CEG-7360: Embedded Systems, Fall 2025 with Bayley King

Final Project Report

The PicoAirSense Indoor Air Quality Monitor

Adrien Abbey – abbey.12@wright.edu
12-10-2025

Introduction

Brief Overview: PicoAirSense

This course helped me bridge the gap from having hardware project ideas which I lacked the background knowledge to even beginning working on to having all the tools and skills I needed to make them happen. One such idea popped into my head when the seasons started to change, and I realized I could finally start looking into building a system to monitor the humidity and air quality in my apartment. With a little bit of research, I found that I could purchase all the parts needed at a very reasonable price too. And thus, PicoAirSense was born. This was something I have been wanting to build for about a month now, and I have already done quite a bit of research and documentation into the steps I would need to make it happen.

The overall goal of the PicoAirSense is to provide a device with an e-ink display that will measure and display the air quality, humidity, and temperature inside one's home. Knowing these values can help people have a more accurate understanding of the conditions in their home. By making this design open source and using common off-the-shelf parts, anyone can assemble and pull the code from the public GitHub repository to build their own. Finally, and most importantly to me personally, this opens the door for me to gain valuable experience while also creating a noteworthy project that I can put on my resume and share with the world.

Heilmeier Catechism

What are you trying to do? Articulate your objectives using absolutely no jargon.

I am trying to create an indoor air quality monitoring device that measures and displays air quality, humidity, and temperature. This serves a multitude of objectives: it fulfills the requirements of this project, it allows me to create something I've been wanting to create, creates a publicly available project that I can add to my resume, gives me an opportunity to build valuable skills, while also allowing me to potentially contribute something meaningful to the open source community.

I also intend to write a MicroPython library for the BME280 sensor for this project. Writing this library has the added benefit that I would need to learn how to implement I2C communications with the sensor, as well as how to properly read and calculate the sensor values on a low level.

How is it done today, and what are the limits of current practice?

Speaking of commercial products available today, there are already several available for purchase online. These products range widely in terms of sensors, features, price, and quality, and many of them include IoT features that users may or may not want. While this is a viable option for many, it is not a perfect fit for everyone. Purchasing a commercially available product does not present the opportunity for users to learn how to assemble and program their own devices, and the lack of open-source firmware means they are not extensible should the user wish to build upon it.

There are several open-source projects available online that aim to accomplish similar goals. The microcontroller, sensors and software tooling used to accomplish the goals vary widely however, and I could not find any repo that uses the same parts I am using. There are many projects with some degree of overlap in the design, and while I could take their projects and adapt them to meet my needs, I have chosen not to.

What is new in your approach and why do you think it will be successful?

If I am measuring success based on my above objectives (the project itself, resume building, and personal growth), then I would argue that completing this project would be a success all in its own. This project is unique in the specific parts being used to create it, meaning it could be valuable to someone who wants to use the same parts. The skills and knowledge I will gain while working on this project would also be something I can share with others on campus, especially given my involvement in the Piano Staircase project, knowledge which we are currently severely lacking among current participants.

Who cares? If you are successful, what difference will it make?

Given that this project is directly tied to my Embedded Systems course, I care, and hopefully my instructor cares about my success as well. This means my success would not only help keep my GPA up, but also reflect well on my instructor, as this course was instrumental in making this project a reality. This also allows me to demonstrate my personal and professional skills in a way that I can show potential employers, which could not only improve my chances at landing a respectable job, but also potentially help them find a qualified and highly skilled employee in the future. Finally, it also provides me with a strong knowledge base from which I can help mentor others, which will directly improve the chances of success for the Piano Staircase, a project that will potentially influence future students to choose to study at Wright State University.

What are the mid-term and final “exams” to check for success?

I have already done a lot of research into how I intend to build this project, but I have barely begun the actual implementation process. One of the requirements for this project (and the most difficult) is the creation of a MicroPython library for the BME280 sensor, which I would say would make for a good "mid-term exam" for this project. The "final exam" would then be connecting everything together, reading all the sensor values and displaying them on the e-ink display.

What Changed

While getting my code together and evaluating the hardware, I ran into an issue where my e-ink display was failing to function. At first I thought it might be a wiring issue or problem with my code, but during testing, I noticed that the ribbon cable connecting the display to its controller board had a small rip in it, severing one of the connectors in it. RIP. Given the quickly approaching deadline and the time it would take for a replacement to arrive, I made the decision to table the display for now and focus on ensuring everything else functioned appropriately. As it is now, I can monitor the sensor data from the REPL terminal on a connected PC.

This also meant I could pivot the project to real-time monitoring with the possibility of interactive controls. I added additional functions that could be called from the REPL terminal, such as the ability to manually save calibration data. I also added experimental personalized metrics, such as perceived comfort based on the current temperature and humidity. While the product loses some of its appeal due to its dependence on being connected to a PC to read data, it instead gains the ability to provide real-time results with a history displayed on a larger display.

Methodology

At the heart of the PicoAirSense is a Raspberry Pi Pico 2 W, acting as a low-cost microcontroller providing enough processing power to read and process the sensor inputs. For the sensors, I used a BME280 to monitor temperature, humidity, and barometric pressure alongside a SGP30 to monitor TVOC and eCO2 levels. These both shared the same I2C bus, power, and ground from the Pico. To keep things simple and easy to maintain, these were all connected directly to the same bread board, making wiring easy and dependable.

On the software side, I decided to flash MicroPython firmware to the Pico. While I could have used the official C SDK instead, I felt more comfortable working in MicroPython, which was both straightforward and sufficient for this project. While open source MicroPython libraries for the SGP30 sensor are available, I could not find a BME280 library I was comfortable working with. Rather than see this as a limitation, I instead saw it as an opportunity to write my own library. This not only would give me valuable experience and insight into how these libraries work but also function as a valuable addition to my resume, as well as be something I can share with the rest of the world. As such, I have decided to use the MIT license for both the BME280 library and my main program.

Speaking of which, all the code is available on my public GitHub page here: https://github.com/adrienabbey/PicoAirSense

Basic Testing with MicroPython

With these goals in mind, the first step was to wire the hardware together and run a simple test harness. The wiring details are available at the link above. With the sensors connected, the Pico can then be connected to a PC using a USB cable. Holding the BOOTSEL button on the Pico while plugging the USB cable into a PC will cause it to show up as a USB flash drive, allowing the user to copy the MicroPython file from the official website over to the device. Once the file is copied, the board will automatically reboot.

Next, the user can install the official Raspberry Pi Pico extension into VSCode to interface with the device. Once the extension is installed, it should automatically detect the device and show MicroPython's REPL terminal. Into the terminal we can paste the following code:

```python
from machine import Pin, I2C
i2c = I2C(0, scl=Pin(1), sda=Pin(0))
print(i2c.scan())
```

If we get the following values in return, then the sensors are properly wired up and being detected:

```
[88, 118]
```

Writing my Code and Using Libraries

This project (sans the display) requires three Python files:

- main.py: This interfaces with the sensor libraries, passes values between sensors, provides ratings for different metrics, and displays the output to the REPL terminal of a connected PC.

- bme280.py: This is the sensor library I wrote. It manages initialization, and writing to configuration and data registers, reading raw sensor values, and calculating the compensated values before exposing them back to the main class.
- adafruit_sgp30.py: This is an open source MicroPython library adapted from Adafruit's CircuitPython library. It also uses the same MIT license as my code does (all of which is properly attributed on my repository).
- epaper2in13.py: This *would have been* used if my e-ink display had not ripped. It would have allowed me to manage displaying sensor data on the display. I will try using it again once I get a replacement display in the future. It is included in my repo for future use, which also uses the MIT license (with attribution).

Writing the bme280.py MicroPython Library

Writing this library involved a *lot* of research, reading the official BME280 data sheet, and mentoring from ChatGPT, as I had never done something like this before. This meant understanding everything from all the different registers and how to manage them, to realizing that there was a *lot* of math that I will not even *pretend* to understand, it just needs to exist and work.

The first step was to wrap my head around all the different registers and what they did. This meant a lot of reading the data sheet and taking a lot of notes in code comments about the various registers and how to properly read or write to them. For example, I needed to configure certain settings to ensure the sensor behaved as I needed it to, and some registers had multiple values that I needed to manipulate with bitwise operations to ensure I was reading or writing to them in particular ways.

Once I had wrapped my head around registers, I then needed to wrap my head around all the different values that I was going to be working with. There were a lot of different settings that I could use, so I needed to pick rational settings appropriate to the project, while also making sure that my library would manage modes I was not planning to use, as I'm hoping others might use this library.

From here, I needed to implement the functions that would use the device's internal calibration data to convert raw sensor data into more accurate values in the appropriate units. The math here is beyond my experience, but fortunately, the data sheet provides all the formulas needed to implement them, so all I needed to do was translate those from C to MicroPython.

Finally, I simply needed to provide functions to effectively manage reading data registers (properly respecting whatever mode the sensor was set into), parse them through the calibration function, and return the properly formatted values back to the caller. This ended up being both a *lot* more work than I expected, but also a *lot* more fun than I thought it would be.

Before writing out my project's main.py, I put together a simple test harness with ChatGPT's help and confirmed that my library was functioning as expected. There is something really satisfying about writing a complex library and seeing it *just working* on the first run. Seeing that it was working, I then moved onto writing a proper main.py file.

Writing the main.py Application

The main.py file ties everything together: it imports the various libraries, initializes the I2C bus and both sensors, reads and passes sensor values, manages baseline persistence, and exposes helper functions to REPL so the user can interact with the device. I even went as far as to add an air quality rating function and a personalized comfort classification function: the former because most people have no idea what good eCO2 and TVOC values would be, and the latter because I thought it would be cool.

The first batch of functions initialize the I2C bus and both sensors. Initializing the I2C bus involves specifying which of the two buses to use, the pins to use, and the frequency to run at. Then each of the two sensors need to be initialized, which involves pointing to the initialized I2C bus, as well as any configuration options. For now, my code just uses the defaults specified by the library, but I might adjust both in the future.

The next batch manages the SGP30 sensor's baseline persistence. The SGP30 calibrates itself to its current environment, creating baseline values that adjust over time. Without adding persistence, these values would be lost every time the device was powered off. By storing and updating these values on the Raspberry Pi Pico's internal flash memory semi-regularly, I can load these values as needed, maintaining the baseline.

Given that most people are likely to be unfamiliar to what CO2 and TVOC value ranges are considered good or bad, I decided to add additional functions to help classify those values into ratings ranging from "Excellent" to "Poor". These functions simply take sensor values and assign a rating based on ranges recommended by ChatGPT, which given the scope of this project I felt was "good enough". I also added an additional function which looks at the temperature and humidity and classifies it into comfort ratings based on my personal preferences, just because I can. I noticed that I like it warmer when the air is dryer in winter, while I prefer it cooler in the summer humidity, so I based the algorithm around what I set my thermostat in the summer and winter and what the humidity likely is during those months here in Ohio.

This naturally leads into the most essential functions for this project, those which grab the sensor data and display the results to the REPL terminal. The function that reads the sensor values also passes the temperature and humidity values from the BME280 to the SGP30 before grabbing its values, allowing the SGP30 to adjust its readings to the environment for more accurate readings. The function that prints out the readings formats the sensor values into a human-readable format alongside comfort and air quality ratings. There is also a function which simply grabs sensor data continuously every second, calling the print function to display that to the terminal and updating the baseline file as needed.

The final bit of this file is the main function, which simply uses all the above functions to initialize all the devices, load the baseline, and launch the continuous loop function. It also has an optional REPL argument to skip the loop, allowing the user to call it to initialize the device without it immediately launching into continuous readings. This also means other functions can be called from REPL, such as manually saving the baseline file, or taking only a single reading.

Putting It All Together

With the code complete and the device wired together, the last step is simply copying the code over to the Pico and running the code through REPL. Using VSCode with the official Raspberry Pi Pico extension, this is as easy as clicking my project files, copying the entire project source directory over, and then running the main.py file from the VSCode interface. This will launch the code on the Pico, which then displays the continuous sensor data on the REPL terminal within VSCode. The user can also optionally instead call functions within the REPL terminal if they prefer.

Results

Overall, I am incredibly happy with the results of this project. I was pleasantly surprised to find that my code ran very well without any significant bugs once I got most of it fleshed out, with the only real hiccup being the damaged e-ink display preventing me from properly implementing that aspect of the project. I was also fortunate that the third-party library I used for the SGP30 worked as expected and without any issues.

Hardware Assembly

As noted, the hardware used by this project includes a Raspberry Pi Pico 2 W, a BME280 temperature/humidity/pressure sensor board, and an SGP30 eCO2/TVOC sensor board. I was able to solder pins onto the devices so that I could put them on a bread board in the correct orientation. This made it easy to wire everything together. I then wired the 3.3V out and ground from the Pico to the shared bus at the bottom of the board (see Figure 1), allowing the two sensors to share the same power source. I then wired the two I2C connectors on the Pico (SDA, SCL) to the bus at the top of the bread board, allowing the sensors to share the same bus as well. Unfortunately, all I had on hand were longer connector cables to wire everything together; I hope to acquire cables of more appropriate lengths to wire everything together more neatly in the future.

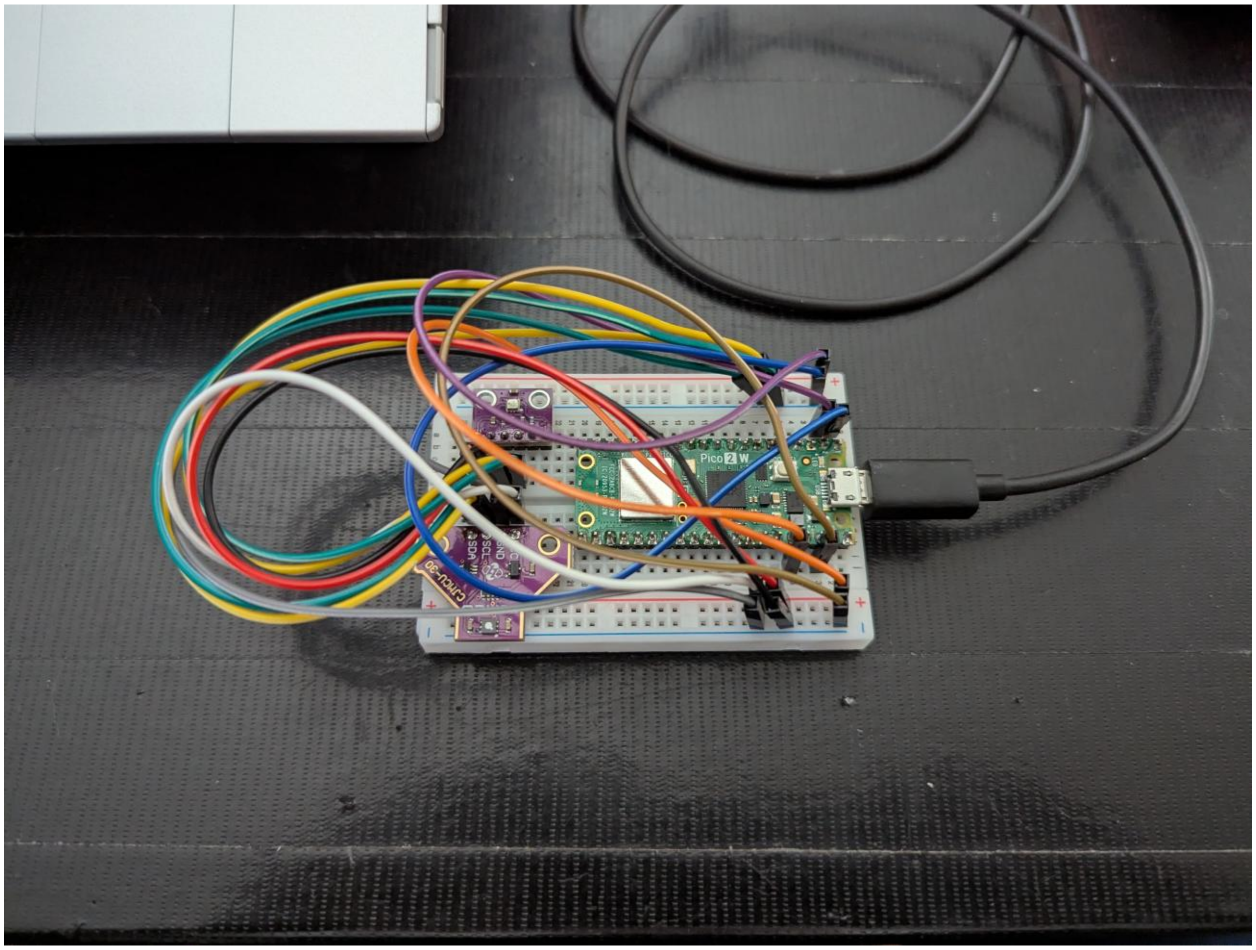
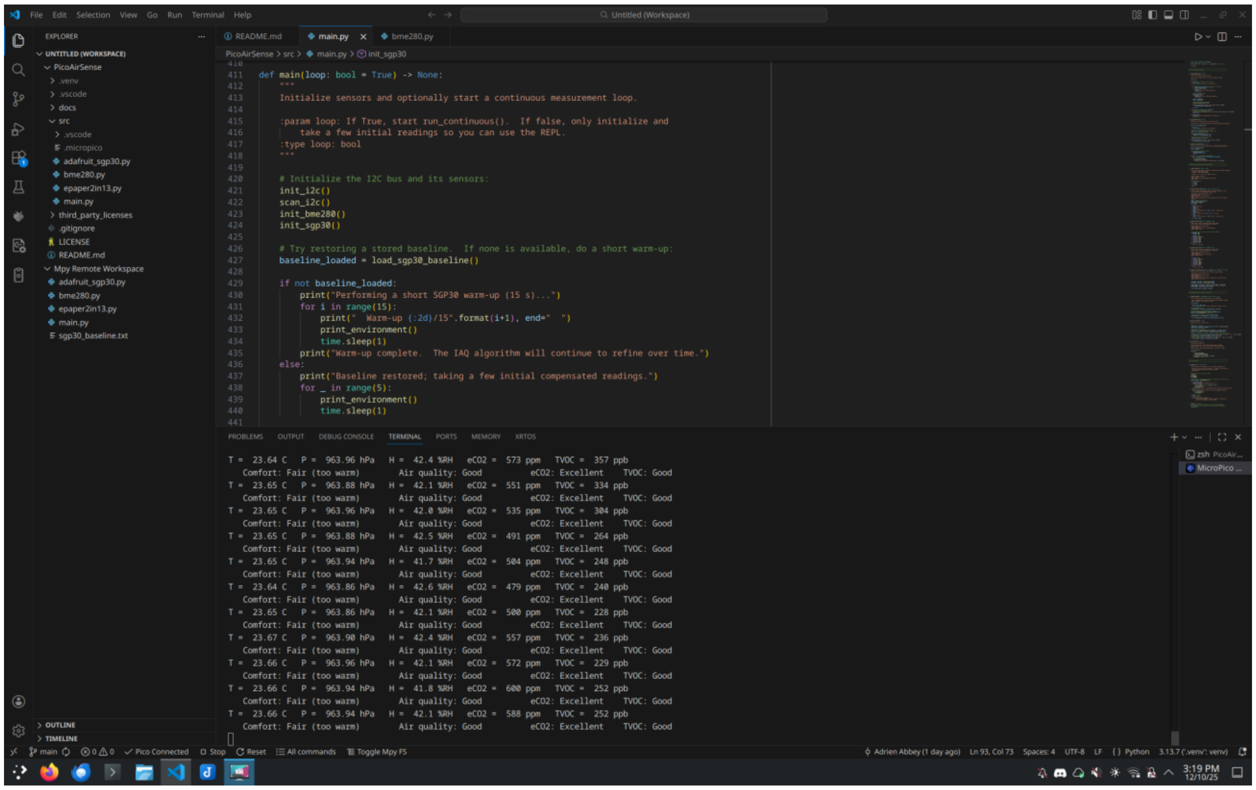


Figure 1: Image of Assembled PicoAirSense

Software Setup

I did all my software development in VSCode, pushing changes to my public GitHub repository. The official Raspberry Pi Pico extension was used to interface with the hardware, allowing me to easily push my Python files to the Pico and access the REPL terminal within VSCode so that I can see the software run. Admittedly, I did not make use of git branches or GitHub issues like I should have, as my primary goal was just to get something working ASAP. This will hopefully be something I rectify should I decide to continue developing this project later.

In Figure 2, you can see my VSCode window open showing the file structure of my project, the file structure on the Raspberry Pi Pico, some of the source code, and the REPL terminal at the bottom displaying sensor readings updating every second.



```python
def main(loop: bool = True) -> None:
    """
    Initialize sensors and optionally start a continuous measurement loop.

    :param loop: If True, start run_continuous().  If false, only initialize and
        take a few initial readings so you can use the REPL.
    :type loop: bool
    """

    # Initialize the I2C bus and its sensors:
    init_i2c()
    scan_i2c()
    init_bme280()
    init_sgp30()

    # Try restoring a stored baseline.  If none is available, do a short warm-up:
    baseline_loaded = load_sgp30_baseline()

    if not baseline_loaded:
        print("Performing a short SGP30 warm-up (15 s)...")
        for i in range(15):
            print("  Warm-up {:2d}/15".format(i+1), end="  ")
            print_environment()
            time.sleep(1)
        print("Warm-up complete.  The IAQ algorithm will continue to refine over time.")
    else:
        print("Baseline restored; taking a few initial compensated readings.")
        for _ in range(5):
            print_environment()
            time.sleep(1)
```

```
T =  23.64 C   P =  963.96 hPa   H =  42.4 %RH   eCO2 =  573 ppm   TVOC =  357 ppb
   Comfort: Fair (too warm)        Air quality: Good          eCO2: Excellent    TVOC: Good
T =  23.65 C   P =  963.88 hPa   H =  42.1 %RH   eCO2 =  551 ppm   TVOC =  334 ppb
   Comfort: Fair (too warm)        Air quality: Good          eCO2: Excellent    TVOC: Good
T =  23.65 C   P =  963.96 hPa   H =  42.0 %RH   eCO2 =  535 ppm   TVOC =  304 ppb
   Comfort: Fair (too warm)        Air quality: Good          eCO2: Excellent    TVOC: Good
T =  23.65 C   P =  963.88 hPa   H =  42.5 %RH   eCO2 =  491 ppm   TVOC =  264 ppb
   Comfort: Fair (too warm)        Air quality: Good          eCO2: Excellent    TVOC: Good
T =  23.65 C   P =  963.94 hPa   H =  41.7 %RH   eCO2 =  504 ppm   TVOC =  248 ppb
   Comfort: Fair (too warm)        Air quality: Good          eCO2: Excellent    TVOC: Good
T =  23.64 C   P =  963.86 hPa   H =  42.6 %RH   eCO2 =  479 ppm   TVOC =  240 ppb
   Comfort: Fair (too warm)        Air quality: Good          eCO2: Excellent    TVOC: Good
T =  23.65 C   P =  963.86 hPa   H =  42.1 %RH   eCO2 =  500 ppm   TVOC =  228 ppb
   Comfort: Fair (too warm)        Air quality: Good          eCO2: Excellent    TVOC: Good
T =  23.67 C   P =  963.90 hPa   H =  42.4 %RH   eCO2 =  557 ppm   TVOC =  236 ppb
   Comfort: Fair (too warm)        Air quality: Good          eCO2: Excellent    TVOC: Good
T =  23.66 C   P =  963.96 hPa   H =  42.1 %RH   eCO2 =  572 ppm   TVOC =  229 ppb
   Comfort: Fair (too warm)        Air quality: Good          eCO2: Excellent    TVOC: Good
T =  23.66 C   P =  963.94 hPa   H =  41.8 %RH   eCO2 =  600 ppm   TVOC =  252 ppb
   Comfort: Fair (too warm)        Air quality: Good          eCO2: Excellent    TVOC: Good
T =  23.66 C   P =  963.94 hPa   H =  42.1 %RH   eCO2 =  588 ppm   TVOC =  252 ppb
   Comfort: Fair (too warm)        Air quality: Good          eCO2: Excellent    TVOC: Good
```

Figure 2: Screenshot of VSCode with REPL terminal.

A more detailed example of the REPL terminal output is provided in the appendix of this document. This output (copied verbatim from a real run) shows the initial REPL terminal connection displaying the version of MicroPython being used, a manual call of the main() function, and the initial output shown when it first launches. This example output also displays dramatic changes in the air quality sensor data when exposed to nearby hands recently washed with alcohol products.

What Worked, What Didn't Work, What Could Be Better

The original project proposal discussed including an e-ink display connected to the Raspberry Pi Pico over the SPI interface to display sensor data, removing the need to interface with a PC to display measurements. Unfortunately, at some point the ribbon cable connecting the display to its controller board became damaged, resulting in a non-functional device. Fortunately, I was able to pivot to using the REPL terminal to display the sensor data, mitigating that issue for now.

I was quite fortunate that besides that one glaring flaw, everything else appears to be working quite well. My BME280 sensor library seems to be robust and dependable, as I have been able to run the device continuously for several hours without any obvious issues. I am also quite happy with how my main.py application turned out, as it feels robust.

One unknown is how accurate the sensor readings are. For example, the temperature readings feel a bit higher than expected, leading me to implement a basic temperature offset to help bring it more in line with expectations. I would need access to properly calibrated sensors if I wanted to determine the accuracy of these readings, which is beyond the scope of my project.

Another unknown is how useful this sensor information might be. One of my main goals was to measure the air quality of my apartment, and while eCO2 and TVOC are metrics for that, they do not encompass the entire picture. For example, the eCO2 level is an *estimate* based on the TVOC measurements and does not accurately reflect true CO2 levels. Other products available online also include other measurements, such as air particulate measurements. These are again outside the scope of this project's current goals but could be expanded upon in the future.

Conclusion

In the end, I am quite happy with how this project turned out. As it is, the PicoAirSense can read the temperature, air pressure, relative humidity, eCO2 and TVOC levels, all while providing a contextual rating of the environment for ease of use. While the lack of an integrated e-ink display is a bit of a loss, that is something I could not address given the time constraints.

Future goals for this project start with finding a replacement e-ink display and getting that implemented. From there, I would like to investigate enabling data collection on an external system over the Pico's Wi-Fi connection, allowing me to start collecting historical data, with the potential to allow external monitoring and alerts. The BME280 library also has some untested potential, as right now it is defaulting to using sensor settings according to a 'weather station' settings profile recommended in the official data sheet that appeared appropriate for this project's goals.

I may also attempt to split off the BME280 library into its own repo with documentation so that others can easily discover and use the library in their own projects. One of my primary goals was for this project to function as a resume builder as well as a contribution to the open-source community and splitting off the library could serve that purpose well.

Finally, I would also like to investigate designing and printing a proper enclosure for the final product, protecting the delicate bits while giving it a professional appearance. This too could be included in the GitHub repo alongside more detailed documentation and instructions, allowing others to implement this project themselves.

Appendix

PicoAirSense GitHub Repository

The project's public GitHub repository is available below, along with a tagged release to function as a snapshot for the final project:

- https://github.com/adrienabbey/PicoAirSense/
- https://github.com/adrienabbey/PicoAirSense/tree/wsu-embedded-final-2025-12-10

NOTE: This repository will get additional changes over time, meaning the tagged release linked above should be the one referenced by this paper and should be the one used for grading purposes.

Example REPL Terminal Output

```
MicroPython v1.26.1 on 2025-09-11; Raspberry Pi Pico 2 W with RP2350
Type "help()" for more information or .help for custom vREPL commands.

>>> main()
I2C devices found: ['0x58', '0x76']
BME280 initialized successfully.
SGP30 serial: ['0x0', '0x1f9', '0x1668']
Restored SGP30 baseline from sgp30_baseline.txt: eCO2=37421 TVOC=39182
Baseline restored; taking a few initial compensated readings.
T =  23.00 C   P =  963.75 hPa   H =  44.6 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  22.98 C   P =  963.80 hPa   H =  44.0 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  22.98 C   P =  963.80 hPa   H =  43.6 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  22.98 C   P =  963.72 hPa   H =  43.4 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  22.98 C   P =  963.75 hPa   H =  43.0 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
```

```
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
Starting a continuous measurement loop.  Press Ctrl+C to stop.
T =  22.98 C   P =  963.78 hPa   H =  42.9 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  22.97 C   P =  963.78 hPa   H =  42.5 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  22.98 C   P =  963.87 hPa   H =  42.3 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  22.98 C   P =  963.78 hPa   H =  42.7 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  22.98 C   P =  963.80 hPa   H =  42.7 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  22.99 C   P =  963.80 hPa   H =  44.4 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  22.99 C   P =  963.75 hPa   H =  44.0 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  23.00 C   P =  963.80 hPa   H =  44.6 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  23.01 C   P =  963.83 hPa   H =  44.1 %RH   eCO2 =  400 ppm   TVOC =    0 ppb
   Comfort: Good (slightly warm)   Air quality: Excellent     eCO2: Excellent
TVOC: Excellent
T =  23.01 C   P =  963.88 hPa   H =  43.9 %RH   eCO2 =  845 ppm   TVOC = 2205 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.01 C   P =  963.87 hPa   H =  43.3 %RH   eCO2 =  773 ppm   TVOC = 2092 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.01 C   P =  963.80 hPa   H =  43.8 %RH   eCO2 =  733 ppm   TVOC = 1824 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.01 C   P =  963.83 hPa   H =  43.9 %RH   eCO2 =  721 ppm   TVOC = 1796 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.02 C   P =  963.83 hPa   H =  43.5 %RH   eCO2 =  779 ppm   TVOC = 1884 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.03 C   P =  963.75 hPa   H =  43.7 %RH   eCO2 =  737 ppm   TVOC = 1790 ppb
```

```
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.02 C   P =  963.75 hPa   H =  43.9 %RH   eCO2 =  743 ppm   TVOC = 1757 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.02 C   P =  963.69 hPa   H =  43.5 %RH   eCO2 =  784 ppm   TVOC = 1780 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.04 C   P =  963.79 hPa   H =  44.0 %RH   eCO2 =  775 ppm   TVOC = 1785 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.03 C   P =  963.69 hPa   H =  43.2 %RH   eCO2 =  752 ppm   TVOC = 1723 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.04 C   P =  963.83 hPa   H =  44.3 %RH   eCO2 =  749 ppm   TVOC = 1646 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.05 C   P =  963.80 hPa   H =  44.5 %RH   eCO2 =  799 ppm   TVOC = 1658 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.05 C   P =  963.83 hPa   H =  44.4 %RH   eCO2 =  758 ppm   TVOC = 1660 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
T =  23.06 C   P =  963.83 hPa   H =  45.6 %RH   eCO2 =  798 ppm   TVOC = 1549 ppb
   Comfort: Good (slightly warm)   Air quality: Poor          eCO2: Good
TVOC: High
```

main.py Source Code

```python
# SPDX-License-Identifier: MIT
# Copyright (c) 2025 Adrien Abbey
#
# PicoAirSense main application
#
# - Initializes I2C, BME280, and SGP30
# - Uses BME280 temperature/humidity to compensate SGP30 readings
# - Persists SGP30 IAQ baseline to internal flash
# - Exposes helper functions for REPL use

from machine import Pin, I2C  # type: ignore
import time
from bme280 import BME280
import adafruit_sgp30

# ---------------------------------------------------------------------------
# Configuration
# ---------------------------------------------------------------------------
```

```
SGP30_BASELINE_FILE = "sgp30_baseline.txt"
# Save the baseline file at most once per hour to limit flash wear:
SGP30_BASELINE_SAVE_INTERVAL = 3600  # seconds

# Globals for REPL access
i2c = None
bme = None
sgp = None
_last_baseline_save = 0.0

# Temperature calibration offset (deg C) for display/comfort.
# Negative value means the sensor reads too hot and we lower it.
TEMP_CAL_OFFSET_C = -2.0    # adjust this after comparing with a reference


# ---------------------------------------------------------------------------
# Initialization Helpers
# ---------------------------------------------------------------------------

def init_i2c(sda_pin: int = 0, scl_pin: int = 1, freq: int = 100_000) -> I2C:
    """
    Initialize and return the I2C bus.
    """
    global i2c  # Allow changing the i2c variable

    # Initialize the I2C bus object:
    i2c = I2C(0, sda=Pin(sda_pin), scl=Pin(scl_pin), freq=freq)

    return i2c


def scan_i2c() -> list[int]:
    """
    Scan the I2C bus and print a short report.
    """
    global i2c

    # Make sure the I2C bus has been initialized:
    if i2c is None:
        init_i2c()

    # Scan the I2C bus for connected devices:
    devices = i2c.scan()  # type: ignore
    print("I2C devices found:", [hex(d) for d in devices])

    # Verify that the expected sensors are detected:
```

```
    if 0x76 not in devices and 0x77 not in devices:
        print("Warning: BME280 address not detected on the bus.")
    if 0x58 not in devices:
        print("Warning: SGP30 address not detected on the bus.")

    return devices


def init_bme280(bus: I2C | None = None) -> BME280:
    """
    Initialize the BME280 and return the sensor object.
    """
    global bme, i2c

    if bus is None:
        if i2c is None:
            init_i2c()
        bus = i2c

    # NOTE: This is currently initializing with my class defaults, 'weather station'.
    bme = BME280(i2c=bus)  # type: ignore

    print("BME280 initialized successfully.")

    return bme


def init_sgp30(bus: I2C | None = None) -> adafruit_sgp30.Adafruit_SGP30:
    global sgp, i2c

    if bus is None:
        if i2c is None:
            init_i2c()
        bus = i2c

    sgp = adafruit_sgp30.Adafruit_SGP30(bus)

    print("SGP30 serial:", [hex(x) for x in sgp.serial])  # type: ignore

    return sgp


# ---------------------------------------------------------------------------
# Baseline persistence helpers
# ---------------------------------------------------------------------------
```

```
def load_sgp30_baseline() -> bool:
    """
    Try to restore SGP30 baselines from flash.

    Returns True on success, False if the file is missing or invalid.
    """
    global sgp

    # If the SGP is not initialize, throw an error.
    if sgp is None:
        raise RuntimeError("SGP30 is not initialized.")

    try:
        # Attempt to open the baseline file and read its contents:
        with open(SGP30_BASELINE_FILE, "r") as f:
            line = f.read().strip()
        if not line:
            print("Baseline file is empty; ignoring.")
            return False

        # Parse the baseline file:
        parts = line.split(",")
        if len(parts) != 2:
            print("Baseline file is malformed; ignoring.")
            return False

        co2eq = int(parts[0])
        tvoc = int(parts[1])

        # Apply the baseline to the SGP30:
        sgp.set_iaq_baseline(co2eq, tvoc)

        print("Restored SGP30 baseline from {}: eCO2={} TVOC={}".format(
            SGP30_BASELINE_FILE, co2eq, tvoc))

        return True

    except (OSError, RuntimeError) as error_code:
        # No file yet, or filesystem issue:
        print("No usable SGP30 baseline found; starting with factory baseline.",
error_code)
        return False

    except ValueError:
        print("Could not parse SGP30 baseline file; ignoring.")
        return False
```

```
def save_sgp30_baseline() -> None:
    """
    Read the current SGP30 baselines and store them to flash.

    This function can be called from the REPL once the IAQ algorithm has stabilized.
    """
    global sgp, _last_baseline_save

    if sgp is None:
        raise RuntimeError("SGP30 is not initialized.")

    # Load the current baselines from the SGP30:
    co2eq, tvoc = sgp.get_iaq_baseline()  # type: ignore

    # Write the new baselines to flash:
    with open(SGP30_BASELINE_FILE, "w") as f:
        f.write("{},{}\n".format(co2eq, tvoc))

    # Update the last baseline file save time:
    _last_baseline_save = time.time()

    print("Saved SGP30 baseline to {}: eCO2={} TVOC={}".format(
        SGP30_BASELINE_FILE, co2eq, tvoc))


def maybe_save_sgp30_baseline() -> None:
    """
    Periodically save the SGP30 baseline based on SGP30_BASELINE_SAVE_INTERVAL.
    """
    global _last_baseline_save
    now = time.time()

    # Initialize the timestamp if needed:
    if _last_baseline_save == 0:
        _last_baseline_save = now
        return

    # Check if it's time to update the baseline:
    if now - _last_baseline_save >= SGP30_BASELINE_SAVE_INTERVAL:
        try:
            save_sgp30_baseline()
        except OSError as error_code:
            print("Warning: could not save SGP30 baseline:", error_code)


# ----------------------------------------------------------------------------
# Comfort and air quality classification helpers
# ----------------------------------------------------------------------------


def _comfort_temp_f(rh: float) -> float:
    """
    This is my personal comfort line (deg F) calibrated from my personal preferences:
    - ~73 deg F at ~30% RH (dry winter)
    - ~69 deg F at ~60% RH (humid summer)

    T_set(deg F) ~= 77 - 0.133 * RH, clamped to [68, 74]

    :param rh: Relative Humidity %
    :type rh: float
    :return: Returns an adjusted temperature preference
    :rtype: float
    """
    t = 77.0 - 0.133 * rh
    if t < 68.0:
        t = 68.0
    if t > 74.0:
        t = 74.0
    return t


def classify_thermal_comfort(temp_c: float, rh: float) -> str:
    """
    Classify thermal comfort based on how far the actual temperature in deg F
    is from my personal humidity-dependent comfort line.

    :param temp_c: Current temperature in deg C
    :type temp_c: float
    :param rh: Current relative humidity %
    :type rh: float
    :return: A string describing how I likely perceive the current environment.
    :rtype: str
    """
    temp_f = temp_c * 9.0 / 5.0 + 32.0
    target_f = _comfort_temp_f(rh)
    diff = temp_f - target_f
    d = abs(diff)

    if d <= 1.0:
        label = "Excellent"
        nuance = ""
    elif d <= 3.0:
        label = "Good"
        nuance = " (slightly warm)" if diff > 0 else " (slightly cool)"
    elif d <= 5.0:
        label = "Fair"
        nuance = " (too warm)" if diff > 0 else " (too cool)"
    else:
        label = "Poor"
        nuance = " (much too warm)" if diff > 0 else " (much too cool)"

    return label + nuance


def classify_eco2(eco2_ppm: int) -> tuple[int, str]:
    """
    Classify eCO2 in ppm into a qualitative air quality band.
    Returns (index, label) where higher index = worse air.

    :param eco2_ppm: Current eCO2 in ppm
    :type eco2_ppm: int
    :return: Index value (0 to 4), string description
    :rtype: tuple[int, str]
    """

    # Clamp unrealistic low values to nominal outdoor baseline:
    if eco2_ppm < 400:
        eco2_ppm = 400

    if eco2_ppm < 700:
        return 0, "Excellent"
    elif eco2_ppm < 1000:
        return 1, "Good"
    elif eco2_ppm < 1400:
        return 2, "Fair"
    elif eco2_ppm < 2000:
        return 3, "Poor"
    else:
        return 4, "Very Poor"


def classify_tvoc(tvoc_ppb: int) -> tuple[int, str]:
    """
    Classify TVOC in ppb into a qualitative air quality band.
    Returns (index, label) where higher index = worse air.

    :param tvoc_ppb: Current TVOC in ppb
    :type tvoc_ppb: int
    :return: Index value (0 to 4), description
    :rtype: tuple[int, str]
    """

    if tvoc_ppb < 150:
        return 0, "Excellent"
    elif tvoc_ppb < 500:
        return 1, "Good"
    elif tvoc_ppb < 1000:
        return 2, "Moderate"
    elif tvoc_ppb < 3000:
        return 3, "High"
    else:
        return 4, "Very High"


def classify_air_quality(eco2_ppm: int, tvoc_ppb: int) -> tuple[str, str, str]:
    """
    Combine eCO2 and TVOC classes into an overall air quality rating.

    :param eco2_ppm: Current eCO2 in ppm
    :type eco2_ppm: int
    :param tvoc_ppb: Current TVOC in ppb
    :type tvoc_ppb: int
    :return: Overall quality description, eCO2 description, TVOC description
    :rtype: tuple[str, str, str]
    """

    eco2_index, eco2_label = classify_eco2(eco2_ppm)
    tvoc_index, tvoc_label = classify_tvoc(tvoc_ppb)

    overall_index = eco2_index if eco2_index >= tvoc_index else tvoc_index
    overall_labels = ("Excellent", "Good", "Moderate", "Poor", "Very Poor")
    overall_label = overall_labels[overall_index]

    return overall_label, eco2_label, tvoc_label


# ---------------------------------------------------------------------------
# Measurement helpers (for REPL and main loop)
# ---------------------------------------------------------------------------


def read_environment() -> tuple[float, float, float, int, int]:
    """
    Take a single measurement from both sensors, with compensation.

    :return: (temperature_C, pressure_Pa, humidity_percent, eCO2_ppm, tvoc_ppb)
    :rtype: tuple[float, float, float, int, int]
    """
    global bme, sgp

    if bme is None or sgp is None:
        raise RuntimeError("Sensors are not initialized; call main() first.")

    # Read the BME280 first:
    raw_temperature_c, pressure_pa, humidity_percent = bme.read()

    # Use the temp/RH to compensate the SGP30 readings:
    sgp.set_iaq_rel_humidity(humidity_percent, raw_temperature_c)
    eco2, tvoc = sgp.iaq_measure()  # type: ignore

    # Apply calibration offset for display/comfort purposes:
    cal_temperature_c = raw_temperature_c + TEMP_CAL_OFFSET_C

    return cal_temperature_c, pressure_pa, humidity_percent, eco2, tvoc


def print_environment() -> None:
    """
    Read once and print a formatted line.
    """

    # Grab the sensor values:
    temperature_c, pressure_pa, humidity_percent, eco2, tvoc = read_environment()
    pressure_hpa = pressure_pa / 100.0

    # Classify current conditions:
    thermal_label = classify_thermal_comfort(temperature_c, humidity_percent)
    overall_air, eco2_label, tvoc_label = classify_air_quality(eco2, tvoc)

    # Print the formatted sensor values.  NOTE: This will go to the REPL terminal:
    print("T = {:6.2f} C   P = {:7.2f} hPa   H = {:5.1f} %RH   eCO2 = {:4d} ppm
TVOC = {:4d} ppb".format(
        temperature_c, pressure_hpa, humidity_percent, eco2, tvoc))
    # Print the human-friendly comfort and air quality summary:
    print("   Comfort: {:<20}   Air quality: {:<11}   eCO2: {:<11}  TVOC:
{:<11}".format(
        thermal_label, overall_air, eco2_label, tvoc_label))


def run_continuous() -> None:
    """
    Continuous measurement loop.

    Prints readings every 1 second and periodically saves the SGP30
```

```python
    IAQ baseline to flash.  Press Ctrl+C to stop and return to the REPL.
    """
    print("Starting a continuous measurement loop.  Press Ctrl+C to stop.")

    # Loop continuously:
    while True:
        try:
            print_environment()
            maybe_save_sgp30_baseline()
        except Exception as error_code:
            print("Sensor read failed:", error_code)
        time.sleep(1)  # Sleep 1 second.


# ---------------------------------------------------------------------------
# Main entry point
# ---------------------------------------------------------------------------


def main(loop: bool = True) -> None:
    """
    Initialize sensors and optionally start a continuous measurement loop.

    :param loop: If True, start run_continuous().  If false, only initialize and
        take a few initial readings so you can use the REPL.
    :type loop: bool
    """

    # Initialize the I2C bus and its sensors:
    init_i2c()
    scan_i2c()
    init_bme280()
    init_sgp30()

    # Try restoring a stored baseline.  If none is available, do a short warm-up:
    baseline_loaded = load_sgp30_baseline()

    if not baseline_loaded:
        print("Performing a short SGP30 warm-up (15 s)...")
        for i in range(15):
            print("  Warm-up {:2d}/15".format(i+1), end="  ")
            print_environment()
            time.sleep(1)
        print("Warm-up complete.  The IAQ algorithm will continue to refine over
time.")
    else:
        print("Baseline restored; taking a few initial compensated readings.")
        for _ in range(5):
            print_environment()
            time.sleep(1)

    if loop:
        run_continuous()
    else:
        print("Initialization complete.\n"
              "Use print_environment() or read_environment() from the REPL,\n"
              "or call run_continuous() to start running."
              )


if __name__ == "__main__":
    # On power-up / reset, initialize and start continuous measurements.
    # From the REPL, you can instead do: import main; main.main(loop=False)
    main(loop=True)
```

bme280.py Source Code

```
# SPDX-License-Identifier: MIT
# Copyright (c) 2025 Adrien Abbey
#
# This driver implements the BME280 environmental sensor for MicroPython.
# The compensation algorithms follow the formulas described in Bosch
# Sensortec’s BME280 datasheet (BST-BME280-DS002), reimplemented
# independently in Python. No Bosch source code is included.
#
# The official Bosch BME280 data sheet was heavily referenced for this library:
#   https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-
bme280-ds002.pdf
# Compensation formulas are based on the algorithms described in the above data
sheet.
#   Implementation written independently by Adrien Abbey. Original Bosch source code
is NOT
#   included.
#
# NOTE: Much of this is based on guidance from ChatGPT.  No code was copy/pasted, and
all work
#   is done while being mindful of staying within academic integrity standards.
#
# Assumes the use of the I2C interface
# 1. Reads the calibration registers
# 2. Configures the device
#   - NOTE: This has 'weather station' defaults.  The user can specify these.
# 3. Reads the raw sensor data
# 4. Applies formulas as supplied by the data sheet
# 5. Returns human-readable temperature, humidity and pressure values.
#
#
# Control Registers:
#   ID          0xD0    Chip ID             Should return 0x60 for BME280
#   RESET       0xE0    Soft Reset          Write 0xB6 to initiate a full reset
#   CTRL_HUM    0xF2    Humidity Control    Controls oversampling for humidity.
#                                           Must be written before CTRL_MEAS to take
effect.
#                                           '001' for oversampling x1
#   STATUS      0xF3    Status              Bit 3 (measuring) is '1' when conversion
is running.
#                                           Bit 0 (im_update) is '1'f when NVM memory
is copying.
#   CTRL_MEAS   0xF4    Measurement Control Controls pressure/temp oversampling and
power mode.
#                                           '00100101' for temp x1, press x1, forced
mode
#   CONFIG      0xF5    Configuration       Controls standby time (in Normal mode)
and IIR filter
#                                           settings.  '11100000' for filter off
#
# Weather Monitoring Recommendations:
#   Sensor mode: forced mode, 1 sample / minute
#   Oversampling settings: pressure x1, temperature x1, humidity x1
#   IIR filter settings: filter off
#   NOTE: This is quite fine for my intended purposes.
#
# Compensation formulas:
#   This information can be found starting on page 25 of the official Bosch BME280
data sheet.
#   Trimming parameter registers for calibration are available on page 24.
#
# Bitwise functionality:
#   (e4 << 4) : This will do a bit-shift of e4 4-bits left.
#   (e5 & 0x0F) : Bitwise AND.  This will set the first four bits of e5 to zero.
0x0F = '0000 1111'
#   (e4 << 4) | (e5 & 0x0F) : This will bitwise OR the above two into a single 12-bit
value.
#   if value & 0x800 : This will do a bitwise AND between the given value and the hex
value 0x800
#       This translates to '1000 0000 0000' in binary.  In other words, if the 12th
bit is 1,
#       this will return True (non-zero value).
#
```

```
# Create a Python venv and install Raspberry Pi Pico 2 MicroPython stubs to validate
code better:
#   pip install -U micropython-rp2-rpi_pico2-stubs
#
# TODO: Consider adjusting to allow the user to 'reset' and reconfigure to switch
between different
#   sensor modes, etc.


from machine import I2C  # type: ignore
import time


class BME280:

    # Define register addresses:
    CTRL_HUM = 0xF2
    STATUS = 0xF3
    CTRL_MEAS = 0xF4
    CONFIG = 0xF5
    DATA = 0xF7

    # Mode bit patterns:
    MODE_SLEEP = 0b00
    MODE_FORCED = 0b01
    MODE_NORMAL = 0b11

    def __init__(self, i2c: I2C, address: int = 0x76, spi3w_en: int = 0,
                 osrs_t: int = 0b001, osrs_p: int = 0b001, osrs_h: int = 0b001,
                 filter_coef: int = 0b000, t_sb: int = 0b000, mode: int = 0b00) ->
None:
        self.i2c = i2c
        self.address = address          # Note: this is 0x76 by default for the
BME280 sensor

        # Normalize configuration fields to valid ranges.
        # If enabled, use 3-wire SPI, otherwise use I2C:
        self.spi3w_en = 1 if spi3w_en else 0    # Only 1 or 0.
        # Temperature oversampling value (0, x1, x2, x4, x8, x16):
        self.osrs_t = osrs_t & 0x07   # 3 bits
        # Pressure oversampling value (0, x1, x2, x4, x8, x16):
        self.osrs_p = osrs_p & 0x07
        # Humidity oversampling value (0, x1, x2, x4, x8, x16):
        self.osrs_h = osrs_h & 0x07
        # IIR filter coefficient (0, 2, 4, 8, 16):
        self.filter_coef = filter_coef & 0x07
        # Standby time (for normal mode):
```

```
        self.t_sb = t_sb & 0x07

        # There's three possible modes, with four possible bit values:
        if mode == self.MODE_SLEEP:
            self._mode = self.MODE_SLEEP
        elif mode == self.MODE_NORMAL:
            self._mode = self.MODE_NORMAL
        else:   # Assume that any other inputs are forced mode:
            self._mode = self.MODE_FORCED

        # Set the _t_fine value (set by temperature readings, used by
pressure/humidity):
        self._t_fine = 0

        # Verify the chip ID
        chip_id = self._read_u8(0xD0)
        if chip_id not in (0x60,):
            raise RuntimeError(
                "The BME280 was not found, or wrong chip ID was given:
0x{:02X}".format(chip_id))

        # Read calibration data:
        self._read_calibration_data()

        # Configure oversampling / mode registers
        self._configure()

    def _read_u8(self, reg):
        """
        Reads one unsigned 8-bit value from the specified device register.

        :param reg: The register address on the device to read from.
        """
        return int.from_bytes(self.i2c.readfrom_mem(self.address, reg, 1), "big")

    def _read_calibration_data(self) -> None:
        # TODO: implement according to BME280 data sheet
        # Read blocks of registers into attributes like self.dig_T1, dig_T2, ...

        # From the official BME280 data sheet:
        #   0x88 / 0x89         dig_T1 [7:0] / [15:8]   unsigned short
        #   0x8A / 0x8B         dig_T2 [7:0] / [15:8]   signed short
        #   0x8C / 0x8D         dig_T3 [7:0] / [15:8]   signed short
        #   0x8E / 0x8F         dig_P1 [7:0] / [15:8]   unsigned short
        #   0x90 / 0x91         dig_P2 [7:0] / [15:8]   signed short
        #   0x92 / 0x93         dig_P3 [7:0] / [15:8]   signed short
        #   0x94 / 0x95         dig_P4 [7:0] / [15:8]   signed short
        #   0x96 / 0x97         dig_P5 [7:0] / [15:8]   signed short
        #   0x98 / 0x99         dig_P6 [7:0] / [15:8]   signed short
        #   0x9A / 0x9B         dig_P7 [7:0] / [15:8]   signed short
        #   0x9C / 0x9D         dig_P8 [7:0] / [15:8]   signed short
        #   0x9E / 0x9F         dig_P9 [7:0] / [15:8]   signed short
        #   0xA1                dig_H1 [7:0]            unsigned char
        #   0xE1 / 0xE2         dig_H2 [7:0] / [15:8]   signed short
        #   0xE3                dig_H3 [7:0]            unsigned char
        #   0xE4 / 0xE5[3:0]    dig_H4 [11:4] / [3:0]   signed short
        #   0xE5[7:4] / 0xE6    dig_H5 [3:0] / [11:4]   signed short
        #   0xE7                dig_H6                  signed char

        # Read the temperature and pressure calibration values in a large block:
        tp_buf = self.i2c.readfrom_mem(self.address, 0x88, 26)
        # tp_buf[0] = 0x88, tp_buf[1] = 0x89, ... tp_buf[23] = 0x9F

        # Temperature values:
        self.dig_T1 = self._u16_le(tp_buf[0], tp_buf[1])    # unsigned  short
        self.dig_T2 = self._s16_le(tp_buf[2], tp_buf[3])    # signed    short
        self.dig_T3 = self._s16_le(tp_buf[4], tp_buf[5])    # signed    short

        # Pressure values:
        self.dig_P1 = self._u16_le(tp_buf[6], tp_buf[7])    # unsigned  short
        self.dig_P2 = self._s16_le(tp_buf[8], tp_buf[9])    # signed    short
        self.dig_P3 = self._s16_le(tp_buf[10], tp_buf[11])  # signed    short
        self.dig_P4 = self._s16_le(tp_buf[12], tp_buf[13])  # signed    short
        self.dig_P5 = self._s16_le(tp_buf[14], tp_buf[15])  # signed    short
        self.dig_P6 = self._s16_le(tp_buf[16], tp_buf[17])  # signed    short
        self.dig_P7 = self._s16_le(tp_buf[18], tp_buf[19])  # signed    short
        self.dig_P8 = self._s16_le(tp_buf[20], tp_buf[21])  # signed    short
        self.dig_P9 = self._s16_le(tp_buf[22], tp_buf[23])  # signed    short

        # Humidity values (incomplete):
        self.dig_H1 = tp_buf[25]                            # unsigned  char

        # Humidity calibration gets complicated, as some values are packed.
        # Read the humidity calibration values in a large block:
        h_buf = self.i2c.readfrom_mem(self.address, 0xE1, 7)
        # buf2[0] = 0xE1, ..., buf2[6] = 0xE7

        # Humidity values (easy):
        self.dig_H2 = self._s16_le(h_buf[0], h_buf[1])      # signed    short
        self.dig_H3 = h_buf[2]                              # unsigned  char

        # The following values require some assembly:
        e4 = h_buf[3]   # 0xE4
        e5 = h_buf[4]   # 0xE5
```

```
        e6 = h_buf[5]   # 0xE6

        # Shift 0xE4 left 4 bits [11:4] with E5[3:0] before combining:
        raw_h4 = (e4 << 4) | (e5 & 0x0F)
        # Sign-extend the 12-bit signed to a Python int:
        if raw_h4 & 0x800:  # If the 12th bit is 1:
            raw_h4 -= 1 << 12
        self.dig_H4 = raw_h4                               # signed    short

        # Shift 0xE6 left 4 bits and 0xE5 left 4 bits before combining:
        raw_h5 = (e6 << 4) | (e5 >> 4)
        # Sign-extend the 12-bit signed into a Python int:
        if raw_h5 & 0x800:  # If the 12th bit is 1:
            raw_h5 -= 1 << 12
        self.dig_H5 = raw_h5                               # signed    short

        # Finally, H6 needs to assembly:
        self.dig_H6 = self._s8(h_buf[6])                   # signed    char

    ###
    # These helper functions are necessary to convert raw register values into usable
integers.
    #   @staticmethod allows for the helper functions to avoid needing to use 'self',
as they don't
    #       modify any instance attributes.
    ###

    @staticmethod
    def _u16_le(low: int, high: int) -> int:
        """
        Combine two bytes (little-endian) into an unsigned 16-bit int.
        """
        return low | (high << 8)  # Bit-shifts the high value and then combines the
two values.

    @staticmethod
    def _s16_le(low: int, high: int) -> int:
        """
        Combine two bytes into a signed 16-bit int (two's complement).
        """
        value = low | (high << 8)
        if value & 0x8000:  # check the sign bit
            value -= 0x10000
        return value

    @staticmethod
    def _s8(b: int) -> int:
        """
        Interpret one byte as a signed 8-bit value.
        """
        return b - 0x100 if b & 0x80 else b  # Sign if needed

    def _configure(self) -> None:
        """
        Configures the device according to provided values.  This includes the sensor
mode,
        oversampling, standby time (normal mode only), IIR filter coefficient, and
I2C / 3-wire SPI
        interfaces.
        """

        # NOTE: Humidity oversampling changes only take effect after writing to
'ctrl_meas'
        # NOTE: Writes to 'config' may be ignored in normal mode, guaranteed in sleep
mode

        # ctrl_hum: 0xF2
        #   This controls oversampling of humidity data.  See below for values.
        # ctrl_meas: 0xF4
        #   This controls oversampling of temperature and pressure data, as well as
sensor mode.
        #       Bit 7, 6, 5:    osrs_t[2:0]
        #       Bit 4, 3, 2:    osrs_p[2:0]
        #       Bit 1, 0:       mode[1:0]
        #   osrs_h[2:0], osrs_p[2:0], osrs_t[2:0]
        #       000             skipped (output set to 0x8000)      No measurement
taken
        #       001             oversampling x 1                    Single
measurement
        #       010             oversampling x 2                    2 measurements
per value
        #       011             oversampling x 4                    4 measurements
per value, etc
        #       100             oversampling x 8                    More
sampling/processing
        #       101, others     oversampling x 16                   Less jitter, more
accurate
        #   mode[1:0]
        #       00              Sleep mode                          Default, no
measurements
        #       01 and 10       Forced mode                         Do one measure
cycle, then sleep
        #       11              Normal mode                         Measure
repeatedly, see config
```

```
        # config: 0xF5
        #   This configures standby time, IIR filter constant, and 3-wire SPI
interface
        #       Bit 7, 6, 5     t_sb[2:0]       Configures delay between normal mode
samples
        #       Bit 4, 3, 2     filter[2:0]
        #       Bit 0           spi3w_en[0]     If enabled [1], disables I2C and
enables 3-wire SPI
        #   t_sb[2:0]       t_standby[ms]
        #       000              0.5
        #       001             62.5
        #       010            125
        #       011            250
        #       100            500
        #       101           1000
        #       110             10
        #       111             20
        #   filter[2:0]     Filter coefficient      The IIR filter is a digital low-
pass filter.
        #       000             Filter off              This reduces short-term noise
and jitter.
        #       001                 2                   Affects temperature and
pressure ONLY.
        #       010                 4                   Higher filter values smooths
outputs more,
        #       011                 8                   based on previous
measurements.
        #       100, others        16

        # Configure the humidity oversampling (ctrl_hum) first:
        ctrl_hum = self.osrs_h & 0x07   # bits 2:0
        self.i2c.writeto_mem(self.address, self.CTRL_HUM, bytes([ctrl_hum]))

        # Configure standby time, IIR filtering, and SPI/I2C next:
        config = ((self.t_sb & 0x07) << 5) | (
            (self.filter_coef & 0x07) << 2) | self.spi3w_en  # See comments above
        self.i2c.writeto_mem(self.address, self.CONFIG, bytes([config]))

        # Finally, configure temp/press oversampling and sensor mode:
        ctrl_meas = (self.osrs_t << 5) | (
            self.osrs_p << 2) | (self._mode)  # See comments above.
        self.i2c.writeto_mem(self.address, self.CTRL_MEAS, bytes([ctrl_meas]))

    def _start_forced_measurement(self) -> None:
        """
        Start a single forced measurement using the configured oversampling values.
        After writing this, the BME280 performs exactly one measurement (temp, press,
humid as
        configured), then automatically returns to sleep mode.
        """

        # In order to trigger a forced measurement, we simply need to write the
appropriate bits
        #   to CTRL_MEAS, which includes the temperature and pressure oversampling
values.
        # TODO: Consider "remembering" this value, as it's used in both _configure()
and here.

        # Finally, configure temp/press oversampling and sensor mode:
        # Slightly modified from _configure():
        ctrl_meas = (self.osrs_t << 5) | (
            self.osrs_p << 2) | (self.MODE_FORCED)
        self.i2c.writeto_mem(self.address, self.CTRL_MEAS, bytes([ctrl_meas]))

    def _wait_measuring_clear(self) -> None:
        """
        Wait until the current measurement (if any) has completed.

        This polls the STATUS register and waits for bit 3 ("measuring") to become 0.
        """

        while True:
            # Read one byte from the STATUS register:
            # The [0] treats the memory read as an array and sets 'status' to the
first byte:
            status = self.i2c.readfrom_mem(self.address, self.STATUS, 1)[0]

            # Check bit 3 (measuring):
            if (status & 0x08) == 0:
                break

            # Sensor still busy, wait before looping:
            time.sleep_ms(1)  # type: ignore

    def read_raw(self) -> tuple[int, int, int]:
        """
        Returns the raw sensor values (adc_T, adc_P, adc_H) according to the
configured mode.
        """

        # Ensure there is fresh data available:
        if self._mode in (self.MODE_SLEEP, self.MODE_FORCED):
            # For sleep or forced modes, trigger a forced measurement now:
```

```
            self._start_forced_measurement()
            self._wait_measuring_clear()
        elif self._mode == self.MODE_NORMAL:
            # In normal mode, the sensor runs regularly.
            self._wait_measuring_clear()
        else:
            # Fallback: treat unknown as forced:
            self._start_forced_measurement()
            self._wait_measuring_clear()

        # Data Registers:
        #   NOTE: It's recommended to perform a burst read of all registers (0xF7 to
0xFE) in one
        #       operation to ensure these values all belong to the same measurement
instance.
        #
        #   Pressure: 0xF7 (MSB), 0xF8 (LSB), 0xF9 (XLSB - bits 7:4)
        #   Temperature: 0xFA (MSB), 0xFB (LSB), 0xFC (XLSB - bits 7:4)
        #   Humidity: 0xFD (MSB), 0xFE (LSB)
        #
        #   NOTE: XLSB: Extended Least-Significant Byte.  Standard registers are 8-
bits, but the
        #       sensors provide 20-bits of precision.  Thus this extends an
additional 4-bits into
        #       the the XLSB registers (bits 7:4).
        #   NOTE: If I'm using x1 oversampling with IIR filtering off, while my data
is effectively
        #       16-bit (the XLSB bits have little value), I still need to treat it as
a 20-bit
        #       value.

        # Burst-read 8 data bytes representing the raw sensor values:
        data = self.i2c.readfrom_mem(self.address, self.DATA, 8)

        # Parse the data:
        press_msb, press_lsb, press_xlsb = data[0], data[1], data[2]
        temp_msb, temp_lsb, temp_xlsb = data[3], data[4], data[5]
        hum_msb, hum_lsb = data[6], data[7]

        # Combine the parsed data:
        adc_P = (press_msb << 12) | (press_lsb << 4) | (press_xlsb >> 4)
        adc_T = (temp_msb << 12) | (temp_lsb << 4) | (temp_xlsb >> 4)
        adc_H = (hum_msb << 8) | hum_lsb

        # Return the final raw values:
        return adc_T, adc_P, adc_H
```

```
    def _compensate_temperature(self, adc_T: int) -> float:
        """
        Convert raw temperature ADC value to degrees Celsius.

        This implements Bosch's BME280_compensate_T_int32() routine:
            - Updates self._t_fine for use by pressure/humidity compensation.
            - Returns temperature in degrees C as a float.
        """

        # Calculate var1 and var2 following the compensation algorithm described in
the Bosch data
        #   sheet.
        var1 = (((adc_T >> 3) - (self.dig_T1 << 1)) * self.dig_T2) >> 11
        var2 = (((((adc_T >> 4) - self.dig_T1) *
                  ((adc_T >> 4) - self.dig_T1)) >> 12) * self.dig_T3) >> 14

        # Set _t_fine for other functions to use (the "fine resolution" internal
temp)
        self._t_fine = var1 + var2

        # Calculate the actual temperature:
        T = (self._t_fine * 5 + 128) >> 8

        # Return as a float in degrees Celsius:
        return T / 100.0

    def _compensate_pressure(self, adc_P: int) -> float:
        """
        Convert raw pressure ADC value to pressure in Pascals

        Implements Bosch's BME280_compensate_P_int32() integer algorithm.
        Requires self._t_fine to have been set by _compensate_temperature()
        for the same measurement.
        """

        # NOTE: The following implementation follows Bosch's documented integer
        # compensation algorithm for pressure, translated into Python. It is based on
        # the algorithm description in the BME280 data sheet, not on Bosch source
code.

        var1 = (self._t_fine >> 1) - 64000
        var2 = (((var1 >> 2) * (var1 >> 2)) >> 11) * self.dig_P6
        var2 = var2 + ((var1 * self.dig_P5) << 1)
        var2 = (var2 >> 2) + (self.dig_P4 << 16)
        var1 = (((self.dig_P3 * (((var1 >> 2) * (var1 >> 2)) >> 13))
                 >> 3) + ((self.dig_P2 * var1) >> 1)) >> 18
        var1 = ((((32768 + var1)) * (self.dig_P1)) >> 15)
        # Avoid divide by zero errors:
        if var1 == 0:
            return 0

        p = ((1048576 - adc_P) - (var2 >> 12)) * 3125

        if p < 0x80000000:
            p = (p << 1) // (var1)
        else:
            p = (p // var1) * 2

        var1 = (self.dig_P9 * ((((p >> 3) * (p >> 3)) >> 13))) >> 12
        var2 = ((p >> 2) * self.dig_P8) >> 13
        p = (p + ((var1 + var2 + self.dig_P7) >> 4))

        return float(p)

    def _compensate_humidity(self, adc_H: int) -> float:
        """
        Convert raw humidity ADC value to %RH.

        Implements Bosch's bme280_compensate_H_int32() integer algorithm.
        Requires self._t_fine to have been set by _compensate_temperature()
        for the same measurement.
        """

        # NOTE: The following implementation follows Bosch's documented integer
        # compensation algorithm for humidity, translated into Python. It is based on
        # the algorithm description in the BME280 data sheet, not on Bosch source
code.

        var = (self._t_fine - 76800)
        var = (((((adc_H << 14) - ((self.dig_H4) << 20) - ((self.dig_H5) * var)) +
(16384)) >> 15) *
              (((((((var * (self.dig_H6)) >> 10) * (((var * (self.dig_H3)) >> 11) +
(32768))) >> 10) + (2097152)) * (self.dig_H2) + 8192) >> 14))
        var = (
            var - (((((var >> 15) * (var >> 15)) >> 7) * (self.dig_H1)) >> 4))

        if var < 0:
            var = 0

        if var > 419430400:
            var = 419430400

        h = (var >> 12)
        humidity = h / 1024.0   # Convert to %RH as a float
        return humidity

    def read(self) -> tuple[float, float, float]:
        """
        Returns (temperature_C, pressure_Pa, humidity_percent)
        """

        # Get the raw ADC (Analog to Digital Conversion) values:
        adc_T, adc_P, adc_H = self.read_raw()

        # Compensate the temperature first, as this will update self._t_fine:
        temperature_C = self._compensate_temperature(adc_T)

        # Compensate pressure and humidity using the updated self._t_fine value:
        pressure_Pa = self._compensate_pressure(adc_P)
        humidity_percent = self._compensate_humidity(adc_H)

        return temperature_C, pressure_Pa, humidity_percent
```