

Characterizing the ESP32's ADC

Dec 27, 2024 - 1970 words

Christopher Kalitin Blog (<https://ckalitin.github.io/posts/esp32-adc-characterization/>)

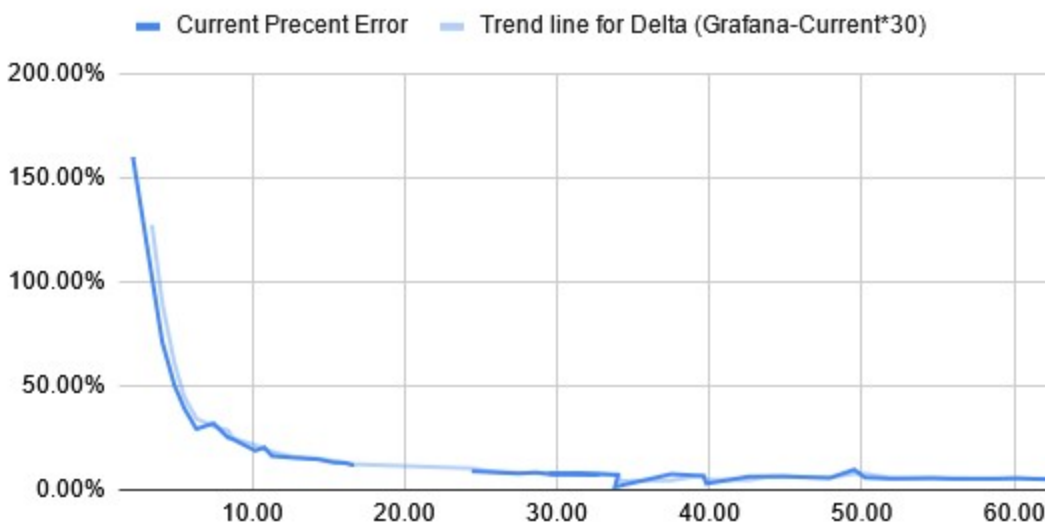
Introduction

I joined the UBC Solar design team in September as a member of the Battery Management System team. I get to talk to wizards every meeting and work on very exciting projects, which makes this is the best team I've ever worked with and most exciting project I've ever worked on. Completely amazing projects and people.

For the previous few months, we've been trying to diagnose an issue we've been having with the current sensor on our battery pack. We use the HASS-100S hall effect sensor and it was giving us inaccurate current data during competition in the summer. We spent months diving deep into what could cause this and the project took many turns. We started with characterizing the current sensor itself, then realized the ADC pin was the issue and tried characterizing it, then realized all ADC pins were off which is where we arrive today. This process took months mostly because we didn't understand these systems from first principles, now that I've done this testing, fixing this kind of issue in the future will take 10x-100x less time.

Current Percentage Error vs. Observed Grafana Current

We have about a constant 1-2 amp error, this matters more when current is low than high



Error Graph from our initial characterization of the current sensor

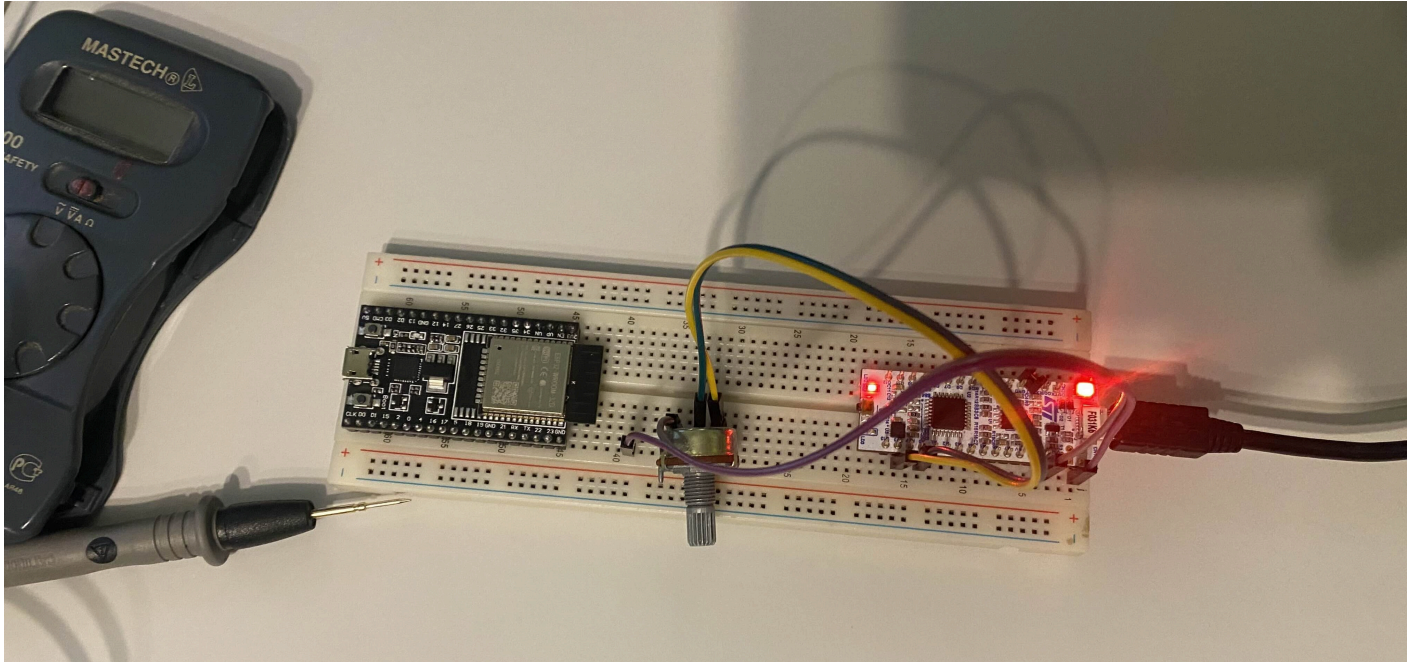
A few days ago I had my last final exam for the term, so the real work of figuring out how microcontrollers actually work could begin. After bouncing around a few of my existing projects I

decided to characterize the ADCs on my STM32 F031K6T6 and ESP32-WROOM-32D to get a better understanding of how to use them and learn more about how they work along the way.

All test data can be found [here](https://docs.google.com/spreadsheets/d/14OSSoQIURZ027khO06cPYy9pPMkBilPu_ZhBbcTdWYQ/edit?usp=sharing) (https://docs.google.com/spreadsheets/d/14OSSoQIURZ027khO06cPYy9pPMkBilPu_ZhBbcTdWYQ/edit?usp=sharing).

My code can be found [here](https://github.com/CKalitin/ESP32-WROOM-32D/blob/master/ADC-Self-Calibration/src/main.cpp) (<https://github.com/CKalitin/ESP32-WROOM-32D/blob/master/ADC-Self-Calibration/src/main.cpp>), if you look through previous commits you'll find earlier test code I wrote for this project.

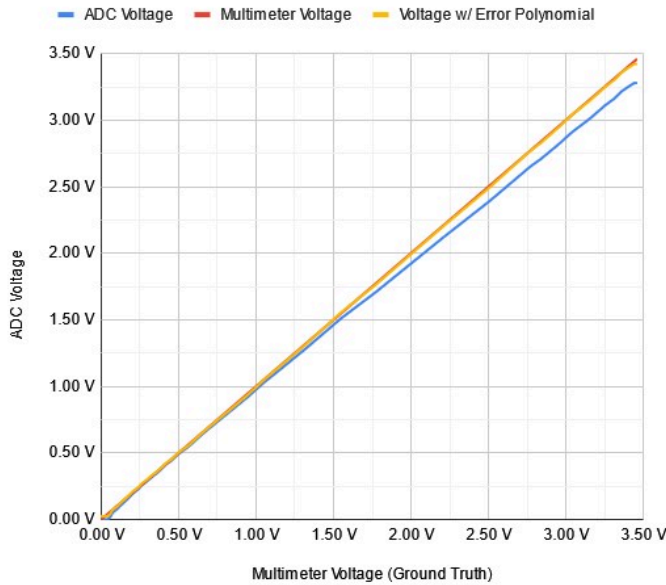
Initial Testing



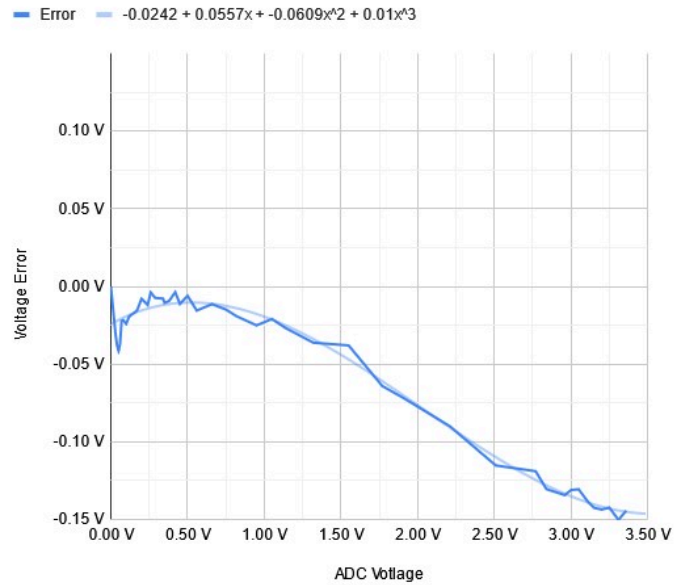
My Initial Testing Setup

I initially wanted a graph of ADC voltage vs. True voltage (as measured by a multimeter) for both microcontrollers. Both have 12-bit SAR ADCs so this testing is applicable to the work on the UBC Solar battery pack. I used a potentiometer to vary voltage between 0-3.3 to get a full range of values. I have no power supply (yet) so I must work with the tools I have. It doesn't look like the potentiometer introduced much error when comparing these results to those later in testing, it only introduced some noise.

STM32 ADC Voltage Reading vs. Multimeter Voltage

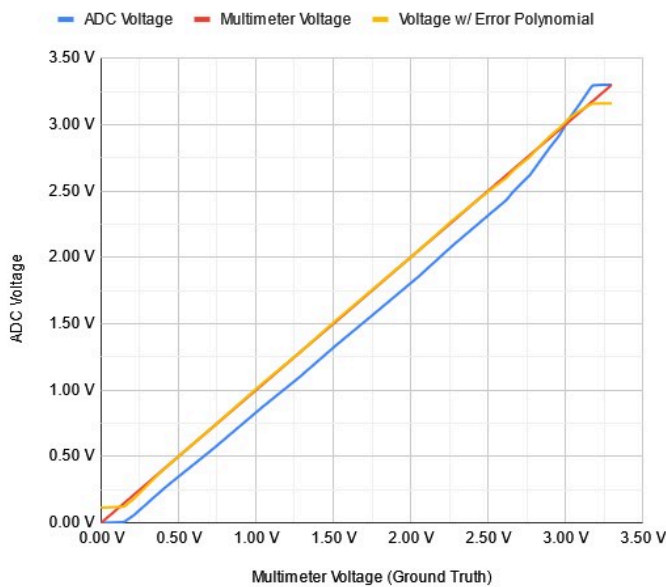


STM32 ADC Voltage Error vs. Multimeter Voltage

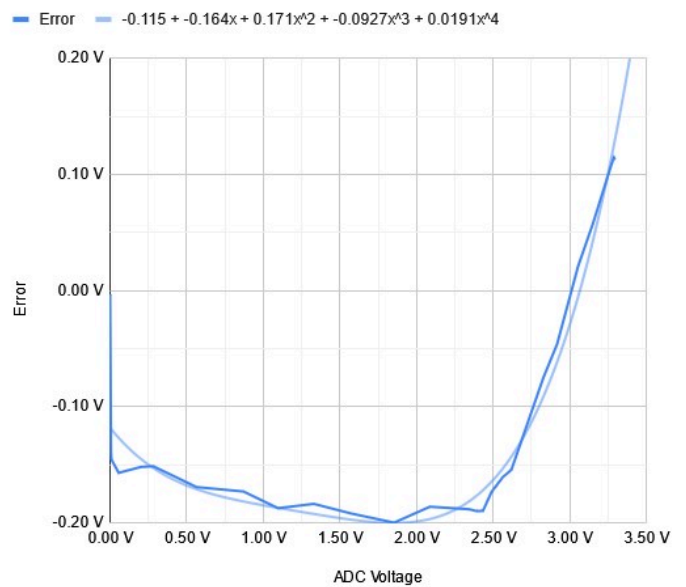


Blue Line = ADC Observed Voltage, Red Line = True Voltage (what we expect), Yellow Line = ADC Voltage plus Error Polynomial

ESP32 ADC Voltage Reading vs. Multimeter Voltage



ESP32 ADC Voltage Error vs. ADC Voltage



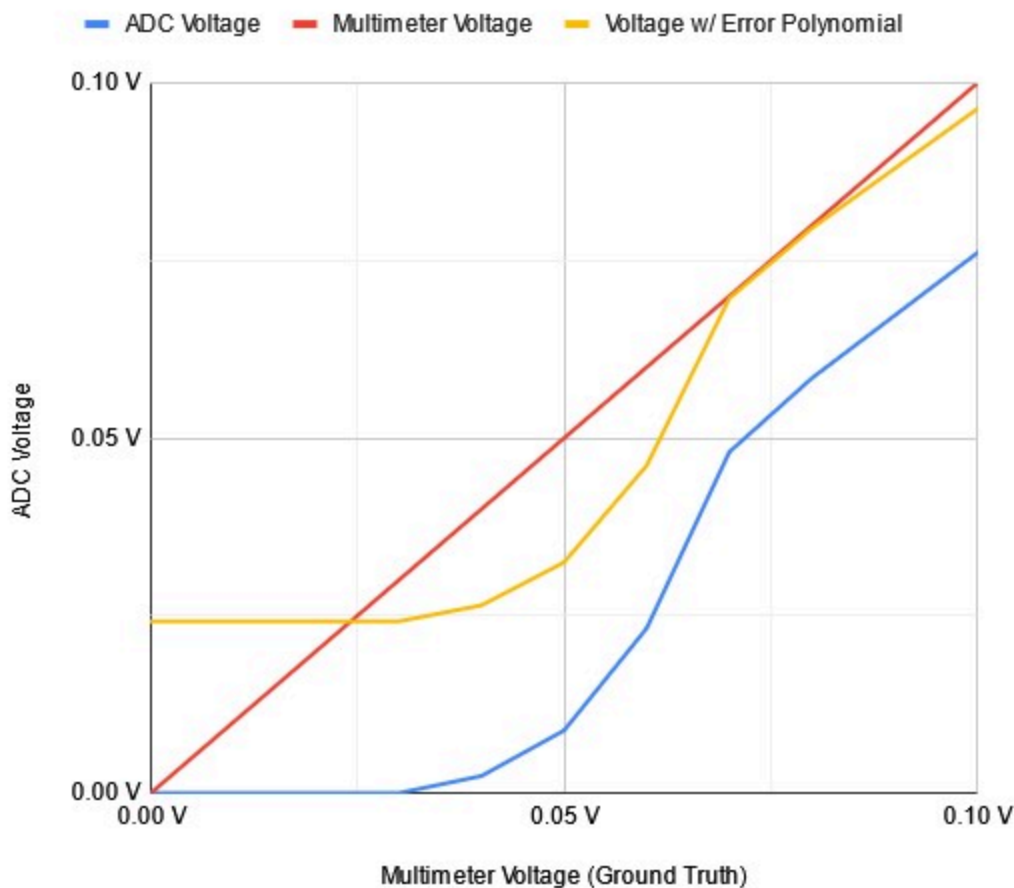
Blue Line = ADC Observed Voltage, Red Line = True Voltage (what we expect), Yellow Line = ADC Voltage plus Error Polynomial

The shape of the ESP32's ADC voltage graph perplexed me because it appears the first value it observed was 0.14V, and it stopped at 3.18V. Some googling & talking to Grok showed that below 0.1V or above 3.1V the ESP32's ADCs are not accurate. In the UBC Solar battery pack, we give the current sensor a 1.8V reference voltage to offset the readings. So, we are right in the middle of the

accurate range of the ESP32's microcontroller. STM32 chips have a similar range in which they are meant to operate accurately.

With the ADC observed voltage and the expected voltage, I fit a polynomial to the error using Google Sheet's built-in function and plotted the voltage + error polynomial on the graph. Note that I didn't redo the test here with the error polynomial applied in firmware, I just added it to the spreadsheet values. With the error polynomial applied, the STM32 and ESP32 ADCs became accurate to within 0.01V in most of the range (0.5V - 3.0V). The reason this range doesn't exactly match the expected 0.1V - 3.1V range that the ESP32 chip should be accurate in is because of improper application of the error polynomial, I should have deleted values that are outside the range I want.

STM32 ADC Voltage vs. Multimeter Voltage



Zoomed in view of the STM32 ADC Voltage vs. True Voltage

This zoomed-in view of the same STM32 ADC Voltage vs. Multimeter Voltage graph from above shows a strange non-linearity point in the graph. Above a voltage of 0.07V, we get a mostly linear error. However, below this, we see this strange non-linear exponential-looking part of the graph. I

don't understand Electrical Engineering or Physics at a low enough level to describe this, but it tells us that the reasonable lower range of the STM32's ADC is 0.07V.

1.2V AAA Battery Test		
Multimeter Voltage	ADC Reading	ADC Voltage
1.20 V	1270	1.02 V
1.20 V	1290	1.04 V
1.20 V	1310	1.06 V

Testing with a 1.2 V Battery

I also tested with a 1.2V AAA battery to ensure that the potentiometer was not an issue and got results that were similarly inaccurate. These results also had a lot of noise. Note that I converted the ADC Reading into a voltage in the spreadsheet with the formula $\text{Voltage} = (\text{ADC Reading} / 4095) * 3.3$. This assumes the reference voltage is 3.3V and converts the ADC reading assuming the conversions: 0 = 0V and 4095 = 3.3V.

Using the esp32-adc-calibration Github repo

```
269 3885.0000,3885.8000,3886.6001,3887.0000,3887.8000,3888.6001,3889.0000,3889.8000,3890.2000,3890.8000,3891.3999,3892.0000,3892.8000,3893.0000,3893.8000,
270 3894.3999,3895.0000,3895.8000,3896.0000,3896.8000,3897.3999,3898.0000,3898.8000,3899.0000,3899.8000,3900.3999,3901.0000,3901.6001,3902.0000,3902.8000,
271 3903.2000,3904.0000,3904.6001,3905.0000,3905.8000,3906.0000,3906.8000,3907.2000,3908.0000,3908.6001,3909.0000,3909.8000,3910.0000,3910.8000,3911.2000,
272 3912.0000,3912.6001,3913.0000,3913.8000,3914.0000,3914.8000,3915.2000,3915.8000,3916.6001,3917.0000,3917.8000,3918.0000,3918.8000,3919.2000,3919.8000,
273 3920.6001,3921.0000,3921.6001,3922.0000,3922.8000,3923.0000,3923.8000,3924.2000,3924.8000,3925.3999,3926.0000,3926.6001,3927.0000,3927.8000,3928.0000,
274 3928.8000,3929.0000,3929.8000,3930.2000,3930.8000,3931.3999,3932.0000,3932.6001,3933.0000,3933.8000,3934.0000,3934.8000,3935.0000,3935.8000,3936.2000,
275 3937.0000,3937.8000,3938.2000,3939.0000,3939.8000,3940.0000,3940.8000,3941.6001,3942.0000,3942.8000,3943.6001,3944.0000,3944.8000,3945.3999,3946.0000,
276 3946.8000,3947.3999,3948.0000,3948.8000,3949.3999,3950.0000,3950.8000,3951.2000,3952.0000,3952.8000,3953.2000,3954.0000,3954.8000,3955.2000,3956.0000,
277 3956.8000,3957.2000,3958.0000,3958.8000,3959.2000,3960.0000,3960.8000,3961.2000,3962.0000,3962.8000,3963.3999,3964.0000,3964.8000,3965.3999,3966.0000,
278 3966.8000,3967.3999,3968.0000,3968.8000,3969.2000,3969.8000,3970.6001,3971.0000,3971.8000,3972.2000,3973.0000,3973.6001,3974.0000,3974.8000,3975.3999,
279 3976.0000,3976.8000,3977.0000,3977.8000,3978.3999,3979.0000,3979.8000,3980.0000,3980.8000,3981.3999,3982.0000,3982.8000,3983.2000,3983.8000,3984.6001,
280 3985.0000,3986.0000,3986.8000,3987.3999,3988.0000,3988.8000,3989.6001,3990.0000,3991.0000,3991.8000,3992.3999,3993.0000,3993.8000,3994.6001,3995.0000,
281 3996.0000,3996.8000,3997.3999,3998.0000,3998.8000,3999.6001,4000.0000,4002.6001,4004.8000,4007.0000,4009.3999,4011.8000,4014.0000,4016.2000,4026.80
282 };
283
284 void setup() {
285     dac_output_enable(DAC_CHANNEL_1); // Enable DAC on pin 25
286     dac_output_voltage(DAC_CHANNEL_1, 0); // Setup output voltage to 0
287     analogReadResolution(12);
288     Serial.begin(500000);
289     while (!Serial) {}
290 }
291
292 void loop() {
293     for (int i=1; i<250; i++) {
294         dac_output_voltage(DAC_CHANNEL_1, i); // DAC output (8-bit resolution)
295         delayMicroseconds(100);
296         int rawReading = analogRead(ADC_PIN); // read value from ADC
297         int calibratedReading = (int)ADC_LUT[rawReading]; // get the calibrated value from LUT
298
299         // Run Serial Plotter to see the results
300         Serial.print(F("DAC = "));
301         Serial.print(i*16);
302         Serial.print(F(" rawReading = "));
303         Serial.print(rawReading);
304         Serial.print(F(" calibratedReading = "));
305         Serial.println(calibratedReading);
306         delay(10);
307     }
308 }
309 }
```

This is the code from the Github repo I used, notice the line count and the massive array at the top

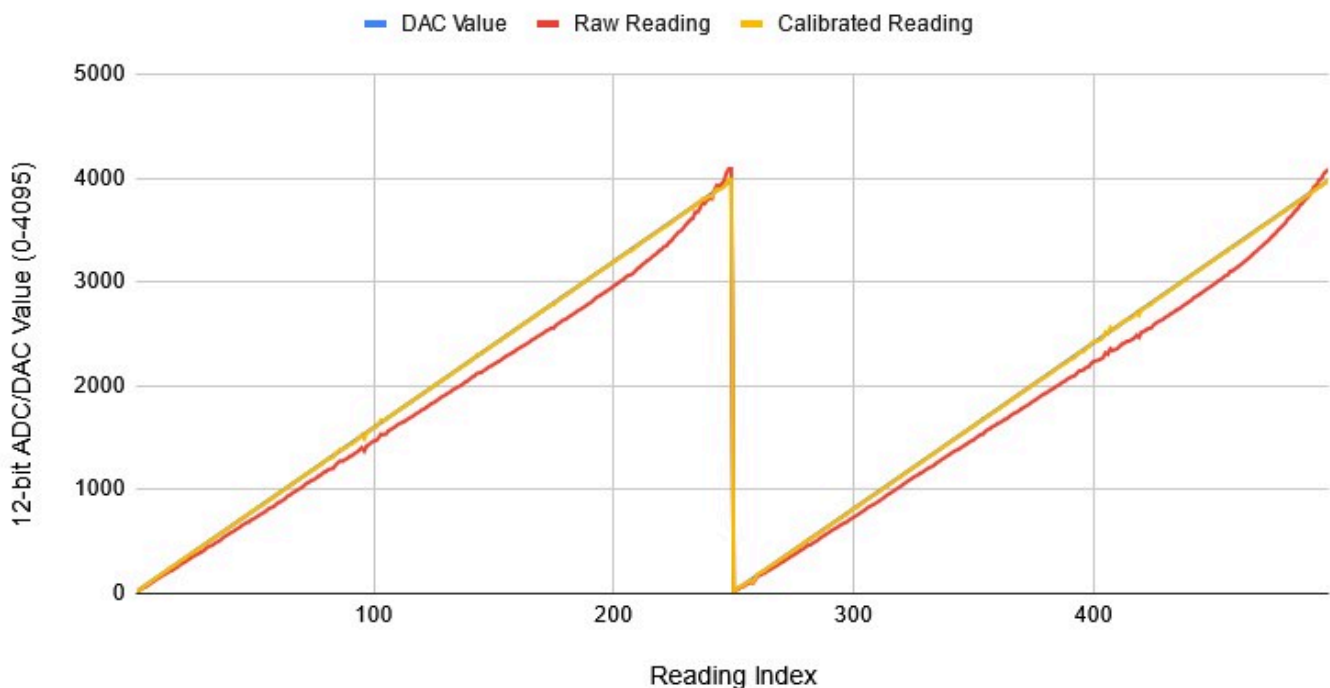
that stores the lookup table.

In researching I found [this Github repo](https://github.com/e-tinkers/esp32-adc-calibrate) (<https://github.com/e-tinkers/esp32-adc-calibrate>) that uses the ESP32's DAC (Digital to Analogue Voltage Converter) to get the chip to characterize itself. If you have a known voltage output from the DAC, you can pipe this directly into the ADC and find your error in firmware instead of on a spreadsheet. This Github repo then takes the error values and feeds them into an Arduino .ino program (ESP32's can run using the Arduino IDE) to get a lookup table of errors. Essentially, it records 256 values and for each range in between them, it calculates the error. With these ranges, you can use them as a lookup table where you input your ADC value as a key and get the expected ADC value back for that particular ADC voltage.

This lookup table approach required 16 KB of flash memory on the ESP32. If we really wanted to do this on the UBC Solar battery pack we could shrink the lookup table to 128 values and probably have enough flash left over for the other programs. However, this is still a very stupid idea. We aren't software devs that can throw memory and compute at all of our problems.

DAC Value, Raw Reading, Calibrated Reading vs. Reading Index

ADC2 - Corrected With a Lookup Table



Blue Line = DAC Output Voltage, Red Line = Unadjusted ADC Reading, Yellow Line = ADC Reading Adjusted with Lookup Table

The shape of the error looks very similar in this test and the calibrated reading is spot on with what we expect. However, I later found out that the output of the DAC cannot be trusted. When testing

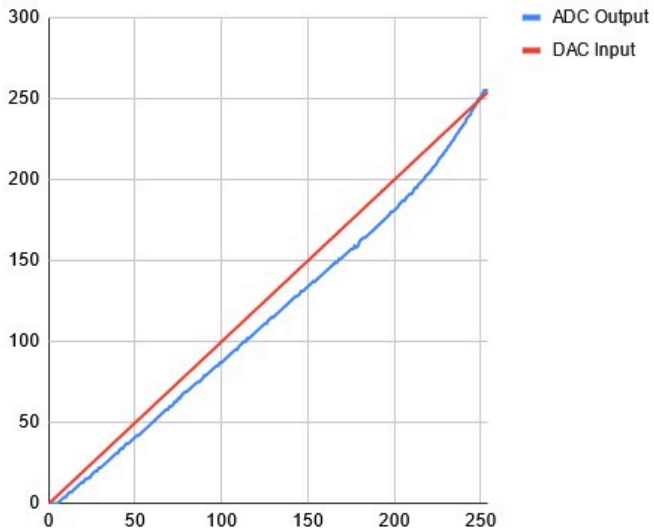
with a multimeter, its output was inaccurate in a similar way to the ADC. So, this kind of test where you use the DAC to calibrate the ADC is not possible on my ESP32 chip.

I initially wanted to write my own self-calibration script for my STM32, but after realizing the DAC may not be trustworthy and the fact that unlike UBC Solar's STM32 chip mine doesn't have a DAC, I decided to abandon this idea. If on Solar we find that the DAC on our chip is trustworthy, this could be a good way to calibrate the ADCs on any of our boards, not just the ECU (Elithion Control Unit) in the Battery Pack.

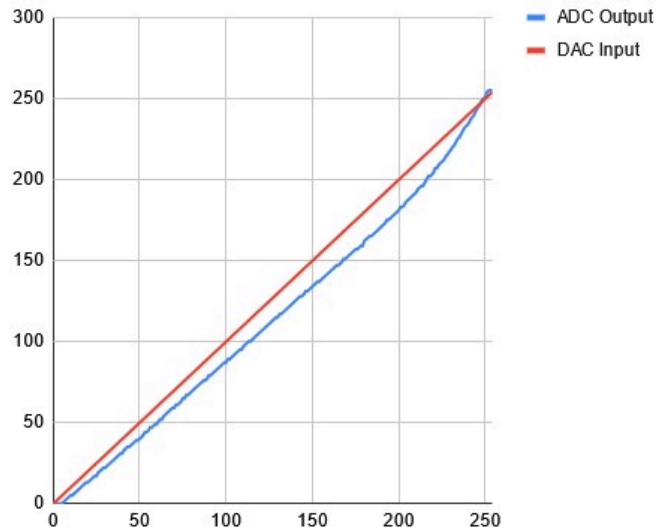
Testing Attenuation

In the [README](https://github.com/e-tinkers/esp32-adc-calibrate/blob/master/README.md) (<https://github.com/e-tinkers/esp32-adc-calibrate/blob/master/README.md>) for the esp32-adc-calibrate repo, the author mentions that reading the [ADC Calibration](https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/adc_calibration.html) (https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/adc_calibration.html) documentation is a good first step before implementing the code. I read it and found the `esp_adc_cal_characterize()` function that allows you to set the attenuation, bit width, and other parameters for either of the two ADCs on the ESP32-WROOM-32D. There's no automatic ADC calibration, but I could set parameters and record the results.

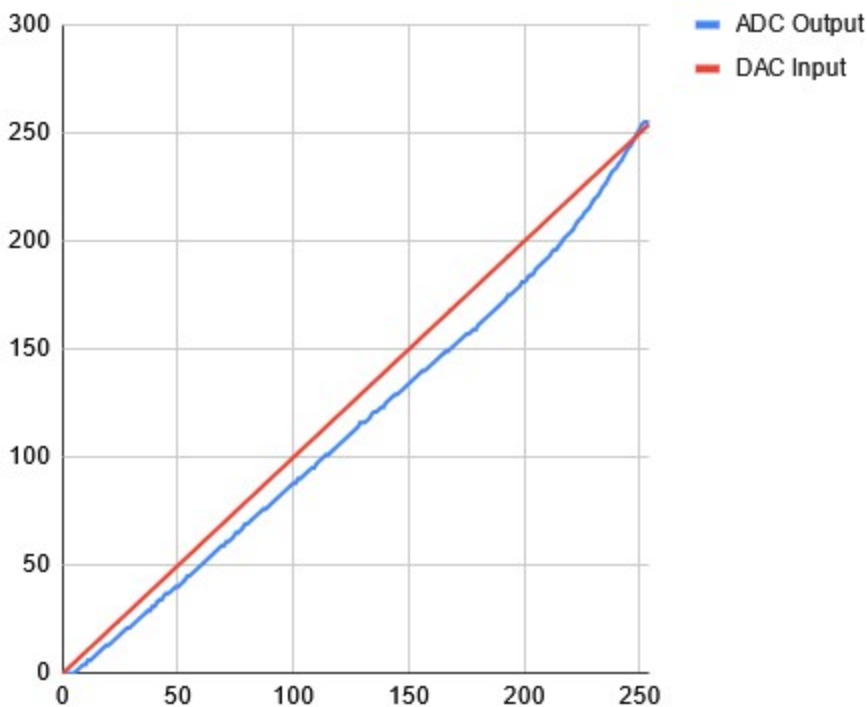
ADC Output vs. DAC Input (0 dB Attenuation)



ADC Output vs. DAC Input (2.5 dB Attenuation)



ADC Output vs. DAC Input (12 dB Attenuation)



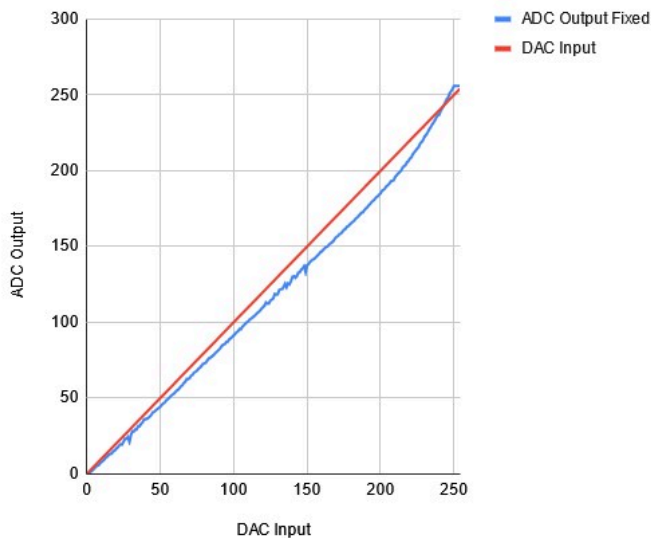
Little difference was found between all the attenuation values.

I [wrote some firmware](https://github.com/CKalitin/ESP32-WROOM-32D/blob/master/ADC-Self-Calibration/src/main.cpp) (<https://github.com/CKalitin/ESP32-WROOM-32D/blob/master/ADC-Self-Calibration/src/main.cpp>) inspired by the Github repo above that automatically tested the ADC given a range of DAC values. This was before I found out the DAC is not trustworthy to output specific voltages within $\sim 0.1V$ in most of the range while I needed $\sim 0.01V$ precision.

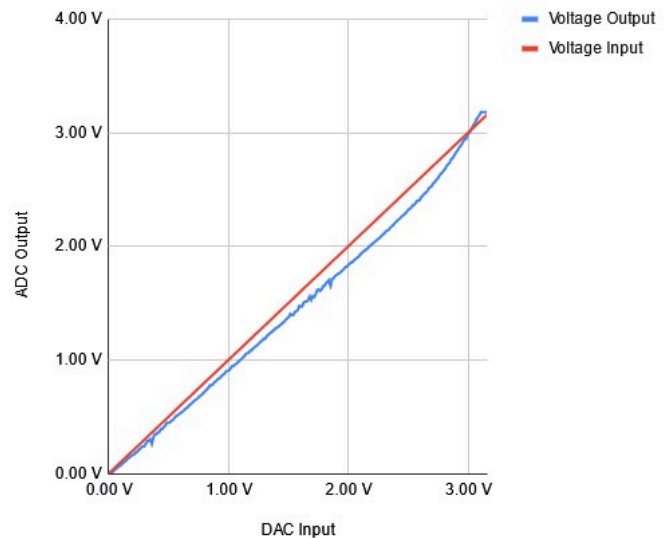
With that firmware, I automatically tested the attenuation values of 0 dB, 2.5dB, and 12dB to see if there was any difference and found none. Attenuation is essentially signal boosting, and I was using a small jumper cable, so it made little difference. However, when swapping out the jumper cable for a short paper clip I found that the noise was reduced. When I get into designing PCBs I'm sure I'll understand this phenomenon better.

Raw ADC Values Testing

ADC Output vs. DAC Input (adc2) (raw, db0, width = 12)



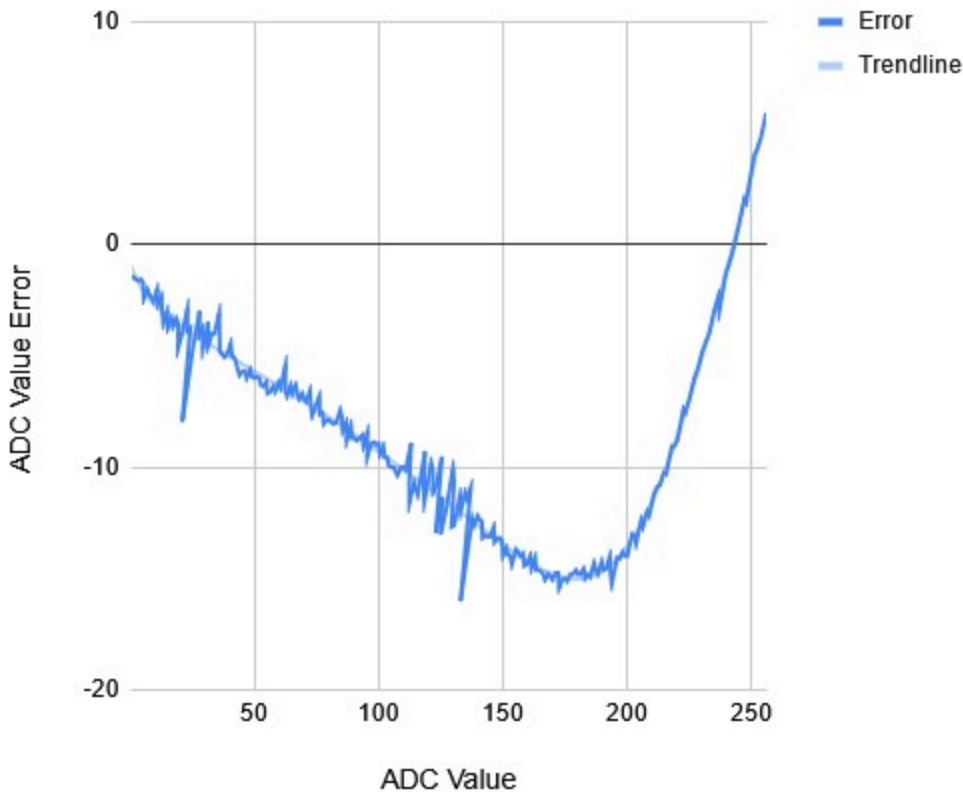
ADC 2 Voltage Output vs. Voltage Input



I tested ADC 2 with lower-level functions and found little improvement.

Looking further into the ESP32 documentation, I found lower-level libraries that can be used to get the raw ADC values directly instead of interfacing through the Arduino library. I wrote a quick function to convert this raw value into a voltage and tested it using the same “DAC voltage into ADC” strategy as before. I found no difference between my own custom conversion function and the Arduino library’s `analogRead()` function. I also tested the second ADC on my ESP32 and found little difference in the results compared to the first ADC. However, the second ADC was accurate down to 0V while ADC 1 was accurate down to 0.08V.

ADC Value Error vs. ADC Value



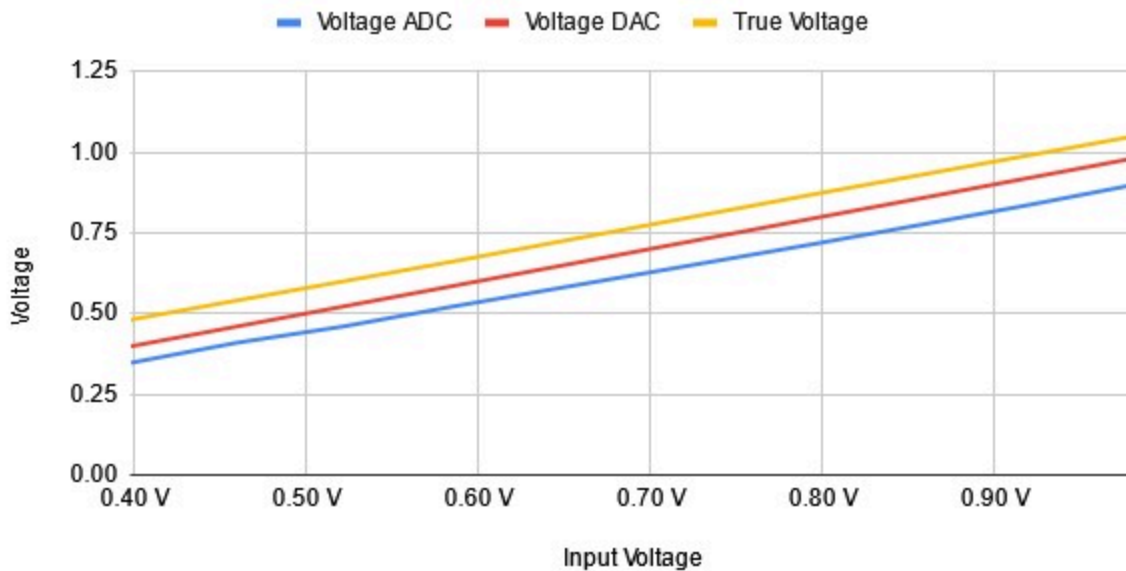
With the graphs above, I got the error graph and fit a Polynomial (trendline) to it.

I then got the error for ADC 2 in raw bits. Instead of converting to voltages and then getting the error, I skipped that step and got the error in raw ADC values between 0-255. This isn't keeping with the ADC's 12-bit range because the DAC only goes to 8-bits, so I was limited to 8-bits of precision. I then fit a polynomial to the error and got the error graph above. The error was very similar to earlier testing, so nothing I had tried fixed the problem and mostly served as an exercise in understanding how to use the ADC on my ESP32.

Discovering the DAC was Imprecise

Voltage Output, Voltage Input, True Voltage vs. Voltage Input

Blue = ADC Reading, Red = DAC Output, Yellow = Real Voltage (Multimeter)



I tested the ADC and DAC with a multimeter and found that all the values differed.

During the previous round of testing with the raw ADC values I decided to take out a multimeter and get a ground truth value for the voltage. I discovered a discrepancy between the DAC's expected output voltage and the voltages the multimeter was showing. I tested it on a range of values between 0.4V and 1.0V and found a difference along the entire range - this is the data charted above.

Above you can see the True (multimeter) voltage, what the DAC thinks it's outputting, and what the ADC thinks the voltage is. All of these values differ by a significant amount (~0.5 - 0.1V). This means a self-calibration is not possible because the DAC's specified output and what it actually outputs are not equal.

Conclusion

To properly characterize and calibrate the ADC I would have to get the graph of ADC Value vs. True Voltage (As observed by a multimeter) and fit a polynomial to it in the range that the ADC error is mostly linear (0.1V - 3.0V). This is probably the solution to the issue we've had with the UBC Solar battery pack. However, if we find that the DAC on our chip is trustworthy, writing some firmware to automatically calibrate the ADCs on all of our boards would be a great way to ensure we never have this issue in the future on any boards.

This solution looks very obvious in retrospect, but it took a while to understand the systems and issues from a fundamental level to come to this conclusion with a high level of certainty. I've

learned a lot about this project and there's even more to learn about why these ADC errors emerge in the first place.