G.E.R.T: A Go-Based Toolkit For Embedded Applications

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

G.E.R.T: A Go-Based Toolkit For Embedded Applications

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 SOCs (system-on-a-chip) with multiple cores, dizzying amounts of peripherals, and complicated virtual memory systems. Unfortunately, performant embedded systems for these SOCs are still largely written from scratch in C because heavy operating systems, such as Linux, can introduce more bugs and degrade real-time performance.

This thesis proposes a new system called G.E.R.T, the Golang Embedded Run-Time, for multicore ARM processors. GERT is a modified version of the Go runtime for bare-metal operation on multicore 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 enabling 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

Introduction

Modern embedded systems are composed of multicore SOCs that require careful thought about race conditions and event serialization. C is the most commonly used language to program such low level systems, but it is a double-edged blade. Low level code written in C can more easily and directly interface with hardware, but it can also be plagued with difficult bugs, especially concurrency-related bugs. Concurrent user-space programs, on the other hand, are usually written in a high-level language, such as Rust or Go, which abstracts concurrency and provides memory safety.

Embedded programmers are often drawn towards using Linux for their work because then they can write their embedded program in user space with a high level language. Popular platforms that use this paradigm include the Raspi and Beaglebone. However, embedded programs that run in user space suffer from significant event latency because external interrupts must shuffle their way through the kernel and the kernel's scheduler will also constantly preempt the program. Usually, an experienced programmer does not even want or need the kernel, but they pay the price just to run the high-level code.

There are ongoing efforts to bring high-level languages to desktop operating system kernels and also single-core microcontrollers, but there is no known system which provides a high-level language environment for multicore SOCs. Singularity <cite> and Biscuit <cite> are desktop operating system kernels, written in Sing# and Go, which focus on hosting user-space programs. Copper <cite> and MicroPy-

thon <cite> are small embedded toolkits, written in Rust and Python, which aim to provide a high-level programming environment for single-core microcontrollers. Multicore SOCs have, so far, been left out of the picture. This thesis presents a new embedded toolkit, G.E.R.T, which is specifically intended for concurrent, bare-metal embedded applications.

G.E.R.T is a bare-metal, Go-based embedded toolkit for multicore ARMv7a processors. It was developed in order to make bare-metal programming easier for ARMv7a SOCs with the help of Go's channels and goroutines. G.E.R.T can run on a single-core processor but its effectiveness is substantially reduced because any blocking operation can lock the whole system. On the other hand, G.E.R.T will automatically scale to utilize all available cpus in multicore systems because the Go runtime automatically scales to all available cpus. It is unlikely that a low-level system implemented with G.E.R.T will outperform the same system written in good C, but results show that G.E.R.T does outperform userspace C code in Linux.

One particular concern about using a garbage-collected language for low-level code is GC pause times. This is not an issue in G.E.R.T because interrupt handlers can execute even when the world is stopped as long as the interrupt code does not call the Go scheduler. This is an acceptable constraint for embedded environments because any interrupt code that calls the Go scheduler is performing an operation that could potentially block, and blocking operations should not be happening during interrupt time in the first place.

1.1 Why Write Low Level System Code in Go?

At first glance, Go code looks a lot like C. There are no classes and every object has a type which is known at compile time. This already makes Go a good systems language, but Go's greatest feature is its built-in support for concurrency through goroutines and channels. Goroutines are lightweight threads that the Go runtime can schedule without help from the operating system. Channels are typed FIFO queues which can help to serialize asynchronous events, perhaps coming from several

goroutines. With these features, Go is like an updated version of C for multicore systems, but without buffer overflows and null pointer dereferences.

Go's implementation mirrors that of a small real-time OS. Go's threads are lightweight and cooperatively scheduled so that execution only transfers during blocking operations. The runtime also manages its own pool of memory and exports its own atomic primitives through the standard "sync" package. In fact, Go provides most of the common OS primitives natively in its standard libraries <https://golang.org/pkg/#stdlib>. This means that G.E.R.T can also provide most of the convenience of a full-blown kernel without latency degredation.

Embedded systems are increasingly relying on dedicated peripherals to provide service, instead of very fast cpus. SOCs contain dedicated silicon peripherals to help with everything from serial communication to interrupt priority filtering. These peripherals free the cpus from bit-banging high-frequency signals so they can spend more time directing program flow instead. Go fits in well with such a system because its goroutines can be used to concurrently monitor state and channels can be used to relay that information back to a central coordinator. When an output must be switched, G.E.R.T simply issues a driver call that changes the behavior of a peripheral.

1.2 Outline

Outline the rest of the thesis

Chapter 2

Booting the Go Runtime

<insert figure from proposal that shows boot process>

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 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. Unlike desktop PCs and laptops which use a BIOS to configure the frequency dividers and RAM timings, the iMX6 has no such thing so u-boot does it instead. 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 loads 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 G.E.R.T's build system uses a link address of 0x10000000, which is the start of RAM on the iMX6. 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 constrained situation: 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

With G.E.R.T, the programmer should be able to implement concurrent embedded programs which are as performant as the equivalent C implementation, but without having to worry about concurency abstractions. 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 costs of using a High Level Language?

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 (3.2) measures the maximum pin toggle frequency. Interrupt response time (3.3) measures the time it takes to respond to an external interrupt. Pulse counting (3.4) counts number of pulses at increasing frequencies.

The second question is harder to answer because difficulty is a subjective measure. This thesis attemps to show that G.E.R.T presents a better framework for concurrency through two case studies: a robot sensor platform (3.6) ,which runs motors and reads sensors, and a galvo laser projector (3.7) which traces images onto a surface by rotating mirrors at high speeds.

3.1 Experimental Setup

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 microcontroller 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. The Teensy platform has poor concurrency support though because the Cortex M4 is a single core processor and it only has 64 kilobytes of RAM.

All of the tests make use of the G.E.R.T "embedded" package, which is an example driver library that was developed for the i.MX6Quad. The embedded package is not intrinsic to G.E.R.T's functionality, nor was it optimized for performance in any way. The embedded package just aims to provide a template for how drivers can be written in the Go language.

3.2 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. G.E.R.T toggles the GPIO pin by directly interfacing with the GPIO peripheral on the iMX6, but userspace Linux code must use the sysfs driver. Results are shown in figure 3-1.

Platform	Avg GPIO Toggle Rate
ASM	1.65MHz
G.E.R.T Static	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 not clear 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 in the embedded package. The GPIO driver in the embedded package uses interfaces to abstract all of the different pins. In order to toggle a single pin with these interfaces requires 47 instructions: 2 function calls, 19 loads, and 11 stores. A more performant driver model would use static device IDs so that the compiler can infer the toggled pin directly. The performance of a static implementation is shown above in the G.E.R.T static row of figure 3-1. Even with a static driver model, G.E.R.T retains all of the benefits of a HLL. It is the programmer's decision which kind driver model to use.

The pin toggle frequencies show that G.E.R.T is capable of more efficient execution than a userspace C program in Linux. Pin toggle inside of a Linux kernel module would likely attain the same performance as G.E.R.T, but would lack the safety a HLL provides.

3.3 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. Faster response times are important for real time control systems, such as ABS brakes in a car. G.E.R.T and the Teensy detect the event with hardware interrupts but Linux polls the input pin in a tight loop because the userspace sysfs driver does not expose interrupt attachment points. Results are shown below in figure 3-2.

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 the Teensy 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

Platform	Event Reponse Time
Teensy 3.2	1μs
G.E.R.T	6.3μs
Linux C	10μs
Linux Go	30μs

Figure 3-2: Event Response Times of Different Platforms

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 Go handler. Despite this complexity, the iMX6 can execute more instructions in less time because of its very high clock rate (792MHz vs 96MHz) so it can keep up with the Teensy.

The Linux configuration is slower because external interrupts cannot directly trigger a response from userspace. In Linux, the GPIO pins are represented by file descriptors so IO is performed by reading/writing from the appropriate file. In response to an external interrupt, the Linux kernel sets a flag on the file descriptor which means that there is data to read. The userspace program does not actually see the data until it is scheduled again.

As before, a Linux kernel module would likely attain the same performance as G.E.R.T (or even more), but would lose the benefits of a HLL.

3.4 Pulse Counting

This test measures G.E.R.T's ability to count incoming pulses. Missed pulses indicate an excessively long interrupt handler that is still executing when the next pulse arrives. None of the platforms were configured for re-entrant interrupt handlers so they can all potentially miss pulses. The pulses are provided by a Xilinx Artix 7 FPGA and they are variable in frequency and count. The Linux benchmarks are run with polling again because the userspace sysfs driver does not support interrupt attachment. Results are shown in figure 3-3.

G.E.R.T and userspace Linux C achieve similar performance on this test. Linux

Platform	Pulse Count	Max Pulse Rate	Missed Pulses
Teensy 3.2	10	2.50MHz	2
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

just missed a few pulses.

The Teensy registers more pulses than any other platform because of its compact and efficient architecture. A dissassembly of the Cortex M4 binary reveals that all instructions are encoded in 16bit ARM Thumb, as opposed to 32bit ARM that armv7A SOCs use. Additionally, the Cortex M4 uses a vectored interrupt controller so that the entire length of its interrupt service routine is only 5 instructions and has no conditionals.

3.5 Benchmark Conclusions

G.E.R.T shows that it can usually outperform userspace Linux C code and it can keep up with the Teensy in event response time. G.E.R.T also has a true interrupt handler whereas Linux must poll due to the sysfs driver. Unfortunately, G.E.R.T has a complicated interrupt pipeline with many conditionals so it can miss more interrupts than the Teensy; G.E.R.T is not suitable for meeting hard deadlines.

The Go garbage collector was never an issue during any of the tests because embedded programs are usually static.

3.6 Case Study: Robot Sensor Platform

Figure 3-4: Code Breakdown of 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 several Sharp GP2Y0A21YK infrared distance sensors on its perimeter. I wrote a program in Go using G.E.R.T to process all of these event sources at the same time and operate the robot.

3.6.1 Overview

The main body of the robot program is an event loop which waits for events coming out of an event channel (fig. 3-5). 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 in fig. 3-5.

```
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 3-5: Robot Event Loop

The robot program uses Go's higher order functions and closures in order to create a sensor polling helper function as shown below in fig. 3-6. In this paradigm, every sensor gets its own goroutine which sends data back into a central event loop.

The robot program also configured the GPIO library to use interrupts in order to count pulses on the encoder (fig. 3-7).

```
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() //sink is usually the event channel
            }
        }
    }(kill)
    return kill
}
```

Figure 3-6: Higher Order Polling Function

```
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 3-7: Encoder Interrupt

With these powerful set of abstractions, adding events or sensors into the event loop is 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.

3.6.2 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 and the signal is a logical "on" for part of the time and "off" for the rest of the time (fig. 3-8). 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.

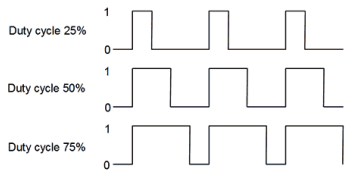

Figure 3-8: Sample PWM Signals

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 PWM peripheral 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.

The driver is organized in a typical C fashion where the memory map of the peripheral is represented in a structure:

```
1  type PWM_regs struct {
2    CR  uint32
3    SR  uint32
4    IR  uint32
5    SAR uint32
6    PR  uint32
7    CNR uint32
8  }
```

3.6.3 Distance Sensor Reading

The Sharp distance sensor outputs an analog voltage proportional to its distance from the nearest object (fig. 3-9). A Microchip MCP3008 8-channel ADC is used to convert this voltage into a digital signal. The MCP3008 communicates in clocked serial (SPI) with 24bit data frames so the robot program uses G.E.R.T's SPI driver. Much like the PWM peripheral, the SPI peripheral has multiple channels that can each concurrently send and receive data. The SPI driver also requires no input from the user except for the data to transmit and receive.

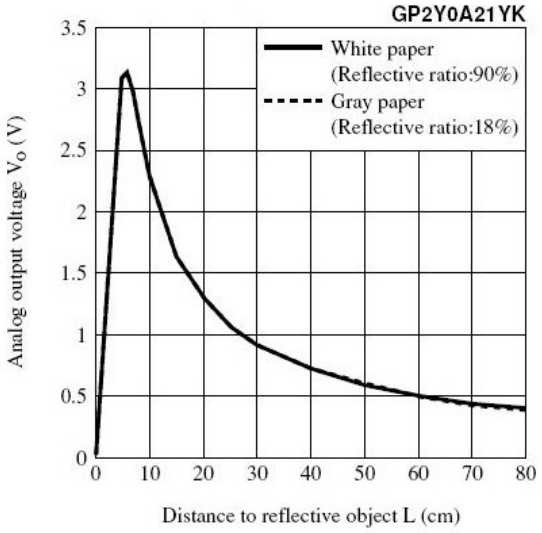

Figure 3-9: Sharp Sensor Distance Curve

3.6.4 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 fig. 3-3. A speed monitor was written in fig. 3-10 to measure the motor rotation (in Hz) and it corresponded very closely to the oscilliscope readings.

```
1  //count is updated by the interrupt routine
2  //and it is the amount of encoder pulses
3  go func() {
4    for {
5      old := count
6      time.Sleep(1 * time.Second)
7      new := count
8      event_chan <- new - old
9    }
10 }()
```

Figure 3-10: Motor Speed Monitor

3.6.5 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 robot's motors are connected to an external power supply before taking measurements. This helps remove noise from the digital circuits. Consequently, the physical robot cannot move when the motors are connected to an external power supply.

3.6.6 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. Channels can be used to serialize asynchronous events and higher-order functions can be used to help with event polling. Additionally, Go's runtime manages all of the memory and allows for the program to scale with the number of available cpus.

3.7 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, as shown in fig. 3-11. 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. In this case study, G.E.R.T is used to implement a laser projector with a red laser.

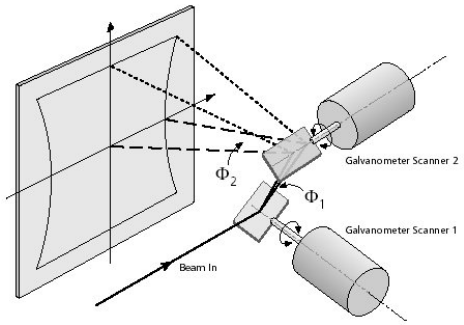

Figure 3-11: Mirror Galvanometers

3.7.1 Overview

Points are generated from a vector graphics file on a desktop computer and stored on an sdcard before G.E.R.T loads them and traces the image. The laser projector, unlike the robot sensor platform, does not have to process any external events or manage concurrency. Instead, the laser program must just be fast enough to trace points smoothly. To do this, G.E.R.T runs a dedicated goroutine, *lasermon* which loops over all of the points in a circular buffer and sends them in order to a Microchip MCP4922 DAC. The DAC converts the digital position signal into an analog voltage and then sends that voltage signal into an analog servo circuit, which sets the position of the mirrors.

3.7.2 Point Serialization

The laser program uses Go's native Gob library in order to serialize and deserialize points. The laser projector is currently limited to two dimensional images with a single red laser, so only three properties must be stored for each point: X position, Y position, and Color. The DAC only has a 12bit resolution so 16bit integers are used to store each point. The struct is shown below in fig. 3-12

```
1 type CompactPoint struct {
2     X     uint16
3     Y     uint16
4     Color uint8 //either 0 or 1
5 }
```

Figure 3-12: Laser Point Structure

In order to store points, a seperate encoding program running on a desktop computer encodes an array of *CompactPoint* structs into a Gob object and writes them to a file. Next, the encoding program stores the file onto the FAT32-formatted sdcard that contains the G.E.R.T program. Then, G.E.R.T reads the same file and de-serializes the Gob object into an array of *CompactPoint*, like in fig. 3-13.

```
1     good, root := embedded.Fat32_som_start(embedded.Init_som_sdcard,
2   embedded.Read_som_sdcard)
3     if !good {
4         fmt.Println("fat32 init failure")
5     }
6     good, bootdir := root.Direnter("BOOT")
7     if !good {
8         panic("dir entry failed")
9     }
10  /* P.TXT contains the Gob'ed points */
11    good, contents := bootdir.Fileread("P.TXT")
12    if !good {
13        panic("file read failure")
14    }
15    r := bytes.NewBuffer(contents)
16    d := gob.NewDecoder(r)
17    err := d.Decode(&points)
18    if err != nil {
19        fmt.Printf("error de-GOBing:\n")
20        panic(err)
21    }
```

Figure 3-13: Laser Point Structure