Lab Notes

Brendan Chharawala, Owen Moogk, Nirmit Parikh

Session 1

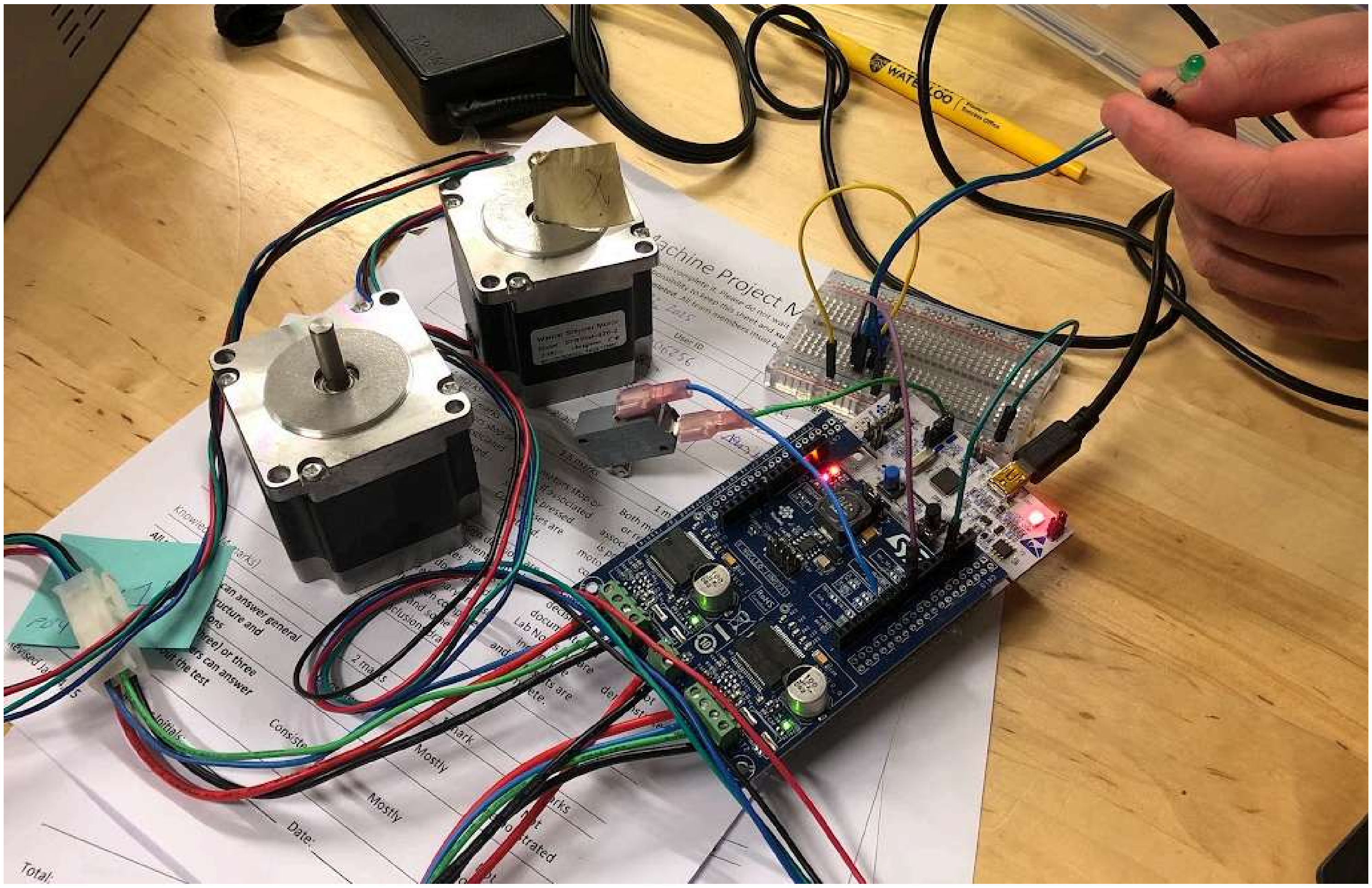
In this session, we're initializing our project with a new git repo, and following the configuration steps.

Getting started

We connected the limit switch to PB4, and ground. Then, we configured it to be a pullup high, and outputted the result to serial. This way we could output the read value and verify that the circuit works.

Next, we used pin PC7 and ground, through a potentiometer, to light up and LED with a GPIO pin.

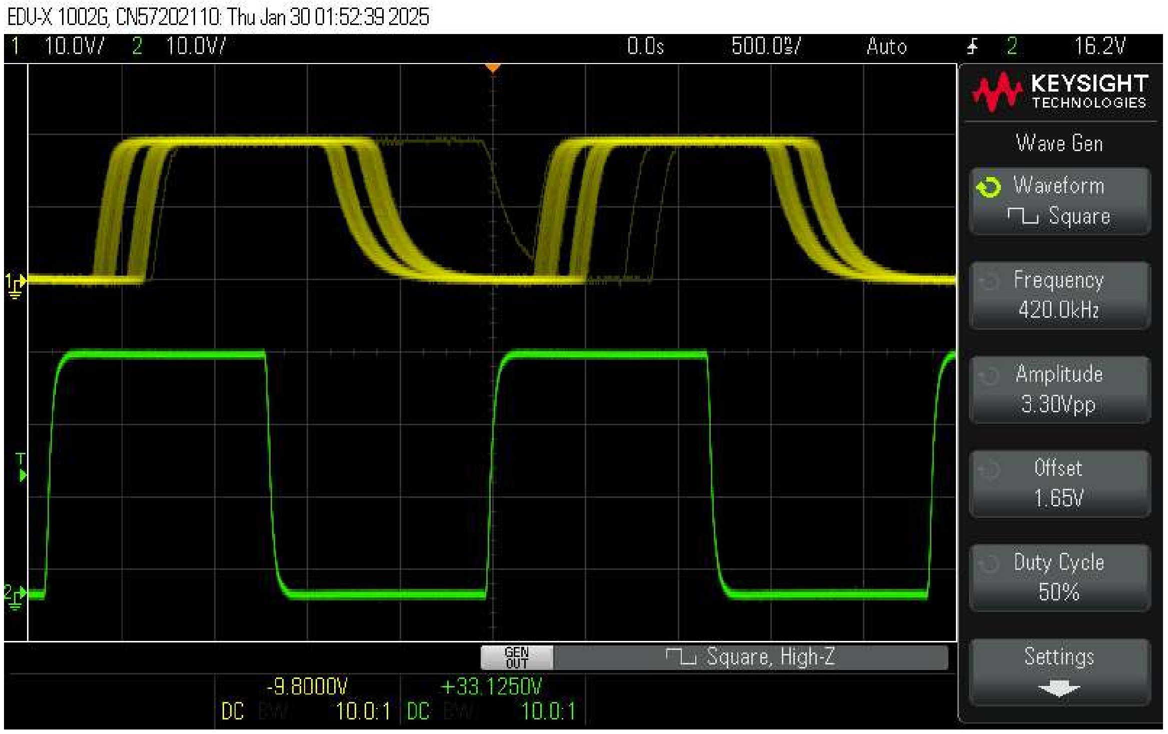
Lastly, we hooked up the motors and got the motors spinning based on the UART commands, or the demo. See below for documentation:



Lab 2

Tight Polling

The procedure used to find the maximum frequency was to increase the frequency until the result became unstable (to a point where it was frequently not turning on and off in time). The maximum frequency square wave that we were able to respond to using polling was 920 kHz. This response was reliable in almost every case, as seen below. However, there was a large amount of variation between the response times, which can also be seen in the yellow curves below.

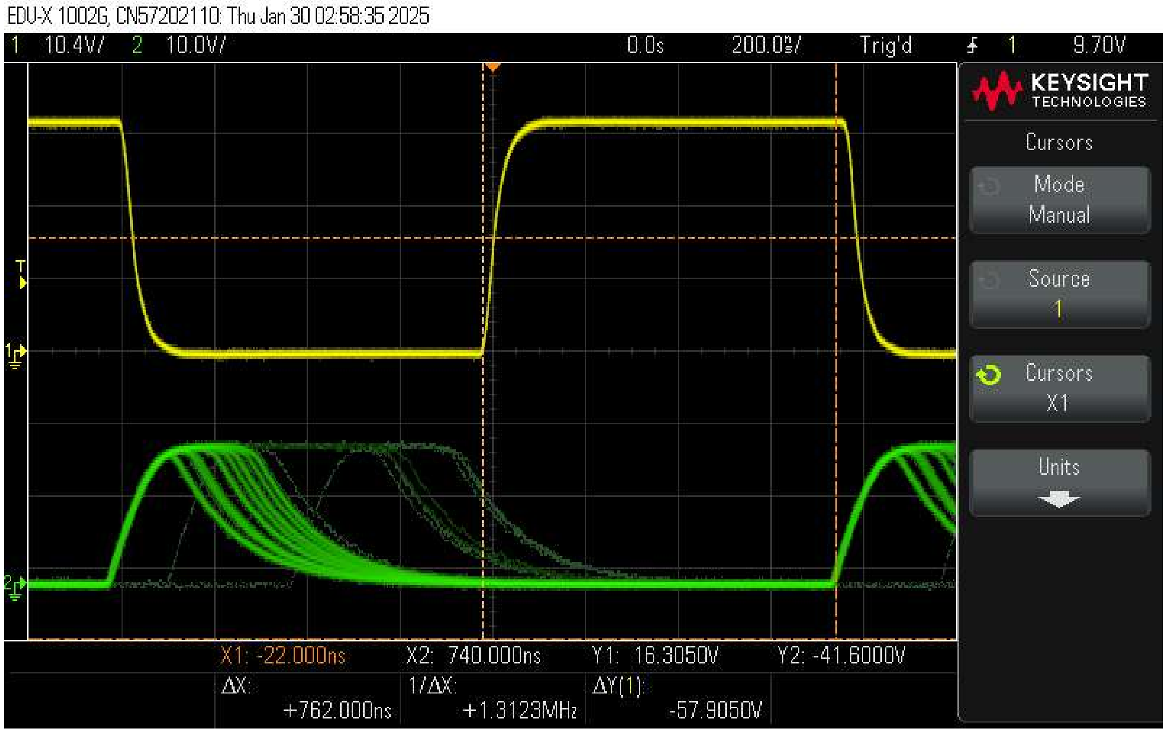


We determined what a reliable response was by checking if the response consistently was able to toggle between on and off. Although there are some very odd edge cases (as can be seen by the ‘ghost’ lines above), it was consistently going to a “high” level, and then returning to a “low” therefore we decided this was acceptable. Increasing the frequency any higher did not allow for this, which is why we decided this was the maximum frequency.

The variation in response time was determined using cursors, measuring the response time of the quickest response, and subtracting it from the slowest response time. The difference in response times is 386 ns, which is likely the time of the while loop that is doing tight polling. This would be caused by the input going high at a different point in this loop, meaning the CPU has a different number of cycles to complete until it recognizes this change and can respond to it.

Interrupts

The procedure to find the maximum frequency was similar to before, increasing the frequency until it was found that the response was no longer turning fully on and off. We determined what a reliable response was with the same criteria, such that almost every wave completes a full on-off cycle in a period. The maximum frequency was found to be 640 Hz. Although most of the time it completed much more quickly than this period, there were a few edge cases where it took much longer (which can be seen below).



The variation in the response time was found to be 240 ns. This is likely due to the context switching that is taking place, which may have varying times depending on where in the program the CPU is. Additionally, depending on what is in the registers, and how many registers are filled, will change how long the context switch takes.

Results

Draw a schematic diagram of your system which includes the port and pin numbers of your input and output connections used for the exercises, as well as the oscilloscope connections. You do not need to include any other pins on the microcontroller or other unused components on the board/shield. Enough detail should be included for someone to connect the hardware required for the experiments without any additional information. You may hand draw the diagram as long as it is tidy and legible. To draw it electronically, you may find a tool such as Visio or SchemeIt (free) useful.

The technique that resulted in the lowest latency was tight polling. This makes sense as the context switch overhead caused by interrupts could be avoided, so there could be a smaller delay between the rising edge and the response. The choice between polling and interrupts is often a matter of trade-offs.

Polling, specifically tight polling, is beneficial for a few reasons. First, because of the simplicity of implementation (ie. just a while loop), there is less complexity and therefore it is less prone to errors caused by abstraction. Moreover, because the CPU cannot do anything else until the loop is finished polling, tight polling will be the fastest option in retrieving data.

Interrupts allow for the system to do other work while it's waiting for an interrupt, which frees up resources for completing other work. Moreover, multiple interrupts can be implemented which allow the system to listen for many inputs simultaneously, while also enforcing different levels of priorities.

Which synchronization technique do you think will be more appropriate for the limit switches on your two-axis machine? Justify your choice.

Interrupts would be more appropriate for the limit switches as high-priority interrupts will guarantee a response window which would be beneficial for safety while simultaneously guaranteeing that the system can do other work. Tight polling wouldn't be an acceptable option, because other work would have to be done by the system while it needs to check the limit switches. However, if it did occasional polling it wouldn't have the same time guarantee, which is an issue for a safety feature. Additionally, adding interrupts would allow the system to be expanded and developed in the future, with increasing scalability.

The latency between when the interrupt is triggered and when it is executed could result in the limit switches not functioning, or responding, as intended. This is critical when an immediate machine stop is required, as any delay in handling the interrupt could prevent the machine from stopping in time. This could be prevented by good system design and properly weighing tradeoffs when making design decisions.

Lab 3: Limit Switch implementation

In this lab, we used machine number 9. We will continue to use this machine for the remainder of the term.

Test cases:

- if the X switches are triggered the X motors should be capable or reversing or stopping
- if the Y switches are triggered the Y motors should be capable or reversing or stopping
- corner case: if the switches are both triggered at a corner the machine should reverse in both directions or stop
- The switches will have bounce, they should be debounced
- If the left and right (or top and bottom) are pressed at the same time, the machine should move in a direction that is safe to do so. (or stop)

Implementation choices:

- The switches were chosen to be normally high and thus triggered on a falling edge with a pulldown resistor. We choose this because if the high lines on any of the switches break we want the system to recognize it is in an undefined state and thus stop as opposed to do nothing.
- For the interrupts we set separate handlers for each switch such that there is a faster response time given that everything has a dedicated handler

Lab 4: ADC Implementation

In this lab, we first setup the debouncing that was required on the interrupts. To do this, we decided to go with the simplest approach with a one-shot timer. After waiting a set amount of time after the edge was detected, we would then read the value and make sure it was the value that we were expecting (in this case, low, after a falling edge). If this condition was true, then we would continue with the interrupt. Otherwise, we would clear the interrupt and continue.

We demo'd this, and then continued to the ADC implementation. We wired it up to pin PB0 using ADC1. Configuring this in code was a bit of a headache, especially since we forgot to call the configuration function (for ~1hr 😅). At the end of this lab we have the ADC configured to what we think is proper, however it's reading all zeros, so it will need to be worked on.

For next time

- We're currently using PA0 for a limit switch interrupt, which will need to be changed as it's used for the motor drivers.
- Fix up the ADC!

Lab 4.1: Working on the ADC

Between lab 4 and 5, we worked on the ADC. The main issue that we were facing with the ADC turned out to be an incorrectly selected channel, which was fixed. We also changed the settings of the ADC configuration to more accurately reflect the sampling and measurements that we wanted to take, which can be seen below:

```
  hadc1.Instance = ADC1;                               // Specifies that we're using ADC1
peripheral
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4; // ADC clock is peripheral clock (PCLK)
divided by 4
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;          // 12-bit resolution (0-4095 range)
  hadc1.Init.ScanConvMode = ENABLE;                    // ADC will convert multiple channels in a
sequence
  hadc1.Init.ContinuousConvMode = DISABLE;             // ADC continuously converts without
stopping
  hadc1.Init.DiscontinuousConvMode = DISABLE;          // Discontinuous mode off (not splitting
conversions into subgroups)
  hadc1.Init.ExternalTrigConvEdge = ADC_SOFTWARE_START; // Using software start
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;    // Data is right-aligned in the register
  hadc1.Init.NbrOfConversion = 2;                // Number of channels to convert in sequence (2
channels)
  hadc1.Init.DMAContinuousRequests = DISABLE;    // DMA not used for continuous requests
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV; // End of Conversion flag set after each single
conversion
```

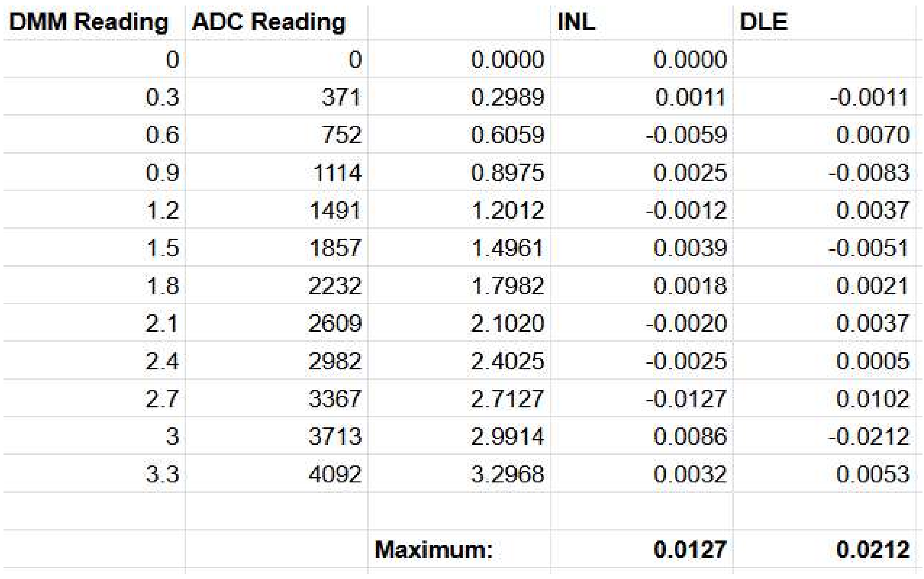
In the next lab, we will work on characterizing the motor control, and ensuring that the readings are linear and the error is minimal (or accounted for).

Lab 5: Completing ADC, starting Motor Control

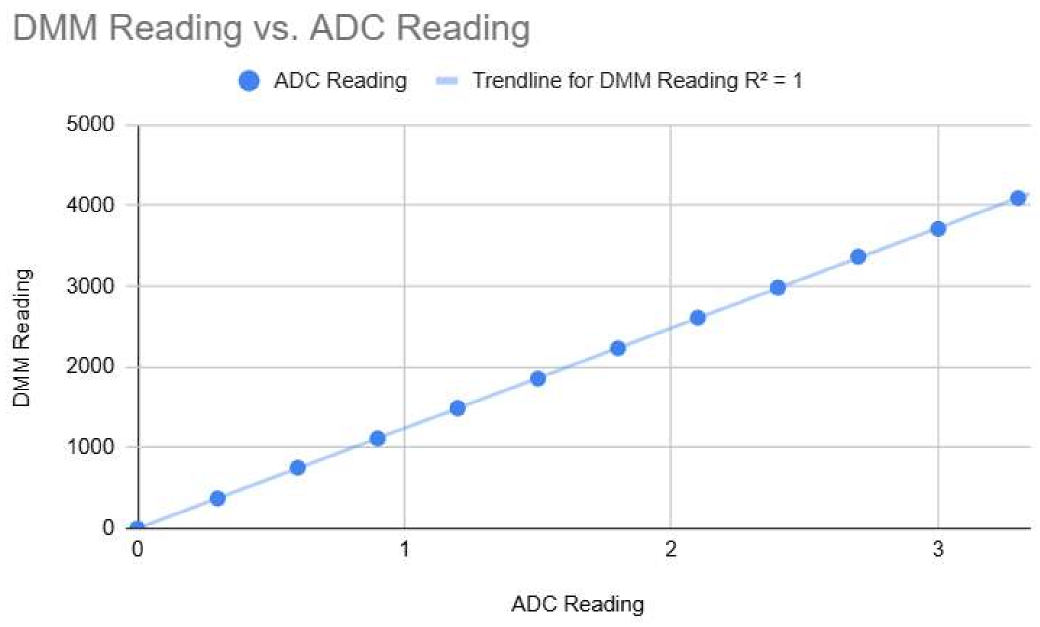
We completed the ADC charactarization, and completed the demo. Now we will impelement motors. This worked fairly well, and we were able to get the motors moving at adequate speeds, and were able to control them well.

ADC Charactarization

The ADC was charactarized by reading the values that were put into it with a multimeter, and then reading the digital output from the ADC. Comparing these two, it was found that the ADC was extremely linear and gave accurate readings, which can be seen from the INL and DLE errors calculated:

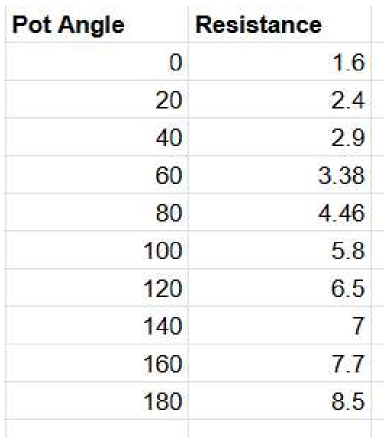


DMM Reading	ADC Reading		INL	DLE
0	0	0.0000	0.0000	
0.3	371	0.2989	0.0011	-0.0011
0.6	752	0.6059	-0.0059	0.0070
0.9	1114	0.8975	0.0025	-0.0083
1.2	1491	1.2012	-0.0012	0.0037
1.5	1857	1.4961	0.0039	-0.0051
1.8	2232	1.7982	0.0018	0.0021
2.1	2609	2.1020	-0.0020	0.0037
2.4	2982	2.4025	-0.0025	0.0005
2.7	3367	2.7127	-0.0127	0.0102
3	3713	2.9914	0.0086	-0.0212
3.3	4092	3.2968	0.0032	0.0053
		Maximum:	**0.0127**	**0.0212**

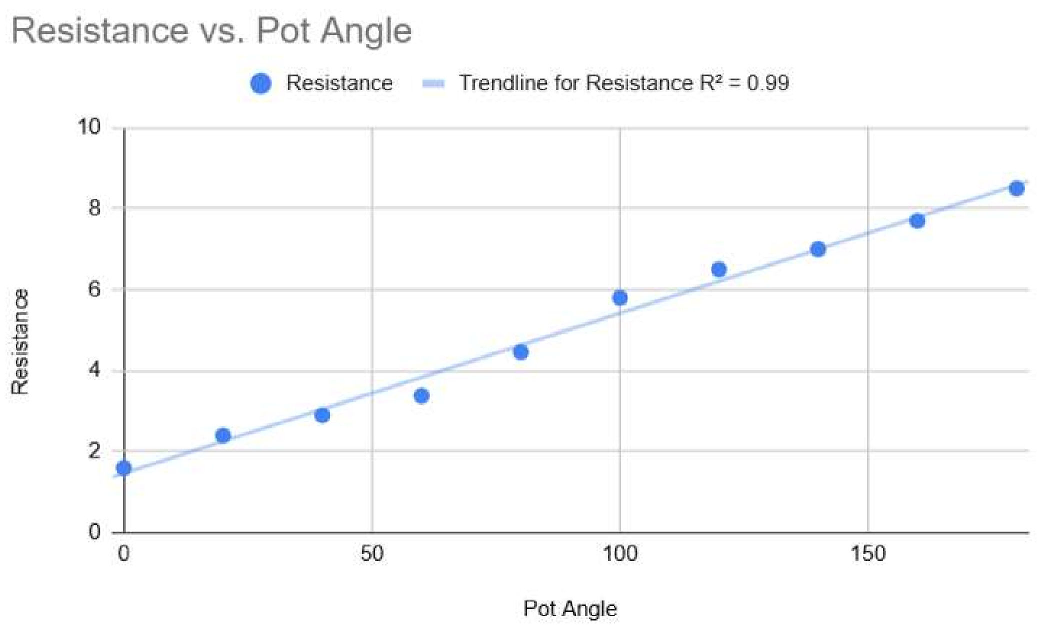


Potentiometer Charactarization

The potententiometer was charactactarized in a similar fashion, reading the input angles and the resistance measured. However, there may be some small error in the measurement of the potentiometer angle, due to a lack of proper measuring equipment. We made this measurement by aligning a protractor to the potentiometer, and then using a screwdriver mounted to a jack-knife to point to the approximate angle. We acknoledge that this is not an ideal or accurate testing setup, however approximate results were all that were needed. The following results were obtained:



Pot Angle	Resistance
0	1.6
20	2.4
40	2.9
60	3.38
80	4.46
100	5.8
120	6.5
140	7
160	7.7
180	8.5



Motor Charactarization

The motors were controlled using a velocity based control. This was decided so that the position of the potentiometer corresponded to a velocity instead of a position. This was decided because we didn't want there to be a situation where the position requested was invalid (if the pot was set all the way to one side, requesting a position that wasn't within the bounds of the machine).

However, this came with a catch. We only updated the motor speeds when the potentiometer position changes, so that the interupts could change the motor speed without being immediately overwritten by the user control. Additionally, we added a "zero" buffer zone in the middle of the potentiometer control, so that the machine would be completely stopped if the control was within a window. This makes it easy for the user to stop the machine, and prevents a situation in which the user thinks the machine is stopped and it is actually moving at an incredibly slow speed, which could have safety implications.

This was done through a few conditional statements, as shwon below:

```
if (abs(adc_value - operating_adc_value) > MOTOR_CHANGE_THRESHOLD)
    {
      operating_adc_value = adc_values[0];
      if (abs(operating_adc_value - ZERO_SPEED_VALUE) < ZERO_SPEED_THRESHOLD)
      {
        L6470_HardStop(L6470_ID(1));
      }

      else
      {
        eL6470_DirId_t direction = 0;
        if ((operating_adc_value - ZERO_SPEED_VALUE) > 0)
        {
          direction = 1;
        }
        L6470_Run(L6470_ID(1), direction, abs((operating_adc_value - ZERO_SPEED_VALUE)) *
Y_SPEED_MULTIPLIER);
      }
    }
```

ADC Document Questions

What is required to turn it on?

This needed to be configured in code, using a configuration function. After the ADC configuration described above, it was initalized with the code below.

ADC_ChannelConfTypeDef channel8; channel8.Channel = ADC_CHANNEL_8; channel8.Rank = 1; channel8.SamplingTime = ADC_SAMPLETIME_3CYCLES; HAL_ADC_ConfigChannel(&hadc1, &channel4); *What resolution did you choose? Why?*

We chose a 12 bit resolution. This was chosen because it provides more than enough resolution for our use case, and is relatively simple to work with. Going larger wouldn't bring any benefits, however going smaller may make the system feel clunky or discrete, which would not be a nice user experience.

What conversion mode? Why?

We are in single-shot conversion mode. This is because we don't really care what the output from the ADC is most of the time, and then we ask for the value each time we go through a while loop and get the value when requested. Although this was slightly slower than using a different mode where the ADC already has a value ready to send to the CPU, this small time difference doesn't matter for this application.

What is your sampling time? Why?

Since we're doing single-shot, we don't have a sampling time. Instead, the ADC is sampling each time our while loop completes, the sampling time is variable. As previously mentioned, although this may not be possible, it's more than good enough.

What happens when you change clock settings?

Again, like mentioned before, this isn't necessary. If we were to do this, it would sample more or less often, up to a limit of the ADC sampling frequency.

How can you ensure your ADC is enabled and working?

We were able to ensure that the ADC was enabled and working because a voltage applied at the ADC was able to be read into a variable in our code.

Ideally your ADC is linear. How accurate is it?

It is very accurate. From the results above, the errors are very small, and using a line of best fit in excel it was found that it has an R-value of 1 (meaning perfectly linear).

What is the error in your ADC?

The INL and DLE are documented above.

POT Document Questions

Does it behave linearly?

Yes, it mostly behaves linearly. It can be seen above that it has an R-value of 0.99, meaning it’s very close to linear. However, the non-linearity here can be chalked up to measurement error, as the accuracy of the potentiometer angle measurement wasn’t great. Since the user will provide feedback into the system (ie. if the motor is moving too fast, the user will slow it down), small errors won’t be a problem.

How can you map the ADC readings to motor speed?

This was discussed above.

Lab 5.1: Reporting, Getting a Second Potentiometer Working

In this time between lab sessions, we spent some time reporting where we were, and getting the ADC working to read two different potentiometers at the same time.