# CSE 157 - Lab 2

Dallas Meyer

October 27, 2023

## 1 Starting Out: learning about the sensors

When figuring out how to connect the sensors, I thoroughly read through specific parts of the documentation. This is in terms of how I2C works to gather a better understanding of how all these sensors can connect. For the sensor store pages, guides, and libraries, I mostly skimmed through them as there wasn't much conceptual learning and rather is better suited as a reference for when I need to actually set those up.

This approach did work out, as I had a good starting point on how I2C works and what the sensors do. As when connecting the sensors, I ended up going back and forth to these pages when needed.

I didn't have a clear understanding of what the ADC does or how it works. The good news is that I there was really useful guide within the Adafruit ADC store page description. I did try searching online for how it works, but found that the stores description's guide to be really effective. As it went into detail what each pin does, the different modes, etc.

### 1.1 Setting up the CircuitPython and Adafruit Blinka Libraries

The Adafruit Blinka and CircuitPython library installation was somewhat straightforward. Following the included Adafruit tutorial (learn.adafruit.com/circuitpython-on-raspberrypi-linux) listed all of the needed commands.

There was one issue I had, where the last command in the automated install section (see fig.2) – "sudo python3 -E env PATH=$PATH raspi-blinka.py" – did not work as the RPi didn't have those directories. This

required me to omit the "env PATH=$PATH" to get the script to run. Although this could have been attributed from not setting a virtual environment. See figure 1.

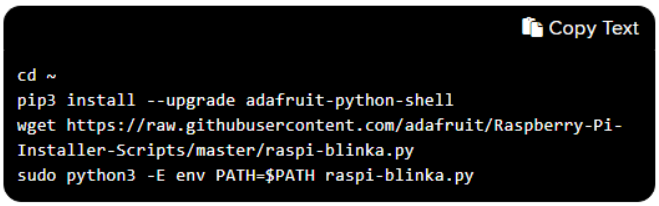


Figure 1: Automated Install commands, where last command needed to be modified.

After running that python script, we can check whether I2C and SPI were enabled.

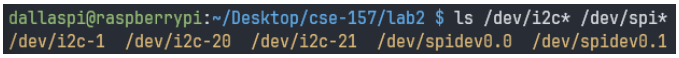


Figure 2: Confirming that I2C and SPI were enabled

## 1.2 Missing documentation

The Adafruit website sensor guides have lots of good information and visual schematics. However it lacked information specifically on what the GPIO pins were for the Raspberry Pi. To find this information, I simply did a google search for "Raspberry Pi pinout diagram", and found it on the pinout.xyz website (figure 3).

In addition when setting up the Anemometer, there wasn't much information on how to connect it. However within the included anemometer product page, there was a weather station guide that uses the sensor (learn.adafruit.com/pyportal-iot-weather-station/assembly-and-wiring) and provided the wiring. The trick was to connect the wind sensor's ground to the RPI's ground.

## 1.3 The Breadboard and the Raspberry Pi

As seen in figure 4, there are 4 wires connected to the RPi's GPIO pins. Where importantly is the white data wire and brown clock wire to do I2C.

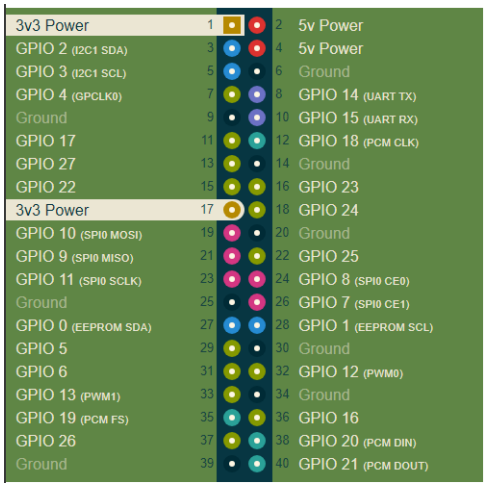


Figure 3: Raspberry Pi GPIO Pins (pinout.xyz)

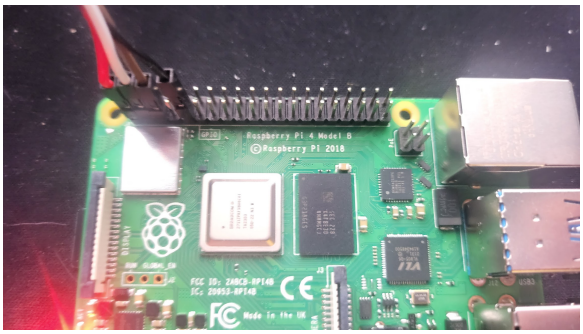

Figure 4: RPi's wiring,

- Red = 3v3
- Black = gnd
- White = SDA
- Brown = SCL

In figure 5, in the top-left we have the positive and ground in their respective rails at the top of the breadboard. The I2C wires are also seen, white as the data and brown as the clock.

The SHT30 humidity sensor is the green box at the top left. Between the two green "wire boxes" the is the moisture sensor (4 wires). The right

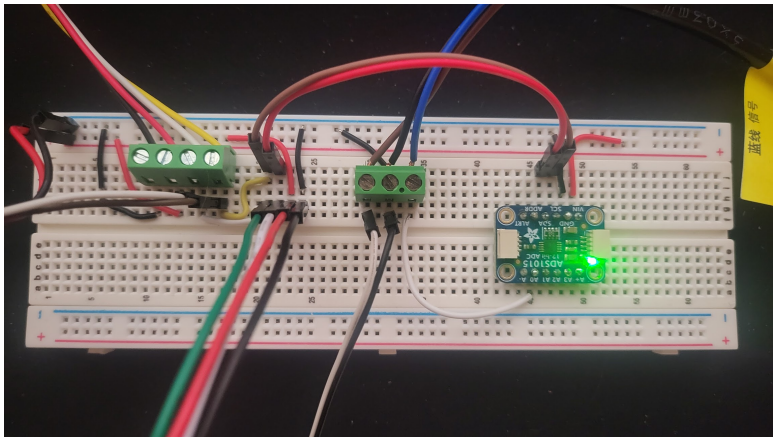

Figure 5: Breadboard wiring. Red = positive. Black = ground. White = data. Yellow/brown = clock

green wire box goes to the anemometer. The blue chip/device is obviously the ADC sensor. In the ADC sensor's "inputs" at A0, there is a white wire connected to the blue voltage/data wire that is sent from the anemometer.

## 2 Interfacing with the Sensors

To interface with these sensors, the I2C protocol was used here. This was implemented through the Python's Blinka library.

An example is seen below for the SHT30 sensor, where its mostly just creating a sensor object from the library, and reading from it.

```
1  import board
2  import busio
3  import adafruit_sht31d
4  i2c = busio.I2C(board.SCL, board.SDA)
5  sensor = adafruit_sht31d.SHT31D(i2c)
6
7  print('Humidity: {0}%'.format(sensor.relative_humidity))
8  print('Temperature: {0}C'.format(sensor.temperature))
```

Figure 6: Demo python script that reads prints the humidity & temperature of the SHT30 sensor.

## 2.1 I2C protocol

The I2C protocol works through using two wires to connect from the parent/computer to the child/sensor.

One wire is the "data" wire, which sends data through a "frame". This frame contains an address to indicate which sensor to communicate with, along with other info such as reading/writing, a ack/nack, and start & stop conditions.

The other wire provides the "clock" from the parent, so that the sensor can read and send data reliably as it synchronizes its clocks with the computer.

## 2.2 Compared to other Protocols

The I2C (inter-integrates circuit) protocol offers simplicity and flexibility in that it only requires 2 lines in the bus, making the wiring less messy when using numerous devices. UART also uses two wires. On the other hand, the SPI (serial peripheral interface) protocol requires 4 wires: parent data input/output and child input/output, parent clock signal, and a child signal.

However SPI does have its advantage that it can communicate without using addresses (preventing address conflicts) and also has better speed. In addition both SPI and UART it has full-duplex or simultaneous communication back and forth, which I2C doesn't have. This makes I2C's communication a bit more complex.

As for data size, I2C still has a good amount of data able to transfered compared to UART's smaller data size and SPI larger data size.

Overall I2C is more middle of the road specs. It also has advantage of using multiple parents and types of parent devices, of which UART and SPI don't support.

## 2.3 Unique Address

I noticed that if the sensors are unable to be found, then an error message would show up. Within this message also shows the sensor's address. From this I noticed that the sensors have unique addresses.

- SHT30 - 0x44
- Soil/Seesaw - 0x36

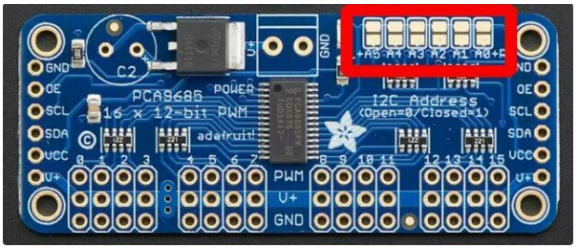


Figure 7: Example of address jumper pads used to modify the address. (learn.adafruit.com)

- ADC - 0x48

### 2.3.1 Address conflicts

If both sensors have the same address, then we could have address conflicts where both receive the same message! Not good.

One workaround is to use a secondary I2C port, however the RPi doesn't appear to support this.

Another way is to modify the sensor. Specifically the ADC sensor has one address jumper labeled A0. Soldering this jumper closed allows the I2C address to go from 0x48 to 0x49. An example of this is in figure 7.

In the case that the above alternative addresses is not an option, we can use an I2C multiplexer to resolve this address conflicts.

# 3 Polling the Sensors in a Program

In this part of the lab I wrote a python file that periodically read or polled the three sensors. After getting the sensors info, it then saves it into a text file.

In relation to hardware, having multiple sensors made it a tad bit more difficult since it required connecting the same SDA and SCL wires. As such, it required a bit of daisy chaining.

As for the code, it was relatively simple as for this step I had already written the scripts for each individual sensor. Here I just had to re-use the code.

As seen in figure 8, my name and date is also written into the file. This show cases the wind's speed, which required converting the voltage into m/s.

```
Dallas Meyer
10-27-2023 13:54:51
Temperature: 20.298°C
Humidity: 46.282%
Soil Moisture: 340.000
Soil Temperature: 23.641°C
Wind speed: 0.000m/s
```

Figure 8: Sensor-polling text output

Since the voltage range was from 0.4v to 2v and the max speed was 32.4m/s, the conversion formula was (voltage - 0.4v)*32.4m/s÷2$v$.

```
with open("polling-log.txt", "w") as file:
    file.write("Dallas Meyer\n")
    while(True):
        # time
        now = datetime.now() # https://www.programiz.com/python-programming/datetime/current-datetime
        t = now.strftime("%H:%M:%S")
        date = now.strftime("%m-%d-%Y")
        file.write(date + " " + t + "\n")
        file.write('Temperature: {0:.3f}°C\n'.format(sht30_sensor.temperature))
        file.write('Humidity: {0:.3f}%\n'.format(sht30_sensor.relative_humidity))
        file.write('Soil Moisture: {0:.3f}\n'.format(ss.moisture_read()))
        file.write('Soil Temperature: {0:.3f}°C\n'.format(ss.get_temp()))
        chan = AnalogIn(ads, ADS.P0) # pin 0 analog input channel
        # convert voltage, where min appears to be 4.06
        file.write('Wind speed: {0:.3f}m/s\n\n'.format((chan.voltage - 0.406)*32.4/2))
        print("wrote to file")
        time.sleep(5)
```

Figure 9: Main loop of sensor-polling.py

When the program is ran, it goes through a infinite while loop. Python's "sleep" function was used here so that it can poll the sensors every 5 seconds and write it into a file, as can be seen in figure 9.

## 3.1 Implementing the Sensors

When implementing these sensors, it should be noted that the SHT30, soil sensor, and wind sensor all required different Adafruit libraries. This required lots of different imports, along with different sensor "objects" to interact with. This made it a bit tricky, since I had to keep track of what libraries

overlapped. As for getting the data from these sensors when, I noticed that the soil/seesaw sensor object used "get" functions. Whereas the SHT30 and wind sensor object accessed a variable. Since they had different naming on how to access these data values, I had to keep track of them. Regardless it was simple to just combine them all in one place.

The ADC sensor required setting up the most settings, as different analog input channels can be used. Thus a channel had to be declared in the code, where pin 0 was used in this case. As such I had to do a bit more wiring for the ADC as well to read the data from the wind sensor.

### 3.2 Functionality

Since each sensor data is accessed in a certain order (SHT30, soil, then wind), it doesn't poll the sensors at the exact same time. Realistically it does poll it at the same time, but technically it doesn't. This is due to program counter waiting for all the data from one sensor, before moving on to collecting the next sensor's data. In addition some sensors may require a bit more time to process and send data.

## 4 Cooperative multitasking

```
41 async def sht30_function(sht30_sensor, interval=5):
42     while True:
43         temp = sht30_sensor.temperature
44         hum = sht30_sensor.relative_humidity
45         logger_1.info(f"Current time: {datetime.datetime.no
46         logger_1.info('Temperature: {0:.3f}°C'.format(temp)
47         logger_1.info('Humidity: {0:.3f}%\n'.format(hum))
48         await asyncio.sleep(interval)
```

Figure 10: The SHT30 coroutine/function from concurrent-sensor-reading.py

In the next part of the lab, cooperative multi-tasking is implemented through the Python AsyncIO library. This library has **asyncio.sleep()** functions that don't block code from running. Instead this function allows other "async" routines/tasks to be ran in the meantime. An example is seen in figure 10, where the SHT30 sensor function collects the sensor's data. Once it reaches the "await" function, the code must sleep for a certain amount of

time (5 seconds in this case) before restarting that loop. The advantage is that as it is in sleep, the program can now focus on a different task until the sleep function is done/expired.

This is done for the other sensors as well, where instead they have their own individual asyncio.sleep() functions.

```
74 async def main():
75     """
76     The main coroutine, just awaits our concurrent functions.
77     """
78
79     # init sht30
80     import adafruit_sht31d
81     i2c = busio.I2C(board.SCL, board.SDA)
82     sht30_sensor = adafruit_sht31d.SHT31D(i2c)
83
84     # init moisture
85     from adafruit_seesaw.seesaw import Seesaw
86     i2c_bus = board.I2C()  # uses board.SCL and board.SDA
87     ss = Seesaw(i2c_bus, addr=0x36)
88
89     # init anemometer
90     # Voltage [0.4v,2.0v] max 32.4m/s wind speed
91     import adafruit_ads1x15.ads1015 as ADS
92     from adafruit_ads1x15.analog_in import AnalogIn
93     ads = ADS.ADS1015(i2c) # ADC object
94
95
96     await asyncio.gather(
97         sht30_function(sht30_sensor),
98         soil_function(ss)
99         wind_function(ads,AnalogIn, ADS)
100    )
```

Figure 11: Main function in the concurrent-sensor-reading.py

As for the main loop (see figure 11, it also uses an "await" keyword, where it waits for the three sensor coroutines to finish. However since they are infinite while loops, the program will be multitasking forever.

## 4.1 Comparing Cooperative Multitasking and Parallelism

Taking a step-back, **Cooperative multi-tasking** is a system that works on single-thread devices to execute multiple tasks at once, through yielding or taking turns. Parallelism is performing multiple tasks at the same time,

using multiple cores or threads for each task. However since cooperative multi-tasking works on single-threading or one device, it is instead a style of concurrent programming.

The advantage of using Async IO is that it doesn't need the complex techniques (eg. blocking) that parallel processing requires. Especially in regards to keeping data relevant and up to date across multiple threads.

The main task runner in cooperative multitasking, is the CPU core, as it simply follows the program counter to determine what commands to execute. Regardless, two tasks cannot run simultaneously in this situation, as it simply takes a break from a task for a certain amount of time, and works on other tasks in the meantime.

### 4.2 Cooperative Multi-tasking applications

Cooperative multi-tasking or Async IO works really well when needing to execute multiple tasks that could be delayed from "sleep" or other non-blocking functions.

Examples include a network setting, where the program acts as a client or server and shouldn't keep busy-waiting.

Another example is doing read/write operations where instead of using locks or mutexes, we can use Async IO.

The issue is that if the function within the co-routine still blocks (eg. has a "sleep" function), then the advantage for Async IO is not there.

## 5 Cooperative Multi-Tasking Across Different Machines

When looking at larger distributed systems, if each computer/device has its own timer or clock, then the devices risk of going out of sync. If a main server/computer has its own timer process, then it can ensure that all devices are at acting upon the timer at relatively the same pace. This can be important if one of the devices ends up having an error, where its own timer is thrown off from every other device.

### 5.1 Pros and Cons of Cooperative Multitasking over to serial polling

When comparing the Cooperative multitasking implementation with the serial polling, there are pro's and con's with each.

With the serial polling, all the sensors can be ran at the same time reliably because of that main timer. However the issue is that we can poll each sensor at different times. Let say if we did want to have different timers for each sensor, then the "sleep" functions of longer timers would still end up blocking the other timers from running.

This is the advantage of cooperative multitasking, in that it allows the use of multiple different timers where certain coroutines can be ran in out of order.

## 6 Conclusion

For the breadboard and sensors, it is a very hands-on experience. The breadboard itself makes it very convenient to create these circuits that allow the sensors to be powered on and work.

There also appears to be lots of useful online documentation regarding these sensors and its implementations in Python.

In addition, the I2C protocol seems very simple to set up, and has its pros and cons compared to other protocols. This is because SPI would require lots of wiring that would've made this lab more time-consuming.

Overall, this python-based lab proves how with the use of certain libraries, data from sensors can be read relatively easily. This is because python is more user-friendly, and doesn't require complex memory management or pointers.

## 7 Extra: Comparing first approach in Part 3 with second approach of cooperative multiasking

In part 3, each sensor had its own timer. Whereas in the extra-credit, each sensor was activated from a separate coroutine timer. The plus of this coroutine timer is that it again synchronizes all the sensors to work all at once, without each sensor having to wait on each other.

The part 3 implementation did make it simple to adjust all the different timing intervals for each sensor, so that the wind sensor can be read more frequently than the soil sensor. However if one of the sensors takes a while to read data, then eventually the timers can go out of sync.