**Final Report:**

**Operational Switching - Interfacing with Hardware Using Machine Code**

**Thomas Morton**

**CS3443 Honors Contract**

**Dr. Shital Joshi**

**November 14, 2022**

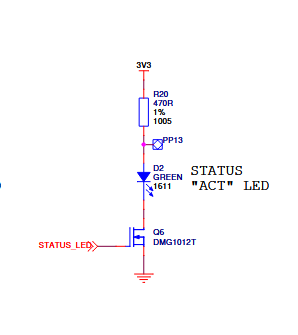
**Introduction**

At its most fundamental level, the process of computation performs a series of calculations by regulating a change in voltage. Computer scientists represent and signal these changes in voltage through binary representation, with ‘0’ corresponding to a low power state and a ‘1’ representing a higher power state. The series of instructions formed by these binary signals are interpreted by the different processing units of a computer and are known as machine code. Because this representation can be tedious, if not impossible, to understand, programmers created assembly language as a representation of machine code instructions. These instructions translate the processes given by higher-level programming languages such as C++ and Python and act as a middle ground between programmer and machine.

**Purpose**

The purpose of this project is to display an understanding of the underlying function of assembly language and its corresponding utility in computation. By powering on and off LEDs using a Raspberry Pi 3 B V1.2, this project exhibits the interaction of assembly language with hardware to achieve the desired change in state. This state change is indeed the goal of modern computation, driving all technological functionality from remote garage-door openers to household appliances and more. In doing so, this project parallels the essential operations that drive all modern technology.

**Initial Details**

The initial goal was to interact with the Raspberry Pi’s green ACT/OK LED. First, I needed to locate information on how these LEDs are mapped on the hardware. A reduced version of the hardware schematics was located on Raspberry Pi’s website [[[1]](#footnote-1)]. A brief overview of the wiring scheme of the green “ACT” LED is shown in Figure 1.1, which displays the LED receives power through a 3.3V rail.

**Figure 1.1:** Simplified wiring diagram

of the green “ACT/OK” LED

From there, more information would be needed on the system’s address scheme for the LED. I located information from the Raspberry Pi forums indicating that the GPIO responsible for controlling this LED is GPIO 16[[[2]](#footnote-2)]. With this information at hand, my next task was to find the address space that maps to this GPIO.

I found the address ranges for the Raspberry Pi’s peripheral devices in Robert Plantz’s *Introduction to Computer Organization: ARM Assembly Language Using the Raspberry Pi*. In this book, Plantz notes the GPIO memory address space to be between 0x3e000000 and 0x3effffff on the Raspberry Pi 3[[[3]](#footnote-3)]. In this section, Plantz also details how to map the physical address space to Raspberry Pi’s virtual address scheme. From here, I need to narrow down the address space to find the specific address for GPIO 16.

Searching for further details on the Raspberry Pi’s addressing scheme, I found documentation for the GPIO’s register view in Broadcom’s *BCM2835 ARM Peripherals.* This documentation manual shows that the Raspberry Pi’s GPIO contains 41 32-bit registers, along with their address space, read/write capabilities, field name, and description[[[4]](#footnote-4)].

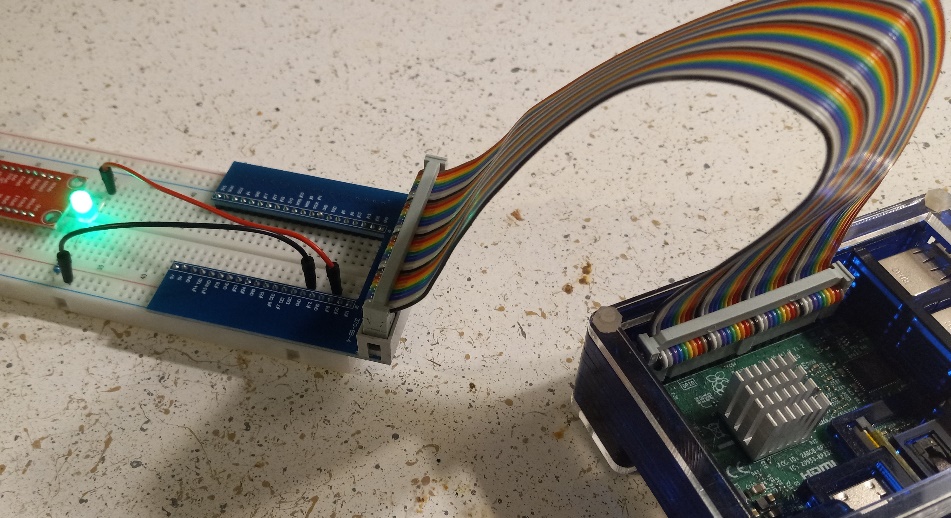
**Execution**

To serve as a proof-of-concept that I can interact with the ACT/OK LED, I wrote a simple bash script to signal the LED to blink on and off. This script writes a value to the “/sys/class/leds/led0/trigger/” file responsible for signaling the LED to turn on. By running this script, I was able to blink the LED, proving the LED is not outside of the user’s control. Something of note, however, is that writing to this file is not possible without root access. This hints at potential complications for the later steps in this process.

Following this discovery, I attempted to work directly with the Raspberry Pi’s ACT/OK LED using the above addressing scheme. While doing so, I encountered multiple technical difficulties in accessing the LED directly. Due to these difficulties, I chose to shift my focus to working with an LED wired to a breadboard using GPIO 16 instead. An additional bit of instruction was given following this change of task, so in addition I also wired another LED to GPIO 26 to blink opposite of GPIO 16’s power state. I conducted further research into the address space for the Raspberry Pi’s GPIOs and found that the physical address space for peripherals maps to a different virtual address space for Linux kernels[[[5]](#footnote-5)]. With this information, I found that instead of using 0x20000000 as the starting address space for the GPIOs, my version of the Raspberry Pi requires me to use 0x3F200000.

After locating the base address space for my GPIOs, I then needed to find how to set GPIO #16 and #26 as my target and then how to signal the pin as an output. Some searching led me to find that the pin designation for GPIO 16 is FSEL 16, with GPFSEL1 being the target register to set the GPIO’s function, and GPIO 26 under FSEL 26 and using GPFSEL2[[[6]](#footnote-6)]. Using this information, I then referenced *BCM2835 ARM Peripherals* and found the offset for GPFSEL1 at 0x04, GPFSEL2 at 0x08, and FSEL16 at 0x10000, and FSEL26 at 0x4000000 from the base address for the GPIOs[[[7]](#footnote-7)]. Following this, I needed the offsets to set and clear both GPIO 16 and 26. This information was also found in *BCM2835 ARM Peripherals* as 0x1C and 0x28 respectively[[[8]](#footnote-8)].

With the required information to load, set, and clear the GPIOs, I then began writing the assembly code to blink the LED. After writing the code, I then needed to compile a Linux kernel. A template for the kernel was downloaded from the University of Cambridge website[[[9]](#footnote-9)], which I then used to compile my written assembly into a working Linux kernel by copying the main.s file into the source directory of the template and issuing the *make* command from the Raspberry Pi’s terminal. After finishing the above steps, I succeeded in my goal of using the Raspberry Pi’s GPIOs to blink two LEDs.



**Figure 2.1:** Blinking LED in action (before the addition of the second LED)

**Conclusion**

The level of interaction with computer hardware that this project entails follows the same processes required for all forms of modern technology, whether the hardware’s purpose is as simple as powering an electric toothbrush to as complex as monitoring the fuel-to-air ratio in a modern engine. The required steps of locating the address space for the hardware, locating the registers responsible for designating the operation to be performed, then setting these registers are the same regardless of the technology in question. While this project posed many difficulties, it has given me a deeper respect for low-level programming, and I value the experience and knowledge I have gained.

Bibliography

Broadcom Europe Ltd. “BCM2835 ARM Peripherals - Raspberry Pi.” BCM2835 ARM Peripherals. Accessed November 13, 2022. https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf.

Canyon. “Can we control the on-board leds.” Raspberry Pi Forums. Accessed November 12, 2022. https://forums.raspberrypi.com/viewtopic.php?t=12530.

Department of Computer Science and Technology. “Department of Computer Science and Technology: Computer Lab Raspberry Pi Tutorials.” Accessed November 12, 2022. https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads/template.tar.gz.

Raspberry Pi Datasheets. “SCHEMATIC1: PAGE 2 – POWER XOS - raspberry-pi-3-b-plus-reduced-schematics.” Accessed November 12, 2022. https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-reduced-schematics.pdf.

Plantz, Robert G. “Introduction to Computer Organization: ARM Assembly Language Using the Raspberry Pi.” GPIO I/O Memory. Accessed November 12, 2022. https://bob.cs.sonoma.edu/IntroCompOrg-RPi/sec-gpio-mem.html.

1. SCHEMATIC1: PAGE 2 – POWER XOS, - raspberry-pi-3-b-plus-reduced-schematics, <https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-reduced-schematics.pdf> [↑](#footnote-ref-1)
2. Canyon, “Can we control the on-board leds,” from https://forums.raspberrypi.com/viewtopic.php?t=12530 [↑](#footnote-ref-2)
3. Robert Plantz, “GPIO I/O Memory,” in *Introduction to Computer Organization: ARM Assembly Language Using the Raspberry Pi,* Chapter 19.1 [↑](#footnote-ref-3)
4. Broadcom Corporation, *BCM2835 ARM Peripheral,* p. 90, from https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf [↑](#footnote-ref-4)
5. Broadcom Corporation, *BCM2835 ARM Peripheral,* p. 6, from https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf [↑](#footnote-ref-5)
6. Robert Plantz, “GPIO I/O Memory,” in *Introduction to Computer Organization: ARM Assembly Language Using the Raspberry Pi,* Chapter 19.2 [↑](#footnote-ref-6)
7. Broadcom Corporation, *BCM2835 ARM Peripheral,* p. 90, from https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf, p. 90, 92 [↑](#footnote-ref-7)
8. Broadcom Corporation, *BCM2835 ARM Peripheral,* p. 90, from https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf, p. 90 [↑](#footnote-ref-8)
9. “Index of /projects/raspberrypi/tutorials/os/downloads” from https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads/template.tar.gz [↑](#footnote-ref-9)