MECH 49X
Dossier 11 - Code

Team 26

March 24, 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.

```
/*
 * 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 <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 expression
 * Indicate whether sensor has
 * been started successfully
 */
bool start_co2 = false;
bool start_voc = false;
bool start_sht = false;
bool start_pm = false;
bool start_mrt = 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;
float 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 co2_transistor_control = A3;

void setup() {
    /*
     * start Serial and wire connections
     * try to start each sensor
     * assign true false to each of the relevant booleans
     */
    Serial.begin(9600);
    Wire.begin();
    pinMode(pm_transistor_control,OUTPUT);
    pinMode(co2_transistor_control,OUTPUT);
    Serial.println("Initializing");

    Serial.println("Trying to start CO2 sensor");
    delay(1000);
    digitalWrite(co2_transistor_control, HIGH);
    start_co2 = myCO2.start_sensor();
```

```
70      Serial.println("-----------------------");
71      digitalWrite(co2_transistor_control, LOW);
72
73      start_mrt = myMRT.start_mrt();
74      Serial.println("-----------------------");
75      start_sht = mySHT.start_sht();
76      Serial.println("-----------------------");
77
78      start_voc = myVOC.start_voc();
79      Serial.println("-----------------------");
80
81      Serial.println("Trying to start PM sensor");
82      digitalWrite(pm_transistor_control, HIGH);
83      delay(1000);
84      start_pm = myPM.run_PM_sensor();
85      digitalWrite(pm_transistor_control, LOW);
86
87      if(start_pm){Serial.println("Successfully started PM sensor");}
88      else if(!start_pm){Serial.println("Failed to start PM sensor");}
89      Serial.println("-----------------------");
90  }
91
92  void loop() {
93      /*
94       * Run each sensor if it has been started
95       * If the sensor has not been started, print error message
96       * After all values have been read, prepare to publish data
97       */
98      if(start_co2) {
99          digitalWrite(co2_transistor_control, HIGH);
100         delay(1800);
101         start_co2 = myCO2.run_sensor();
102         digitalWrite(co2_transistor_control, LOW);
103         delay(1000);
104     }
105
106     if(start_pm) {
107         Serial.println("Reading from PMS Sensor");
108         Serial.println("-----------------------");
109         digitalWrite(pm_transistor_control, HIGH);
110         delay(10000);
111         start_pm = myPM.run_PM_sensor();
112         digitalWrite(pm_transistor_control, LOW);
113         delay(500);
114     }
115     else if(!start_pm) {
116         Serial.println("Not reading from PMS Sensor");
117         Serial.println("---------------------------");
118         delay(500);
119     }
120
121     if(start_mrt) {
122         myMRT.run_mrt();
123         delay(500);
124     }
125     else if(!start_mrt) {
126         Serial.println("Not reading from MRT Sensor");
127         Serial.println("---------------------------");
128         delay(500);
129     }
130
131     if(start_sht) {
132         Serial.println("Reading from SHT Sensor");
133         Serial.println("-----------------------");
134         mySHT.run_sht();
135         Serial.println("-----------------------");
136     }
137     else if(!start_sht) {
138         Serial.println("Not reading from SHT Sensor");
```

```
        Serial.println("--------------------------");
        delay(500);
    }

    if(start_voc) {
        Serial.println("Reading from VOC Sensor");
        Serial.println("-----------------------");
        myVOC.run_voc();
        Serial.println("-----------------------");
    }
    else if(!start_voc) {
        Serial.println("Not reading from VOC Sensor");
        Serial.println("--------------------------");
        delay(500);
    }

    if(publish_data) {
        char data[1000];
        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();
        }
        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_pm){pm_ave = myPM.getpm();}
        else {pm_ave = -1;}

        if(start_co2){co2_ave = myCO2.get_co2_ave();}
        else{co2_ave = -1;}

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


        sprintf(data,"{ \"Mean Radiant Temperature\": \"%f\", \"Operating
        Temperature\": \"%f\", \"Globe Temperature\": \"%f\", \"CO2 Concentration\":
        \"%i\", \"TVOC\": \"%f\",\"PM 2.5 (Counts/m^3)\": \"%f\", \"Air Temperature\":
        \"%f\",\"Relative Humidity of Air\": \"%f\"}" , T_mrt, T_ot, T_g, co2_ave,
        voc_TVOC_ave, pm_ave, T_a, sht_rh_ave);
        Serial.println(data);
```

```
204             Particle.publish("IEQ Data", data, PRIVATE);
205
206         }
207     }
208
209
```

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"

#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

class PM_7003 {
    public:
        PM_7003();
        virtual ~PM_7003();
        bool run_PM_sensor(void);
        int getpm(void);

private:
    int current_byte;
    bool sync_state;
    char print_buffer[256];
    uint16_t byte_sum;
    int drain;
    uint16_t current_data;
    float pm_avgpm2_5;
    int pm2_5;

    bool done_reading;
    int read_count;
    int frame_sync_count;

    char frame_buffer[MAX_FRAME_LENGTH];
    int frame_count;
    int frame_length;

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

    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;
```

```
70          } packetdata;
71      };
72
73
```

```cpp
/*
 * 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
  */
```

```
70        sync_state = false;
71        frame_count = 0;
72        byte_sum = 0;
73
74        while (!sync_state && frame_sync_count < MAX_FRAME_SYNC_COUNT){
75            current_byte = Serial1.read();
76
77            if(current_byte == FIRST_BYTE && frame_count == 0) {
78                frame_buffer[frame_count] = current_byte;
79                packetdata.start_frame[0] = current_byte;
80                byte_sum = current_byte;
81                frame_count = 1;
82            }
83            else if(current_byte == SECOND_BYTE && frame_count == 1){
84                frame_buffer[frame_count] = current_byte;
85                packetdata.start_frame[1] = current_byte;
86                byte_sum = byte_sum + current_byte;
87                frame_count = 2;
88                sync_state = true;
89            }
90            else{
91                frame_sync_count++;
92                Serial.println("frame is syncing");
93                Serial.print("Current character: ");
94                Serial.println(current_byte, HEX);
95                Serial.print("frame count: ");
96                Serial.println(frame_sync_count);
97                delay(500);
98
99                if(frame_sync_count >= MAX_FRAME_SYNC_COUNT) {
100                   Serial.println("------------------------");
101                   Serial.println("Max frame count exceeded");
102                   Serial.println("------------------------");
103               }
104
105           }
106       }
107   }
108
109   void PM_7003::read_sensor(void) {
110       /*
111        * Sync the frames
112        * read bytes and fill frame_buffer
113        * use data_switch to calculate different parameters
114        * print_messages once all values have been read.
115        * done_reading = true if enough values have been read
116        */
117       frame_sync();
118
119       while(sync_state == true && Serial1.available() > 0) {
120           current_byte = Serial1.read();
121           frame_buffer[frame_count] = current_byte;
122           byte_sum = byte_sum + current_byte;
123           frame_count++;
124           uint16_t current_data = frame_buffer[frame_count-1]+(frame_buffer[frame_count-2
          ]<<8);
125           data_switch(current_data);
126
127           if (frame_count >= frame_length && read_count <= MAX_READ_COUNT) {
128               print_messages();
129               pm_avgpm2_5 = pm_avgpm2_5 + pm2_5;
130               read_count++;
131               break;
132           }
133       }
134
135       if (read_count > MAX_READ_COUNT) {
136           pm_avgpm2_5 = exp((pm_avgpm2_5/MAX_READ_COUNT + 109314)/15990)*10000;
137           done_reading = true;
```

```
138
139         }
140     }
141
142     void PM_7003::data_switch(uint16_t current_data) {
143         /*
144          * data_switch uses current data and frame_count
145          * to assign values to parameters
146          */
147         switch (frame_count) {
148         case 4:
149             packetdata.frame_length = current_data;
150             frame_length = current_data + frame_count;
151             break;
152         case 6:
153             packetdata.concPM1_0_factory = current_data;
154             break;
155         case 8:
156             packetdata.concPM2_5_factory = current_data;
157             break;
158         case 10:
159             packetdata.concPM10_0_factory = current_data;
160             break;
161         case 12:
162             packetdata.concPM1_0_ambient = current_data;
163             break;
164         case 14:
165             packetdata.concPM2_5_ambient = current_data;
166             break;
167         case 16:
168             packetdata.concPM10_0_ambient = current_data;
169             break;
170         case 18:
171             packetdata.countPM0_3um = current_data;
172             break;
173         case 20:
174             packetdata.countPM0_5um = current_data;
175             break;
176         case 22:
177             packetdata.countPM1_0um = current_data;
178             break;
179         case 24:
180             packetdata.countPM2_5um = current_data;
181             break;
182         case 26:
183             packetdata.countPM5_0um = current_data;
184             break;
185         case 28:
186             packetdata.countPM10_0um = current_data;
187             break;
188         case 29:
189             current_data = frame_buffer[frame_count-1];
190             packetdata.version = current_data;
191           break;
192         case 30:
193             current_data = frame_buffer[frame_count-1];
194             packetdata.error = current_data;
195             break;
196         case 32:
197             packetdata.checksum = current_data;
198             byte_sum -= ((current_data>>8)+(current_data&0xFF));
199             break;
200         default:
201             break;
202         }
203     }
204
205     void PM_7003::print_messages(void){
206         /*
```

```
207         * Print messages to string and Serial screen
208         */
209       Serial.println("-----------------------");
210       Serial.print("PMS 7003 - Reading #");
211       Serial.println(read_count);
212       Serial.println("-----------------------");
213       sprintf(print_buffer, ", %02x, %02x, %04x, ",
214           packetdata.start_frame[0], packetdata.start_frame[1], packetdata.frame_length);
215       sprintf(print_buffer, "%s%04d, %04d, %04d, ", print_buffer,
216           packetdata.concPM1_0_factory, packetdata.concPM2_5_factory, packetdata.
              concPM10_0_factory);
217       sprintf(print_buffer, "%s%04d, %04d, %04d, ", print_buffer,
218           packetdata.concPM1_0_ambient, packetdata.concPM2_5_ambient, packetdata.
              concPM10_0_ambient);
219       sprintf(print_buffer, "%s%04d, %04d, %04d, %04d, %04d, %04d, ", print_buffer,
220           packetdata.countPM0_3um, packetdata.countPM0_5um, packetdata.countPM1_0um,
221           packetdata.countPM2_5um, packetdata.countPM5_0um, packetdata.countPM10_0um);
222       sprintf(print_buffer, "%s%02d, %02d, ", print_buffer,
223           packetdata.version, packetdata.error);
224
225       pm2_5 = packetdata.countPM1_0um - packetdata.countPM2_5um + packetdata.countPM0_5um
          - packetdata.countPM1_0um + packetdata.countPM0_3um - packetdata.countPM0_5um;
226       Serial.println(print_buffer);
227       Serial.println("-----------------------");
228       delay(500);
229   }
230
231
```

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 .h file for the MH-Z19 CO2 Sensor
 * This code was written exclusively 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 20
#define DISCARD_VALUES 10
#define STARTUP_TIME 10
#define MAX_FRAME_READ_COUNT 40
#include "WProgram.h"


class MHZ19 {
    public:
        MHZ19();
        virtual ~MHZ19();
        int get_co2_reading(void);
        bool run_sensor(void);
        bool start_sensor(void);
        int get_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 sync_state;
        bool does_sensor_work;
        bool is_average_taken;

        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 mhz19_buffer[NUMBER_OF_VALUES];

        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);
        void start_countdown(int start_time);
};

#endif /* MHZ19_H_ */

```

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

#include "MHZ19.h"

MHZ19::MHZ19() {
}

MHZ19::~MHZ19() {
}

bool MHZ19::start_sensor(void) {
    /*
     * Start sequence for MHZ19
     * returns true if sensor on, false if sensor off
     * uses run_sensor() function
     */
    return (run_sensor());
}

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();
    start_countdown(STARTUP_TIME);

    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::start_countdown(int start_time) {
    /*
     * Countdown so that users can visualize how long before the sensor starts
     */
    while(start_time > 0) {
        Serial.print("Starting CO2 Sensor in: ");
        Serial.print(start_time);
        Serial.println("s");
        delay(1000);
        start_time = start_time - 1;
    }
}

```

```
70  void MHZ19::print_current_reading(void) {
71      /*
72       * Prints current reading if reading is valid (i.e. co2_ppm > 0)
73       * and if the maximum number of readings haven't been exceeded
74       */
75      if(co2_ppm > 0 && reading_count > DISCARD_VALUES) {
76        Serial.print("MHZ19 CO2 PPM Reading ");
77        Serial.print(reading_count);
78        Serial.print(": ");
79        Serial.println(co2_ppm);
80      }
81      else if(co2_ppm > 0 && reading_count <= DISCARD_VALUES) {
82          Serial.print("DISCARD - MHZ19 CO2 PPM Reading ");
83          Serial.print(reading_count);
84          Serial.print(": ");
85          Serial.println(co2_ppm);
86      }
87      else {
88        Serial.println("Error reading CO2 PPM from MHZ19");
89      }
90  }
91
92  void MHZ19::add_to_ave_buf(void) {
93      /*
94       * IF a valid value of co2 is read and the number of reading is less than the max,
95       * THEN add current value to buffer
96       */
97      if(co2_ppm > 0 && reading_count <= NUMBER_OF_VALUES) {
98          mhz19_buffer[reading_count - 1] = co2_ppm;
99          reading_count += 1;
100     }
101 }
102
103 void MHZ19::calculate_average_reading(void) {
104     /*
105      * IF the number of readings exceeds the number of values to be read,
106      * THEN calculate the average
107      */
108     if(reading_count > NUMBER_OF_VALUES) {
109         for(int k = DISCARD_VALUES; k < NUMBER_OF_VALUES; k++) {co2_ppm_average +=
        mhz19_buffer[k];}
110
111         co2_ppm_average = co2_ppm_average / ( NUMBER_OF_VALUES - DISCARD_VALUES );
112
113         is_average_taken = true;
114     }
115 }
116
117 void MHZ19::print_average_reading(void) {
118     /*
119      * IF the average has been taken (co2_ppm_average > 0)
120      * THEN print the average
121      */
122     if(co2_ppm_average > 0) {
123         Serial.print("CO2 PPM Average Reading: ");
124         Serial.println(co2_ppm_average);
125     }
126 }
127
128 void MHZ19::read_sensor(void) {
129     /*
130      *  Start Serial1 connection
131      *  Send command to read from sensor to the sensor
132      *  Read from the sensor (fill_from_buffer();)
133      *  Calculate PPM for CO2
134      *  End Serial connection
135     */
136
137     Serial1.begin(9600);
```

```
138       Serial1.write(mhz19_read_command, 9);
139       delay(1000);
140       fill_frame_buffer();
141       co2_ppm = 256*frame_buffer[2] + frame_buffer[3];
142       Serial1.end();
143   }
144
145   void MHZ19::serial_drain(void) {
146     /*
147      * Drains serial buffer when sensor is turned on
148      */
149       while (Serial1.available() > 0) {
150           drain = Serial1.available();
151           Serial.print("-- Draining buffer: ");
152       }
153   }
154
155   void MHZ19::frame_sync(void) {
156       /*
157        * Sync frames so that frames are added to the frame_buffer in the correct order
158        * IF correct byte is read, THEN add to buffer and move on to next byte
159        * ELSE read byte and discard
160        * IF no bytes are available to read and the frames have not been synced, THEN send
          command to read from sensor again
161        *
162        * frame_sync_count keeps track of how many frames are added to frame_buffer
163        * frame_read_count keeps track of how many frames are read but not added to buffer
          (fails if too many frames read)
164        */
165       sync_state = false;
166       frame_sync_count = 0;
167       frame_read_count = 0;
168       byte_sum = 0;
169
170       while (!sync_state && Serial1.available() > 0 && frame_read_count <
          MAX_FRAME_READ_COUNT) {
171           current_byte = Serial1.read();
172
173           if (current_byte == MHZ19_ZEROTH_BYTE && frame_sync_count == 0) {
174               frame_buffer[frame_sync_count] = current_byte;
175               byte_sum = current_byte;
176               frame_sync_count = 1;
177           }
178           else if (current_byte == MHZ19_FIRST_BYTE && frame_sync_count == 1) {
179               frame_buffer[frame_sync_count] = current_byte;
180               byte_sum += current_byte;
181               sync_state = true;
182               frame_sync_count = 2;
183           }
184           else {
185               Serial.print("-- Frame syncing... ");
186               Serial.println(current_byte, HEX);
187               frame_read_count ++;
188           }
189
190           if (!sync_state && !(Serial1.available() > 0) && frame_read_count <
              MAX_FRAME_READ_COUNT) {
191               Serial1.write(mhz19_read_command, 9);
192               Serial.println("Read command has been sent to CO2 sensor");
193               delay(500);
194           }
195       }
196   }
197
198   void MHZ19::fill_frame_buffer(void) {
199     /*
200      * Sync frames
201      * Read byte into frame_buffer
202      */
```

```
203         frame_sync();
204
205         while(sync_state && Serial1.available() > 0 && frame_sync_count < MAX_FRAME_LEN) {
206             current_byte = Serial1.read();
207             frame_buffer[frame_sync_count] = current_byte;
208             byte_sum += current_byte;
209             frame_sync_count++;
210         }
211     }
212
213     // getter functions
214     int MHZ19::get_co2_ave(void) {return co2_ppm_average;}
215     int MHZ19::get_co2_reading(void) {return co2_ppm;}
216
```

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

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

/**************************************************************************/
/*!
```

```
70      @brief  Class that stores state and functions for interacting with CCS811 gas
        sensor chips
71  */
72  /**************************************************************************/
73  class Adafruit_CCS811 {
74    public:
75      //constructors
76      Adafruit_CCS811(void) {};
77      ~Adafruit_CCS811(void) {};
78
79      bool start_voc(void);
80      void run_voc(void);
81      float get_eCO2_ave(void);
82          float get_TVOC_ave(void);
83
84      bool begin(uint8_t addr = CCS811_ADDRESS);
85
86      void setEnvironmentalData(uint8_t humidity, double temperature);
87
88      //calculate temperature based on the NTC register
89      double calculateTemperature();
90
91      void setThresholds(uint16_t low_med, uint16_t med_high, uint8_t hysteresis = 50);
92
93      void SWReset();
94
95      void setDriveMode(uint8_t mode);
96      void enableInterrupt();
97      void disableInterrupt();
98
99          /**************************************************************************/
100         /*!
101             @brief  returns the stored total volatile organic compounds measurement.
                This does does not read the sensor. To do so, call readData()
102             @returns TVOC measurement as 16 bit integer
103         */
104         /**************************************************************************/
105     uint16_t getTVOC() { return _TVOC; }
106
107         /**************************************************************************/
108         /*!
109             @brief  returns the stored estimated carbon dioxide measurement. This does
                does not read the sensor. To do so, call readData()
110             @returns eCO2 measurement as 16 bit integer
111         */
112         /**************************************************************************/
113     uint16_t geteCO2() { return _eCO2; }
114
115         /**************************************************************************/
116         /*!
117             @brief  set the temperature compensation offset for the device. This is
                needed to offset errors in NTC measurements.
118             @param offset the offset to be added to temperature measurements.
119         */
120         /**************************************************************************/
121     void setTempOffset(float offset) { _tempOffset = offset; }
122
123     //check if data is available to be read
124     bool available();
125     uint8_t readData();
126
127     bool checkError();
128
129   private:
130     float eCO2_buf[MAX_READ_COUNT];
131         float TVOC_buf[MAX_READ_COUNT];
132         float eCO2_ave;
133         float TVOC_ave;
134         void read_voc(void);
```

```
135          void fill_buffer(void);
136          void print_readings(void);
137          void calculate_average_reading(void);
138          void print_average_reading(void);
139          int read_count;
140          bool is_average_taken;
141
142      uint8_t _i2caddr;
143      float _tempOffset;
144
145      uint16_t _TVOC;
146      uint16_t _eCO2;
147
148      void      write8(byte reg, byte value);
149      void      write16(byte reg, uint16_t value);
150          uint8_t   read8(byte reg);
151
152      void read(uint8_t reg, uint8_t *buf, uint8_t num);
153      void write(uint8_t reg, uint8_t *buf, uint8_t num);
154      void _i2c_init();
155
156  /*=========================================================================
157    REGISTER BITFIELDS
158      -----------------------------------------------------------------------*/
159      // The status register
160          struct status {
161
162              /* 0: no error
163              *  1: error has occurred
164              */
165              uint8_t ERROR: 1;
166
167              // reserved : 2
168
169              /* 0: no samples are ready
170              *  1: samples are ready
171              */
172              uint8_t DATA_READY: 1;
173              uint8_t APP_VALID: 1;
174
175        // reserved : 2
176
177              /* 0: boot mode, new firmware can be loaded
178              *  1: application mode, can take measurements
179              */
180              uint8_t FW_MODE: 1;
181
182              void set(uint8_t data){
183                ERROR = data & 0x01;
184                DATA_READY = (data >> 3) & 0x01;
185                APP_VALID = (data >> 4) & 0x01;
186                FW_MODE = (data >> 7) & 0x01;
187              }
188          };
189          status _status;
190
191          //measurement and conditions register
192          struct meas_mode {
193            // reserved : 2
194
195            /* 0: interrupt mode operates normally
196              *  1: Interrupt mode (if enabled) only asserts the nINT signal (driven low)
                if the new
197            ALG_RESULT_DATA crosses one of the thresholds set in the THRESHOLDS register
198            by more than the hysteresis value (also in the THRESHOLDS register)
199                */
200            uint8_t INT_THRESH: 1;
201
202            /* 0: int disabled
```

```
203             *  1: The nINT signal is asserted (driven low) when a new sample is ready in
204           ALG_RESULT_DATA. The nINT signal will stop being driven low when
205           ALG_RESULT_DATA is read on the I²C interface.
206             */
207           uint8_t INT_DATARDY: 1;
208
209           uint8_t DRIVE_MODE: 3;
210
211           uint8_t get(){
212             return (INT_THRESH << 2) | (INT_DATARDY << 3) | (DRIVE_MODE << 4);
213           }
214         };
215         meas_mode _meas_mode;
216
217         struct error_id {
218           /* The CCS811 received an I²C write request addressed to this station but with
219       invalid register address ID */
220           uint8_t WRITE_REG_INVALID: 1;
221
222           /* The CCS811 received an I²C read request to a mailbox ID that is invalid */
223           uint8_t READ_REG_INVALID: 1;
224
225           /* The CCS811 received an I²C request to write an unsupported mode to
226       MEAS_MODE */
227           uint8_t MEASMODE_INVALID: 1;
228
229           /* The sensor resistance measurement has reached or exceeded the maximum
230       range */
231           uint8_t MAX_RESISTANCE: 1;
232
233           /* The Heater current in the CCS811 is not in range */
234           uint8_t HEATER_FAULT: 1;
235
236           /*  The Heater voltage is not being applied correctly */
237           uint8_t HEATER_SUPPLY: 1;
238
239           void set(uint8_t data){
240             WRITE_REG_INVALID = data & 0x01;
241             READ_REG_INVALID = (data & 0x02) >> 1;
242             MEASMODE_INVALID = (data & 0x04) >> 2;
243             MAX_RESISTANCE = (data & 0x08) >> 3;
244             HEATER_FAULT = (data & 0x10) >> 4;
245             HEATER_SUPPLY = (data & 0x20) >> 5;
246           }
247         };
248         error_id _error_id;
249
250   /*=========================================================================*/
251   };
252
253   #endif
254
```

```
/*
 * 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.
*/
/**************************************************************************/
```

```
69  void Adafruit_CCS811::enableInterrupt()
70  {
71    _meas_mode.INT_DATARDY = 1;
72    this->write8(CCS811_MEAS_MODE, _meas_mode.get());
73  }
74
75  /**************************************************************************/
76  /*!
77      @brief  disable the data ready interrupt pin on the device
78  */
79  /**************************************************************************/
80  void Adafruit_CCS811::disableInterrupt()
81  {
82    _meas_mode.INT_DATARDY = 0;
83    this->write8(CCS811_MEAS_MODE, _meas_mode.get());
84  }
85
86  /**************************************************************************/
87  /*!
88      @brief  checks if data is available to be read.
89      @returns True if data is ready, false otherwise.
90  */
91  /**************************************************************************/
92  bool Adafruit_CCS811::available()
93  {
94    _status.set(read8(CCS811_STATUS));
95    if(!_status.DATA_READY)
96      return false;
97    else return true;
98  }
99
100 /**************************************************************************/
101 /*!
102     @brief  read and store the sensor data. This data can be accessed with getTVOC()
        and geteCO2()
103     @returns 0 if no error, error code otherwise.
104 */
105 /**************************************************************************/
106 uint8_t Adafruit_CCS811::readData()
107 {
108   if(!available())
109     return false;
110   else{
111     uint8_t buf[8];
112     this->read(CCS811_ALG_RESULT_DATA, buf, 8);
113
114     _eCO2 = ((uint16_t)buf[0] << 8) | ((uint16_t)buf[1]);
115     _TVOC = ((uint16_t)buf[2] << 8) | ((uint16_t)buf[3]);
116
117     if(_status.ERROR)
118       return buf[5];
119
120     else return 0;
121   }
122 }
123
124 /**************************************************************************/
125 /*!
126     @brief  set the humidity and temperature compensation for the sensor.
127     @param humidity the humidity data as a percentage. For 55% humidity, pass in
        integer 55.
128     @param temperature the temperature in degrees C as a decimal number. For 25.5
        degrees C, pass in 25.5
129 */
130 /**************************************************************************/
131 void Adafruit_CCS811::setEnvironmentalData(uint8_t humidity, double temperature)
132 {
133   /* Humidity is stored as an unsigned 16 bits in 1/512%RH. The
134   default value is 50% = 0x64, 0x00. As an example 48.5%
```

```
135     humidity would be 0x61, 0x00.*/
136
137     /* Temperature is stored as an unsigned 16 bits integer in 1/512
138     degrees; there is an offset: 0 maps to -25□C. The default value is
139     25□C = 0x64, 0x00. As an example 23.5% temperature would be
140     0x61, 0x00.
141     The internal algorithm uses these values (or default values if
142     not set by the application) to compensate for changes in
143     relative humidity and ambient temperature.*/
144
145     uint8_t hum_perc = humidity << 1;
146
147     float fractional = modf(temperature, &temperature);
148     uint16_t temp_high = (((uint16_t)temperature + 25) << 9);
149     uint16_t temp_low = ((uint16_t)(fractional / 0.001953125) & 0x1FF);
150
151     uint16_t temp_conv = (temp_high | temp_low);
152
153     uint8_t buf[] = {hum_perc, 0x00,
154       (uint8_t)((temp_conv >> 8) & 0xFF), (uint8_t)(temp_conv & 0xFF)};
155
156     this->write(CCS811_ENV_DATA, buf, 4);
157
158   }
159
160   /**************************************************************************/
161   /*!
162       @brief  calculate the temperature using the onboard NTC resistor.
163       @returns temperature as a double.
164   */
165   /**************************************************************************/
166   double Adafruit_CCS811::calculateTemperature()
167   {
168     uint8_t buf[4];
169     this->read(CCS811_NTC, buf, 4);
170
171     uint32_t vref = ((uint32_t)buf[0] << 8) | buf[1];
172     uint32_t vntc = ((uint32_t)buf[2] << 8) | buf[3];
173
174     //from ams ccs811 app note
175     uint32_t rntc = vntc * CCS811_REF_RESISTOR / vref;
176
177     double ntc_temp;
178     ntc_temp = log((double)rntc / CCS811_REF_RESISTOR); // 1
179     ntc_temp /= 3380; // 2
180     ntc_temp += 1.0 / (25 + 273.15); // 3
181     ntc_temp = 1.0 / ntc_temp; // 4
182     ntc_temp -= 273.15; // 5
183     return ntc_temp - _tempOffset;
184
185   }
186
187   /**************************************************************************/
188   /*!
189       @brief  set interrupt thresholds
190       @param low_med the level below which an interrupt will be triggered.
191       @param med_high the level above which the interrupt will ge triggered.
192       @param hysteresis optional histeresis level. Defaults to 50
193   */
194   /**************************************************************************/
195   void Adafruit_CCS811::setThresholds(uint16_t low_med, uint16_t med_high, uint8_t
      hysteresis)
196   {
197     uint8_t buf[] = {(uint8_t)((low_med >> 8) & 0xF), (uint8_t)(low_med & 0xF),
198     (uint8_t)((med_high >> 8) & 0xF), (uint8_t)(med_high & 0xF), hysteresis};
199
200     this->write(CCS811_THRESHOLDS, buf, 5);
201   }
202
```

```
203  /**************************************************************************/
204  /*!
205      @brief  trigger a software reset of the device
206  */
207  /**************************************************************************/
208  void Adafruit_CCS811::SWReset()
209  {
210    //reset sequence from the datasheet
211    uint8_t seq[] = {0x11, 0xE5, 0x72, 0x8A};
212    this->write(CCS811_SW_RESET, seq, 4);
213  }
214
215  /**************************************************************************/
216  /*!
217      @brief   read the status register and store any errors.
218      @returns the error bits from the status register of the device.
219  */
220  /**************************************************************************/
221  bool Adafruit_CCS811::checkError()
222  {
223    _status.set(read8(CCS811_STATUS));
224    return _status.ERROR;
225  }
226
227  /**************************************************************************/
228  /*!
229      @brief  write one byte of data to the specified register
230      @param  reg the register to write to
231      @param  value the value to write
232  */
233  /**************************************************************************/
234  void Adafruit_CCS811::write8(byte reg, byte value)
235  {
236    this->write(reg, &value, 1);
237  }
238
239  /**************************************************************************/
240  /*!
241      @brief  read one byte of data from the specified register
242      @param  reg the register to read
243      @returns one byte of register data
244  */
245  /**************************************************************************/
246  uint8_t Adafruit_CCS811::read8(byte reg)
247  {
248    uint8_t ret;
249    this->read(reg, &ret, 1);
250
251    return ret;
252  }
253
254  void Adafruit_CCS811::_i2c_init()
255  {
256    Wire.begin();
257  }
258
259  void Adafruit_CCS811::read(uint8_t reg, uint8_t *buf, uint8_t num)
260  {
261    uint8_t value;
262    uint8_t pos = 0;
263
264    //on arduino we need to read in 32 byte chunks
265    while(pos < num){
266
267      uint8_t read_now = min((uint8_t)32, (uint8_t)(num - pos));
268      Wire.beginTransmission((uint8_t)_i2caddr);
269      Wire.write((uint8_t)reg + pos);
270      Wire.endTransmission();
271      Wire.requestFrom((uint8_t)_i2caddr, read_now);
```

```
272
273         for(int i=0; i<read_now; i++){
274           buf[pos] = Wire.read();
275           pos++;
276         }
277       }
278     }
279
280     void Adafruit_CCS811::write(uint8_t reg, uint8_t *buf, uint8_t num)
281     {
282       Wire.beginTransmission((uint8_t)_i2caddr);
283       Wire.write((uint8_t)reg);
284       Wire.write((uint8_t *)buf, num);
285       Wire.endTransmission();
286     }
287
288     /*
289      * Part 2: code written by team 26
290      * This code was written by Team 26
291      * This code is properly commented
292      */
293
294     bool Adafruit_CCS811::start_voc(void) {
295         /*
296          * Start voc sensor using the library's begin() function
297          * If sensor is started, calibrate temperature
298          */
299         Serial.println("Trying to start VOC Sensor...");
300         if(!begin()){
301             Serial.println("Failed to start CC2821 VOC sensor! Wiring is likely incorrect.");
302             return false;
303         }
304         else {
305             Serial.println("Successfully started VOC Sensor!");
306             return true;
307         }
308     }
309
310     void Adafruit_CCS811::run_voc(void) {
311         /*
312          * Run the VOC sensor
313          * Take measurements until enough measurements have been taken to calculate the
          average
314          * use read_voc() to read from sensor
315          */
316         is_average_taken = false;
317         read_count = 1;
318         while(is_average_taken == false) {read_voc();}
319     }
320
321     void Adafruit_CCS811::read_voc(void) {
322         /*
323          * Read values from voc sensor
324          * IF data is read and max read count has not been exceed
325          * THEN fill_buffer and print_readings and read_count ++
326          * calculate_average_reading
327          * print_average_reading
328          */
329         if(available()){
330             float temp = calculateTemperature();
331             if(!readData() && read_count <= MAX_READ_COUNT){
332                 fill_buffer();
333                 print_readings();
334                 read_count += 1;
335             } else{Serial.println("ERROR!");}
336         }
337         calculate_average_reading();
338         print_average_reading();
339     }
```

```
340
341 void Adafruit_CCS811::fill_buffer(void) {
342     /*
343      * add new values to buffers
344      */
345     eCO2_buf[read_count-1] = geteCO2();
346     TVOC_buf[read_count-1] = getTVOC();
347 }
348
349 void Adafruit_CCS811::print_readings(void) {
350     /*
351      * Print readings
352      */
353     Serial.print("Reading #:");
354     Serial.print(read_count);
355     Serial.print(", CO2: ");
356     Serial.print(geteCO2());
357     Serial.print("ppm, TVOC: ");
358     Serial.print(getTVOC());
359     Serial.println("ppb");
360 }
361
362 void Adafruit_CCS811::calculate_average_reading(void) {
363     /*
364      * Calculate the average reading if enough readings have been taken
365      */
366     if(read_count > MAX_READ_COUNT) {
367         eCO2_ave = 0;
368         TVOC_ave = 0;
369         for(int k = 0; k < MAX_READ_COUNT; k++) {
370             eCO2_ave += eCO2_buf[k];
371             TVOC_ave += TVOC_buf[k];
372         }
373         eCO2_ave = eCO2_ave / MAX_READ_COUNT;
374         TVOC_ave = TVOC_ave / MAX_READ_COUNT;
375
376         read_count = 1;
377         is_average_taken = true;
378     }
379 }
380
381 void Adafruit_CCS811::print_average_reading(void) {
382     /*
383     * print average reading values
384     */
385     if(is_average_taken) {
386         Serial.println("-----------------------");
387         Serial.println("VOC Sensor Average Readings:");
388         Serial.println("-----------------------");
389         Serial.print("CCS eCO2 Average: ");
390         Serial.println(eCO2_ave);
391         Serial.print("CCS TVOC Average: ");
392         Serial.println(TVOC_ave);
393     }
394 }
395
396 // Getter functions for VOC parameters
397 float Adafruit_CCS811::get_eCO2_ave(void) {return eCO2_ave;}
398 float Adafruit_CCS811::get_TVOC_ave(void) {return TVOC_ave;}
399
```

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.

```
/*
 * This is the .h file for the SHT35D Tempearture
 * and relative humidity sensor.
 */

#ifndef SHT35D
#define SHT35D
#define MAX_READ_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 {
```

```
139  /*
140   * Class definition for ClosedCube_SHT31D
141   */
142  public:
143      ClosedCube_SHT31D();
144
145      bool start_sht(void);
146      void run_sht(void);
147      float get_t_ave(void);
148      float get_rh_ave(void);
149
150
151      SHT31D_ErrorCode begin(uint8_t address);
152      SHT31D_ErrorCode clearAll();
153      SHT31D_RegisterStatus readStatusRegister();
154
155      SHT31D_ErrorCode heaterEnable();
156      SHT31D_ErrorCode heaterDisable();
157
158      SHT31D_ErrorCode softReset();
159      SHT31D_ErrorCode generalCallReset();
160
161      SHT31D_ErrorCode artEnable();
162
163      uint32_t readSerialNumber();
164
165      SHT31D printResult(String text, SHT31D result);
166      SHT31D readTempAndHumidity(SHT31D_Repeatability repeatability, SHT31D_Mode mode,
         uint8_t timeout);
167      SHT31D readTempAndHumidityClockStretch(SHT31D_Repeatability repeatability);
168      SHT31D readTempAndHumidityPolling(SHT31D_Repeatability repeatability, uint8_t
         timeout);
169
170      SHT31D_ErrorCode periodicStart(SHT31D_Repeatability repeatability, SHT31D_Frequency
         frequency);
171      SHT31D periodicFetchData();
172      SHT31D_ErrorCode periodicStop();
173
174      SHT31D_ErrorCode writeAlertHigh(float temperatureSet, float temperatureClear, float
         humiditySet, float humidityClear);
175      SHT31D readAlertHighSet();
176      SHT31D readAlertHighClear();
177
178      SHT31D_ErrorCode writeAlertLow(float temperatureClear, float temperatureSet, float
         humidityClear, float humiditySet);
179      SHT31D readAlertLowSet();
180      SHT31D readAlertLowClear();
181
182  private:
183      float t_buf[MAX_READ_COUNT];
184      float rh_buf[MAX_READ_COUNT];
185      bool is_average_taken;
186      int read_count;
187      float t_average;
188      float rh_average;
189
190      SHT31D save_to_buffer(SHT31D result);
191      SHT31D read_sht(void);
192      void calculate_average(void);
193
194      uint8_t _address;
195      SHT31D_RegisterStatus _status;
196
197      SHT31D_ErrorCode writeCommand(SHT31D_Commands command);
198      SHT31D_ErrorCode writeAlertData(SHT31D_Commands command, float temperature, float
         humidity);
199
200      uint8_t checkCrc(uint8_t data[], uint8_t checksum);
201      uint8_t calculateCrc(uint8_t data[]);
```

```
202
203         float calculateHumidity(uint16_t rawValue);
204         float calculateTemperature(uint16_t rawValue);
205
206         uint16_t calculateRawHumidity(float value);
207         uint16_t calculateRaWTemperature(float value);
208
209         SHT31D readTemperatureAndHumidity();
210         SHT31D readAlertData(SHT31D_Commands command);
211         SHT31D_ErrorCode read(uint16_t* data, uint8_t numOfPair);
212
213         SHT31D returnError(SHT31D_ErrorCode command);
214     };
215
216     #endif
217
```

```
/*
 * 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:
```

```
69          switch (frequency)
70          {
71          case SHT3XD_FREQUENCY_HZ5:
72              error = writeCommand(SHT3XD_CMD_PERIODIC_HALF_M);
73              break;
74          case SHT3XD_FREQUENCY_1HZ:
75              error = writeCommand(SHT3XD_CMD_PERIODIC_1_M);
76              break;
77          case SHT3XD_FREQUENCY_2HZ:
78              error = writeCommand(SHT3XD_CMD_PERIODIC_2_M);
79              break;
80          case SHT3XD_FREQUENCY_4HZ:
81              error = writeCommand(SHT3XD_CMD_PERIODIC_4_M);
82              break;
83          case SHT3XD_FREQUENCY_10HZ:
84              error = writeCommand(SHT3XD_CMD_PERIODIC_10_M);
85              break;
86          default:
87              error = SHT3XD_PARAM_WRONG_FREQUENCY;
88              break;
89          }
90          break;
91
92      case SHT3XD_REPEATABILITY_HIGH:
93          switch (frequency)
94          {
95          case SHT3XD_FREQUENCY_HZ5:
96              error = writeCommand(SHT3XD_CMD_PERIODIC_HALF_H);
97              break;
98          case SHT3XD_FREQUENCY_1HZ:
99              error = writeCommand(SHT3XD_CMD_PERIODIC_1_H);
100             break;
101         case SHT3XD_FREQUENCY_2HZ:
102             error = writeCommand(SHT3XD_CMD_PERIODIC_2_H);
103             break;
104         case SHT3XD_FREQUENCY_4HZ:
105             error = writeCommand(SHT3XD_CMD_PERIODIC_4_H);
106             break;
107         case SHT3XD_FREQUENCY_10HZ:
108             error = writeCommand(SHT3XD_CMD_PERIODIC_10_H);
109             break;
110         default:
111             error = SHT3XD_PARAM_WRONG_FREQUENCY;
112             break;
113         }
114         break;
115
116     default:
117         error = SHT3XD_PARAM_WRONG_REPEATABILITY;
118         break;
119     }
120     delay(100);
121     return error;
122 }
123
124 SHT31D ClosedCube_SHT31D::readTempAndHumidity(SHT31D_Repeatability repeatability,
    SHT31D_Mode mode, uint8_t timeout)
125 {
126     SHT31D result;
127
128     switch (mode) {
129     case SHT3XD_MODE_CLOCK_STRETCH:
130         result = readTempAndHumidityClockStretch(repeatability);
131         break;
132     case SHT3XD_MODE_POLLING:
133         result = readTempAndHumidityPolling(repeatability, timeout);
134         break;
135     default:
136         result = returnError(SHT3XD_PARAM_WRONG_MODE);
```

```
137             break;
138         }
139         return result;
140     }
141
142
143     SHT31D ClosedCube_SHT31D::readTempAndHumidityClockStretch(SHT31D_Repeatability
        repeatability)
144     {
145         SHT31D_ErrorCode error = SHT3XD_NO_ERROR;
146         SHT31D_Commands command;
147
148         switch (repeatability)
149         {
150         case SHT3XD_REPEATABILITY_LOW:
151             error = writeCommand(SHT3XD_CMD_CLOCK_STRETCH_L);
152             break;
153         case SHT3XD_REPEATABILITY_MEDIUM:
154             error = writeCommand(SHT3XD_CMD_CLOCK_STRETCH_M);
155             break;
156         case SHT3XD_REPEATABILITY_HIGH:
157             error = writeCommand(SHT3XD_CMD_CLOCK_STRETCH_H);
158             break;
159         default:
160             error = SHT3XD_PARAM_WRONG_REPEATABILITY;
161             break;
162         }
163
164         delay(50);
165
166         if (error == SHT3XD_NO_ERROR) {
167             return readTemperatureAndHumidity();
168         } else {
169             return returnError(error);
170         }
171
172     }
173
174
175     SHT31D ClosedCube_SHT31D::readTempAndHumidityPolling(SHT31D_Repeatability repeatability,
         uint8_t timeout)
176     {
177         SHT31D_ErrorCode error = SHT3XD_NO_ERROR;
178         SHT31D_Commands command;
179
180         switch (repeatability)
181         {
182         case SHT3XD_REPEATABILITY_LOW:
183             error = writeCommand(SHT3XD_CMD_POLLING_L);
184             break;
185         case SHT3XD_REPEATABILITY_MEDIUM:
186             error = writeCommand(SHT3XD_CMD_POLLING_M);
187             break;
188         case SHT3XD_REPEATABILITY_HIGH:
189             error = writeCommand(SHT3XD_CMD_POLLING_H);
190             break;
191         default:
192             error = SHT3XD_PARAM_WRONG_REPEATABILITY;
193             break;
194         }
195
196         delay(50);
197
198         if (error == SHT3XD_NO_ERROR) {
199             return readTemperatureAndHumidity();
200         } else {
201             return returnError(error);
202         }
203
```

```
}

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());
    }
```

```
266
267         return error;
268     }
269
270
271     SHT31D_ErrorCode ClosedCube_SHT31D::writeCommand(SHT31D_Commands command)
272     {
273         Wire.beginTransmission(_address);
274         Wire.write(command >> 8);
275         Wire.write(command & 0xFF);
276         return (SHT31D_ErrorCode)(-10 * Wire.endTransmission());
277     }
278
279     SHT31D_ErrorCode ClosedCube_SHT31D::softReset() {
280         return writeCommand(SHT3XD_CMD_SOFT_RESET);
281     }
282
283     SHT31D_ErrorCode ClosedCube_SHT31D::generalCallReset() {
284         Wire.beginTransmission(0x0);
285         Wire.write(0x06);
286         return (SHT31D_ErrorCode)(-10 * Wire.endTransmission());
287     }
288
289     SHT31D_ErrorCode ClosedCube_SHT31D::heaterEnable() {
290         return writeCommand(SHT3XD_CMD_HEATER_ENABLE);
291     }
292
293     SHT31D_ErrorCode ClosedCube_SHT31D::heaterDisable() {
294         return writeCommand(SHT3XD_CMD_HEATER_DISABLE);
295     }
296
297     SHT31D_ErrorCode ClosedCube_SHT31D::artEnable() {
298         return writeCommand(SHT3XD_CMD_ART);
299     }
300
301
302     uint32_t ClosedCube_SHT31D::readSerialNumber()
303     {
304         uint32_t result = SHT3XD_NO_ERROR;
305         uint16_t buf[2];
306
307         if (writeCommand(SHT3XD_CMD_READ_SERIAL_NUMBER) == SHT3XD_NO_ERROR) {
308             if (read(buf, 2) == SHT3XD_NO_ERROR) {
309               result = (buf[0] << 16) | buf[1];
310             }
311         }
312
313         return result;
314     }
315
316     SHT31D_RegisterStatus ClosedCube_SHT31D::readStatusRegister()
317     {
318         SHT31D_RegisterStatus result;
319
320         SHT31D_ErrorCode error = writeCommand(SHT3XD_CMD_READ_STATUS);
321         if (error == SHT3XD_NO_ERROR)
322             error = read(&result.rawData, 1);
323
324         return result;
325     }
326
327     SHT31D_ErrorCode ClosedCube_SHT31D::clearAll() {
328         return writeCommand(SHT3XD_CMD_CLEAR_STATUS);
329     }
330
331
332     SHT31D ClosedCube_SHT31D::readTemperatureAndHumidity()
333     {
334         SHT31D result;
```

```
335
336         result.t = 0;
337         result.rh = 0;
338
339         SHT31D_ErrorCode error;
340         uint16_t buf[2];
341
342         if (error == SHT3XD_NO_ERROR)
343             error = read(buf, 2);
344
345         if (error == SHT3XD_NO_ERROR) {
346             result.t = calculateTemperature(buf[0]);
347             result.rh = calculateHumidity(buf[1]);
348         }
349         result.error = error;
350
351         return result;
352     }
353
354     SHT31D ClosedCube_SHT31D::readAlertData(SHT31D_Commands command)
355     {
356         SHT31D result;
357
358         result.t = 0;
359         result.rh = 0;
360
361         SHT31D_ErrorCode error;
362         uint16_t buf[1];
363
364         error = writeCommand(command);
365
366         if (error == SHT3XD_NO_ERROR)
367             error = read(buf, 1);
368
369         if (error == SHT3XD_NO_ERROR) {
370             result.rh = calculateHumidity(buf[0] << 7);
371             result.t = calculateTemperature(buf[0] & 0xFE00);
372         }
373
374         result.error = error;
375
376         return result;
377     }
378
379     SHT31D_ErrorCode ClosedCube_SHT31D::read(uint16_t* data, uint8_t numOfPair)
380     {
381         uint8_t checksum;
382         char buf[2];
383         uint8_t buffer[2];
384
385
386         const uint8_t numOfBytes = numOfPair * 3;
387         Wire.requestFrom(_address, numOfBytes);
388
389         int counter = 0;
390
391         for (counter = 0; counter < numOfPair; counter++) {
392             Wire.readBytes(buf, 2);
393             checksum = Wire.read();
394
395             for (int i = 0; i < 2; i++){buffer[i] = uint8_t(buf[i]);}
396
397
398             if (checkCrc(buffer, checksum) != 0)
399                 return SHT3XD_CRC_ERROR;
400
401             data[counter] = (buf[0] << 8) | buf[1];
402         }
403
```

```
404         return SHT3XD_NO_ERROR;
405     }
406
407
408     uint8_t ClosedCube_SHT31D::checkCrc(uint8_t data[], uint8_t checksum)
409     {
410         return calculateCrc(data) != checksum;
411     }
412
413     float ClosedCube_SHT31D::calculateTemperature(uint16_t rawValue)
414     {
415         return 175.0f * (float)rawValue / 65535.0f - 45.0f;
416     }
417
418
419     float ClosedCube_SHT31D::calculateHumidity(uint16_t rawValue)
420     {
421         return 100.0f * rawValue / 65535.0f;
422     }
423
424     uint16_t ClosedCube_SHT31D::calculateRaWTemperature(float value)
425     {
426         return (value + 45.0f) / 175.0f * 65535.0f;
427     }
428
429     uint16_t ClosedCube_SHT31D::calculateRawHumidity(float value)
430     {
431         return value / 100.0f * 65535.0f;
432     }
433
434     uint8_t ClosedCube_SHT31D::calculateCrc(uint8_t data[])
435     {
436         uint8_t bit;
437         uint8_t crc = 0xFF;
438         uint8_t dataCounter = 0;
439
440         for (; dataCounter < 2; dataCounter++) {
441             crc ^= (data[dataCounter]);
442             for (bit = 8; bit > 0; --bit) {
443                 if (crc & 0x80){crc = (crc << 1) ^ 0x131;}
444                 else {crc = (crc << 1);}
445             }
446         }
447
448         return crc;
449     }
450
451     SHT31D ClosedCube_SHT31D::returnError(SHT31D_ErrorCode error) {
452         SHT31D result;
453         result.t = 0;
454         result.rh = 0;
455         result.error = error;
456         return result;
457     }
458
459     /*
460      * Part 2: Code Written by team 26
461      * Team 26 understands this code,
462      * therefore it is properly commented.
463      */
464     bool ClosedCube_SHT31D::start_sht(void) {
465         /*
466          * Start sequence for SHT35D
467          * Return true: sensor was succesfully started
468          * Return false: sensor was not started
469          * Try to read from sensor
470          * If no error, return true
471          * Else return false
472          */
```

```
473       Serial.println("Trying to start SHT sensor...");
474       delay(500);
475       begin(ADDR_SHT); // I2C address: 0x44 or 0x45
476       Serial.print("Serial #");
477       Serial.println(readSerialNumber());
478       delay(500);
479
480       if (periodicStart(SHT3XD_REPEATABILITY_HIGH, SHT3XD_FREQUENCY_10HZ) !=
          SHT3XD_NO_ERROR) {
481           Serial.println("[ERROR] Cannot start periodic mode");
482           return false;
483       }
484       else {
485           Serial.println("Successfully started SHT sensor!");
486           return true;
487       }
488   }
489
490   void ClosedCube_SHT31D::run_sht(void) {
491       /*
492        * Run SHT sensor
493        * start read_count from 1
494        * is_average_taken is false until average is taken
495        * take reading from sht until enough values are read to take an average
496        */
497       is_average_taken = false;
498       read_count = 1;
499       while(!is_average_taken) {read_sht();}
500   }
501
502   SHT31D ClosedCube_SHT31D::read_sht(void) {
503       /*
504        * Read from SHT35D, and assign values to my_result
505        * print results
506        * save results to buffer
507        * calculate average if enough values have been read
508        */
509       SHT31D my_result = periodicFetchData();
510       printResult("Periodic Mode", my_result);
511       save_to_buffer(my_result);
512       calculate_average();
513       delay(500);
514   }
515
516   SHT31D ClosedCube_SHT31D::save_to_buffer(SHT31D result) {
517       /*
518        * Save current t and rh readings to their respective buffers
519        *
520        * if no error and the number of readings is less than the max
521        * then save values
522        *
523        * else -> report error, do not save any values
524        */
525       if (result.error == SHT3XD_NO_ERROR && read_count <= MAX_READ_COUNT) {
526           t_buf[read_count - 1] = result.t;
527           rh_buf[read_count - 1] = result.rh;
528           read_count++;
529       }
530       else {
531           Serial.print("[ERROR] Code #");
532           Serial.println(result.error);
533       }
534   }
535
536   SHT31D ClosedCube_SHT31D::printResult(String text, SHT31D result) {
537       /*
538        * Prints current reading if no error and not exceeded max count
539        * else print error message
540        */
```

```
    if (result.error == SHT3XD_NO_ERROR && read_count <= MAX_READ_COUNT) {
        Serial.print(text);
        Serial.print(" Reading #");
        Serial.print(read_count);
        Serial.print(": T=");
        Serial.print(result.t);
        Serial.print("C, RH=");
        Serial.print(result.rh);
        Serial.println("%");
    }
    else {
        Serial.print(text);
        Serial.print(": [ERROR] Code #");
        Serial.println(result.error);
    }
}

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 ADDR_MRT 0x40
#define DEFAULT_AVERAGE 128
#include <Arduino.h>

class ClosedCube_Si7051 {
public:
    ClosedCube_Si7051();

    float readT(); // short-cut for readTemperature
    float 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;
};

#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
 */

bool ClosedCube_Si7051::start_mrt(void) {
    /*
     * Start globe thermometer sensor
     *
```

```
70        * The code will read a value of 128 or greater
71        * if the sensor is broken or disconnected
72        *
73        * The start sequence returns false (sensor does not work)
74        * if a value of 128 is read
75        *
76        * If the value is less than 128, it returns true
77        * (sensor works)
78        *
79        * The code retrieved from the on line library should be improved
80        * to fix this.
81        */
82       begin(ADDR_MRT);
83       delay(500);
84       if(run_mrt() > DEFAULT_AVERAGE) {
85           Serial.println("Failed to start MRT sensor!");
86           return false;
87       } else {
88           Serial.println("Successfully started MRT sensor!");
89           return true;
90       }
91   }
92
93   float ClosedCube_Si7051::run_mrt(void) {
94       /*
95        * Takes globe thermometer measurements until read_count
96        * is exceeded.
97        * Once read_count is exceeded, the average is taken.
98        */
99       read_count = 1;
100
101      while(read_count <= MAX_READ_COUNT) {
102          T_buf[read_count - 1] = readTemperature();
103          Serial.print("Reading #");
104          Serial.print(read_count);
105          Serial.print(": Tg is: ");
106          Serial.println(T_buf[read_count - 1]);
107          read_count ++;
108          delay(250);
109      }
110      if(read_count > MAX_READ_COUNT) {
111          T_ave = 0;
112          for(int k = 0; k < MAX_READ_COUNT; k++) {
113              T_ave = T_ave + T_buf[k];
114          }
115          T_ave = T_ave / MAX_READ_COUNT;
116          Serial.print("Average Tg is: ");
117          Serial.println(T_ave);
118          return(T_ave);
119      }
120  }
121
122  // Getter function for Globe Thermometer average value
123  float ClosedCube_Si7051::get_MRT_ave(void) {return T_ave;}
```