SWE4211 Lab 2: Basic IO with the Raspberry Pi

1 Introduction

In traditional software development, the canonical "Hello, world" program serves as a rite of passage—an initial proof of concept that confirms toolchain setup and basic output functionality. While symbolic, it's rarely representative of real-world system behavior.

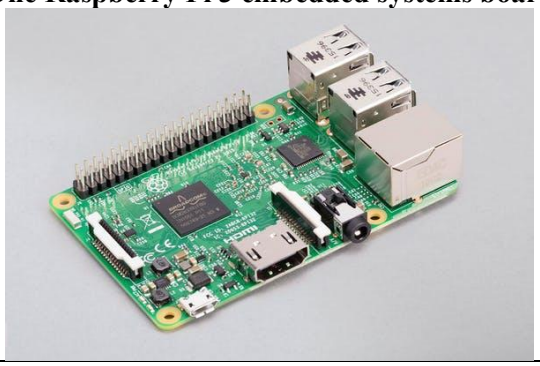
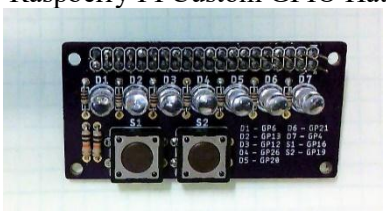
Embedded systems present a different challenge. Text output is often impractical or unavailable, so we substitute blinking LEDs as our first functional milestone. This isn't just a pedagogical workaround—it mirrors production realities. Many devices rely on visual indicators like blinking lights to signal operational status, fault conditions, or mode transitions.

In this lab, you'll examine two core aspects of embedded I/O: basic read/write operations and the mechanics of cross-compilation. Your culminating task will be to implement a general-purpose I/O routine using polling. You'll then analyze its impact on CPU utilization, considering trade-offs in responsiveness, efficiency, and scalability.

2 Lab Objectives

- Describe the role of device drivers in enabling low-level I/O operations within Embedded Linux systems.
- Develop and deploy an Embedded Linux application that performs direct I/O operations to read from and write to hardware output pins.
- Configure and utilize CMake and Eclipse to cross-compile, link, and debug embedded programs for target hardware platforms.
- Implement GPIO-based control logic to manipulate the state of other GPIO pins through software.
- Monitor and analyze real-time CPU usage of running programs using the htop utility.

3 Laboratory Equipment Needed

One Raspberry Pi 3 embedded systems board 	**One Raspberry Pi Power Supply** 
2 Ethernet networking cables 	1 Raspberry PI Custom GPIO Hats 
1 Trendnet 2.0 USB Ethernet 10/100 Adapter 	

4 Deliverables / Submission

Each team must ensure that their GitHub Classroom repository is fully updated with all relevant source code, documentation, and supporting materials. All changes must be committed and pushed to the remote repository before submission.

Each lab group will submit one PDF report to Canvas, containing the following:

1) Title Page
   a) Assignment Name
   b) Submission Date
   c) GitHub Classroom Repository URL (Please include the full URL as plain text—do not embed hyperlinks.)
   d) Demo Video URL (Include the full URL to your video demonstrating the controlled LED. Again, no embedded hyperlinks.)
2) Introduction
   a) Clearly explain the purpose and objectives of the lab.
   b) Use multiple, grammatically correct sentences written in your own words.
   c) Describe what the lab is intended to demonstrate or explore.

3) Controlled LED Experiment
   a) Raw Data Present the data you collected during the experiment. Ensure all units are correct and tables are clearly formatted.
   b) Data Visualizations Include plots showing:
      i) CPU usage vs. blink rate
      ii) CPU usage vs. sampling period
   c) Language Comparison Based on your experimental results, discuss which programming language appears more suitable for real-time systems. Support your conclusion with the data you collected.
   d) Latency Analysis From a theoretical perspective:
      i) Estimate the minimum and maximum latency expected based on the sampling period.
      ii) Calculate the average latency and justify your reasoning using system characteristics.
4) Reflection: Successes and Challenges
   a) What Went Well Describe aspects of the experiment that worked as expected or exceeded expectations.
   b) What Went Wrong Identify any issues, unexpected behaviors, or difficulties encountered during the lab.
5) Conclusions
   a) Summarize what you learned from the lab experience.
   b) Suggest improvements for future iterations of the lab—whether in setup, methodology, or tooling.

In addition, each team should submit a bruief demo video demonstrating
1) The LED blinking on your device
2) Your ability to control the LED through software

If you have any questions or need clarification, please reach out to your instructor.

5 Setting up our hardware using a GPIO Hat

The next part of our exercise involves getting the hardware setup to allow GPIO. GPIO stands for General Purpose Input and Output. Basically, each pin on the raspberry PI can be an input, output, or may have some advanced functionality assigned to it. We want to start by writing basic programs that will allow us to do general purpose input and output.

If using the kit from ECBE technical support, you can skip many of the following steps, as your board is already configured. But, if you are using a separate Raspberry Pi, you will need to plug in the HAT. In raspberry pi terminology, a HAT is a plug-in peripheral that expends the base capability of the device. In our case, the hat has 8 light emitting diodes (LEDs) on it that can be used to display status and 2 push buttons that can be used to control the behavior of the system. The hat also passes through the pins, allowing other hats to be stacked as well as engineers to probe the pins using development tools, such as an oscilloscope or logic analyzer.

The hat itself has been designed to hang outside of the board space, allowing for other boards to be stacked. To use the hat, simply plug the hat into the board, as is shown in Figure 1. This should be done with the power turned off to avoid damaging your Raspberry Pi.

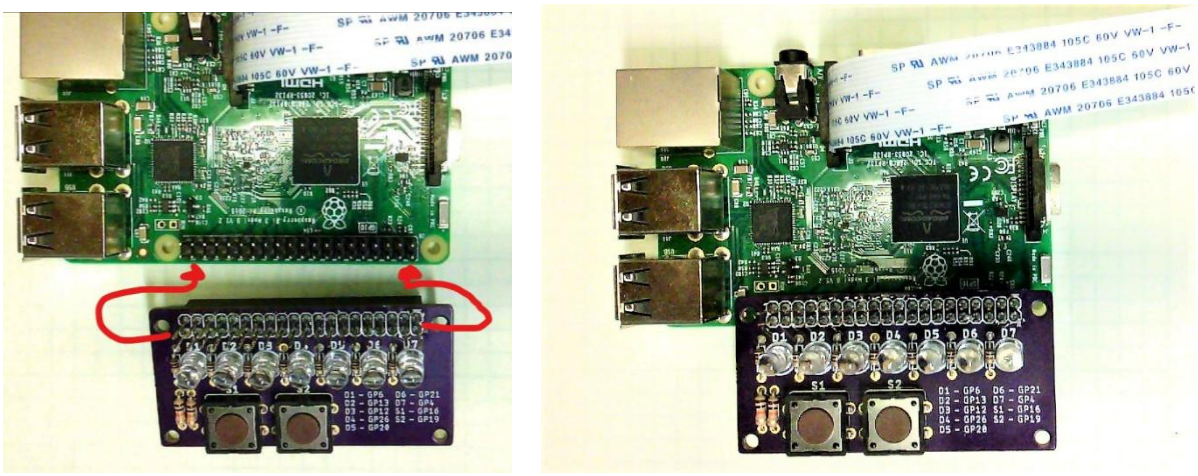


Figure 1 Raspberry PI Custom GPIO HAT. (Left) Prior to attachment to the Raspberry Pi. (Right) After attaching to the Raspberry Pi. (Note: The color of the board that you have may be slightly different.)

Once this step is completed, plug in a USB to ethernet adapter on your laptop and connect to the segregated network in the lab. Connect a second ethernet cable from the Raspberry Pi to the network. Then plug the power supply into boot the Raspberry Pi.

6 Installing the GPIO Libraries

One of the first things we need to do is to install a library on our Raspberry PI devices that will allow us to control GPIO pins. We are going to use a library called wiringPi (https://github.com/wiringpi/wiringpi). This library is an extensive library that supports the Pi and is very capable. The documentation on the website is quite extensive. However, for the most part, we will be using a set of C++ wrapper classes to interface with the wiring pi libraries, as the libraries are pure C and do not support object-oriented design approaches.

To set this up, we first need to install git on the pi. After connecting to the pi via ssh, issue the command:

```
sudo apt-get update
sudo apt-get install git
```

After this completes, you can now directly clone to the pi. We normally will not be doing too much in the way of building on the pi, as the pi is extremely slow to compile the more complex code we will be working with in the near future, but we do have the capability. This part is just easier to do on the pi.

```
mkdir temp
cd temp
git clone https://github.com/WiringPi/WiringPi.git
cd WiringPi
./build
```

When finished, issue the commands:

```
cd ~
gpio -v
gpio readall
```

This will verify the installation as well as give you the status of all the GPIO pins on the Raspberry Pi. The mode of "IN" means that the pin is configured as an input pin. OUT would indicate the pin is configured as an output pin. The column labeled BCM matches the GPIO pin number marked on the schematic and board (and the numbers we will use throughout the course.)

7 Controlling a Light

The first exercise that we want to do is to check and make certain that we can turn on and turn off a light from the console of the Raspberry Pi. The LED we are going to control is attached to GPIO 6, and has the schematic shown in Figure 2. This configuration is referred to as a sinking configuration or "active low" configuration. In a sinking configuration, a device is connected between a power supply and a control pin. Current flows from the power supply into the microcontroller. In our case, this has been done because the Broadcom microcontroller of the Raspberry Pi is not capable of supplying enough current to light all 8 LEDs simultaneously, but it can sink current from all 8 LEDs being illuminated simultaneously.

Given this setup, the LED is referred to as being active low. In other words, if a logic 0 is output on the pin, the LED will be on, and if a logic 1 is output the LED will be off. This takes some effort to get used to but is not that complex if you apply DeMorgan's law from Boolean Algebra to your logic design.

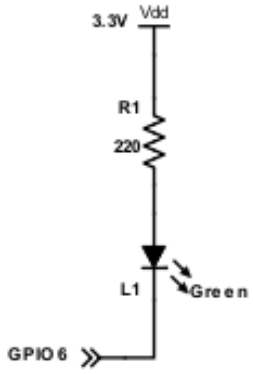


Figure 2 Basic Schematic. GPIO6 is low to turn on the light.

In the terminal, issue the following commands:

```
gpio -g mode 6 out
gpio readall
```

At this point, you should see that pin 6 has been changed into an output mode pin from the previously configured input pin. When looking at the gpio command output, the pin number is in the BCM column. Furthermore, one of the LEDs should have been turned on (i.e. D1 as is labeled on the board). This occurs because of the way our board is wired. Our board is active low, meaning that to turn on an LED, one will write a 0 to the GPIO pin that is connected to the LED. However, by default, GPIO pins using this library have a value of 0 written to them. So, in our case, changing the direction of the pin from being an input to being an output causes the light to turn on.

To turn the light off in this configuration, write a 1 to the GPIO pin using the following commands:

```
gpio -g write 6 1
gpio readall
```

At this point, D1 should have turned off and the value on the pin should change from 0 to 1. The enter the command:

```
gpio -g write 6 0
```

At this point, D1 should have turned back on and the value on the pin should change from 1 to 0.

This will convert pin 6 to an output pin (i.e. a pin which is driven by the logic) read the status, and then turn the light on and off that is connected to PIN 6.

While we will not rountely be controlling the GPIO pins from the console like this, this demonstrates that we can do it simply from the console.

8 The Blinking Light in Python using plain Raspberry Pi Linux

To begin with, we want to blink a light using Python. Your instructor will provide you with a repository that you can clone. Inside of this repository is a python script that will allow you to blink a light. Python, while generally an easy language to learn, is not a very commonly used language for real time systems because it lacks responsiveness. That generally is not good for a real time system.

Before we can run this script, we will need to install some additional support libraries. The wiringPi libraries we installed previously work with C and C++, but do not have a currently supported set of bindings for python. Thus, we will be using a slightly different library when working with Python. To install these libraries, issue the command:
`sudo apt install python3-lgpio`

The script takes three parameters. The first parameter is the number of the port that is to be used. In our case, the number is 6, as that is how the board was wired. The second parameter is how fast the LED is to blink in blinks per second. For example, entering 1 will cause the LED to be turned on for 500ms and turned off for another 500ms. A value of two would cause 2 blinks per second, with the LED being on for 250ms and off for 250ms. The third parameter is the number of seconds that the blinks should continue before the program exits.

Once you have done this, sftp the file up to your Raspberry Pi. Then, open a ssh connection to your pi and issue the command `sudo python blinkLight.py 6 1 10` to make the LED blink for 10 seconds.

Verify that the program works properly by counting the number of blinks in each time period. For example, over 10 seconds, the program should blink 10 times if the rate is set to once per second. It should blink 20 times if set to twice per second.

With a blinking light, we now want to collect some performance data on the system. Open a second window to the pi and issue the command `htop`.[1] This will start the htop tool, which shows CPU usage for each task. Run the program with for a single blink each second. Record the cpu usage that is shown. As you did in operating systems, you may find it useful to filter the output. A sample is shown in Figure 3. Due to the way htop works, the CPU % will likely bounce around a bit. Try to capture the max value you see as it changes. However, for 1 Hz blink rate, this CPU usage should be low.

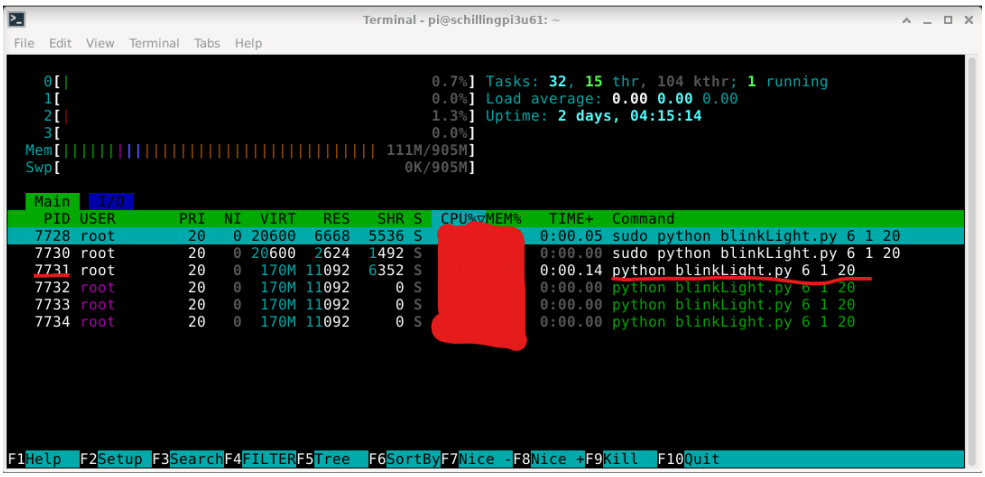


Figure 3 Sampel htop output. Note that we care about the child process that is actually running python, not the processes that may have been forked to create this child process.

[1] If htop is not installed on your pi, install it with `sudo apt-get install htop`.

You also should not the forking of the initial process call. This occurs because of the way sudo works. From the man page for sudo:

> *"If an I/O logging plugin is configured to log terminal I/O, or if the security policy explicitly requests it, a new pseudo-terminal ("pty") is allocated and fork(2) is used to create a second sudo process, referred to as the monitor. The monitor creates a new terminal session with itself as the leader and the pty as its controlling terminal, calls fork(2) again, sets up the execution environment as described above, and then uses the execve(2) system call to run the command in the child process. The monitor exists to relay job control signals between the user's terminal and the pty the command is being run in. This makes it possible to suspend and resume the command normally. Without the monitor, the command would be in what POSIX terms an "orphaned process group" and it would not receive any job control signals from the kernel. When the command exits or is terminated by a signal, the monitor passes the command's exit status to the main sudo process and exits. After receiving the command's exit status, the main sudo process passes the command's exit status to the security policy's close function, as well as the close function of any configured audit plugin, and exits."*

With htop operating, record the CPU usage for python for a 1Hz blink rate, a 5Hz blink rate, and 10Hz blink rate, a 50Hz blink rate, a 100Hz blink rate, a 500Hz blink rate, a 1000Hz blink rate, a 5000Hz blink rate, and a 10000 Hz blink rate. Save this data to a spreadsheet for your report and for usage with next week's lab as well.

9 The Blinking Light in C++

In the repository that you have cloned from github classroom, you will find a piece of code that will blink a given light at a given rate. It is written in C++ and follows the conventions we will be using for much of the class. The repo is setup to use cmake to create an eclipse project, allowing you to compile the code in eclipse. Cmake is a tool that greatly simplifies project configuration. Many of the steps we had to manually complete last week are now automatically handled for us once everything is set up.

To start, open a shell up and browse into your repository. (i.e. ….. /part2/). If you issue the `ls -lR` command, you should see something very similar to that which is shown in Figure 8.

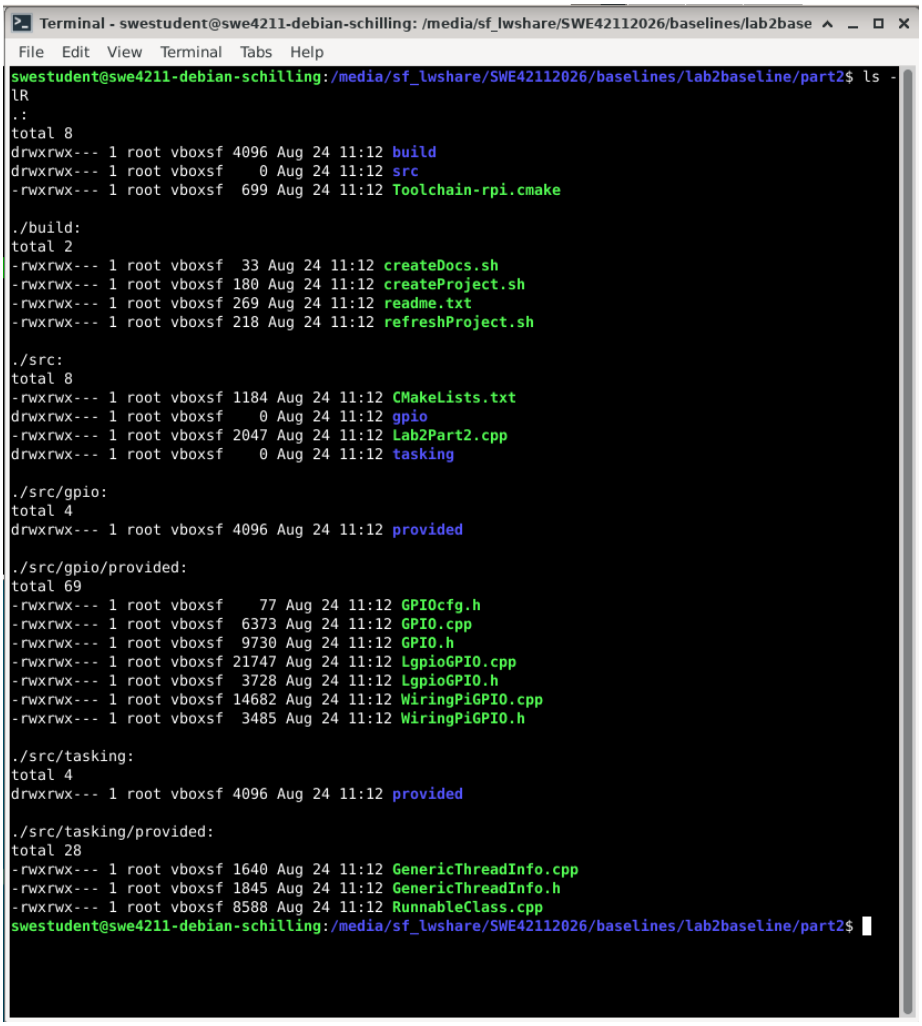


Figure 4 Source directory output. Dates and file sizes may be slightly different.

The Toolchain-rpi.cmake file defines the host and how the cross compiler is to work, creating source code for the Raspberry Pi on the Linux Virtual Machine. The CMakeLists.txt file contains a set of directives and instructions describing the project's source files are to be translated into targets and what dependencies may need to be fulfilled.[2] From this, we will automatically create out makefiles, Eclipse Project, and other parts of the program.

[2] A basic tutorial is available here: https://cmake.org/cmake-tutorial/. CMAKE is extremely prevalent GitLab environment.

9.1 Installing cmake

To make this work, we need to setup cmake. The following describes the purpose for CMake.

> *"CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner. Unlike many cross-platform systems, CMake is designed to be used in conjunction with the native build environment. Simple configuration files placed in each source directory (called CMakeLists.txt files) are used to generate standard build files (e.g., makefiles on Unix and projects/workspaces in Windows MSVC) which are used in the usual way. CMake can generate a native build environment that will compile source code, create libraries, generate wrappers and build executables in arbitrary combinations. CMake supports in-place and out-of-place builds, and can therefore support multiple builds from a single source tree. CMake also supports static and dynamic library builds."*[3]

To do this, issue the command
`sudo apt-get install cmake`

This will install cmake.

9.2 Shortening the compiler name

One of the things that can be a challenge with cross compilation is the length of the names for compilers. We saw in last weeks lab how long the names are for these compilers, and we would like to simplify this. To do this, change into /usr/bin. In this folder, there is a file aarch64-linux-gnu-g++ which is actually a link to the actual binary. We want to simplify this even more, by creating a link with the name arm64-g++ which links to this link.

To do this, change to the path /usr/bin and issue the command
`sudo ln -s aarch64-linux-gnu-g++ arm64-g++`

This will create a link with the name arm64-g++ that links to the compiler. To make everything work properly, repeat the procedure with gcc, creating a link to aarch64-linux-gnu-gcc from arm64-gcc. **If you do not complete this second step, problems will occur later.**

9.3 Creating and working with Projects

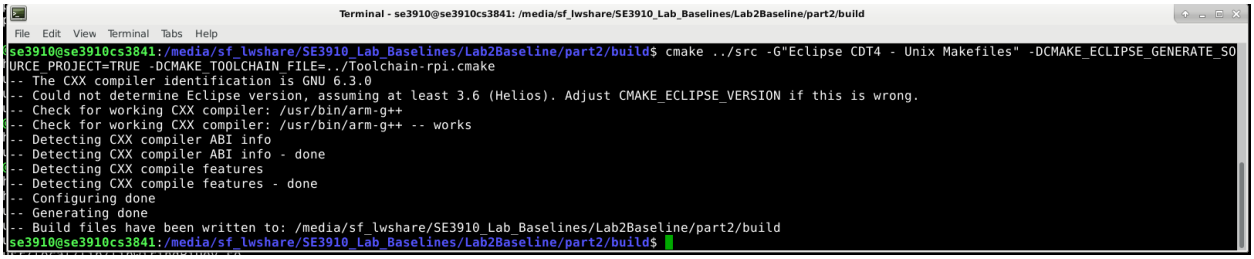


Figure 5 Example output from running cmake.

On the console, change into the "build" directory by issuing the command **`cd build`**. On the command line, issue the command **`cat createProject.sh`**. This will print out the contents of this shell script to the console, and you will see inside of it the cmake commands that will create the projects for us. Two scripts are provided, one to create the project and one to refresh the project if additional source code files are added to the project. The **`createProject.sh`**

[3] https://cmake.org/overview/

shell script will cause cmake to run, using the source (and CMakeLists.txt file) located up one level relative to the current location and in the src folder, generating Eclipse CDT4 project configurations and unix makefiles, and using the cross compiler settings in the file Toolchain-rpi.cmake. This should result in an output like that shown in Figure 5. In the build directory, there should be multiple files that have been created from the codebase, as is shown in Figure 6.[4]

```
swestudent@schillingswe4211vm:/media/sf_lwshare/swe4211AllLabSolutions/Lab2/part2/build$ ls -l
total 50
-rwxrwx--- 1 root vboxsf 27358 Aug 30 12:27 CMakeCache.txt
drwxrwx--- 1 root vboxsf  4096 Aug 30 12:27 CMakeFiles
-rwxrwx--- 1 root vboxsf  1681 Aug 30 12:27 cmake_install.cmake
-rwxrwx--- 1 root vboxsf    31 Aug 30 12:26 createDocs.sh
-rwxrwx--- 1 root vboxsf   175 Aug 30 12:26 createProject.sh
-rwxrwx--- 1 root vboxsf 11299 Aug 30 12:27 Makefile
-rwxrwx--- 1 root vboxsf   213 Aug 30 12:26 refreshProject.sh
swestudent@schillingswe4211vm:/media/sf_lwshare/swe4211AllLabSolutions/Lab2/part2/build$
```

Figure 6 Files created by cmake.

An extra advantage of this step is that it has automatically created our Eclipse project for us, allowing us to compile our code right inside of the Eclipse environment. To do this, startup Eclipse CDT and select a relevant workspace. Typically, the workspace is located one directory level above your projects. For example, in this case, based upon the directory structure (and the fact that there are multiple projects we will be working within inside of the Lab2Baseline directory, the workspace would be Lab2Baseline.

In Eclipse, select File -> Import -> General -> Existing Projects into workspace. After selecting next, browse so that the root directory matches where your project is located and make sure that only the project with the @build is selected. Then click finish. These steps are shown in Figure 7.

[4] Depending on your file system, you may run into an issue with the script. If you did the clone under Windows, you may obtain an output error similar to this: **./createProject.sh: line 2: $'\r': command not found**. This problem occurs because there is a difference in how lines end between Linux and Windows. In essence, your script file is in Windows format but needs to be in Linux for the script to be interpreted correctly. This is a common (and easy) fix. To resolve the problem, issue the command **`sudo apt-get install dos2unix`** to install the dos2unix conversion scripts in your environment. Then issue the command **`dos2unix *.sh`** to convert all the shell scripts from DOS format to UNIX format.

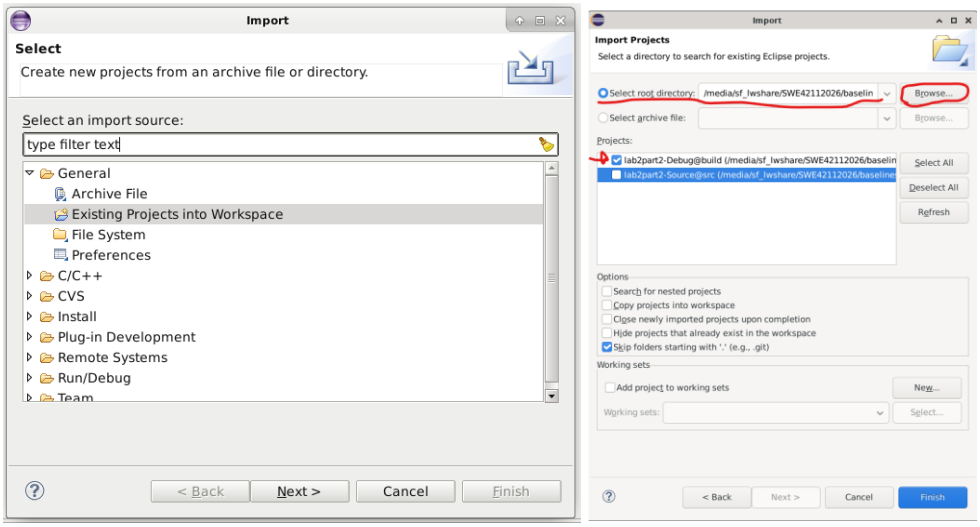


Figure 7 Importing the project into Eclipse.

With the project being successfully imported into Eclipse, you now should be able to browse the codebase as you have done previously as well as build the codebase. To build the codebase, simply select Project and build all, as is shown in Figure 8. Please note that you will get errors when you do a build all right now, as we do not have all of the libraries we need in our VM for the cross compilation to work properly.

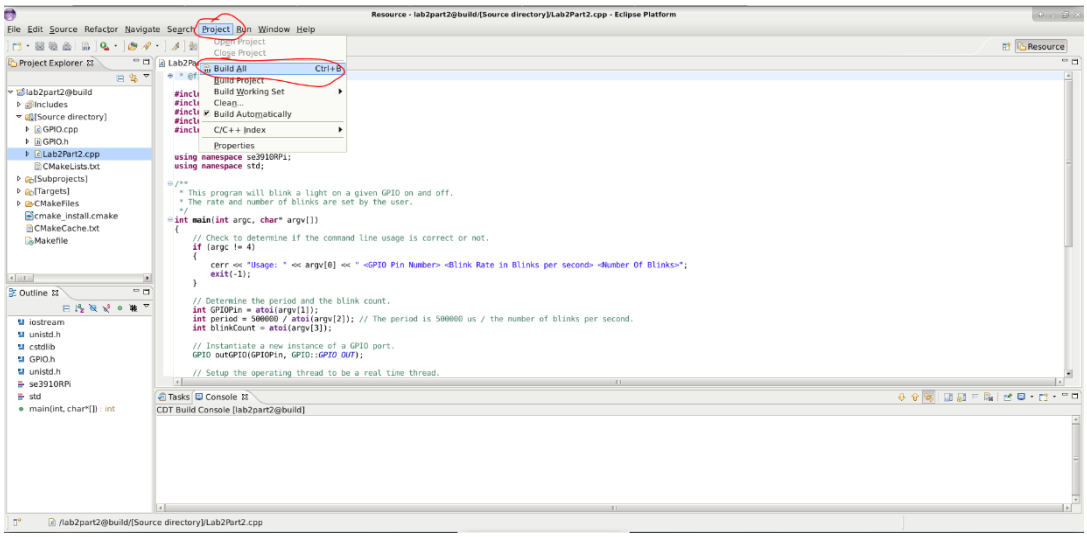


Figure 8 Building the codebase.

The virtual machine may not have all of the libraries that are on the Raspberry Pi. In specific, it does not know anything about Wiring Pi. There are different ways we can solve this problem. We are going to solve this problem by copying parts of the pi's environment back into our virtual machine, allowing us to have exactly the libraries that have been installed on the pi as a part of our development environment.

To do this, we are going to use the **rsync** command. Rsync is a tool that allows one to synchronize two different folders over a network, and it is commonly used in scripts and as part of deployment.

To start, open up a command shell in your VM and issue the following commands[5]

```
sudo apt-get install rsync
cd /
sudo mkdir rpi_sysroot
sudo rsync -azP pi@<yourpi>:/usr/ /rpi_sysroot/usr
```

These commands will copy the environment from your raspberry Pi back to your development environment. You may get an error message from this, as it is not always possible to transfer every file.

Once this is complete, which should take about 5 minutes or so[6], the libraries will be available in your virtual machine. You then should be able to run the code on your Raspberry Pi. Because you will not need to debug this code, it is probably prudent to simply sftp the code to your pi.

However, we do want to make one slight change to the codebase that should not require significant debugging. Add some code so that when the program first starts up, it will print out your name and your partner's name to the console. Now rebuild the code, then push your code from the VM out to the github repository.[7]

For experimental purposes, run this program on the shell. To do this, if you do not already have one, open up an ssh connection to your pi and change to the folder where your code was placed. Issue the command `sudo ./lab2part2 6 1 10` to blink the light once per second for 10 seconds. Repeat the experiment you did with python.

[5] Make sure to replace <yourpi> with the name of your raspberry pi or the ip address of your pi.
[6] When my machine ran, time returned the following:

```
real    4m10.285s
user    0m0.153s
sys     0m1.727s
```

[7] This seems mundane, and it is. The purpose is to make sure you can push to the repo with your lab partner.

10 Working with Rust

Rust is a modern systems programming language designed for performance, safety, and concurrency without sacrificing developer productivity. Unlike C++, Rust enforces strict memory safety through its ownership model, eliminating entire classes of bugs like null pointer dereferencing and data races at compile time. Compared to Python, Rust offers dramatically faster execution and fine-grained control over system resources, making it ideal for embedded systems, game engines, and high-performance applications. Its expressive type system, zero-cost abstractions, and vibrant tooling (like cargo) make it a compelling choice for developers who want both speed and reliability without the pitfalls of manual memory management or runtime overhead.

In this class, we are not going to work extensively with Rust – it is too complex to try and learn in this context – but we are going to do some experimentation with it.

Unlike the C/C++ code we will write, the rust code we are going to work with will be written and compiled directly on the pi, as setting up cross compilation is beyond our scope right now. To do this, issue the command

```
curl https://sh.rustup.rs -sSf | sh
```

This command will provide you with prompts to install rust on your pi. When the install finishes, activate rust with the command
`source $HOME/.cargo/env`.

You should be able to see the info about the rust compiler with the command `rustc -version`.

```
pi@schillingpi3u51:~ $ rustc --version
rustc 1.89.0 (29483883e 2025-08-04)
pi@schillingpi3u51:~ $
```

Figure 9 Checking rust version.

Once you have done this, transfer the tar file from the template up to your Raspberry Pi. This tar file includes the project and the rust code. Extract the tar file as you learned in operating systems (hint: `tar -xvzf filename.tar.gz`). Change into the folder and issue the command `cargo build --release` to build the rust project. This will take roughly a minute the first time, but will be faster afterwards. When finished, run the program with the command `sudo ./target/release/lab2part3`. The parameters are the same as we have used for the other programs. Again, capture the same data.

11 Controlling a Light in Python using Polling

Output is great. However, what we would like to do now is to read input into the system. This is a little bit harder to do. Well, not harder, but just different. With reading input, we need to setup one of the pins as an input. This involves a slightly different circuit. The circuit consists of a pushbutton, resistor, and some wiring.

There are two main ways of connecting in a pushbutton. These are referred to as pull-up and pull-down configurations. In a pull up configuration, a large resistor is connected from the GPIO pin to the power supply. In the case of our Raspberry Pi, this is a 3.3V connection. The switch is connected between here and ground. If the switch is opened, the input pin will assume a logic high, as the resistor will pull the pin to the voltage of the power supply. If the switch is closed (pressed), then the input will go to a logic zero or low value.

In a pull-down configuration, the setup is just the opposite. In a pull-down configuration, a large resistor is connected from the GPIO pin to the ground. The switch is then connected between the supply and the input pin. If the switch is opened, the input pin will assume a logic low, as the resistor will pull the pin to ground. If the switch is closed (pressed), then the input will go to a logic one or high value.

The choice of a pull-up or pull-down setup is dependent upon electrical characteristics of the system. However, we will generally use a pull-up resistor with most of our systems, and this is how our expansion board has been designed.

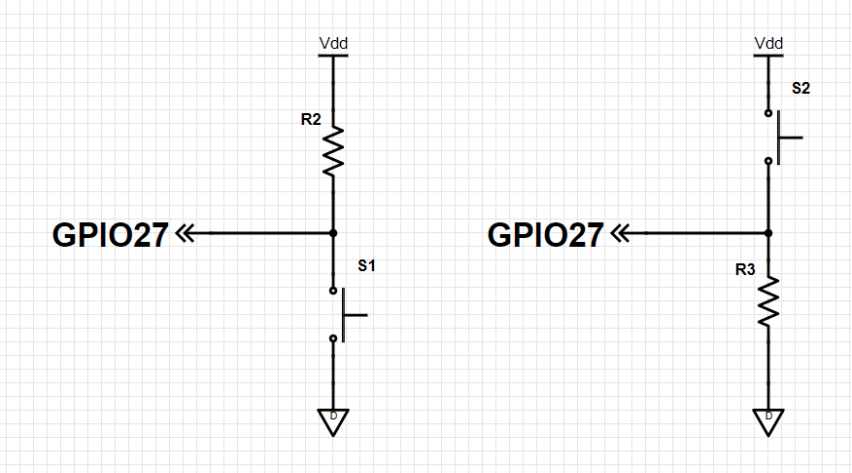


Figure 10 Pull up (Left) and Pull down (Right) circuit configurations. The resistor tends to be approximately 10K Ohm.

Start the Python program controlLight.py in the part 3 folder using the command `sudo python controlLight.py 6 16 0`. This program will control the led. When the pushbutton is released, the light will be off. When it is pressed, the light will turn on. The first two parameters are the GPIO pins for the LED and pushbutton. The last is a delay, given in ms. A value of 0 indicates there is no delay at all in the code between input samples. The program will exit by pressing Ctrl-C. Verify that the program works.

Exit out of the python script and restart the script with a 1ms delay. Again, record the CPU usage with htop. Then switch to 5ms delay and repeat, and then to a 10ms delay. Again, record all of these into Excel. Repeat for 50ms, 100ms, 500ms, and 1000ms delay.

12 Controlling a Light in C++

Now that we have mastered blinking a light, you are tasked with building a program to allow a light to be controlled. To do this, you will need to write a main which will have 2 GPIO pins in usage. One GPIO pin will be read from and set as an input. The other pin will control whether the light is on or off. The program you should write should turn LED D2 on when push button S1 is pressed. The command line parameters should match that which was passed in in the python script.

In the repository, there is a template for this code. Comments have been provided. You simply need to do the implementation. When you have finished your implementation, push your code into GitHub classroom. With your mobile device or phone, capture a brief video demonstrating that your code is working properly. When you have captured this, upload it to a sharable location (YouTube, google docs, vidgrid, etc.)

Then repeat the experiment that was done with python, recording the same CPU usage items versus delay.

13 Controlling a light in Rust

As a last aspect of lab, we want to see how rust behaves for controlling a light. To do this, sftp the tar file up to your Raspberry Pi and extract it. Change into the project directory and issue the command `cargo build --release` to build the code. Then run with the command `sudo ./target/release/lab2part6 <LED> <Pushbutton> <Delay>`. Run this with the same setups as before, measuring the CPU usage with each polling delay.

14 Analysis of Data

Once you have captured your data, create a set of plots. On one plot, show the relationship between blink rate in blinks per second and CPU usage, while on the other plot show the relationship between polling delay and CPU usage. Make sure your plot is well formatted and uses logarithmic scales if appropriate.

From the experimental data collected and plots, analyze the results.