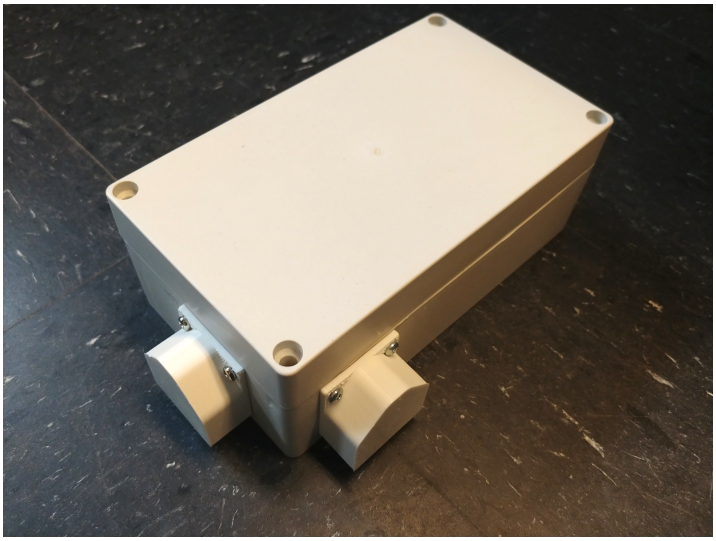
James Li
11.007
Prof. David Hsu
October 20, 2018

Lab Report 1



The completed sensor.

Introduction and Objectives

The purpose of this project is to learn to build a low-cost Internet-connected air quality sensor for eventual deployment as part of a network of sensors to monitor air quality across the MIT campus. This lab report describes the construction of an initial "version 1" sensor: decisions made during implementation, lessons learned, and possible future work.

Performance objectives for the sensor include:

- Reliability/longevity: The sensor package should be robust enough to withstand the rigors of use, including hazards such as shock and water. Software and connectivity should be stable, and battery life should be reasonable for the intended use.
- Quality of measurement: Sensor readings should accurately reflect environmental conditions outside the sensor.

Hardware and Software Overview

The sensor is built around an Arduino MKR GSM 1400 microprocessor, which handles data processing, has cellular network capabilities, and can be battery powered. A common

DHT22 sensor measures temperature and humidity, while a Plantower PMS5003 laser scattering sensor measures particulate matter. We use a 3.7V 5200 mAh lithium-ion battery for power, and we use a small Raspberry Pi computer fan to circulate air. All sensor components are enclosed in a plastic waterproof electronics enclosure. Integration of a GPS sensor is planned for future versions of the sensor.

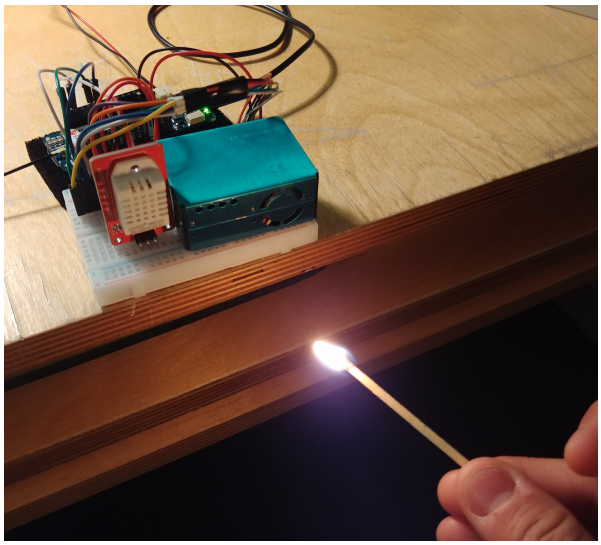
The sensor connects to the Internet via a GSM network and the Hologram network service for connected devices. It logs data to the Thinger webservice, which provides an Arduino library and a web interface for interacting with the sensor and viewing data.

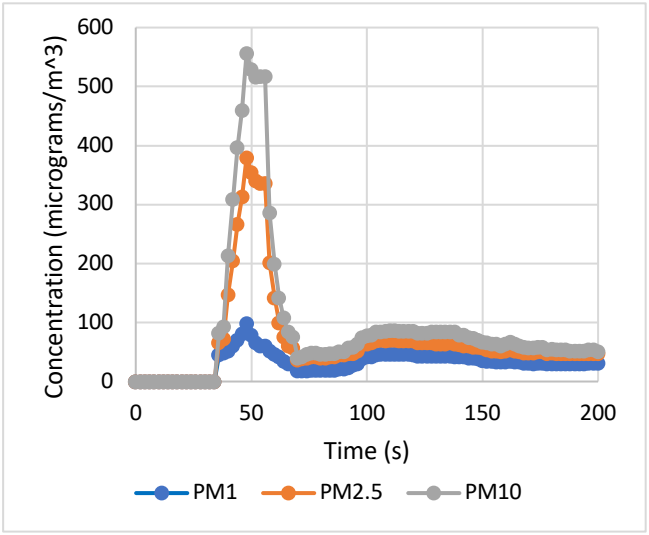
Hardware Integration

Integration of the hardware components consisted of physically connecting the components to the Arduino via a breadboard, writing and testing Arduino sketches for each component, and then integrating these separate sketches into a single sketch with combined functionality. This section details issues that were encountered and decisions that were made during hardware integration.

Several issues and decisions related to the PMS5003 sensor:

- The PMS5003 requires 5V power in order to run the internal fan, but in the absence of USB power to the Arduino only 3.3V could be supplied to the PMS sensor. This does not appear to inhibit the operation of the sensor, except that the internal fan does not run. Because this source of power consumption was eliminated, I elected not to implement the sleep functionality of the PM sensor at this stage.
- Powering the PM sensor with a 5V supply allowed me to examine the airflow from the fan and determine that the four small holes are the sensor air inlet, and the large fan opening is the sensor air outlet. I used this information in my design of airflow through the sensor housing (see “Housing and Packaging”).
- Initially, the sensor consistently reported 0 particulate matter, apparently because the air in my workspace is very clean. In order to verify that the sensor was active and responsive, I lit a match near the sensor and observed that the sensor measurements spiked due to the smoke (see below).





Testing the PM sensor with a match.

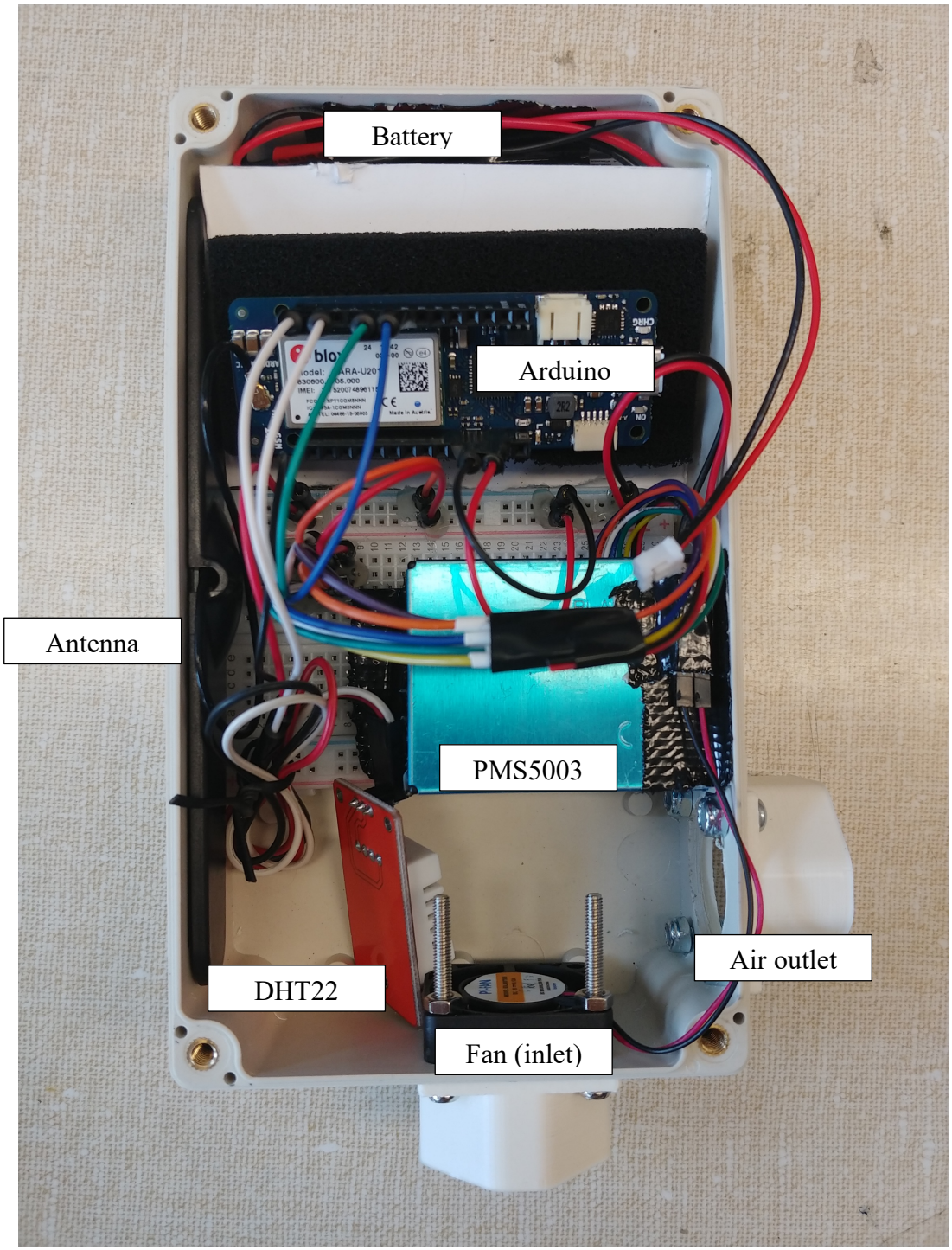
Other notes:

- Reversing the polarity of the battery (as indicated by the red and black wires) was necessary to match the polarity of the battery port on the Arduino board.
- I chose to program any errors in sensor readings to be reported as values of -1. This allows errors to be reported in the standard numerical output format (where a text-based error message might cause issues).
- I hoped to power the case fan from a digital output pin so that it could be controlled via software, but it appears to draw more current than can be provided, so the case fan needs to be powered from the main 3.3V (VCC) supply pin.

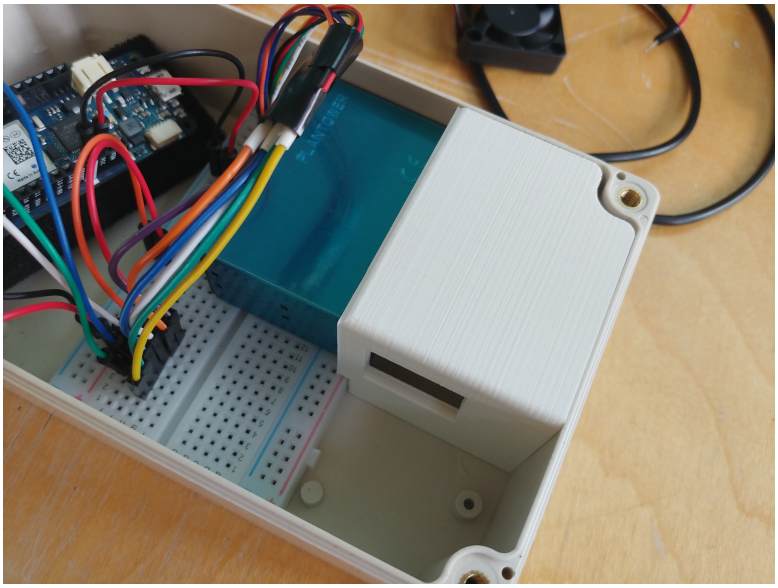
Housing and Packaging

I packaged the sensor components into the housing as shown in the below diagram. In fitting the components into the housing, I sought to keep wiring connections organized and manageable to simplify future troubleshooting. Airflow through the housing is confined to the smallest and least constricted space possible in order to maximize the rate of airflow through the sensor. By minimizing the volume of air circulating in the sensor, the relatively low-powered Raspberry Pi fan can move air more quickly. Faster air circulation should allow the internal conditions of the sensor to more closely track the outside environment. The fan is mounted to drive air past the DHT22 and into the inlet holes of the PMS5003, and the outlet hole is placed next to the outlet of the PMS5003. The hope is that this configuration will compensate for the internal fan of the PMS5003 not being powered. The PMS5003 datasheet recommends that the

inlet and outlets of the sensor be physically separated to prevent air mixing, but such a barrier was left out of this design for simplicity. A 3D-printed “wall” confines the airflow and serves as a mount for the DHT22.

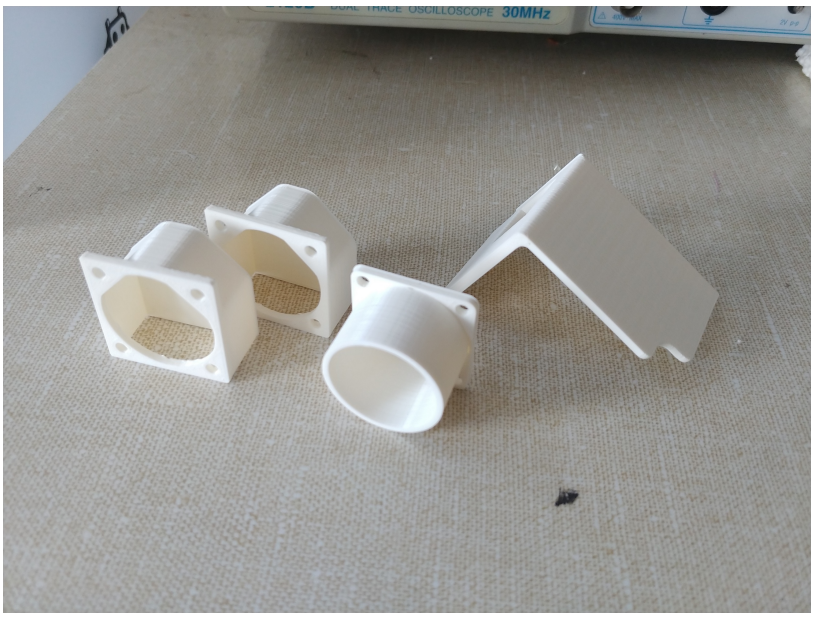


Layout of components in housing (airflow wall removed for visibility).



The 3D-printed airflow “wall” (in white) with cutout for DHT22.

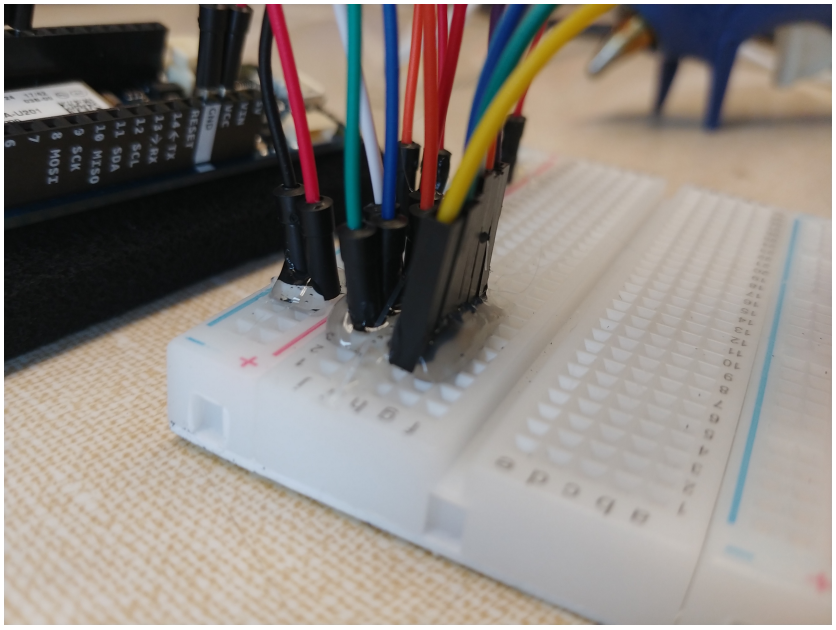
I selected 3D-printed external hoods as the solution for preventing water ingress. The hoods have the same hole pattern as the Raspberry Pi fan, so that the same M3 screws can be used for these components. 3D printing the hoods allows for future design flexibility; in this iteration, I printed an angled hood and a straight hood for different shielding needs. The hoods can also be installed in multiple orientations (increments of 90º) to allow for different sensor mounting configurations.



3D printed hoods (left and center) and airflow wall (right).

A piece of foam core board was placed between the Arduino and the battery pack to locate the soft foam backing for the Arduino and prevent the Arduino pins from puncturing the battery. The Arduino can be easily pulled out of its soft foam to access the USB port.

All jumper wire connections into the breadboard were hot glued for strain relief. Other jumper wire connections (e.g. to the Arduino) were taped to prevent disconnections. Duct tape was used to secure the PM sensor, the battery, and the 3D-printed airflow wall. The breadboard and foam core board are friction fit into the housing.



Hot glued breadboard connections.

Cloud Integration

Using Thinger "data buckets." data sent from the sensor can be saved with timestamps for future download and analysis. I found several different possible ways for the sensor to interact with the Thinger cloud service:

- Thinger can pull data from the sensor. This allows for the refresh rate to be controlled from the Thinger dashboard, but requires that the sensor be constantly running through its main loop without delays.
- The sensor can stream data to Thinger whenever a change is detected in sensor values.

- The sensor can push data (using the write_bucket method) to the data bucket. The sensor controls when it connects to Thinger, and delays can be used in the main loop.

In the interest of power and data conservation, I chose the third option – to push data to Thinger from the sensor. The rate of data acquisition to the bucket is limited to once per 60 seconds by Thinger; upgrading to a paid account allows for shorter write intervals. My sketch thus performs a measurement and pushes data to Thinger once per loop, with a 60-second delay in each loop. Based on the Thinger device dashboard, each data upload is about 65 bytes in size.

The Thinger dashboard can be used to display nice graphs of sensor measurements over time.

The Thinger dashboard, reflecting PM from another match test.

Future Work

The following are directions for future improvements to the sensor:

- The GPS sensor needs to be integrated.

- Using a transistor or other switching device to allow the case fan to be controlled by the Arduino would reduce power consumption. Alternatively, an external timing chip could periodically power the entire system on and off.
- A battery voltage measurement would allow the charge level of the battery to be monitored from the Thinger dashboard.
- Rubber seals should be added between the hoods and the housing to prevent water ingress. In addition, bug screens on the inlet and outlet would prevent debris from entering the enclosure.
- Status LEDs visible from the outside of the enclosure would help in quickly determining the state of the sensor.
- A hardware on/off switch between the battery and the Arduino would make powering the sensor on and off easier.

```
// temperature, humidity, particulate matter sensing with thinger integration

//////////
// SETUP //
//////////

// include necessary libraries
#include <DHT.h>
#include <DHT_U.h>
#include <PMS.h>
#include <MKRGSM.h>
#include <ThingerMKRGSM.h>

// DHT setup
#include "DHT.h"
#define DHTPIN 5
DHT dht(DHTPIN, DHT22); // DHT 22 (AM2302), AM2321

// Enable a new serial port
#include <Arduino.h>
#include "wiring_private.h" //no need to download this library
Uart mySerial (&sercom3, 0, 1, SERCOM_RX_PAD_1, UART_TX_PAD_0); // Create the new
UART instance assigning it to pin 0 and 1

// PM Sensor setup
#include "PMS.h"      //Use the library manager to install the PMS library
PMS pms(mySerial);    //PMS pms(pmsSerial); software serial not working with PMS and
FONA strange cant explain
PMS::DATA data;

// GSM and Thinger setup
#define USERNAME "jamesli"        // These credentials need to be changed to match
your credentials
#define DEVICE_ID "jamesli_v1"             // must match device created in thinger
#define DEVICE_CREDENTIAL "E&ATSUB9c417"  //must match credentials created in thinger
#define PIN_NUMBER    "" //sim pin, leave blank
#define GPRS_APN      "hologram"   //apn for the hologram sim card
```

```
#define GPRS_LOGIN    ""           //leave blank
#define GPRS_PASSWORD ""           //leave blank
ThingerMKRGSM thing(USERNAME, DEVICE_ID, DEVICE_CREDENTIAL);

// Set up variables for measurements
int pm1;
int pm25;
int pm10;
float h;
float t;

////////////////////
// MAIN FUNCTIONS //
////////////////////

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  setupThinger();
  setupPMS();
  printHeader(); // print header for CSV
}

void loop() {
  // update sensor values
  readPMS();
  readDHT22();
  Serial.println(); // print to serial monitor

  // connect to thinger.io and upload reading to bucket
  thing.handle();
  thing.write_bucket("sensorData", "sensorData");

  delay(60000); // wait 50 seconds (total 1 minute between measurements)
}

/////////////////////////
```

```
// SUPPORT FUNCTIONS //
//////////////////////

// Setup for Thinger
void setupThinger() {
  thing.set_pin(PIN_NUMBER); // optional set pin number
  thing.set_apn(GPRS_APN, GPRS_LOGIN, GPRS_PASSWORD); // set APN

  thing["sensorData"] >> [](pson& out) {
    out["temp"] = t;
    out["humidity"] = h;
    out["PM1"] = pm1;
    out["PM2_5"] = pm25;
    out["PM10"] = pm10;
  };
}

// Setup for PMS
void setupPMS() {
  mySerial.begin(9600);  // activate communications to the PMS sensor **sensor TX ->
board RX; sensor RX -> board TX; these are MKR board pins 13 and 14**
  pinPeripheral(0, PIO_SERCOM); //Assign TX function to pin 0
  pinPeripheral(1, PIO_SERCOM); //Assign RX function to pin 1
}

// Print header for CSV
void printHeader() {
  delay(2000);
  Serial.println("PM1, PM2.5, PM10, Humidity, Temp");
}

// Read DHT
void readDHT22(){
  h = dht.readHumidity(); // Read and store humidity
  t = dht.readTemperature(); // Read temp as Celsius (the default)

  // Set failed reads to -1 to indicate error
```

```
  if (isnan(h)){
    h = -1;
  }
  if (isnan(t)){
    t = -1;
  }

  // Print data to serial monitor
  Serial.print(h); Serial.print(", "); Serial.print(t); Serial.print(", ");
}

// Read and print the data from the PMS sensors
void readPMS() {
  pms.requestRead();

  if(pms.readUntil(data,2000)) {
    pm1 = data.PM_AE_UG_1_0;
    pm25 = data.PM_AE_UG_2_5;
    pm10 = data.PM_AE_UG_10_0;
  }

  // Set failed reads to -1 to indicate error
  else {
    pm1 = -1;
    pm25 = -1;
    pm10 = -1;
  }

  // Print data to serial monitor
  Serial.print(pm1); Serial.print(", "); Serial.print(pm25); Serial.print(", ");
  Serial.print(pm10); Serial.print(", ");
}

// Attach the interrupt handler to the SERCOM, function to eneable the myserial
interface
void SERCOM3_Handler() {
  mySerial.IrqHandler();}
```