Bootstrapping The Go Runtime For Use On Embedded Systems

by

Yanni Coroneos

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author . . . .
Department of Electrical Engineering and Computer Science
May 18, 2017

Certified by . . . .
Dr. Frans Kaashoek
Charles Piper Professor
Thesis Supervisor

Accepted by . . . .
Dr. Christopher Terman
Chairman, Masters of Engineering Thesis Committee

Bootstrapping The Go Runtime For Use On Embedded Systems

by

Yanni Coroneos

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

0.1 abstract

Embedded systems are becoming increasingly complicated due to the emergence of SOC's (system-on-a-chip) with multiple cores, dizzying amounts of peripherals, and complicated virtual memory systems that mirror that of x86. However, performant embedded programs are still largely written from scratch in C, which leads to constant re-implementation of the same subsystems. This wastes time and increases the likelihood of bugs. Real-Time Operating Systems (RTOS's) do exist but they impose their own driver models and are also written in C, still increasing the likelyhood of bugs.

This thesis explores a system named G.E.R.T : a modified a version of the Go runtime intended to run bare-metal on ARMv7a SOC's in order to evaluate the effectiveness of using a high-level, type-safe, and garbage collected language for embedded applications. G.E.R.T, the Golang Embedded RunTime, provides the multiprocessor support and basic memory abstractions of a typical RTOS while also freeing the user to leverage the language features of Go in order to develop custom driver models and build more complicated concurrent programs that are easier to reason about than similar ones written in C.

Thesis Supervisor: Dr. Frans Kaashoek
Title: Charles Piper Professor

Acknowledgments

Not so fast

Contents

List of Figures

List of Tables

Chapter 1

Why Write Low Level System Code in Go?

Modern embedded systems are composed of multicore SOCs that require careful thought about race conditions and event serialization. Like operating system kernels, most of these embedded systems are still implemented in C because it is a simplistic language that makes it good for "bare-metal" development.

outline: computers are quicker and multicore programming is scary, but everyone still uses C

C's simplicity makes it error prone for concurrent programs. Because C is very simple, the programmer must implement additional complexities in order to write concurrent programs.

try2: Low-level system code has been written in C since the 1970's because it is powerful and reliable. C can be used to express any operation a computer can do and it can also be compiled to fast byte code. Once, during an interview, an engineer even remarked: "If you can't do it in C, you can't do it". This does not mean that C is always the best choice though.

Even though multicore systems are commonplace now, kernels are still written in C.

Writing complex concurrent programs in C is too hard. Because C is very simple, the programmer must implement additional complexities in order to write concurrent programs.

There are very few built-in abstractions so it is left to the programmer to layer additional complexities in order to accomplish a task.

try 1: Low-level system code has been written in C since the 1970's because C is powerful and reliable. C can be used to express any operation a computer can do and it doesn't come with any baggage like languages with a runtime do. C is also easy to learn because it doesn't require advanced degrees in order to comprehend, like Haskell and Coq do. The problems with C only begin to show when concurrency comes into play. C, by itself, has no idea of concurrency or concurrent programming patterns. It is really all up to the programmer to lay down these abstractions. Combined with the burden of manual memory management, concurrent programming in C almost always results in pouring over JTAG trace logs for hints of a race condition. Faced with this bleak outlook, perhaps it is reasonable to take a performance hit in exchange for faster development and less bugs. After all, computers have gotten significantly quicker in the last 20 years. This is where Go can come in. Go is meant to be a systems language that provides fundamental support for concurrency and cummincation

Chapter 2

Booting the Go Runtime

Even though Go code is compiled, it relies on a runtime to coordinate certain actions with the OS. Timers, locks, and file descriptors are just a few of the OS abstractions that the runtime hinges on in order to function at all. This means that getting compiled Go code to run bare metal on an SOC requires more than just a boot loader, the Go runtime itself must be modified to work without any OS abstractions. This poses a bootstrapping problem because any modifications made to the Go runtime's initialization process must not inadvertently cause it to use an abstraction that does not yet exist. For example, creating a new object with $make()$ would be disasterous if the GC has not yet been initialized. In observation of these constraints, G.E.R.T boots via a 3-stage process. The first stage is u-boot, which configures the clocks and copies the second stage off of an sdcard into memory before jumping into it. The second stage bootloader is a small C program which contains the entire G.E.R.T kernel ELF in its data section. This stage sets up the inital Go stack and loads the G.E.R.T ELF into memory before jumping to its entry point. The third stage of the bootloader lives inside G.E.R.T and is mostly written in Go, along with some Plan 9 assembly. It finishes the boot process.

Working off the initial stack from stage 2, the stage 3 bootloader enumerates all of RAM into page tables and creates an idenity mapping with a new stack before turning on the MMU. After this, a thread scheduler is setup and synchronization primitives, like $futex()$ are enabled. Additional CPU's are booted in main after the

Go runtime has finished initializing.

2.1 System Specification

G.E.R.T is written on a Freescale i.MX6 Quad SOC which implements the (32 bit) ARMv7a Cortex A9 MPCore architecture. The SOC sits on a Wandboard Quad baseboard. The i.MX6 Quad has 2GB of RAM and a wealth of memory mapped peripherals. Look at the memory map below. The rest of this chapter will discuss the implementation details of booting and running the Go runtime bare-metal on this SOC.

2.2 Stage 1 Bring Up

The u-boot loader is used to initialize device clocks and load the G.E.R.T bootloader into RAM. When the SOC is powered on, the program counter starts executing from ROM. The code in the ROM reads u-boot into RAM and jumps into it. U-boot programs the myriad of frequency dividers which are required to run the i.MX6 at a frequency of 792MHz per core. After this, u-boot is instructed to load the G.E.R.T kernel off the sdcard and into RAM at address 0x50000000. This address is specifically chosen because it does not overlap with any ELF program headers of the G.E.R.T kernel which are loaded in stage 2. After the stage 2 bootloader is in RAM, uboot jumps into it.

2.3 Stage 2 Decompression

The G.E.R.T bootloader sets up the initial Go stack and decompresses the Go kernel ELF into RAM. Much like Linux, the kernel of G.E.R.T is wrapped in a custom bootloader stage. This is necessary because G.E.R.T is compiled as a user space Linux program which expects a stack and the standard ELF auxiliary vectors. By default, the Go compiler links all programs at address 0x0. This would normally be a disaster for the i.MX6 because the first megabyte of RAM is either inaccessable or reserved for peripherals. One solution around this is to simply turn on the MMU in the stage 2 bootloader but this creates a headache with preserving page tables across the transition to Go land. An alternative, and much simpler, solution is to just change the link address of the Go ELF. This is the preferred approach so the link address was changed to 0x10000000. After loading the Go binary into RAM, the stage 2 bootloader reserves 4kb of initial stack and jumps into G.E.R.T.

2.4 Stage 3 Go Abstraction Synthesis

The thread scheduler and virtual memory system are statically initialized in order to prevent Go runtime subsystems from running before the environment is ready. At the beginning of execution, G.E.R.T is in a really constrained spot: Linux is not there but the Go runtime thinks it is. Specifically, there is no scheduler, no virtual memory, no syscalls. Nothing but a 4kb stack and the program counter. This is clearly inadequate for the Go runtime to do anything but crash, but G.E.R.T creates all of these missing subsystems (in Go) before the runtime actually uses them.

Chapter 3

Evaluation

G.E.R.T is evaluated in conditions that are representative of embedded workloads. The first few tests are benchmarks which measure G.E.R.T's maximum interrupt frequency and pin toggle rate. The last two tests are each case studies: a mobile sensor platform and a scanning-mirror galvanometer laser projector.

3.1 Pin Toggle Frequency

This test measures the raw speed at which G.E.R.T can toggle a simple GPIO pin on the iMX6 Quad SOC. In ARM assembly, this can be implemented in 4 lines. Higher pin toggling frequency represents more efficient code. In realistic workloads, GPIO pins would not be switched at more than a few kilohertz because there are usually dedicated peripherals for servicing complicated protocols, but the data is still an informative benchmark. Results are shown below in figure 3-1. If PWM is desired, G.E.R.T can produce a true PWM signal by using its PWM peripheral, as shown in the robot sensor platform case study.

As expected, assembly is much faster than any of the other platforms. Additionally, it is no surprise that GERT outperforms Go running in userspace Linux. It is not obvious why GERT is slower than userspace C code though. An inspection of the GERT bytecode yields the answer. To toggle a single GPIO pin in GERT, while using the embedded package, takes 47 instructions. Of these 47 instructions, 2 are function calls, 19 are loads, and 11 are stores. This clutter is a result of the abstractions present in the embedded package. When the same pin is toggled in GERT by directly manipulating the memory addresses (instead of using the embedded library abstractions) the frequency jumps to 500KHz. This measurement is indicated by the G.E.R.T unlayered row in figure 3-1.

Platform	Avg GPIO Toggle Rate
ASM	1.65MHz
G.E.R.T Unlayered	568KHz
Linux C	263KHz
G.E.R.T	154KHz
Linux Go	127 KHz

Figure 3-1: GPIO Toggle Rates of Different Platforms

3.2 Interrupt Response Time

This test measures the time it takes G.E.R.T to respond to an external event with another external event. Specifically, it is the time it takes to produce a rising edge on a GPIO pin in response to a falling edge on a different GPIO pin.

<table>

3.3 Pulse Counting

This test measures G.E.R.T's ability to count incoming pulses. This is a very important metric because motor encoders operate by sending out pulses to indicate motor rotation frequency and direction. The pulses are provided by a Xilinx Artix 7 FPGA and they are variable in frequency and count. In the mobile sensor platform case study below, the pulses are provided by an encoder.

<table>

3.4 Case Study: Robot Sensor Platform

In order to evaluate G.E.R.T on a realistic workload, I put it on a robot that was donated to me from MIT's MASLAB competition. Among other things, the robot has two drive motors with encoders and Sharp infrared distance sensors. I wrote a program in Go using G.E.R.T to interface all of these event sources at the same time.

3.4.1 Overview

The main body of the program is an event loop which waits for events coming out of an event channel. Independent goroutines monitor each sensor and send events into the event channel. The code for the event loop is shown below.

```go
select {
    case event := <-event_chan:
        fmt.Printf("%v\n", event)
        switch event {
...
```

Golang's higher order functions and closures were also leveraged in order to create a sensor polling helper function as shown below.

```go
type Pollfunc func() interface{}

func Poll(f Pollfunc, period time.Duration,
sink chan interface{}) chan bool {
    kill := make(chan bool)
    go func(kill chan bool) {
        for {
            select {
            case <-kill:
                return
            default:
                if period > 0 {
                    time.Sleep(period)
                }
                sink <- f()
            }
        }
    }(kill)
    return kill
}
```

The GPIO library was also configured to use interrupts in order to count pulses on the encoder as shown below.

```
    embedded.WB_JP4_10.SetInput()
    embedded.WB_JP4_10.EnableIntr(embedded.INTR_FALLING)
    embedded.Enable_interrupt(99, 0) //send GPIO1 interrupt to CPU0
.
.
.
//go:nosplit
//go:nowritebarrierec
func irq(irqnum uint32) {
    switch irqnum {
...
    case 99:
        inc()
        embedded.ClearIntr(1)
...
    }
}
.
.
.
func inc() {
    count += 1
}
```

With these powerful set of abstractions, adding events or sensors into the event loop is very simple because only a Pollfunc() must be implemented. As an added bonus, this G.E.R.T program is automatically concurrent because the Go and G.E.R.T schedulers will move idle cpus to any available goroutine. The robot implementation is not finished though because the motors and distance sensor must be interfaced with.

3.4.2 PWM Motor Control

The robot has an MDD10A motor speed controller for controlling the two drive motors. This device expects a pulse-width modulated signal (PWM) on its input pins in order to direct power into the motors. A PWM signal has a constant period but the signal is a logical "on" for part of the time and "off" for the rest of the time. The ratio of "on" time vs the period is called the duty cycle. It is this percentage which the motor controller translates into a speed for the motor.

The iMX6Q includes an on-board PWM peripheral which can output several channels of PWM at a variety of periods and duty cycles. G.E.R.T contains a driver for this in its "embedded" package. The PWM peripheral requires no maintenance once it is configured so the cost of outputting a PWM signal is essentially a few loads and stores every time the period or duty cycle changes.

3.4.3 Distance Sensor Reading

The Sharp distance sensor outputs an analog voltage proportional to its distance from the nearest object. A Microchip MCP3008 8-channel ADC is used to convert this into a digital signal. The MCP3008 only talks in clocked serial (SPI) though, with 24bit data frames. Luckily, the iMX6Q also has an SPI peripheral. Much like the PWM peripheral, it has multiple channel that can each concurrently send and receive data. The "embedded" package also contains a driver for the ECSPI peripheral and it works much the same way the PWM driver - requiring no input from the user other than the data to transmit and the length of the expected response.

3.4.4 Encoder Reading

Encoders emit a pulse every time the motor rotates a known amount. This amount is variable depending on the encoder resolution. The encoders on the test motors emit pulses at a max rate of 4KHz, corresponding to maximum motor speed. G.E.R.T had no difficulty picking up these pulses because this frequency is far less than the max pulse frequency in the benchmarks above.

3.4.5 Complications

Systems do not work perfectly, and this robot is no exception. When the motor controller turns on, it emits so much noise into the rest of the circuit that SPI stops working and the iMX6Q registers interrupts that did not happen. This is because the iMX6 uses 3.3v logic signals but the noise spikes in excess of 5v. To deal with this, the motor is run off of its own dedicated power supply when taking encoder measurements.

3.4.6 Result

G.E.R.T is a plausible system for robots with sensors. It is not too slow to miss encoder pulses and Go's language features effortlessly allow for useful primitives such as event loops and Polling function types. As with any embedded system though, care must be taken to minimize the influence of noise or else nothing will work.

3.5 Case Study: Laser Projector

A scanning-mirror galvanometer laser projector is a device that deflects a laser beam off of several mirrors in order to draw an image on another surface. If the entire image can be scanned faster than 24Hz, then the light appears to blend and the human brain perceives it as a single image rather than many points. The maximum rate at which the projector can trace points is bounded below by the speed of the galvos and bounded above by the speed of the software.