Introduction

The ESP32-S2 microcontoller has a true DAC (Digital to Analog Converter.) This feature is relatively rare in microcontollers but it enables some unique experiments. In this project you will use the DAC to control the voltage to a device, and two of the ADC's (Analog to Digital Converters) to measure the resistance (inverse of the conductance) of a device.

The microcontroller will measure both the current through and voltage across the device as the voltage across it changes and fit the result to a line to get the resistance. I also have a list in an appendix of other applications of this project.

To make the project more interesting you will also wire a Type K thermocouple to a microcontroller breakout board using a digital protocol. A type K thermocouples can read from -200 °C to 1350 °C. You can use them to measure the temperature coefficient of the resistance and the transition temperature of high T_c superconductors.

I set up a GitHub repository with all of the documents and code for this project at **`https://github.com/profhuster/mBFY22-Conductance`**.

Bill of Materials

Items needed with approximate cost.

- Case, 3D printed – $2 of PLA filament.
- Half breadboard – $5.
- Lolin S2Mini microcontroller board – $5.
  - Option: TinyS2 – ESP32-S2 Development Board from AdaFruit – $19.
  - Adafruit ESP32-S2 Feather – $18.
- Thermocouple breakout board, MAX31855 – $15.
- Type K thermocouple – $10.
- Miscellaneous electronic supplies:
  - 100 Ω resistor.
  - Sample resistor <= 100 Ω.
  - Hook up wire, 22AWG Solid Core.
- **Total**: $56

Tools, Equipment, and Software:

- Computer.
  - Need to install *Thonny* editor/IDE.
  - Optional: jupyter notebook.
- USB C cable.
- Soldering iron.
- Wire strippers.
- Optional:
  - Wire cutter.
  - Needle-nosed pliers.

Getting Started

Follow the steps in the *Setting Up CircuitPython* to load CircuitPython onto the Lolin S2Mini microcontroller.The Lolin S2Mini.

Hardware

The pin assignments for the S2Mini are in *Figure 1*. To use on a breadboard, I only use the outside pins. Almost any pin can be used for digital in/out and ADC, but only pins 18 (and 17 which is not wired up) have a DAC which can be used to output a voltage. The DAC has 8-bits of resolution (0 – 0xff.) CircuitPython maps the DAC resolution to 16-bits (0 – 0xffff) so setting pin 18 from 0 to the max should gives= an output voltage of 0 to 3.3 V. In the first project you will test this out.

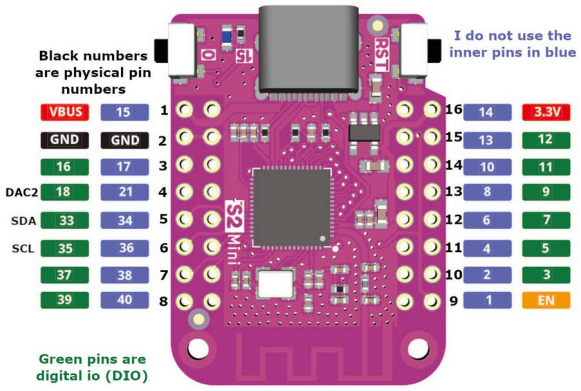

Figure 1: S2Mini pinout. The only DAC pins are 17 on the inside row and 18. Not all pins have ADC's. We will use pins 9, 11, and 12.

DAC Complication

The DAC's on the ESP32S2 have internal pull-up resistors, so to get a decent output voltage range you need to supply a pull-down resistor. *Figure 2* shows the output voltage versus DAC setting for different pulldown resistors. To get a good range on the DAC, we need to have 1 kΩ or less connected to the DAC pin. Later when we design a circuit to measure conductance, we need to make sure it has an equivalent resistance of 1 kΩ or less.

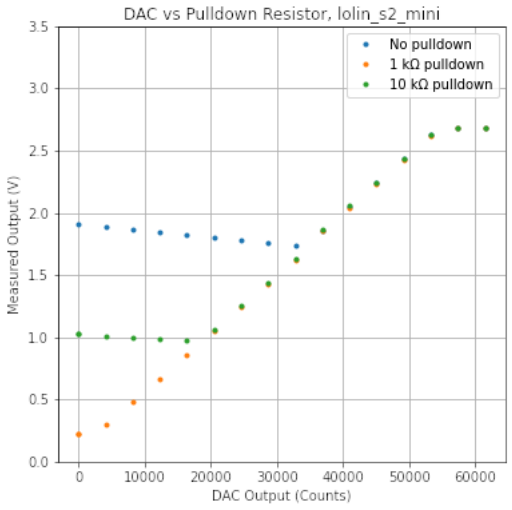

Figure 2: DAC output on pin 18 with different pulldown resistors.

Project 1 – Take DAC Voltage versus DAC Counts Data

See *Figure 3* for a picture of the circuit.

- Plug the S2Mini into a breadboard.
- Wire GND to the blue "–" power rail.
- Wire pin 16 to row 12 after the S2Mini.
- Wire pin 18 to the same row.
- Wire a 100 Ω resistor from the same row and to the ground rail. *Note*: the figure shows a 1 kΩ instead of 100 Ω.

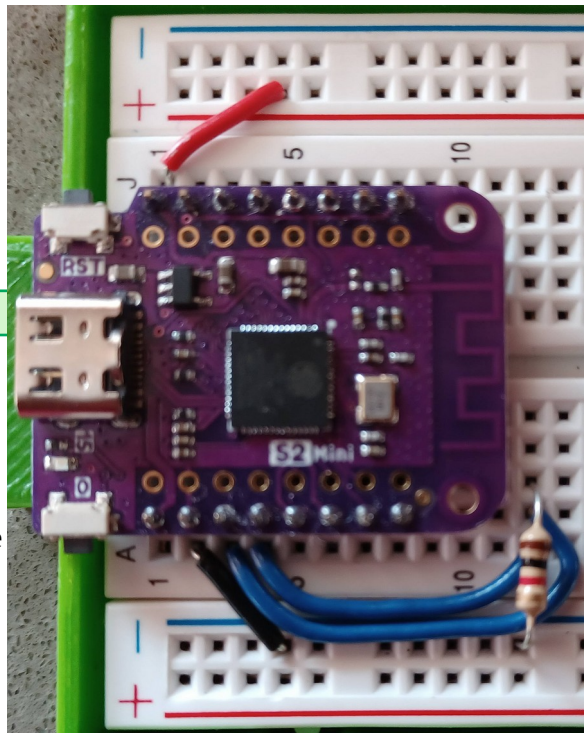

Figure 3: Wiring for DAC test.

The Code

First you need to load the libraries we will be using to the S2Mini. From the zip file, locate the folder lib. Drag it to the CIRCUITPY disk. If it prompts you that the folder there exists, answer Replace, or copy anyway.

In your code the libraries (sometimes called *modules*) are imported first. The library **`board`** imports the pins available for the specific board and a few other items. I encourage that you explore these libraries. You can do that at the python prompt by typing, for example,

```
>>> import board
>>> help(board)
>>> dir(board)
```

I do this a lot with libraries and objects so I can learn what can be done with CPy. The other source of information is searching the **`CircuitPython web site`**. The library **`analogio`** handles DAC and ADC functions for microcontrollers. I also import **`sleep`** and **`sqrt`** from standard python libraries.

This file for this code is **`testDAC.py`**. *Note*: The code on GitHub is slightly different. Please us the GitHub code.

```
import board
import analogio
from time import sleep
from math import sqrt
```

Next I create variables that parameterize the program behavior. I strongly advocate the programming proverb *Never bury a constant in your code; assign it to a variable or constant at the top of your code*. Here I had to look up which pins are DAC compatible (pins 17 & 18) and which are ADC compatible (many pins.)
For each data point I I sleep for **`tADC`** seconds to give time for the DAC to settle then average over **`nAvg`** points. The variable **`counts`** sets the DAC voltage, the variables **`countsMax`** and **`countsInc`** set the loop for stepping the DAC. The CircuitPython library maps all DAC's to 16 bits, so I use hex to code the maximum value as **`0xffff`**. Setting an increment of **`0x0400`** gives 64 data point in the scan. I find hex easier than remembering 65535 and 1024.

```
dacPin = board.IO18
adcPin = board.IO16
nAvg = 10
tADC = 0.01
countsMax = 0xffff
countsInc = 0x0400
```

To grab a pin for analog use, the **`analogio`** library does a lot of work for you. I set up the variable **`dToV`** to convert counts to voltage for ADC readings.

```
dac = analogio.AnalogOut(dacPin)
adc = analogio.AnalogIn(adcPin)
dToV = adc.reference_voltage / 0xffff
```

In this section you can input a comment that will get stored in the data file. I used this to describe the pull down resistors. I also print important information about the microcontoller and the code. The **`board_id`** variable describes the microcontoller, in this case the string is **`lolin_s2_mini`**. Note I print out the parameters for the run. Starting these lines with `#` makes it easier to import the data into an analysis program.

```
comment = input("Enter comment: ")

print(f"# board_id = {board.board_id}")
print(f"# dacPin: {dacPin}, adcPin: {adcPin}, dToV: {dToV}")
print(f"# nAvg: {nAvg}, tADC: {tADC}")
print(f"# {comment}")
```

Next I define a function that returns an ADC reading and its standard deviation. This is where the parameters **`nAvg`** and **`tADC`** are used. I strongly encourage my students to always be thinking about uncertainties in measurement. In this case the standard deviation tell you how spread out the measurements are.

```
def readADC():
    sum = 0.0
    sumSq = 0.0
    for i in range(nAvg):
        reading = adc.value * dToV
        sum += reading
        sumSq += reading * reading
        sleep(tADC)

    avg = sum / nAvg
    std = sumSq / nAvg - avg * avg
    if std <= 0.0:
        std = 0
    else:
        std = sqrt(sumSq / nAvg - avg * avg)
    return (avg, std)
```

Below is the main loop. It set the DAC value, sleeps, then reads the ADC and prints the result. I always use a `#` in the first line of comments, and make the data in comma separated columns, the CSV format. This makes reading and analyzing the data easy. Also note I manually add the last point otherwise I wouldn't get a data point for the maximum value of the DAC.

```
for out in range(0, countsMax, countsInc):
    print(f"{out}, ", end="")
    dac.value = out
    sleep(0.1)
    (V, dV) = readADC()
    print(f"{V}, {dV}")

out = 0xffff
print(f"{out}, ", end="")
dac.value = out
sleep(0.1)
(V, dV) = readADC()
print(f"{V}, {dV}")
```

These last two lines release the pins. If these lines are not included, when you rerun the code it can't grab the pins.

```
dac.deinit()
adc.deinit()
```

Analyzing the DAC Data

The output is written in the *Shell* window at the bottom of *Thonny*. Below is an example run of the output:

```
# board_id = lolin_s2_mini
# dacPin: board.IO18, adcPin: board.IO16, dToV: 5.03547e-05
# nAvg: 10, tADC: 0.01
# 1 k pulldown
0, 0.241481, 0.00211784
4096, 0.297284, 0.00316913
8192, 0.487575, 0.00245358
12288, 0.672477, 0.00304931
16384, 0.869571, 0.00223759
20480, 1.06437, 0.00316442
24576, 1.26018, 0.00292969
28672, 1.45058, 0.00276214
32768, 1.64508, 0.00169146
36864, 1.85078, 0.000976563
40960, 2.04587, 0.00195313
45056, 2.23348, 0.00195313
49152, 2.43148, 0.00239208
53248, 2.63266, 0.00338291
57344, 2.68199, 0.00276214
61440, 2.68199, 0.00276214
65535, 2.68199, 0.00276214
```

I cut and paste this into an editor, edit to delete any extra lines and save it with a file name that encodes the date and time the data was taken, **`2022-04-28-1546.csv`**, for example.

I use jupyter notebooks running python to analyze the data. They are in the jupyter folder on GitHub. One nice thing about the CSV data format is that is can be read by many programs including spreadsheets like Microsoft Excel. *Figure 4* is a plot from my jupyter notebook **`plotDAC.ipynb`**.

The curve looks linear between 5000 and 45000 counts and it does not go down to zero volts nor up to 3.3 V. These are limitations we will have to work with when we make a conductivity measurement.

I took the data in *Figure 2* using this program and different pull down resistors.

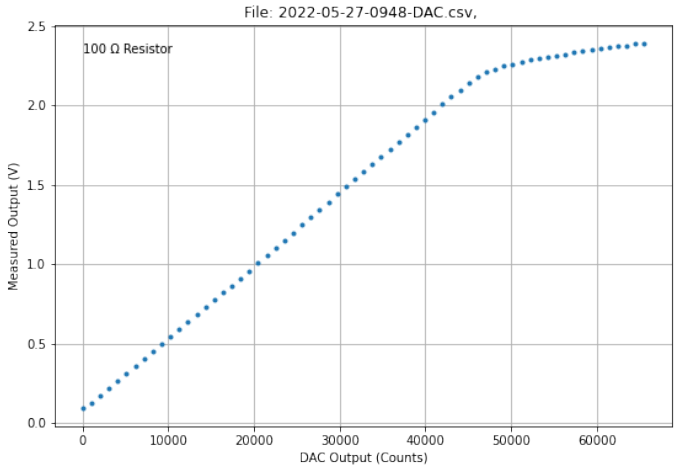

Figure 4: DAC output voltage versus input counts. The curve depends on the value of the pulldown resistor.

PROJECT 2: TWO-WIRE CONDUCTIVITY MEASUREMENT

The next project will use the microcontoller to measure the resistance (inverse of the conductivity) of a device.

The basic idea, *Figure 5*, is that the voltage drop across the sensing resistance is the current through the sample. So

$$I_{sample} = \frac{V_0 - V_1}{R_{sense}}$$

and

$$V_{sample} = V_1 \quad .$$

So

$$R_{sample} = \frac{V_{sample}}{I_{sample}}$$

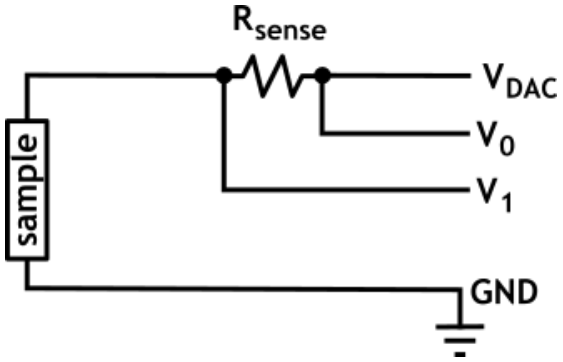

Figure 5: Schematic for a two-wire resistance measurement.

We will step V_{DAC} from 0 to 3.3V, so a graph of V_{sample} versus I_{sample} should be a straight line with a slope of R_{sample}. For each DAC setting, we will read V_0 and V_1 multiple times and get an uncertainty for each. We will carry the uncertainty through the linear fit to calculate an uncertainty for the resistance.

I wired this circuit up with a 100 Ω sense resistor and 69 Ω resistor for the sample.

Figure 6 shows a graph of V_0 and V_1 versus *DAC* setting. You can see at counts less than about 4000 the curves flattens out or slightly rises. I believe this is an effect of the pull up resistor. At the higher end, the voltage does not go above 2.58 V. I believe this is a limit of the ADC. In between the two graphs look linear. In the software I will limit the data to DAC counts to greater than 4000 and voltages less than 2.50.

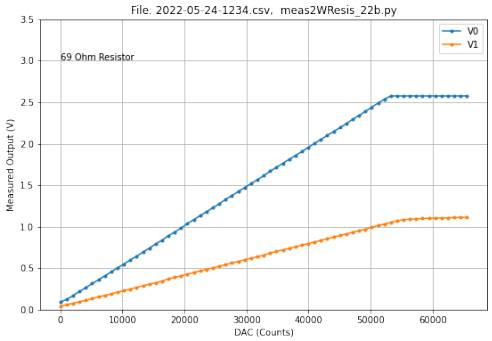

Figure 6: V0 and V1 versus DAC Counts for a 60 Ohm resistor as a sample.

In the software I actually average some number of ADC readings together at each DAC step. This gives more accuracy to each data point and I calculate the standard deviation for the point, so I also have an uncertainty estimate for each point.

I do a linear fit with uncertainty to the data. This gives both a measurement of the sample resistance, but also an uncertainty estimate for the resistance. *Figure 7* is an example of one measurement.

Once the code for making the measurement was checked and debugged, the basic resistance measurement is in a loop that measures the resistance versus time. *Figures 8 & 9* show a run where the resistance is repeated. From running the program versus time, the standard deviation of the resistance measurements is about 0.1 Ω.

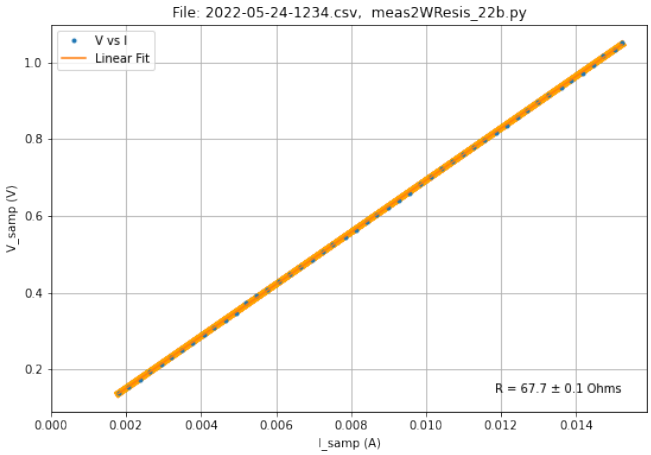

Figure 7: Linear fit of V_{sample} versus I_{sample}.

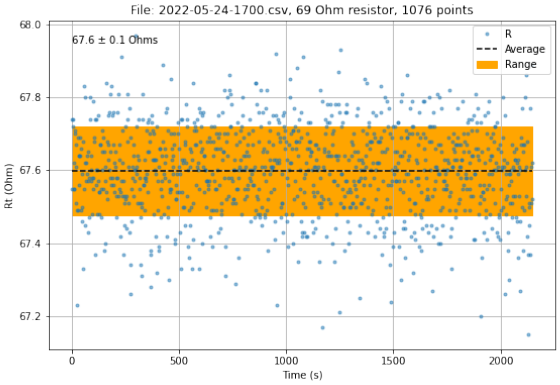

Figure 8: Resistance versus time. The orange band is +/- one standard deviation.

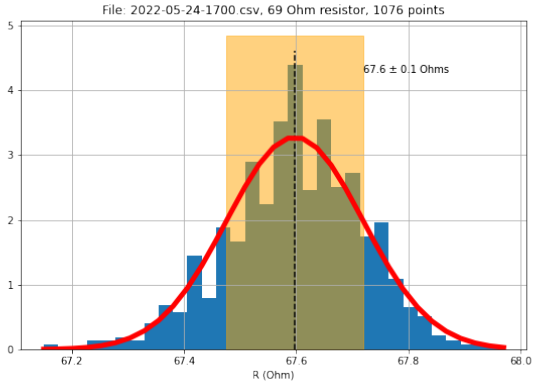

Figure 9: The data in Figure 8 as a histogram. A normal distribution matching the average and standard deviation is shown.

The Hardware

A picture of the circuit is in *Figure 10*. The sample is a 68 Ω resistor. You can see the wires are tightly twisted and soldered to a two pin connector.

To wire the circuit:

- Wire the 3.3V to the “+” power rail (red.)
- Wire pin GND to the blue “-” ground rail (black.)
- Wire the DAC, pin 18 to the 100 Ω sensing resistor. Also wire that end of the resistor to pin DIO5 which is V0. They are the yellow wires in the figure.
- Finally the last wire of the sample connector is wired to ground, the “-” ground rail (black.)

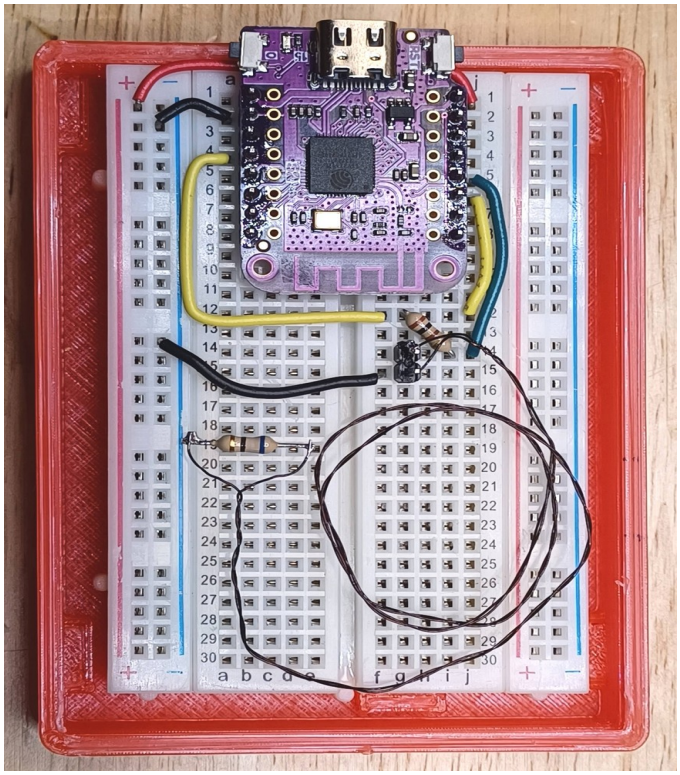

Figure 10: Circuit wired for resistance measurement. The sample is attached by a pair of twisted wires to the circuit.

The Code

A lot of the code is the same as the previous project. The file is **`meas2WResis.py`**. I will just highlight the new code. The code below does a fit to a straight line and weights the fit by the uncertainty in the y-coordinate. Since I did not use numpy because only a subset of **`numpy`** is available as the **`ulab`** library. Instead I use the **`array`** library so I have to explicitly loop over the data. The function returns the best fit values for the intercept and slope and the covariance matrix.

```
# Do linear fit
def linReg(n, x, y, sigY):
    """linReg - linear fit with returned uncertainties."""
    invSigY2 = 0
    S = 0.0
    Sx = 0.0
    Sy = 0.0
    Sxx = 0.0
    Sxy = 0.0
    for i in range(n):
        invSigY2 = 1.0 / (sigY[i] * sigY[i])
        S += invSigY2
        Sx += x[i] * invSigY2
        Sy += y[i] * invSigY2
        Sxx += x[i] * x[i] * invSigY2
```

```
        Sxy += x[i] * y[i] * invSigY2

    Delta = S * Sxx - Sx * Sx
    a = (Sxx * Sy - Sx * Sxy) / Delta
    b = (S * Sxy - Sx * Sy) / Delta
    varA = Sxx / Delta
    varB = S / Delta
    covAB = -Sx / Delta
    return (a, b, varA, varB, covAB)
```

Next is the function that returns the resistance. First I create arrays for intermediate data and define **`first`**.

```
# Make arrays for data
ISamp = array('f', nCounts * [0.0])
VSamp = array('f', nCounts * [0.0])
sigV = array('f', nCounts * [0.0])
first = True # Boolean to print nGood only once
```

The next part of the code loops over the DAC values. If you are not familiar with **`enumerate()`**, it is an iterator over an array, but it returns two values: a counting index, here the variable **`i`**, and the next array value, **`out`**. Next, the first time it is called, it prints the number of data points used in the linear fit. Finally it calculates the uncertainty in the resistance as the square root of the variance and returns both the fit value and its uncertainty.

```
def measureR():
    global first, ISamp, VSamp, sigV

    # Collect the data into arrays
    nGood = 0
    for (i, out) in enumerate(range(0, countsMax, countsInc)):
        dac.value = out
        sleep(tADC)
        (V0, dV0) = readADC(v0)
        if out > 4000 and V0 < 2.65:
            (V1, dV1) = readADC(v1)
            ISamp[nGood] = (V0 - V1) / rSens
            VSamp[nGood] = V1
            sigV[nGood] = sqrt(dV0 * dV0 + dV1 * dV1)
            if debug:
                print(f"{out}, {ISamp[nGood]}, {VSamp[nGood]}, {sigV[nGood]}")
            nGood += 1

    # On first call print the number of data points
    if first:
        first = False
        print(f"# {nGood} good points")

    # Fit data to a line
    (a, resis, varA, varB, covAB) = linReg(nGood, ISamp, VSamp, sigV)
    uncResis = sqrt(varB)
    if debug:
        print(f"# Resistance = {resis:.2f}, {uncResis:.2f}")

    # Return resistance and uncertainty
    return (resis, uncResis)
```

Finally the code below is the main loop that takes data. The function monotonic() returns the number of seconds since the microcontoller booted. The entire loop is in a **`try`** – **`finally`** structure. The reason for this is to release the pins when the program is stopped with a **`<Ctrl>C`** keystroke.

```
# Loop forever taking data
# Stop with ^C
t0 = monotonic()
try:
    while True:
        while monotonic() < t0 + loopTime:
            sleep(0.01)
        t0 += loopTime
        (R, sigR) = measureR()
        print(f"{monotonic():.1f}, {R:.2f}, {sigR:.2f}")
finally:
    # Clean up
    print("# Cleaning up")
    dac.deinit()
    v0.deinit()
    v1.deinit()
```

Project 3 – Resistance versus Temperature

The final step is to use a breakout board to measure the sample temperature with a thermocouple. A Type K thermocouple has a useful range of -200 ºC (73 K) to 1350 ºC. They are nice because they can be used for liquid nitrogen experiments. Thermocouples are hard to use without specialized circuits, but don't fear! They are nicely handled with a $16 breakout board like **this one**. We are going to add this to the previous project.

Hardware

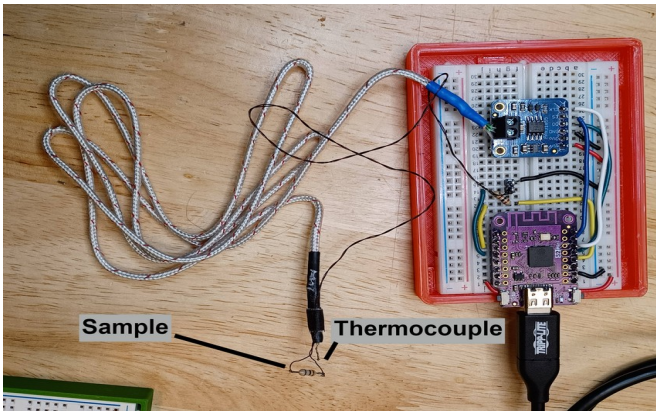

Figure 11: The complete circuit.

The overall device is shown in *Figure 11*, a close up of the wiring is in *Figure 12*.

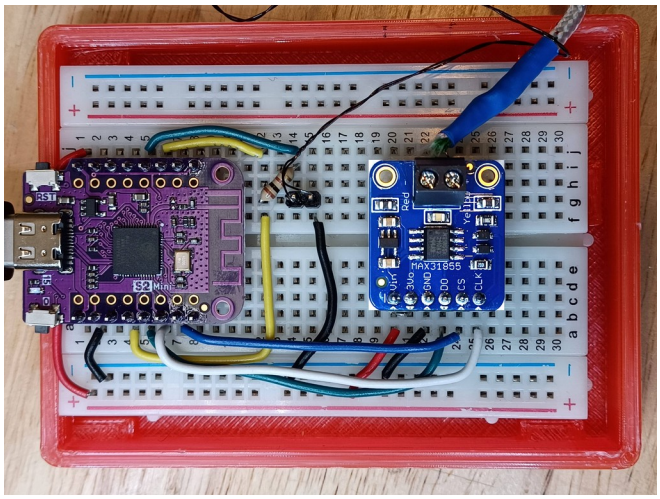

Figure 12: Close up of the wiring including the thermocouple breakout board.

The wiring is the same as the previous plus:

- Plug the breakout board into the breadboard.
- Wire S2Mini pin DIO33 to the DO (Data Out) pin on the breakout board (green wire.)
- Wire S2Mini pin DIO35 to the SCL pin on the breakout board (white.)
- Wire S2Mini pin DIO37 to the CS pin on the breakout board (blue.)
- Wire the 3.3V pin of the S2Mini across to the red "+" power rail on the breadboard (red.)
- Wire the breakout board VIN to the "+" power rail (red.)
- Put a bare wire between the VIN and 3V3 pins on the breakout board. (This can't be seen in the figure because it is underneath the breakout board.
- Wire the GND pin on the breakout board to the "-" ground rail on the breadboard (black.)
- Screw the thermocouple into the breakout board. Once it is running, hold the thermocouple. If the temperature goes down, switch the wires.

The Code

The code is in file **meas2WRvsT.py**. In addition to the code for *Project 2* it has the following code:

```
import digitalio
import busio
import adafruit_max31855
import microcontroller
import watchdog
```

The first three lines import libraries needed for the MAX31855 breakout board. For a description of how this code was developed, see the appendix in the handout *2022-mBFY-Introduction*.

```
spi = busio.SPI(board.IO35, board.IO36, board.IO33)
cs = digitalio.DigitalInOut(board.IO37)
max31855 = adafruit_max31855.MAX31855(spi, cs)
```

The first line defines the SPI interface using the pins designated. The second line defines the chip select pin. The last line uses the AdaFruit library for this breakout board. I chose this board because it is only for Type K thermocouples. If you have a different type of thermocouple or a platinum resistance thermometer, there are other boards you can use.

The first three lines import libraries needed for the MAX31855 breakout board. For a description of how this code was developed, see the appendix in the handout *2022-mBFY-Introduction*. Once the breakout board is ready reading the temperature is as simple as reading the value of **`max31855.temperature`**. Easy peasy!

The code allows the user to enter a comment that is stored in the data file, but I didn't want the code to hang up if the data be taken remotely. The solution is a *watchdog* timer. You set the watchdog and if it isn't fed or canceled, it goes off and does some contingent action. Here is the code around getting a comment:

```
wdt = microcontroller.watchdog
wdt.timeout = 10
wdt.mode = watchdog.WatchDogMode.RAISE
try:
    comment = input("Enter comment: ")

except watchdog.WatchDogTimeout as e:
    comment = "No comment"
wdt.deinit()
```

It sets the watchdog with a 10 second timeout. It set the mode of the watchdog to raise an exception. If you don't set the mode, the microcontoller is completely reset!

Now the line getting a comment is in a **`try`** – **`except`** structure. If no comment is entered in 10 seconds, it sets the comment to "no comment".

Finally the watchdog is deinitialized, or turned off.

```
print(f"{monotonic():.1f}, {R:.2f}, {sigR:.2f}, {max31855.temperature:.2f}")
```

Finally, getting the temperature is easy. The dynamic variable **`max31855.temperature`** retrieves the thermocouple temperature.

An Experiment

With the setup as built I measured the resistance versus temperature. I do not know what type of resistor it is in the device. I taped the sample and thermocouple wires together to the thermocouple was about 2 mm away from the sample.

Figure 13 shows the temperature versus time as I cooled and heated the sample and thermocouple.

The result is shown in *Figure 14*. The temperature coefficient of resistance is -168 ppm/°C. (ppm is parts per million.)

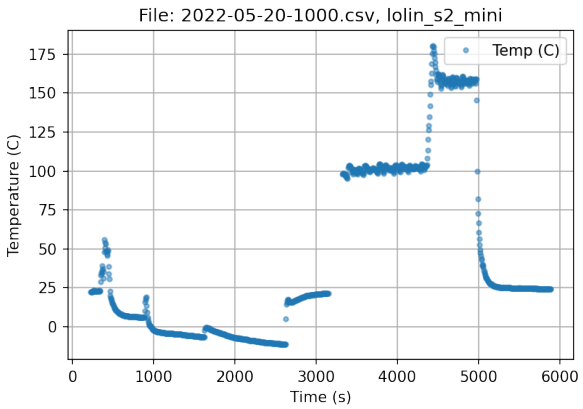

Figure 13: The temperature versus time of the experiment.

Finally, *Figure 14* shows the resistance is temperature dependent. The temperature dependence is usually give as ppm (Parts Per Million) per degree Celsius. The value of about -200 ppm/°C. I do not know what type of resistor this is, but, for example, carbon film resistors are quoted as having a temperature coefficient of -200 ppm/°C, so this value is right on that value, probably just a change occurance.

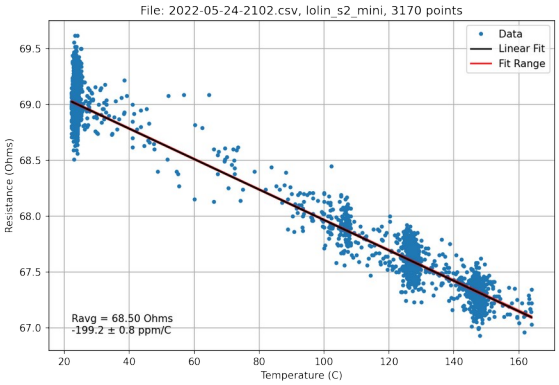

Figure 14: Resistance versus temperature. The temperature coefficient of resistance is about -200 ppm/°C.

Extension Project: Four-Wire Conductivity Measurement

The next project will use the microcontoller to measure the resistance (inverse of the conductivity) of a device. If you want to put the device some distance away, like in a cryostat, the resistance of the wires would be added to the resistance of the device. This can be significant for some experiments. A standard, and very clever, method to avoid the added resistance of the wire is to use a *four-point* measurement technique.

The basic idea, *Figure 5*, is that you two wires to supply current through the sample, and two different wires to measure the voltage across the sample as close to the sample as possible. Voltage measurements are be made with only tiny amounts of current, so the ohmic loss in the voltage-measuring wires is insignificant. Also since tiny currents flow through the voltage measuring circuit, nearly all of the current measured flowing through the sensing resistor also flows through the sample.

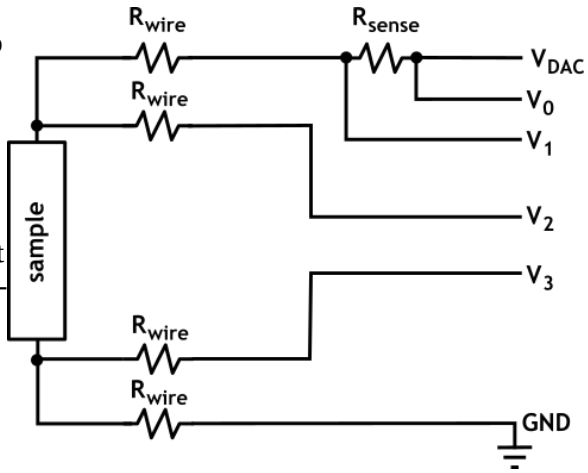

Figure 15: Schematic for a four-wire resistance measurement. The R_{wire}'s represent potentially long wires of unknown resistance.

So

$$I_{sample} = \frac{V_0 - V_1}{R_{sense}}$$

and

$$V_{sample} = V_2 - V_3 \quad .$$

We will step V_{DAC} from 0 to 3.3V, so a graph of V_{sample} versus I_{sample} should be a straight line with a slope of R_{sample}.

I wired this circuit up with a 100 Ω sense resistor and about 69 Ω for the sample.

Figure 6 shows a graph of V_0 and V_1 versus *DAC* setting. You can see at counts less than about 4000 the curves flattens out or slightly

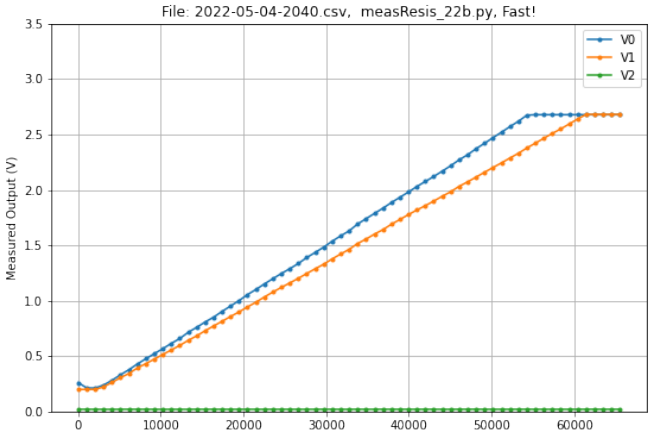

Figure 16: A plot of V0, V1, and V3 versus DAC counts.

rise. I believe this is an effect of the pull up resistor. At the higher end, the voltage does not go above 2.65 V. I believe this is a limit of the ADC. In between the two graphs look linear. In the software I will limit the data to DAC counts greater than 4000 and voltages less than 2.65.

In the software I actually average some number of ADC readings together at each DAC step. This gives more accuracy to each data point and I calculate the standard deviation for the point, so I also have an uncertainty estimate for each point.

I do a linear fit with uncertainty to the data. This gives both a measurement of the sample resistance, but also an uncertainty estimate for the resistance. *Figure 7* is an example of one measurement.

Once the code for making the measurement was checked and debugged, the basic resistance measurement is in a loop that measures the resistance versus time. *Figures 8 & 9* show a run where the resistance is repeated. From running the program versus time, the standard deviation of the resistance measurements is about 0.1 Ω.

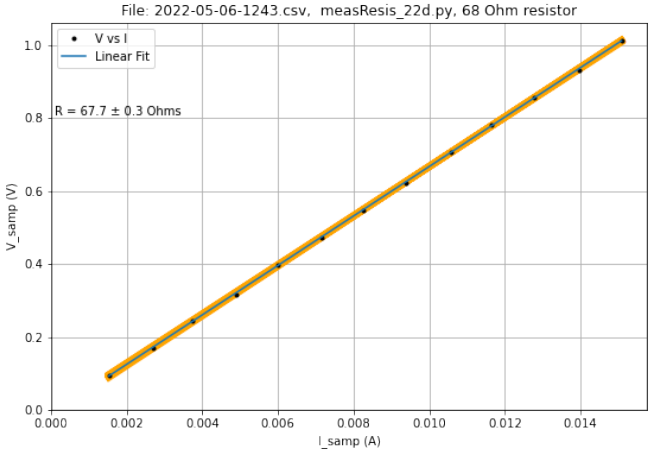

Figure 17: Linear fit of V_sample versus I_sample.

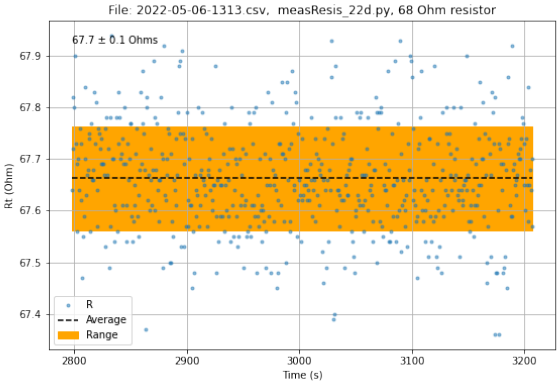

Figure 18: Resistance versus time. The orange band is +/- one standard deviation.

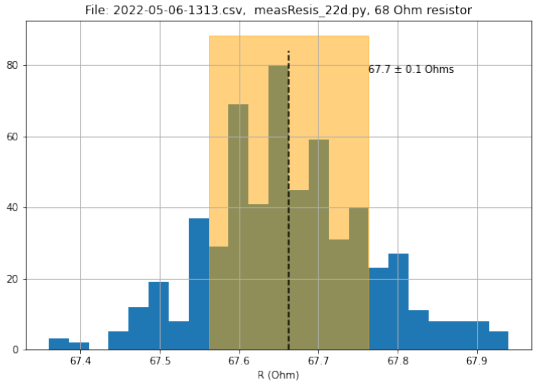

Figure 19: The data in Figure 8 as a histogram. The data appears to be normally distributed.

The Hardware

A picture of the circuit is in *Figure 10*. The sample is a 68 Ω resistor. You can see there are two wires soldered to each end of the resistor. The four wired are tightly twisted and soldered to a four pin connector.

To wire the circuit:

- Wire pin GND to the blue "-" ground rail.
- Wire the DAC, pin 18 to the 100 Ω sensing resistor. Also wire that end of the resistor to pin DIO5 which is V0. They are the yellow wires in the figure.
- The other end of the sensing resistor is wired to both the sample connector and pin DIO7, V1 (green.)
- The second wire of the sample connector is wired to pin DIO9, V2 (blue.)
- The third wire of the sample connector is wired to pin DIO11, V3 (white.)
- Finally the last wire of the sample connector is wired to ground, the "-" ground rail (black.)

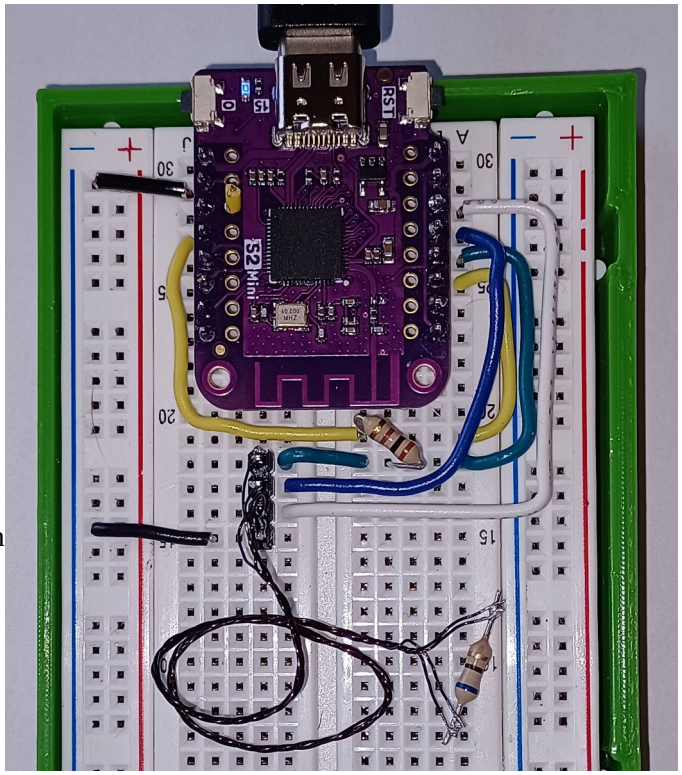

Figure 20: Circuit wired for four-point resistance measurement. The sample is attached by a bundle of four twisted wires to the circuit.

The Code

The code has some straightforward changes, using four ADC values to calculate I_{sample} and V_{sample}. The code file is **`meas4WRvsT_22a.py`**.

EXTENSIONS

1) The ESP32S2 has a built-in WiFi, so you can program it to connect with a WiFi and post its data to a cloud server. A popular protocol is MQTT, and a freee server I use is **`https://io.adafruit.com/`**.

2) The DAC on the S2Mini has issues like not going down to 0V and not being linear versus the counts. These are probably related issues. There are two ways to improve this:

   a) *Figure 13* shows how to wire the DAC to ground with a lower resistance, like 100 Ω, then use an op amp to provide a low impedance voltage source. I love the MCP6022 op amp. It works on 3.3 V supply, has rail to rail input and output, comes in plastic DIP that plugs into breadboards, and has a gain-bandwidth of 10 MHz. It is a good practice to wire a 1 uF capacitor from 3.3 V to GND at the op amp.

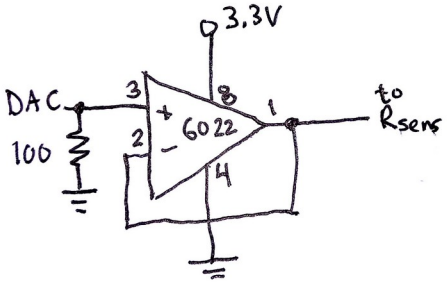

Figure 21: An op amp circuit to buffer the S2Mini DAC.

b) You can use an external DAC output board to provide the voltage to the circuit. These board cost less than $10. See **https://www.adafruit.com/product/935**

The ADC part of the circuit can also be improved:

3) Instrumentation amplifiers can be used for more accurate differential measurements of the voltage across R*sens* and the sample, *Figure 8*. For low resistance samples the gain of the amplifier measuring the sample can be increase up to G = 1000. Figure 8 is a schematic from a high T_c superconductor experiment using a different microcontroller. (The Teensy 3.2 is my favorite microcontroller for critical applications. It does not have a version of CircuitPython, but you can run MicroPython on it.)

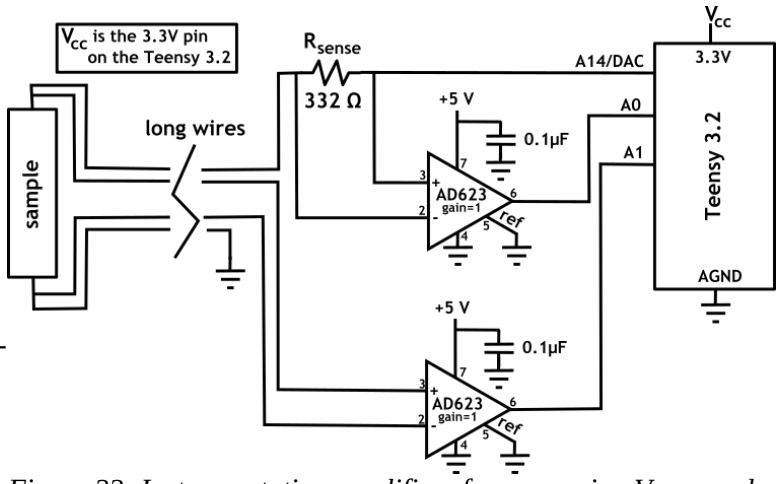

Figure 22: Instrumentation amplifiers for measuring Vsens and Vsample.

4) For working down to cryogenic temperatures a platinum resistance thermometer can be used. The *4-POINT TC EXPERIMENT KIT* from **https://quantumlevitation.com/** has a PT-100 (100 Ω) resistor built in. You can program the microcontroller to measure its resistance or use a PRT breakout board like **https://www.adafruit.com/product/3328** which is connected to the microcontroller with a SPI protocol.

5) For a simple cryostat, use a *sand cryostat*. A sand cryostat is just an insulated foam cooler full of sand. You bury your experiment in the sand, pour liquid nitrogen in until it stops violently boiling and wait. It takes 5+ hours to warm back up to room temperature, so that makes a very nice run for high T_c superconductor measurements.

6) You can modify the code to measure the I-V curve for a diode. This opens the door for measuring $V_{forward}$ in LED's to get Plank's constant or measuring the bandgap of a diode versus temperature.

7) You can make a transistor characterizing device. You would also wire the second DAC, pin 17 on the inside row, to provide a base voltage, then trace out the emitter-collector response versus base voltage, like in *Figure 23*.

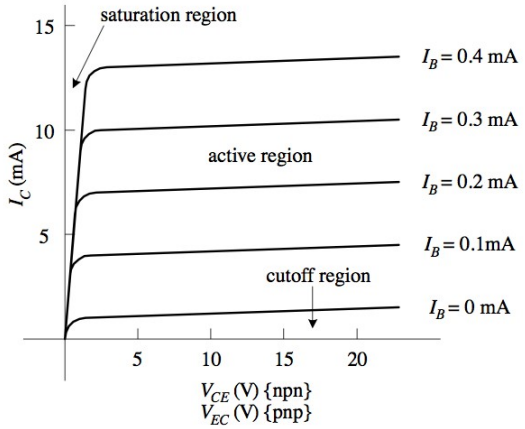

Figure 23: Bipolar transistor curve trace.