memory safety. By default, Swift makes sure a variable is not accessed before its first use, after it has been deallocated, or that array indices are not out of bounds. A more complex assurance Swift applies is *exclusive access* to variables (also called the *Law of Exclusivity*). Even though this is a typical problem for concurrent or multi-threaded code, the problem exists in a single-threaded program not involving any concurrency. [1, 5]

Listing 2 demonstrates such a violation of exclusive access. The function `increment` has an `inout` argument `number`. As a function can write to `inout` argument at any point within its body, the caller has to have write access for the argument for the duration of the call. In this case, that means acquiring write access at line 7 until the function returns. However, the body of the function reads the global value `stepSize`, therefore requiring read access.

```
1 var stepSize = 1
2
3 func increment(_ number: inout Int) {
4     number += stepSize
5 }
6
7 increment(&stepSize) // Error: conflicting accesses to stepSize
```

Listing 2: Demonstration of a violation of the *Law of Exclusivity*. [1]

Having an overlapping read and write access to a variable is a violation of the exclusivity access being described.

Even though the example in Listing 2 rewritten to some other language (e.g. C++) would be a perfectly valid code, it can often lead to confusing results. More importantly, when the compiler can not rely on exclusive access to variables, it leads to very conservative optimizations [6].

To enforce the Law of Exclusivity, Swift currently implements checks in multiple places. The compiler generates an error at compile-time when such violation can be found using static analysis. Alternatively, the compiler generates runtime checks, possibly aborting the program's execution (thus introducing some performance cost).

1.1.4 Error Handling

An interesting feature of Swift is its error handling. Before explaining the semantics and implementation of its `throw`/`catch` mechanism, the following is an outline of the types of errors we generally deal with:

simple domain errors An example can be: `"not-a-number".toInt()`.

recoverable errors Errors a programmer should be encouraged to handle such as "network timeout" or "file not found".

universal errors Errors a programmer usually should not worry about. An example might be a memory allocation failure or stack overflow.

logic failures A logical error in user's code (such as an out-of-bounds access to an array). [7]

Simple *domain errors* are solved in Swift by returning `Optional<T>` type, thus forcing the user to handle them, but without the burden of a heavy `try`/`catch` syntax.

Universal errors are not usually solvable at runtime, as those can happen anywhere in the code. Attempting to recover from them would introduce a high risk of leaving the program in an invalid state. The same is true for *logical failures*, as those represent some mistake in the user's code and there is no safe way to continue execution of the program without fixing the code first. Therefore, Swift currently handles those errors by stopping the execution of the program.

Swift implements a mechanism similar to exceptions (from a syntax point of view) to handle *recoverable errors*. We show an example of its use in Listing 3. An error is thrown by using the `throw` keyword and an error can be anything conforming to the `Error` protocol (which has no requirements). An important part is that all functions by default cannot throw an error. If they do, they *must* be marked `throws`. Also, all function calls that are can throw an error must be explicitly marked with the `try` keyword and be enclosed

```swift
enum AnimalError : Error {
    case UnknownAnimal(animal: String)
}

func makeSound(of animal: String) throws {
    switch animal {
    case "cat":
        print("meow")
    case "dog":
        print("bark")
    default:
        throw AnimalError.UnknownAnimal(animal: animal)
    }
}

do {
    try makeSound(of: "snake")
} catch AnimalError.UnknownAnimal(let animal) {
    print("unknown animal: \(animal)")
} catch {
    print("an error: \(error)")
}
```

Listing 3: Example of error handling.

within a `do {} catch {}` block or be inside a throwing function. Those rules force the user to be aware of possible errors and to be encouraged not to ignore them.

Implementation

As Swift's error handling is syntactically comparable to C++'s exceptions, we can compare the implementations of those two. C++ uses an implementation based on stack unwinding. The assembly of call sites to functions that may throw an exception (which is expected by default) is the same as if no exceptions were in place. The function is not returning any information to the callee as to whether an exception occurred. Instead, the compiler generates tables for the C++ runtime with information on how to safely unwind the stack of every function. When an exception is thrown, the runtime is called with the given exception. The runtime starts unwinding the current stack unless it finds an appropriate exception handler. This implementation of exceptions is called "zero-cost." This means that, unless the code throws an exception, its performance is the same as if no exceptions were allowed. On the other side, throwing an exception is an expensive operation, and the required runtime with all the necessary information for unwinding the stack represents a

significant increase in code size for embedded systems.

Swift's approach is different. As Swift defines its own calling convention, when calling a function that can throw an error (it is marked `throws`), the caller reserves a specific register for the error. When the function returns, the caller has to check this register and react appropriately. This system does not penalize code that does not use error handling. When used, it can be compared to a slightly more optimized version of manual error passing in C (using a pointer to error variable).

1.2 Swift Runtime

Runtime of a programming language is the part of an executable that is included implicitly by a compiler to support the execution of code explicitly written by a programmer. In the simplest case, such support might mean setting up the stack and calling the program's `main` function. However, most high-level languages today require greater runtime to support language constructs such as error handling or dynamic type casting. Those features are then implemented in a separate library, usually referred to as the runtime library and this library is automatically included in every program by the compiler.

The size of the runtime library is an important factor in the case of embedded systems, as it represents a fixed price in respect to code size. The language C, heavily used on embedded systems, requires minimal runtime. It includes just the essentials such as zeroing the section of uninitialized data or calling the global constructors before transferring control to the `main` function. The runtime of C++, in comparison to C, gets significantly bigger. It has to implement stack unwinding to support exceptions, symbol unmangling for dynamic type identification or dynamic dispatch for `virtual` functions. Nonetheless, as those features are often not needed, it is possible to disable them, thus reducing the size of the runtime to a minimum.

Swift's runtime library implements a wide variety of features. To name a few, it covers memory management, reflection, dynamic type casting, protocol conformance registration or generic instantiation. Swift's runtime library also requires the compiler to emit metadata for every defined type, protocol and implemented protocol conformance. The library itself is implemented in C++ but does not require features like C++ exceptions or C++ runtime type identification. Therefore, those features can be disabled, slightly reducing the library's size.

1.3 Swift Standard Library

The Swift standard library is a swift module (also referred to as SwiftCore), that is implicitly imported and available in every Swift source code. It imple-

ments features that are essential to most programs, such as fundamental data types (Int, Double, String), common data structures (Array, Dictionary) and standard protocols e.g. Collection or Equatable.[8]

Even though the library implements the very essentials for every program like the `Int` or `Optional` type, it is fully implemented in Swift with some bindings to the compiler. It thus makes it easy to change or, in theory, be replaced by another one.

As Swift's standard library aims to be as small as possible and provide just the features which are felt to be "part of the language," it leaves a lot to be implemented and shipped with the standard toolchain somewhere else. Therefore, things like file access or manipulation with dates are implemented in separate "core libraries" (e.g. `Foundation`).

1.4 Swift Compiler

The compiler incorporates several other projects to achieve its goal. Mainly, it uses Clang to import C and Objective-C code as Swift modules and to compile them. As a backend, to generate machine code, it uses LLVM. This is an important fact, as LLVM supports architectures used on embedded devices, thus ensuring that one will not have to deal with machine code generation when bringing Swift to the embedded world.

1.4.1 Compilation Process

The process of compilation is depicted in Figure 11. Swift sources are firstly parsed by the `Parser` transforming them into the Abstract Syntax Tree (AST). C and Objective-C sources are imported using the `ClangImporter` component, also transforming them into AST.

The AST is then run through the semantic analysis module, which is responsible for inferring types and checking for semantic issues and errors. At the end of this step, the AST is well-formed, fully-typed and safe for code generation.

The next step in the compilation process is taking the AST and transforming it into Swift Intermediate Language (SIL). The purpose of this high-level intermediate representation is to allow further analysis and high-level optimizations. The compiler *always* runs a set of mandatory passes analyzing dataflow and correctness of the program. Based on configuration, it then performs optional optimizations (such as generic specialization or automatic reference counting optimizations).

When the processing of SIL is finished, the `IRGen` module transforms it into LLVM's IR Intermediate Representation. LLVM then performs further generic low-level optimization on this representation and then it generates machine code.

1.4.2 Building The Toolchain

Building a ready-to-use Swift toolchain is not an easy task. Many variables must be taken into account – the platform we want to run the compiler on, or the targets we want the compiler to support (as the compiler is often used as a cross-compiler). However, there are numerous other different aspects, such as building and configuring all the dependent projects, compiling the runtime, standard library and Swift's core libraries or preparing dependencies required for testing the toolchain.

Swift's project uses CMake to handle this problem. CMake is a cross-platform set of tools designed to control the compilation process using simple platform and compiler-independent configuration files. In order to add support for cross-compilation to bare metal, not only will adjustments have to be made to the compiler's code itself, but also substantial changes have to be made to the process of how the toolchain is built.

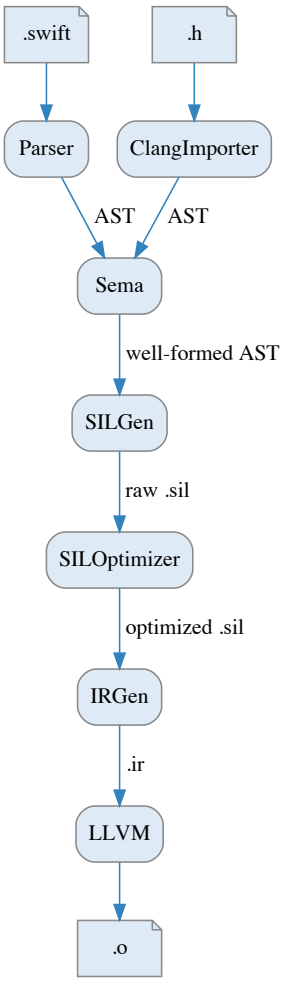

Figure 11: Swift compilation process.

Chapter 2

Realization

Before diving into the process of adding support for a new platform, let us overview what the different components of the compiler already support. As earlier mentioned, Swift is often used on platforms like watchOS or iOS. In addition, it is possible to target Android devices or Linux on RaspberryPi. All those platforms run on chips with ARM architecture – more specifically, the chips usually implement ARMv7-A or ARMv8-A architectures. The "A" at the end of the architecture name is short for Application Profile (A-Profile), and those architectures are the ones used in the most powerful ARM processors.

As we want to step-down on the chip's performance scale and find the limits of Swift on bare metal, it is natural to look at some other – lower performance – families of chips. In addition to the high-performance A-profile, ARM currently offers two other architecture profiles. R-Profile optimized for Real-Time Operating Systems and M-Profile designed for microcontrollers. Our choice is the ARMv7-M architecture of the M-Profile family. Those chips are highly optimized for power-efficiency and target the smallest embedded applications. Even though we could have chosen architecture outside the ARM world, the M-Profile architectures have good support in LLVM and Clang and are very popular in the embedded world, making them an excellent starting place.

The M-Profile are 32-bit RISC ARM processors, but they do not include the standard ARM instruction set. Instead, they implement an instruction set called Thumb. This one is simplified to allow the smallest silicon die while reducing code size (mixing 16-bit and 32-bit instructions). The microcontrollers with the M-Profile core (Cortex-M) usually offer program memory sizes somewhere between tens of kilobytes to a few megabytes. This also works well for our purpose, as it is safe to expect that the code size of the standard library and runtime will fall somewhere within this range.

In this chapter, we add support for a new platform called BareMetal into the Swift Compiler (which will be referred to as *bare-metal* in this document).

```
#if os(macOS)
print("hello world from macOS")
#elseif os(none)
print("hello world from baremetal")
#else
print("hello world from somewhere else")
#endif
```

Listing 4: An example of the new spelling for compile-time conditions supporting the bare-metal environment.

The architectures supported are going to be ARMv7-M and ARMv7E-M. The first section provides an overview of the necessary changes to the compiler and its build process. The following section describes how we handled building the runtime and standard libraries for the new platform. As soon as the system can produce some machine code for bare metal, it is time to test it in an emulator – that is described in Section 2.3.

In Section 2.4, we ensure that the implementation works as expected by porting and running a part of Swift's test suite on the new platform. Next, a code-size issue will be addressed in Section 2.5. Finally, in the last two sections, we add support to the Swift Package Manager and run our solution on real hardware (Sections 2.6 and 2.7).

2.1 Initial Compiler Adjustments

Adding support for a new platform to the compiler requires a lot of small changes across the project. Those changes are usually self-explanatory. However, identifying the parts to be changed in such a large project is not trivial. In the following paragraphs, the changes are outlined which are needed to be made in order to build a Swift compiler that is aware of the new platform.

2.1.1 Compile-time Conditions

The Swift compiler does not have any preprocessor. However, to allow code to be conditionally included based on the operating system, compiler version, and other compile-time variables, it implements simple *Conditional Compilation Blocks*. One of the supported conditions is the operating system the code is being compiled for. An example of such a condition can be `#if os(Windows)`. We added a new possible value for this condition, which is spelled `os(none)`. An example of it is shown in Listing 4. Another considered spelling was `os(BareMetal)`. Such a spelling might be clearer to a reader seeing it for the first time, but we rejected it as being inaccurate (as there is no operating system named BareMetal).

Some platforms also support a compile-time condition based on the version of the operating system (e.g. `#available(iOS 9.0, *)`). We decided not to support this, as there will be no need in a short time to have fine-grained API version management. Also, a lot of other platforms supporting Swift do not implement this (such as Linux, Windows or Android).

Affected files: 1) `src/swift/include/swift/Basic/LangOptions.h` and 2) `src/swift/lib/Basic/LangOptions.cpp`.

2.1.2 Platform And Target Triple

The platform we added is named "BareMetal" (inspired by Clang and other compilers, where it has the same name). Initially, we support two architectures – ARMv7-M and ARMv7E-M. When using a cross-compiler like Clang or Swift Compiler, the general way of configuring it for a specific platform and architecture is the *Target Triple*. The format of a target triple is: `<arch><subarch>-<vendor>-<sys>-<abi>`. For our newly supported targets, the triples are `thumbv7m-unknown-none-eabi` and `thumbv7em-unknown-none-eabi`. The `<vendor>`, which is `unknown`, can be omitted, as it is unimportant in our context and `unknown` is the default. `none` in the place of `<sys>` means running without an operating system (bare metal). `eabi` at the end of the triple guides the compiler to use the ARM's Embedded Application Binary Interface (EABI).

When using the Swift compiler, we use the `-target` option to specify the target triple. The compiler then has to know what platform to use with the specified triple. Such a mapping had to be implemented in `src/swift/lib/basic/Platform.cpp`.

2.1.3 Compiler Driver and Toolchain

When we run the `swift` command line utility to build a program out of some source code, the compiler's driver is run. The driver is responsible for performing all the necessary steps in order to produce the final product. Such steps can be invoking Clang or Swift's frontend to produce object files out of any source code and then invoking a linker to link the object files. Those actions are platform-dependent. The driver has to know things such as where to look for header files, what linker to use, where standard C++ libraries are stored to link against or where to find the Swift runtime library for the given platform and architecture. The package with all those dependencies for a given platform is called a *Toolchain*, and the compiler decides what Toolchain to use based on the Target Triple.

Toolchains for bare-metal systems are usually more complicated in their structure, as they include support for a range of hardware devices with varying parameters. Instead of hard-wiring support for some specific toolchain into the driver, we switch the driver to a very generic behavior when targeting bare

metal and leave all the options to be specified via the command line. This decision makes building a Swift program by hand harder[1] but does not tie the compiler to some specific bare-metal toolchain.

Affected files: 1) `lib/Driver/Driver.cpp`, 2) `lib/Driver/UnixToolchains.cpp` and 3) `lib/basic/Platform.cpp`.

2.1.4 Changes In How The Compiler Is Built

The entry point for building the Swift toolchain is a Python script located at `utils/build-script`. This script abstracts the build process, so users do not have to interact with CMake directly. The script validates the user's configuration for the build, translates it to CMake flags and invokes CMake. CMake loads the `CMakeLists.txt` in the root of Swift's repository, which in turn loads all the `*.cmake` files in `cmake/modules`. Then, based on configuration, it loads `CMakeLists.txt` of the project subdirectories.

Our changes add a few new command line options to the `build-script` configuring the build for bare metal. Furthermore, it adds support for the new platform and architectures, configures LLVM to target those architectures and handles compilation of the Swift standard library and runtime. How the libraries are built for bare metal is covered in the next section (2.2).

For completeness, the following is an overview of the changes:

`utils/swift_build_support/swift_build_support/targets.py`
: Registered the new BareMetal platform and its architectures.

`utils/build_swift/driver_arguments.py`
: Added new `--baremetal`, `--skip-build-baremetal`, `--skip-test-baremetal` and `--baremetal-toolchain` options.

`utils/build-script` and `utils/build-script-impl`
: Registered command-line options listed above. Added configuration of the new platform.

`cmake/modules/SwiftBareMetalSupport.cmake`
: Contains utility functions for the new platform. Mostly functions returning command line options for cross-compilation of the standard libraries and runtime.

`cmake/modules/AddSwift.cmake`
: Added support for building Swift libraries targeting the new platform.

`cmake/modules/SwiftConfigureSDK.cmake`
: Added macro setting up global variables for the BareMetal platform.

[1] This is not an issue with support of Swift Package Manager – see Section 2.6.

`cmake/modules/SwiftSetIfArchBitness.cmake`
: Added support for the new architectures.

`CMakeLists.txt`
: Added configuration of the BareMetal platform, if requested in CMake flags from the `build-script`.

2.2 Building Swift Runtime and Standard Library

In the previous section, the necessary adjustments to the Swift project were described to produce a Swift compiler aware of the new bare-metal platform. However, when building the whole toolchain, an important part is building the Swift runtime and standard library for the targeted platform. We have mentioned the parts of the projects that had to be changed in order to configure such build, but we have not covered what those changes were. In this section, we explain how we ported the runtime and standard library to bare metal and how they are built.

The standard library is entirely written in Swift and has two dependencies. The first is a library called StdlibStubs, which is implemented in C++ and contains an implementation of some platform-dependent features (like `random()` or getting the environment variables). The second dependency, as of every Swift source code, is the Swift runtime. The runtime library is also written in C++ and is dependent on the C++ Standard Library. The C++ Standard Library, in turn, incorporates C Standard Library. The languages C++ and C also require their runtime libraries. Thus, we have a nice list of dependencies to resolve for bare metal.

2.2.1 C/C++ Standard Libraries

As the project already uses Clang and LLVM, it would make sense to cross-compile its implementation of the C++ standard library – `libc++`. Unfortunately, the tools around LLVM do not have their own implementation of the C standard library. Therefore, we would also have to find and cross-compile some external implementation suitable for bare metal of the C standard library.

The second option we considered is to use a ready-to-use GNU ARM Embedded Toolchain. It includes all the key elements to build a C/C++ source code when targeting bare-metal, such as the GNU C/C++ Compiler, C++ standard library (*libstdc++*) or C standard library (*newlib*). We decided to go with this option, as it covers all our needs in one additional dependency and is already highly optimized for embedded systems.

2.2.2 Standard Libraries' Search Paths

When compiling a C/C++ source code, the compiler has to be able to locate the headers of the C++ standard libraries, or the linker has to know where to find all the compiled standard libraries. Those *search paths* can get complicated, when using the GNU ARM Embedded Toolchain, as they change based on the targeted device. The GNU Compiler – `arm-none-eabi-gcc` – knows the toolchain and sets up the search paths automatically. Unfortunately, the choice of using `arm-none-eabi-gcc` instead of Clang would most probably introduce more problems than it would solve, so there is no option but to manually configure Clang to the same configuration that `arm-none-eabi-gcc` uses implicitly.

As the layout of the GNU ARM Embedded Toolchain can change with every version, hardcoding its search paths in the Swift project would be impractical. Instead, we obtain them by running `arm-none-eabi-gcc` itself. To get the header search paths, we collect the command-line arguments we want to run `clang` with that affect the search paths (like `-target` or `-march`). Then we invoke `arm-none-eabi-gcc` on an empty file, with the arguments mentioned above, running only its preprocessor (`-E`) and switching the preprocessor to verbose output (`-Wp,-v`). This results in the preprocessor printing the header search paths it would use. We parse them and configure CMake to use them with Clang when compiling the Swift runtime and other Swift's underlying C/C++ libraries. Configuring library search paths (for the linker) is even easier, as `arm-none-eabi-gcc` has an option `-print-search-dirs` which prints them for us in a similar way.

2.2.3 Porting The Runtime And StdlibStubs

Having the compiler adequately configured, it is time to look at the source code of the runtime and StdlibStubs library. The source code has to be multiplatform, so it often uses the C preprocessor to include parts of code based on the targeted platform. We have introduced a new `_BAREMETAL` macro that is defined when compiling for bare metal. Some of the changes we made are:

- Disabled backtrace reporting in fatal-error logs, as we will not have the required machinery to support this on bare metal.
- Disabled dynamic lookup of symbols when dynamically linking a library, as executables for bare-metal are always going to be statically linked.
- Swift on most platforms expects a valid pointer to be higher than 4096 and uses lower values to store some extra information. On bare metal, we had to disable this behavior by setting the "least valid pointer" to 1, as programs are often being linked to addresses close to zero and this would introduce a potential problem. We can expect this change

to have negative consequences on Swift's performance and it is worth future investigation.

- For platform-dependant functions in StdlibStubs that are not expected to be used on bare-metal systems (e.g. getting the environment variables) we just replaced their body with a call to Swift's `fatalError`. For those which a user might want to be able to override (e.g. writing to stdout), we provide a weak symbol that the user can implement and it will automatically replace the default implementation at link-time.

- Unicode support in `SwiftCore` partially uses functions in StdlibStubs, and those functions are then implemented using libraries from an open-source project ICU. For the initial port of Swift to bare-metal, we decided to drop Unicode support, so those functions currently also call `fatalError`.

2.2.4 Missing Libraries

When trying to compile the runtime and StdlibStubs with the changes mentioned above, we still get many compiler errors. One problem is that the code uses some functions from the C++ Thread Support Library and pthread, such as `std::call_once` or `pthread_getspecific`. Those features are not implemented in the standard library provided by the GNU ARM Embedded Toolchain, as there is no support for threads when running without an operating system. One option would be to go through all the call sites and remove the use of those functions. However, this would make the code less readable and introduce a lot of small changes in the codebase, making it harder to merge our work to the upstream potentially. Instead, we decided to implement a simple single-threaded version of the library and have it as a dependency to the runtime and StdlibStubs.

A similar problem to the previous one araises when we try to link an executable. We find that there are many missing symbols, mostly from the *atomic* library (`libatomic`). This library is also not available in the GNU ARM Embedded Toolchain. We took the same approach as mentioned above and implemented a simple single-threaded version of those symbols.

It is worth mentioning that, even though there is no support for threads on bare metal, we still often have to deal with concurrency-related problems. The source of concurrency on bare metal are interrupts. This makes space for improvements, and future implementations of those libraries can make the execution safer by concerning interrupts.

The source code for the features mentioned above is located in a separate folder `src/swift/stdlib/public/stubs-baremetal`. It all compiles to a standalone library called StdlibStubsBaremetal, which can then be linked to bare-metal executables.

2.2.5 Objects' Incompatibility

Finally, we can build a simple Swift toolchain and try to produce the first executable for bare metal. However, if we do so, the linker will emit several warnings. The problem is that all our objects compiled by the Swift Compiler or Clang do not use *short enums*, but the standard libraries from the GNU ARM Embedded Toolchain do so. When short enums are enabled (flag `-fshort-enums`), it allows the compiler to use the smallest possible data type for an enumeration that can hold all the enumerator values. Using short enums can usually improve memory usage at the expense of performance. The problem is that linking two objects with different settings for short enums might produce an invalid binary.[9]

The first option is to change the way our libraries are built. Changing it for Clang and the C/C++ source code is not a problem. However, Swift does not support such an option and produces object files having short enums flag disabled. There is a chance that this would break cooperability between Swift and C code, so we decided to go with the other option and compile a GNU ARM Embedded Toolchain ourselves with short enums disabled.

To make the custom build of GNU ARM Embedded Toolchain easily reproducible, we created a bash script located at `src/swift/utils/baremetal/build-gcc-arm-none-eabi`. It does the following:

1. Downloads the source code of GNU ARM Embedded Toolchain from `developer.arm.com`.
2. Unpacks it under `src/arm-none-eabi-gcc` alongside other projects.
3. Applies our patches located at `src/swift/utils/baremetal/patches-gcc-arm-none-eabi` on the toolchain's build scripts disabling the use of `-fno-short-enums`.
4. Builds the toolchain using its build scripts.

With this version of the GNU ARM Embedded Toolchain, we are finally able to compile some program for bare metal without any warnings, and it is time to try to run it in an emulator.

2.3 Running in Emulator

In previous sections, we have built a full Swift Toolchain with support for the new bare-metal platform. It is time to compile and run the first program. Debugging hardware is usually not the easiest thing to do; therefore we decided to run the program within an emulator first. The one we chose is QEMU, which is an open-source machine emulator, supporting a wide range of architectures and allowing everything from emulation of small embedded devices to virtualization of a desktop computer[10].

QEMU supports a lot of different ARM machines, but when it comes to the architectures we are interested in, we can only choose between two very similar devices from Stellaris. We selected a machine with code name *lm3s811evb*, which is a Stellaris LM3S811 with Cortex-M3. When trying to compile the simplest program for bare metal with the newly built toolchain, it has a size of almost ten megabytes. The device we have just selected has 64Kb of program memory. As now is not the time to implement optimizations shrinking the program by a few orders of magnitude in size, we decided to change the size of the program memory in a source code of QEMU and compile it ourselves. This allows us to test programs of arbitrary size without the limits of some specific device.

To make the build of our version of QEMU easily reproducible without the need to have its full source code in our project, we applied the same approach described in the previous section with the custom build of `arm-none-eabi-gcc`. A script is available at `src/swift/utils/baremetal/build-qemu` which downloads, patches and builds QEMU in `src/qemu` directory.

The following paragraphs describe how a minimal "Hello World" application can be built and then run inside this version of QEMU.

2.3.1 Assembling Minimal Application

Let us first outline what a typical application start looks like on the M-Profile architectures. When the microcontroller is powered up, it sets up the Stack Pointer register with the initial value loaded from address 0x04. Then, it starts execution at the address stored at 0x00. This is where a *startup* code begins. It 1) copies all the data of statically initialized variables to RAM, 2) clears the parts of RAM reserved for uninitialized variables, 3) calls `SystemInit()` function, which usually sets up essentials of the chip and 4) calls the entry function of the program called `_start()`. The `_start` function is usually the entry point of C Runtime, which only calls all static initializers and then gives control to the `main()` function.

The program we are going to run is simple – it will just send some text to the serial port and then go to an infinite loop. The files of the project are:

`main.swift`
: The Swift source code of the program. The code is listed in Listing 5. Instead of using the `print` function from the standard library, for simplicity, we define our own. This function iterates through all the characters of the string and writes them to a specific register of the serial port peripheral.

`startup.c`
: This file contains a standard *startup* procedure mentioned in the previous paragraph.

```
1  func print(_ string: String) {
2      let uart = UnsafeMutablePointer<UInt32>(bitPattern: 0x4000C000)!
3
4      for character in string {
5          uart.pointee = UInt32(character.asciiValue!)
6      }
7  }
8
9  print("Hello World from Swift on BareMetal!")
10
11 while(true) { }
```

Listing 5: Content of `main.swift` file of the "HelloWorld" application.

`crt.c`
: Contains a minimal `_start` function. When linking a bare-metal application using `arm-none-eabi-gcc`, the toolchain includes its implementation implicitly. We do not use `arm-none-eabi-gcc` for linking, so it is easier to define our own.

`linker.ld`
: Linker script for the application. It instructs the linker what the memory layout of the device is and where to place which sections of the application.

The linker script is based on a standard one for Cortex-M3, but has to handle one extra feature. Swift Compiler emits extra metadata used by the runtime for reflection, dynamic dispatch and other features of the language. Those metadata are emitted to specific sections, such as `swift5_type_metadata` or `swift5_reflstr`. Hence, we have to extend the linker script to make sure those sections will be properly located in the final product. Also, the linker script has to define `__start` and `__stop` symbols for those sections so that the runtime can locate them. Such an adjustment of the linker script for a single section is shown in Listing 6.

2.3.2 Compiling The Application

In order to compile the application, we have created a simple Makefile. The `startup.c` and `crt.c` files are standard C source codes, thus can be easily compiled using Clang and we will not go through their rules in the Makefile. On the other side, the compilation of `main.swift` and subsequent linking is more complex and worth mentioning. The invocation of the Swift compiler is depicted in Listing 7. The following must be done:

```
.text :
{
    . = ALIGN(4);
    __start_swift5_typeref = .;
    KEEP(*(swift5_typeref*))
    __stop_swift5_typeref = .;
} > FLASH
```

Listing 6: Part of a linker script handling one section of the runtime's metadata.

a) set the linker to be used to the one from the GNU ARM Embedded Toolchain,

b) instruct the driver to link the standard library statically (`-static-stdlib`),

c) set the target triple,

d) make sure the compiler processes any C header files with the `_BAREMETAL` macro defined[2],

e) link the C part of our application,

f) link Swift and C++ standard library,

g) set the search paths for any libraries we are linking to[3],

h) and pass options to the linker specifying the linker script.

2.3.3 Running The Application

Finally, running the application is as simple as starting the QEMU emulator with our just-built executable:

```
> qemu-system-arm -machine lm3s811evb -kernel build/main -nographic
Hello World from Swift on BareMetal!
```

[2]This is currently necessary. However, it would make sense to have the driver define this macro for us automatically when targeting BareMetal.

[3]We have manually listed the necessary paths here for simplicity. The preferred method would be to resolve them automatically, as described in Section 2.2.2 on page 16.

```
$(SWIFTC) \
    -use-ld=$(ARM_TOOLCHAIN)/arm-none-eabi/bin/ld \
    -static-stdlib \
    -target thumbv7m-unknown-none-eabi \
    -Xcc -D_BAREMETAL \
    -l:startup.o -l:crt.o \
    -lswiftCore -lswiftStdlibStubsBaremetal -lstdc++ \
    -lc -lg -lm -lgcc -lnosys \
    -L $(ARM_TOOLCHAIN)/lib/gcc/arm-none-eabi/*/thumb/v7-m/nofp \
    -L $(ARM_TOOLCHAIN)/arm-none-eabi/lib/thumb/v7-m/nofp \
    -L $(SRC_ROOT)/build/$(BUILD_VARIANT)/swift-$(HOST_VARIANT)/... \
    -L ./build \
    -Xlinker -T -Xlinker ./linker.ld \
    main.swift -o ./build/main
```

Listing 7: A command from `Makefile` of our minimal application responsible for compiling Swift source code and linking the application.

2.4 Testing

Up until this point, we have made it possible to compile a Swift program for bare metal and checked our solution by running a minimal application within an emulator. We have already mentioned some issues that make it harder to run any application on real hardware. Most importantly, code size – the compiled version of the minimal application required several megabytes of program memory, which is more than we can afford on the M-Profile devices. Before attempting the process of reducing the code size, it would be helpful to be able to thoroughly test the solution we already have, so we can quickly identify bugs that could potentially be introduced.

The Swift project applies multiple approaches to testing. The primary test suites are based on a tool called *lit*, and they consist of thousands of test suits. In the long term, it is necessary to ensure that all of them pass, even when targeting bare metal. However, for our purposes, we focus on a selection of tests testing the standard library and execution of the language itself. [11]

2.4.1 Implementing The Semihosting Interface

To run the tests, we will use QEMU as described in the previous section. What such a test can look like is shown in Listing 8. On the first line, we instruct *lit* to compile the file, run it and then pipe the standard output of the program to a *FileCheck* utility, which compares the piped output to the `CHECK:` directives in the file. This ensures that the program prints the expected text “Nice shoes”.

In order to pass such tests when running inside QEMU, the program has

```
// RUN: %target-run-simple-swift | %FileCheck %s
// REQUIRES: executable_test

// CHECK: Nice shoes
func hello() {
  print("Nice shoes")
}

hello()
```

Listing 8: A trivial test from the Swift project located at `test/Interpreter/hello_func.swift`.

to be able to interact with the system. For example, the `print()` statement must be bridged adequately to the standard output of QEMU. Another example would be that a call to the `fatalError` function must make QEMU exit with a non-zero status code so that a failure within a test can be detected. Fortunately, QEMU implements the Semihosting interface defined by ARM. This interface enables a bare-metal program to communicate with the environment it is running in, such as QEMU, by 1) placing arguments to predefined registers, 2) executing the `BKPT` instruction, 3) which stops the execution and resolves the request of the running program and 4) the program reads back the result from registers.

To make the Swift standard library use the Semihosting interface, we have to provide an implementation of some symbols from the C standard library, such as `_write` or `_exit`. When targeting bare metal, those symbols are defined in the `nosys` library provided by the GNU ARM Embedded Toolchain. We have created a replacement implementation to `nosys` featuring:

1. Basic *file access*, including *stdin*, *stdout* and *stderr* (`_open()`, `_read()`, `_write()`, `_close()`, `_lseek()` and `_isatty()`)
2. Swift's `CommandLine.arguments` in respect to the arguments passed to QEMU on the command line.
3. Properly exiting QEMU with zero status code on success and non-zero on failure (`_exit`).

By linking the test programs against this library, we can make the emulation inside QEMU behave similarly to running the program natively on the host computer, making it possible for *lit* to inspect the program's execution.

```
// RUN: %target-run-simple-swift
// REQUIRES: executable_test
// REQUIRES: OS=none-eabi

import StdlibUnittest

var suite = TestSuite("Important aspects of life")

suite.test("We can run Swift on bare-metal") {
  #if os(none)
  let runningOnBareMetal = true
  #else
  let runningOnBareMetal = false
  #endif
  expectTrue(runningOnBareMetal)
}

suite.test("The answer is divisable by three") {
  expectTrue(42 % 3 == 0)
}

runAllTests()
```

Listing 9: An example of a test using the StdlibUnittest library.

2.4.2 Porting Required Libraries

By implementing the Semihosting interface, the requirements of a significant portion of the tests have been met. However, some tests do not use the *FileCheck* utility to inspect the output of the tests, but use a unit testing library instead. An example of such a test is depicted in Listing 9. The library it uses is called `StdlibUnittest` which is compiled as part of the toolchain when configured for testing.

`StdlibUnittest` provides all the necessities in order to define – in code – a test suite, add some tests to it and run it. By default, those tests are run concurrently by executing each of them in a subprocess. The library is implemented in Swift and is dependent on `SwiftPrivateLibcExtras` and `SwiftPrivatePthreadExtras` to support subprocesses and threads respectively. As there is no kind of support for threads or processes on bare metal, there is no option of porting those two libraries in a meaningful way. Instead, we dropped the dependency on those libraries when compiling for bare metal, and changed the default behavior of the library to run the tests sequentially and in-process.

The Swift code of StdlibUnitttest also imports another library – *Glibc* on non-Apple platforms or Darwin on Apple platforms. Glibc is the C standard

library imported through Clang Importer. On bare metal, we compile the code against another implementation of the C standard library – Newlib – which is mostly compatible with Glibc. Instead of introducing a new spelling for the C standard library and requiring the existing Swift code to import one or the other based on the platform, we decided to make Newlib available under the Glibc name, to make the existing code compatible.

2.4.3 Running The Tests

The previously presented examples of tests (Listings 8 and 9) compile and run the Swift code based on the first line of the test, which contains:

```
// RUN: %target-run-simple-swift
```

The directive `RUN:` instructs *lit* to execute a given shell command. In our case, the shell command is `%target-run-simple-swift`, which is substituted for a bash invocation which compiles and runs the current file. Those kinds of substitutions are defined for each platform in a `test/lit.cfg` file.

The mentioned substitution `%target-run-simple-swift` is defined in terms of other, more narrow-focused, substitutions, such as `%target-run` which has to expand to a command running given executable on the target or `%target-build-swift-dylib` which must build a dynamic library out of source code. Those substitutions can get quite complex (let us remember the compilation of a minimal program in Listing 7 on page 22). Part of the complexity comes from the fact that all the search paths of the GNU ARM Embedded Toolchain must be configured. As we have solved this already when configuring the build of Swift standard library in CMake, and because *lit* is also configured by CMake, we reused the method by which some compilation options are calculated. When configuring *lit*, we resolve the required compilation options in CMake and pass them to the `test/lit.cfg` file, where they are used to define the substitutions for bare metal.

With all the changes we have presented, we are finally able to run the tests we are interested in. We still had to go through some of them and make changes in order to support the new platform. Among others, the following had to be done: a) disable some tests ensuring program crashes, as our version of StdlibUnittest runs tests in-process and a crash would prevent other tests from running, b) disable tests testing features not available on bare metal, e.g. backtracing or c) disable tests requiring Foundation or other libraries not available on bare metal. With that, we can successfully run all the tests in `test/Interpreter` and truly validate our solution works on the language level and its features.

2.5 Code Size Reduction

The goal of this thesis is to make it possible to use Swift on bare metal, specifically on devices with minimal resources such as those using the M-Profile architectures. We have already successfully done that in an emulator with artificially big program memory, and we have proven that our solution works by running a subset of tests on it. Therefore, the missing piece of the puzzle is reducing the code size sufficiently to run a program on a real device. If we look at the boards being regularly used in the segment, the highest program memory size that is commonly available is two megabytes[4]. In the following paragraphs, we aim at reducing the code size sufficiently to fit the two megabytes and still have some extra space for a reasonable program.

We begin with a program having only one statement `print("Hello World")`. The way it is compiled is similar to the minimal example we ran in QEMU (Section 2.3.2 on page 20). The only difference is that we link our Semihosting library implemented in the previous section, in order to have the `Swift.print` function print the string to the standard output of QEMU. The program-memory size requirements of the program can be inspected by running

```
> arm-none-eabi-size -B build/main
   text    data     bss     dec     hex filename
5586752  168732  121588 5877072  59ad50 build/main
```

It is seen that it currently requires 5.7 megabytes of program memory. The Swift toolchain we used for the compilation was built in a *Debug* mode, which turns off some basic optimizations, making the Swift standard library bigger than necessary. We intentionally start with such settings to make the following list of things we considered concerning code-size complete.

To confirm the fact that, as expected, most of the code size comes from the Swift standard library and runtime, we used the `-Map` flag of the `arm-none-eabi` linker. This command-line flag makes the linker emit a map file documenting the link process and lists useful information such as what object files were included in the final product. The map file is quite verbose, so we used a LinkerMapViz[5] tool to visualize the content of the executable. The visualization is shown in Figure 21 and confirms that the majority of the executable is composed of the Swift standard library. Therefore, unless otherwise stated, the rest of this section will focus on the build process of the Swift standard library.

2.5.1 Optimization Passes

The compiler has built-in optimization passes that can be controlled via the command line. By default, none are enabled (which is also true for the Swift

[4] `https://os.mbed.com/platforms/`

[5] `https://github.com/PromyLOPh/linkermapviz`

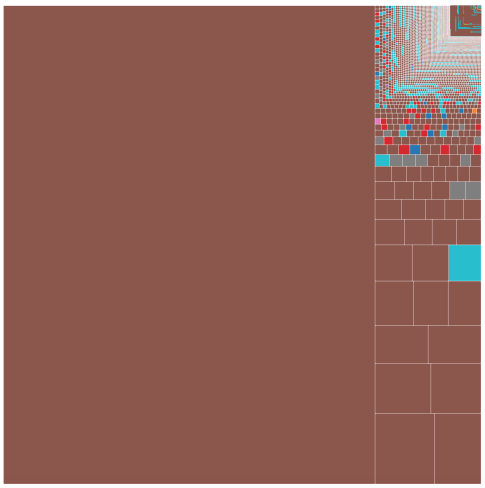

Figure 21: Visualization of sections assembling the executable's `.text` segment for initial version without code-size optimizations.
(`libswiftCore.a` (97 %), `libstdc++.a` (0.02 %), `libc.a` (0.01 %), . . .)

Figure 22: Code-size comparison for the full Swift standard library when built with different optimization settings.

standard library in our case). In `Release` builds of the Swift toolchain, the standard library is usually built with the `-O` flag, enabling the default set of optimization passes. This set of passes is designed to maximize the speed of execution with the possible trade-off of increased code size. The compiler provides an alternative set of optimization passes, that can be enabled with the `-Osize` flag, optimizing the executable for code size. The `-Osize` flag is expected to reduce code size somewhere between 5 % to 30 % while keeping the performance hit usually below five percent[12]. Based on our measurements, the use of `-Osize` does not affect the code-size as expected, decreasing it by only three percents (about 100 kilobytes). However, even such a reduction is notable; hence we made the `-Osize` flag the default when compiling the standard library for bare metal.

Linking the *HelloWorld* application against a standard library built with the `-Osize` flag reduced the program-memory requirements by 35 % down to 3.7 megabytes.

2.5.2 Disabling C++ Features

Even though the binary is mainly composed of a compiled Swift code, it incorporates some C++ code, which makes the compiler emit data used by the C++ runtime for stack unwinding and type identification. As mentioned in Section 1.2 on page 8, the runtime does not use those features and they can be safely disabled. That is possible by passing the `-fno-rtti` and `-fno-exceptions` flags to Clang.

By passing those flags to Clang, the compilation of the runtime is affected. However, those C++ runtime metadata can come from other places, such as the C++ standard library we are linking the program against. The reason they are included in the final executable is that the linker script explicitly tells the linker to do so. The C++ metadata for stack unwinding are stored in sections with names starting with `.ARM.extab` and `.ARM.exidx`. Thus by removing the code placing them in the `FLASH` section (program memory) and adding

```
__exidx_start = .;
__exidx_end = .;

/DISCARD/ : {
        *(.ARM.exidx*)
        *(.ARM.extab*)
}
```

the linker can be instructed to discard all those sections from the final executable.

By disabling the C++ features and excluding the C++ metadata from the executable, we were able to reduce the code size by 253 kilobytes down to 3.3 megabytes.

2.5.3 Using The Nano Version Of Newlib

The GNU ARM Embedded Toolchain comes with the Newlib C standard library. It also comes with a drop-in replacement of the standard Newlib implementation with another one, called *nano*, which is highly optimized for code size. It replaces complex algorithms focusing on the performance of the library with simpler ones reducing its size. Also, it removes features that are rarely used on bare-metal systems, such as advanced string formatting.

Using the Newlib-nano library with the minimal HelloWorld application decreased its size by another 82 kilobytes.

2.5.4 Disabling Assertions

What other platforms already do for `Release` builds in order to decrease code size and execution time of the standard library is to disable *assertions* and *runtime function counters*. The latter is a feature of the Swift runtime used primarily in tests, therefore can be safely disabled. The former, assertions, are function calls like `assert(arg > 0, "arg must be a positive integer")` or `precondition(...)`. By disabling the Swift conditional compilation flag `INTERNAL_CHECKS_ENABLED`, the `assert` calls are fully removed; `precondition` calls only abort execution if their condition evaluates to false and non-negligible part of the standard library is simplified.

In the case of the bare-metal platform, disabling the features mentioned above resulted in a significant decrease in code size. The HelloWorld application now requires 2.8 megabytes, a decrease of more than 600 kilobytes when compared to the previous code size.

2.5.5 Reflection Metadata

Swift has built-in support for reflection, making it possible to get a structured view of an object at runtime and inspect its stored properties. Because of Swift's strong focus on compile-time validation, this feature is typically used for debugging or logging purposes only. Reflection requires the compiler to emit *reflection metadata* for every type, therefore disabling it can decrease the program's code size. That is possible by passing the `-disable-reflection-metadata` flag to the Swift Frontend. Also, disabling the reflection metadata for the Swift standard library does not affect their generation for user-compiled code, so it is still possible to use reflection with user-defined types.

Building the standard library with reflection metadata disabled reduced the size of the HelloWorld application by 51 kilobytes.

2.5.6 Elimination Of Unused Sections

The optimizations thus far described focus on reducing the code size uniformly across the produced executable. However, considering our program only prints

"Hello World" to the console and exits, it is clear that the problem is primarily the fact that the executable contains plenty of unreachable code.

The elimination of dead code can be handled on multiple layers. Compilers usually analyze the single compilation unit being worked on, and if it is possible to determine that some code is inaccessible both from outside and inside the unit, it is eliminated before an object file is produced. The Swift compiler supports compiling a whole module as a single compilation unit; therefore it can eliminate dead code within an individual module. However, a minimal Swift program involves two modules – the standard library and the program's module itself. As the compiler compiles each module separately, it can not eliminate any code from the standard library itself, because the program that it will be linked to can use none of the libraries functions or all of them. Thus the only place to eliminate the unused code from the standard library is at link time when the program's module is being statically linked with the standard library.

The linker has a more low-level view of the code than the compiler and, in a way, a much simpler one. Code and data are organized in sections and dependencies between those sections are mostly represented by relocations (e.g. function calls). In the case of the HelloWorld application, the linker at the beginning includes the section containing the `main` function. This section includes relocations referencing a symbol for the `Swift.print` function, so the linker has to include the section from the standard library defining this symbol. Recursively, the linker loads increasingly more object files and their sections, to resolve dependencies of the already included ones.

By default, the linker includes all sections of loaded object files (until the linker script explicitly discards them). It includes even those sections that are not referenced from some required section. To make the linker discard those, it must be invoked with the `--gc-sections` flag. This flag makes the linker include only the sections that are (transitively) dependent on an explicitly included section. However, using this flag in our case reduced the code by 21 kilobytes only (so the application still requires 2.7 megabytes of program memory).

The reason for `--gc-sections` having such a small effect can be seen by inspecting the Swift standard library:

```
> arm-none-eabi-readelf libswiftCore.a -S | head
File: libswiftCore.a(Swift.o)
There are 49 section headers, starting at offset 0x127a5dc:

Section Headers:
  [Nr] Name              Type            Addr     Off     Size
  [ 0]                   NULL            00000000 000000  000000
  [ 1] .strtab           STRTAB          00000000 1116264 164377
  [ 2] .text             PROGBITS        00000000 000034  1e48ae
```

```
[ 3] .rel.text     REL        00000000 b7e7c4  067e58
```

The library contains a single `.text` section with all the code, which is almost two megabytes in size. Therefore, by depending on a single function located in the `.text` section, the linker has to add the whole section to the final executable. When using the `--gc-section` with C/C++ code, the compiler is usually invoked with two other flags – `-ffunction-sections` and `-fdata-sections`. Those flags make the compiler emit functions and data to separate sections (e.g. one function per section), increasing the granularity of the object file and allowing the linker to remove more unused code. However, the Swift compiler does not support those flags.

Splitting Function And Data Sections For Swift Code

The fact that the Swift compiler does not have a parallel to the `-ffunction--sections/-fdata-sections` flags is not surprising, as Swift is still a young language and has not yet been used on platforms with such significant memory constraints. We decided to implement those flags to reduce the code size further. The compiler's backend, LLVM, has direct support for those features, so we needed to add command-line options enabling them only. The options added are spelled `-function-sections` and `-data-sections`, and must be passed to the Swift frontend. Hence, when invoking the Swift driver (`swiftc`), the usage is:

```
swiftc -Xfrontend -function-sections -Xfrontend -data-sections
```

When we compiled the Swift standard library with the new command line options and linked the program with `--gc-sections`, the code size decreased by almost a megabyte – ending up at 1.6 megabytes.

2.5.7 A Tool For Object File Analysis

Even though 1.6 megabytes is still a lot for a "HelloWorld" application, it finally fits the memory constraints of commonly available M-Profile devices. Nevertheless, we decided to continue in our attempts to reduce the code size.

At this point, the biggest difficulty was understanding the content of the object files and tracing why a specific section was included in the final executable. It is possible to inspect the object files themselves by using tools such as `arm-none-eabi-nm` or `arm-none-eabi-readelf`, but their output is very verbose for such a big object file like the Swift standard library, making it very hard to get an overall picture. Another option is to use the `-Map` flag and make the linker output a linker-map file documenting the linking process. However, the linker-map does not say why a particular section was included, only giving the name of the object file from which the section was referenced. In our case, that is not helpful, as the Swift standard library is compiled into

a single object file and we want to be able to trace the dependency chains within that file.

We searched for tools able to analyze ELF object files or the linking process and found a number of them – for example *LinkerMapViz* (we presented its output in Figure 21 on page 27) or *bloaty*[6]. provided information on what sections were included in the executable, without answering the question of *why* they were included, nor giving a more in-depth insight into the dependencies between the sections.

To gain such knowledge, we decided to create a new tool. The goal was to get a graph of a linked executable, where each vertex would represent a section and an edge a dependency between two sections. We implemented this tool in Python using the *jupyter notebook* to allow easy further processing of the graph. The steps the tool performs in order to create the graph are as follows:

1. It parses the command-line invocation of the linker to know what object files (and archives) were involved in the linking process.

2. It unpacks all archives (`.a` files) for easier further processing.

3. Then it analyzes all the object files using the `arm-none-eabi-*` command-line tools, creating in-memory indexes of all the symbols, sections, COMDAT groups, and relocations.

4. The next step is to manually set the root section(s) of the dependency graph we want to create.

5. Then the algorithm starts, adding dependent sections to the graph, documenting the process, effectively simulating the linking process of a real linker.

6. Finally, we are left with a graph for further processing and visualization. An example of such a graph illustrating the HelloWorld application in the current state is shown in Figure 23.

The jupyter notebook with the tool is located at `dominator-symbol/ObjectDependencies.ipynb`.

2.5.8 Analyzing The Object Files

By inspection of the dependency graph depicted in Figure 23 we can immediately observe that, even though we have turned on the section splitting with the newly implemented `-function-sections`/`-data-sections` flags, the executable is still composed of one large and unbreakable cluster of sections.

Finding out what is holding such a cluster together is one of the primary reasons we developed the tool for object file analysis. As the dependency

[6] `https://github.com/google/bloaty`

Figure 23: An example of a dependency graph generated by our tool for analysis of the linking process. It shows the HelloWorld application. Nodes represent sections and edges dependencies between them. Size of a node represents the relative size of the section.

graph and all the metadata are available within a jupyter notebook, it is easy to examine them further – in this case, by transforming them to a *Dominator Tree*. The dominator tree helps us identify the symbols, that might be small themselves, but introduce a massive chain of dependencies. To explain this more formally, let us start with the definition of dominators:

Definition. "Dominators are defined in a directed graph with respect to a source vertex S. Formally, a node u is said to dominate a node w w.r.t source vertex S if all the paths from S to w in the graph must pass through node u."[13]

In other words, a section dominates another section, if the latter section is a dependency of the former and discarding the former section subsequently discards the latter section also. The dominator tree than builds on the *dominator* property of two nodes in the original graph – a node is an ancestor of another node if the former dominates it.

Listing 10 shows part of the dominator tree in textual form for the graph depicted in Figure 23. By quick examination of the tree, we can easily identify a few code-size related issues. However, the major one is visible on the fourth

```
> print_dependency_tree(dg, dg.gp.root, depth=3,
                        above_size=10_000, demangle=None)
1601414 linker-script(root)  (0B,#0)
  1174790 .rodata(libswiftCore.a)  (57860B,#2)
    15028   .data.rel.ro.$ss6SIMD16VyxGs28CustomDebugStringConverti...
    14532   .data.rel.ro.$ss6SIMD64VyxGs28CustomDebugStringConverti...
    14454   .data.$sSdSBsWp(libswiftCore.a)  (72B,#13638)
    12894   .data.$ss5Int64VSzsWp(libswiftCore.a)  (148B,#14916)
    12860   .text.$sSBsE8_convert4fromx5value_Sb5exacttqd___tSBRd__...
    11724   .data.rel.ro.$ss5SIMD8VyxGs28CustomDebugStringConvertib...
    11612   .data.rel.ro.$ss7UnicodeO6ScalarVs28CustomDebugStringCo...
    11416   .data.$ss6UInt64VSzsWp(libswiftCore.a)  (148B,#14777)
    ... (skipped 10 lines)
  102583  .text._ZN5swift15nameForMetadataB5cxx11EPKNS_14TargetMeta...
    56184   .text._ZN5swift8Demangle12nodeToStringB5cxx11EPNS0_4Nod...
    43908   .text._ZN5swift33_swift_buildDemanglingForMetadataEPKNS...
  42725   swift5_typeref(libswiftCore.a)  (19051B,#2614)
  21167   .rodata.str1.16(libswiftCore.a)  (21167B,#2772)
  11505   .text.fprintf(libc.a)  (40B,#2362)
    11465   .text._vfprintf_r(libc.a)  (7020B,#2363)
  10328   .text.swift_dynamicCast(libswiftCore.a)  (108B,#5588)
    10152   .text._ZL21swift_dynamicCastImplPN5swift11OpaqueValueES...
```

Listing 10: Part of a dominator tree of the graph depicted in Figure 23. Each line represents a single node, the root node is on the third line. Each line shows: *size including its dependencies*, *section name (object file name)*, and *(section size, section identifier)*.

line: the section `.rodata` from `libswiftCore.a` dominates the sections that have 1.174 megabytes in total. In other words – just by including the single `.rodata` section, which itself has only around 60 kilobytes, we bring more than a megabyte of data into the executable.

We can use the original dependency graph and simple Depth-First Search to trace the inclusion of this section. One of the possible paths from the root section to `.rodata` is:

```
Starting in section 'linker-script'
-> symbolic dependency KEEP(swift5_type_metadata) resolved with
↪  swift5_type_metadata(libswiftCore.a)
-> relocation of '$sSo10HeapObjectVMn' found in
↪  .rodata(libswiftCore.a)
```

The output is interpreted as follows – the linker script includes the `swift5_type_metadata` section with the type metadata required by the runtime. That section includes a relocation of symbol `$sSo10HeapObjectVMn`,

which is resolved in the `.rodata` section. By demangling the symbol `$sSo10HeapObjectVMn`, we find that it is a nominal type descriptor. That is expected – the section `swift5_type_metadata` should include pointers to structures describing all the types defined. However, this creates a few problems for us.

As there is only one section `swift5_type_metadata` in `libswiftCore.a`, it must contain pointers to all the types defined in the Swift standard library. The type metadata structures then contain pointers to the type's methods and all its other dependencies. Therefore, by including the metadata, we can expect to bring all the bits of the type's implementation into the executable. This graph of dependencies is effectively making a single cluster out of the standard library. In order to break this cluster, we have to split the `.rodata` section into smaller sections, so that every type metadata structure is stored separately. Also, we have to break the `swift5_type_metadata` section and change the linking process, so that only the types that are referenced in the program are included.

By further analysis of the dependency graph, we can identify that the other two sections with runtime metadata – `swift5_protocols` and `swift5_protocol_conformances` – have a similar problem to the one described above.

2.5.9 Splitting Metadata Sections

In order to loosen the dependency graph of the standard library, we changed how the compiler places all the metadata into sections.

- All the runtime metadata structures, such as protocol descriptors, nominal type descriptors or protocol conformance descriptors the compiler now puts into separate sections. The name of the section is usually `.rodata.<suffix>` where `<suffix>` is a mangled name of the entity stored.

- The sections named `swift5_*` that are used for runtime registration of all the types and protocols are now also split. The pattern is as described above – for example, a record of protocol conformance is put into a separate section named `swift5_protocol_conformances.<suffix>` where `<suffix>` is the mangled name of the protocol conformance descriptor.

As we do not want to change the standard behavior of the compiler, splitting the metadata sections is disabled by default. It can be enabled with a new `-metadata-sections` frontend flag, that can be used similarly to the already introduced `-function-sections` and `-data-sections`.

To make use of the Swift standard library compiled with the `-metadata-sections` flag, it is necessary to change the linker script of the HelloWorld application. Currently, the script includes the metadata sections as follows:

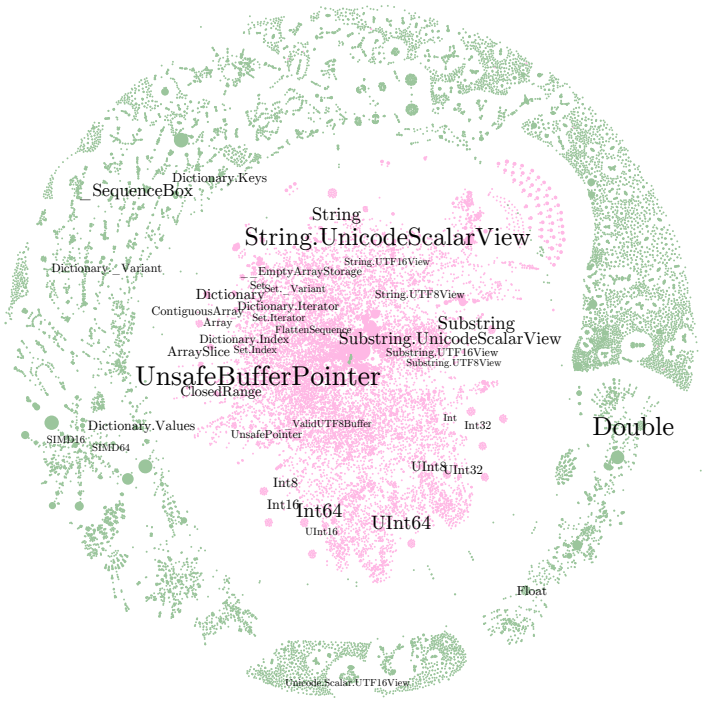

Figure 24: Dependency graph visualization of the HelloWorld application with split sections. Green sections are the ones discarded by the linker thanks to the implemented section splitting.

```
1 __start_swift5_type_metadata = .;
2 KEEP(*(swift5_type_metadata))
3 __stop_swift5_type_metadata = .;
```

Firstly, we have to reflect that the type metadata can be stored in sections with a suffix. Secondly, the `KEEP(...)` directive forces the linker to include all sections with the pattern in the parentheses. As previously described, we want to keep metadata of the types that are referenced by our code only. Therefore, the second line has to be changed to `*(swift5_type_metadata*)`.

With the above changes, the code-size of the HelloWorld application drops significantly. However, the linker now discards too many sections, and most applications will not work correctly. The first problem is that nothing is referencing the `swift5_*` sections and they all get discarded. When a program instantiates some type, then the code references the type's descriptor (located in `.rodata.<suffix>` section). Therefore, the linker does not discard the descriptor. However, the `swift5_type_metadata.<suffix>` section contains a pointer to the descriptor only, so there is no reason for the linker to keep it in place. The situation is similar with protocols – when a specific protocol is used in code, the code then references its protocol descriptor. However, nothing

references the section `swift5_protocols.<suffix>` with the protocol record. To solve this, we made the compiler put the pairs of a `swift5_*.<suffix>` section and a section with the corresponding descriptor into a COMDAT group. When some sections are in the same COMDAT group, and one of the sections is included by the linker, the linker also has to include all the other sections from the group. Therefore, if the linker includes some descriptor, it now also includes the proper record in a `swift5_*` section.

The last problem is that the linker discards all the *protocol conformance* records and their descriptors, because code potentially using protocol conformance does not necessarily create a reference to its descriptor. As the linker has limited ways of dealing with this, two possible solutions were attempted. Protocol conformance defines conformance of a type to some protocol. Therefore, we tried putting the protocol conformances into the same COMDAT group as the type's metadata are in, for which the protocol conformance is defined (meaning when a type is included, all its conformances are included too). In addition, we tried putting the protocol conformances into the same COMDAT group as the protocol is in. We decided to choose the former solution as it produced smaller executables.

The result of using the standard library compiled with the `-metadata-sections` flag is a reduction of the HelloWorld application's code size by four hundred kilobytes.

2.5.10 Summary

With the changes introduced in this section, we have reduced the code size of a minimal application to close to one megabyte. The primary obstacle in reducing it further is a tight coupling of different components of the standard library. This coupling is caused by the runtime metadata, where often inclusion of a single entity causes a chain reaction of the inclusion of another type's metadata, its witness tables, protocol conformances and so on. This behavior is well demonstrated in Figure 24, where it is seen that only linking a program printing some text to the standard output brings a significant part of the standard library to the final executable.

There are several ways of approaching such a problem. Link-time optimizations can be implemented, that would perform dead-code analysis similarly to what the compiler does on the compilation-unit level. Also, it would be possible to reduce the code size of bare-metal programs substantially by minor changes in the standard library (such as by removing parts that are big and unnecessary in respect to bare metal or by breaking problematic dependency chains). Alternatively, a light-weight version of the standard library can be implemented targeting bare metal. However, we decided against such changes in this initial phase, as we believe this should potentially be discussed with the Swift community.

At this point, the size of the application is no longer a problem, and we can proceed towards the goal of running Swift on devices with the M-Profile architectures.

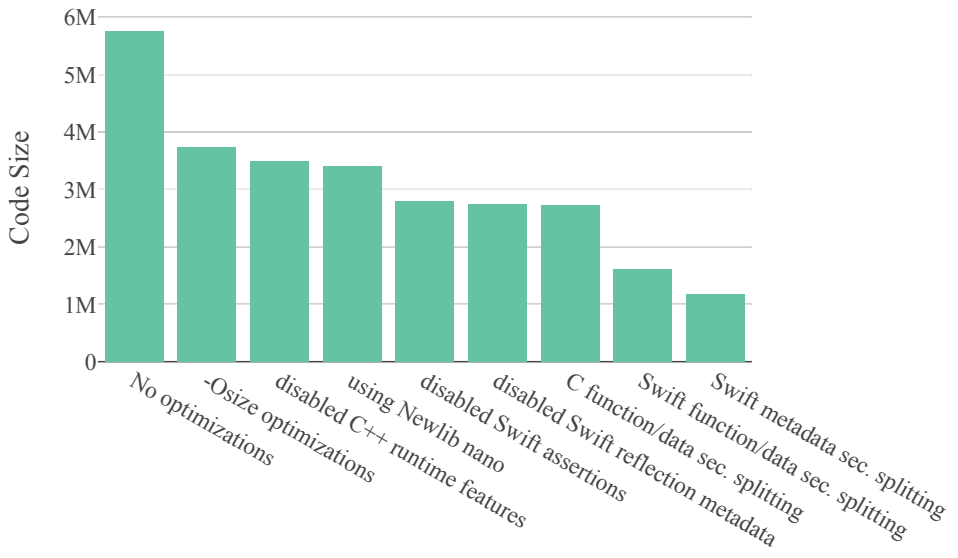

Figure 25: Comparison of the program's code size with the optimizations applied.

2.6 Swift Package Manager

At this point, the Swift toolchain is fully capable of producing an executable which can be run on bare metal. However, managing the build process of such an application would not be easy. Once more, let us remember the compilation of a minimal application from Section 2.3.2. It required many command line options to be specified, most of them related to the GNU ARM Embedded Toolchain or the targeted device. Also, the listing in Section 2.3 only showed the compilation of a single Swift file, not concerning how the C/C++ part of the application is built. The compilation of a real bare-metal application is going to be much more complex – the necessity to cooperate between C and Swift code is to be expected, producing multiple Swift modules and linking them together. Configuring the build process using Makefiles or some alternative build tool would be error-prone, and would require comprehensive knowledge of the compilers involved.

Fortunately, those problems are already solved in the Swift ecosystem with the Swift Package Manager – a tool addressing challenges such as compiling and linking Swift packages, managing dependencies, versioning or distribution of source code. The possibility of using the Swift Package Manager with the new bare-metal platform would undoubtedly improve the experience of building programs for this platform. This is the issue to be examined in the following section.

2.6.0.1 Adding Support For BareMetal

The Swift Package Manager is a separate project from the Swift compiler, and it is fully implemented in Swift. It is mostly target-platform agnostic; it just needs to be able to parse the new target triple (`thumb*-unknown-none-eabi`) and be aware of the new bare-metal platform so it can be used in package configurations. Adding support for those two things required straightforward changes in `src/swiftpm/Sources/Build/Triple.swift` and `src/swiftpm/Sources/PackageDescription4/SupportedPlatforms.swift`.

2.6.0.2 Cross-Compilation

By default, when building a package with the Swift Package Manager, the package is built for the host platform. However, we need the package to be cross-compiled for an embedded device instead. We can instruct the package manager to do so by using the `--destination <file>` command-line option. The file specified must be a JSON file containing a toolchain configuration that should be used to build the package. That includes information such as a path to the Swift compiler, extra C and C++ flags, the target triple or linker options. Therefore, we only need to create such a file for the target device and pass it to the Swift Package Manager to configure the cross-compilation.

Because the destination file can get quite big and the primary part of it contains search paths of the GNU ARM Embedded Toolchain, we created a simple Python script that automatically generates it for every supported architecture. The script is located in `projects/_Destinations` and by running `python generate.py` it generates files `thumbv7m.json` and `thumbv7em.json`. Cross-compiling a Swift package is then as easy as running

```
1 > swift build --destination thumbv7m.json
```

2.7 Running on Hardware

In this section, we are finally going to run an application on real hardware. The board we have selected for our experiments is NUCLEO-F439ZI[7]. It has good support on other platforms such as Mbed, Arduino or Micropython, thus allowing us later comparison with Swift for Embedded Systems. Also, the board is relatively well equipped and will not pose any limitations to further evaluation. The basic features of the board are:

- The onboard Micro-Controller Unit (MCU) is STMF439ZI[8]. It has an Arm Cortex-M4 Core using the ARMv7E-M architecture, and it has a built-in single-precision FPU.
- The MCU operates at frequencies up to 180 Mhz and has two megabytes of a FLASH memory and 256 kilobytes of SRAM.
- It has an extensive range of peripherals, such as up to 168 I/O ports, multiple I2C interfaces, UARTs, SPIs, CAN, Ethernet Controller or LCD-TFT Controller.
- Also, the board has onboard debugger and programmer (STLink).

2.7.1 Assembling The Application

The HelloWorld application is again going to be the first to be run. To make it as simple as possible, it will not use serial port peripheral (UART) to print the text, but it will use the Semihosting interface (as we used when running inside QEMU). However, this time, we will use the Swift Package Manager to build the application. A Swift package is a directory with source files and a `Package.swift` manifest file in its root. The manifest file can define two types of products to be built – a library or an executable. Those products are then defined in terms of targets (Swift modules), which can be either defined within the same package or be a library from another package. The structure of the packages involved in the HelloWorld application is:

***package* Crt0**
: This package provides a library named `crt0` implementing the C runtime entry function.

***package* STM32F4**
: This package contains the necessities related to the STM32F4 family of microcontrollers. Currently, it provides a `STM32F439ZIStartup` library with the startup code required for the STM32F439ZI microcontroller.

[7] `https://www.st.com/en/evaluation-tools/nucleo-f439zi.html`
[8] `https://www.st.com/en/microcontrollers-microprocessors/stm32f439zi.html`

***package* Semihosting**
: A drop-in replacement for the `nosys` library defined in the GNU ARM Embedded Toolchain. It implements a system-related function with respect to the Semihosting interface.

***package* HelloWorld**
: This is the main package of the application. It defines a single product of type *executable* incorporating four targets: a) the library `STM32F439ZIStartup` with the startup code for the device, b) the `Crt0` library with the C runtime entry function, c) the `Semihosting` library from the package above and d) the module with the program itself.

The source files of the program mostly remained the same as in Section 2.3 when building the application for QEMU, with the exception of the startup code, because the STM32F439ZI MCU requires a different vector table (as it has a Cortex-M4 core, instead of Cortex-M3 which was emulated by QEMU).

The directory structure for all the packages is depicted in Listing 26. The linker script had to be changed to reflect the memory layout of the MCU. As the goal of the `STM32F4` package is to provide everything related to running Swift on the STM32F4 family of devices, we located the linker script within this package (under `Linker/stm32f439.ld`). However, there is a drawback to locating it under the STM32F4 package. The only way to specify the linker script for the linker is from the manifest file of the executable that is being built (in this case `HelloWorld/Package.swift`), and it can not be specified from a manifest file of a library it uses. Therefore, there is a small workaround needed in the manifest file of the HelloWorld application to reach the linker script (line 5 in Listing 11).

```
/projects
├── Semihosting
│   ├── Package.swift
│   └── Sources
│       └── semihosting
│           ├── include
│           └── semihosting.c
├── Crt0
│   ├── Package.swift
│   └── Sources
│       └── crt0
│           ├── include
│           └── crt0.c
├── STM32F4
│   ├── Package.swift
│   ├── Sources
│   │   └── STM32F439ZIStartup
│   │       ├── include
│   │       ├── startup.c
│   │       └── system.c
│   └── Linker
│       └── stm32f439.ld
└── HelloWorld
    ├── Package.swift
    └── Sources
        └── HelloWorld
            └── main.swift
```

Figure 26: Directory structure of the packages used for the HelloWorld application (simplified).

Because the Swift Package Manager automatically detects that a defined target contains C code and handles it appropriately, there is no significant difference

```
// swift-tools-version:5.0
import PackageDescription
import Foundation

let linkerScript = FileManager.default.currentDirectoryPath
                    + "/../STM32F4/Linker/stm32f439.ld"

let package = Package(
    name: "HelloWorld",
    products: [
        .executable(name: "HelloWorld", targets: ["HelloWorld"]),
    ], dependencies: [
        .package(path: "../Semihosting"),
        .package(path: "../STM32F4"),
    ], targets: [
        .target(
            name: "HelloWorld",
            dependencies: ["STM32F439ZIStartup", "Semihosting"],
            linkerSettings: [
                .unsafeFlags([
                    "-Xlinker", "--gc-sections",
                    "-Xlinker", "-T",
                    "-Xlinker", linkerScript,
                ])
            ]
        )
    ]
)
```

Listing 11: The manifest file of the HelloWorld package targeting STM32F439ZI.

between manifest files of the libraries containing C code and the root HelloWorld package containing Swift code only. Therefore, we present the main manifest file of the HelloWorld package only (see Listing 11).

2.7.2 Building And Running The Application

To build the application, we need to enter the `projects/HelloWorld` directory and invoke the package manager:

```
> swift build --destination ../_Destinations/thumbv7em.json
[1/7] Compiling CSTM32F439ZIStartup system.c
[2/7] Compiling Semihosting semihosting.c
[3/7] Compiling Crt0 crt0.c
[4/7] Compiling CSTM32F439ZIStartup startup.c
```

```
[5/7] Archiving
↪  ./.build/thumbv7em-unknown-none-eabi/debug/libSemihosting.a
[6/7] Compiling Swift Module 'HelloWorld' (1 sources)
[7/7] Linking ./.build/thumbv7em-unknown-none-eabi/debug/HelloWorld
```

The linked executable is now located in the build directory. However, it is in an ELF format. Before flashing it to the memory of the device, we have to convert it to a format the flashing utility can handle:

```
> arm-none-eabi-objcopy -O ihex
↪  .build/thumbv7em-unknown-none-eabi/debug/HelloWorld{,.hex}
```

Then we can flash it to the device and start the debugger:

```
> st-flash --format ihex write
↪  .build/thumbv7em-unknown-none-eabi/debug/HelloWorld.hex
...
> arm-none-eabi-gdb -ex 'target remote-extended :4242' -ex 'continue'
...
```

In another terminal, we have to start a GDB server that the debugger will connect to, and we can immediately see the application output:

```
> st-util --semihosting 2>/dev/null
st-util 1.5.1
Hello, world
```

It works! We have successfully run a Swift program on an embedded device with no operating system involved.

2.7.3 Hardware Abstraction Layer

The application ran in the previous section was simple, and its Swift code did not interact with the hardware in any way. Having an embedded application with no interaction with hardware peripherals is pointless; therefore, in this section, we demonstrate how a basic Hardware Abstraction Layer can be built.

The goal of a Hardware Abstraction Layer (HAL) is to provide an application programming interface (API) for interaction with the hardware peripherals of a device. Realization of such abstraction is possible on multiple levels: a) it can abstract communication with a peripheral on a specific device (for example by providing access to its registers); hence the client of such API still has to be aware of the particular device, b) it can abstract the interaction with peripherals for a whole family of devices, where the configuration might still be device-specific, but general interaction with the peripherals is shared across the device family or c) it can provide a general abstraction of the peripherals and its typical features across all devices.

Embedded devices are usually provided with a HAL library written in C, common for a family of devices. Such a library is then configured and compiled

for a specific device using C macros. In the case of the STM32F439ZI device we use in this thesis, STMicroelectronics provides it with the *STM32F4 HAL* library. This library provides high-level access to peripherals like I2C, SPI, GPIO or clock configuration.

An abstraction of HAL shared for a family of devices is often a good compromise between having good access to the specifics of a device and being able to share code between similar devices when developing an embedded application. On the other side, when writing a general piece of software – for example, a driver for an external peripheral communicating using I2C – it is not desirable to have such a driver tied to a specific device family as it might only require a generic I2C interface. For those reasons, we have decided to model the HAL in Swift For Embedded Systems as follows:

- The `Hardware`[9] package/library provides a high-level abstraction of hardware peripherals and their API. It is expected to be used by code shared across applications. It does not provide API for configuration of those peripherals, as that is expected to be handled in each application using a lower-level API specific for the device family used.

- A device or device-family-specific HAL library, where types representing common hardware peripherals are expected to implement the general interface defined by the `Hardware` library.

The pattern for using such a hardware access layer is 1) the application imports the library specific for the device/device family, 2) it instantiates and configures necessary peripherals in a device-specific manner and 3) it either uses the created instances directly or indirectly using the protocols defined in the `Hardware` library (for example when using from a general third-party library requiring "some I2C" only).

2.7.3.1 The Hardware Library

The `Hardware` library is defined within a `Hardware` package (via Swift Package Manager). It defines common protocols to be implemented by types providing access to hardware peripherals. To keep things simple, we start with its minimal implementation defining an interface for GPIO depicted in Listing 12.

2.7.3.2 The STM32F4 Library

In Section 2.7.1, we have already presented the `STM32F4` package containing the Startup library with code required for proper device startup. The next

[9] We also considered the name "HAL", as that is commonly used for such a library. However, we believe "Hardware" is less cryptic for people new to embedded development and having the "Abstraction Layer" in the name is just stating the obvious.

```
1  public enum PinState {
2      case low
3      case high
4  }
5
6  public protocol DigitalIn {
7      func get() -> PinState
8  }
9
10 public protocol DigitalOut {
11     func set(_ value: PinState)
12 }
13
14 public protocol ToggleableDigitalOut : DigitalOut {
15     func toggle()
16 }
17
18 public protocol StatefulDigitalOut : DigitalOut {
19     func get() -> PinState
20 }
```

Listing 12: Universal GPIO interface as defined in the `Hardware` library in `projects/Hardware/Sources/gpio.swift`.

thing is to extend this package with a new `STM32F4` library providing the hardware abstraction layer. As mentioned above, STMicroelectronics provides a C HAL library. Therefore, we decided to implement the Swift version of the library as a wrapper around the C version. Thus, the manifest file of the `STM32F4` package now declares two more libraries:

`CSTM32F4` library
: This is the C version of the HAL library as provided by STMicroelectronics. The Swift Package Manager automatically compiles the C source code and builds a module importable from Swift.

`STM32F4` library
: This is the HAL library to be used by user code. It provides high-level access to the hardware peripherals implemented using the `CSTM32F4` library.

2.7.3.3 Using the HAL library

With the libraries described above, we can create a simple application interacting with some hardware peripheral. We have made an application called `Blinky` which blinks with a blue LED built-in on the NUCLEO board we

```
1  import STM32F4
2
3  print("Welcome to the Blinky Demo")
4
5  do {
6      // Initialize the Hardware
7      let device = try STM32F4()
8      // Configure the pin
9      let blue = device.gpio.pin(peripheral: .B, number: 7,
10                                mode: .output)
11     blue.set(.high)
12
13     // Toggle the led periodically
14     while (true) {
15         device.delay(ms: 500)
16         blue.toggle()
17     }
18 } catch {
19     print("failure: \(error)")
20 }
```

Listing 13: A demo application using the STM32F4 library to interact with hardware.

are using. The source code is listed in Figure 13. The application does the following:

1. Initializes an instance of `STM32F4` class representing the device. The initialization is decorated with a **`try`** because the underlying implementation of the initializer calls a `HAL_Init()` function of the underlying C HAL library. If that function returns an error, it is translated to a `STM32F4Error` type which is later handled on line 19.

2. On line 9, the pin to which the LED is connected is configured as an output and on the next line, set to high polarity.

3. The program then goes to an infinite loop, where it always waits for five hundred milliseconds and then toggles the LED pin's polarity.

The source code for this application can be found under the `projects/Blinky` directory. It can be built and uploaded to the device, using the same commands as presented in the previous section.

Chapter 3

Evaluation

In the previous chapter, the process of bringing Swift to the world of low-performance embedded devices was described. In the following paragraphs, we ensure that the presented solution has the properties a user might expect from a language like Swift (such as high performance), while identifying some of the possible challenges of this solution.

The board used for this evaluation is again the NUCLEO-F439ZI from the previous chapter. As we believe the use of Swift for embedded systems in the near future is in the segment of hobbyist and makers building systems for the Internet of Things, the focus is on comparing our solution to two popular platforms from this segment. The first one is Arduino (C/C++), representing the mainstream in this segment offering high performance. The second one is MicroPython – lean and efficient implementation of Python 3 targeting embedded devices. Even though the comparison with MicroPython might seem inappropriate (one being a compiled language, the second interpreted), they are both high-level languages representing an alternative to the languages from the C family currently dominating this segment.

3.1 Performance

3.1.1 Bit Banging

The first test is simple. It switches the polarity of a pin as fast as possible, giving us a notion of the overhead required to do such an operation. This might seem an artificially simple test; however, based on the overhead we get an idea whether it is viable to, for example, implement a software-driven version of I2C and achieve reasonable throughput.

During every test, the board was configured to run at maximum speed (180Mhz), and both the Arduino and the Swift versions were built with performance optimizations turned on (`-O2` flag and `-O` flag respectively). For the Arduino version shown in Listing 18, it might seem redundant to create the `while` loop on line 4, when there is already the `loop` function which is periodically called (therefore, the `digitalWrite` lines can be placed there). However, that would create a discontinuous signal because of the overhead from calls to the `loop` function[10]. Similarly, the body of the `test` function in the MicroPython case could be a top-level code in the file. However, that decreases the performance significantly[14].

The measured frequencies of the generated signals are shown in the following table.

Swift	3.9 MHz
Arduino	2.9 MHz
MicroPython	0.089 MHz

It is interesting to compare the results of the Arduino and Swift versions. Even though both the Arduino API and Swift API use the STM32F4 C HAL library provided by STMicroelectronics, the Swift version is faster. The C compiler did not inline the `digitalWrite` function (it is not defined as `inline`). This `digitalWrite` function must translate the given arguments to different ones required by the STM32F4 HAL library and then call the underlying function from that library. On the other side, the Swift compiler was able to inline the `.set()` method call and precalculate all the required parameters at compile-time, therefore effectively replacing the call with a call to the underlying C HAL library only.

[10] For pedantic readers – even the while loop creates an uneven signal, as there must be an extra instruction to jump back to the start of its body (and possibly evaluation of the condition). This is also a reason why all the programs include two pairs of "set high"/"set low" statements – to get a longer period of continuous signal.

```swift
import STM32F4

let device = try! STM32F4()
let pin = device.gpio.pin(peripheral: .B, number: 4, mode: .output)

while (true) {
    pin.set(.high)
    pin.set(.low)
    pin.set(.high)
    pin.set(.low)
}
```

Listing 14: Swift source code for the bit-banging performance test.

```c
void setup() {
    pinMode(PB4, OUTPUT);

    while (1) {
        digitalWrite(PB4, 1);
        digitalWrite(PB4, 0);
        digitalWrite(PB4, 1);
        digitalWrite(PB4, 0);
    }
}

void loop() { }
```

Listing 15: C source code using the Arduino platform for the bit-banging test.

```python
from machine import Pin

def test():
    pin = Pin('B4', Pin.OUT)
    while True:
        pin.value(1)
        pin.value(0)
        pin.value(1)
        pin.value(0)

test()
```

Listing 16: MicroPython version of the source code for the bit-banging test.

3.1.2 Response To Interrupts

The next test is to estimate the time it takes for the platform to respond to an interrupt. We have connected one pin of the board to a signal generator (of a square wave) and configured that pin to invoke a user-defined function when the pin's polarity changes. The function then toggles the output of another pin. Using an oscilloscope, we then measured the delay between those two signals. The results are shown in the following table.

Swift	1.8 µs
Arduino	0.7 µs
MicroPython	20.8 µs

Interpreted MicroPython is unsurprisingly still behind the two compiled languages. This time, C/Arduino was faster than Swift. We compared the assembly of the two programs, hoping to find a single reason for the latter being slightly slower. However, it is more a combination of several aspects of Swift – the additional time is mostly spent on memory management and extra safety checks.

In addition, it is important to mention that the Swift code was compiled with the `-enforce-exclusivity=none` flag. This flag disables runtime checks of exclusive access to variables as described in Section 1.1.3 on page 5. With the checks enabled, it takes the Swift program an extra microsecond to respond to the interrupt.

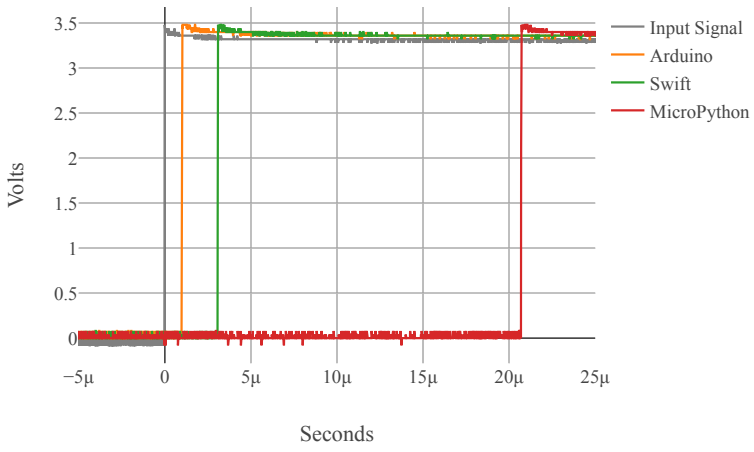

Figure 31: Measured data from an oscilloscope – the microcontroller responding to an input signal (interrupt) by toggling another pin's polarity.

```swift
import STM32F4

let device = try! STM32F4()
let output = device.gpio.pin(peripheral: .B, number: 7, mode: .output)
let input = device.gpio.pin(peripheral: .B, number: 4)

input.configure(.interrupt(edge: [.rising, .falling], pull: .down,
↪  handler: {
    output.toggle()
}))

while (true) {}
```

Listing 17: Swift source code testing response time to interrupts.

```c
void setup() {
    pinMode(PB7, OUTPUT);
    pinMode(PB4, INPUT_PULLDOWN);
    attachInterrupt(PB4, onInterrupt, CHANGE);
}

void onInterrupt() {
    digitalToggle(PB7);
}

void loop() { }
```

Listing 18: C source code using the Arduino platform testing response time to interrupts.

```python
from machine import Pin

def on_interrupt(_):
    output_pin.value(not output_pin.value())

output_pin = Pin('B7', Pin.OUT)
input_pin = Pin('B4', Pin.IN, Pin.PULL_DOWN)
input_pin.irq(on_interrupt)
```

Listing 19: MicroPython version of the source code testing response time to interrupts.

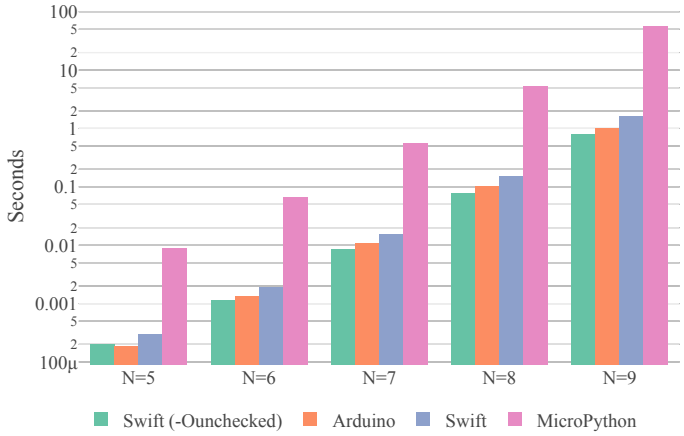

Figure 32: Results of the Fannkuch Benchmark.

3.1.3 Fannkuch Benchmark

With the last test evaluating execution speed, we decided to run a general algorithm not strictly related to embedded systems. The main requirement for running such a test is having three separate implementations of the algorithm, that are reasonably written concerning the language they are for, while still being similar enough to allow meaningful comparison.

Fortunately, there is a project called "The Computer Language Benchmarks Game,"[11] which defines a set of algorithms. People can submit their implementations of those algorithms in various languages to compete against others. As the project defines strict rules for the implementations with respect to how much they can differ, and a specific algorithm must be followed (not merely a solution implemented for a problem), it perfectly aligns with the goal of this section. The selection of the algorithm for this test was mostly based on the restrictions of the embedded system we want to run it on and the available implementations in the project (often, all the implementations required support for threads or arbitrary-precision arithmetics). The test selected is *Fannkuch Benchmark*, defining an algorithm to solve a combinatoric problem[15]. We then picked the best implementations for the languages we are interested in[12] and measured the execution time for different sizes of the problem (different N). The results are shown in Figure 32.

The results show that the Swift version performed slightly worse than (but

[11] `https://benchmarksgame-team.pages.debian.net/benchmarksgame`

[12] Arduino: fannkuchredux-gcc-1, Python: fannkuchredux-python3-6, Swift: fannkuchredux-swift-1

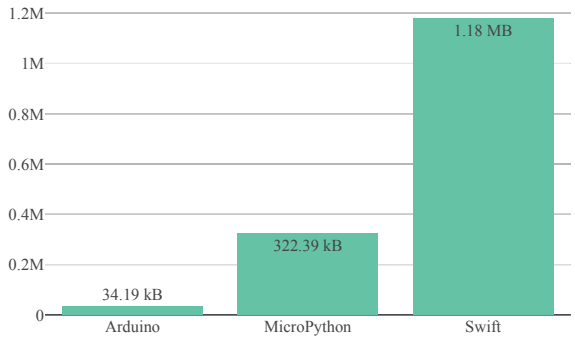

Figure 33: Program-memory requirements comparison of programs running the Fannkuch benchmark.

was still comparable to) the C version, while the MicroPython was consistently almost two orders of magnitude slower. The reason for Swift being slightly slower is the runtime checks it makes to achieve memory safety (e.g. out-of-bounds checks for array access or checks of overflows for basic arithmetic operations). It is possible to turn off all those checks, resulting in even lower execution times than those of the C version (see "Swift (-Ounchecked)" in Figure 32). However, turning this feature off contradicts one of the main goals of Swift (to be a safe language), and we present it here as a demonstration of the price for the runtime safety checks only.

The measured execution times for the C and Swift versions were stable; however, in the case of MicroPython, the times varied by up to a few dozens of milliseconds in the case of higher N because of the unpredictable nature of GC pauses.

3.2 Code Size

The code sizes of the programs generated by the platforms being compared in this chapter vary greatly. Comparison of program-memory requirements for the Fannkuch Benchmarks is shown in Figure 33.

The code size is lowest in the Arduino case. While it is only about 35 kilobytes, most of it is still not generated by our code. It would not be a problem to lower this even further (this result was obtained with default settings mostly, the only difference being using the `-O2` optimizations and linking against Newlib nano with support for string formatting). The user code with Fankuch benchmark required 344 bytes of program memory (without

dependencies in the C standard library).

Micropython required more than three hundred kilobytes as it has to include the interpreter and full standard library[13]. This represents a fixed cost, and a user code can be provided in two ways: 1) uploaded to the device as standard Python source code, requiring further compilation to bytecode, thus using more resources at runtime, or 2) as precompiled byte code. MicroPython is then able to execute this bytecode directly from FLASH memory, therefore using a minimum of resources of the device. The precompiled file with the Fannkuch Benchmark required 597 bytes only.

The same application written in Swift for Embedded Systems required more than one megabyte of program memory. The main reason is that the executable contains much unreachable code, as described in the Realization chapter (Section 2.5). The question here is whether this represents a fixed cost, or whether the application will grow substantially when a user code references more symbols from the standard library. Figure 34 shows code sizes for different applications we have written. It indicates that, even when an application uses different parts of the standard library, the code size grows proportionally to the used features. More importantly – there are no significant sudden increases in code size as a result of the use of some additional features. The BigSymbols application in Figure 34 uses large data types from the Swift standard library that are not included in the other applications (such as `Dictionary` or `Double`). Nevertheless, the application has grown by a few dozens of kilobytes only.

The code size of the compiled Fannkuch Benchmark written in Swift is 1416 kilobytes; therefore, two or four times bigger than the same code written for MicroPython or Arduino respectively.

3.3 Using Swift for Embedded Systems

The speed of execution and memory requirements are without a doubt important aspects of an embedded platform. However, other characteristics – often more subjective – significantly influence the final experience. In the following paragraphs, some of our findings are listed from using Swift on embedded systems.

3.3.0.1 Handling Interrupts

When an interrupt occurs, the microcontroller usually immediately stops the execution of the current code and starts executing adequate Interrupt Service Routine (ISR). When the ISR finishes, the control is given back, and execution continues from the point when the ISR occurred. This introduces the

[13] Meaning the standard library of MicroPython, which is a subset of the usual Python standard library.

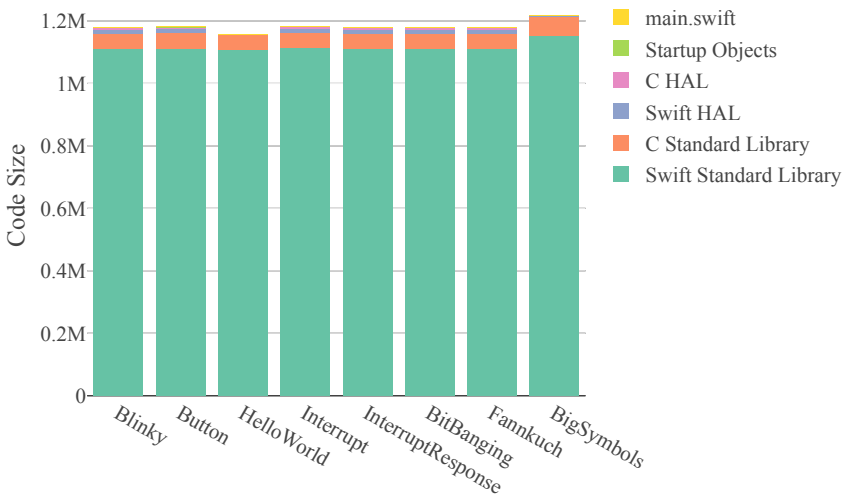

Figure 34: Code size of different Swift applications. The code size does change dramatically when additional features of the Swift standard library are used.

same problem as in a multithreaded environment. It is therefore required to appropriately handle access to shared resources between ISRs and the main routine.

In C, this means, for example, not using the `malloc` function (or other non-reentrant function) within an ISR. Swift, as a language, does not specify when an allocation of memory can take place. Currently, the worst must be expected – meaning any line of code can potentially be unsafe and access some shared resource. However, in practice, it is not hard to tell whether a simple code after optimizations involves a call to the Swift runtime. Therefore, with extra care, it is possible to use Swift within ISR.

An example of an ISR defined within the STM32F4 Swift library is

```
1 @_silgen_name("EXTI2_IRQHandler")
2 func EXTI2_IRQHandler() {
3     HAL_GPIO_EXTI_IRQHandler(1 << 2)
4 }
```

This ISR only delegates processing of the interrupt to the lower-level C HAL library. However, worth mentioning is the `@_silgen_name` attribute next to the function definition. It forces the compiler to give the symbol with this function the name specified in parentheses. This is required, as the startup code registers a symbol with this specific name to be the ISR, and the default

symbol name for the function would be Swift-specific, including information such as the module name or full signature of the function.

The `@_silgen_name` attribute is currently not a documented feature of Swift and will probably be replaced in the future with another spelling.

3.3.1 Volatile

In C, `volatile` is a type qualifier making the compiler aware that a variable with the type can change at any point in time. Let us consider the following C code:

```
1 volatile char* reg = (char*)0x0800000;
2 *reg = 'a';
3 *reg = 'b';
```

Without the `volatile` qualifier, the compiler can deduce that the code writes two times to the same memory location and can remove the first write. However, the write to the memory can have special semantics – for example, the memory can be mapped to a serial port peripheral and writing to it can send the character via the port. The `volatile` qualifier prevents such optimizations from happening.

Currently, Swift does not have any parallel to the `volatile` qualifier. Therefore, when accessing raw memory in order to control some hardware peripheral, a user has to be aware of the possible consequences of optimizations.

3.3.2 Package Manager

Full support for the Swift Package Manager is crucial in terms of usability, as it allows Swift for embedded systems to leverage all the tooling that comes with Swift. This is something other embedded platforms often struggle with. For example, there is no unified way to distribute C/C++ libraries, and each platform usually builds its own. Another problem is often having a good-quality IDE working with the embedded platform – Arduino comes with a very simplistic IDE (not having such features as auto-complete), and MicroPython does not work out-of-the-box with existing Python IDEs (the problem here being, for example, a separate package manager). Some of the consequences of support of the Swift Package Manager are:

- Seamless cooperability between Swift and C code. The Swift Package Manager automatically detects if a module contains C code and handles it without any further configuration required. This is extra important for an embedded system, as most of the code available is written in C.

- Easy-to-use management even of a very complex application involving many dependencies.
- Ability to use existing third-party libraries.
- The possibility of using existing tools for Swift – for example a Language Server Protocol Server[14] that is now being officially developed, extending existing IDEs with full support for projects using the Swift Package Manager (with features such as auto-complete, diagnostics or jump-to-definition).

[14] `https://github.com/apple/sourcekit-lsp`

Conclusion

Swift is a language language which is receiving significant attention. While still being very young, it is quickly broadening its possible fields of usage. Starting in application development, soon the community brought it to server-side development and recently, Swift took a step towards data science when a Python interoperability and integration with TensorFlow were introduced. This thesis aims to take the first step on another Swift journey, making it possible to use it in the field of embedded systems.

In the Implementation chapter, the entire process was documented of adjusting the Swift toolchain in order to be able to target bare-metal systems. To ensure that the generated machine code works as expected, we also made it possible to run a subset of Swift's test suite within an emulator. Next, as to date, Swift has not been used in applications with high memory constraints, a code-size issue had to be addressed. We extended the compiler with a set of features allowing Swift programs to fit commonly available microcontrollers with the focus on the ARM M-Profile architectures. In order to leverage all the tooling available, we also extended the Swift Package Manager to support the newly added bare-metal platform and created several packages required when targeting bare metal. Finally, the whole solution was evaluated using an STM32F4 board.

Our results show that the use of Swift for Embedded Systems can result in comparable performance to that of a C-based platform, while making use of all the high-level features Swift offers. The current difficulty is the code size of such programs, limiting the use to microcontrollers with higher program-memory sizes only. On the other side, Swift for Embedded Systems benefits significantly from the support of the Swift Package Manager, making it very easy to develop an embedded application, using Swift together with C. As C is dominating the segment and most of the available code is written in it, we believe the high interoperability with C is essential for the further success of Swift in this segment.

As a result of the work described in this thesis, it is possible to use Swift

on embedded systems. We therefore consider the goal of this thesis to be achieved. We are planning to open-source all the code shortly[15] and start a discussion with the Swift community about merging the changes to the Swift upstream repository.

Future Work

This thesis opens up space for a number of follow-up works, such as:

- *Increasing the number of supported boards and peripherals.* Currently, we added basic support for the NUCLEO-F439ZI board. We plan to improve this support, possibly providing access to most of its peripherals. However, supporting other boards is as simple as creating a new package with a linker script and startup code.
- *Implementing further optimizations reducing the code size.* Code size of the standard library is currently the main obstacle to using Swift on smaller embedded devices. This can be addressed by implementing link-time optimizations removing unreachable code.
- *Making it more accessible for beginners.* Swift aims to be a great first language[16]. There is an opportunity to create a simple Arduino-like experience with Swift for Embedded Systems, making it possible to run Swift on an embedded device with one click. Thanks to the already existing tools, this might be simple to implement in comparison to other platforms.
- *Adding support for other architectures.* A very popular chip frequently used in the Internet of Things is ESP-32 made by Espressif. LLVM currently does not support code generation for its architecture; however, this can change soon, as Espressif is now officially working on adding such a support to LLVM.

[15] `https://github.com/swift-embedded`

Bibliography

[1] Inc, A. Memory Safety. Available from: https://docs.swift.org/swift-book/LanguageGuide/MemorySafety.html

[2] Health, S. *Embedded Systems Design – Second Edition*. 2003. Available from: https://books.google.cz/books?id=BjNZXwH7HlkC&lpg=PA2&hl=cs&pg=PA2

[3] Embedded Systems Glossary. Available from: https://barrgroup.com/Embedded-Systems/Glossary-E

[4] StackOverflow – Developer Survey Results 2019. Available from: https://insights.stackoverflow.com/survey/2019#technology

[5] John, M. Ownership Manifesto. Feb 2017. Available from: https://github.com/apple/swift/blob/master/docs/OwnershipManifesto.md

[6] John, M. Enforce Exclusive Access to Memory. Available from: https://github.com/apple/swift-evolution/blob/master/proposals/0176-enforce-exclusive-access-to-memory.md

[7] John, M. Error Handling Rationale and Proposal. Available from: https://github.com/apple/swift/blob/master/docs/ErrorHandlingRationale.rst

[8] Inc., A. Swift Compiler And Standard Library. Available from: https://swift.org/compiler-stdlib

[9] Limited, A. Compiler Reference Guide Short Enums. Available from: http://www.keil.com/support/man/docs/armclang_ref/armclang_ref_chr1411640303038.htm

[10] QEMU the FAST! processor emulator. Available from: https://www.qemu.org

[11] Testing Swift. Available from: https://github.com/apple/swift/blob/ad59d90/docs/Testing.md

[12] Code Size Optimization Mode in Swift 4.1. Available from: https://swift.org/blog/osize/

[13] Khattar, T. Dominator Tree of a Directed Graph. Available from: https://tanujkhattar.files.wordpress.com/2016/01/dominator.pdf

[14] Micropython – Performance. Available from: https://github.com/micropython/micropython/wiki/Performance

[15] Fannkuch-Redux – Description. Available from: https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/fannkuchredux.html

[16] Introducing Swift 5 – Great First Language. Available from: https://developer.apple.com/swift/

Acronyms

API Application Programming Interface. 13, 43

ARC Automatic Reference Counting. 5

AST Abstract Syntax Tree. 9

EABI Embedded Application Binary Interface. 13

FPU Floating-Point Unit. 40

GC Garbage Collection. 5

HAL Hardware Abstraction Layer. 43–45

IDE Integrated Development Environment. 56, 57

IR Intermediate Representation. 9

ISR Interrupt Service Routine. 54, 55

MCU Micro-Controller Unit. 40, 41

OOP Object-Oriented Programming. 3

OS Operating System. 1

RISC Reduced Instruction Set Computer. 11

RTOS Real-Time Operating System. 1

SIL Swift Intermediate Language. 9

UART Universal Asynchronous Receiver-Transmitter. 40

Appendix A

Contents of enclosed SD card

- `Dockerfile` ..... a Dockerfile building the Swift toolchain
- `src` ..... the directory of source codes
- `dominator-symbol` .... the directory with the tools for object-file analysis
- `projects` ..... the directory with Swift packages for bare-metal
- `thesis` ..... the directory of LaTeX source codes of the thesis
  - `DP_Dragomirecky_Alan_2019.pdf` ..... the thesis text in PDF format