SWE4211 Lab 3: Raspberry Pi performance analysis

1 Prelab

This lab will give you background on deterministic response, as well as experience working with GPIO on your Raspberry Pi.

This lab will use the Diligent Analog Discovery board to measure digital signal performance. To do this, you will need to install Lab software on your machine to support the device. Before the lab, download and install the Analog Discovery software package from Canvas. If you have difficulty, watch the video on using the oscilloscope to take measurements.

2 Introduction

Last week's lab introduced basic GPIO functionality, including tasks like blinking and controlling an LED. This week, we'll shift our focus to evaluating the Raspberry Pi performance characteristics, particularly in the context of real-time systems.

In real-time computing, deterministic performance is critical. Systems must respond to events within strict timing constraints. Failure to do so can lead to undesirable or even dangerous outcomes. However, we also operate within practical limitations, and it may not always be feasible to improve performance beyond a certain point. No matter how many horses you have pulling a wagon, it is not going to be possible for that wagon to move at the speed of sound.

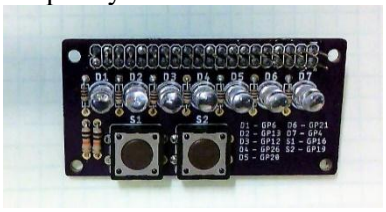
In this lab, you will reuse the code developed in last week's lab. No new code needs to be written. Instead, focus on performance measurement and interpretation. You will need to use a tool usually relegated to electrical and computer engineers, namely an oscilloscope, to make a few rudimentary measurements. In doing this, you will see the real-world ramifications of different performance constraints.

You will also work with a slightly larger group to collect more experimental data in a timely fashion. This lab will essentially result in us collecting data twice with two different settings on the Raspberry Pi. Rather than collecting all the data twice, you will collect half the data and share it with the group adjacent to you at your table.

3 Lab Objectives

- Compare the performance of GPIO operations implemented in Python, native C/C++, and Rust.
- Measure signal latency and analyze how design constraints affect system responsiveness.
- Explain how Embedded Linux can perform low-level input and output through device drivers.

4 Laboratory Equipment Needed

Raspberry embedded systems board 	**Power Supplie** 
2 Ethernet networking cables 	Raspberry PI Custom GPIO Hats 
1 Trendnet 2.0 USB Ethernet 10/100 Adapter 	One Diligent Analog Discovery Oscilloscope 

5 Deliverables / Submission

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

1) Title Page
   a) Assignment Name
   b) Submission Date
   c) Classroom 50 Repository URL (Please include the full URL as plain text—do not embed 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) Blinking Lights Tests
   a) Include the raw data that you recorded when testing the ability of the PI to blink the light.
      i) Be sure to include correct units and format the data appropriately.
      ii) You will ultimately have two tables, one for the data you directly collected and one for the data collected by your table mates.
      iii) Be sure to identify whose data you are sharing.
   b) Include any derived data that was calculated based upon the collected data.
      i) Be sure to explain how the calculations were made and ensure that appropriate accuracy is maintained.
   c) Data Visualizations/plots.
      i) Include plots of the data. Plots should be properly formatted and labeled.
         (1) Actual Blink Rate versus desired blink rate
         (2) %error versus desired blink rate
         (3) CPU Usage versus desired blink rate
   d) Based upon the experiment, discuss which language performs better for real-time systems.
      i) Justify the conclusion based upon the data captured and the plots.
4) Controlled LED
   a) Include the raw data that you captured.
      i) Be sure to include correct units and format the data table appropriately.
   b) Data Visualizations
      i) Include plots of the data. Plots should be properly formatted, labeled, etc.
         (1) Max measured latency versus delay in milliseconds
         (2) CPU usage versus delay in milliseconds
   c) Based upon the experiment, discuss the relationship between sampling rate and CPU utilization. Also discuss the relationship between sampling rate and latency. Justify the statements based upon the data captured.
   d) From a theoretical standpoint, can you determine the minimum and maximum latency that should be observed based upon sampling rate? Can you identify what the average latency should be based upon these two items? Explain any differences as to why your data may not match this theoretical set of values.
5) Reflection: Successes and Challenges
   a) What Went Well

   i) Describe aspects of the experiment that worked as expected or exceeded expectations.
b) What Went Wrong
   i) Identify any issues, unexpected behaviors, or difficulties encountered during the lab.

6) 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.

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

6 Setup on the Pi

Before you begin, you need to complete a small setup step on your Raspberry Pi image. The `cpufrequtils` package provides essential tools for managing CPU frequency scaling, which helps balance performance and power efficiency. It includes two key utilities: **`cpupower frequency-info`**, which displays current CPU settings, and **`cpupower frequency-set`**, which allows you to adjust them.

By selecting different frequency governors—such as **`performance`**, **`powersave`**, **`ondemand`**, or **`conservative`** - you can control how dynamically the CPU responds to workload demands. This is particularly important in embedded and real-time systems, where predictable timing and thermal constraints are critical. For instance, the **`performance`** governor maximizes responsiveness, while **`powersave`** reduces energy consumption. These options make **`cpufrequtils`** a valuable tool for tuning system behavior to meet specific application requirements.

To install the package, open a terminal on your Raspberry Pi and run **`sudo apt-get install linux-cpupower`**

7 The Blinking Light

The following section describes how to perform the experiment in Power save mode and then in section 7.5 it explains that you should collect the data again in a different mode. To help save time yet still provide for a meaningful experience, you will work with the other team that is at your table to collect data. One of you will run the steps given below in power-save mode, and one will run them in performance mode. You will then share your data. This means you will only need to collect half of the data that you will be using for your lab write-up. You will, however, still be able to do the full analysis.

7.1 Entering Powersave Mode

To begin with, we want to accurately measure the accuracy of a blinking light using Python in powersave mode. The code you will use is the same as last week. As you recall, the script takes three parameters. The first parameter is the port number to use. 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 turns the LED on for 500ms and 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 blinks before the program exits.

To start, we want to place the Raspberry Pi into power-save mode. This mode causes the CPU to run slightly slower, saving power. If the device is battery-operated, battery lifespan will be extended. To do this, open a shell to your Raspberry Pi and issue the command **`sudo cpupower frequency-set -g powersave`**

The experiment we would like to do is to measure the output frequency / period that we receive and compare it with the frequency / period we desire. To do this, we will want to connect a device called an oscilloscope to the board. An oscilloscope is a device which is used to measure analog signals. We also want to look at CPU utilization, which should have similar results to last week.

For reference, here are the Linux cpufreq governors—the policies that decide what clock speed the CPU runs at.

Governor	What it does	Behavior on load	Typical use
performance	Pins the CPU at scaling_max_freq and leaves it there	Never scales down	Benchmarking, latency-sensitive work, eliminating frequency as a variable in measurements
powersave	Pins the CPU at scaling_min_freq	Never scales up	Thermal or power-limited situations; on the Pi it's a real throttle, not a "smart" mode
ondemand	Samples utilization periodically; jumps straight to max when load crosses a threshold, then steps back down	Aggressive ramp up, gradual ramp down	Default on Raspberry Pi OS; good general-purpose interactive response
conservative	Same sampling idea as ondemand, but steps up gradually instead of jumping to max	Slow ramp up, slow ramp down	Battery-powered or thermally constrained systems where you'd rather not spike
schedutil	Driven by the kernel scheduler's own utilization tracking rather than a separate polling timer	Reacts faster and more accurately than ondemand because it uses data the scheduler already has	Modern default on most distros; the successor to ondemand
userspace	Makes no decisions at all — exposes scaling_setspeed so a program or a person sets the frequency	Whatever you tell it	Manual control, testing specific operating points, custom daemons

7.2 Connecting the Oscilloscope

The Raspberry Pi has the Pinout shown in Figure 1. The Raspberry Pi pinout connects through the HAT, allowing us to attach oscilloscopes and other devices to the pins to monitor their state. For this first exercise, GPIO 6 will blink, turning the LED on and off. To measure the rate at which this occurs, an oscilloscope will be connected such that the signal is connected to Pin 31 and the ground of the oscilloscope is connected to Pin 39.

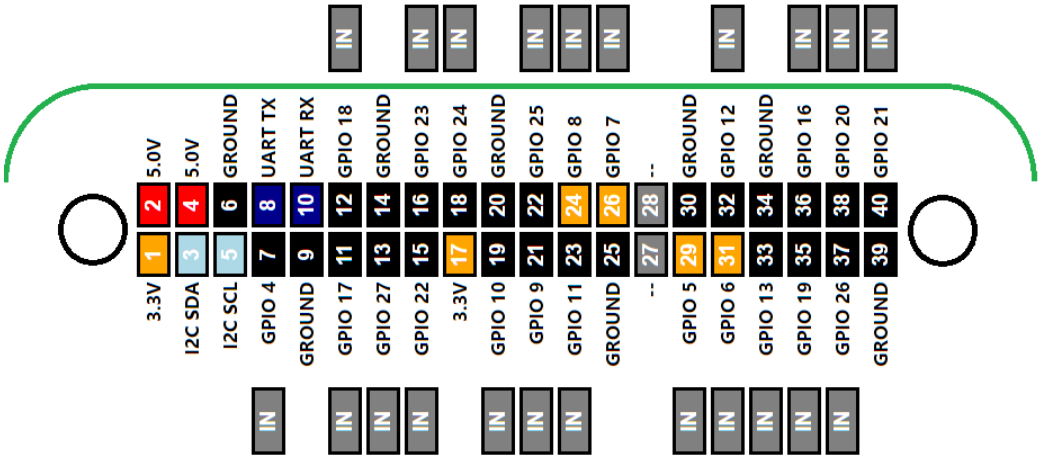


Figure 1 The Raspberry Pi Pinouts.

To achieve this, attach the 2+ (Blue) lead from the oscilloscope to pin 31 and the 2- (also blue but with a white line) lead to pin 39, as is shown in Figure 2.

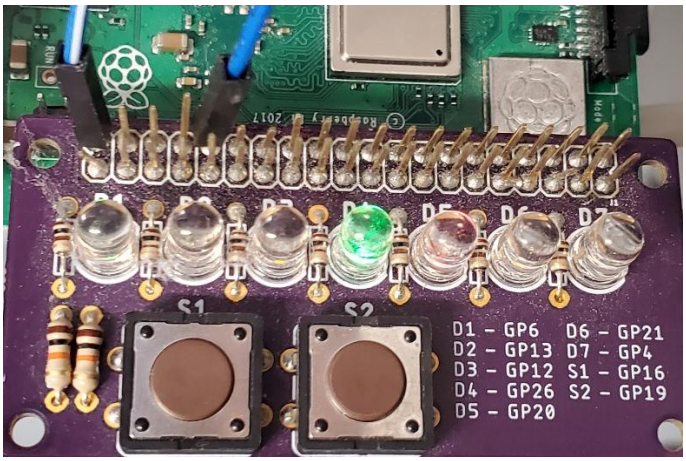


Figure 2 Pinouts and connections for measuring blink rate.

With the analog discovery device attached to your computer via USB, start the Waveforms software. This will give you a screen like that shown in Figure 5. Click on the button that says Scope. This should bring up a screen like that shown in Figure 3. On the screen, click the checkbox beside channel 1 to disable the channel, and in the time area, set the base to between 100ms and 500ms per division. This will make the display such that each horizontal square represents 100ms or 500ms of time. On channel 2, set the range to be 1V per division. Near the top there is a box that says source. Set it to Channel 2, falling, and 2 volts.

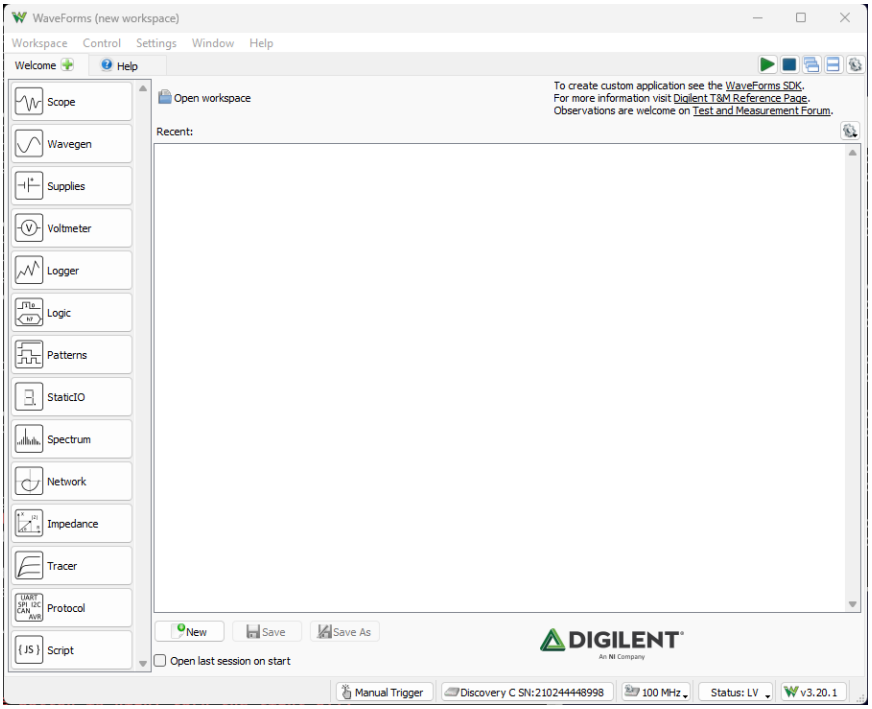


Figure 3 Analog Discovery Waveforms Basic setup.

Enable the measurements view by selecting the Measurements menu. Then add a frequency measurement for channel 2, showing the value and minimum.

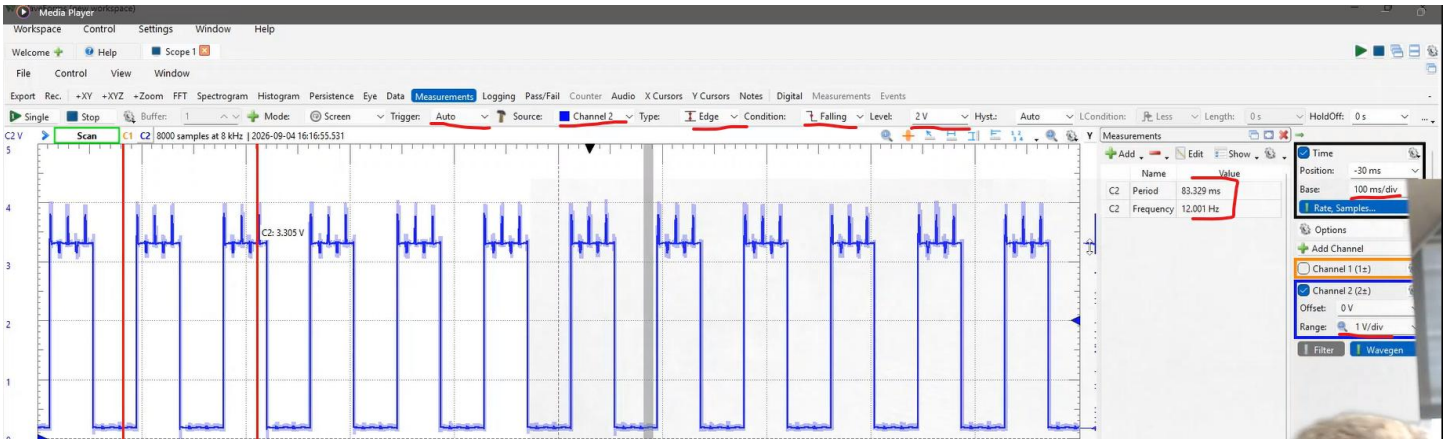


Figure 4 Analog discovery configured with initial signal trace. Note that the "noise" shown in this screen capture may be different depending on your configuration. Note that this frequency is 12 Hz, not the 1Hz you will start with.

Generate a 1 Hz waveform and measure the frequency. To do this, set the first value to 6 for the output port, 1 for the frequency, and an adequate time to measure. Probably 60 will work for the third parameter, representing 60 seconds, but you can increase this if you need more time. In an Excel spreadsheet tab, record the frequency value, typically the average or minimum, and calculate the percent error. If the frequency bounces around a bit, record the lowest frequency you see. You may experiment with the reset button in the measurements area to get more accurate results. In addition, record the CPU utilization with htop for the given task.

After recording the results for 1 Hz, change the frequency to 5 Hz and repeat the same steps. You may need to adjust the horizontal time scale when doing this, as ideally, you want to have 2-5 cycles displaying on the oscilloscope for the measurement to be most accurate. Repeat the process measuring 10 Hz, 50 Hz, and 100 Hz, 500Hz, 1000Hz, 5000Hz, and 10000Hz. Record the data in Excel, as you will be creating plots from it for your report.

7.3 The Blinking Light in C++

Next, we want to repeat the procedure for the light using the C++ code. You should be able to blink the light by running `sudo ./lab2part2 <GPIO PIN> <blink period> <number of times>` So, for example, `sudo ./lab2part2 6 1 50` would cause the LED D5 to blink on and off with a period of 1 second fifty times.

With this being the case, repeat what you did with python, recording the frequency that the light blinks at for flash rates of 1, 5, 10, 50, 100, 500, 1000, 5000, and 10000 times per second. You can record the data on the same table you used for Python providing appropriate headers are given.

7.4 The Blinking Light in Rust

Next, we want to repeat the procedure for the light using the Rust code. You will need to change into the directory where the code has been built to on your pi. You should be able to blink the light by running `sudo ./lab2part3 <GPIO PIN> <blink period> <number of times>` So, for example, `sudo ./lab2part3 6 1 50` would cause the LED D5 to blink on and off with a period of 1 second fifty times.

Repeat what you did with C++, recording the blink frequency for flash rates of 1, 5, 10, 50, 100, 500, 1000, 5000, and 10000 times per second. You can record the data in the same table you used for python and c++ providing appropriate headers are given.

7.5 Performance Mode

When finished recording data with the Rust program, switch the pi into performance mode with the command `sudo cpupower frequency-set -g performance`. This will cause the Raspberry Pi to run faster and, in theory, perform better.

With the Pi running in performance mode, repeat the experiment.

7.6 Data Analysis of blinking lights

This is best done after lab, but it is written here because it is part of the blinking-lights experiment, so you can see what you are doing before you collect different data. From the data collected, we want to determine which language (implementation) best suits a real-time system. To do this, you are going to generate some plots.

On the first plot, plot the actual frequency versus the expected frequency using a logarithmic scale if that makes sense for power-save mode. Ideally, you will have 3 lines, one for each language you used. If the data align perfectly, you will ideally see a straight line. That is unlikely here. You will likely see 3 distinct lines.

On a second plot, plot the percent error in frequency versus the expected frequency. Note that the percent error is to be calculated as:

$$PercentError = \left|\frac{(ActualFrequency - DesiredFrequency)}{DesiredFrequency}\right| \times 100\% \quad \text{(Equation 1)}$$

Again, make sure you plot all 3 data trends on the graph.

On the last graph, plot the CPU usage versus the desired frequency for each of the 3 cases.

You will repeat the generation of these graphs for performance mode.

From these graphs, discuss which language is best suited for real time systems development.

8 The Controlled Light

8.1 Setup

To begin with, we want to accurately measure the latency only in power-save mode. The code you will use is the same as last week. To do this, open a shell on your Raspberry Pi and issue the command `sudo cpupower frequency-set -g powersave`

With this background, we want to set up the GPIO board to measure latency. Latency is essentially the amount of time that it takes from when an action occurs until there is a noticeable response. In this case, we want to measure how long it takes from when the button is pressed until the light changes state.

To do this, we will make a small change to the oscilloscope connections. Connect channel 2 of the scope to the LED (GPIO 6, as before) and channel 1 to the pushbutton (GPIO 16). This is shown in Figure 5.

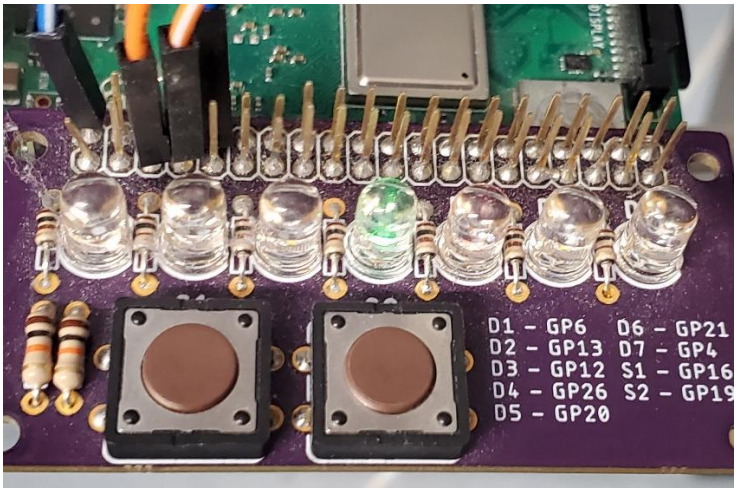


Figure 5 Wiring connections to measure processing latency.

8.2 Python Experimentation

Now that the board is wired, start the Python program controlLight.py in the part 3 folder using the command `sudo python controlLight.py 13 16 0`. This program will control the LED. When you press the push button, the light will be on. When you release it, the light turns off. 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 no delay in the code between input samples. You can exit the program by pressing Ctrl-C.

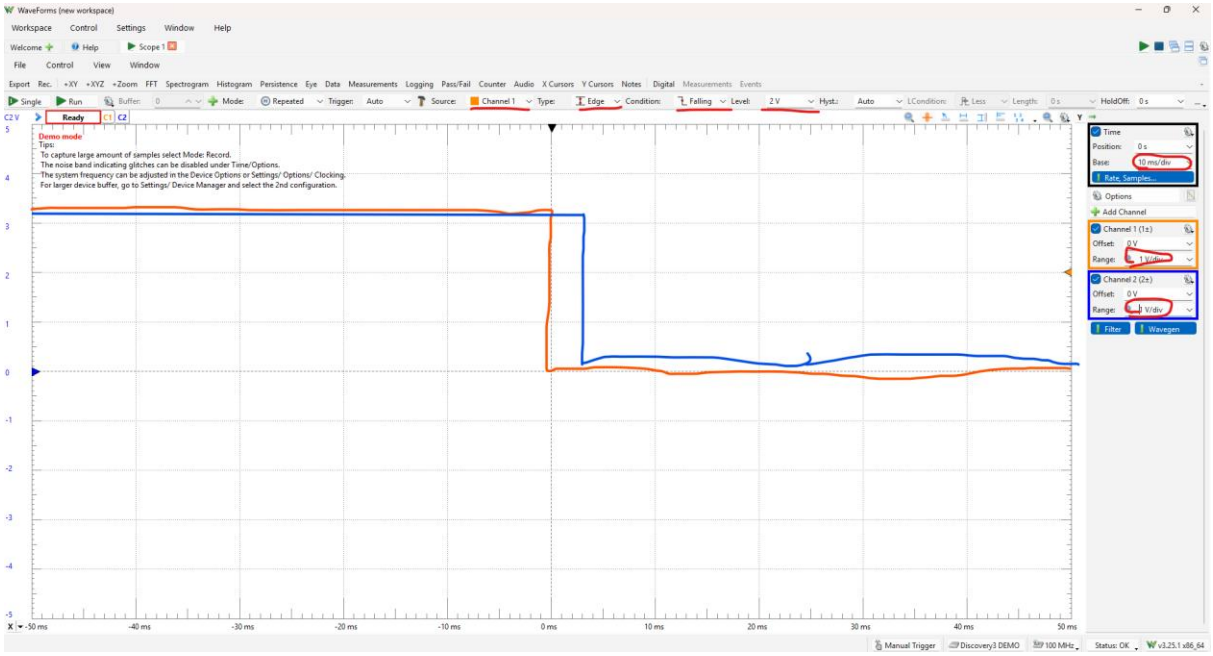
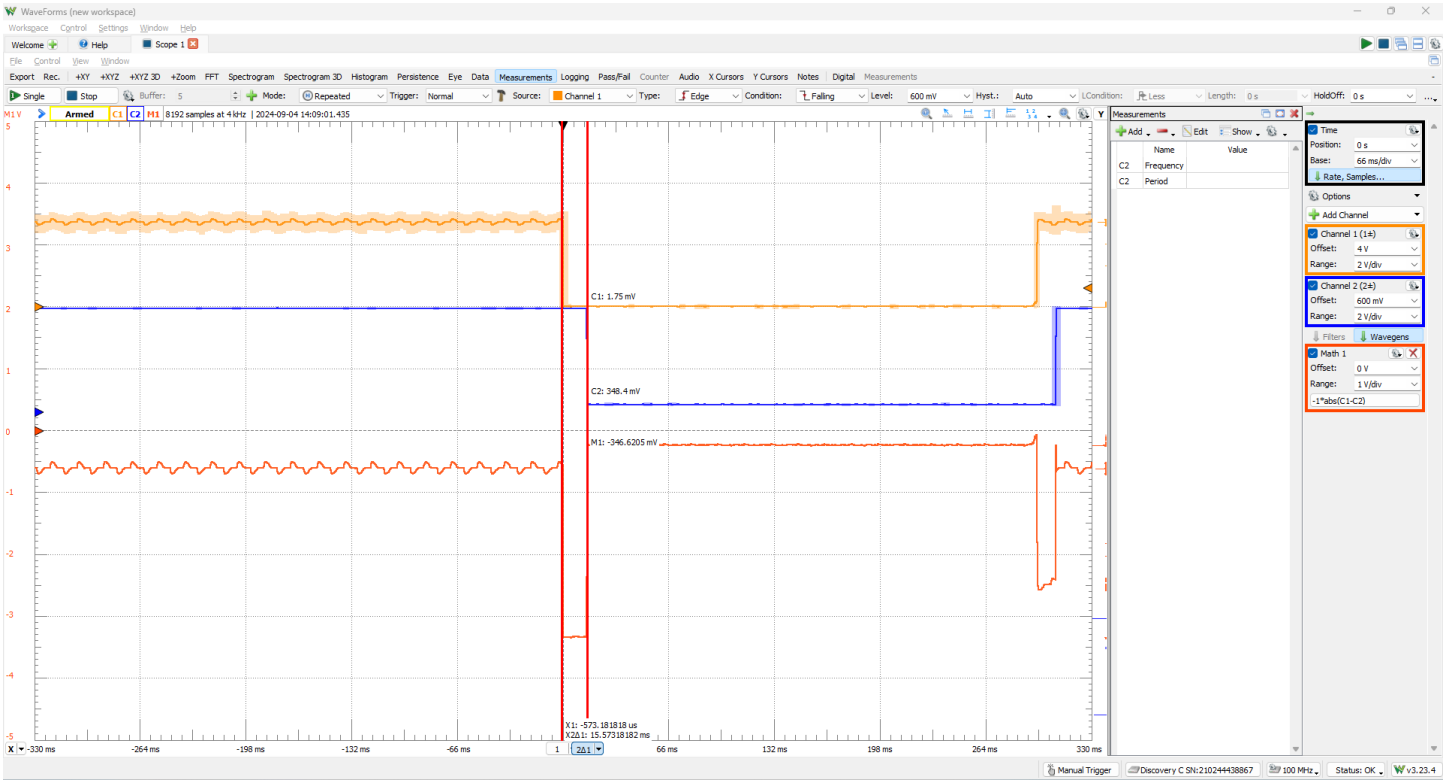


Figure 6 Setting up the oscilloscope to capture latency.

To measure with the oscilloscope, start the Waveforms software (if you exited it previously). Enable both channel 1 and channel 2, set the channel ranges to 2 volts per division, and adjust the base to approximately 100ms per division. This will set the grid up for what you want to see. Next, set channel 1 as the trigger, set the signal to a falling edge, and set the level to 1 to 2 volts. The trigger is the condition that causes the oscilloscope to capture a screen, like the shutter button on a camera. Click on the single button to begin recording the signal. Then press the button. You should get a single scan showing the measurable latency.

Right now, the image will not seem like much. However, when you press the button on your board, the light will change, and you can measure the latency. You may need to adjust the scale slightly and scroll left or right with the mouse. Once you can clearly see the latency, click the X in the lower-right corner. This activates a set of cursors you can use to measure latency. Click once to display the first cursor and click a second time to display the second cursor. You can then move these left or right and align them with the signal transitions to measure latency. Figure 16 shows such a measurement. In this case, the latency is 15.6 milliseconds This is shown on the cursors and can also be determined by looking at the grid. Your results will differ across simulation runs.


WaveForms (new workspace)
Armed
8192 samples at 4 kHz | 2024-09-04 14:09:01.435
C1: 1.75 mV
C2: 348.4 mV
M1: -346.6205 mV
X1: -573.181818 us
XΔ1: 15.57318182 ms
Measurements
Frequency
Period
Channel 1 (1±)
Channel 2 (2±)
Math 1
-1*abs(C1-C2)
Manual Trigger
Status: OK

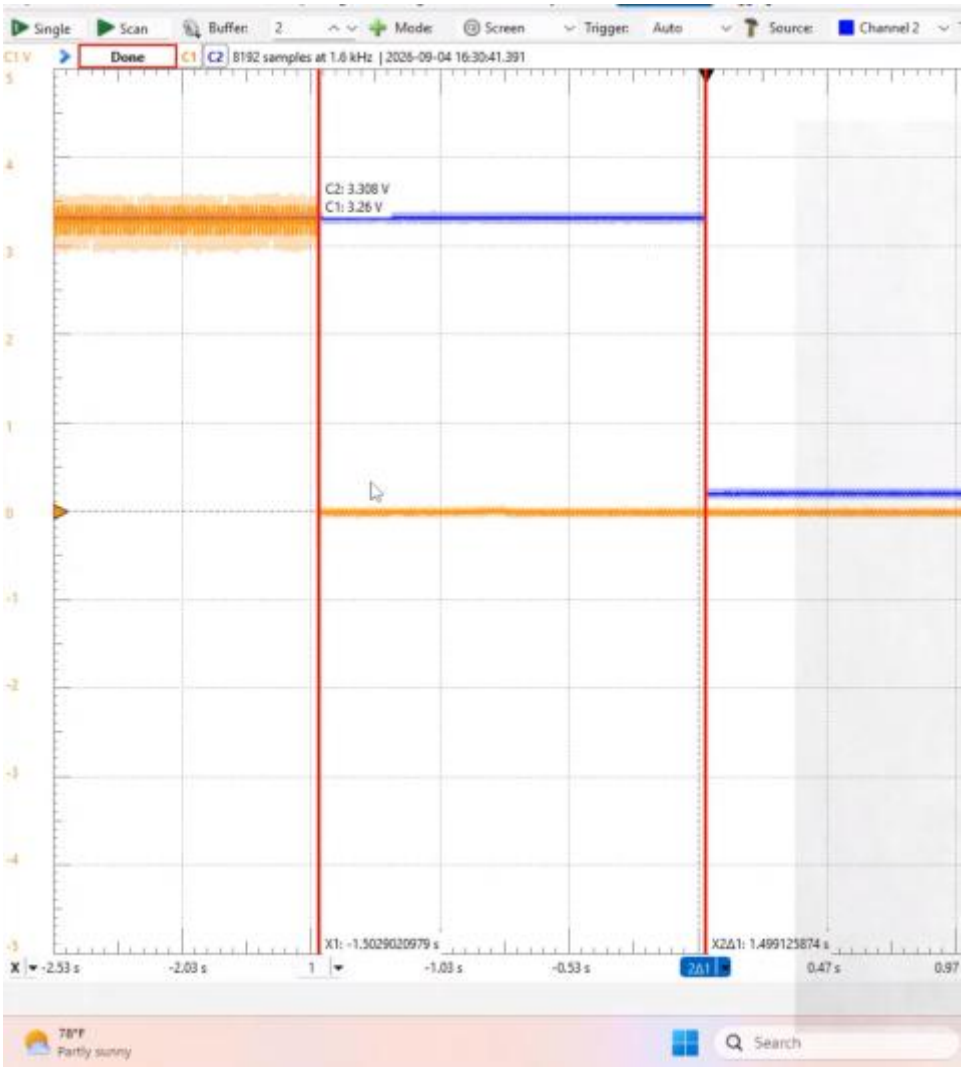


Figure 7 Zoomed-in capture showing latency measurement. Note that this is likely a much longer latency than you will measure.

Measure the latency in your program with the delay set to 0 three different times, recording each one. You will likely get different latencies with multiple tries. That is OK. Ultimately, we will care most about the maximum latency we measure. This one also may be harder to measure, as it is likely the smallest latencies. If you have trouble measuring this one, try starting with a 10ms or 100ms period to get the hang of making measurements and record the data in the appropriate spot, eventually coming back to 0.

Exit the Python script and restart the script with a 10ms delay, repeating the process. You will most likely need to adjust the horizontal scale. Repeat, recording the 3 values you find experimentally. Then switch to a 100 ms delay and repeat. Lastly, record 3 values with a 1000 ms delay. Again, record all of these into Excel.

8.3 Controlling a Light in C++

Repeat the previous experiment with the C++ code you developed to control an LED. Record the system latency in power-saving mode with the delay set to 0, 10, 100, and 1000ms. As with the previous steps, record the data in Excel.

8.4 Controlling the Light with Rust

As you have done before, repeat the measurements with rust.

8.5 Analysis

When finished plot the results for Python, C++ and Rust. Using the maximum latency measured for each of the 3 data points, plot the latency versus the delay for the Python code, the C++ code, and the Rust code. Using last week's data, plot the CPU utilization versus sampling rate as well on a second plot for the values recorded.