Happy Plant

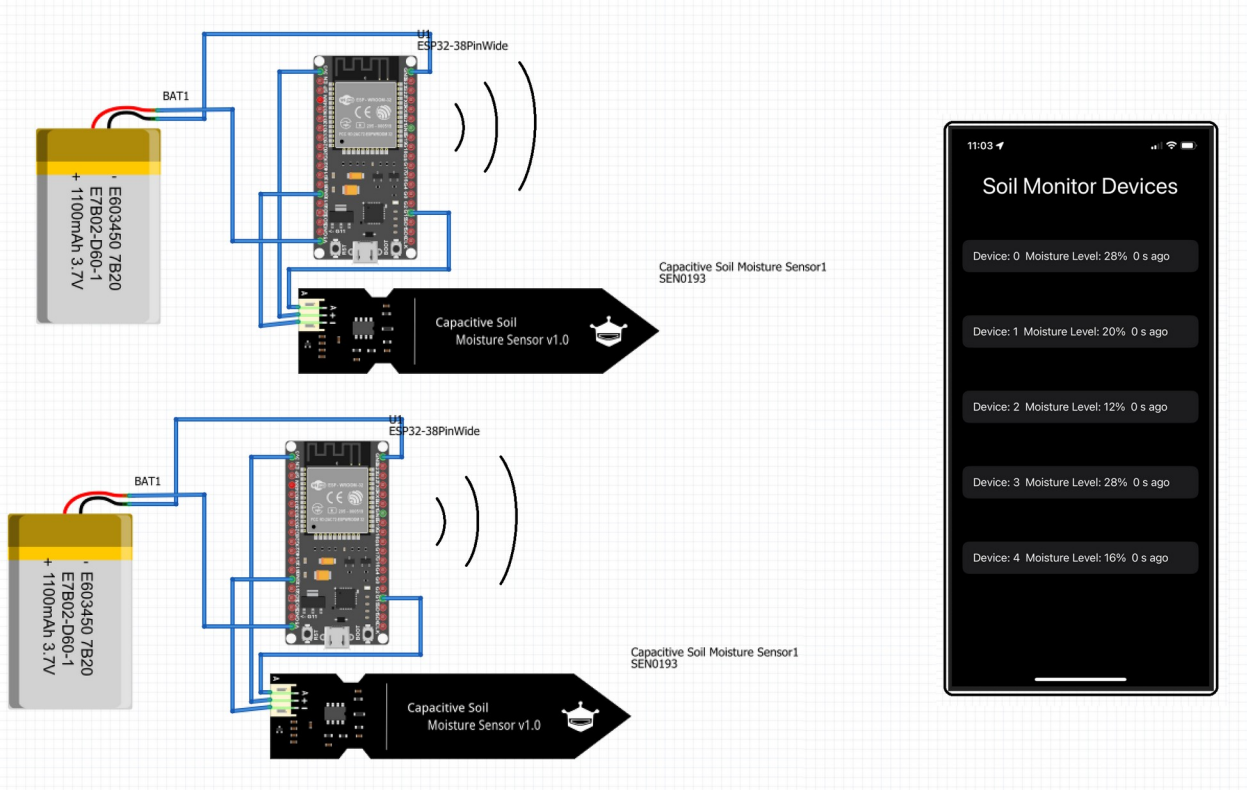
Joshua Montoya

Motivation

This project was performed with the goal of employing a plant health monitor. The initial effort would be measuring the water content in the soil and being able to monitor this data in real time for multiple plants on a single device, preferably a mobile one, with the ability to add functionality later. Added functionality might include monitoring soil pH and ambient temperature and humidity levels. Also, the app functionality could be improved from simply monitoring real-time data, to assisting a user with adding a device, and plotting and viewing historical data from the sensors. This app could be used to achieve healthier outcomes for houseplants.

Technical Approach:

After some research, I decided a simple approach to this would include a battery-powered microcontroller with wireless functionality, connected to a sensor for measuring the water content, and a mobile device for viewing soil water content levels of all devices within range. The sensor values would be read periodically by the microcontroller, and then sent wirelessly to the mobile device. A diagram of this approach is as follows:



Made with Fritzing and Paint.

Implementation Details:

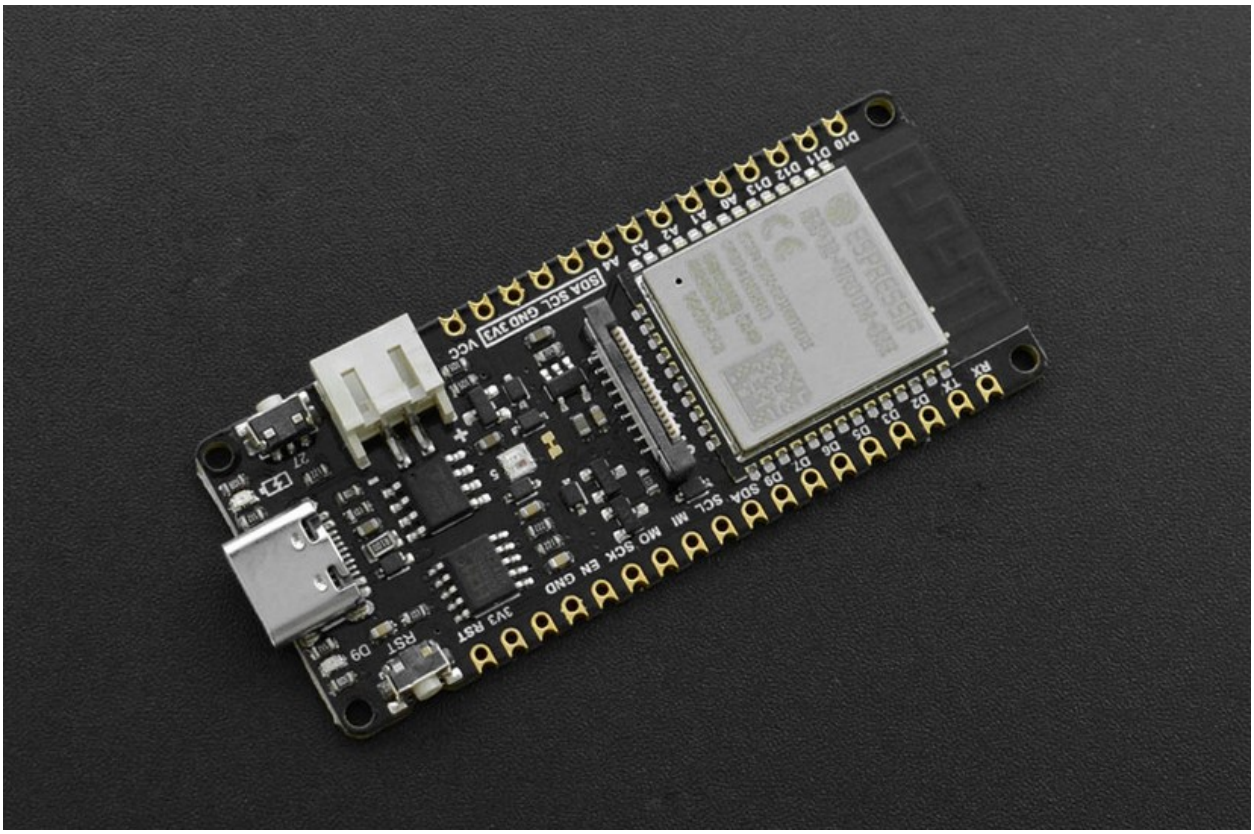
For the sensor, there were two main options. One was a conductive sensor, the other capacitive. The conductive sensor quickly appeared to be a bad option, as many reports were made about how quickly these sensors corrode and become unusable. For the capacitive sensors, there is widespread availability for both v1.2 and v2.0. Practically, they are the same product, and either one would work. I purchased some v1.2 sensors from vendor Songhe:

When researching possible issues with this device, it became clear a conformal coating would need to be applied to greatly extend the life of the item, so I purchased MG Chemicals 419D Conformal Coating and applied two coats to each sensor.

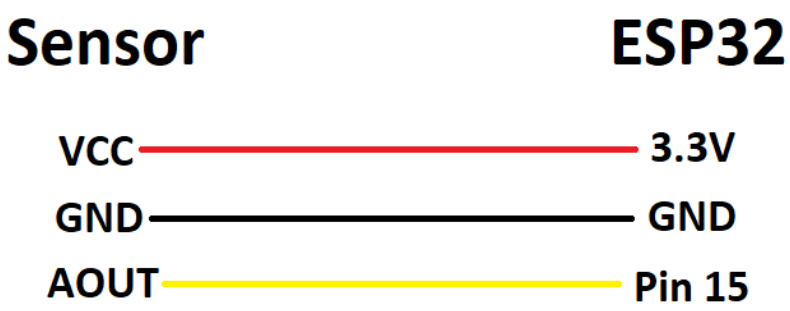
For the microcontroller, there were a lot more options available. It was important for this device to be relatively small so it could fit in snug spaces. Also, power consumption was a crucial consideration. After some research, I came across the ESP32, a compact, WiFi and Bluetooth compatible device, as well as a guide to reduce power consumption of the ESP32 at the following link: https://diyi0t.com/reduce-the-esp32-power-consumption/, where measurements of power consumption for different ESP32 devices were provided:

	Reference [mA]	Light-Sleep [mA]	Deep-Sleep [mA]	Hibernation [mA]
ESP32 – DevKitC	51	10	9	9
Ai-Thinker NodeMCU-32S	55	15.05	6.18	6.18
Adafruit HUZZAH32	47	8.43	6.81	6.80
Sparkfun ESP32 Thing	41	5.67	4.43	4.43
FireBeetle ESP32	39	1.94	0.011	0.008
WiPy 3.0	192	-	0.015	-

I decided to further research the FireBeetle ESP32. After researching, it appeared many users of the FireBeetle device were able to attain a deep sleep current draw well under 0.1 mA. I purchased the FireBeetle ESP32-E:

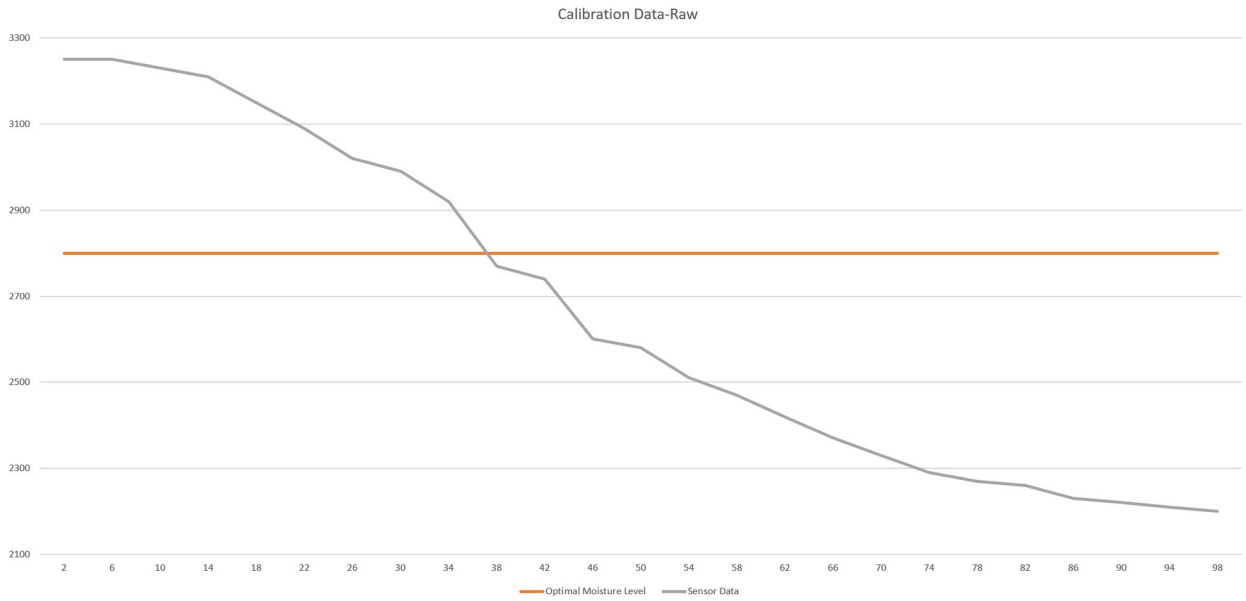


I cut off the connector head from the wires coming from the sensor and soldered these directly to the ESP32 board, with connections as follows:

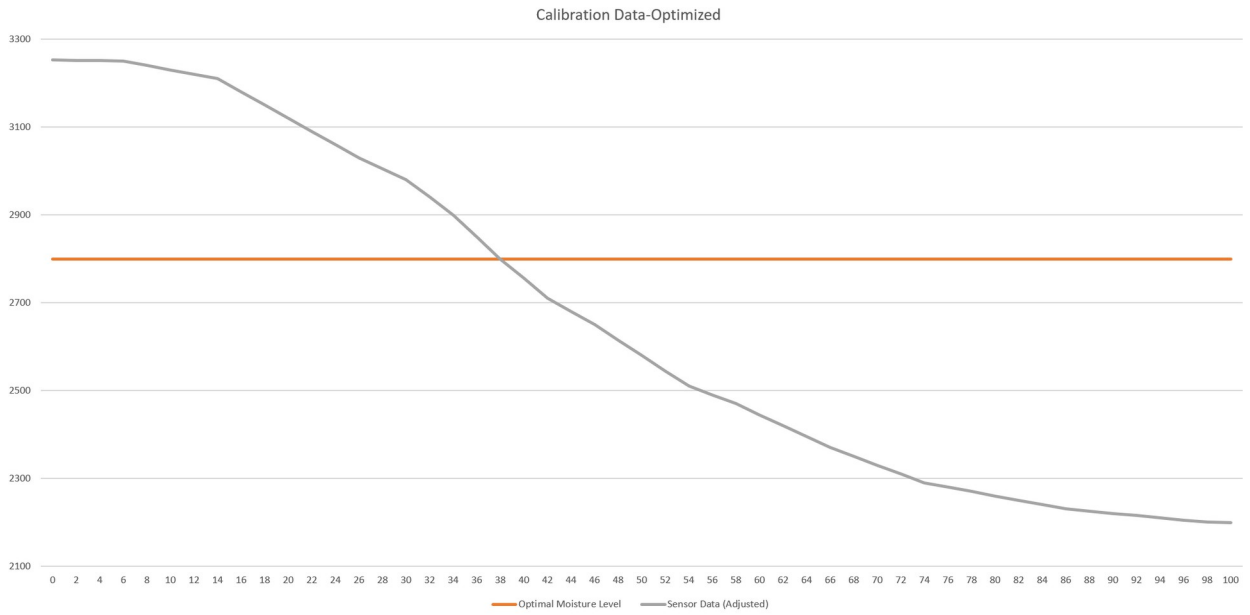


Now it was time to program the device with the Arduino IDE and calibrate the sensor data. Since the relationship between the water content and the sensor output is not linear, I needed to gather extensive data to produce proper readings to my mobile device. Upon reading the sensor outputs, I found that a sensor exposed to dry air produced an output around 3450, and one inserted in water outputted ~2200. Each sensor produced very similar outputs for these same cases. This was good news as I would need to get calibration data from only one sensor that could be applied to the rest.

To get this data, I filled a pot with dry soil and inserted a sensor until the notch was level with the surface. I took note of the output, then added 2 tablespoons of water, and recorded the output again. I repeated this process until the soil was completely soaked with water. I then translated these sensors to a linear water content output between 0 and 100. The results are plotted below:



The pattern became clear, and I manually created a smoother output to use for my translations, plotted at 51 points (0-100 with a step size of 2; due to this, the output will always be even). See this plot below:



The orange line in these plots is an approximate ideal water content level just after the plant is watered, and the soil for most plants should never have a higher water content than this level. This mapped out to be a little less than 40 percent. So, if a reading is 40 percent or higher in our app a few minutes after the plant has been watered, the plant roots are likely oversaturated, and the soil should be amended. On the dry side, a plant could safely be left until the soil reaches about 10 percent, or even less. Of course, this depends on the plant and the density of the soil. Loose soil will naturally produce lower water content percentages.

I manually wrote out this data in the Arduino IDE to produce a map called calibrationData:

```
//this maps the sensor read value to an accurate moisture level between 0 and 100%
std::map <int, int> calibrationData { {10000 , 0}, {3253 , 0}, {3252 , 2}, {3251 , 4}, {3250 , 6}, {3240 , 8},
{3230 , 10}, {3220 , 12}, {3210 , 14}, {3180 , 16}, {3150 , 18}, {3120 , 20}, {3090 , 22},
{3060 , 24}, {3030 , 26}, {3005 , 28}, {2980 , 30}, {2940 , 32}, {2900 , 34}, {2850 , 36},
{2800 , 38}, {2755 , 40}, {2710 , 42}, {2680 , 44}, {2650 , 46}, {2615 , 48}, {2580 , 50},
{2545 , 52}, {2510 , 54}, {2490 , 56}, {2470 , 58}, {2445 , 60}, {2420 , 62}, {2395 , 64},
{2370 , 66}, {2350 , 68}, {2330 , 70}, {2310 , 72}, {2290 , 74}, {2280 , 76}, {2270 , 78},
{2260 , 80}, {2250 , 82}, {2240 , 84}, {2230 , 86}, {2225 , 88}, {2220 , 90}, {2215 , 92},
{2210 , 94}, {2205 , 96}, {2200 , 98}, {2199 , 100}, {0 , 100} };
```

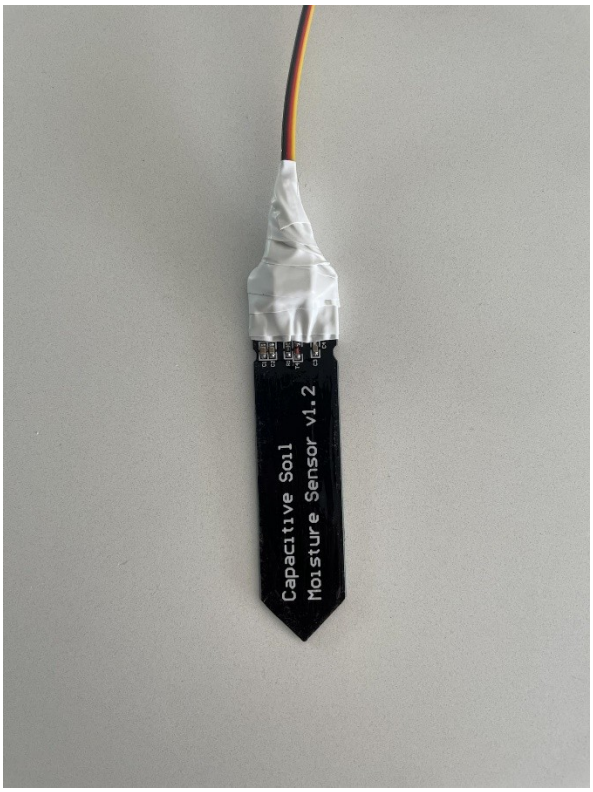
With the data in order, the next step is to decide on the protocol for sending it. I decided Bluetooth LE would be the best protocol for this task, as there is very little data involved, and we want to conserve as much power as possible. I experimented with a few ways of sending and displaying the data. Initially, for test purposes, I was sending that signal to another ESP32 and displaying the result on a screen. I had plenty of issues here. I was able to communicate one device's values easily, but when I tried multiple devices simultaneously, it was more difficult to maintain a connection, especially when I tried to implement it with deep sleep periods.

I decided to use BLE GATT (Generic Attribute Profile), the standard server-client BLE protocol designed for exchanging structured data. Each ESP32 acts as a GATT peripheral, advertising a custom service with a single readable and notifiable characteristic that holds the current moisture level as a single byte. Each

device is identified by a unique BLE device name (e.g., "HappyPlant_4"). The peripheral advertises for a short window after each sensor reading, extending that window if a client connects, then enters deep sleep. This approach is well-suited for the application: there is little data to send, the connections are short-lived, and BLE's low energy profile is compatible with battery-powered operation.

The iOS app uses CoreBluetooth to act as a GATT central. It scans for peripherals advertising the custom service UUID, connects to each one, discovers the moisture characteristic, and reads and subscribes to its value. Discovered devices are displayed dynamically as they are found, identified by their BLE device name. The data displayed reads "Device: [name] Moisture Level: ##% [time] ago."

Before I placed the sensors in the plants, I added some electrical tape around the connector so it won't get wet:



It's not a permanent solution, but it works.

I then attached the batteries (1200 mAh LiPoly Rechargeable batteries) to the ESP32 devices and installed all the devices in the plants most in need of monitoring.

Device 0 with a Satin Pothos:

Device 1 with an Areca Palm:

Device 2 with a Croton:

Device 3 with a Spider Plant:

And Device 4 with a Neon Pothos:

The total cost was as follows:

Component	**Price**
5 Capacitive Sensors	$ 10.99
5 Fire Beetle ESP32-e	$ 36.90
5 1200 mAh LiPoly Rechargeable Batteries	$ 44.80
1 55mL Bottle Conformal Coating	$ 12.95
Total	**$ 105.64**
Total per device	**$ 21.13**

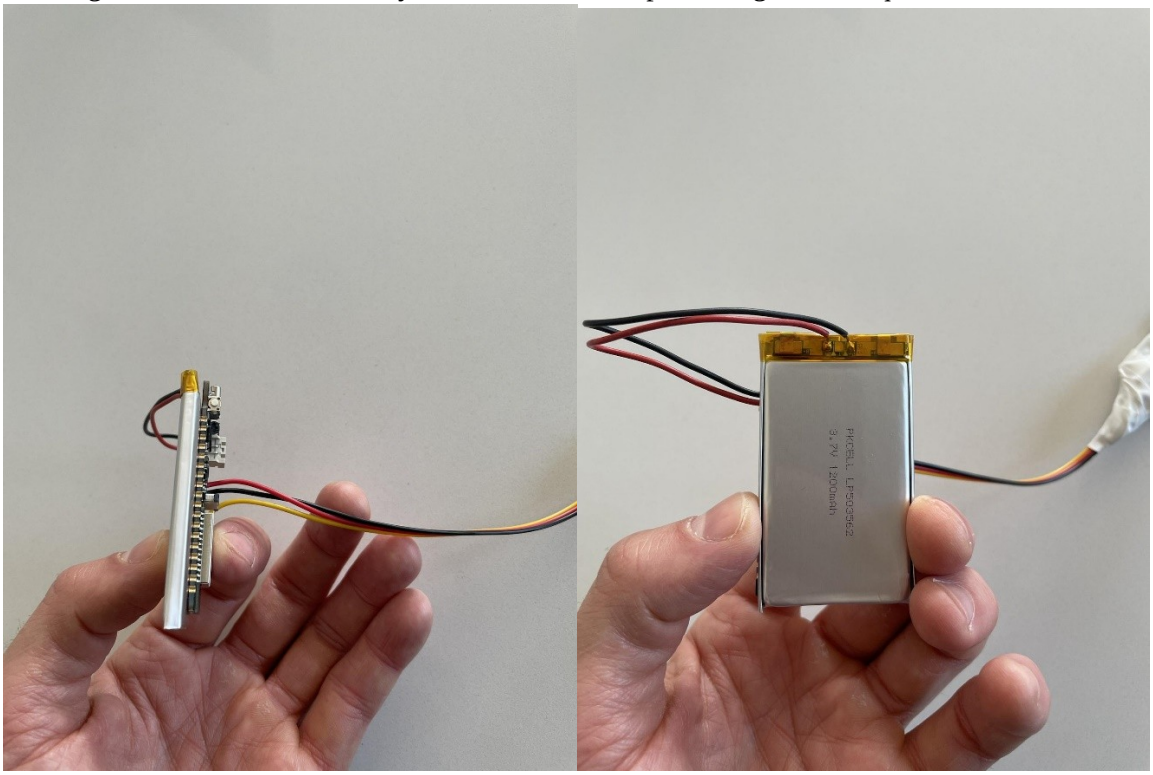
I used a small portion of the bottle of conformal coating so the cost for device is really a bit lower. Costs could be cut substantially by using smaller batteries, especially non-rechargeable ones.

Results:

The resulting device and app were simple, but a success. Upon opening the app, the readings never take more than a few seconds to show up. Regularly checking the readings is a lot easier than regularly checking the soil manually and has provided insight into how the plants use water differently. I initially thought this preliminary device would be underwhelming and a hassle, but I'm pleased with it and want to add some functionality and finishing touches to it.

The device functionality could be expanded in the following ways, some of which would be quite simple:

1. Housing for the ESP32 and battery. The devices when placed together are quite thin:



   A 3D-printable CAD design could be made of a case that fits the two, with a small opening for the wire to the sensor, an opening on the other side for access to the USB port for charging, and a clip on the side that would allow the device to be attached to the side of a pot.
2. Better, waterproof protection for the connector.
3. More sensors: pH, ambient humidity, ambient light, and temperature sensors could all be useful. With GATT, each additional sensor value can be added as a new characteristic on the existing service, making expansion straightforward.
4. A more functional app that looks better, helps a user add any number of new devices that have been loaded with the software, sends notifications when sensor outputs reach action levels, and plots historical data received from our sensors. The app could be set up to periodically receive data in a background process.
5. Store data in RTC memory of the ESP32 device when the mobile device is not in range. When back in range, send the buffered history via a dedicated GATT characteristic and free up memory. Additional data points would be stored only every few minutes. There are 8 kB of RTC memory available, which would be plenty for temporary storage of a small array of values. Since GATT

already establishes a server-client connection, transmitting this historical data would be a natural extension of the current implementation, and would allow a data plot on the mobile device to be without gaps.

6. In addition to storing this data and plotting, a machine learning model could be developed over time to better identify individual plant needs.
7. Optimized battery life, and possibly a smaller battery. Ideally, it could be run for a year or two with a coin battery.
8. A battery level indicator.

This device could theoretically be refined and sold, as there would be demand for such a product, including from myself and many plant aficionados I know.