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. Despite this, performant embedded programs are still largely written from scratch in C, which leads to constant re-implementation of the same subsystems and difficult bugs.

This thesis explores a new system called G.E.R.T, the Golang Embedded Run-Time, for ARM processors. GERT is a modified version of the Go runtime for bare-metal operation 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 provides the multiprocessor support and basic memory abstractions of a typical embedded toolkit while also freeing the user to leverage the language features of Go in order to develop concurrent embedded programs that are easier to reason about than similar ones written in C.

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

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Why Write Low Level System Code in Go?

Stay tuned!

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.

<memory map of imx6 here>

<memory map of GERT here>

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 Kernel 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. 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, which is the start of RAM. 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 Runtime

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, so G.E.R.T creates all of these missing subsystems (in Go) before the runtime actually uses them.

2.4.1 Virtual Memory Setup

2.4.2 Thread Scheduling and Trapframes

2.4.3 Interrupts

2.4.4 Keeping Time

2.4.5 Booting Secondary CPUs

Chapter 3

Evaluation

G.E.R.T is a toolkit for embedded applications which is written in Go. The goal is for the programmer to be able to implement concurrent embedded programs which are as performant as the equivalent C implementation, but without having to worry about concurency itself. In order to evaluate G.E.R.T, there are two questions that must be answered:

1. Does G.E.R.T retain good performance despite the costly abstractions of Go?
2. Do the concurrency patterns of Go simplify the task of the programmer?

The first question is addressed by benchmarks below which measure G.E.R.T's speed in creating and responding to external events. Pin toggle (3.1) measures the maximum pin toggle frequency. Interrupt response time (3.2) measures the time it takes to respond to an external interrupt. Pulse counting (3.3) counts number of pulses at increasing frequencies.

The second question is harder to answer because difficulty is a subjective measure. I attempt to show that G.E.R.T presents a nicer framework for concurrency through two case studies: a robot sensor platform (3.4) ,which runs motors and reads sensors, and a galvo laser projector (3.5) which traces images onto a surface by rotating mirrors at high speeds.

In all tests, G.E.R.T is run on an i.MX6Quad SOC and all measurements are taken with a Rigol DS1054Z oscilloscope. When G.E.R.T is compared to Linux, the SOC runs Debian 8 "Jessie" with hardfloat support. G.E.R.T is also occasionally compared to a Teensy 3.2 running C. The Teensy 3.2 uses a Cortex M4, which is specifically intended for microcontroller applications, and has good real-time performance. Its event response times represent a good comparison point for G.E.R.T and Linux.

3.1 Pin Toggle Frequency

This test measures the 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, but compilers and abstractions can increase the instruction count. Higher pin toggling frequency indicates less code in the critical path. 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 for abstraction cost. Results are shown 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	127KHz

Figure 3-1: GPIO Toggle Rates of Different Platforms

The results show that G.E.R.T does suffer a performance decrease because of Go's abstraction cost, but it is tolerable. Go's abstraction cost is already evident from the two Linux benchmarks, what is not clear is exactly why G.E.R.T underperforms user-space Linux C code. Without all of the syscalls a user-space program must endure, there should have been a speed increase.

The slowdown is caused by Go's interfaces. GPIO pins are represented by an interface in the G.E.R.T embedded package. In order to toggle a single pin with these interfaces requires 47 instructions: 2 function calls, 19 loads, and 11 stores. If the benchmark program does not use the embedded package, but toggles the GPIO directly with regular functions, G.E.R.T achieves a significant speedup. This is indicated by the G.E.R.T unlayered row in figure 3-1.

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. A Teensy 3.2 is included in this test to establish a baseline for good response time. G.E.R.T and the Teensy detect the event with interrupt routines but Linux polls because the userspace sysfs driver does not expose interrupt attachment points. Results are shown below in figure 3-2.

In more complicated programs, a polling loop is disasterous because the program can miss events. G.E.R.T allows the programmer to directly interface with the interrupt controller in order to write true interrupt service routines that miss events far less often. To get the same behavior in Linux, the programmer either has to write a kernel module or hope that a better driver will eventually trickle into the kernel.

The Teensy has the fastest interrupt response time but G.E.R.T isn't far behind. G.E.R.T is meant to run on a powerful multi-core ARMv7a with more than 1 GB of RAM but the Cortex M4 is only single core and has less than 1MB of RAM. It is also programmed in C.

Platform	Event Reponse Time
Teensy 3.2	1μs
G.E.R.T	6.3μs
Linux C	10μs
Linux Go	TBD

Figure 3-2: Event Response Times of Different Platforms

The event response times follow the increasing abstraction cost for each system. The Teensy is very fast because its interrupt controller is vectored and interrupts do not cause a stack switch. This means that Corex M4 can flip a pin within a few cycles of receiving the interrupt. G.E.R.T is slower because the iMX6 does not have a vectored interrupt controller, the interrupt stack must be switched, and the interrupt handler is written in Go. When G.E.R.T gets an interrupt, it must save its current state, decide which interrupt it received, and execute its handler. This is much more complicated than the Cortex M4, but G.E.R.T benefits from the very high clock rate of the iMX6 (792MHz vs 96MHz).

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. The number of pulses missed at the limiting frequency is also recorded. In the robot sensor platform case study below, the pulses are provided by an encoder. Results are shown in figure 3-3.

Platform	Pulse Count	Max Pulse Rate	Missed Pulses
Teensy 3.2	10	TBD	TBD
G.E.R.T	10	161KHz	1
Linux C	10	161KHz	4
Linux Go	10	50KHz	1

Figure 3-3: Pulse Counts of Different Platforms

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 also Sharp GP2Y0A21YK infrared distance sensors on its perimeter. I wrote a program in Go using G.E.R.T to interface all of these event sources at the same time and operate the robot.

3.4.1 Overview

The main body of the program is an event loop which waits for events coming out of an event channel (fig. 3-4). Independent goroutines monitor each sensor and send events into the event channel. There is a single goroutine that monitors the event channel and manipulates state in a non-blocking manner. The code for the event loop is shown below.

```go
select {
    case event := <-event_chan:
        fmt.Printf("%v\n", event)
        switch event {
        case "p":
            val := adc.Read(0)
            fmt.Printf("adc reads %v\n", val)
        case "w":
            drive.Forward(0.2)
        case "s":
            drive.Backward(0.2)
        case "a":
            drive.TurnRight(0.2)
        case "d":
            drive.TurnLeft(0.2)
        case " ":
            drive.Stop()
        }
    }
```

Figure 3-4: Robot Event Loop

Golang's higher order functions and closures were also leveraged in order to create a sensor polling helper function as shown below in fig. 3-5. In this paradigm, every sensor gets its own goroutine which sends data back into a central event loop.

```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
}
```

Figure 3-5: Higher Order Polling Function

The GPIO library was also configured to use interrupts in order to count pulses on the encoder (fig. 3-6).

```
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
}
```

Figure 3-6: Encoder Interrupt

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 rest of this case study explains how the sensors are interfaced with GERT.

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 user changes the period or duty cycle.

3.4.3 Distance Sensor Reading

The Sharp distance sensor outputs an analog voltage proportional to its distance from the nearest object (fig 3-7). A Microchip MCP3008 8-channel ADC is used to convert this into a digital signal. The MCP3008 only communicates 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 channels 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.

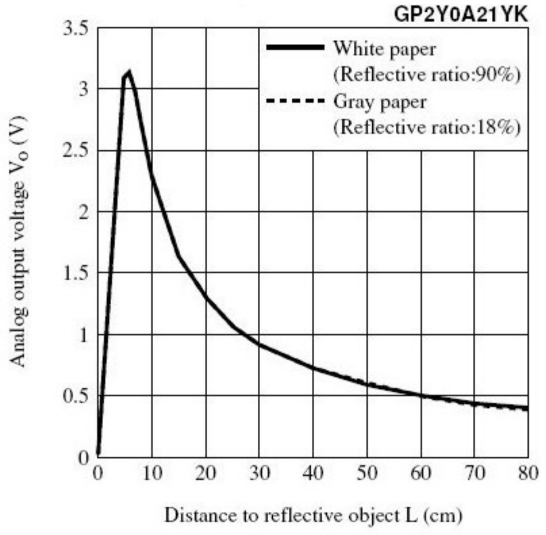

Figure 3-7: Sharp Sensor Distance Curve

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. A speed monitor was written in fig. 3-8 to measure the motor rotation (in Hz) and it corresponded very closely to the oscilliscope readings.

```go
//count is updated by the interrupt routine
//and it is the amount of encoder pulses
go func() {
  for {
    old := count
    time.Sleep(1 * time.Second)
    new := count
    event_chan <- new - old
  }
}()
```

Figure 3-8: Motor Speed Monitor

3.4.5 Complications

Systems do not work perfectly, and this robot is no exception. The switching motor controller used on this robot emits a lot of noise. The 5v noise spikes measured on the oscilloscope wreaked havoc on the 3.3v single-ended signals that the iMX6 operates with, causing serial communication failures and spurious interrupts. To deal with this, the motor is operated from a dedicated power supply when taking encoder measurements. Consequently, the physical robot cannot move when the motors are connected to an external power supply.

3.4.6 Result

G.E.R.T is a plausible embedded toolkit to use for robots that incorporate many sensor systems. By utilizing Go's language features, an embedded firmware engineer can implement a complicated sensor integration platform on top of GERT without worrying about issues like scheduling or shared memory. Go's runtime manages all of that headache and goroutines also allow for the program to scale with the number of available cpus.

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. I haven't written the rest of this yet so you will have to use your imagination from here.