

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

```

1  /*
2  * Script_all.ino
3  * This script runs the sensor package
4  * Uses objects for each of the sensors
5  * Prints information to Serial screen
6  * Publishes data to ThingSpeak
7  */
8
9  #include "CALCULATE_MRT.h"
10 #include "MHZ19.h"
11 #include "CCS821.h"
12 #include "SHT35D.h"
13 #include "MRT.h"
14 #include "PM.h"
15 #include "Time.h"
16 #include <Wire.h>
17
18 // create instances of objects
19 PM_7003 myPM;
20 ClosedCube_Si7051 myMRT;
21 ClosedCube_SHT31D mySHT;
22 Adafruit_CCS811 myVOC;
23 MHZ19 myCO2;
24 mrt_and_ot my_MRT_OT;
25
26 /*
27 * Boolean expressions
28 * start_xxx indicate whether sensor has been read from properly
29 * read_from_xxx indicate whether or not to read from sensor_xxx (changes throughout
   code)
30 * finished_xxx indicates whether done reading from a sensor (read a good average)
31 */
32 bool start_co2 = false;
33 bool start_voc = false;
34 bool start_sht = false;
35 bool start_pm = false;
36 bool start_mrt = false;
37
38 bool read_from_co2 = true;
39 bool read_from_pm = false;
40
41 bool finished_co2 = false;
42 bool finished_pm = false;
43 bool finished_other_sensors = false;
44 bool finished_mrt_ot = false;
45 bool finished_voc = false;
46
47 // average reading values
48 int co2_ave = -1;
49 float sht_rh_ave = -1;
50 float sht_t_ave = -1;
51 float voc_eCO2_ave = -1;
52 float voc_TVOC_ave = -1;
53 int pm_ave = -1;
54 float T_g = -1;
55 float T_a = -1;
56 float T_mrt = -1;
57 float T_ot = -1;
58
59 bool publish_data = true; // should we publish data?
60
61 // pin numbers for pm and co2 sensors
62 int pm_transistor_control = A4;
63 int pm_tx_transistor_control = A5;
64 int co2_transistor_control = A3;
65
66 void setup() {
67     /*
68     * Start Serial and Wire connections

```

```

69     * Initialize transistor control for CO2 and PM
70     * Turn CO2 sensor on (make_sensor_read())
71     * Test all I2C sensors (MRT, SHT, VOC)
72     * Stop and wait for 30 seconds (warm-up)
73     */
74     Serial.begin(9600);
75     Wire.begin();
76     Serial.println("Initializing");
77
78     myCO2.set_transistor(co2_transistor_control);
79     myPM.set_transistor(pm_transistor_control, pm_tx_transistor_control);
80
81     myCO2.make_sensor_read();
82
83     start_mrt = myMRT.start_mrt();
84     Serial.println("-----");
85
86     start_sht = mySHT.start_sht();
87     Serial.println("-----");
88
89     start_voc = myVOC.start_voc();
90     Serial.println("-----");
91     Serial.println("30 second delay");
92     Serial.println("-----");
93     delay(30000);
94
95 }
96
97 void loop() {
98     /*
99     * Wait for CO2 sensor to warm-up (PM sensor is off)
100    * Read from MRT, SHT, and VOC sensors while CO2 sensor warms-up
101    * After CO2 sensor warms-up, read from CO2 sensor and save average reading
102    * Save average value from MRT, SHT, and VOC sensors
103    * Turn off CO2 sensor, turn on PM sensor
104    * Read from MRT, SHT, and VOC sensors while PM sensor warms-up
105    * After PM sensor warms-up, read from PM sensor and push all data to ThingSpeak
106    * Repeat
107    */
108
109    // Decide which of CO2 or PM sensor to read from
110    if(read_from_co2) {
111        start_co2 = myCO2.make_sensor_read();
112        start_pm = false;
113
114        if(start_co2) {
115            read_from_co2 = false;
116            read_from_pm = true;
117            finished_co2 = true;
118        }
119    }
120    else if(read_from_pm) {
121        start_pm = myPM.make_sensor_read();
122        start_co2 = false;
123
124        if(start_pm) {
125            read_from_pm = false;
126            read_from_co2 = true;
127            finished_pm = true;
128        }
129    }
130
131    start_mrt = myMRT.run_mrt(); //read from MRT sensor
132
133    // Read from SHT sensor, or restart SHT sensor
134    if(start_sht) {
135        Serial.println("Reading from SHT Sensor");
136        Serial.println("-----");
137        start_sht = mySHT.run_sht();

```

```

138     Serial.println("-----");
139 }
140 else if(!start_sht) {
141     Serial.println("-----");
142     Serial.println("Not reading from SHT Sensor");
143     Serial.println("-----");
144     Serial.println("Try to start SHT");
145     start_sht = mySHT.start_sht();
146     Serial.println("-----");
147 }
148 // Read from VOC sensor, or restart VOC sensor
149 if(start_voc) {
150     Serial.println("Reading from VOC Sensor");
151     Serial.println("-----");
152     start_voc = myVOC.run_voc();
153     Serial.println("-----");
154 }
155 else if(!start_voc) {
156     start_voc = myVOC.start_voc();
157     Serial.println("Reading from VOC Sensor");
158     Serial.println("-----");
159     start_voc = myVOC.run_voc();
160     Serial.println("-----");
161 }
162
163 // If done reading from CO2 sensor, save CO2, MRT, SHT, and VOC readings
164 if(finished_co2 && !finished_other_sensors) {
165     finished_other_sensors = true;
166
167     if(!finished_mrt_ot) {
168         if(start_mrt && start_sht){
169             T_g = myMRT.get_MRT_ave();
170             T_a = mySHT.get_t_ave();
171             sht_rh_ave = mySHT.get_rh_ave();
172             my_MRT_OT.calculate_mrt_and_ot(T_g, T_a);
173             T_mrt = my_MRT_OT.get_mrt();
174             T_ot = my_MRT_OT.get_ot();
175             finished_mrt_ot = true;
176         }
177         else if(start_mrt && !start_sht) {
178             T_g = myMRT.get_MRT_ave();
179             T_a = -1;
180             sht_rh_ave = -1;
181             T_mrt = -1;
182             T_ot = -1;
183         }
184         else if(!start_mrt && start_sht) {
185             T_g = -1;
186             T_a = mySHT.get_t_ave();
187             sht_rh_ave = mySHT.get_rh_ave();
188             T_mrt = -1;
189             T_ot = -1;
190         }
191         else {
192             T_g = -1;
193             T_a = -1;
194             sht_rh_ave = -1;
195             T_mrt = -1;
196             T_ot = -1;
197         }
198     }
199
200     if(start_voc && !finished_voc){
201         voc_eCO2_ave = myVOC.get_eCO2_ave();
202         voc_TVOC_ave = myVOC.get_TVOC_ave();
203         finished_voc = true;
204     } else {
205         voc_eCO2_ave = -1;
206         voc_TVOC_ave = -1;

```

```

207     }
208
209     co2_ave = myCO2.get_co2_ave();
210
211     if(finished_mrt_ot && finished_voc) {
212         finished_other_sensors = true;
213     }
214 }
215
216 // If done reading from PM and CO2 sensors, save PM reading and push to ThingSpeak
217 if(finished_co2 && finished_pm) {
218     pm_ave = myPM.get_pm_ave();
219     finished_co2 = false;
220     finished_pm = false;
221     finished_mrt_ot = false;
222     finished_voc = false;
223     finished_other_sensors = false;
224
225     if(publish_data) {
226         char data[1000];
227         sprintf(data, "{ \"Mean Radiant Temperature\": \"%3.2f\", \"Operating
228         Temperature\": \"%3.2f\", \"CO2 Concentration\": \"%i\", \"eCO2\":
229         \"%4.2f\", \"TVOC\": \"%4.2f\", \"PM 2_5\": \"%i\", \"Air Temperature\":
230         \"%3.2f\", \"Relative Humidity of Air\": \"%3.2f\"} ", T_mrt, T_ot, co2_ave,
231         voc_eCO2_ave, voc_TVOC_ave, pm_ave, T_a, sht_rh_ave);
232         Serial.println("-----");
233         Serial.print("Data:");
234         Serial.println(data);
235         Serial.println("-----");
236
237         Particle.publish("IEQ Final Prototype", data, PRIVATE);
238
239         myCO2.reset_co2_ave();
240         myPM.reset_pm_ave();
241     }
242 }

```

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.

```

1  /*
2  * This is the .h file for calculating MRT and OT
3  * This code was written entirely by Team 26
4  * using formulas found in Literature.
5  */
6  #ifndef CALCULATE_MRT_H
7  #define CALCULATE_MRT_H
8
9  #if ARDUINO >= 100
10     #include "Arduino.h"
11 #else
12     #include "WProgram.h"
13 #endif
14
15 class mrt_and_ot {
16     public:
17         mrt_and_ot(void);
18
19         void calculate_mrt_and_ot(float T_g, float T_a);
20         float get_mrt(void);
21         float get_ot(void);
22
23     private:
24         float calculate_convection_coefficient(float T_g, float T_a);
25         float h;
26         float T_mrt;
27         float T_ot;
28         float T_a;
29         float T_g;
30         float convection_coefficient;
31
32         const float epsilon = 0.94;
33         const float diameter = 0.04;
34         const float diameter_to_power = pow(diameter, 0.4);
35         const float kelvin_conversion = 273.15;
36 };
37 #endif
38

```



```

1  /*
2  * This is the .cpp file for calculating MRT and OT
3  * This code was written entirely by Team 26
4  * using formulas found in Literature.
5  */
6  #include "CALCULATE_MRT.h"
7
8  mrt_and_ot::mrt_and_ot(void)
9  {
10 }
11
12 float mrt_and_ot::calculate_convection_coefficient(float T_g, float T_a) {
13     /*
14      * Calculate convection coefficient using formula in Literature
15      */
16     h = abs(T_g - T_a) / diameter_to_power;
17     h = pow(h, 0.25);
18     return(1.4 * h);
19 }
20
21 void mrt_and_ot::calculate_mrt_and_ot(float T_g, float T_a) {
22     /*
23      * Calculate MRT and OT using formulas found in Literature
24      */
25     T_g = T_g + kelvin_conversion;
26     T_a = T_a + kelvin_conversion;
27     convection_coefficient = calculate_convection_coefficient(T_g, T_a);
28     T_mrt = convection_coefficient / epsilon * (T_g - T_a);
29     T_mrt = T_mrt + pow(T_g, 4);
30     T_mrt = pow(T_mrt, 0.25);
31     T_ot = 0.5 * (T_a + T_mrt);
32 }
33
34 // Getter functions for MRT and OT
35 float mrt_and_ot::get_mrt(void) {return(T_mrt);}
36 float mrt_and_ot::get_ot(void) {return(T_ot);}
37
38

```

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.

```

1  /*
2  * This is the .h file for the PMS7003 sensor
3  * This code was written exclusively by MECH 45X Team 26
4  */
5
6  #include <stdint.h>
7  #include "WProgram.h"
8  #include "Time.h"
9
10 #define LIB_PM_H
11 #define FIRST_BYTE 0x42
12 #define SECOND_BYTE 0x4D
13 #define SENSOR_OUTPUT_PIN A0
14 #define MAX_FRAME_LENGTH 64
15
16 #define START_TIME 6000
17 #define SAMPLING_TIME 280
18 #define SLEEP_TIME 912
19 #define MAX_READ_COUNT 5
20 #define MAX_FRAME_SYNC_COUNT 40
21 #define PMS_START_UP_TIME 120
22 #define MAX_FUNCTION_CALL_COUNT 1
23
24 class PM_7003 {
25 public:
26     PM_7003();
27     virtual ~PM_7003();
28     int get_pm_ave(void);
29     void set_transistor(int ground_pin, int tx_pin);
30     bool make_sensor_read(void);
31     void calibrate_sensor(void);
32     void reset_pm_ave(void);
33
34 private:
35     int current_byte;
36     bool sync_state;
37     char print_buffer[256];
38     uint16_t byte_sum;
39     int drain;
40     uint16_t current_data;
41     int pm_ground_control;
42     int pm_tx_control;
43     char frame_buffer[MAX_FRAME_LENGTH];
44     int frame_count;
45     int frame_length;
46
47     bool debug = false;
48
49     int pm_avgpm2_5;
50     int pm2_5_buf[MAX_READ_COUNT];
51
52     bool done_reading;
53     int read_count;
54     int function_call_count;
55     int frame_sync_count;
56     bool first_time;
57
58     bool run_PM_sensor(void);
59     void drain_serial(void);
60     void frame_sync(void);
61     void read_sensor(void);
62     void data_switch(uint16_t current_data);
63     void print_messages(void);
64
65     //time
66     void begin_timer(void);
67     bool check_begin_reading(void);
68     time_t start_time;
69     time_t current_time;

```

```

70     time_t duration;
71
72
73     struct PMS7003data {
74         uint8_t start_frame[2];
75         uint16_t frame_length;
76         uint16_t concPM1_0_factory;
77         uint16_t concPM2_5_factory;
78         uint16_t concPM10_0_factory;
79         uint16_t concPM1_0_ambient;
80         uint16_t concPM2_5_ambient;
81         uint16_t concPM10_0_ambient;
82         uint16_t countPM0_3um;
83         uint16_t countPM0_5um;
84         uint16_t countPM1_0um;
85         uint16_t countPM2_5um;
86         uint16_t countPM5_0um;
87         uint16_t countPM10_0um;
88         uint8_t version;
89         uint8_t error;
90         uint16_t checksum;
91     } packetdata;
92 };
93

```

```

1  /*
2  * This is the .cpp file for the PMS7003 sensor
3  * This code was written exclusively by MECH 45X Team 26
4  */
5  #include "PM.h"
6
7  PM_7003::PM_7003() {
8      current_byte = 0;
9      packetdata.frame_length = MAX_FRAME_LENGTH;
10     frame_length = MAX_FRAME_LENGTH;
11 }
12
13 PM_7003::~PM_7003() {
14 }
15
16 int PM_7003::getpm(void) {
17     return pm_avgpm2_5;
18 }
19
20 bool PM_7003::run_PM_sensor(void) {
21     /*
22     * run the PM sensor
23     * Start serial connection
24     *
25     * drain_serial() and read_sensor() until enough values have been read
26     * to take the average
27     */
28     Serial1.begin(9600);
29     read_count = 1;
30     done_reading = false;
31     frame_sync_count = 0;
32     pm_avgpm2_5 = 0;
33     while(!done_reading && frame_sync_count < MAX_FRAME_SYNC_COUNT) {
34         drain_serial();
35         delay(500);
36         read_sensor();
37     }
38
39     Serial1.end();
40
41     if(done_reading) {
42         Serial.println("-----");
43         Serial.println("Done reading from PM sensor");
44         Serial.println("-----");
45         Serial.println(" ");
46         return true;
47     }
48     else if(!done_reading && frame_sync_count >= MAX_FRAME_SYNC_COUNT){return false;}
49 }
50
51 void PM_7003::drain_serial(void) {
52     /*
53     * Drains serial buffer if there are more than 32 entries
54     * Reads entries to drain serial buffer
55     */
56     if (Serial1.available() > 32) {
57         drain = Serial1.available();
58         Serial.println("-- Draining buffer: ");
59         Serial.println(Serial1.available(), DEC);
60         for (int drain_index = drain; drain_index > 0; drain_index--) {Serial1.read();}
61     }
62 }
63
64 void PM_7003::frame_sync(void) {
65     /*
66     * syncs frames for PM sensor
67     * checks that frames are being read in correct order
68     * exits when it confirms that frames are being read correctly
69     */

```

```

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 void PM_7003::read_sensor(void) {
109     /*
110     * Sync the frames
111     * read bytes and fill frame_buffer
112     * use data_switch to calculate different parameters
113     * print messages once all values have been read.
114     * done_reading = true if enough values have been read
115     */
116     frame_sync();
117
118     while(sync_state == true && Serial1.available() > 0) {
119         current_byte = Serial1.read();
120         frame_buffer[frame_count] = current_byte;
121         byte_sum = byte_sum + current_byte;
122         frame_count++;
123         uint16_t current_data = frame_buffer[frame_count-1]+(frame_buffer[frame_count-2]
124             ]<<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.
217             concPM10_0_factory);
218     sprintf(print_buffer, "%s%04d, %04d, %04d, ", print_buffer,
219             packetdata.concPM1_0_ambient, packetdata.concPM2_5_ambient, packetdata.
220             concPM10_0_ambient);
221     sprintf(print_buffer, "%s%04d, %04d, %04d, %04d, %04d, %04d, ", print_buffer,
222             packetdata.countPM0_3um, packetdata.countPM0_5um, packetdata.countPM1_0um,
223             packetdata.countPM2_5um, packetdata.countPM5_0um, packetdata.countPM10_0um);
224     sprintf(print_buffer, "%s%02d, %02d, ", print_buffer,
225             packetdata.version, packetdata.error);
226
227     pm2_5 = packetdata.countPM1_0um - packetdata.countPM2_5um + packetdata.countPM0_5um
228             - packetdata.countPM1_0um + packetdata.countPM0_3um - packetdata.countPM0_5um;
229     Serial.println(print_buffer);
230     Serial.println("-----");
231     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.

```

1  /*
2  * This is the .cpp file for the MH-Z19 CO2 Sensor
3  * This code was exclusively written by MECH 45X Team 26
4  */
5
6  #ifndef MHZ19_H
7  #define MHZ19_H
8  #define MHZ19_ZERO_BYTE 0xFF
9  #define MHZ19_FIRST_BYTE 0x86
10 #define MAX_FRAME_LEN 9
11 #define NUMBER_OF_VALUES 5
12 #define CO2_START_UP_TIME 210
13 #define MAX_FRAME_READ_COUNT 40
14 #define MAX_FUNCTION_CALL_COUNT 1
15 #include "WProgram.h"
16 #include "Time.h"
17
18
19 class MHZ19 {
20     public:
21         MHZ19();
22         virtual ~MHZ19();
23         int get_co2_reading(void);
24         int get_co2_ave(void);
25         void set_transistor(int pin);
26         bool make_sensor_read(void);
27         void calibrate_sensor(void);
28         void reset_co2_ave(void);
29
30     private:
31         char frame_buffer[MAX_FRAME_LEN];
32         const uint8_t mhz19_read_command[MAX_FRAME_LEN] = {0xFF,0x01,0x86,0x00,0x00,0x00,
33             ,0x00,0x00,0x79};;
34
35         bool debug = false;
36
37         bool sync_state;
38         bool does_sensor_work;
39         bool is_average_taken;
40         bool first_time;
41         int co2_transistor_control;
42
43         int frame_sync_count;
44         int frame_read_count;
45         int byte_sum;
46         int current_byte;
47         int drain;
48         int co2_ppm;
49         int co2_ppm_average;
50         int reading_count;
51         int function_call_count;
52         int mhz19_buffer[NUMBER_OF_VALUES];
53
54         bool run_sensor(void);
55         void frame_sync(void);
56         void read_sensor(void);
57         void serial_drain(void);
58         void fill_frame_buffer(void);
59         void add_to_ave_buf(void);
60         void print_current_reading(void);
61         void calculate_average_reading(void);
62         void print_average_reading(void);
63         void take_average(void);
64
65         //Timer
66         time_t start_time;
67         time_t current_time;
68         time_t duration;

```

```
69         void begin_timer(void);
70         bool check_begin_reading(void);
71     };
72
73 #endif /* MHZ19_H_ */
```

```

1  /*
2  * This is the .cpp file for the MH-Z19 CO2 Sensor
3  * This code was exclusively written by MECH 45X Team 26
4  */
5
6  #include "MHZ19.h"
7  #include "Time.h"
8
9  MHZ19::MHZ19() {
10     first_time = true;
11 }
12
13 MHZ19::~MHZ19() {
14 }
15
16 void MHZ19::set_transistor(int pin) {
17     /*
18      * Set transistor pin
19      * set pinMode for transistor pin
20      */
21     co2_transistor_control = pin;
22     pinMode(co2_transistor_control, OUTPUT);
23 }
24
25 void MHZ19::begin_timer(void) {
26     /*
27      * Turn transistor on
28      * Save time at which transistor is turned on
29      * Time is used for timing purposes
30      * change first_time to false
31      *
32      * first_time indicates whether or not timer has been started
33      * and transistor has been turned on
34      */
35     co2_ppm_average = -1;
36     digitalWrite(co2_transistor_control, HIGH);
37     start_time = now();
38     Serial.println("-----");
39     Serial.print("CO2 start time: ");
40     Serial.println(start_time);
41     Serial.println("-----");
42     first_time = false;
43 }
44
45 bool MHZ19::check_begin_reading(void) {
46     /*
47      * Check whether enough time has passed to begin reading
48      * return true if enough time has passed
49      * else false
50      */
51     current_time = now();
52     duration = current_time - start_time;
53     Serial.println("-----");
54     Serial.print("CO2 Duration: ");
55     Serial.println(duration);
56     Serial.println("-----");
57
58     if(duration >= CO2_START_UP_TIME) {
59         Serial.println("Three minutes have elapsed since starting CO2 sensor!");
60         return(true);
61     } else{return(false);}
62 }
63
64 bool MHZ19::make_sensor_read(void) {
65     /*
66      * turn transistor on and start timer if this hasn't already been done
67      * read from sensor if enough time has passed
68      * return true if enough measurements have been taken
69      * else false

```

```

70     */
71     if(first_time) {
72         function_call_count = 0;
73         begin_timer();
74         return(false);
75     }
76     else if(function_call_count < MAX_FUNCTION_CALL_COUNT) {
77         if(check_begin_reading()) {
78             Serial.println("-----");
79             Serial.print("Function Call Count: ");
80             Serial.println(function_call_count);
81             Serial.println("-----");
82             run_sensor();
83             function_call_count ++;
84         } else {return(false);}
85     }
86
87
88     if(function_call_count >= MAX_FUNCTION_CALL_COUNT) {
89         first_time = true;
90         digitalWrite(co2_transistor_control, LOW);
91         return(true);
92     } else{return(false);}
93 }
94
95 void MHZ19::calibrate_sensor(void) {
96     /*
97     * Turn sensor on and wait for warm-upper_bound
98     * Following warm-up, read forever
99     */
100     if(first_time) {
101         function_call_count = 0;
102         begin_timer();
103     }
104
105     if(check_begin_reading()) {
106         Serial.println("-----");
107         Serial.print("Function Call Count: ");
108         Serial.println(function_call_count);
109         Serial.println("-----");
110         run_sensor();
111         function_call_count ++;
112     }
113 }
114
115 bool MHZ19::run_sensor(void) {
116     /*
117     * Run the MHZ19 sensor
118     * Set ppm to zero
119     * clear the frame_buffer
120     * drain the serial buffer
121     * read from the sensor
122     * print reading
123     * add the reading to the average value buffer
124     * calculate average value
125     */
126     co2_ppm = -1;
127     co2_ppm_average = 0;
128     is_average_taken = false;
129     does_sensor_work = true;
130     reading_count = 1;
131
132     serial_drain();
133
134     while(is_average_taken == false && does_sensor_work == true) {
135         memset(frame_buffer, 0, 9);
136         read_sensor();
137         print_current_reading();
138         add_to_ave_buf();

```

```

139         calculate_average_reading();
140         print_average_reading();
141         co2_ppm = -1;
142     }
143     if(is_average_taken == true) {return(true);}
144     else {return(false);}
145 }
146
147 void MHZ19::print_current_reading(void) {
148     /*
149     * Prints current reading if reading is valid (i.e. co2_ppm > 0)
150     * and if the maximum number of readings haven't been exceeded
151     */
152     if(co2_ppm > 0) {
153         Serial.print("MHZ19 CO2 PPM Reading ");
154         Serial.print(reading_count);
155         Serial.print(": ");
156         Serial.println(co2_ppm);
157     }
158     else {
159         Serial.println("Error reading CO2 PPM from MHZ19");
160     }
161 }
162
163 void MHZ19::add_to_ave_buf(void) {
164     /*
165     * IF a valid value of co2 is read and the number of reading is less than the max,
166     * THEN add current value to buffer
167     */
168     if(co2_ppm > 0 && reading_count <= NUMBER_OF_VALUES) {
169         mhz19_buffer[reading_count - 1] = co2_ppm;
170         reading_count += 1;
171     }
172 }
173
174 void MHZ19::calculate_average_reading(void) {
175     /*
176     * IF the number of readings exceeds the number of values to be read,
177     * THEN calculate the average
178     */
179     if(reading_count > NUMBER_OF_VALUES) {
180         for(int k = 0; k < NUMBER_OF_VALUES; k++) {co2_ppm_average += mhz19_buffer[k];}
181
182         co2_ppm_average = co2_ppm_average / ( NUMBER_OF_VALUES );
183
184         is_average_taken = true;
185     }
186 }
187
188 void MHZ19::print_average_reading(void) {
189     /*
190     * IF the average has been taken (co2_ppm_average > 0)
191     * THEN print the average
192     */
193     if(co2_ppm_average > 0) {
194         Serial.println("-----");
195         Serial.print("CO2 PPM Average Reading: ");
196         Serial.println(co2_ppm_average);
197         Serial.println("-----");
198     }
199 }
200
201 void MHZ19::read_sensor(void) {
202     /*
203     * Start Serial1 connection
204     * Send command to read from sensor to the sensor
205     * Read from the sensor (fill_from_buffer());
206     * Calculate PPM for CO2
207     * End Serial connection

```

```

208     */
209
210     Serial1.begin(9600);
211     Serial1.write(mhz19_read_command, 9);
212     delay(1000);
213     fill_frame_buffer();
214     co2_ppm = 256*frame_buffer[2] + frame_buffer[3];
215     Serial1.end();
216 }
217
218 void MHZ19::serial_drain(void) {
219     /*
220     * Drains serial buffer when sensor is turned on
221     */
222     while (Serial1.available() > 0) {
223         drain = Serial1.available();
224         Serial.print("-- Draining buffer: ");
225     }
226 }
227
228 void MHZ19::frame_sync(void) {
229     /*
230     * Sync frames so that frames are added to the frame_buffer in the correct order
231     * IF correct byte is read, THEN add to buffer and move on to next byte
232     * ELSE read byte and discard
233     * IF no bytes are available to read and the frames have not been synced, THEN send
234     * command to read from sensor again
235     *
236     * frame_sync_count keeps track of how many frames are added to frame_buffer
237     * frame_read_count keeps track of how many frames are read but not added to buffer
238     (fails if too many frames read)
239     */
240     sync_state = false;
241     frame_sync_count = 0;
242     frame_read_count = 0;
243     byte_sum = 0;
244
245     while (!sync_state && Serial1.available() > 0 && frame_read_count <
246           MAX_FRAME_READ_COUNT) {
247         current_byte = Serial1.read();
248
249         if (current_byte == MHZ19_ZEROTH_BYTE && frame_sync_count == 0) {
250             frame_buffer[frame_sync_count] = current_byte;
251             byte_sum = current_byte;
252             frame_sync_count = 1;
253         }
254         else if (current_byte == MHZ19_FIRST_BYTE && frame_sync_count == 1) {
255             frame_buffer[frame_sync_count] = current_byte;
256             byte_sum += current_byte;
257             sync_state = true;
258             frame_sync_count = 2;
259         }
260         else {
261             if(debug) {
262                 Serial.print("-- Frame syncing... ");
263                 Serial.println(current_byte, HEX);
264             }
265
266             frame_read_count ++;
267         }
268
269         if (!sync_state && !(Serial1.available() > 0) && frame_read_count <
270             MAX_FRAME_READ_COUNT) {
271             Serial1.write(mhz19_read_command, 9);
272
273             if(debug) {
274                 Serial.println("-----");
275                 Serial.println("Read command has been sent to CO2 sensor");
276                 Serial.println("-----");
277             }
278         }
279     }
280 }

```

```

273         }
274
275         delay(500);
276     }
277 }
278 }
279
280 void MHZ19::fill_frame_buffer(void) {
281     /*
282     * Sync frames
283     * Read byte into frame_buffer
284     */
285     frame_sync();
286
287     while(sync_state && Serial1.available() > 0 && frame_sync_count < MAX_FRAME_LEN) {
288         current_byte = Serial1.read();
289         frame_buffer[frame_sync_count] = current_byte;
290         byte_sum += current_byte;
291         frame_sync_count++;
292     }
293 }
294
295 // getter and setter functions
296 int MHZ19::get_co2_ave(void) {
297     return co2_ppm_average;
298 }
299
300 int MHZ19::get_co2_reading(void) {
301     return co2_ppm;
302 }
303
304 void MHZ19::reset_co2_ave(void) {
305     co2_ppm_average = -1;
306 }
307

```



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.

```

1  /*
2  * This is the .h file for the ccs821 VOC sensor
3  */
4  #ifndef LIB_ADAFRUIT_CCS811_H
5  #define LIB_ADAFRUIT_CCS811_H
6
7  #if (ARDUINO >= 100)
8  #include "Arduino.h"
9  #else
10 #include "WProgram.h"
11 #endif
12
13 #include <Wire.h>
14
15 /*=====
16 I2C ADDRESS/BITS
17 -----*/
18 #define CCS811_ADDRESS          (0x5A)
19 /*=====
20
21 #define MAX_READ_COUNT 5
22 #define MAX_ERROR_COUNT 5
23
24 /*=====
25 REGISTERS
26 -----*/
27 enum
28 {
29     CCS811_STATUS = 0x00,
30     CCS811_MEAS_MODE = 0x01,
31     CCS811_ALG_RESULT_DATA = 0x02,
32     CCS811_RAW_DATA = 0x03,
33     CCS811_ENV_DATA = 0x05,
34     CCS811_NTC = 0x06,
35     CCS811_THRESHOLDS = 0x10,
36     CCS811_BASELINE = 0x11,
37     CCS811_HW_ID = 0x20,
38     CCS811_HW_VERSION = 0x21,
39     CCS811_FW_BOOT_VERSION = 0x23,
40     CCS811_FW_APP_VERSION = 0x24,
41     CCS811_ERROR_ID = 0xE0,
42     CCS811_SW_RESET = 0xFF,
43 };
44
45 //bootloader registers
46 enum
47 {
48     CCS811_BOOTLOADER_APP_ERASE = 0xF1,
49     CCS811_BOOTLOADER_APP_DATA = 0xF2,
50     CCS811_BOOTLOADER_APP_VERIFY = 0xF3,
51     CCS811_BOOTLOADER_APP_START = 0xF4
52 };
53
54 enum
55 {
56     CCS811_DRIVE_MODE_IDLE = 0x00,
57     CCS811_DRIVE_MODE_1SEC = 0x01,
58     CCS811_DRIVE_MODE_10SEC = 0x02,
59     CCS811_DRIVE_MODE_60SEC = 0x03,
60     CCS811_DRIVE_MODE_250MS = 0x04,
61 };
62
63 /*=====
64
65 #define CCS811_HW_ID_CODE      0x81
66
67 #define CCS811_REF_RESISTOR    100000
68
69 /*=====

```

```

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

```

```

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

```

```

203
204     /* 0: int disabled
205      * 1: The nINT signal is asserted (driven low) when a new sample is ready in
206      ALG_RESULT_DATA. The nINT signal will stop being driven low when
207      ALG_RESULT_DATA is read on the I2C interface.
208      */
209     uint8_t INT_DATARDY: 1;
210
211     uint8_t DRIVE_MODE: 3;
212
213     uint8_t get(){
214         return (INT_THRESH << 2) | (INT_DATARDY << 3) | (DRIVE_MODE << 4);
215     }
216 };
217 meas_mode _meas_mode;
218
219 struct error_id {
220     /* The CCS811 received an I2C write request addressed to this station but with
221     invalid register address ID */
222     uint8_t WRITE_REG_INVALID: 1;
223
224     /* The CCS811 received an I2C read request to a mailbox ID that is invalid */
225     uint8_t READ_REG_INVALID: 1;
226
227     /* The CCS811 received an I2C request to write an unsupported mode to
228     MEAS_MODE */
229     uint8_t MEASMODE_INVALID: 1;
230
231     /* The sensor resistance measurement has reached or exceeded the maximum
232     range */
233     uint8_t MAX_RESISTANCE: 1;
234
235     /* The Heater current in the CCS811 is not in range */
236     uint8_t HEATER_FAULT: 1;
237
238     /* The Heater voltage is not being applied correctly */
239     uint8_t HEATER_SUPPLY: 1;
240
241     void set(uint8_t data){
242         WRITE_REG_INVALID = data & 0x01;
243         READ_REG_INVALID = (data & 0x02) >> 1;
244         MEASMODE_INVALID = (data & 0x04) >> 2;
245         MAX_RESISTANCE = (data & 0x08) >> 3;
246         HEATER_FAULT = (data & 0x10) >> 4;
247         HEATER_SUPPLY = (data & 0x20) >> 5;
248     }
249 };
250 error_id _error_id;
251
252 /*=====*/
253 };
254
255 #endif
256

```

```

1  /*
2  * This is the .cpp file for the ccs821 VOC sensor
3  * The library for this sensor was retrieved on line:
4  * https://learn.adafruit.com/adafruit-ccs811-air-quality-sensor/arduino-wiring-test
5  * MECH 45X Team 26 did not write Part 1, the on line library
6  *
7  * Therefore Part 1 is not properly commented because the
8  * the team does not understand the code.
9  *
10 * Part 2 was written by Team 26 and is properly commented.
11 *
12 * Part 1 begins...
13 */
14
15 #include "CCS821.h"
16
17 /*****
18  *!
19  @brief  Setups the I2C interface and hardware and checks for communication.
20  @param  addr Optional I2C address the sensor can be found on. Default is 0x5A
21  @returns True if device is set up, false on any failure
22  */
23 /*****
24 bool Adafruit_CCS811::begin(uint8_t addr)
25 {
26     _i2caddr = addr;
27
28     _i2c_init();
29
30     SWReset();
31     delay(100);
32
33     //check that the HW id is correct
34     if(this->read8(CCS811_HW_ID) != CCS811_HW_ID_CODE)
35         return false;
36
37     //try to start the app
38     this->write(CCS811_BOOTLOADER_APP_START, NULL, 0);
39     delay(100);
40
41     //make sure there are no errors and we have entered application mode
42     if(checkError()) return false;
43     if(!_status.FW_MODE) return false;
44
45     disableInterrupt();
46
47     //default to read every second
48     setDriveMode(CCS811_DRIVE_MODE_1SEC);
49
50     return true;
51 }
52
53 /*****
54  *!
55  @brief  sample rate of the sensor.
56  @param  mode one of CCS811_DRIVE_MODE_IDLE, CCS811_DRIVE_MODE_1SEC,
57          CCS811_DRIVE_MODE_10SEC, CCS811_DRIVE_MODE_60SEC, CCS811_DRIVE_MODE_250MS.
58  */
59 void Adafruit_CCS811::setDriveMode(uint8_t mode)
60 {
61     _meas_mode.DRIVE_MODE = mode;
62     this->write8(CCS811_MEAS_MODE, _meas_mode.get());
63 }
64
65 /*****
66  *!
67  @brief  enable the data ready interrupt pin on the device.
68  */
69 /*****

```

```

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()
103           and geteCO2()
104   @returns 0 if no error, error code otherwise.
105  */
106 /*****
107 uint8_t Adafruit_CCS811::readData()
108 {
109     if(!available())
110         return false;
111     else{
112         uint8_t buf[8];
113         this->read(CCS811_ALG_RESULT_DATA, buf, 8);
114
115         _eCO2 = ((uint16_t)buf[0] << 8) | ((uint16_t)buf[1]);
116         _TVOC = ((uint16_t)buf[2] << 8) | ((uint16_t)buf[3]);
117
118         if(_status.ERROR)
119             return buf[5];
120
121         else return 0;
122     }
123 }
124
125 /*****
126  *!
127   @brief  set the humidity and temperature compensation for the sensor.
128   @param humidity the humidity data as a percentage. For 55% humidity, pass in
129           integer 55.
130   @param temperature the temperature in degrees C as a decimal number. For 25.5
131           degrees C, pass in 25.5
132  */
133 /*****
134 void Adafruit_CCS811::setEnvironmentalData(uint8_t humidity, double temperature)
135 {
136     /* Humidity is stored as an unsigned 16 bits in 1/512%RH. The
137        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) & 0xFF);
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  *!
167  * @double Adafruit_CCS811::calculateTemperature()
168  * {
169  *     uint8_t buf[4];
170  *     this->read(CCS811_NTC, buf, 4);
171  *
172  *     uint32_t vref = ((uint32_t)buf[0] << 8) | buf[1];
173  *     uint32_t vntc = ((uint32_t)buf[2] << 8) | buf[3];
174  *
175  *     //from ams ccs811 app note
176  *     uint32_t rntc = vntc * CCS811_REF_RESISTOR / vref;
177  *
178  *     double ntc_temp;
179  *     ntc_temp = log((double)rntc / CCS811_REF_RESISTOR); // 1
180  *     ntc_temp /= 3380; // 2
181  *     ntc_temp += 1.0 / (25 + 273.15); // 3
182  *     ntc_temp = 1.0 / ntc_temp; // 4
183  *     ntc_temp -= 273.15; // 5
184  *     return ntc_temp - _tempOffset;
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 be triggered.
192  * @param hysteresis optional hysteresis level. Defaults to 50
193  */
194 /*****
195  *!
196  * void Adafruit_CCS811::setThresholds(uint16_t low_med, uint16_t med_high, uint8_t
197  * hysteresis)
198  * {
199  *     uint8_t buf[] = {(uint8_t)((low_med >> 8) & 0xF), (uint8_t)(low_med & 0xF),
200  *                     (uint8_t)((med_high >> 8) & 0xF), (uint8_t)(med_high & 0xF), hysteresis};
201  *
202  *     this->write(CCS811_THRESHOLDS, buf, 5);
203  * }

```



```

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.beginTransaction((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.beginTransaction((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         delay(5000);
307         return true;
308     }
309 }
310
311 bool Adafruit_CCS811::run_voc(void) {
312     /*
313      * Run the VOC sensor
314      * Take measurements until enough measurements have been taken to calculate the
315      * average
316      * use read_voc() to read from sensor
317      */
318     is_average_taken = false;
319     read_count = 1;
320     error_count = 0;
321     while(is_average_taken == false && error_count < MAX_ERROR_COUNT) {read_voc();}
322
323     if(is_average_taken) {return true;}
324     else if(error_count >= MAX_ERROR_COUNT) {return false;}
325 }
326
327 void Adafruit_CCS811::read_voc(void) {
328     /*
329      * Read values from voc sensor
330      * IF data is read and max read count has not been exceed
331      * THEN fill_buffer and print_readings and read_count ++
332      * calculate_average_reading
333      * print_average_reading
334      */
335     if(available()){
336         float temp = calculateTemperature();
337         if(!readData() && read_count <= MAX_READ_COUNT){
338             fill_buffer();
339             print_readings();
340             read_count += 1;

```

```

340         error_count = 0;
341     }
342     else {
343         error_count ++;
344         Serial.print("ERROR #");
345         Serial.println(error_count);
346         delay(500);
347     }
348 }
349 calculate_average_reading();
350 print_average_reading();
351 }
352
353 void Adafruit_CCS811::fill_buffer(void) {
354     /*
355      * add new values to buffers
356      */
357     eCO2_buf[read_count-1] = geteCO2();
358     TVOC_buf[read_count-1] = getTVOC();
359 }
360
361 void Adafruit_CCS811::print_readings(void) {
362     /*
363      * Print readings
364      */
365     Serial.print("VOC Reading #:");
366     Serial.print(read_count);
367     Serial.print(", CO2: ");
368     Serial.print(geteCO2());
369     Serial.print("ppm, TVOC: ");
370     Serial.print(getTVOC());
371     Serial.println("pph");
372 }
373
374 void Adafruit_CCS811::calculate_average_reading(void) {
375     /*
376      * Calculate the average reading if enough readings have been taken
377      */
378     if(read_count > MAX_READ_COUNT) {
379         eCO2_ave = 0;
380         TVOC_ave = 0;
381         for(int k = 0; k < MAX_READ_COUNT; k++) {
382             eCO2_ave += eCO2_buf[k];
383             TVOC_ave += TVOC_buf[k];
384         }
385         eCO2_ave = eCO2_ave / MAX_READ_COUNT;
386         TVOC_ave = TVOC_ave / MAX_READ_COUNT;
387
388         read_count = 1;
389         is_average_taken = true;
390     }
391 }
392
393 void Adafruit_CCS811::print_average_reading(void) {
394     /*
395      * print average reading values
396      */
397     if(is_average_taken) {
398         Serial.println("-----");
399         Serial.println("VOC Sensor Average Readings:");
400         Serial.println("-----");
401         Serial.print("CCS eCO2 Average: ");
402         Serial.println(eCO2_ave);
403         Serial.print("CCS TVOC Average: ");
404         Serial.println(TVOC_ave);
405     }
406 }
407
408 // Getter functions for VOC parameters

```

```
409     float Adafruit_CCS811::get_eCO2_ave(void) {  
410         return eCO2_ave;  
411     }  
412     float Adafruit_CCS811::get_TVOC_ave(void) {  
413         return TVOC_ave;  
414     }  
415
```

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](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.

```

1  /*
2  * .h file for SHT35D
3  */
4
5  #ifndef SHT35D
6  #define SHT35D
7  #define MAX_READ_COUNT 5
8  #define MAX_ERROR_COUNT 5
9  #define ADDR_SHT 0x45
10
11 #include <Arduino.h>
12
13 //List of Commands for SHT35D Sensor:
14 typedef enum {
15     SHT3XD_CMD_READ_SERIAL_NUMBER = 0x3780,
16
17     SHT3XD_CMD_READ_STATUS = 0xF32D,
18     SHT3XD_CMD_CLEAR_STATUS = 0x3041,
19
20     SHT3XD_CMD_HEATER_ENABLE = 0x306D,
21     SHT3XD_CMD_HEATER_DISABLE = 0x3066,
22
23     SHT3XD_CMD_SOFT_RESET = 0x30A2,
24
25     SHT3XD_CMD_CLOCK_STRETCH_H = 0x2C06,
26     SHT3XD_CMD_CLOCK_STRETCH_M = 0x2C0D,
27     SHT3XD_CMD_CLOCK_STRETCH_L = 0x2C10,
28
29     SHT3XD_CMD_POLLING_H = 0x2400,
30     SHT3XD_CMD_POLLING_M = 0x240B,
31     SHT3XD_CMD_POLLING_L = 0x2416,
32
33     SHT3XD_CMD_ART = 0x2B32,
34
35     SHT3XD_CMD_PERIODIC_HALF_H = 0x2032,
36     SHT3XD_CMD_PERIODIC_HALF_M = 0x2024,
37     SHT3XD_CMD_PERIODIC_HALF_L = 0x202F,
38     SHT3XD_CMD_PERIODIC_1_H = 0x2130,
39     SHT3XD_CMD_PERIODIC_1_M = 0x2126,
40     SHT3XD_CMD_PERIODIC_1_L = 0x212D,
41     SHT3XD_CMD_PERIODIC_2_H = 0x2236,
42     SHT3XD_CMD_PERIODIC_2_M = 0x2220,
43     SHT3XD_CMD_PERIODIC_2_L = 0x222B,
44     SHT3XD_CMD_PERIODIC_4_H = 0x2334,
45     SHT3XD_CMD_PERIODIC_4_M = 0x2322,
46     SHT3XD_CMD_PERIODIC_4_L = 0x2329,
47     SHT3XD_CMD_PERIODIC_10_H = 0x2737,
48     SHT3XD_CMD_PERIODIC_10_M = 0x2721,
49     SHT3XD_CMD_PERIODIC_10_L = 0x272A,
50
51     SHT3XD_CMD_FETCH_DATA = 0xE000,
52     SHT3XD_CMD_STOP_PERIODIC = 0x3093,
53
54     SHT3XD_CMD_READ_ALR_LIMIT_LS = 0xE102,
55     SHT3XD_CMD_READ_ALR_LIMIT_LC = 0xE109,
56     SHT3XD_CMD_READ_ALR_LIMIT_HS = 0xE11F,
57     SHT3XD_CMD_READ_ALR_LIMIT_HC = 0xE114,
58     SHT3XD_CMD_WRITE_ALR_LIMIT_HS = 0x611D,
59     SHT3XD_CMD_WRITE_ALR_LIMIT_HC = 0x6116,
60     SHT3XD_CMD_WRITE_ALR_LIMIT_LC = 0x610B,
61     SHT3XD_CMD_WRITE_ALR_LIMIT_LS = 0x6100,
62
63     SHT3XD_CMD_NO_SLEEP = 0x303E,
64 } SHT31D_Commands;
65
66 // List of repeatability options for SHT35D:
67 typedef enum {
68     SHT3XD_REPEATABILITY_HIGH,
69     SHT3XD_REPEATABILITY_MEDIUM,

```

```

70     SHT3XD_REPEATABILITY_LOW,
71 } SHT31D_Repeatability;
72
73 // List of modes:
74 typedef enum {
75     SHT3XD_MODE_CLOCK_STRETCH,
76     SHT3XD_MODE_POLLING,
77 } SHT31D_Mode;
78
79 // List of frequency choices
80 typedef enum {
81     SHT3XD_FREQUENCY_HZ5,
82     SHT3XD_FREQUENCY_1HZ,
83     SHT3XD_FREQUENCY_2HZ,
84     SHT3XD_FREQUENCY_4HZ,
85     SHT3XD_FREQUENCY_10HZ
86 } SHT31D_Frequency;
87
88 // List of errors:
89 typedef enum {
90     SHT3XD_NO_ERROR = 0,
91
92     SHT3XD_CRC_ERROR = -101,
93     SHT3XD_TIMEOUT_ERROR = -102,
94
95     SHT3XD_PARAM_WRONG_MODE = -501,
96     SHT3XD_PARAM_WRONG_REPEATABILITY = -502,
97     SHT3XD_PARAM_WRONG_FREQUENCY = -503,
98     SHT3XD_PARAM_WRONG_ALERT = -504,
99
100 // Wire I2C translated error codes
101
102     SHT3XD_WIRE_I2C_DATA_TOO_LOG = -10,
103     SHT3XD_WIRE_I2C_RECEIVED_NACK_ON_ADDRESS = -20,
104     SHT3XD_WIRE_I2C_RECEIVED_NACK_ON_DATA = -30,
105     SHT3XD_WIRE_I2C_UNKNOW_ERROR = -40
106 } SHT31D_ErrorCode;
107
108 // List of statuses:
109 typedef union {
110     uint16_t rawData;
111     struct {
112         uint8_t WriteDataChecksumStatus : 1;
113         uint8_t CommandStatus : 1;
114         uint8_t Reserved0 : 2;
115         uint8_t SystemResetDetected : 1;
116         uint8_t Reserved1 : 5;
117         uint8_t T_TrackingAlert : 1;
118         uint8_t RH_TrackingAlert : 1;
119         uint8_t Reserved2 : 1;
120         uint8_t HeaterStatus : 1;
121         uint8_t Reserved3 : 1;
122         uint8_t AlertPending : 1;
123     };
124 } SHT31D_RegisterStatus;
125
126 struct SHT31D {
127     /*
128     * Structure for SHT31D
129     * t - temperature
130     * rh - relative humidity
131     * error - error of type SHT31D_ErrorCode
132     */
133     float t;
134     float rh;
135     SHT31D_ErrorCode error;
136 };
137
138 class ClosedCube_SHT31D {

```

```

139  /*
140  * Class definition for ClosedCube_SHT31D
141  */
142  public:
143      ClosedCube_SHT31D();
144
145      bool start_sht(void);
146      bool 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,
167                                uint8_t timeout);
168      SHT31D readTempAndHumidityClockStretch(SHT31D_Repeatability repeatability);
169      SHT31D readTempAndHumidityPolling(SHT31D_Repeatability repeatability, uint8_t
170                                        timeout);
171
172      SHT31D_ErrorCode periodicStart(SHT31D_Repeatability repeatability, SHT31D_Frequency
173                                     frequency);
174      SHT31D periodicFetchData();
175      SHT31D_ErrorCode periodicStop();
176
177      SHT31D_ErrorCode writeAlertHigh(float temperatureSet, float temperatureClear, float
178                                     humiditySet, float humidityClear);
179      SHT31D readAlertHighSet();
180      SHT31D readAlertHighClear();
181
182      SHT31D_ErrorCode writeAlertLow(float temperatureClear, float temperatureSet, float
183                                     humidityClear, float humiditySet);
184      SHT31D readAlertLowSet();
185      SHT31D readAlertLowClear();
186
187  private:
188      float t_buf[MAX_READ_COUNT];
189      float rh_buf[MAX_READ_COUNT];
190      bool is_average_taken;
191      int read_count;
192      int error_count;
193      float t_average;
194      float rh_average;
195
196      SHT31D save_to_buffer(SHT31D result);
197      SHT31D read_sht(void);
198      void calculate_average(void);
199
200      uint8_t _address;
201      SHT31D_RegisterStatus _status;
202
203      SHT31D_ErrorCode writeCommand(SHT31D_Commands command);
204      SHT31D_ErrorCode writeAlertData(SHT31D_Commands command, float temperature, float
205                                     humidity);
206
207      uint8_t checkCrc(uint8_t data[], uint8_t checksum);

```



```

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

```

```

1  /*
2  * This is the .cpp file for the SHT35D Temperature
3  * and relative humidity sensor.
4  *
5  * Part 1 of this code was retrieved online:
6  * https://github.com/closedcube/ClosedCube\_SHT31D\_Arduino
7  *
8  * Part 2 was written by MECH 45X Team 26
9  *
10 * Part 1 begins...
11 */
12
13 #include <Wire.h>
14 #include "SHT35D.h"
15
16 ClosedCube_SHT31D::ClosedCube_SHT31D()
17 {
18 }
19
20 SHT31D_ErrorCode ClosedCube_SHT31D::begin(uint8_t address) {
21     SHT31D_ErrorCode error = SHT3XD_NO_ERROR;
22     _address = address;
23     return error;
24 }
25
26 SHT31D_ClosedCube_SHT31D::periodicFetchData()
27 {
28     SHT31D_ErrorCode error = writeCommand(SHT3XD_CMD_FETCH_DATA);
29     if (error == SHT3XD_NO_ERROR)
30         return readTemperatureAndHumidity();
31     else
32         returnError(error);
33 }
34
35 SHT31D_ErrorCode ClosedCube_SHT31D::periodicStop() {
36     return writeCommand(SHT3XD_CMD_STOP_PERIODIC);
37 }
38
39 SHT31D_ErrorCode ClosedCube_SHT31D::periodicStart(SHT31D_Repeatability repeatability,
40 SHT31D_Frequency frequency)
41 {
42     SHT31D_ErrorCode error;
43
44     switch (repeatability)
45     {
46     case SHT3XD_REPEATABILITY_LOW:
47         switch (frequency)
48         {
49             case SHT3XD_FREQUENCY_HZ5:
50                 error = writeCommand(SHT3XD_CMD_PERIODIC_HALF_L);
51                 break;
52             case SHT3XD_FREQUENCY_1HZ:
53                 error = writeCommand(SHT3XD_CMD_PERIODIC_1_L);
54                 break;
55             case SHT3XD_FREQUENCY_2HZ:
56                 error = writeCommand(SHT3XD_CMD_PERIODIC_2_L);
57                 break;
58             case SHT3XD_FREQUENCY_4HZ:
59                 error = writeCommand(SHT3XD_CMD_PERIODIC_4_L);
60                 break;
61             case SHT3XD_FREQUENCY_10HZ:
62                 error = writeCommand(SHT3XD_CMD_PERIODIC_10_L);
63                 break;
64             default:
65                 error = SHT3XD_PARAM_WRONG_FREQUENCY;
66                 break;
67         }
68         break;
69     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,
125 SHT31D_Mode mode, uint8_t timeout)
126 {
127     SHT31D result;
128
129     switch (mode) {
130         case SHT3XD_MODE_CLOCK_STRETCH:
131             result = readTempAndHumidityClockStretch(repeatability);
132             break;
133         case SHT3XD_MODE_POLLING:
134             result = readTempAndHumidityPolling(repeatability, timeout);
135             break;
136         default:
137             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 }

```

```

204 }
205
206 SHT31D_ClosedCube_SHT31D::readAlertHighSet() {
207     return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_HS);
208 }
209
210 SHT31D_ClosedCube_SHT31D::readAlertHighClear() {
211     return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_HC);
212 }
213
214 SHT31D_ClosedCube_SHT31D::readAlertLowSet() {
215     return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_LS);
216 }
217
218 SHT31D_ClosedCube_SHT31D::readAlertLowClear() {
219     return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_LC);
220 }
221
222
223 SHT31D_ErrorCode_ClosedCube_SHT31D::writeAlertHigh(float temperatureSet, float
temperatureClear, float humiditySet, float humidityClear) {
224     SHT31D_ErrorCode error = writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_HS,
temperatureSet, humiditySet);
225     if (error == SHT3XD_NO_ERROR)
226         error = writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_HC, temperatureClear,
humidityClear);
227
228     return error;
229 }
230
231 SHT31D_ErrorCode_ClosedCube_SHT31D::writeAlertLow(float temperatureClear, float
temperatureSet, float humidityClear, float humiditySet) {
232     SHT31D_ErrorCode error = writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_LS,
temperatureSet, humiditySet);
233     if (error == SHT3XD_NO_ERROR)
234         writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_LC, temperatureClear, humidityClear);
235
236     return error;
237 }
238
239 SHT31D_ErrorCode_ClosedCube_SHT31D::writeAlertData(SHT31D_Commands command, float
temperature, float humidity)
240 {
241     SHT31D_ErrorCode error;
242
243     if ((humidity < 0.0) || (humidity > 100.0) || (temperature < -40.0) || (temperature
> 125.0))
244     {
245         error = SHT3XD_PARAM_WRONG_ALERT;
246     }
247     else {
248         uint16_t rawTemperature = calculateRawTemperature(temperature);
249         uint16_t rawHumidity = calculateRawHumidity(humidity);
250         uint16_t data = (rawHumidity & 0xFE00) | ((rawTemperature >> 7) & 0x001FF);
251
252         uint8_t buf[2];
253         buf[0] = data >> 8;
254         buf[1] = data & 0xFF;
255
256         uint8_t checksum = calculateCrc(buf);
257
258         Wire.beginTransaction(_address);
259         Wire.write(command >> 8);
260         Wire.write(command & 0xFF);
261         Wire.write(buf[0]);
262         Wire.write(buf[1]);
263         Wire.write(checksum);
264         return (SHT31D_ErrorCode) (-10 * Wire.endTransmission());
265     }

```

```

266         return error;
267     }
268 }
269
270
271 SHT31D_ErrorCode ClosedCube_SHT31D::writeCommand(SHT31D_Commands command)
272 {
273     Wire.beginTransaction(_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.beginTransaction(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 bool 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     error_count = 1;
499     read_count = 1;
500     while(read_count <= MAX_READ_COUNT && error_count <= MAX_ERROR_COUNT) {
501         read_sht();
502     }
503
504     return(is_average_taken);
505 }
506
507 SHT31D ClosedCube_SHT31D::read_sht(void) {
508     /*
509     * Read from SHT35D, and assign values to my_result
510     * print results
511     * save results to buffer
512     * calculate average if enough values have been read
513     */
514     SHT31D my_result = periodicFetchData();
515     printResult("Periodic Mode", my_result);
516     save_to_buffer(my_result);
517     calculate_average();
518     delay(250);
519 }
520
521 SHT31D ClosedCube_SHT31D::printResult(String text, SHT31D result) {
522     /*
523     * Prints current reading if no error and not exceeded max count
524     * else print error message
525     */
526     if (result.error == SHT3XD_NO_ERROR && read_count <= MAX_READ_COUNT ) {
527         float current_t = result.t;
528         float current_rh = result.rh;
529
530         if(current_t > 0 && current_rh > 0) {
531             //Serial.print(text);
532             Serial.print("SHT Reading #");
533             Serial.print(read_count);
534             Serial.print(": T=");
535             Serial.print(current_t);
536             Serial.print("C, RH=");
537             Serial.print(current_rh);
538             Serial.println("%");
539         }
540     }

```

```

541 }
542
543 SHT31D ClosedCube_SHT31D::save_to_buffer(SHT31D result) {
544     /*
545     * Save current t and rh readings to their respective buffers
546     *
547     * if no error and the number of readings is less than the max
548     * then save values
549     *
550     * else -> report error, do not save any values
551     */
552     if (result.error == SHT3XD_NO_ERROR && read_count <= MAX_READ_COUNT) {
553         float current_t = result.t;
554         float current_rh = result.rh;
555
556         if (current_t > 0 && current_rh > 0) {
557             t_buf[read_count - 1] = current_t;
558             rh_buf[read_count - 1] = current_rh;
559             read_count++;
560             error_count = 1;
561         } else {
562             Serial.print("SHT Error count: ");
563             Serial.println(error_count);
564             error_count++;
565         }
566     } else if (result.error != SHT3XD_NO_ERROR) {
567         Serial.print("[ERROR] Code #");
568         Serial.println(result.error);
569         Serial.print("SHT Error count: ");
570         Serial.println(error_count);
571         error_count++;
572     }
573 }
574
575 void ClosedCube_SHT31D::calculate_average(void) {
576     /*
577     * Calculate average if enough values have been read
578     * assign t ave to t_average
579     * assign rh ave to rh_average
580     * change is_average_taken to true so that while loop will exit
581     */
582     if (read_count > MAX_READ_COUNT) {
583         t_average = 0.00;
584         rh_average = 0.00;
585         for (int k = 0; k < MAX_READ_COUNT; k++) {
586             t_average += t_buf[k];
587             rh_average += rh_buf[k];
588         }
589         t_average = t_average / MAX_READ_COUNT;
590         rh_average = rh_average / MAX_READ_COUNT;
591
592         delay(500);
593         Serial.println("-----");
594         Serial.println("SHT Sensor Average Readings");
595         Serial.println("-----");
596         Serial.print("SHT T Average: ");
597         Serial.println(t_average);
598         Serial.print("SHT RH Average: ");
599         Serial.println(rh_average);
600         is_average_taken = true;
601     }
602 }
603
604 // getter function to get average temperature reading
605 float ClosedCube_SHT31D::get_t_ave(void) {
606     return t_average;
607 }
608 // getter function to get average relative humidity reading
609 float ClosedCube_SHT31D::get_rh_ave(void) {

```

```
610     return rh_average;
611 }
612
```

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](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.

```

1  /*
2  * This is the .h file for the Si7051 sensor
3  * This sensor is used in the globe thermometer
4  */
5
6  #ifndef _CLOSEDCUBE_SI7051_h
7
8  #define _CLOSEDCUBE_SI7051_h
9  #define MAX_READ_COUNT 5
10 #define MAX_ERROR_COUNT 40
11 #define ADDR_MRT 0x40
12 #define DEFAULT_AVERAGE 128
13 #include <Arduino.h>
14
15 class ClosedCube_Si7051 {
16 public:
17     ClosedCube_Si7051();
18
19     float readT(); // short-cut for readTemperature
20     bool run_mrt(void);
21     bool start_mrt(void);
22     float get_MRT_ave(void);
23
24 private:
25     uint8_t _address;
26     void begin(uint8_t address);
27     float readTemperature();
28     float T_buf[MAX_READ_COUNT];
29     float T_ave;
30     int read_count;
31     int error_count;
32 };
33
34 #endif
35

```

```

1  /*
2  * This is the .cpp file for the Si7051 sensor
3  * The Si7015 is being used as the Globe Thermometer Sensor
4  * The bulk of this library was retrieved on line:
5  * https://github.com/closedcube/ClosedCube\_Si7051\_Arduino
6  *
7  * Part 1 of this library was retrieved on line,
8  * while Part 2 was written by MECH 45X Team 26
9  *
10 * Team 26 does not fully understand how the on line
11 * library works, so Part 1 is not commented
12 *
13 * Team 26 commented Part 2 as they wrote Part 2
14 * and understand how the code in Part 2 works
15 *
16 * Please note that the Globe Thermometer does not
17 * measure Mean Radiant Temperature (MRT), it
18 * actually measures the globe temperature.
19 * MRT is calculate later using air temperature and
20 * globe temperature.
21 */
22
23 #include <Wire.h>
24 #include "MRT.h"
25
26 ClosedCube_Si7051::ClosedCube_Si7051()
27 {
28 }
29
30 void ClosedCube_Si7051::begin(uint8_t address) {
31     _address = address;
32     Wire.begin();
33
34     Wire.beginTransmission(_address);
35     Wire.write(0xE6);
36     Wire.write(0x0);
37     Wire.endTransmission();
38
39 }
40
41 float ClosedCube_Si7051::readT() {
42     return readTemperature();
43 }
44
45 float ClosedCube_Si7051::readTemperature() {
46     Wire.beginTransmission(_address);
47     Wire.write(0xF3);
48     Wire.endTransmission();
49
50     delay(15);
51
52     Wire.requestFrom(_address, (uint8_t)2);
53     delay(25);
54     byte msb = Wire.read();
55     byte lsb = Wire.read();
56
57     uint16_t val = msb << 8 | lsb;
58
59     return (175.72*val) / 65536 - 46.85;
60 }
61
62 //*****//
63 // Part 2: Si7051 MECH 45X Team 26 library //
64 // The following code was written by MECH 45X Team 26 //
65 // It is properly commented //
66 //*****//
67
68
69 bool ClosedCube_Si7051::start_mrt(void) {

```

```

70  /*
71  * Start MRT sensor
72  *
73  * The code will read a value of 128 or greater
74  * if the sensor is broken or disconnected
75  *
76  * The start sequence returns false (sensor does not work)
77  * if a value of 128 is read
78  *
79  * If the value is less than 128, it returns true
80  * (sensor works)
81  *
82  * The code retrieved from the online library should be improved
83  * to fix this.
84  */
85  begin(ADDR_MRT);
86  delay(500);
87  return(run_mrt());
88  }
89
90  bool ClosedCube_Si7051::run_mrt(void) {
91  /*
92  * Takes MRT measurements until read_count is exceeded
93  * once read_count is exceeded, the average is taken
94  */
95  read_count = 1;
96  error_count = 1;
97
98  while(read_count <= MAX_READ_COUNT && error_count <= MAX_ERROR_COUNT) {
99      float current_T = readTemperature();
100
101      if(current_T >= DEFAULT_AVERAGE) {
102          Serial.println("-----");
103          Serial.print("Error reading from Globe Thermometer, Tg: ");
104          Serial.println(current_T);
105          Serial.println("-----");
106          error_count++;
107          delay(1000);
108      } else{
109          T_buf[read_count - 1] = readTemperature();
110          Serial.print("Globe Thermometer Reading #");
111          Serial.print(read_count);
112          Serial.print(": Tg is: ");
113          Serial.println(T_buf[read_count - 1]);
114          read_count++;
115          error_count = 1;
116          delay(250);
117      }
118  }
119
120  if(read_count > MAX_READ_COUNT) {
121      T_ave = 0;
122      for(int k = 0; k < MAX_READ_COUNT; k++) {
123          T_ave = T_ave + T_buf[k];
124      }
125      T_ave = T_ave / MAX_READ_COUNT;
126      Serial.println("-----");
127      Serial.print("Average Tg is: ");
128      Serial.println(T_ave);
129      Serial.println("-----");
130      return(true);
131  }
132  else if(error_count > MAX_ERROR_COUNT) {
133      T_ave = -1;
134      Serial.println("-----");
135      Serial.println("Error reading from Globe Thermometer, no average Tg calculated");
136      Serial.println("-----");
137      return(false);
138  }

```

```

139         else{
140             Serial.println("-----");
141             Serial.println("Failure for no known reason");
142             Serial.println("-----");
143             return(false);}
144     }
145
146     // Getter function for Globe Thermometer average temperature
147     float ClosedCube_Si7051::get_MRT_ave(void) {
148         return T_ave;
149     }

```