Joshua Pollock

CS 200 Project 11

5 May 2017

Joshua Pollock

CS 200

Project 11 – Direct IO

5 May 2017

**Purpose:**

This project is purely an extra credit project. There are very few guidelines and constraints for the project. All that I must do is use an assembly language to perform some sort of IO on a LED. In my case I decided to use the Raspberry Pi 3, and since the Raspberry Pi 3 uses ARM, I will be using ARM as my assembly language.

**Research:**

When I first decided to do this project, I was unsure if I was going to get it done or not. This was a few weeks after the start of the semester when we were still on circuitry. I ordered a Raspberry Pi 3, and then sat on this project for a while.

The next time I picked up this project was when we were beginning to learn the C and C++ languages. I started off by researching how to blink a LED using the C language. I kept running into one thing mentioned on many different sites called wiringPi (<http://wiringpi.com/>). The basic rundown of wiringPi is that it is a library usable by C that allows GPIO pin access. The website linked above gave me a general idea of what wiringPi was and how I would be able to use it in the future. Further research into Morse code with a Raspberry Pi, I found that there was a lot of python programs already created to do what I needed. While these programs were unusable in my project, they did prove useful in demonstrating how to wire an LED to the GPIO pins on my Pi. This is where I found I would need 220-330 ohms of resistance so I would not fry the Raspberry Pi with the LED. With no further leads, I again put this project on hold for a few more weeks.

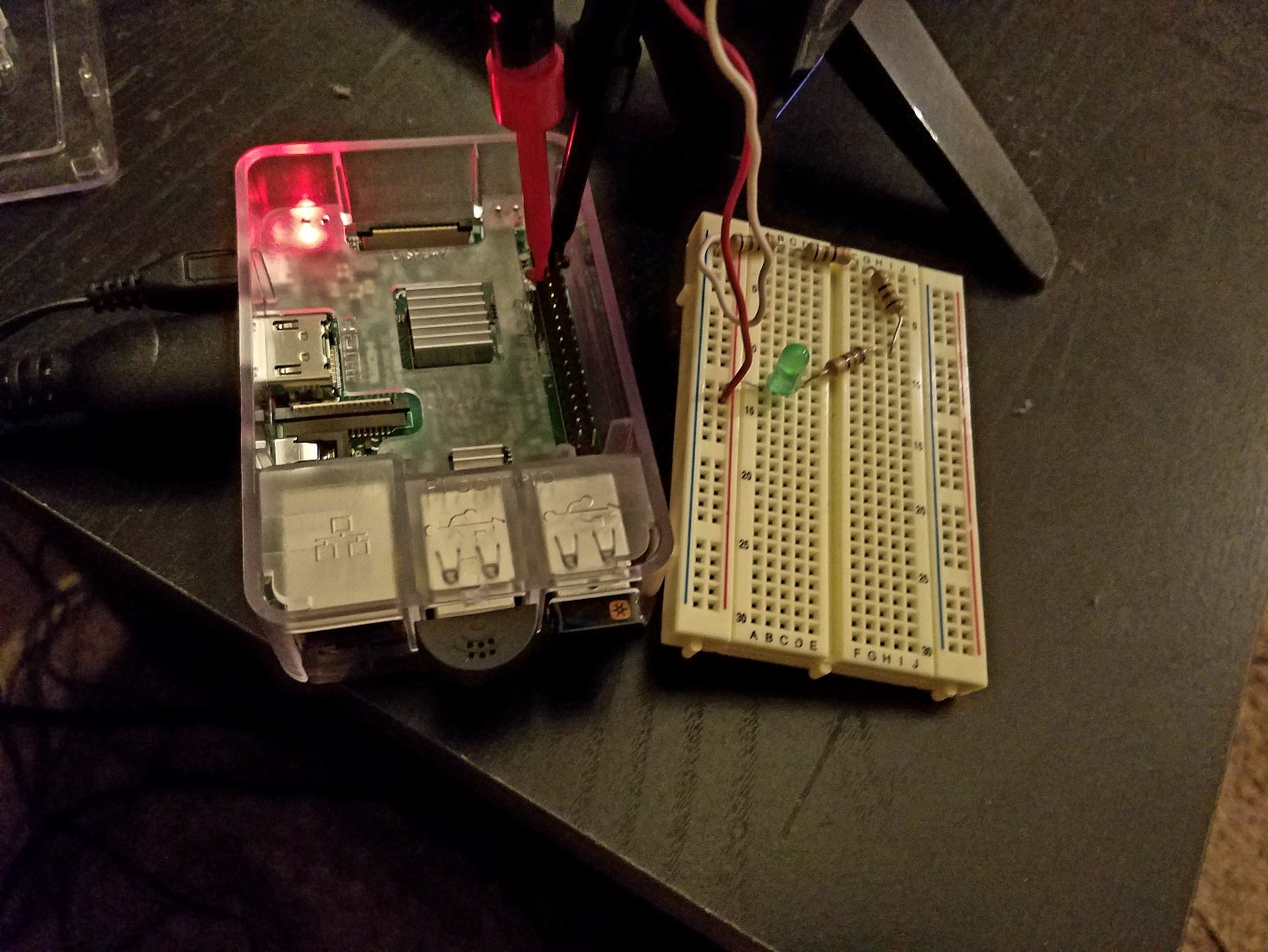
When I came back to this project again, I decided to run to RadioShack for the breadboard, LED, and resistors I would need. Funnily enough RadioShack was all out of 220-ohm and 330-ohm resistors. I decided to buy a 5 pack of 68-ohm resistors, and wire them in series to the achieve proper resistance. I also picked up a pack of these little wire grabber things which fit perfectly over the GPIO pins. I began to research more into C programs (we had not started MIPs yet) for Morse code on the GPIO pins, and finally found a very helpful GitHub repository (<https://github.com/fabiensonpar/Morse_code_RPi>). From this code, I made a C program that fit the project, and was quite happy.

By this time, we had begun learning MIPS assembly. However, there is no way for MIPS assembly to be ran on a Raspberry Pi (at least as far as my research got me). This meant I would have to dive into ARM assembly. There were quite a few helpful references that I found, but one was the most helpful. The website I found (<http://cs.smith.edu/dftwiki/index.php/Tutorial:_Assembly_Language_with_the_Raspberry_Pi>) contained several C and ARM programs meant for the Raspberry Pi. There was one useful one in particular called blink.s. This program made a LED blink rapidly using wiringPi in assembly. This was an excellent find for my project as I was having difficulties implementing system(“gpio write 7 1”) from my C program into assembly. While this was a great breakthrough, I still had a bit of coding to go. Instead of just writing a bunch of on and off commands, I wanted to make my code easily changeable. I wanted to add functions that I could jump to dependent on if I wanted a dot, a dash, or a space. This would make the program versatile and not be specific to my name.

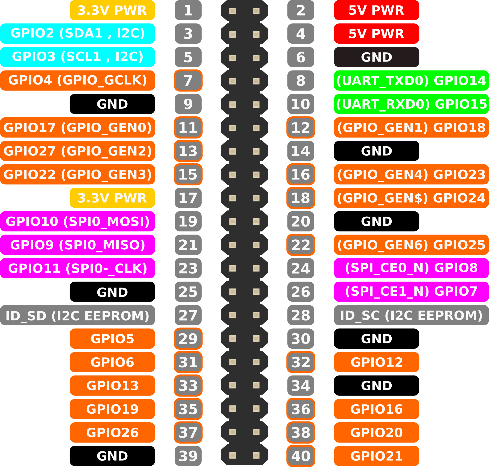
**Program/Implementation:**

Beginning implementation of the solution was quite simplified by the blink.s program mentioned above. However, with my desire to implement functions to blink the dot or dash patterns required additional research before my code was functional. In MIPS, jumps and links are quite easy to implement, but in ARM they can be a bit confusing when compared to MIPS. Instead of saving the return address before jumping, you save it after the jump to another function has occurred. I found this out from this website: <https://community.arm.com/processors/b/blog/posts/how-to-call-a-function-from-arm-assembler>. This website contained an example of code jumping and was quite helpful to figure out how to jump and link in ARM.  
  
 After this there was a couple of things I did not understand about the blink.s program. For some reason the program contained the line “ldr r0, [r0]” throughout the program. I did not know why this was needed, and tried to remove it which caused errors in my code. The rundown of it is that the line before this (“ldr r0, =pin”) loads the object’s memory address. From here the line “ldr r0, [r0]” reads the word from the memory address that is held in r0. I found this out by searching why it was necessary and ran into this link: <https://electronics.stackexchange.com/questions/214318/is-it-necessary-for-ldr-r0-r0-when-saving-from-r15-in-arm>.

The next part I did not understand was using .extern printf and the other .extern commands. I did not just want to trust this random program I found online, so I started research into using them. These two websites came up: <https://stackoverflow.com/questions/40442133/how-to-use-printf-in-raspberry-pi-assembly-language> and <https://www.kevinhooke.com/2015/05/20/calling-c-printf-from-arm-asm/> . These websites explained the printf function only, but did so very clearly so I felt I understood how to use them. To use them you save all the information you need to the registers, and then when you jump to the premade external function, the program will use these registers to accomplish its purpose. This made sense as on the websites <http://wiringpi.com/reference/setup/> and <http://wiringpi.com/reference/core-functions/> we can see where the remaining .extern functions come from.



The above is a picture of the project completed and hooked up to the Raspberry Pi GPIO pins. Currently it is hooked up to pin 6 (Ground) and pin 7 (Power). The pin specifications were an easy google search which is where this image popped up.



With this I would be able to hook it up onto any of the pins labeled GPIO and I could provide power without executing the code by hooking it into pin 1.

**Conclusion:**

This project, while frustrating at times, was easily my favorite for the semester. Not only did it give me an excuse to spend $40 on a Raspberry Pi, but it also allowed me to experiment with an assembly language outside of MIPS. The C program I created proved to be semi useful in creating an ARM program, but it also confused me greatly. This is because I tried to cheat a little by compiling C into assembly. This obviously did not end well, and was probably more harmful than helpful. After these attempt, I gave up on the project for a while. I would google morse code on the Raspberry Pi occasionally, but I had no way of creating a working program. This is where the blink.s program saved me. The blink program was quite useful in figuring out how to turn on and off the LED using ARM. I wanted to repurpose this program and make the code more universal and easy to change in the future. I would love to continue this project and even ask for user input, either through keyboard or even a button on the breadboard. Thinking back on the project, I probably could have used the system(“gpio set 7 1”) by using .extern system, but I did not think of this until I was already done with the code. I am very happy with how this turned out. After the demonstration in class, I modified my code quite a bit. I changed the timing between dots and dashes, as well as the spacing between letters. I also added in a function that would perform the sleep operation when jumped to. After this, the code worked fantastic and I did not see any more room for improvement. The current issue with the code is simply compiling. Someone who does not know that the argument -lwiringPi must be added to the gcc lines will not be able to get the code to function. I listed the compilation steps in the code on lines 13 to 16. Overall, I am really happy that I was able to complete this project, and feel greatly accomplished that I was able to get a finished working product.