Dylan Falzone
10/2/23
4539-8897

EE Design 1 Technical Report
Microcontroller ADC & LCD

Introduction

What would life be like without any of the five senses? If you could not taste, touch, smell, see or hear, what would you experience? Perhaps you would be able to think, but what would you think about with no way to experience *anything*? Some would consider this a useless existence, though a question like that is better left for philosophy majors. Instead of a human, what if a digital system was inflicted with this sensory void? Now that would be a useless existence.

Our world is analog. It is continuous and vast, and barring any future innovations, the only way to meaningfully interact with it is by interfacing the digital/biological with the analog. In a human, this occurs somewhere between our sensory organs and our brain. In machines, this occurs in the analog to digital converter, or ADC for short. The ADC is what makes digital systems relevant and useful, and in this lab, we will use the Raspberry Pi Pico's ADC to measure a resistor, demonstrating just a microcosm of what the ADC is capable of.

Methods

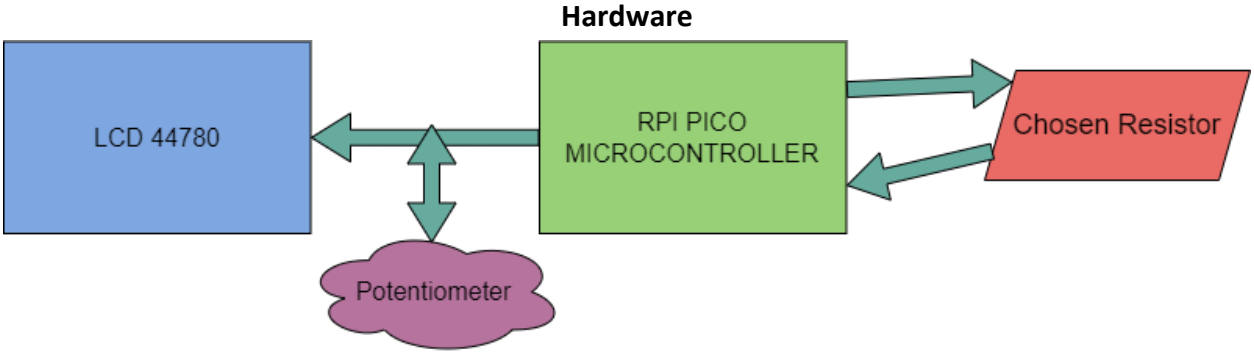


Figure 1: Hardware Block Diagram for the ADC-LCD system.

The basic components of this system included an LCD display, a potentiometer to adjust the contrast on said LCD, the Raspberry Pi Pico microcontroller, and a resistor or load to measure across. The LCD is purely output, the potentiometer takes power from the circuit but also provides an input back into it, and the resistor requires a power and ground connection to the microcontroller, so I also considered it bidirectional.

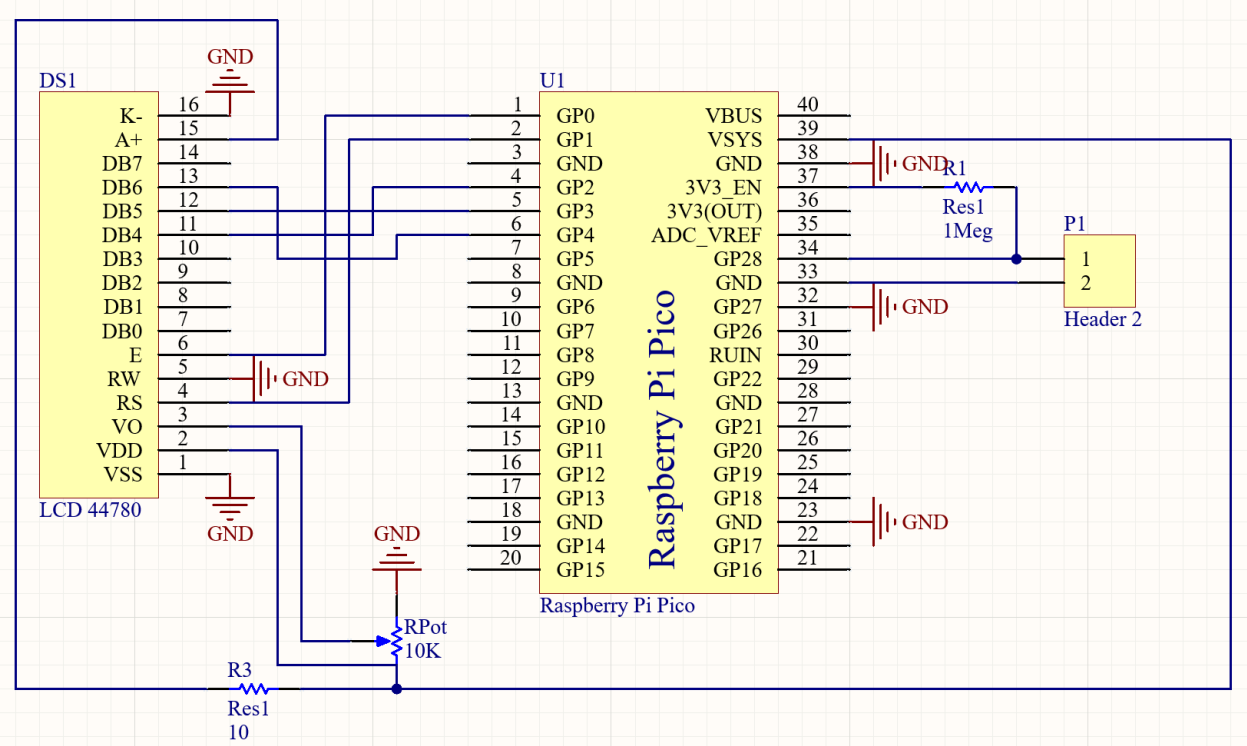


Figure 2: Altium schematic of the ADC-LCD circuit.

In order to measure the value of a resistor, I used a simple voltage divider. I'm sure there are more complicated ways to do this, but I found the voltage divider very effective [1]. As far as the value of R1, that was the major pain point for me in this lab. Initially I used 1k, which worked great until R2 exceeded 600K. At this point, it became incredibly unreliable. This occurred because the voltage differential on the ADC at these higher ranges was extremely compressed due to the nature of the voltage divider. For example, the difference between 800k and 1Meg was only 0.002. Combined with the ADC's limited resolution and the base level of noise in the system, it was impossible to get an accurate reading at high resistances. After two entire days of trying to reduce the noise in the system, I realized that using R1=1Meg would solve my problems. I confirmed this experimentally and mathematically, finding that the voltage differential between the closest low-end resistance values was double that of the previous high-end, which proved to be much more distinguishable to the ADC.

The rest of the circuit is fairly straightforward, mostly consisting of data and instruction lines to the LCD. As previously stated, there is a potentiometer to adjust LCD's contrast, but this was a largely set-it and forget-it component, and with some testing it could be replaced by a simple resistor.

Software

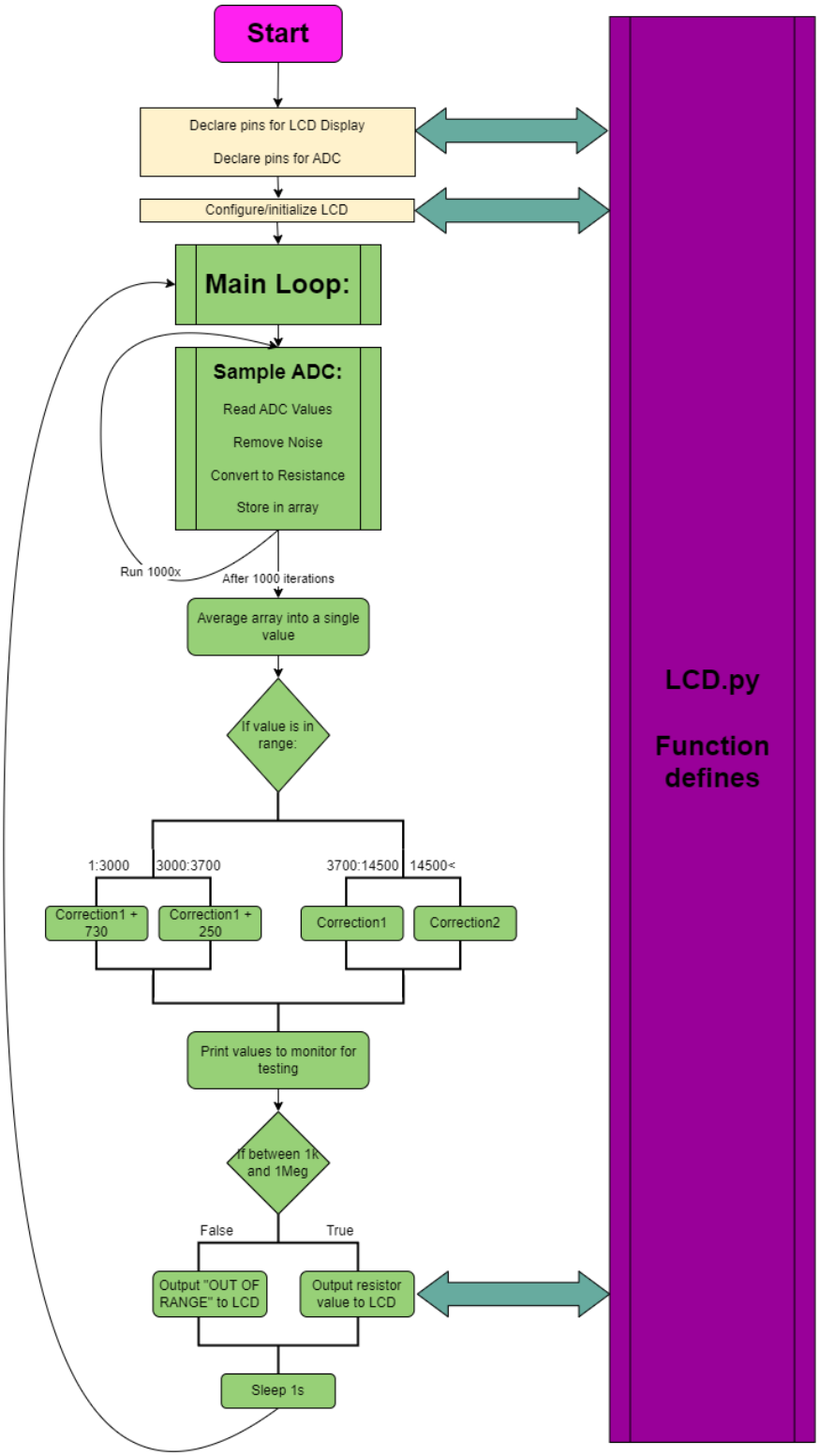


Figure 3: Software Block Diagram for the ADC-LCD system.

The code for this system ended up being similarly straightforward, but it was not always like that. Originally, when I was using R1=1k, I spent a lot of time trying code-based solutions to

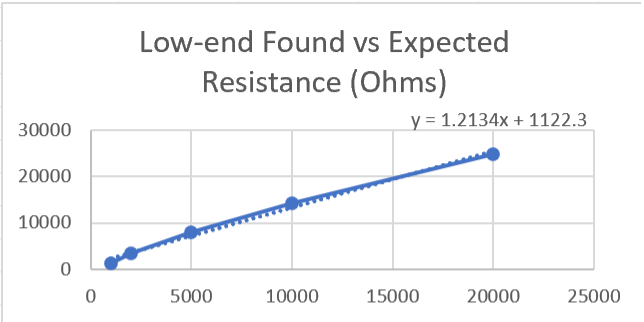
filter out the noise. Most attempts involved taking 1000 samples from the ADC, storing them in an array and performing some filtering algorithm. I first tried using the average, but it varied too greatly. I switched to the median, which was better, but I frequently found the median of the higher R2 values to be the same. I then tried outlier rejection, and a moving average [2]. In the final version, I simply used a regular average of the 1000 values, and that coupled with the increased R1 was enough to produce a reliable and consistent result.

The LCD.py library was used 3 times in my program. It was used twice at the beginning to initialize the LCD, and once at the end to write the output to the LCD, whether that be the resistance value or the 'out of range' message.

Other than the failed attempts at noise reduction, the most difficult part of the code was implementing the correction algorithm on the resistance values. Though this will be discussed more in the next section, I originally used a massive if-else statement with 10 sections so that I could tune the values more accurately. This was only necessary when the system had much more range, and the equation was thus much less accurate. In the final version, I only used 4 cases. 3 for the low end, and 1 blanket equation for the values greater than 15k, which logically fits with what we'd expect from such a large R1.

Calibration

Real Resistance	Found average	Found min	Found max
1000	1405	1340	1470
2000	3550	3500	3600
5000	7934.5	7880	7989
10000	14250	14200	14300
20000	24950	24800	25100
56000	59500	59000	60000
100000	104570	104440	104700
200000	200500	200400	200600
330000	327500	327000	328000
470000	462500	462000	463000
560000	552000	551000	553000
680000	663000	662000	664000
780000	759500	759000	760000
820000	798000	797000	799000
876000	850000	849000	851000
920000	896000	895000	897000
1000000	973500	957000	990000
1100000	1067000	1059000	1075000



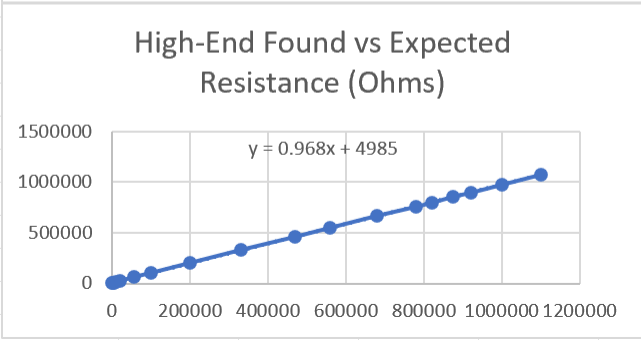


Figure 4: Data and graphs of resistance values used in the calibration process.

The calibration was yet another aspect made more complicated due to the previously discussed early issues. High variance led me to use average resistance values from a range of found expected values. This was potentially overkill, but I kept it in the final revision because it couldn't hurt. Overall, I found two linear curves to be useful for correcting the resistance values [3]. One curve specifically designed for the low-end, only considering values up to 25k found and 20k real. This curve was used for found values up to 15k. The other equation used every single data point, including the low end. Despite its all-encompassing design, this curve only worked for values over 15k, but it was very accurate past that point.

Three versions of the low-end equation were used, each with a certain offset to make them more accurate in their ranges. I suspect the necessity of a low-end curve at all, along with such individual differences stemmed from the use of the 1Meg resistor as R1. Differences between each resistor on the low end were approaching such small values that noise interfered more here, and values were overall less consistent, needing individual tuning.

It is also worth noting that the equations pictured above are backwards from those used in my program. Y's above are the found resistance values, so the equations needed to be reversed to solve for X, the real resistance values.

Results

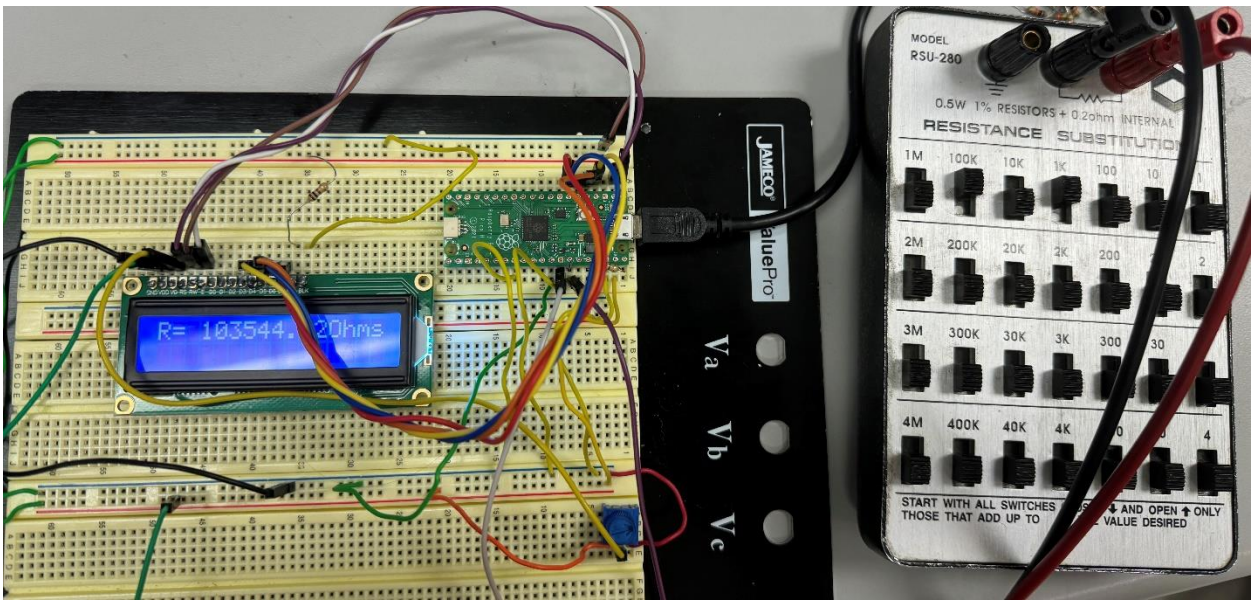


Figure 5: System response for a 100k load.

By the end, everything worked as intended. It was a rude awakening when I had everything working on regular through-hole resistors at home and then had to switch to the

resistor box in the lab, as it required entirely redoing my characterization curves (for something like the 10th time) but that is my fault for not making it to lab sooner.

Using the previously discussed curves and adjustments to their values at the lower ranges, everything was within 10% of real values when all was said and done. Though most values, especially those above 15k were in even tighter ranges than that. The equations generated in excel were incredibly accurate and effective for producing a fully functional Ohmmeter.

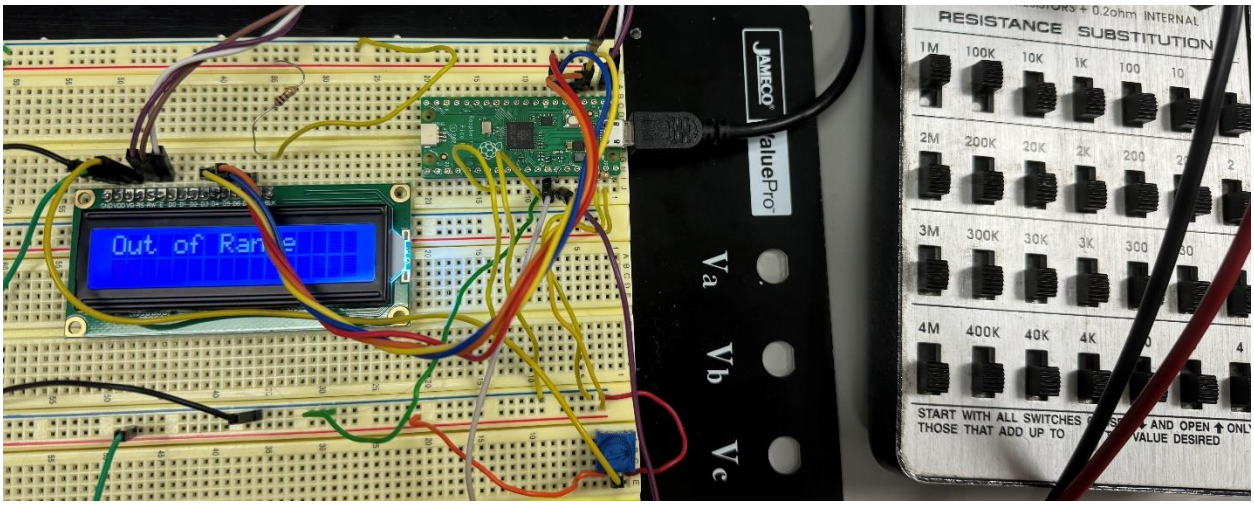


Figure 6: System response for 1.1 million Ohms.

And as can be seen in figure 6, values outside of the 1k-1Meg range displayed as “out of range” on the LCD.

Conclusion

What is a microcontroller without an ADC? Thankfully, we don’t have to worry about that, since our Raspberry Pi Pico has one that is very capable, especially when used in conjunction with a well-designed circuit and calibration curve. Such an ADC is part of what makes microcontrollers useful for such a wide range of applications, as discussed in my previous report. We were able to make a simple but incredibly effective and largely accurate Ohmmeter with nothing but a Pico, an LCD and a couple of resistors, and if that isn’t a testament to the usefulness of the Pico or its ADC, I don’t know what is.

Appendix

Main.py:

```
import math
import machine
from machine import Pin
import utime
import LCD  # Import the LCD module

# Define LCD pins
EN = Pin(0, Pin.OUT)
RS = Pin(1, Pin.OUT)
D4 = Pin(2, Pin.OUT)
D5 = Pin(3, Pin.OUT)
D6 = Pin(4, Pin.OUT)
D7 = Pin(5, Pin.OUT)
PORT = [D4, D5, D6, D7]

adc_pin1 = machine.ADC(28)  # GPIO 28 as ADC1
adc_pin2 = machine.ADC(27)  # GPIO 27 as ADC2



adc = machine.ADC(1)

LCD.Configure()
LCD.lcd_init()



# Reference voltage
Vref = 3.3

while True:
    #Sample for 1000 pre-adjusted resistance values
    i=0
```

```
values = [None] * 1001
    for i in range(1001):

            # Read the ADC value
        adc_value1 = adc_pin1.read_u16()
        adc_value2 = adc_pin2.read_u16()

            #Format the values for voltage over R (adc1) and Ground (adc2, used for noise reduction)
        voltage = (adc_value1 / 65535.0) * Vref
        ground = (adc_value2 / 65535.0) * Vref

            #Remove any easy noice by using differential between adc1 and adc2
        final_voltage = (voltage-ground)

            #Create base level voltage (smallest I could record) so that the equations don't break with v=0
        if final_voltage <= 0:
           final_voltage = 0.000805676

            #1Meg known resistor
        known_resistor_value = 1000000

            #Calculate resistance (pre-adjustment) using voltage divider equation
        resistance_pre = (known_resistor_value)/((Vref/final_voltage)-1)

            #Store in array of values
        values[i] = resistance_pre
    values.sort()

    #Average the values
    total = sum(values)
    num_elements = len(values)
    resistance = total / num_elements
```

```
#Implement correction equations on certain ranges
if 1<=resistance<3000:
    real_resistance = (resistance - 1122.3)/1.2134+730
elif 3000<=resistance<3700:
    real_resistance = (resistance - 1122.3)/1.2134+250
elif 3700<=resistance<14500:
    real_resistance = (resistance - 1122.3)/1.2134
else:
    real_resistance = (resistance - 4985)/0.968

#Print output to monitor for testing
print("ADC Value:", adc_value1)
print("Voltage (V):", voltage)
print("Resistance (Ohms):", real_resistance)
print(resistance_pre)

#Output to LCD
LCD.lcd_clear()
LCD.lcd_home()
if 900 < real_resistance < 1003000:
    LCD.lcd_puts("R= {:.2f}Ohms".format(real_resistance))
else:
    LCD.lcd_puts("Out of Range")

utime.sleep(1)
```

References

[1] J. Braza, “How voltage dividers work,” Circuit Basics, https://www.circuitbasics.com/what-is-a-voltage-divider/ (accessed Oct. 2, 2023).

[2] B. Baker, “Why and how to use digital filters for high-resolution, high-speed, analog-to-digital conversions,” DigiKey, https://www.digikey.com/en/articles/why-and-how-to-use-digital-filters-for-analog-to-digital-

conversions#:~:text=Averaging%20digital%20filters&text=ADC%20users%20use%20averaging%20algorithms,resolution%20by%20system%20noise%20reduction. (accessed Oct. 2, 2023).

[3] Evaluating ADC and DAC Performance characteristics, https://www.electronicdesign.com/technologies/analog/article/21165089/per-vices-corp-evaluating-adc-and-dac-performance-characteristics (accessed Oct. 2, 2023).