**Developing an All-In-One IoT Wireless Data Logging Device to Aid 3D Printing**

Jacob Diaz

Department Of Computer Science

COMP 4300: Computer Networks

Dr. Sara Rouhani

December 10, 2022

https://github.com/diazjacob/COMP-4300-Project

**Abstract**

3D printers can be very susceptible to failure due to external factors. This report describes these problems and gives an in-depth look at the implementation, experimentation, and code that went into developing the prototype wireless IoT solution. I investigate a novel solution for detecting 3D printer failures using simple and affordable electrical components combined into an all-in-one IoT sensing device. This device logs data, and seamlessly pairs to its partner application on a mobile device to supply live updates as well as it's backlog of locally recorded data. This wireless IoT device can be used to actively monitor live ambient room conditions, or to leave running independently so that readings can be requested by the user later time.

**3D printing**

I own and use multiple 3D printers regularly for projects, leisure, and repairs. There exist two primary types of hobby-grade 3D printer:

* Fused Deposition Modelling (FDM): A method of print that requires a spool of PLA or ABS material. This spool is extruded through a heated nozzle onto a heated plate to create an object layer-by-layer. This is the most common type of consumer 3D printer and is very easily identified by it’s 3-axis CNC-like carriage system required to build objects.
* Stereolithography (SLA): A method of printing that utilizes liquid UV resin that cures under a wavelength of 405nm. This resin is poured into a clear-bottomed vat and a brushed metal build plate is lowered into the liquid. Underneath the vat is a high-powered UV LED matrix that selectively cures the resin onto the build plate via a transparent monochromatic LCD display. SLA printers print an entire layer at a time and have a single vertical movement axis that pulls the model vertically out of the vat.

**The Challenge**

3D printers fail. Printing failures exist on a spectrum between minor surface imperfections to large messes of unsalvageable printing materials that wastes time and money. Minimizing these failures is important and a part of minimising these failures is removing external factors. FDM prints will warp and bend if the ambient room temperature fluctuates, and the printing material becomes tougher to use in higher humidity environments. For example, if someone turns on a fan on the other side of a room with a 3D printer at work it can very easily cause the model to severely warp mid-print. SLA printers do not suffer from these same issues, but the UV resin is very sensitive to UV light. Any source of UV will actively degrade the resin as I handle it, and it would be very helpful to measure UV to understand possible sources near my setup. All types of 3D printer can also fail due to incorrect configuration, therefore our primary goal with this project was to be able to easily distinguish printing failures due to ambient conditions and failures due to configuration to solve them much faster.

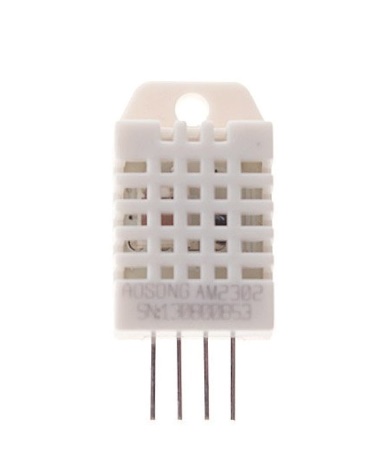
**The Hardware**

The goal was to create a wireless all-in-one IoT device that can log data locally and to another device on the network. When used alongside a mobile app, all aspects of networking should be transparent to the user to give the best user experience. This means that wireless communication must be automatic, consistent, and fault tolerant. 4 datapoints are needed: UV index, temperature, humidity, and time. Each datapoint requires its own hardware to add onto the microcontroller. This is all the hardware used:

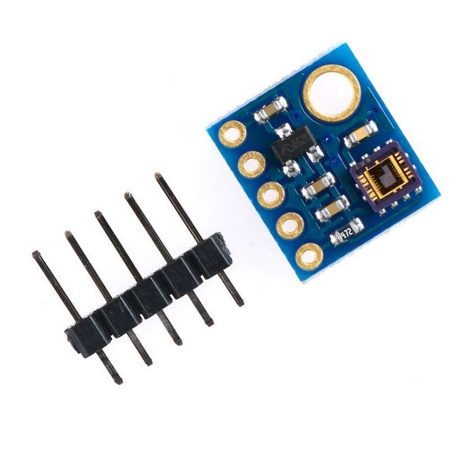
* *Raspberry Pi Pico W:* 
  + **A circuit board with many chips

    Description automatically generated with low confidence**I chose a microcontroller instead of a full computer for its small size, small power draw, and relative simplicity. Raspberry Pi computers are also still under a heavy chip shortage, and so this made the Pi Pico W microcontroller an obvious choice.
  + The Raspberry Pi Pico has a Dual Core ARM M0+ processor running at 133MHz, 40 General Purpose IO pins (ADC, SPI, I2C, UART), 2MB of flash memory, and an integrated single-band 2.4GHz wireless interface (802.11n)

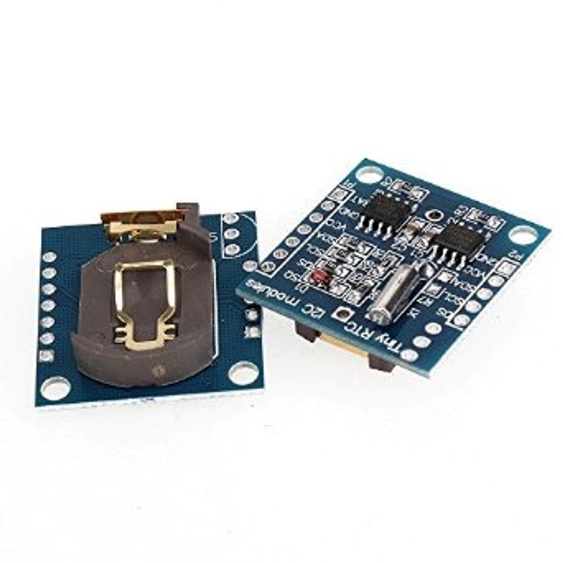
Source: www.canakit.com

* *DHT22 Sensor:*

Source: PiShop.ca

* + The DHT22 is a very common and cheap temperature sensor among hobby electronics. It is accurate down to 0.1C and 0.1%RH.
  + The DHT22 uses a single-bus digital data transmission method requiring only a VCC, GND, and DAT connection.
* *GYML8511 Sensor:*
  + We are targeting a sensing range around 405nm. 405nm is at the edge of human vision, and some would simply call it a deep purple. It is not easy to find UV sensors to measure this 405nm value due to a couple reasons:
    - The majority of “UV Sensors” on the market do not measure UV. They measure visible light under the assumption that it is sunlight and approximate the UV index based on known ratios.
    - 405nm falls directly between ultraviolet and visible light with ultraviolet light officially beginning below the 400nm point, so hobby-grade sensors containing this overlap are not as prevalent.
  + I selected the GYML8511 sensor as I believe it is the best fit. This sensor directly measures UV light in the 290-395nm range, very close to our targeted value. Moreover, because it measures UV directly it can better capture the true UV index of light that has passed through different mediums like glass and plastics.

Source: PiShop.ca

* *DS1307 RTC:*
  + The DS1307 is a quartz oscillator chip with a built in I2C connection and 2023 coin cell battery. This is needed because the Pico W does not actually contain any timekeeping hardware. The only alternative to this is a HTTP request to a server to fetch the time, but HTTP requests can be heavy on the RP2040 chip and a RTC hardware implementation is by far the most accurate and reliable solution.

Source: PiShop.ca

* *Pico LiPo Shim & Battery:*
  + A close-up of a computer chip

    Description automatically generated with low confidenceA shim module slides onto the existing Pico W header structure to add functionality. In this case we have a shim that adds a voltage regulator, LiPo discharge protector, and LiPo 215mah charger. This allows my wireless IoT device to stay wireless and packed into a small self-sufficient device enclosure.

Source: PiShop.ca

* + An *850mah single-cell LiPo battery* was selected. The Pico W datasheet does not list current draw statistics, but the Raspberry Pi community forms cite the Pico W function with networking active at 50-60mah. This gives an ideal 850/55 = 15.5 hours of continuous logging, which would most likely fully populate the Pico’s memory long before it drains the battery.

**The Software:**

The Raspberry Pi Pico W can run both C and Python, I chose Python. I flashed my Pico W’s memory with the newest version of MicroPython (V1.19.1 nightly) that supports the (W)ireless variant of the Pico. MicroPython is a heavily simplified version of Python3 designed to run on microcontrollers. Even though it is simplified, it can still perform advanced tasks like threading, machine learning, networking, and more. The Pico W implements a WLAN functionality that was very simple to implement on my home network. Upon connection the Pico W gives its configuration in the form of a tuple:

('**10.0.0.183**', '**255.255.255.0**', '**10.0.0.1**', '**64.59.176.13**')

From left to right, this is our **local IP**, **subnet mask, router IP** (i.e. our “first hop”), and our router-supplied **DNS** address. In this case, I am a Shaw customer and we can verify 64.59.176.13 to be the default Shaw DNS service IP. My router has a DHCP lease time of 2 days so it is safe to say that these credentials will be consistent. I also used the **dht.py** and **ds1307.py** libraries for sensing. The former is included in MicroPython (<https://micropython.org/> ), the latter was found here: <https://github.com/mcauser/micropython-tinyrtc-i2c>.

For the application I chose Unity 2021.1 because I am very familiar with the Unity prototyping process. Unity does incur some processing overhead but overall; it is still very efficient for this task because my networking implementation will be in pure C#. This is a list of the libraries/tools used:

* **JSON.NET:** Used to serialize json to/from C# classes.
  + *By Newtonsoft*: [*https://www.newtonsoft.com/json*](https://www.newtonsoft.com/json)
* **Shapes:** A procedural vector graphic library to improve the look of my app.
  + *By Freya Holmér***:** [*https://acegikmo.com/shapes/*](https://acegikmo.com/shapes/)
* **C#’s System & .NET Network Framework.**
  + *By Microsoft:* [*https://learn.microsoft.com/en-us/dotnet/csharp/*](https://learn.microsoft.com/en-us/dotnet/csharp/)

All Unity assets were used solely to improve the look and feel of my application, and they in no way aided in the implementation of my application’s core networking functionality. The networking implementation is my own.

**The Implementation Process**

There are many ways to approach a networking solution like this. We must construct a networking protocol for transferring readings, determine the transport-layer protocol(s), and make the entire process transparent to the user. One of the most common solutions to IoT sensing is hosting a webserver. The popular 3D printer wireless control software *OctoPi* uses this method, albeit on much more powerful Raspberry Pi hardware. This is a usable solution, and the Pico W can handle lightweight webserver requests, but I feel as though it is very restricting. A webserver places the data handling and presentation into scripts sent to the client, and in turn with the rising project complexity forces the Pico W to fulfill larger and larger HTTP requests. The average simplistic HTTP webpage requires magnitudes more data to be transmitted compared to the measurements themselves. To prioritize the measurements, we need to implement a lightweight proprietary networking solution onto the Pico to offload all presentation overhead onto another device. My networking solution interfaces with any mobile phone to present the user with a seamless two-way connection that receives data and send out sensor control requests.

I decided on creating a simple proprietary TCP-based JSON networking protocol for data transmission, and a UDP broadcasting protocol used for network discovery. When both the user’s mobile device and the Pico are on the same network the mobile device will periodically broadcast a small JSON-encoded UDP datagram on port 51519 like so:

{"ID":"PicoCast","iter":12,"ip":"10.0.0.170","port":51520}

This datagram contains an ID, a counter, and the address the data logging server is currently being hosted on within the mobile device. This is implemented with the C# *UdpClient*class with a specified IP endpoint of 255.255.255.255, a special address that signifies to the router to broadcast the UDP message to the entire subnet. The mobile device threads both the UDP broadcast functionality and the TCP server at all times, allowing for the Unity application to continue running with all simultaneous networking functionality enabled. The Pico receives this UDP message upon initializing and connects to the TCP server to begin data transmission. Both the C# and MicroPython implementations heavily utilize try-catch statements, enabling both devices to fully drop from the network with no issue. Any device can drop the connection and the other respective device will immediately be ready for the UDP broadcast exchange once again to re-establish the connection. This is very advantageous for a mobile app because the user will constantly be starting and stopping it.

The TCP connection relies on a simple JSON formatted protocol with two fields, a ‘STATUS’ field, and a ‘DATA’ field. The STATUS field contains the commands:

|  |  |
| --- | --- |
| **CONN** | Sent from the Pico to the app as the first connection message |
| **ACK** | Sent from the app to the Pico to acknowledge a CONN, a data transfer, or a WAIT |
| **WAIT** | Sent from the Pico to the app to signify it is waiting. |
| **MES** | Sent from the Pico to the app to signify that it included a single measurement as a payload. Sent from the app to the Pico to request a single live measurement. |
| **DATA** | Sent from the app to the Pico to the app to request the entire saved backlog of sensing data. Sent from the Pico to the app to signify that it included the entire saved backlog as payload. |
| **CLOSE** | Sent from the Pico to the app to signify it is closing the connection, although the app does not strictly require this message to function properly after a disconnect. |
| **RST** | Sent from the app to the Pico to request all backlogged sensing data be cleared. |

And the DATA field contains measurements. The data is also in the form of a JSON encoded array containing mostly doubles like so:

{"TIME": 11, "TEMP": 25.2, "HUM": 29.6, "UV": 0.2643453}

With and entire measurement being transmitted from the Pico to the app like so:

{"DATA": [{"TIME": 13, "TEMP": 25.1, "HUM": 29.8, "UV": 0.2756926}],

"STATUS": "MES"}

To further illustrate how the systems works together, this is the entire program output for the mobile application from program start to receiving a handful of measurements: Text

Description automatically generated

Source: The Unity Editor Window

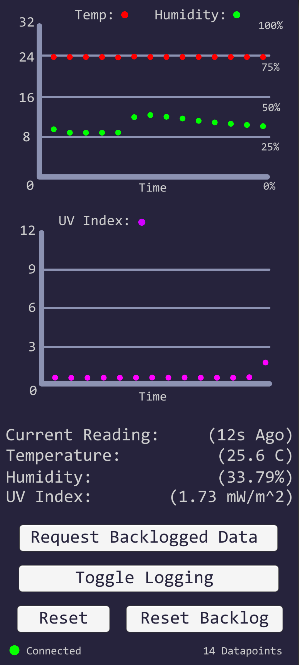
And this is the entire program output from the Pico USB REPL from program start as it performs the other half of the networking seen in the previous screenshot:

Text

Description automatically generated

Source: The Pico output from the PyCharm IDE REPL connection

After testing this solution numerous times, I have concluded that the data sensing process takes approximately 600ms and therefore the actual generation of sensing data is the bottleneck for this device. The network data transmission is an order of magnitude faster. Because of this, I consider this to be an efficient and successful networking implementation for this device. The configuration I have put in place gives live transmitted sensing at a 1s interval for active app usage, and passive locally recorded Pico sensing at a 20s interval for long-term recording. Please see the included ***AppDemonstration.mp4*** to see a real-time demonstration of the system

**Practical Usage**

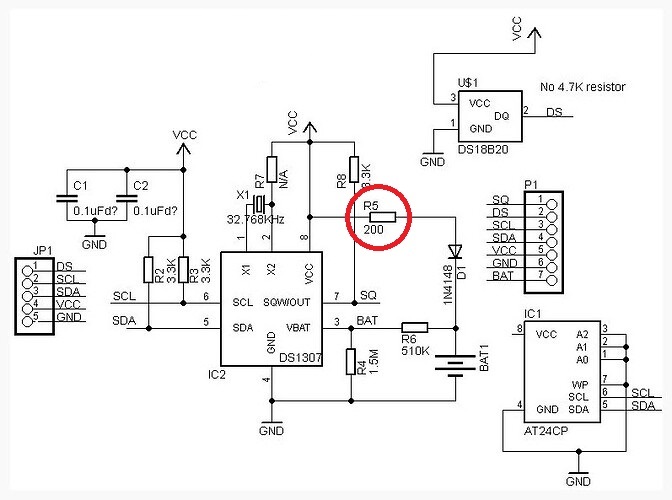
The application is easy to use and boots directly into the menu in this figure. The graphs represent the readings over time, the most recent reading is listed in the lower-middle, and there are buttons at the bottom. With the buttons we can request the stored Pico data, we can toggle the logging of data, we can reset the phone’s data storage, and we can send a request to clear the Pico’s data backlog. We can also see a live feed of the connection status and datapoint quantity at the bottom. After plenty of bug fixing the app works very seamlessly and any rare connection issue is almost always solved with a quick power-cycling of the Pico. There is a mobile recording available with this project demonstrating the functionality.

Source: My mobile application running on a Samsung Galaxy S20 FE 5G

**Implementation Hurdles**

There were many bugs to fix throughout the project, but a select few issues really brought the project to a halt. These are the issues and how I handled them:

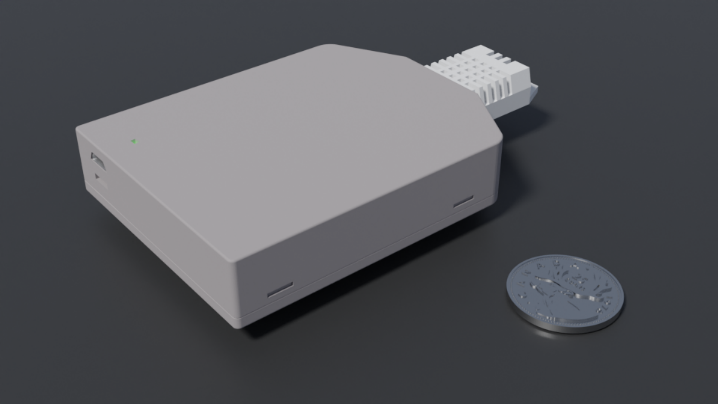
***ADC battery issues:*** The raspberry Pi Pico has many different forms of communication/processing on it with SPI, I2C, UART, and ACD being the main ones. I needed to utilize the three 12-bit analog-to-digital converter channels on the raspberry Pi for a couple different reasons. The initial plan was to use the first channel to measure the UV sensor’s output, and the second channel to measure the UV sensor’s regulated 3.3 volts to compare to voltage fluctuations in the sensor input for a more accurate reading. This is well documented on this website: <https://learn.sparkfun.com/tutorials/ml8511-uv-sensor-hookup-guide/all>**.** Finally, the third ADC channel was going to be used to measure the system’s voltage so that the LiPo battery voltage could be relayed to the app and display the module’s level of charge. The issue I encountered is that the Pico would become completely unresponsive when the system voltage pin (GPIO29) was read. This was a very difficult issue to track down, but in the end it was determined that a small quirk of the Pi Pico W is that it can not have WLAN active while reading system voltage. I do not know the reason for this, but it is not easy to get around. The only possible solutions would be to disable WLAN momentarily to read VSYS before reenabling it. I felt as though this would very tough to implement without determining the user experience, so battery level reading was omitted from the project. Because the Pico LiPo shim includes a battery cut-off circuit the device will simply be prevented from starting if the battery is dead. If a user wishes to perform a longer logging session, then they will have to ensure they have charged the device beforehand.

***RTC Recharge Circuit:*** The real time clock module I implemented over I2C strictly required a ML2032 coin cell battery, i.e., a rechargeable lithium coin cell battery. These batteries were not that easy to find while the much more prevalent CR2032 coin cell battery can be found in almost any store or gas station. I am not well versed in electrical engineering but after looking over the schematic I realized that this module has no charge circuit at all. Instead, it simply wires the Pico’s system voltage to the coin cell battery while it’s on. This was Odd. I was able to remove a single surface mounted resistor to disable this recharge functionality and because the quartz oscillator draws a current measured in picoamps to keep time, this should not be an issue.

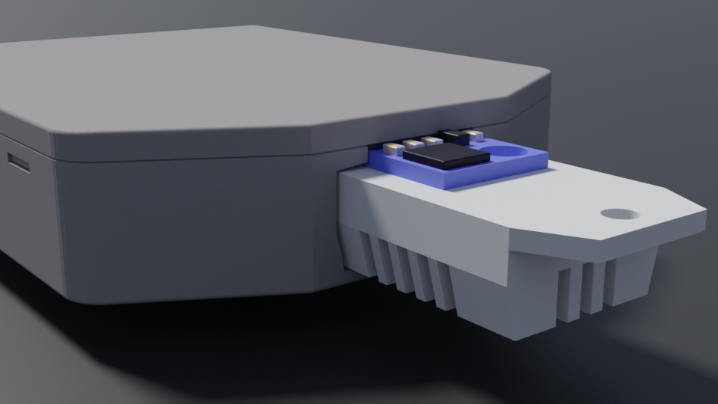
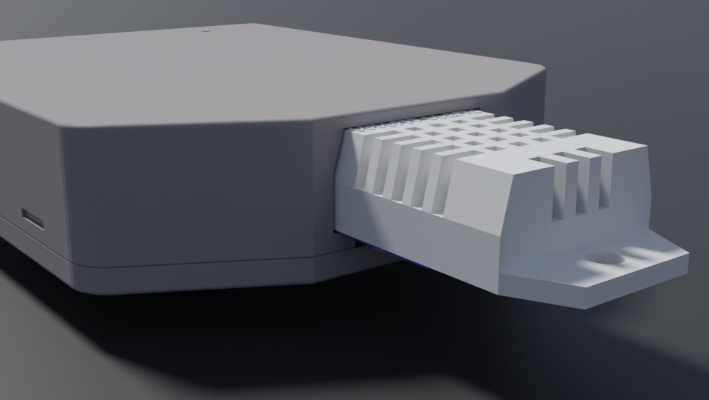
Source: DS1307 Datasheet

***Background Unity Apps:*** Once I began my Unity implementation it was my intention to have the mobile app run in the background to allow the Pico to never require any on-board logging. Unfortunately, upon further research, Unity runs as an “Activity” within Android, and activities are only present in the foreground of processing. “Services” can only run in the background on Android, and to create a service I would have to leave unity and develop directly in something like Android Studio. This was a hidden detriment to using Unity, but because the Pico is capable of minor data storage the background logging of data could ultimately be kept local. This solution does also reduce network traffic.

***The Final Device Enclosure:*** The purpose of this project was to create a wireless IoT device that I could use to easily log data, and I have completed the code and hardware implementations required. Commonly the final step in creating a physical device like this is to package it up into a compact, handheld, usable device. I unfortunately due to time constraints I could not fully complete the long process of iteratively 3D printing and testing enclosures to find the best one, although my 3D models do exist. I have rendered out example images that demonstrate the look and size of my device with a Canadian quarter for scale:



Source: Created and Rendered for this project in Blender, by Jacob Diaz.



We can see the back-to-back sensors on one side and the connector, power button, and status light on the other. The connector is from the Pico and acts as a charging port as well as a debug port, the black power button is below it, and the status light on the top displays the WLAN connectivity status. We can also see that the enclosure has enough room to fit all the components, as well as snap-fit tabs to keep it closed. This particular enclosure is large enough to fit all loose components wired together while still being handheld, but its size could be further reduced with the production of a dedicated circuit board.

***Firewalls:*** When this project was just starting earlier in the semester I ran into a lot of networking issues. The Pico could not talk to my other devices, and I did not understand why. I initially attributed it to my lack of experience with C# networking, but this was not the cause. After loading up Wireshark and using some of the knowledge from COMP4300 I was able to narrow down the issue: A screenshot of a computer

Description automatically generated with medium confidence

Here we can see the UDP broadcasts in blue and the grey TCP transmission from the Pico to the app. We can clearly see that the Pico is fully functioning, but something is blocking the TCP transmission. Constant TCP retransmissions fail, and nothing gets through. This was caused by my Windows Defender firewall being active on the PC I was testing the mobile app on. Luckily using Wireshark allowed me to narrow down this issue a lot faster than other regular debugging techniques would.

**Further Work**

I am satisfied with the outcome of this project, and I feel like the device works very well. But if we were to take it one step further, I have some ideas that could make the entire implementation much more robust and take the project one step further. Supporting more Pico devices within the server would not be difficult to implement and having multiple sensors available within the app’s UI would open up a lot more use-cases past the initial 3D printing applications. Some Picos could automatically act as sensors and some Picos could simply log data via a UART MicroSD card connection. Other types of sensors could also be incorporated. For example, a Volatile Organic Compound (VCO) sensor could be implemented to measure VCO ppm, because some resins are harmful to inhale for long periods of time.

Currently I must flash my home network’s credentials onto the Pico to allow it to correctly log into the network. If I were to dive deeper into this project, I would investigate ways to automatically verify a “headless” device on the network without this single memory flashing requirement. I believe many consumer smart home devices solve this by hosting a small WiFi network on the smart device itself. The app then connects to the smart device’s network, provides credentials and then the smart device can log into the real home network. I do not believe the Pico W has this capability, although it would be an interesting implementation to investigate.

Finally, if I were to take this device into a more serious production context it would be quite simple to create a custom schematic and circuit board for all the components to be mounted to. There are many cheap PCB manufacturing services, and it would make the assembly process much smoother. A dedicated PCB would also make the creation of an enclosure much easier as there could be mounting holes and no more loose parts.

**Conclusion**

This device has solved the problem. I use it to detect fluctuations in the 3D printer room’s temperature while printing, I can use its live sensing feature to check for a temperature gradient across a room, and I can leave it with my stored printer filaments to monitor humidity levels in the material if I have any worries. I have also been able to measure UV light around me 3D print room, and it turns out that my windows do not let in much UV light with the room at a UV index of 0.4 during the day. But, even just opening my windows does allow enough indirect UV to enter the room and be detected, which is very good to know. I have also discovered another use for the device that I did not anticipate. I use UV lights to perform the final curing process on a model when it has left the SLA printer, and I have always wondered if the UV lights I use have the potential to tan or burn me. When I use the UV sensor at a distance of1m the light is negligible, but at a distance of 1 foot I can register a UV index of 3.5 which can burn me in 25 minutes! With enough time if I was curing something on my desk while I worked, I could get a burn, very good to know. The entire project cost approximately $25.00 CAD in hardware (not counting 3D printer filament used to prototype the enclosure), and with the information it has given me I consider it to be well worth the price.