MECH 49X
Dossier 11 - Code

Team 26

March 31, 2018

The code for running for running the sensor package is:

`’_all.ino’.`

This code runs all of the sensors, prints data to Serial connection, and publishes the data to ThingSpeak. The code is presented on the following pages. The logic to the code is as follows:

1. Turn on the sensor package
2. Turn on CO2 sensor
3. Read from MRT, SHT, and VOC sensors while CO2 sensor warms up (PM sensor is off)
4. Read from CO2 sensor
5. Save CO2, MRT, SHT, and VOC average readings
6. Turn off CO2 sensor and turn on PM sensor
7. Read from MRT, SHT, and VOC sensors while PM sensor warms up (CO2 sensor is off)
8. Read from PM sensor and save value
9. Push CO2, PM, MRT, SHT, and VOC readings to ThingSpeak
10. Turn off PM sensor and turn on CO2 sensor
11. Repeat forever

```
/*
 * Script _all.ino
 * This script runs the sensor package
 * Uses objects for each of the sensors
 * Prints information to Serial screen
 * Publishes data to ThingSpeak
 */

#include "CALCULATE_MRT.h"
#include "MHZ19.h"
#include "CCS821.h"
#include "SHT35D.h"
#include "MRT.h"
#include "PM.h"
#include "Time.h"
#include <Wire.h>

// create instances of objects
PM_7003 myPM;
ClosedCube_Si7051 myMRT;
ClosedCube_SHT31D mySHT;
Adafruit_CCS811 myVOC;
MHZ19 myCO2;
mrt_and_ot my_MRT_OT;

/*
 * Boolean expressions
 * start_xxx indicate whether sensor has been read from properly
 * read_from_xxx indicate whether or not to read from sensor_xxx (changes throughout code)
 * finished_xxx indicates whether done reading from a sensor (read a good average)
 */
bool start_co2 = false;
bool start_voc = false;
bool start_sht = false;
bool start_pm = false;
bool start_mrt = false;

bool read_from_co2 = true;
bool read_from_pm = false;

bool finished_co2 = false;
bool finished_pm = false;
bool finished_other_sensors = false;
bool finished_mrt_ot = false;
bool finished_voc = false;

// average reading values
int co2_ave = -1;
float sht_rh_ave = -1;
float sht_t_ave = -1;
float voc_eCO2_ave = -1;
float voc_TVOC_ave = -1;
int pm_ave = -1;
float T_g = -1;
float T_a = -1;
float T_mrt = -1;
float T_ot = -1;

bool publish_data = true; // should we publish data?

// pin numbers for pm and co2 sensors
int pm_transistor_control = A4;
int pm_tx_transistor_control = A5;
int co2_transistor_control = A3;

void setup() {
    /*
     * Start Serial and Wire connections
     * Initialize transistor control for CO2 and PM
     * Turn CO2 sensor on (make_sensor_read())
     * Test all I2C sensors (MRT, SHT, VOC)
     * Stop and wait for 30 seconds (warm-up)
     */
    Serial.begin(9600);
    Wire.begin();
    Serial.println("Initializing");

    myCO2.set_transistor(co2_transistor_control);
    myPM.set_transistor(pm_transistor_control, pm_tx_transistor_control);

    myCO2.make_sensor_read();

    start_mrt = myMRT.start_mrt();
    Serial.println("-----------------------");

    start_sht = mySHT.start_sht();
    Serial.println("-----------------------");

    start_voc = myVOC.start_voc();
    Serial.println("-----------------------");
    Serial.println("30 second delay");
    Serial.println("-----------------------");
    delay(30000);

}

void loop() {
    /*
     * Wait for CO2 sensor to warm-up (PM sensor is off)
     * Read from MRT, SHT, and VOC sensors while CO2 sensor warms-up
     * After CO2 sensor warms-up, read from CO2 sensor and save average reading
     * Save average value from MRT, SHT, and VOC sensors
     * Turn off CO2 sensor, turn on PM sensor
     * Read from MRT, SHT, and VOC sensors while PM sensor warms-up
     * After PM sensor warms-up, read from PM sensor and push all data to ThingSpeak
     * Repeat
     */

    // Decide which of CO2 or PM sensor to read from
    if(read_from_co2) {
        start_co2 = myCO2.make_sensor_read();
        start_pm = false;

        if(start_co2) {
            read_from_co2 = false;
            read_from_pm = true;
            finished_co2 = true;
        }
    }
    else if(read_from_pm) {
        start_pm = myPM.make_sensor_read();
        start_co2 = false;

        if(start_pm) {
            read_from_pm = false;
            read_from_co2 = true;
            finished_pm = true;
        }
    }

    start_mrt = myMRT.run_mrt(); //read from MRT sensor

    // Read from SHT sensor, or restart SHT sensor
    if(start_sht) {
        Serial.println("Reading from SHT Sensor");
        Serial.println("-----------------------");
        start_sht = mySHT.run_sht();
        Serial.println("-----------------------");
    }
    else if(!start_sht) {
        Serial.println("---------------------------");
        Serial.println("Not reading from SHT Sensor");
        Serial.println("---------------------------");
        Serial.println("Tring to start SHT");
        start_sht = mySHT.start_sht();
        Serial.println("-----------------------");
    }
    // Read from VOC sensor, or restart VOC sensor
    if(start_voc) {
        Serial.println("Reading from VOC Sensor");
        Serial.println("-----------------------");
        start_voc = myVOC.run_voc();
        Serial.println("-----------------------");
    }
    else if(!start_voc) {
        start_voc = myVOC.start_voc();
        Serial.println("Reading from VOC Sensor");
        Serial.println("-----------------------");
        start_voc = myVOC.run_voc();
        Serial.println("-----------------------");
    }

    // If done reading from CO2 sensor, save CO2, MRT, SHT, and VOC readings
    if(finished_co2 && !finished_other_sensors) {
        finished_other_sensors = true;

        if(!finished_mrt_ot) {
            if(start_mrt && start_sht){
                T_g = myMRT.get_MRT_ave();
                T_a = mySHT.get_t_ave();
                sht_rh_ave = mySHT.get_rh_ave();
                my_MRT_OT.calculate_mrt_and_ot(T_g, T_a);
                T_mrt = my_MRT_OT.get_mrt();
                T_ot = my_MRT_OT.get_ot();
                finished_mrt_ot = true;
            }
            else if(start_mrt && !start_sht) {
                T_g = myMRT.get_MRT_ave();
                T_a = -1;
                sht_rh_ave = -1;
                T_mrt = -1;
                T_ot = -1;
            }
            else if(!start_mrt && start_sht) {
                T_g = -1;
                T_a = mySHT.get_t_ave();
                sht_rh_ave = mySHT.get_rh_ave();
                T_mrt = -1;
                T_ot = -1;
            }
            else {
                T_g = -1;
                T_a = -1;
                sht_rh_ave = -1;
                T_mrt = -1;
                T_ot = -1;
            }
        }

        if(start_voc && !finished_voc){
            voc_eCO2_ave = myVOC.get_eCO2_ave();
            voc_TVOC_ave = myVOC.get_TVOC_ave();
            finished_voc = true;
        } else {
            voc_eCO2_ave = -1;
            voc_TVOC_ave = -1;
        }

        co2_ave = myCO2.get_co2_ave();

        if(finished_mrt_ot && finished_voc) {
            finished_other_sensors = true;
        }
    }

    // If done reading from PM and CO2 sensors, save PM reading and push to ThingSpeak
    if(finished_co2 && finished_pm) {
        pm_ave = myPM.get_pm_ave();
        finished_co2 = false;
        finished_pm = false;
        finished_mrt_ot = false;
        finished_voc = false;
        finished_other_sensors = false;

        if(publish_data) {
            char data[1000];
            sprintf(data,"{ \"Mean Radiant Temperature\": \"%3.2f\", \"Operating Temperature\": \"%3.2f\", \"CO2 Concentration\": \"%i\", \"eCO2\": \"%4.2f\", \"TVOC\": \"%4.2f\",\"PM 2_5\": \"%i\", \"Air Temperature\": \"%3.2f\",\"Relative Humidity of Air\": \"%3.2f\"}" , T_mrt, T_ot, co2_ave, voc_eCO2_ave, voc_TVOC_ave, pm_ave, T_a, sht_rh_ave);
            Serial.println("-------------------------");
            Serial.print("Data:");
            Serial.println(data);
            Serial.println("-------------------------");

            Particle.publish("IEQ Final Prototype", data, PRIVATE);

            myCO2.reset_co2_ave();
            myPM.reset_pm_ave();
        }
    }
}

```

The code for calculating Mean Radiant Temperature and Operating Temperature is:

`’calculate_MRT.cpp’ and ’calculate_MRT.h’`

This code uses the globe thermometer temperature, the air temperature, and the convection coefficient to calculate MRT and OT. This code was written entirely by Team 26 using equations from the literature. The .h file is presented first, followed by the .cpp file.

```
/*
 * This is the .h file for calculating MRT and OT
 * This code was written entirely by Team 26
 * using formulas found in Literature.
 */
#ifndef CALCULATE_MRT_H
#define CALCULATE_MRT_H

#if ARDUINO >= 100
  #include "Arduino.h"
#else
  #include "WProgram.h"
#endif

class mrt_and_ot {
    public:
        mrt_and_ot(void);

        void calculate_mrt_and_ot(float T_g, float T_a);
        float get_mrt(void);
        float get_ot(void);

    private:
        float calculate_convection_coefficient(float T_g, float T_a);
        float h;
        float T_mrt;
        float T_ot;
        float T_a;
        float T_g;
        float convection_coefficient;

        const float epsilon = 0.94;
        const float diameter = 0.04;
        const float diameter_to_power = pow(diameter, 0.4);
        const float kelvin_conversion = 273.15;
};
#endif

```

```cpp
/*
 * This is the .cpp file for calculating MRT and OT
 * This code was written entirely by Team 26
 * using formulas found in Literature.
 */
#include "CALCULATE_MRT.h"

mrt_and_ot::mrt_and_ot(void)
{
}

float mrt_and_ot::calculate_convection_coefficient(float T_g, float T_a) {
    /*
     * Calculate convection coefficient using formula in Literature
     */
    h = abs(T_g - T_a) / diameter_to_power;
    h = pow(h, 0.25);
    return(1.4 * h);
}

void mrt_and_ot::calculate_mrt_and_ot(float T_g, float T_a) {
    /*
     * Calculate MRT and OT using formulas found in Literature
     */
    T_g = T_g + kelvin_conversion;
    T_a = T_a + kelvin_conversion;
    convection_coefficient = calculate_convection_coefficient(T_g, T_a);
    T_mrt = convection_coefficient / epsilon * (T_g - T_a);
    T_mrt = T_mrt + pow(T_g,4);
    T_mrt = pow(T_mrt,0.25);
    T_ot = 0.5 * (T_a + T_mrt);
}

// Getter functions for MRT and OT
float mrt_and_ot::get_mrt(void) {return(T_mrt);}
float mrt_and_ot::get_ot(void) {return(T_ot);}
```

The code for running the PMS7003 Particulate Matter sensor is:

`'PM.cpp' and 'PM.h'`

This code reads from the PM sensor several time and takes an average value of all of the readings. The PMS7003 communicates using a UART connection. This code was written entirely by Team 26. The .h file is presented first, followed by the .cpp file.

```
/*
 * This is the .h file for the PMS7003 sensor
 * This code was written exclusively by MECH 45X Team 26
 */

#include <stdint.h>
#include "WProgram.h"
#include "Time.h"

#define LIB_PM_H
#define FIRST_BYTE 0x42
#define SECOND_BYTE 0x4D
#define SENSOR_OUTPUT_PIN A0
#define MAX_FRAME_LENGTH 64

#define START_TIME 6000
#define SAMPLING_TIME 280
#define SLEEP_TIME 912
#define MAX_READ_COUNT 5
#define MAX_FRAME_SYNC_COUNT 40
#define PMS_START_UP_TIME 120
#define MAX_FUNCTION_CALL_COUNT 1

class PM_7003 {
public:
    PM_7003();
    virtual ~PM_7003();
    int get_pm_ave(void);
    void set_transistor(int ground_pin, int tx_pin);
    bool make_sensor_read(void);
    void calibrate_sensor(void);
    void reset_pm_ave(void);

private:
    int current_byte;
    bool sync_state;
    char print_buffer[256];
    uint16_t byte_sum;
    int drain;
    uint16_t current_data;
    int pm_ground_control;
    int pm_tx_control;
    char frame_buffer[MAX_FRAME_LENGTH];
    int frame_count;
    int frame_length;

    bool debug = false;

    int pm_avgpm2_5;
    int pm2_5_buf[MAX_READ_COUNT];

    bool done_reading;
    int read_count;
    int function_call_count;
    int frame_sync_count;
    bool first_time;

    bool run_PM_sensor(void);
    void drain_serial(void);
    void frame_sync(void);
    void read_sensor(void);
    void data_switch(uint16_t current_data);
    void print_messages(void);

    //time
    void begin_timer(void);
    bool check_begin_reading(void);
    time_t start_time;
    time_t current_time;
    time_t duration;


    struct PMS7003data {
        uint8_t  start_frame[2];
        uint16_t frame_length;
        uint16_t concPM1_0_factory;
        uint16_t concPM2_5_factory;
        uint16_t concPM10_0_factory;
        uint16_t concPM1_0_ambient;
        uint16_t concPM2_5_ambient;
        uint16_t concPM10_0_ambient;
        uint16_t countPM0_3um;
        uint16_t countPM0_5um;
        uint16_t countPM1_0um;
        uint16_t countPM2_5um;
        uint16_t countPM5_0um;
        uint16_t countPM10_0um;
        uint8_t  version;
        uint8_t  error;
        uint16_t checksum;
    } packetdata;
};

```

```
/*
 * This is the .cpp file for the PMS7003 sensor
 * This code was written exclusively by MECH 45X Team 26
 */
#include "PM.h"

PM_7003::PM_7003() {
    current_byte = 0;
    packetdata.frame_length = MAX_FRAME_LENGTH;
    frame_length = MAX_FRAME_LENGTH;
}

PM_7003::~PM_7003() {
}

int PM_7003::getpm(void) {
    return pm_avgpm2_5;
}

bool PM_7003::run_PM_sensor(void) {
    /*
     * run the PM sensor
     * Start serial connection
     *
     * drain_serial() and read_sensor() until enough values have been read
     * to take the average
     */
    Serial1.begin(9600);
    read_count = 1;
    done_reading = false;
    frame_sync_count = 0;
    pm_avgpm2_5 = 0;
    while(!done_reading && frame_sync_count < MAX_FRAME_SYNC_COUNT) {
        drain_serial();
        delay(500);
        read_sensor();
    }

    Serial1.end();

    if(done_reading) {
        Serial.println("---------------------------");
        Serial.println("Done reading from PM sensor");
        Serial.println("---------------------------");
        Serial.println(" ");
        return true;
    }
    else if(!done_reading && frame_sync_count >= MAX_FRAME_SYNC_COUNT){return false;}
}

void PM_7003::drain_serial(void) {
    /*
    * Drains serial buffer if there are more than 32 entries
    * Reads entries to drain serial buffer
    */
    if (Serial1.available() > 32) {
        drain = Serial1.available();
        Serial.println("-- Draining buffer: ");
        Serial.println(Serial1.available(), DEC);
        for (int drain_index = drain; drain_index > 0; drain_index--) {Serial1.read();}
    }
}

void PM_7003::frame_sync(void) {
  /*
   * syncs frames for PM sensor
   * checks that frames are being read in correct order
   * exits when it confirms that frames are being read correctly
  */
```

```
    sync_state = false;
    frame_count = 0;
    byte_sum = 0;

    while (!sync_state && frame_sync_count < MAX_FRAME_SYNC_COUNT){
        current_byte = Serial1.read();

        if(current_byte == FIRST_BYTE && frame_count == 0) {
            frame_buffer[frame_count] = current_byte;
            packetdata.start_frame[0] = current_byte;
            byte_sum = current_byte;
            frame_count = 1;
        }
        else if(current_byte == SECOND_BYTE && frame_count == 1){
            frame_buffer[frame_count] = current_byte;
            packetdata.start_frame[1] = current_byte;
            byte_sum = byte_sum + current_byte;
            frame_count = 2;
            sync_state = true;
        }
        else{
            frame_sync_count++;
            Serial.println("frame is syncing");
            Serial.print("Current character: ");
            Serial.println(current_byte, HEX);
            Serial.print("frame count: ");
            Serial.println(frame_sync_count);
            delay(500);

            if(frame_sync_count >= MAX_FRAME_SYNC_COUNT) {
                Serial.println("------------------------");
                Serial.println("Max frame count exceeded");
                Serial.println("------------------------");
            }

        }
    }
}

void PM_7003::read_sensor(void) {
    /*
     * Sync the frames
     * read bytes and fill frame_buffer
     * use data_switch to calculate different parameters
     * print_messages once all values have been read.
     * done_reading = true if enough values have been read
     */
    frame_sync();

    while(sync_state == true && Serial1.available() > 0) {
        current_byte = Serial1.read();
        frame_buffer[frame_count] = current_byte;
        byte_sum = byte_sum + current_byte;
        frame_count++;
        uint16_t current_data = frame_buffer[frame_count-1]+(frame_buffer[frame_count-2]<<8);
        data_switch(current_data);

        if (frame_count >= frame_length && read_count <= MAX_READ_COUNT) {
            print_messages();
            pm_avgpm2_5 = pm_avgpm2_5 + pm2_5;
            read_count++;
            break;
        }
    }

    if (read_count > MAX_READ_COUNT) {
        pm_avgpm2_5 = exp((pm_avgpm2_5/MAX_READ_COUNT + 109314)/15990)*10000;
        done_reading = true;

    }
}

void PM_7003::data_switch(uint16_t current_data) {
    /*
     * data_switch uses current data and frame_count
     * to assign values to parameters
     */
    switch (frame_count) {
    case 4:
        packetdata.frame_length = current_data;
        frame_length = current_data + frame_count;
        break;
    case 6:
        packetdata.concPM1_0_factory = current_data;
        break;
    case 8:
        packetdata.concPM2_5_factory = current_data;
        break;
    case 10:
        packetdata.concPM10_0_factory = current_data;
        break;
    case 12:
        packetdata.concPM1_0_ambient = current_data;
        break;
    case 14:
        packetdata.concPM2_5_ambient = current_data;
        break;
    case 16:
        packetdata.concPM10_0_ambient = current_data;
        break;
    case 18:
        packetdata.countPM0_3um = current_data;
        break;
    case 20:
        packetdata.countPM0_5um = current_data;
        break;
    case 22:
        packetdata.countPM1_0um = current_data;
        break;
    case 24:
        packetdata.countPM2_5um = current_data;
        break;
    case 26:
        packetdata.countPM5_0um = current_data;
        break;
    case 28:
        packetdata.countPM10_0um = current_data;
        break;
    case 29:
        current_data = frame_buffer[frame_count-1];
        packetdata.version = current_data;
      break;
    case 30:
        current_data = frame_buffer[frame_count-1];
        packetdata.error = current_data;
        break;
    case 32:
        packetdata.checksum = current_data;
        byte_sum -= ((current_data>>8)+(current_data&0xFF));
        break;
    default:
        break;
    }
}

void PM_7003::print_messages(void){
    /*
```

```
     * Print messages to string and Serial screen
     */
    Serial.println("-----------------------");
    Serial.print("PMS 7003 - Reading #");
    Serial.println(read_count);
    Serial.println("-----------------------");
    sprintf(print_buffer, ", %02x, %02x, %04x, ",
        packetdata.start_frame[0], packetdata.start_frame[1], packetdata.frame_length);
    sprintf(print_buffer, "%s%04d, %04d, %04d, ", print_buffer,
        packetdata.concPM1_0_factory, packetdata.concPM2_5_factory, packetdata.concPM10_0_factory);
    sprintf(print_buffer, "%s%04d, %04d, %04d, ", print_buffer,
        packetdata.concPM1_0_ambient, packetdata.concPM2_5_ambient, packetdata.concPM10_0_ambient);
    sprintf(print_buffer, "%s%04d, %04d, %04d, %04d, %04d, %04d, ", print_buffer,
        packetdata.countPM0_3um, packetdata.countPM0_5um, packetdata.countPM1_0um,
        packetdata.countPM2_5um, packetdata.countPM5_0um, packetdata.countPM10_0um);
    sprintf(print_buffer, "%s%02d, %02d, ", print_buffer,
        packetdata.version, packetdata.error);

    pm2_5 = packetdata.countPM1_0um - packetdata.countPM2_5um + packetdata.countPM0_5um - packetdata.countPM1_0um + packetdata.countPM0_3um - packetdata.countPM0_5um;
    Serial.println(print_buffer);
    Serial.println("-----------------------");
    delay(500);
}
```

The code for running the MH-Z19 CO2 sensor is:

`’MHZ19.cpp’ and ’MHZ19.h’`

This code reads from the CO2 sensor several time and takes an average value of all of the readings. The MH-Z19 communicates using a UART connection. This code was written entirely by Team 26. The .h file is presented first, followed by the .cpp file.

```
/*
 * This is the .cpp file for the MH-Z19 CO2 Sensor
 * This code was exclusively written by MECH 45X Team 26
 */

#ifndef MHZ19_H
#define MHZ19_H
#define MHZ19_ZEROTH_BYTE 0xFF
#define MHZ19_FIRST_BYTE 0x86
#define MAX_FRAME_LEN 9
#define NUMBER_OF_VALUES 5
#define CO2_START_UP_TIME 210
#define MAX_FRAME_READ_COUNT 40
#define MAX_FUNCTION_CALL_COUNT 1
#include "WProgram.h"
#include "Time.h"


class MHZ19 {
    public:
        MHZ19();
        virtual ~MHZ19();
        int get_co2_reading(void);
        int get_co2_ave(void);
        void set_transistor(int pin);
        bool make_sensor_read(void);
        void calibrate_sensor(void);
        void reset_co2_ave(void);

    private:
        char frame_buffer[MAX_FRAME_LEN];
        const uint8_t mhz19_read_command[MAX_FRAME_LEN] = {0xFF,0x01,0x86,0x00,0x00,0x00,0x00,0x00,0x79};;

        bool debug = false;

        bool sync_state;
        bool does_sensor_work;
        bool is_average_taken;
        bool first_time;
        int co2_transistor_control;

        int frame_sync_count;
        int frame_read_count;
        int byte_sum;
        int current_byte;
        int drain;
        int co2_ppm;
        int co2_ppm_average;
        int reading_count;
        int function_call_count;
        int mhz19_buffer[NUMBER_OF_VALUES];

        bool run_sensor(void);
        void frame_sync(void);
        void read_sensor(void);
        void serial_drain(void);
        void fill_frame_buffer(void);
        void add_to_ave_buf(void);
        void print_current_reading(void);
        void calculate_average_reading(void);
        void print_average_reading(void);
        void take_average(void);

        //Timer
        time_t start_time;
        time_t current_time;
        time_t duration;

        void begin_timer(void);
        bool check_begin_reading(void);
};

#endif /* MHZ19_H_ */
```

```cpp
/*
 * This is the .cpp file for the MH-Z19 CO2 Sensor
 * This code was exclusively written by MECH 45X Team 26
 */

#include "MHZ19.h"
#include "Time.h"

MHZ19::MHZ19() {
    first_time = true;
}

MHZ19::~MHZ19() {
}

void MHZ19::set_transistor(int pin) {
    /*
     * Set transistor pin
     * set pinMode for transistor pin
     */
    co2_transistor_control = pin;
    pinMode(co2_transistor_control,OUTPUT);
}

void MHZ19::begin_timer(void) {
    /*
     * Turn transistor on
     * Save time at which transistor is turned on
     * Time is used for timing purposes
     * change first_time to false
     *
     * first_time indicates whether or not timer has been started
     * and transistor has been turned on
     */
    co2_ppm_average = -1;
    digitalWrite(co2_transistor_control, HIGH);
    start_time = now();
    Serial.println("--------------");
    Serial.print("CO2 start time: ");
    Serial.println(start_time);
    Serial.println("--------------");
    first_time = false;
}

bool MHZ19::check_begin_reading(void) {
    /*
     * Check whether enough time has passed to begin reading
     * return true if enough time has passed
     * else false
     */
    current_time = now();
    duration = current_time - start_time;
    Serial.println("-----------------");
    Serial.print("CO2 Duration: ");
    Serial.println(duration);
    Serial.println("-----------------");

    if(duration >= CO2_START_UP_TIME) {
        Serial.println("Three minutes have elapsed since starting CO2 sensor!");
        return(true);
    } else{return(false);}
}

bool MHZ19::make_sensor_read(void) {
    /*
     * turn transistor on and start timer if this hasn't already been done
     * read from sensor if enough time has passed
     * return true if enough measurements have been taken
     * else false
     */
    if(first_time) {
        function_call_count = 0;
        begin_timer();
        return(false);
    }
    else if(function_call_count < MAX_FUNCTION_CALL_COUNT) {
        if(check_begin_reading()) {
            Serial.println("---------------------");
            Serial.print("Function Call Count: ");
            Serial.println(function_call_count);
            Serial.println("---------------------");
            run_sensor();
            function_call_count ++;
        } else {return(false);}
    }


    if(function_call_count >= MAX_FUNCTION_CALL_COUNT) {
        first_time = true;
        digitalWrite(co2_transistor_control, LOW);
        return(true);
    } else{return(false);}
}

void MHZ19::calibrate_sensor(void) {
    /*
     * Turn sensor on and wait for warm-upper_bound
     * Following warm-up, read forever
     */
    if(first_time) {
        function_call_count = 0;
        begin_timer();
    }

    if(check_begin_reading()) {
        Serial.println("---------------------");
        Serial.print("Function Call Count: ");
        Serial.println(function_call_count);
        Serial.println("---------------------");
        run_sensor();
        function_call_count ++;
    }
}

bool MHZ19::run_sensor(void) {
    /*
     * Run the MHZ19 sensor
     * Set ppm to zero
     * clear the frame_buffer
     * drain the serial buffer
     * read from the sensor
     * print reading
     * add the reading to the average value buffer
     * calculate average value
     */
    co2_ppm = -1;
    co2_ppm_average = 0;
    is_average_taken = false;
    does_sensor_work = true;
    reading_count = 1;

    serial_drain();

    while(is_average_taken == false && does_sensor_work == true) {
        memset(frame_buffer, 0, 9);
        read_sensor();
        print_current_reading();
        add_to_ave_buf();
        calculate_average_reading();
        print_average_reading();
        co2_ppm = -1;
    }
    if(is_average_taken == true) {return(true);}
    else {return(false);}
}

void MHZ19::print_current_reading(void) {
    /*
     * Prints current reading if reading is valid (i.e. co2_ppm > 0)
     * and if the maximum number of readings haven't been exceeded
     */
    if(co2_ppm > 0) {
      Serial.print("MHZ19 CO2 PPM Reading ");
      Serial.print(reading_count);
      Serial.print(": ");
      Serial.println(co2_ppm);
    }
    else {
      Serial.println("Error reading CO2 PPM from MHZ19");
    }
}

void MHZ19::add_to_ave_buf(void) {
    /*
     * IF a valid value of co2 is read and the number of reading is less than the max,
     * THEN add current value to buffer
     */
    if(co2_ppm > 0 && reading_count <= NUMBER_OF_VALUES) {
        mhz19_buffer[reading_count - 1] = co2_ppm;
        reading_count += 1;
    }
}

void MHZ19::calculate_average_reading(void) {
    /*
     * IF the number of readings exceeds the number of values to be read,
     * THEN calculate the average
     */
    if(reading_count > NUMBER_OF_VALUES) {
        for(int k = 0; k < NUMBER_OF_VALUES; k++) {co2_ppm_average += mhz19_buffer[k];}

        co2_ppm_average = co2_ppm_average / ( NUMBER_OF_VALUES );

        is_average_taken = true;
    }
}

void MHZ19::print_average_reading(void) {
    /*
     * IF the average has been taken (co2_ppm_average > 0)
     * THEN print the average
     */
    if(co2_ppm_average > 0) {
        Serial.println("-----------------------------");
        Serial.print("CO2 PPM Average Reading: ");
        Serial.println(co2_ppm_average);
        Serial.println("-----------------------------");
    }
}

void MHZ19::read_sensor(void) {
    /*
     *  Start Serial1 connection
     *  Send command to read from sensor to the sensor
     *  Read from the sensor (fill_from_buffer();)
     *  Calculate PPM for CO2
     *  End Serial connection
    */

    Serial1.begin(9600);
    Serial1.write(mhz19_read_command, 9);
    delay(1000);
    fill_frame_buffer();
    co2_ppm = 256*frame_buffer[2] + frame_buffer[3];
    Serial1.end();
}

void MHZ19::serial_drain(void) {
  /*
   * Drains serial buffer when sensor is turned on
   */
    while (Serial1.available() > 0) {
        drain = Serial1.available();
        Serial.print("-- Draining buffer: ");
    }
}

void MHZ19::frame_sync(void) {
    /*
     * Sync frames so that frames are added to the frame_buffer in the correct order
     * IF correct byte is read, THEN add to buffer and move on to next byte
     * ELSE read byte and discard
     * IF no bytes are available to read and the frames have not been synced, THEN send command to read from sensor again
     *
     * frame_sync_count keeps track of how many frames are added to frame_buffer
     * frame_read_count keeps track of how many frames are read but not added to buffer (fails if too many frames read)
     */
    sync_state = false;
    frame_sync_count = 0;
    frame_read_count = 0;
    byte_sum = 0;

    while (!sync_state && Serial1.available() > 0 && frame_read_count < MAX_FRAME_READ_COUNT) {
        current_byte = Serial1.read();

        if (current_byte == MHZ19_ZEROTH_BYTE && frame_sync_count == 0) {
            frame_buffer[frame_sync_count] = current_byte;
            byte_sum = current_byte;
            frame_sync_count = 1;
        }
        else if (current_byte == MHZ19_FIRST_BYTE && frame_sync_count == 1) {
            frame_buffer[frame_sync_count] = current_byte;
            byte_sum += current_byte;
            sync_state = true;
            frame_sync_count = 2;
        }
        else {
            if(debug) {
                Serial.print("-- Frame syncing... ");
                Serial.println(current_byte, HEX);
            }

            frame_read_count ++;
        }

        if (!sync_state && !(Serial1.available() > 0) && frame_read_count < MAX_FRAME_READ_COUNT) {
            Serial1.write(mhz19_read_command, 9);

            if(debug) {
                Serial.println("----------------------------------------");
                Serial.println("Read command has been sent to CO2 sensor");
                Serial.println("----------------------------------------");
            }

            delay(500);
        }
    }
}

void MHZ19::fill_frame_buffer(void) {
  /*
   * Sync frames
   * Read byte into frame_buffer
   */
    frame_sync();

    while(sync_state && Serial1.available() > 0 && frame_sync_count < MAX_FRAME_LEN) {
        current_byte = Serial1.read();
        frame_buffer[frame_sync_count] = current_byte;
        byte_sum += current_byte;
        frame_sync_count++;
    }
}

// getter and setter functions
int MHZ19::get_co2_ave(void) {
    return co2_ppm_average;
}

int MHZ19::get_co2_reading(void) {
    return co2_ppm;
}

void MHZ19::reset_co2_ave(void) {
    co2_ppm_average = -1;
}

```

The code for running the CCS821 VOC sensor is:

`’ccs821.cpp’ and ’ccs821.h’`

This code reads from the VOC sensor several time and takes an average value of all of the readings. The CCS821 communicates using an I2C connection. This code was retrieved from:

`https://learn.adafruit.com/adafruit-ccs811-air-quality-sensor`
`/arduino-wiring-test`

The on line library was supplemented by additional methods added by Team 26. The .h file is presented first, followed by the .cpp file.

```
/*
 * This is the .h file for the ccs821 VOC sensor
 */
#ifndef LIB_ADAFRUIT_CCS811_H
#define LIB_ADAFRUIT_CCS811_H

#if (ARDUINO >= 100)
 #include "Arduino.h"
#else
 #include "WProgram.h"
#endif

#include <Wire.h>

/*=========================================================================
    I2C ADDRESS/BITS
    -----------------------------------------------------------------------*/
    #define CCS811_ADDRESS                (0x5A)
/*=========================================================================*/

    #define MAX_READ_COUNT 5
    #define MAX_ERROR_COUNT 5

/*=========================================================================
    REGISTERS
    -----------------------------------------------------------------------*/
    enum
    {
        CCS811_STATUS = 0x00,
        CCS811_MEAS_MODE = 0x01,
        CCS811_ALG_RESULT_DATA = 0x02,
        CCS811_RAW_DATA = 0x03,
        CCS811_ENV_DATA = 0x05,
        CCS811_NTC = 0x06,
        CCS811_THRESHOLDS = 0x10,
        CCS811_BASELINE = 0x11,
        CCS811_HW_ID = 0x20,
        CCS811_HW_VERSION = 0x21,
        CCS811_FW_BOOT_VERSION = 0x23,
        CCS811_FW_APP_VERSION = 0x24,
        CCS811_ERROR_ID = 0xE0,
        CCS811_SW_RESET = 0xFF,
    };

  //bootloader registers
  enum
  {
    CCS811_BOOTLOADER_APP_ERASE = 0xF1,
    CCS811_BOOTLOADER_APP_DATA = 0xF2,
    CCS811_BOOTLOADER_APP_VERIFY = 0xF3,
    CCS811_BOOTLOADER_APP_START = 0xF4
  };

  enum
  {
    CCS811_DRIVE_MODE_IDLE = 0x00,
    CCS811_DRIVE_MODE_1SEC = 0x01,
    CCS811_DRIVE_MODE_10SEC = 0x02,
    CCS811_DRIVE_MODE_60SEC = 0x03,
    CCS811_DRIVE_MODE_250MS = 0x04,
  };

/*=========================================================================*/

#define CCS811_HW_ID_CODE     0x81

#define CCS811_REF_RESISTOR     100000

/**************************************************************************/
```

```
/*!
    @brief  Class that stores state and functions for interacting with CCS811 gas sensor chips
*/
/***************************************************************************/
class Adafruit_CCS811 {
  public:
    //constructors
    Adafruit_CCS811(void) {};
    ~Adafruit_CCS811(void) {};

    bool start_voc(void);
    bool run_voc(void);
    float get_eCO2_ave(void);
        float get_TVOC_ave(void);

    bool begin(uint8_t addr = CCS811_ADDRESS);

    void setEnvironmentalData(uint8_t humidity, double temperature);

    //calculate temperature based on the NTC register
    double calculateTemperature();

    void setThresholds(uint16_t low_med, uint16_t med_high, uint8_t hysteresis = 50);

    void SWReset();

    void setDriveMode(uint8_t mode);
    void enableInterrupt();
    void disableInterrupt();

        /***************************************************************************/
        /*!
            @brief  returns the stored total volatile organic compounds measurement. This does does not read the sensor. To do so, call readData()
            @returns TVOC measurement as 16 bit integer
        */
        /***************************************************************************/
    uint16_t getTVOC() { return _TVOC; }

        /***************************************************************************/
        /*!
            @brief  returns the stored estimated carbon dioxide measurement. This does does not read the sensor. To do so, call readData()
            @returns eCO2 measurement as 16 bit integer
        */
        /***************************************************************************/
    uint16_t geteCO2() { return _eCO2; }

        /***************************************************************************/
        /*!
            @brief  set the temperature compensation offset for the device. This is needed to offset errors in NTC measurements.
            @param offset the offset to be added to temperature measurements.
        */
        /***************************************************************************/
    void setTempOffset(float offset) { _tempOffset = offset; }

    //check if data is available to be read
    bool available();
    uint8_t readData();

    bool checkError();

  private:
    float eCO2_buf[MAX_READ_COUNT];
    float TVOC_buf[MAX_READ_COUNT];
    float eCO2_ave;
    float TVOC_ave;
    void read_voc(void);
    void fill_buffer(void);
    void print_readings(void);
    void calculate_average_reading(void);
    void print_average_reading(void);
    int read_count;
    int error_count;
    bool is_average_taken;

    uint8_t _i2caddr;
    float _tempOffset;

    uint16_t _TVOC;
    uint16_t _eCO2;

    void      write8(byte reg, byte value);
    void      write16(byte reg, uint16_t value);
        uint8_t   read8(byte reg);

    void read(uint8_t reg, uint8_t *buf, uint8_t num);
    void write(uint8_t reg, uint8_t *buf, uint8_t num);
    void _i2c_init();

/*=========================================================================
  REGISTER BITFIELDS
    -----------------------------------------------------------------------*/
    // The status register
        struct status {

            /* 0: no error
            *  1: error has occurred
            */
            uint8_t ERROR: 1;

            // reserved : 2

            /* 0: no samples are ready
            *  1: samples are ready
            */
            uint8_t DATA_READY: 1;
            uint8_t APP_VALID: 1;

      // reserved : 2

            /* 0: boot mode, new firmware can be loaded
            *  1: application mode, can take measurements
            */
            uint8_t FW_MODE: 1;

            void set(uint8_t data){
              ERROR = data & 0x01;
              DATA_READY = (data >> 3) & 0x01;
              APP_VALID = (data >> 4) & 0x01;
              FW_MODE = (data >> 7) & 0x01;
            }
        };
        status _status;

        //measurement and conditions register
        struct meas_mode {
          // reserved : 2

          /* 0: interrupt mode operates normally
            *  1: Interrupt mode (if enabled) only asserts the nINT signal (driven low) if the new
        ALG_RESULT_DATA crosses one of the thresholds set in the THRESHOLDS register
        by more than the hysteresis value (also in the THRESHOLDS register)
            */
          uint8_t INT_THRESH: 1;
```

```

     /* 0: int disabled
      *  1: The nINT signal is asserted (driven low) when a new sample is ready in
    ALG_RESULT_DATA. The nINT signal will stop being driven low when
    ALG_RESULT_DATA is read on the I²C interface.
      */
     uint8_t INT_DATARDY: 1;

     uint8_t DRIVE_MODE: 3;

     uint8_t get(){
      return (INT_THRESH << 2) | (INT_DATARDY << 3) | (DRIVE_MODE << 4);
     }
    };
    meas_mode _meas_mode;

    struct error_id {
     /* The CCS811 received an I²C write request addressed to this station but with
   invalid register address ID */
     uint8_t WRITE_REG_INVALID: 1;

     /* The CCS811 received an I²C read request to a mailbox ID that is invalid */
     uint8_t READ_REG_INVALID: 1;

     /* The CCS811 received an I²C request to write an unsupported mode to
   MEAS_MODE */
     uint8_t MEASMODE_INVALID: 1;

     /* The sensor resistance measurement has reached or exceeded the maximum
   range */
     uint8_t MAX_RESISTANCE: 1;

     /* The Heater current in the CCS811 is not in range */
     uint8_t HEATER_FAULT: 1;

     /*  The Heater voltage is not being applied correctly */
     uint8_t HEATER_SUPPLY: 1;

     void set(uint8_t data){
      WRITE_REG_INVALID = data & 0x01;
      READ_REG_INVALID = (data & 0x02) >> 1;
      MEASMODE_INVALID = (data & 0x04) >> 2;
      MAX_RESISTANCE = (data & 0x08) >> 3;
      HEATER_FAULT = (data & 0x10) >> 4;
      HEATER_SUPPLY = (data & 0x20) >> 5;
     }
    };
    error_id _error_id;

/*=========================================================================*/
};

#endif

```

```
/*
 * This is the .cpp file for the ccs821 VOC sensor
 * The library for this sensor was retrieved on line:
 * https://learn.adafruit.com/adafruit-ccs811-air-quality-sensor/arduino-wiring-test
 * MECH 45X Team 26 did not write Part 1, the on line library
 *
 * Therefore Part 1 is not properly commented because the
 * the team does not understand the code.
 *
 * Part 2 was written by Team 26 and is properly commented.
 *
 * Part 1 begins...
 */

 #include "CCS821.h"

/**************************************************************************/
/*!
    @brief  Setups the I2C interface and hardware and checks for communication.
    @param  addr Optional I2C address the sensor can be found on. Default is 0x5A
    @returns True if device is set up, false on any failure
*/
/**************************************************************************/
bool Adafruit_CCS811::begin(uint8_t addr)
{
  _i2caddr = addr;

  _i2c_init();

  SWReset();
  delay(100);

  //check that the HW id is correct
  if(this->read8(CCS811_HW_ID) != CCS811_HW_ID_CODE)
    return false;

  //try to start the app
  this->write(CCS811_BOOTLOADER_APP_START, NULL, 0);
  delay(100);

  //make sure there are no errors and we have entered application mode
  if(checkError()) return false;
  if(!_status.FW_MODE) return false;

  disableInterrupt();

  //default to read every second
  setDriveMode(CCS811_DRIVE_MODE_1SEC);

  return true;
}

/**************************************************************************/
/*!
    @brief  sample rate of the sensor.
    @param  mode one of CCS811_DRIVE_MODE_IDLE, CCS811_DRIVE_MODE_1SEC,
    CCS811_DRIVE_MODE_10SEC, CCS811_DRIVE_MODE_60SEC, CCS811_DRIVE_MODE_250MS.
*/
void Adafruit_CCS811::setDriveMode(uint8_t mode)
{
  _meas_mode.DRIVE_MODE = mode;
  this->write8(CCS811_MEAS_MODE, _meas_mode.get());
}

/**************************************************************************/
/*!
    @brief  enable the data ready interrupt pin on the device.
*/
/**************************************************************************/
```

```
void Adafruit_CCS811::enableInterrupt()
{
  _meas_mode.INT_DATARDY = 1;
  this->write8(CCS811_MEAS_MODE, _meas_mode.get());
}

/**************************************************************************/
/*!
    @brief  disable the data ready interrupt pin on the device
*/
/**************************************************************************/
void Adafruit_CCS811::disableInterrupt()
{
  _meas_mode.INT_DATARDY = 0;
  this->write8(CCS811_MEAS_MODE, _meas_mode.get());
}

/**************************************************************************/
/*!
    @brief  checks if data is available to be read.
    @returns True if data is ready, false otherwise.
*/
/**************************************************************************/
bool Adafruit_CCS811::available()
{
  _status.set(read8(CCS811_STATUS));
  if(!_status.DATA_READY)
    return false;
  else return true;
}

/**************************************************************************/
/*!
    @brief  read and store the sensor data. This data can be accessed with getTVOC()
    and geteCO2()
    @returns 0 if no error, error code otherwise.
*/
/**************************************************************************/
uint8_t Adafruit_CCS811::readData()
{
  if(!available())
    return false;
  else{
    uint8_t buf[8];
    this->read(CCS811_ALG_RESULT_DATA, buf, 8);

    _eCO2 = ((uint16_t)buf[0] << 8) | ((uint16_t)buf[1]);
    _TVOC = ((uint16_t)buf[2] << 8) | ((uint16_t)buf[3]);

    if(_status.ERROR)
      return buf[5];

    else return 0;
  }
}

/**************************************************************************/
/*!
    @brief  set the humidity and temperature compensation for the sensor.
    @param humidity the humidity data as a percentage. For 55% humidity, pass in
    integer 55.
    @param temperature the temperature in degrees C as a decimal number. For 25.5
    degrees C, pass in 25.5
*/
/**************************************************************************/
void Adafruit_CCS811::setEnvironmentalData(uint8_t humidity, double temperature)
{
  /* Humidity is stored as an unsigned 16 bits in 1/512%RH. The
  default value is 50% = 0x64, 0x00. As an example 48.5%
  humidity would be 0x61, 0x00.*/

  /* Temperature is stored as an unsigned 16 bits integer in 1/512
  degrees; there is an offset: 0 maps to -25□C. The default value is
  25□C = 0x64, 0x00. As an example 23.5% temperature would be
  0x61, 0x00.
  The internal algorithm uses these values (or default values if
  not set by the application) to compensate for changes in
  relative humidity and ambient temperature.*/

  uint8_t hum_perc = humidity << 1;

  float fractional = modf(temperature, &temperature);
  uint16_t temp_high = (((uint16_t)temperature + 25) << 9);
  uint16_t temp_low = ((uint16_t)(fractional / 0.001953125) & 0x1FF);

  uint16_t temp_conv = (temp_high | temp_low);

  uint8_t buf[] = {hum_perc, 0x00,
    (uint8_t)((temp_conv >> 8) & 0xFF), (uint8_t)(temp_conv & 0xFF)};

  this->write(CCS811_ENV_DATA, buf, 4);

}

/**************************************************************************/
/*!
    @brief  calculate the temperature using the onboard NTC resistor.
    @returns temperature as a double.
*/
/**************************************************************************/
double Adafruit_CCS811::calculateTemperature()
{
  uint8_t buf[4];
  this->read(CCS811_NTC, buf, 4);

  uint32_t vref = ((uint32_t)buf[0] << 8) | buf[1];
  uint32_t vntc = ((uint32_t)buf[2] << 8) | buf[3];

  //from ams ccs811 app note
  uint32_t rntc = vntc * CCS811_REF_RESISTOR / vref;

  double ntc_temp;
  ntc_temp = log((double)rntc / CCS811_REF_RESISTOR); // 1
  ntc_temp /= 3380; // 2
  ntc_temp += 1.0 / (25 + 273.15); // 3
  ntc_temp = 1.0 / ntc_temp; // 4
  ntc_temp -= 273.15; // 5
  return ntc_temp - _tempOffset;

}

/**************************************************************************/
/*!
    @brief  set interrupt thresholds
    @param low_med the level below which an interrupt will be triggered.
    @param med_high the level above which the interrupt will ge triggered.
    @param hysteresis optional histeresis level. Defaults to 50
*/
/**************************************************************************/
void Adafruit_CCS811::setThresholds(uint16_t low_med, uint16_t med_high, uint8_t hysteresis)
{
  uint8_t buf[] = {(uint8_t)((low_med >> 8) & 0xF), (uint8_t)(low_med & 0xF),
  (uint8_t)((med_high >> 8) & 0xF), (uint8_t)(med_high & 0xF), hysteresis};

  this->write(CCS811_THRESHOLDS, buf, 5);
}

```

```
/**************************************************************************/
/*!
    @brief  trigger a software reset of the device
*/
/**************************************************************************/
void Adafruit_CCS811::SWReset()
{
  //reset sequence from the datasheet
  uint8_t seq[] = {0x11, 0xE5, 0x72, 0x8A};
  this->write(CCS811_SW_RESET, seq, 4);
}

/**************************************************************************/
/*!
    @brief   read the status register and store any errors.
    @returns the error bits the status register of the device.
*/
/**************************************************************************/
bool Adafruit_CCS811::checkError()
{
  _status.set(read8(CCS811_STATUS));
  return _status.ERROR;
}

/**************************************************************************/
/*!
    @brief  write one byte of data to the specified register
    @param  reg the register to write to
    @param  value the value to write
*/
/**************************************************************************/
void Adafruit_CCS811::write8(byte reg, byte value)
{
  this->write(reg, &value, 1);
}

/**************************************************************************/
/*!
    @brief  read one byte of data from the specified register
    @param  reg the register to read
    @returns one byte of register data
*/
/**************************************************************************/
uint8_t Adafruit_CCS811::read8(byte reg)
{
  uint8_t ret;
  this->read(reg, &ret, 1);

  return ret;
}

void Adafruit_CCS811::_i2c_init()
{
  Wire.begin();
}

void Adafruit_CCS811::read(uint8_t reg, uint8_t *buf, uint8_t num)
{
  uint8_t value;
  uint8_t pos = 0;

  //on arduino we need to read in 32 byte chunks
  while(pos < num){

    uint8_t read_now = min((uint8_t)32, (uint8_t)(num - pos));
    Wire.beginTransmission((uint8_t)_i2caddr);
    Wire.write((uint8_t)reg + pos);
    Wire.endTransmission();
    Wire.requestFrom((uint8_t)_i2caddr, read_now);
    for(int i=0; i<read_now; i++){
      buf[pos] = Wire.read();
      pos++;
    }
  }
}

void Adafruit_CCS811::write(uint8_t reg, uint8_t *buf, uint8_t num)
{
  Wire.beginTransmission((uint8_t)_i2caddr);
  Wire.write((uint8_t)reg);
  Wire.write((uint8_t *)buf, num);
  Wire.endTransmission();
}

/*
 * Part 2: code written by team 26
 * This code was written by Team 26
 * This code is properly commented
 */

bool Adafruit_CCS811::start_voc(void) {
    /*
     * Start voc sensor using the library's begin() function
     * If sensor is started, calibrate temperature
     */
    Serial.println("Trying to start VOC Sensor...");
    if(!begin()){
        Serial.println("Failed to start CC2821 VOC sensor! Wiring is likely incorrect.");
        return false;
    }
    else {
        Serial.println("Successfully started VOC Sensor!");
        delay(5000);
        return true;
    }
}

bool Adafruit_CCS811::run_voc(void) {
    /*
     * Run the VOC sensor
     * Take measurements until enough measurements have been taken to calculate the average
     * use read_voc() to read from sensor
     */
    is_average_taken = false;
    read_count = 1;
    error_count = 0;
    while(is_average_taken == false  && error_count < MAX_ERROR_COUNT) {read_voc();}

    if(is_average_taken) {return true;}
    else if(error_count >= MAX_ERROR_COUNT) {return false;}
}

void Adafruit_CCS811::read_voc(void) {
    /*
     * Read values from voc sensor
     * IF data is read and max read count has not been exceed
     * THEN fill_buffer and print_readings and read_count ++
     * calculate_average_reading
     * print_average_reading
     */
    if(available()){
        float temp = calculateTemperature();
        if(!readData() && read_count <= MAX_READ_COUNT){
            fill_buffer();
            print_readings();
            read_count += 1;
            error_count = 0;
        }
        else {
          error_count ++;
          Serial.print("ERROR #");
          Serial.println(error_count);
          delay(500);
        }
    }
    calculate_average_reading();
    print_average_reading();
}

void Adafruit_CCS811::fill_buffer(void) {
    /*
     * add new values to buffers
     */
    eCO2_buf[read_count-1] = geteCO2();
    TVOC_buf[read_count-1] = getTVOC();
}

void Adafruit_CCS811::print_readings(void) {
    /*
     * Print readings
     */
    Serial.print("VOC Reading #:");
    Serial.print(read_count);
    Serial.print(", CO2: ");
    Serial.print(geteCO2());
    Serial.print("ppm, TVOC: ");
    Serial.print(getTVOC());
    Serial.println("ppb");
}

void Adafruit_CCS811::calculate_average_reading(void) {
    /*
     * Calculate the average reading if enough readings have been taken
     */
    if(read_count > MAX_READ_COUNT) {
        eCO2_ave = 0;
        TVOC_ave = 0;
        for(int k = 0; k < MAX_READ_COUNT; k++) {
            eCO2_ave += eCO2_buf[k];
            TVOC_ave += TVOC_buf[k];
        }
        eCO2_ave = eCO2_ave / MAX_READ_COUNT;
        TVOC_ave = TVOC_ave / MAX_READ_COUNT;

        read_count = 1;
        is_average_taken = true;
    }
}

void Adafruit_CCS811::print_average_reading(void) {
    /*
    * print average reading values
    */
    if(is_average_taken) {
        Serial.println("-----------------------");
        Serial.println("VOC Sensor Average Readings:");
        Serial.println("-----------------------");
        Serial.print("CCS eCO2 Average: ");
        Serial.println(eCO2_ave);
        Serial.print("CCS TVOC Average: ");
        Serial.println(TVOC_ave);
    }
}

// Getter functions for VOC parameters
```

```
float Adafruit_CCS811::get_eCO2_ave(void) {
    return eCO2_ave;
}
float Adafruit_CCS811::get_TVOC_ave(void) {
    return TVOC_ave;
}

```

The code for running the SHT35D Temperature and Relative Humidity sensor is:

`’SHT35D.cpp’ and ’SHT35D.h’`

This code reads from the SHT35D sensor several time and takes an average value of all of the readings. The SHT35D communicates using an I2C connection. This code was retrieved from:

`https://github.com/closedcube/ClosedCube_SHT31D_Arduino`

The on line library was supplemented by additional methods added by Team 26. The .h file is presented first, followed by the .cpp file.

```
/*
 * .h file for SHT35D
 */

#ifndef SHT35D
#define SHT35D
#define MAX_READ_COUNT 5
#define MAX_ERROR_COUNT 5
#define ADDR_SHT 0x45

#include <Arduino.h>

//List of Commands for SHT35D Sensor:
typedef enum {
    SHT3XD_CMD_READ_SERIAL_NUMBER = 0x3780,

    SHT3XD_CMD_READ_STATUS = 0xF32D,
    SHT3XD_CMD_CLEAR_STATUS = 0x3041,

    SHT3XD_CMD_HEATER_ENABLE = 0x306D,
    SHT3XD_CMD_HEATER_DISABLE = 0x3066,

    SHT3XD_CMD_SOFT_RESET = 0x30A2,

    SHT3XD_CMD_CLOCK_STRETCH_H = 0x2C06,
    SHT3XD_CMD_CLOCK_STRETCH_M = 0x2C0D,
    SHT3XD_CMD_CLOCK_STRETCH_L = 0x2C10,

    SHT3XD_CMD_POLLING_H = 0x2400,
    SHT3XD_CMD_POLLING_M = 0x240B,
    SHT3XD_CMD_POLLING_L = 0x2416,

    SHT3XD_CMD_ART = 0x2B32,

    SHT3XD_CMD_PERIODIC_HALF_H = 0x2032,
    SHT3XD_CMD_PERIODIC_HALF_M = 0x2024,
    SHT3XD_CMD_PERIODIC_HALF_L = 0x202F,
    SHT3XD_CMD_PERIODIC_1_H = 0x2130,
    SHT3XD_CMD_PERIODIC_1_M = 0x2126,
    SHT3XD_CMD_PERIODIC_1_L = 0x212D,
    SHT3XD_CMD_PERIODIC_2_H = 0x2236,
    SHT3XD_CMD_PERIODIC_2_M = 0x2220,
    SHT3XD_CMD_PERIODIC_2_L = 0x222B,
    SHT3XD_CMD_PERIODIC_4_H = 0x2334,
    SHT3XD_CMD_PERIODIC_4_M = 0x2322,
    SHT3XD_CMD_PERIODIC_4_L = 0x2329,
    SHT3XD_CMD_PERIODIC_10_H = 0x2737,
    SHT3XD_CMD_PERIODIC_10_M = 0x2721,
    SHT3XD_CMD_PERIODIC_10_L = 0x272A,

    SHT3XD_CMD_FETCH_DATA = 0xE000,
    SHT3XD_CMD_STOP_PERIODIC = 0x3093,

    SHT3XD_CMD_READ_ALR_LIMIT_LS = 0xE102,
    SHT3XD_CMD_READ_ALR_LIMIT_LC = 0xE109,
    SHT3XD_CMD_READ_ALR_LIMIT_HS = 0xE11F,
    SHT3XD_CMD_READ_ALR_LIMIT_HC = 0xE114,
    SHT3XD_CMD_WRITE_ALR_LIMIT_HS = 0x611D,
    SHT3XD_CMD_WRITE_ALR_LIMIT_HC = 0x6116,
    SHT3XD_CMD_WRITE_ALR_LIMIT_LC = 0x610B,
    SHT3XD_CMD_WRITE_ALR_LIMIT_LS = 0x6100,

    SHT3XD_CMD_NO_SLEEP = 0x303E,
} SHT31D_Commands;

// List of repeatability options for SHT35D:
typedef enum {
  SHT3XD_REPEATABILITY_HIGH,
  SHT3XD_REPEATABILITY_MEDIUM,
  SHT3XD_REPEATABILITY_LOW,
} SHT31D_Repeatability;

// List of modes:
typedef enum {
  SHT3XD_MODE_CLOCK_STRETCH,
  SHT3XD_MODE_POLLING,
} SHT31D_Mode;

// List of frequency choices
typedef enum {
  SHT3XD_FREQUENCY_HZ5,
  SHT3XD_FREQUENCY_1HZ,
  SHT3XD_FREQUENCY_2HZ,
  SHT3XD_FREQUENCY_4HZ,
  SHT3XD_FREQUENCY_10HZ
} SHT31D_Frequency;

// List of errors:
typedef enum {
  SHT3XD_NO_ERROR = 0,

  SHT3XD_CRC_ERROR = -101,
  SHT3XD_TIMEOUT_ERROR = -102,

  SHT3XD_PARAM_WRONG_MODE = -501,
  SHT3XD_PARAM_WRONG_REPEATABILITY = -502,
  SHT3XD_PARAM_WRONG_FREQUENCY = -503,
  SHT3XD_PARAM_WRONG_ALERT = -504,

  // Wire I2C translated error codes

  SHT3XD_WIRE_I2C_DATA_TOO_LOG = -10,
  SHT3XD_WIRE_I2C_RECEIVED_NACK_ON_ADDRESS = -20,
  SHT3XD_WIRE_I2C_RECEIVED_NACK_ON_DATA = -30,
  SHT3XD_WIRE_I2C_UNKNOW_ERROR = -40
} SHT31D_ErrorCode;

// List of statuses:
typedef union {
    uint16_t rawData;
    struct {
        uint8_t WriteDataChecksumStatus : 1;
        uint8_t CommandStatus : 1;
        uint8_t Reserved0 : 2;
        uint8_t SystemResetDetected : 1;
        uint8_t Reserved1 : 5;
        uint8_t T_TrackingAlert : 1;
        uint8_t RH_TrackingAlert : 1;
        uint8_t Reserved2 : 1;
        uint8_t HeaterStatus : 1;
        uint8_t Reserved3 : 1;
        uint8_t AlertPending : 1;
    };
} SHT31D_RegisterStatus;

struct SHT31D {
    /*
     * Structure for SHT31D
     * t - temperature
     * rh - relative humidity
     * error - error of type SHT31D_ErrorCode
     */
    float t;
    float rh;
    SHT31D_ErrorCode error;
};

class ClosedCube_SHT31D {
```

```
/*
 * Class definition for ClosedCube_SHT31D
 */
public:
    ClosedCube_SHT31D();

    bool start_sht(void);
    bool run_sht(void);
    float get_t_ave(void);
    float get_rh_ave(void);


    SHT31D_ErrorCode begin(uint8_t address);
    SHT31D_ErrorCode clearAll();
    SHT31D_RegisterStatus readStatusRegister();

    SHT31D_ErrorCode heaterEnable();
    SHT31D_ErrorCode heaterDisable();

    SHT31D_ErrorCode softReset();
    SHT31D_ErrorCode generalCallReset();

    SHT31D_ErrorCode artEnable();

    uint32_t readSerialNumber();

    SHT31D printResult(String text, SHT31D result);
    SHT31D readTempAndHumidity(SHT31D_Repeatability repeatability, SHT31D_Mode mode, uint8_t timeout);
    SHT31D readTempAndHumidityClockStretch(SHT31D_Repeatability repeatability);
    SHT31D readTempAndHumidityPolling(SHT31D_Repeatability repeatability, uint8_t timeout);

    SHT31D_ErrorCode periodicStart(SHT31D_Repeatability repeatability, SHT31D_Frequency frequency);
    SHT31D periodicFetchData();
    SHT31D_ErrorCode periodicStop();

    SHT31D_ErrorCode writeAlertHigh(float temperatureSet, float temperatureClear, float humiditySet, float humidityClear);
    SHT31D readAlertHighSet();
    SHT31D readAlertHighClear();

    SHT31D_ErrorCode writeAlertLow(float temperatureClear, float temperatureSet, float humidityClear, float humiditySet);
    SHT31D readAlertLowSet();
    SHT31D readAlertLowClear();

private:
    float t_buf[MAX_READ_COUNT];
    float rh_buf[MAX_READ_COUNT];
    bool is_average_taken;
    int read_count;
    int error_count;
    float t_average;
    float rh_average;

    SHT31D save_to_buffer(SHT31D result);
    SHT31D read_sht(void);
    void calculate_average(void);

    uint8_t _address;
    SHT31D_RegisterStatus _status;

    SHT31D_ErrorCode writeCommand(SHT31D_Commands command);
    SHT31D_ErrorCode writeAlertData(SHT31D_Commands command, float temperature, float humidity);

    uint8_t checkCrc(uint8_t data[], uint8_t checksum);
    uint8_t calculateCrc(uint8_t data[]);

    float calculateHumidity(uint16_t rawValue);
    float calculateTemperature(uint16_t rawValue);

    uint16_t calculateRawHumidity(float value);
    uint16_t calculateRaWTemperature(float value);

    SHT31D readTemperatureAndHumidity();
    SHT31D readAlertData(SHT31D_Commands command);
    SHT31D_ErrorCode read(uint16_t* data, uint8_t numOfPair);

    SHT31D returnError(SHT31D_ErrorCode command);
};

#endif
```

```
/*
 * This is the .cpp file for the SHT35D Temperature
 * and relative humidity sensor.
 *
 * Part 1 of this code was retrieved online:
 * https://github.com/closedcube/ClosedCube_SHT31D_Arduino
 *
 * Part 2 was written by MECH 45X Team 26
 *
 * Part 1 begins...
 */

#include <Wire.h>
#include "SHT35D.h"

ClosedCube_SHT31D::ClosedCube_SHT31D()
{
}

SHT31D_ErrorCode ClosedCube_SHT31D::begin(uint8_t address) {
    SHT31D_ErrorCode error = SHT3XD_NO_ERROR;
    _address = address;
    return error;
}

SHT31D ClosedCube_SHT31D::periodicFetchData()
{
    SHT31D_ErrorCode error = writeCommand(SHT3XD_CMD_FETCH_DATA);
    if (error == SHT3XD_NO_ERROR)
        return readTemperatureAndHumidity();
    else
        returnError(error);
}

SHT31D_ErrorCode ClosedCube_SHT31D::periodicStop() {
    return writeCommand(SHT3XD_CMD_STOP_PERIODIC);
}

SHT31D_ErrorCode ClosedCube_SHT31D::periodicStart(SHT31D_Repeatability repeatability,
SHT31D_Frequency frequency)
{
    SHT31D_ErrorCode error;

    switch (repeatability)
    {
    case SHT3XD_REPEATABILITY_LOW:
        switch (frequency)
        {
        case SHT3XD_FREQUENCY_HZ5:
            error = writeCommand(SHT3XD_CMD_PERIODIC_HALF_L);
            break;
        case SHT3XD_FREQUENCY_1HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_1_L);
            break;
        case SHT3XD_FREQUENCY_2HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_2_L);
            break;
        case SHT3XD_FREQUENCY_4HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_4_L);
            break;
        case SHT3XD_FREQUENCY_10HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_10_L);
            break;
        default:
            error = SHT3XD_PARAM_WRONG_FREQUENCY;
            break;
        }
        break;
    case SHT3XD_REPEATABILITY_MEDIUM:
        switch (frequency)
        {
        case SHT3XD_FREQUENCY_HZ5:
            error = writeCommand(SHT3XD_CMD_PERIODIC_HALF_M);
            break;
        case SHT3XD_FREQUENCY_1HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_1_M);
            break;
        case SHT3XD_FREQUENCY_2HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_2_M);
            break;
        case SHT3XD_FREQUENCY_4HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_4_M);
            break;
        case SHT3XD_FREQUENCY_10HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_10_M);
            break;
        default:
            error = SHT3XD_PARAM_WRONG_FREQUENCY;
            break;
        }
        break;

    case SHT3XD_REPEATABILITY_HIGH:
        switch (frequency)
        {
        case SHT3XD_FREQUENCY_HZ5:
            error = writeCommand(SHT3XD_CMD_PERIODIC_HALF_H);
            break;
        case SHT3XD_FREQUENCY_1HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_1_H);
            break;
        case SHT3XD_FREQUENCY_2HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_2_H);
            break;
        case SHT3XD_FREQUENCY_4HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_4_H);
            break;
        case SHT3XD_FREQUENCY_10HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_10_H);
            break;
        default:
            error = SHT3XD_PARAM_WRONG_FREQUENCY;
            break;
        }
        break;

    default:
        error = SHT3XD_PARAM_WRONG_REPEATABILITY;
        break;
    }
    delay(100);
    return error;
}

SHT31D ClosedCube_SHT31D::readTempAndHumidity(SHT31D_Repeatability repeatability, SHT31D_Mode mode, uint8_t timeout)
{
    SHT31D result;

    switch (mode) {
    case SHT3XD_MODE_CLOCK_STRETCH:
        result = readTempAndHumidityClockStretch(repeatability);
        break;
    case SHT3XD_MODE_POLLING:
        result = readTempAndHumidityPolling(repeatability, timeout);
        break;
    default:
        result = returnError(SHT3XD_PARAM_WRONG_MODE);
        break;
    }
    return result;
}


SHT31D ClosedCube_SHT31D::readTempAndHumidityClockStretch(SHT31D_Repeatability repeatability)
{
    SHT31D_ErrorCode error = SHT3XD_NO_ERROR;
    SHT31D_Commands command;

    switch (repeatability)
    {
    case SHT3XD_REPEATABILITY_LOW:
        error = writeCommand(SHT3XD_CMD_CLOCK_STRETCH_L);
        break;
    case SHT3XD_REPEATABILITY_MEDIUM:
        error = writeCommand(SHT3XD_CMD_CLOCK_STRETCH_M);
        break;
    case SHT3XD_REPEATABILITY_HIGH:
        error = writeCommand(SHT3XD_CMD_CLOCK_STRETCH_H);
        break;
    default:
        error = SHT3XD_PARAM_WRONG_REPEATABILITY;
        break;
    }

    delay(50);

    if (error == SHT3XD_NO_ERROR) {
        return readTemperatureAndHumidity();
    } else {
        return returnError(error);
    }

}


SHT31D ClosedCube_SHT31D::readTempAndHumidityPolling(SHT31D_Repeatability repeatability, uint8_t timeout)
{
    SHT31D_ErrorCode error = SHT3XD_NO_ERROR;
    SHT31D_Commands command;

    switch (repeatability)
    {
    case SHT3XD_REPEATABILITY_LOW:
        error = writeCommand(SHT3XD_CMD_POLLING_L);
        break;
    case SHT3XD_REPEATABILITY_MEDIUM:
        error = writeCommand(SHT3XD_CMD_POLLING_M);
        break;
    case SHT3XD_REPEATABILITY_HIGH:
        error = writeCommand(SHT3XD_CMD_POLLING_H);
        break;
    default:
        error = SHT3XD_PARAM_WRONG_REPEATABILITY;
        break;
    }

    delay(50);

    if (error == SHT3XD_NO_ERROR) {
        return readTemperatureAndHumidity();
    } else {
        return returnError(error);
    }

```

```
}

SHT31D ClosedCube_SHT31D::readAlertHighSet() {
    return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_HS);
}

SHT31D ClosedCube_SHT31D::readAlertHighClear() {
    return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_HC);
}

SHT31D ClosedCube_SHT31D::readAlertLowSet() {
    return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_LS);
}

SHT31D ClosedCube_SHT31D::readAlertLowClear() {
    return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_LC);
}


SHT31D_ErrorCode ClosedCube_SHT31D::writeAlertHigh(float temperatureSet, float temperatureClear, float humiditySet, float humidityClear) {
    SHT31D_ErrorCode error = writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_HS, temperatureSet, humiditySet);
    if (error == SHT3XD_NO_ERROR)
        error = writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_HC, temperatureClear, humidityClear);

    return error;
}

SHT31D_ErrorCode ClosedCube_SHT31D::writeAlertLow(float temperatureClear, float temperatureSet, float humidityClear, float humiditySet) {
    SHT31D_ErrorCode error = writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_LS, temperatureSet, humiditySet);
    if (error == SHT3XD_NO_ERROR)
        writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_LC, temperatureClear, humidityClear);

    return error;
}

SHT31D_ErrorCode ClosedCube_SHT31D::writeAlertData(SHT31D_Commands command, float temperature, float humidity)
{
    SHT31D_ErrorCode  error;

    if ((humidity < 0.0) || (humidity > 100.0) || (temperature < -40.0) || (temperature > 125.0))
    {
        error = SHT3XD_PARAM_WRONG_ALERT;
    }
    else {
        uint16_t rawTemperature = calculateRaWTemperature(temperature);
        uint16_t rawHumidity = calculateRawHumidity(humidity);
        uint16_t data = (rawHumidity & 0xFE00) | ((rawTemperature >> 7) & 0x001FF);

        uint8_t buf[2];
        buf[0] = data >> 8;
        buf[1] = data & 0xFF;

        uint8_t checksum = calculateCrc(buf);

        Wire.beginTransmission(_address);
        Wire.write(command >> 8);
        Wire.write(command & 0xFF);
        Wire.write(buf[0]);
        Wire.write(buf[1]);
        Wire.write(checksum);
        return (SHT31D_ErrorCode)(-10 * Wire.endTransmission());
    }
    return error;
}


SHT31D_ErrorCode ClosedCube_SHT31D::writeCommand(SHT31D_Commands command)
{
    Wire.beginTransmission(_address);
    Wire.write(command >> 8);
    Wire.write(command & 0xFF);
    return (SHT31D_ErrorCode)(-10 * Wire.endTransmission());
}

SHT31D_ErrorCode ClosedCube_SHT31D::softReset() {
    return writeCommand(SHT3XD_CMD_SOFT_RESET);
}

SHT31D_ErrorCode ClosedCube_SHT31D::generalCallReset() {
    Wire.beginTransmission(0x0);
    Wire.write(0x06);
    return (SHT31D_ErrorCode)(-10 * Wire.endTransmission());
}

SHT31D_ErrorCode ClosedCube_SHT31D::heaterEnable() {
    return writeCommand(SHT3XD_CMD_HEATER_ENABLE);
}

SHT31D_ErrorCode ClosedCube_SHT31D::heaterDisable() {
    return writeCommand(SHT3XD_CMD_HEATER_DISABLE);
}

SHT31D_ErrorCode ClosedCube_SHT31D::artEnable() {
    return writeCommand(SHT3XD_CMD_ART);
}


uint32_t ClosedCube_SHT31D::readSerialNumber()
{
    uint32_t result = SHT3XD_NO_ERROR;
    uint16_t buf[2];

    if (writeCommand(SHT3XD_CMD_READ_SERIAL_NUMBER) == SHT3XD_NO_ERROR) {
        if (read(buf, 2) == SHT3XD_NO_ERROR) {
          result = (buf[0] << 16) | buf[1];
        }
    }

    return result;
}

SHT31D_RegisterStatus ClosedCube_SHT31D::readStatusRegister()
{
    SHT31D_RegisterStatus result;

    SHT31D_ErrorCode error = writeCommand(SHT3XD_CMD_READ_STATUS);
    if (error == SHT3XD_NO_ERROR)
        error = read(&result.rawData, 1);

    return result;
}

SHT31D_ErrorCode ClosedCube_SHT31D::clearAll() {
    return writeCommand(SHT3XD_CMD_CLEAR_STATUS);
}


SHT31D ClosedCube_SHT31D::readTemperatureAndHumidity()
{
    SHT31D result;

    result.t = 0;
    result.rh = 0;

    SHT31D_ErrorCode error;
    uint16_t buf[2];

    if (error == SHT3XD_NO_ERROR)
        error = read(buf, 2);

    if (error == SHT3XD_NO_ERROR) {
        result.t = calculateTemperature(buf[0]);
        result.rh = calculateHumidity(buf[1]);
    }
    result.error = error;

    return result;
}

SHT31D ClosedCube_SHT31D::readAlertData(SHT31D_Commands command)
{
    SHT31D result;

    result.t = 0;
    result.rh = 0;

    SHT31D_ErrorCode error;
    uint16_t buf[1];

    error = writeCommand(command);

    if (error == SHT3XD_NO_ERROR)
        error = read(buf, 1);

    if (error == SHT3XD_NO_ERROR) {
        result.rh = calculateHumidity(buf[0] << 7);
        result.t = calculateTemperature(buf[0] & 0xFE00);
    }

    result.error = error;

    return result;
}

SHT31D_ErrorCode ClosedCube_SHT31D::read(uint16_t* data, uint8_t numOfPair)
{
    uint8_t checksum;
    char buf[2];
    uint8_t buffer[2];


    const uint8_t numOfBytes = numOfPair * 3;
    Wire.requestFrom(_address, numOfBytes);

    int counter = 0;

    for (counter = 0; counter < numOfPair; counter++) {
        Wire.readBytes(buf, 2);
        checksum = Wire.read();

        for (int i = 0; i < 2; i++){buffer[i] = uint8_t(buf[i]);}


        if (checkCrc(buffer, checksum) != 0)
            return SHT3XD_CRC_ERROR;

        data[counter] = (buf[0] << 8) | buf[1];
    }

    return SHT3XD_NO_ERROR;
}


uint8_t ClosedCube_SHT31D::checkCrc(uint8_t data[], uint8_t checksum)
{
    return calculateCrc(data) != checksum;
}

float ClosedCube_SHT31D::calculateTemperature(uint16_t rawValue)
{
    return 175.0f * (float)rawValue / 65535.0f - 45.0f;
}


float ClosedCube_SHT31D::calculateHumidity(uint16_t rawValue)
{
    return 100.0f * rawValue / 65535.0f;
}

uint16_t ClosedCube_SHT31D::calculateRaWTemperature(float value)
{
    return (value + 45.0f) / 175.0f * 65535.0f;
}

uint16_t ClosedCube_SHT31D::calculateRawHumidity(float value)
{
    return value / 100.0f * 65535.0f;
}

uint8_t ClosedCube_SHT31D::calculateCrc(uint8_t data[])
{
    uint8_t bit;
    uint8_t crc = 0xFF;
    uint8_t dataCounter = 0;

    for (; dataCounter < 2; dataCounter++) {
        crc ^= (data[dataCounter]);
        for (bit = 8; bit > 0; --bit) {
            if (crc & 0x80){crc = (crc << 1) ^ 0x131;}
            else {crc = (crc << 1);}
        }
    }

    return crc;
}

SHT31D ClosedCube_SHT31D::returnError(SHT31D_ErrorCode error) {
    SHT31D result;
    result.t = 0;
    result.rh = 0;
    result.error = error;
    return result;
}

//***********************************************************//
// Part 2: Code Written by team 26                           //
// Team 26 understands this code                             //
// Therefore it is properly commented                        //
//***********************************************************//
bool ClosedCube_SHT31D::start_sht(void) {
    /*
     * Start sequence for SHT35D
     * Return true: sensor was succesfully started
     * Return false: sensor was not started
     * Try to read from sensor
     * If no error, return true
     * Else return false
     */
    Serial.println("Trying to start SHT sensor...");
    delay(500);
    begin(ADDR_SHT); // I2C address: 0x44 or 0x45
    Serial.print("Serial #");
    Serial.println(readSerialNumber());
    delay(500);

    if (periodicStart(SHT3XD_REPEATABILITY_HIGH, SHT3XD_FREQUENCY_10HZ) != SHT3XD_NO_ERROR) {
        Serial.println("[ERROR] Cannot start periodic mode");
        return false;
    }
    else {
        Serial.println("Successfully started SHT sensor!");
        return true;
    }
}

bool ClosedCube_SHT31D::run_sht(void) {
    /*
     * Run SHT sensor
     * start read_count from 1
     * is_average_taken is false until average is taken
     * take reading from sht until enough values are read to take an average
     */
    is_average_taken = false;
    error_count = 1;
    read_count = 1;
    while(read_count <= MAX_READ_COUNT && error_count <= MAX_ERROR_COUNT) {
        read_sht();
    }

    return(is_average_taken);
}

SHT31D ClosedCube_SHT31D::read_sht(void) {
    /*
     * Read from SHT35D, and assign values to my_result
     * print results
     * save results to buffer
     * calculate average if enough values have been read
     */
    SHT31D my_result = periodicFetchData();
    printResult("Periodic Mode", my_result);
    save_to_buffer(my_result);
    calculate_average();
    delay(250);
}

SHT31D ClosedCube_SHT31D::printResult(String text, SHT31D result) {
    /*
     * Prints current reading if no error and not exceeded max count
     * else print error message
     */
    if (result.error == SHT3XD_NO_ERROR && read_count <= MAX_READ_COUNT ) {
        float current_t = result.t;
        float current_rh = result.rh;

        if(current_t > 0 && current_rh > 0) {
            //Serial.print(text);
            Serial.print("SHT Reading #");
            Serial.print(read_count);
            Serial.print(": T=");
            Serial.print(current_t);
            Serial.print("C, RH=");
            Serial.print(current_rh);
            Serial.println("%");
        }
    }
```

```
}

SHT31D ClosedCube_SHT31D::save_to_buffer(SHT31D result) {
    /*
     * Save current t and rh readings to their respective buffers
     *
     * if no error and the number of readings is less than the max
     * then save values
     *
     * else -> report error, do not save any values
     */
    if (result.error == SHT3XD_NO_ERROR && read_count <= MAX_READ_COUNT) {
        float current_t = result.t;
        float current_rh = result.rh;

        if(current_t > 0 && current_rh > 0) {
            t_buf[read_count - 1] = current_t;
            rh_buf[read_count - 1] = current_rh;
            read_count++;
            error_count = 1;
        } else {
            Serial.print("SHT Error count: ");
            Serial.println(error_count);
            error_count ++;
        }
    } else if (result.error != SHT3XD_NO_ERROR) {
        Serial.print("[ERROR] Code #");
        Serial.println(result.error);
        Serial.print("SHT Error count: ");
        Serial.println(error_count);
        error_count ++;
    }
}

void ClosedCube_SHT31D::calculate_average(void) {
    /*
     * Calculate average if enough values have been read
     * assign t ave to t_average
     * assign rh ave to rh_average
     * change is_average_taken to true so that while loop will exit
     */
    if( read_count > MAX_READ_COUNT ) {
        t_average = 0.00;
        rh_average = 0.00;
        for(int k = 0; k < MAX_READ_COUNT; k++) {
            t_average += t_buf[k];
            rh_average += rh_buf[k];
        }
        t_average = t_average / MAX_READ_COUNT;
        rh_average = rh_average / MAX_READ_COUNT;

        delay(500);
        Serial.println("-----------------------");
        Serial.println("SHT Sensor Average Readings");
        Serial.println("-----------------------");
        Serial.print("SHT T Average: ");
        Serial.println(t_average);
        Serial.print("SHT RH Average: ");
        Serial.println(rh_average);
        is_average_taken = true;
    }
}

// getter function to get average temperature reading
float ClosedCube_SHT31D::get_t_ave(void) {
    return t_average;
}
// getter function to get average relative humidity reading
float ClosedCube_SHT31D::get_rh_ave(void) {
    return rh_average;
}

```

The code for running the Si7015 Globe Thermometer Temperature sensor:

`’MRT.cpp’ and ’MRT.h’`

This code reads from the Si7015 sensor several time and takes an average value of all of the readings. The Si7015 communicates using an I2C connection. This code was retrieved from:

`https://github.com/closedcube/ClosedCube_Si7051_Arduino`

The on line library was supplemented by additional methods added by Team 26. The .h file is presented first, followed by the .cpp file.

```
/*
 * This is the .h file for the Si7051 sensor
 * This sensor is used in the globe thermometer
 */

#ifndef _CLOSEDCUBE_SI7051_h

#define _CLOSEDCUBE_SI7051_h
#define MAX_READ_COUNT 5
#define MAX_ERROR_COUNT 40
#define ADDR_MRT 0x40
#define DEFAULT_AVERAGE 128
#include <Arduino.h>

class ClosedCube_Si7051 {
public:
    ClosedCube_Si7051();

    float readT(); // short-cut for readTemperature
    bool run_mrt(void);
    bool start_mrt(void);
    float get_MRT_ave(void);

private:
    uint8_t _address;
    void begin(uint8_t address);
    float readTemperature();
    float T_buf[MAX_READ_COUNT];
    float T_ave;
    int read_count;
    int error_count;
};

#endif

```

```
/*
 * This is the .cpp file for the Si7051 sensor
 * The Si7015 is being used as the Globe Thermometer Sensor
 * The bulk of this library was retrieved on line:
 * https://github.com/closedcube/ClosedCube_Si7051_Arduino
 *
 * Part 1 of this library was retrieved on line,
 * while Part 2 was written by MECH 45X Team 26
 *
 * Team 26 does not fully understand how the on line
 * library works, so Part 1 is not commented
 *
 * Team 26 commented Part 2 as they wrote Part 2
 * and understand how the code in Part 2 works
 *
 * Please note that the Globe Thermometer does not
 * measure Mean Radiant Temperature (MRT), it
 * actually measures the globe temperature.
 * MRT is calculate later using air temperature and
 * globe temperature.
 */

#include <Wire.h>
#include "MRT.h"

ClosedCube_Si7051::ClosedCube_Si7051()
{
}

void ClosedCube_Si7051::begin(uint8_t address) {
    _address = address;
    Wire.begin();

    Wire.beginTransmission(_address);
    Wire.write(0xE6);
    Wire.write(0x0);
    Wire.endTransmission();

}

float ClosedCube_Si7051::readT() {
    return readTemperature();
}

float ClosedCube_Si7051::readTemperature() {
    Wire.beginTransmission(_address);
    Wire.write(0xF3);
    Wire.endTransmission();

    delay(15);

    Wire.requestFrom(_address, (uint8_t)2);
    delay(25);
    byte msb = Wire.read();
    byte lsb = Wire.read();

    uint16_t val = msb << 8 | lsb;

    return (175.72*val) / 65536 - 46.85;
}

//********************************************************//
// Part 2: Si7051 MECH 45X Team 26 library                //
// The following code was written by MECH 45X Team 26     //
// It is properly commented                               //
//********************************************************//


bool ClosedCube_Si7051::start_mrt(void) {
    /*
     * Start MRT sensor
     *
     * The code will read a value of 128 or greater
     * if the sensor is broken or disconnected
     *
     * The start sequence returns false (sensor does not work)
     * if a value of 128 is read
     *
     * If the value is less than 128, it returns true
     * (sensor works)
     *
     * The code retrieved from the online library should be improved
     * to fix this.
     */
    begin(ADDR_MRT);
    delay(500);
    return(run_mrt());
}

bool ClosedCube_Si7051::run_mrt(void) {
    /*
     * Takes MRT measurements until read_count is exceeded
     * once read_count is exceeded, the average is taken
     */
    read_count = 1;
    error_count = 1;

    while(read_count <= MAX_READ_COUNT && error_count <= MAX_ERROR_COUNT) {
        float current_T = readTemperature();

        if(current_T >= DEFAULT_AVERAGE) {
            Serial.println("------------------------------------------------");
            Serial.print("Error reading from Globe Thermometer, Tg: ");
            Serial.println(current_T);
            Serial.println("------------------------------------------------");
            error_count ++;
            delay(1000);
        } else{
            T_buf[read_count - 1] = readTemperature();
            Serial.print("Globe Thermometer Reading #");
            Serial.print(read_count);
            Serial.print(": Tg is: ");
            Serial.println(T_buf[read_count - 1]);
            read_count ++;
            error_count = 1;
            delay(250);
        }
    }

    if(read_count > MAX_READ_COUNT) {
        T_ave = 0;
        for(int k = 0; k < MAX_READ_COUNT; k++) {
            T_ave = T_ave + T_buf[k];
        }
        T_ave = T_ave / MAX_READ_COUNT;
        Serial.println("--------------------");
        Serial.print("Average Tg is: ");
        Serial.println(T_ave);
        Serial.println("--------------------");
        return(true);
    }
    else if(error_count > MAX_ERROR_COUNT) {
        T_ave = -1;
        Serial.println("--------------------------------------------------------------");
        Serial.println("Error reading from Globe Thermometer, no average Tg calculated");
        Serial.println("--------------------------------------------------------------");
        return(false);
    }
    else{
        Serial.println("---------------------------");
        Serial.println("Failure for no known reason");
        Serial.println("---------------------------");
        return(false);}
}

// Getter function for Globe Thermometer average temperature
float ClosedCube_Si7051::get_MRT_ave(void) {
    return T_ave;
}
```