# Porting Go to NetBSD/arm64

Maya Rashish

## Abstract

Go makes the unusual choice of a custom toolchain and hand-written assembly. The rationale behind some of those choices is explained and techniques used for solving problems are mentioned.

## 1 Introduction

Golang or Go[1] is a statically typed, compiled, garbage-collected language. It enables easy concurrency and cross-compilation. The most popular implementation of Go is self-hosted (written in Go), and is independent of libc. The choices made by Go create difficulties for adapting the compiler for new targets. This paper will discuss the adaptation of Go to a NetBSD Aarch64 machines and the difficulties involved in the process.

## 2 Difficulties

### 2.1 Custom tooling

In many languages today, the implementation can be described as the following steps:

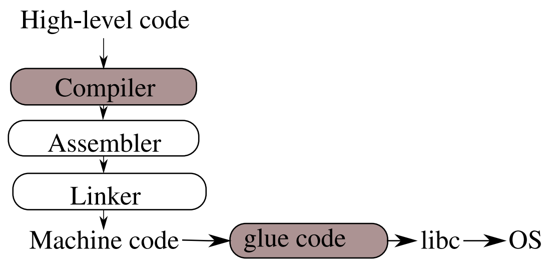

Figure 1: Typical language overview.
Language-specific parts are in brown.

[1] `https://golang.org/`

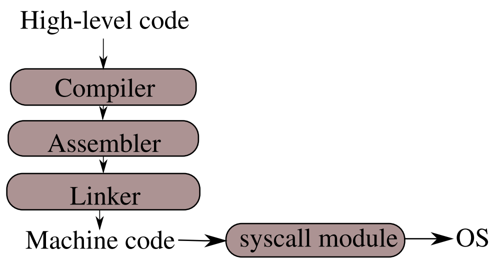

Figure 2: Go top-level overview.
Go-specific parts are in brown.

Go has chosen to take a different approach. Due to limitations to existing tools at the time, Go has chosen to incorporate tools implemented by its authors for the Plan9 project[2]. These include, unusually, a custom assembler with a special assembly syntax[3] and linker. Go doesn't use any external toolchain code.

This allows the Go code to support cross-compilation without needing to adjust any external packages, but comes at the cost of being another component that needs to be taught about new architectures.

Since the port to NetBSD/arm64 was the third OS (after Linux and Darwin), this didn't require any additional changes by the author.

### 2.2 CGo overhead

**Stack** The Go stack is not suitable for running arbitrary foreign C code, so another stack is used for foreign function calls.

**ABI** Go's internal function calls know not to trample certain global state. It must be saved/restored on function calls into foreign code.

**Internal accounting** Go manages scheduling of goroutines

[2] Plan9 operating system
[3] `https://golang.org/doc/asm`

and a garbage-collector. These require timing information to handle.

Go code communicates whether a function may block, and foreign code doesn't do this, so it has a special accounting state. We must communicate this to the accounting before switching to the foreign code.

## 3 Implementation

Custom code to call the operating system

Functions like "sleep for a few seconds" are very commonly used, so as an optimization, it is preferable to use custom Go-like code for them.

For this reason, Go had to be taught a lot of information normally contained within libc, like how to open files, how to exit. These interfaces are specific to the operating system, so code specific to NetBSD had to be written.

The majority of the effort of porting Go to NetBSD/arm64 was spent on teaching Go about how to ask the operating system to do certain things normally done in libc.

Other ports to arm64 exist, Linux and Darwin. The Linux/arm64 implementation was a source of inspiration, as was the cost for NetBSD/amd64 systems. In NetBSD, libc contains system call code and was inspected for comparison.

The Darwin implementation will call into system libraries, as Darwin doesn't offer backwards compatibility for code using system calls directly.

### 3.1 ABI

Using system calls is typically a matter of passing arguments in a previously agreed upon manner, and calling a special "syscall" instructions which switches into the kernel.

Typical calling convention for Aarch64 functions[4]:

| | |
|---|---|
| SP | Stack pointer |
| r0..r7 | Input and output registers |

Most system calls within NetBSD follow the function calling convention, and would use r0 for the first input argument, r1 for the second, and so forth. Additional arguments are passed on the stack.

[4]AArch64 Procedure Call Standard

```
mmap(0, 0x8000, 0x3, 0x1002, 0xffffffff,
0, 0) = 0x7f7ff7ef7000
open("/etc/ld.so.conf", 0, 0x7f7ff7e12768)
Err#2 ENOENT
```

Figure 3: Typical ktrace output

However not all system calls followed this convention. SYS_syscall (syscall number #0), which has the syscall number as the first argument, uses r17 for passing the syscall number.

Similarly, linux chooses to use r8 for passing the syscall number, instead of passing the paramter using the Aarch64 syscall instruction ("SVC") paramter.

### 3.2 Debugging

The porting effort consisted of writing around 500 lines of assembly code, a very error-prone effort, prior to any testing. Unsurprisingly, the first attempt to run any code didn't work.

For the purpose of debugging, ktrace[5] was used.

System call numbers appear in their names, and the arguments are enclosed in parentheses, similar to C function calls. Return values or errors are shown after the closing paren.

ktrace was an invaluable tools, as most of the mistakes were within Go code.

## 4 Results

At the end, simple programs run. Additional work is done to build the compiler natively. The code is available online and is awaiting review by Go upstream. [6]

[5]Running binaries prepended by the ktruss(1) command

[6]https://github.com/golang/go/pull/29398