MECH 49X Dossier 11 - Code

Team 26

March 31, 2018

The code for running for running the sensor package is:

'_all.ino'.

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

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

```
/*
* Script _all.ino
      * This script runs the sensor package
      * Uses objects for each of the sensors
      * Prints information to Serial screen
      * Publishes data to ThingSpeak
     #include "CALCULATE_MRT.h"
#include "MHZ19.h"
10
     #include "CCS821.h"
11
     #include "SHT35D.h"
#include "MRT.h"
13
     #include "PM.h"
#include "Time.h"
14
15
    #include <Wire.h>
16
     // create instances of objects
18
19
     PM 7003 myPM;
     ClosedCube_Si7051 myMRT;
     ClosedCube_SHT31D mySHT;
22
     Adafruit CCS811 myVOC;
     MHZ19 myCO2;
24
     mrt_and_ot my_MRT_OT;
25
    /*
* Boolean expressions
26
27
     * start xxx indicate whether sensor has been read from properly
* read_from_xxx indicate whether or not to read from sensor_xxx (changes throughout
28
29
      code)
     * finished_xxx indicates whether done reading from a sensor (read a good average)
30
31
     bool start co2 = false;
     bool start_voc = false;
bool start_sht = false;
34
     bool start_pm = false;
36
     bool start_mrt = false;
38
     bool read from co2 = true;
     bool read_from_pm = false;
39
40
41
     bool finished co2 = false;
     bool finished_pm = false;
42
     bool finished_other_sensors = false;
bool finished_mrt_ot = false;
43
44
45
     bool finished voc = false;
46
     // average reading values
48
     int co2 ave = -1;
     float sht_rh_ave = -1;
49
     float sht_t_ave = -1;
     float voc_eCO2_ave = -1;
float voc TVOC ave = -1;
51
53
      int pm ave = -\overline{1};
     float T_g = -1;
float T_a = -1;
54
55
56
      float T mrt = -1;
57
      float T ot = -1;
58
     bool publish_data = true; // should we publish data?
59
60
      // pin numbers for pm and co2 sensors
61
      int pm_transistor_control = A4;
62
63
      int pm_tx_transistor_control = A5;
     int co2_transistor_control = A3;
64
65
66
      void setup() {
           * Start Serial and Wire connections
68
```

```
* Initialize transistor control for CO2 and PM
          * Turn CO2 sensor on (make_sensor_read())
          * Test all I2C sensors (MRT, SHT, VOC)
72
          * Stop and wait for 30 seconds (warm-up)
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
         start_mrt = myMRT.start_mrt();
83
84
         Serial.println("-----
85
86
         start sht = mySHT.start sht();
87
         Serial.println("-----
88
89
         start_voc = myVOC.start_voc();
90
         Serial.println("-----
         Serial.println("30 second delay");
92
         Serial.println("----");
93
         delay(30000);
94
95
97
     void loop() {
98
          * Wait for CO2 sensor to warm-up (PM sensor is off)
99
          * Read from MRT, SHT, and VOC sensors while CO2 sensor warms-up
          * After CO2 sensor warms-up, read from CO2 sensor and save average reading
102
          * Save average value from MRT, SHT, and VOC sensors
          * Turn off CO2 sensor, turn on PM sensor
103
          ^{\star} Read from MRT, SHT, and VOC sensors while PM sensor warms-up
104
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) {
             start co2 = myCO2.make sensor read();
111
             start_pm = false;
113
             if(start_co2) {
114
115
                 read_from_co2 = false;
116
                 read from pm = true;
117
                 finished co2 = true;
118
             1
119
         else if(read_from_pm) {
121
             start pm = myPM.make sensor read();
             start_co2 = false;
122
             if(start_pm) {
    read_from_pm = false;
124
126
                  read from co2 = true;
                 finished pm = true;
128
             }
129
         }
130
         start mrt = myMRT.run mrt(); //read from MRT sensor
133
         // Read from SHT sensor, or restart SHT sensor
         if(start_sht) {
134
             Serial.println("Reading from SHT Sensor");
135
136
             Serial.println("----");
             start sht = mySHT.run sht();
137
```

```
Serial.println("----");
138
139
140
          else if(!start sht) {
              Serial.println("----");
141
              Serial.println("Not reading from SHT Sensor");
142
              Serial.println("----");
143
144
              Serial.println("Tring to start SHT");
              start sht = mySHT.start sht();
145
              Serial.println("-----
146
147
          ^{\prime}// Read from VOC sensor, or restart VOC sensor
148
149
          if(start voc) {
              Serial.println("Reading from VOC Sensor");
150
              Serial.println("----");
151
              start_voc = myVOC.run_voc();
              Serial.println("----");
154
155
          else if(!start voc) {
              start_voc = myVOC.start_voc();
156
              Serial.println("Reading from VOC Sensor");
157
              Serial.println("-----");
158
159
              start voc = myVOC.run voc();
              Serial.println("----");
160
161
          1
162
          // If done reading from CO2 sensor, save CO2, MRT, SHT, and VOC readings if(finished\_co2 \&\& !finished\_other\_sensors) {}
163
164
              finished_other_sensors = true;
165
166
              if(!finished_mrt_ot) {
    if(start_mrt && start_sht){
167
168
169
                       T_g = myMRT.get_MRT_ave();
170
                       T_a = mySHT.get_t_ave();
                       sht_rh_ave = mySHT.get_rh_ave();
my_MRT_OT.calculate_mrt_and_ot(T_g, T_a);
171
172
                       T_mrt = my_MRT_OT.get_mrt();
173
174
                       T_ot = my_MRT_OT.get_ot();
175
                       finished mrt ot = true;
176
                  else if(start_mrt && !start_sht) {
178
                       T_g = myMRT.get_MRT_ave();
179
                       T a = -1;
180
                       \overline{\text{sht}} rh ave = -1;
                       T mrt = -1;
181
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;
                       T_a = -1;
193
194
                       \overline{sht} rh ave = -1;
195
                       T \text{ mrt} = -1;
                       T ot = -1;
196
197
                  }
198
              }
199
              if(start voc && !finished voc){
                  voc_eCO2_ave = myVOC.get_eCO2_ave();
voc_TVOC_ave = myVOC.get_TVOC_ave();
                  finished voc = true;
              } else {
204
205
                  voc eCO2 ave = -1;
                  voc_TVOC_ave = -1;
206
```

```
208
209
                 co2 ave = myCO2.get co2 ave();
210
211
                 if(finished_mrt_ot && finished_voc) {
212
                      finished other sensors = true;
213
214
           }
215
            // If done reading from PM and CO2 sensors, save PM reading and push to ThingSpeak
216
            if(finished_co2 && finished_pm) {
217
218
                 pm ave = myPM.get pm ave();
                 finished_co2 = false;
219
                 finished_pm = false;
220
221
                 finished_mrt_ot = false;
                 finished_voc = false;
223
                 finished_other_sensors = false;
224
225
                 if(publish data) {
226
                      char data[1000];
                      cnar data[1000];
sprintf(data,"{ \"Mean Radiant Temperature\": \"%3.2f\", \"Operating
Temperature\": \"%3.2f\", \"CO2 Concentration\": \"%i\", \"eCO2\":
\"%4.2f\", \"TVOC\": \"%4.2f\",\"PM 2_5\": \"%i\", \"Air Temperature\":
\"%3.2f\",\"Relative Humidity of Air\": \"%3.2f\"}" , T_mrt, T_ot, co2_ave,
                      voc_eCO2_ave, voc_TVOC_ave, pm_ave, T_a, sht_rh_ave);
228
                      Serial.println("----");
                      Serial.print("Data:");
229
                      Serial.println(data);
                      Serial.println("--
233
                      Particle.publish("IEQ Final Prototype", data, PRIVATE);
234
235
                      myCO2.reset co2 ave();
236
                      myPM.reset_pm_ave();
237
                 }
238
            }
239
       }
```

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.
 4
     #ifndef CALCULATE_MRT_H
#define CALCULATE_MRT_H
     #if ARDUINO >= 100
#include "Arduino.h"
10
     #else
11
       #include "WProgram.h"
13
     #endif
14
15
     class mrt_and_ot {
16
        public:
17
               mrt_and_ot(void);
18
               void calculate_mrt_and_ot(float T_g, float T_a);
19
20
21
               float get_mrt(void);
               float get_ot(void);
22
               float calculate_convection_coefficient(float T_g, float T_a);
float h;
24
25
               float T_mrt;
float T_ot;
float T_a;
float T_g;
float T_g;
26
27
28
29
30
               float convection_coefficient;
31
32
               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;
34
35
      #endif
37
```

```
/*
    * This is the .cpp file for calculating MRT and OT
      * This code was written entirely by Team 26
 4
      * using formulas found in Literature.
     #include "CALCULATE MRT.h"
     mrt and ot::mrt and ot(void)
 9
10
11
      float mrt_and_ot::calculate_convection_coefficient(float T_g, float T_a) {
13
           * Calculate convection coefficient using formula in Literature
14
15
         h = abs(T_g - T_a) / diameter_to_power;
h = pow(h, 0.25);
return(1.4 * h);
16
17
18
19
     }
20
     void mrt_and_ot::calculate_mrt_and_ot(float T_g, float T_a) {
22
           * Calculate MRT and OT using formulas found in Literature
23
24
          T_g = T_g + kelvin_conversion;
T_a = T_a + kelvin_conversion;
25
26
27
          convection coefficient = calculate convection coefficient(T g, T a);
          T_mrt = convection_coefficient / epsilon * (T_g - T_a);
28
          T_mrt = T_mrt + pow(T_g,4);
T_mrt = pow(T_mrt,0.25);
T_ot = 0.5 * (T_a + T_mrt);
29
30
31
32
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
```

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
 4
     #include <stdint.h>
#include "WProgram.h"
#include "Time.h"
10
     #define LIB PM H
     #define FIRST BYTE 0x42
11
     #define SECOND BYTE 0x4D
     #define SENSOR OUTPUT PIN A0
13
     #define MAX_FRAME_LENGTH 64
14
15
     #define START_TIME 6000
#define SAMPLING_TIME 280
16
     #define SLEEP TIME 912
18
     #define MAX_READ_COUNT 5
#define MAX_FRAME_SYNC_COUNT 40
#define PMS_START_UP_TIME 120
#define MAX_FUNCTION_CALL_COUNT 1
19
22
24
     class PM_7003 {
25
     public:
          PM_7003();
26
27
          virtual ~PM 7003();
28
          int get_pm_ave(void);
          void set_transistor(int ground_pin, int tx_pin);
29
30
          bool make_sensor_read(void);
          void calibrate_sensor(void);
31
32
          void reset_pm_ave(void);
34
     private:
          int current_byte;
35
36
          bool sync_state;
37
          char print_buffer[256];
          uint16_t byte_sum;
39
          int drain;
40
          uint16_t current_data;
41
          int pm_ground_control;
42
          int pm tx control;
          char frame_buffer[MAX_FRAME_LENGTH];
43
          int frame_count;
int frame_length;
44
45
46
47
          bool debug = false;
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;
5.5
          int frame_sync_count;
56
          bool first_time;
57
          bool run PM sensor(void);
          void drain_serial(void);
void frame_sync(void);
59
60
61
          void read_sensor(void);
62
          void data_switch(uint16_t current_data);
63
          void print_messages(void);
64
65
          //time
          void begin_timer(void);
66
67
          bool check_begin_reading(void);
68
          time t start time;
          time_t current_time;
```

```
70
71
72
                                                       time_t duration;
 73
74
75
                                              struct PMS7003data {
    uint8_t start_frame[2];
    uint16_t frame_length;
                                                               uint16_t frame_length;
uint16_t concPM1_0_factory;
uint16_t concPM2_5_factory;
uint16_t concPM0_0_factory;
uint16_t concPM1_0_ambient;
uint16_t concPM1_0_ambient;
uint16_t concPM1_0_ambient;
uint16_t countPM0_3um;
uint16_t countPM0_5um;
uint16_t countPM0_5um;
uint16_t countPM0_0um;
uint16_t countPM1_0um;
uint16_t countPM2_0um;
uint16_t countPM3_0um;
uint16_t countPM1_0um;
uint16_t checksum;
packetdata;
 76
77
78
79
80
  81
  82
  83
  84
  85
  86
  87
  88
  89
  90
  91
                                                         } packetdata;
                        };
```

```
* This is the .cpp file for the PMS7003 sensor
     * This code was written exclusively by MECH 45X Team 26
4
    #include "PM.h"
5
    PM 7003::PM 7003() {
       current byte = 0;
        packetdata.frame_length = MAX_FRAME_LENGTH;
        frame_length = MAX_FRAME LENGTH;
10
11
    PM_7003::~PM_7003() {
13
14
15
    int PM_7003::getpm(void) {
16
17
        return pm_avgpm2_5;
18
19
20
    bool PM_7003::run_PM_sensor(void) {
22
         * run the PM sensor
         * Start serial connection
23
24
25
         * drain_serial() and read_sensor() until enough values have been read
         * to take the average
26
27
28
        Serial1.begin(9600);
29
        read_count = 1;
30
        done_reading = false;
31
        frame_sync_count = 0;
        pm_avgpm2_5 = 0;
        while (!done reading && frame sync count < MAX FRAME SYNC COUNT) {
34
            drain serial();
35
             delav(500);
36
             read_sensor();
37
        }
39
        Serial1.end();
40
41
        if(done reading) {
42
             Serial.println("----");
             Serial.println("Done reading from PM sensor");
43
             Serial.println("-----");
44
             Serial.println(" ");
45
46
             return true;
47
         else if(!done reading && frame sync count >= MAX FRAME SYNC COUNT) {return false;}
49
    }
50
51
    void PM_7003::drain_serial(void) {
52
         * Drains serial buffer if there are more than 32 entries
54
        * Reads entries to drain serial buffer
5.5
56
        if (Serial1.available() > 32) {
57
             drain = Serial1.available();
             Serial.println("-- Draining buffer: ");
59
             Serial.println(Serial1.available(), DEC);
             for (int drain_index = drain; drain_index > 0; drain_index--) {Serial1.read();}
60
61
        }
62
    }
63
    void PM_7003::frame_sync(void) {
64
65
      * syncs frames for PM sensor

* checks that frames are being read in correct order
66
67
68
       * exits when it confirms that frames are being read correctly
```

```
sync_state = false;
 71
          frame_count = 0;
          byte sum = 0;
 73
 74
          while (!sync_state && frame_sync_count < MAX_FRAME_SYNC_COUNT) {</pre>
 75
              current byte = Serial1.read();
 76
              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;
              else if(current byte == SECOND BYTE && frame count == 1){
 83
                  frame buffer[frame_count] = current_byte;
 84
                  packetdata.start_frame[1] = current_byte;
 8.5
 86
                  byte_sum = byte_sum + current_byte;
 87
                  frame count = 2;
 88
                  sync_state = true;
 29
 90
              else{
 91
                  frame sync count++;
                  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);
                  delay(500);
 98
99
                  if(frame_sync_count >= MAX_FRAME_SYNC_COUNT) {
                      Serial.println("----");
                      Serial.println("Max frame count exceeded");
                      Serial.println("-----");
103
                  1
104
              }
106
          }
107
108
109
      void PM_7003::read_sensor(void) {
110
          * Sync the frames
111
           * read bytes and fill frame buffer
           * use data switch to calculate different parameters
           ^{\star} print_messages once all values have been read.
114
           * done_reading = true if enough values have been read
115
116
117
          frame_sync();
118
          while(sync_state == true && Serial1.available() > 0) {
119
              current byte = Serial1.read();
              frame_buffer[frame_count] = current_byte;
              byte sum = byte sum + current byte;
123
              frame_count++;
124
              uint16_t current_data = frame_buffer[frame_count-1]+(frame_buffer[frame_count-2
              1<<8);
              data switch (current data);
126
              if (frame_count >= frame_length && read_count <= MAX_READ_COUNT) {
    print_messages();</pre>
128
129
                  pm_avgpm2_5 = pm_avgpm2_5 + pm2_5;
130
                  read_count++;
                  break;
              }
          1
134
135
          if (read_count > MAX_READ_COUNT) {
136
              pm avgpm2 5 = \exp((pm \text{ 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:
              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
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;
              break:
173
          case 20:
174
              packetdata.countPM0_5um = current_data;
175
              break;
176
          case 22:
              packetdata.countPM1_0um = current_data;
178
              break;
179
              packetdata.countPM2 5um = current data;
180
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;
          case 30:
192
              current_data = frame_buffer[frame_count-1];
193
194
              packetdata.error = current data;
195
              break;
196
          case 32:
197
              packetdata.checksum = current_data;
198
              byte_sum -= ((current_data>>8)+(current_data&OxFF));
199
              break;
          default:
              break;
202
203
      }
204
205
      void PM 7003::print messages(void){
206
```

```
* Print messages to string and Serial screen
208
               Serial.println("----");
209
               Serial.print("PMS 7003 - Reading #");
Serial.println(read_count);
               Serial.println("----");
sprintf(print_buffer, ", %02x, %02x, %04x, ",
212
213
               packetdata.start_frame[0], packetdata.start_frame[1], packetdata.frame_length); sprintf(print_buffer, "%s%04d, %04d, %04d, ", print_buffer, packetdata.concPM1_0_factory, packetdata.concPM2_5_factory, packetdata.
214
215
216
                      concPM10_0_factory);
               sprintf(print_buffer, "%s%04d, %04d, %04d, ", print_buffer,
    packetdata.concPM1_0_ambient, packetdata.concPM2_5_ambient, packetdata.
218
               concPM10_0 ambient);
sprintf(print_buffer, "%s%04d, %04d, %04d, %04d, %04d, %04d, ", print_buffer,
219
               packetdata.countPM0_3um, packetdata.countPM5_0um, packetdata.countPM10_0um, packetdata.countPM2_5um, packetdata.countPM5_0um, packetdata.countPM10_0um); sprintf(print_buffer, "%s%02d, %02d, ", print_buffer,
222
                      packetdata.version, packetdata.error);
223
224
               pm2_5 = packetdata.countPM1_0um - packetdata.countPM2_5um + packetdata.countPM0_5um - packetdata.countPM1_0um + packetdata.countPM0_3um - packetdata.countPM0_5um;
225
226
               Serial.println(print buffer);
               Serial.println("----
228
               delay(500);
229
230
```

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

'MHZ19.cpp' and 'MHZ19.h'

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

```
* This is the .cpp file for the MH-Z19 CO2 Sensor
      * This code was exclusively written by MECH 45X Team 26
 4
     #ifndef MHZ19 H
     #define MHZ19 H
     #define MHZ19 ZEROTH BYTE 0xFF
     #define MHZ19 FIRST BYTE 0x86
10
     #define MAX FRAME LEN 9
     #define NUMBER_OF_VALUES 5
#define CO2_START_UP_TIME 210
#define MAX_FRAME_READ_COUNT 40
11
13
     #define MAX_FUNCTION_CALL_COUNT 1
#include "WProgram.h"
14
15
    #include "Time.h"
16
18
19
    class MHZ19 {
         public:
              MHZ19();
22
              virtual ~MHZ19();
              int get co2 reading(void);
              int get_co2_ave(void);
void set_transistor(int pin);
24
25
26
              bool make_sensor_read(void);
27
              void calibrate sensor(void);
              void reset_co2_ave(void);
29
         private:
30
31
              char frame buffer[MAX FRAME LEN];
              const uint8 t mhz19_read_command[MAX_FRAME_LEN] = \{0xFF,0x01,0x86,0x00,0x00,0x00,0x00,0x00,0x79\};;
34
              bool debug = false;
36
              bool sync_state;
              bool does sensor work;
38
              bool is average taken;
39
              bool first_time;
40
              int co2_transistor_control;
41
42
              int frame_sync_count;
              int frame_read_count;
43
              int byte_sum;
44
45
              int current byte;
46
              int drain;
              int co2_ppm;
48
              int co2_ppm_average;
49
              int reading_count;
50
              int function_call_count;
51
              int mhz19_buffer[NUMBER_OF_VALUES];
53
              bool run_sensor(void);
              void frame_sync(void);
54
55
              void read_sensor(void);
56
              void serial drain(void);
              void fill frame buffer(void);
58
              void add_to_ave_buf(void);
              void print_current_reading(void);
59
60
              void calculate_average_reading(void);
61
              void print_average_reading(void);
62
              void take_average(void);
63
64
              //Timer
              time_t start_time;
65
              time_t current_time;
time_t duration;
66
```

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

#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"
#include "Time.h"
    MHZ19::MHZ19() {
        first_time = true;
10
11
    MHZ19::~MHZ19() {
13
14
15
16
    void MHZ19::set_transistor(int pin) {
         * Set transistor pin
18
         * set pinMode for transistor pin
19
        co2_transistor_control = pin;
22
        pinMode (co2 transistor control, OUTPUT);
23
24
25
    void MHZ19::begin_timer(void) {
26
         * Turn transistor on
27
         * Save time at which transistor is turned on
28
         * Time is used for timing purposes
29
         * change first_time to false
30
31
         * first_time indicates whether or not timer has been started
         * and transistor has been turned on
34
35
        co2_ppm_average = -1;
         digitalWrite(co2_transistor_control, HIGH);
36
37
         start_time = now();
        Serial.println("----
         Serial.print("CO2 start time: ");
39
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
         * return true if enough time has passed
49
         * else false
50
51
        current_time = now();
52
         duration = current_time - start_time;
        Serial.println("----");
54
        Serial.print("CO2 Duration: ");
        Serial.println(duration);
5.5
        Serial.println("----");
56
57
         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
         * turn transistor on and start timer if this hasn't already been done
66
         * read from sensor if enough time has passed
67
         * return true if enough measurements have been taken
         * else false
```

```
71
         if(first_time) {
             function call count = 0;
73
             begin_timer();
             return(false);
75
76
         else if(function call count < MAX FUNCTION CALL COUNT) {</pre>
             if(check_begin_reading()) {
                 Serial.println("-----
78
79
                 Serial.print("Function Call Count: ");
                 Serial.println(function_call_count);
80
81
                 Serial.println("----");
                 run sensor();
                 function call count ++;
83
84
             } else {return(false);}
8.5
86
87
88
         if(function_call_count >= MAX_FUNCTION_CALL_COUNT) {
29
             first_time = true;
90
             digitalWrite(co2_transistor_control, LOW);
91
             return(true);
         } else{return(false);}
93
     }
94
     void MHZ19::calibrate_sensor(void) {
95
96
          * Turn sensor on and wait for warm-upper_bound
          * Following warm-up, read forever
98
99
         if(first_time) {
             function_call_count = 0;
             begin_timer();
103
         1
104
         if(check_begin_reading()) {
106
             Serial.println("----");
107
             Serial.print("Function Call Count: ");
108
             Serial.println(function_call_count);
             Serial.println("----");
109
110
             run sensor();
111
             function call count ++;
113
     1
114
115
     bool MHZ19::run sensor(void) {
116
          * Run the MHZ19 sensor
117
          * Set ppm to zero
118
          * clear the frame_buffer
119
          \star drain the serial buffer
          * read from the sensor
          * print reading
123
          * add the reading to the average value buffer
          * calculate average value
124
          */
125
126
         co2 ppm = -1;
127
         co2 ppm average = 0;
128
         is_average_taken = false;
129
         does_sensor_work = true;
         reading_count = 1;
131
         serial drain();
134
         while(is_average_taken == false && does_sensor_work == true) {
             memset(frame buffer, 0, 9);
136
             read_sensor();
             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
      void MHZ19::print_current_reading(void) {
147
148
           * Prints current reading if reading is valid (i.e. co2_ppm > 0)
149
           * and if the maximum number of readings haven't been exceeded
151
          if(co2_ppm > 0) {
   Serial.print("MHZ19 CO2 PPM Reading ");
152
153
154
            Serial.print(reading_count);
155
            Serial.print(": ");
            Serial.println(co2 ppm);
156
157
          1
158
          else {
159
            Serial.println("Error reading CO2 PPM from MHZ19");
160
161
162
163
      void MHZ19::add_to_ave_buf(void) {
164
           * IF a valid value of co2 is read and the number of reading is less than the max,
165
           * THEN add current value to buffer
166
167
          if(co2_ppm > 0 && reading_count <= NUMBER OF VALUES) {</pre>
168
              mhz19_buffer[reading_count - 1] = co2_ppm;
169
170
              reading_count += 1;
171
172
      1
173
174
      void MHZ19::calculate_average_reading(void) {
175
176
           * IF the number of readings exceeds the number of values to be read,
           * THEN calculate the average
177
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
           * IF the average has been taken (co2_ppm_average > 0)
190
191
           * THEN print the average
192
193
          if(co2_ppm_average > 0) {
    Serial.println("-----
194
195
              Serial.print("CO2 PPM Average Reading: ");
196
              Serial.println(co2 ppm average);
197
              Serial.println("--
198
199
      }
      void MHZ19::read sensor(void) {
           * Start Serial1 connection
           ^{\star} \, Send command to read from sensor to the sensor
204
205
           * Read from the sensor (fill_from_buffer();)
206
           * Calculate PPM for CO2
207
           * End Serial connection
```

```
*/
208
209
210
         Serial1.begin(9600);
          Serial1.write(mhz19_read_command, 9);
          delay(1000);
213
          fill frame buffer();
214
          co2 ppm = 256*frame buffer[2] + frame buffer[3];
215
          Serial1.end();
216
     1
      void MHZ19::serial drain(void) {
218
219
         * Drains serial buffer when sensor is turned on
221
         while (Serial1.available() > 0) {
              drain = Serial1.available();
224
              Serial.print("-- Draining buffer: ");
225
226
     1
228
     void MHZ19::frame_sync(void) {
229
          * Sync frames so that frames are added to the frame buffer in the correct order
230
           * IF correct byte is read, THEN add to buffer and move on to next byte
           * ELSE read byte and discard
          \star IF no bytes are available to read and the frames have not been synced, THEN send
233
           command to read from sensor again
235
          * frame sync count keeps track of how many frames are added to frame buffer
           * frame_read_count keeps track of how many frames are read but not added to buffer
236
           (fails if too many frames read)
          */
238
          sync state = false;
         frame_sync_count = 0;
frame_read_count = 0;
239
240
241
         byte_sum = 0;
242
          while (!sync state && Seriall.available() > 0 && frame read count <
243
          MAX FRAME READ COUNT) {
2.44
              current_byte = Serial1.read();
245
246
              if (current byte == MHZ19 ZEROTH BYTE && frame sync count == 0) {
                  frame buffer[frame_sync_count] = current_byte;
247
248
                  byte sum = current byte;
249
                  frame_sync_count = 1;
251
              else if (current_byte == MHZ19_FIRST_BYTE && frame_sync_count == 1) {
                  frame buffer[frame sync count] = current byte;
253
                  byte sum += current byte;
254
                  sync state = true;
255
                  frame_sync_count = 2;
256
257
              else {
258
                  if(debug) {
                      Serial.print("-- Frame syncing... ");
259
                      Serial.println(current_byte, HEX);
261
262
263
                  frame_read_count ++;
2.64
             1
265
266
              if (!sync_state && !(Serial1.available() > 0) && frame_read_count <</pre>
             MAX_FRAME_READ_COUNT) {
                  Serial1.write(mhz19_read_command, 9);
267
268
269
                  if(debug) {
                      Serial.println("----");
270
                      Serial.println("Read command has been sent to CO2 sensor");
271
                      Serial.println("-----");
272
```

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

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:

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"
10
11
     #endif
13
     #include <Wire.h>
14
15
16
       I2C ADDRESS/BITS
17
        #define CCS811 ADDRESS
18
                                             (0x5A)
19
       #define MAX_READ_COUNT 5
22
        #define MAX ERROR COUNT 5
23
24
     25
        REGISTERS
26
        -----*/
27
        enum
28
       {
             CCS811_STATUS = 0 \times 00,
CCS811_MEAS_MODE = 0 \times 01,
29
30
31
            CCS811\_ALG\_RESULT\_DATA = 0x02,
           CCS811_RAW_DATA = 0 \times 03,
CCS811_ENV_DATA = 0 \times 05,
32
           CCS811_NTC = 0x06,

CCS811_THRESHOLDS = 0x10,

CCS811_BASELINE = 0x11,

CCS811_HW_ID = 0x20,
34
35
36
37
            CCS811 - W VERSION = 0 \times 21,
            CCS811_FW_BOOT_VERSION = 0x23,
CCS811_FW_APP_VERSION = 0x24,
39
40
             CCS811\_ERROR\_ID = 0xE0,
41
42
             CCS811 SW RESET = 0xFF,
43
         };
44
       //bootloader registers
45
46
       enum
47
       {
        CCS811 BOOTLOADER APP ERASE = 0xF1,
        CCS811 BOOTLOADER APP DATA = 0xF2,
CCS811 BOOTLOADER APP VERIFY = 0xF3,
49
50
51
        CCS811_BOOTLOADER_APP_START = 0xF4
52
53
54
       enum
5.5
        CCS811_DRIVE_MODE_IDLE = 0x00,
CCS811_DRIVE_MODE_1SEC = 0x01,
56
57
         CCS811DRIVEMODE_10SEC = 0x02,
        CCS811 DRIVE MODE 60SEC = 0x03,
CCS811 DRIVE MODE 250MS = 0x04,
59
60
61
62
63
     /*----*/
64
65
     #define CCS811_HW_ID_CODE
                                  0x81
66
67
     #define CCS811_REF_RESISTOR 100000
68
```

```
@brief Class that stores state and functions for interacting with CCS811 gas
        sensor chips
72
7.3
     class Adafruit CCS811 {
74
75
      public:
76
        //constructors
77
        Adafruit_CCS811(void) {};
78
        ~Adafruit CCS811(void) {};
79
80
        bool start voc(void);
        bool run voc (void);
81
        float get_eCO2_ave(void);
82
            float get_TVOC_ave(void);
83
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();
92
        void setThresholds(uint16_t low_med, uint16_t med_high, uint8_t hysteresis = 50);
93
94
        void SWReset();
95
        void setDriveMode(uint8 t mode);
97
        void enableInterrupt();
98
        void disableInterrupt();
99
                       *****************
102
               @brief returns the stored total volatile organic compounds measurement.
               This does does not read the sensor. To do so, call readData()
               @returns TVOC measurement as 16 bit integer
104
105
106
        uint16_t getTVOC() { return _TVOC; }
            108
109
            /*!
               @brief returns the stored estimated carbon dioxide measurement. This does
               does not read the sensor. To do so, call readData()
               @returns eCO2 measurement as 16 bit integer
113
114
        uint16_t geteCO2() { return _eCO2; }
115
            116
            /*!
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.
            122
        void setTempOffset(float offset) { tempOffset = offset; }
123
124
        //check if data is available to be read
        bool available();
126
        uint8 t readData();
127
128
        bool checkError();
129
      private:
        float eCO2 buf[MAX READ COUNT];
        float TVOC_buf[MAX_READ_COUNT];
        float eCO2 ave;
134
        float TVOC ave;
```

```
135
         void read_voc(void);
136
         void fill_buffer(void);
         void print readings(void);
137
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
         uint8_t i2caddr;
144
145
         float _tempOffset;
146
147
         uint16_t _TVOC;
148
         uint16_t _eCO2;
149
          void
                  write8(byte reg, byte value);
151
          void
                   write16(byte reg, uint16_t value);
            uint8 t read8(byte reg);
152
153
         void read(uint8_t reg, uint8_t *buf, uint8_t num);
void write(uint8_t reg, uint8_t *buf, uint8_t num);
154
155
156
          void _i2c_init();
157
158
     /*-----
159
      REGISTER BITFIELDS
160
         _____*/
161
         // The status register
162
             struct status {
163
                  /* 0: no error
164
                  * 1: error has occurred
165
166
167
                  uint8_t ERROR: 1;
168
169
                 // reserved : 2
170
171
                  /* 0: no samples are ready
172
                  * 1: samples are ready
173
                  uint8_t DATA_READY: 1;
174
175
                 uint8 t APP VALID: 1;
176
177
           // reserved : 2
178
179
                  /* 0: boot mode, new firmware can be loaded
                  ^{\star} 1: application mode, can take measurements
180
181
182
                  uint8_t FW_MODE: 1;
183
184
                  void set(uint8_t data){
185
                   ERROR = data & 0 \times 01;
                   DATA_READY = (data >> 3) & 0x01;
APP VALID = (data >> 4) & 0x01;
186
187
188
                   FW \overline{\text{MODE}} = (\text{data} >> 7) \& 0 \times 01;
189
                  }
190
              1:
191
              status _status;
192
              //measurement and conditions register
193
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)
                 if the new
199
              ALG RESULT DATA crosses one of the thresholds set in the THRESHOLDS register
              by more than the hysteresis value (also in the THRESHOLDS register)
202
                uint8 t INT THRESH: 1;
```

```
/* 0: int disabled
  * 1: The nINT signal is asserted (driven low) when a new sample is ready in
204
205
               ALG_RESULT_DATA. The nINT signal will stop being driven low when ALG_RESULT_DATA is read on the I^2C interface.
206
208
209
                 uint8 t INT DATARDY: 1;
                 uint8_t DRIVE_MODE: 3;
213
                 uint8_t get(){
214
                   return (INT THRESH << 2) | (INT DATARDY << 3) | (DRIVE MODE << 4);
215
216
               1:
               meas_mode _meas_mode;
218
219
               struct error_id {
                 /* The CCS\overline{8}11 received an I^2C write request addressed to this station but with
220
             invalid register address ID */
221
                 uint8_t WRITE_REG_INVALID: 1;
224
                  /* The CCS811 received an I²C read request to a mailbox ID that is invalid */
225
                 uint8 t READ REG INVALID: 1;
226
                 /\star The CCS811 received an \ensuremath{\text{I}}^2\ensuremath{\text{C}} request to write an unsupported mode to
             MEAS MODE */
228
229
                 uint8 t MEASMODE INVALID: 1;
                 /\star The sensor resistance measurement has reached or exceeded the maximum
             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
                  /* The Heater voltage is not being applied correctly */
238
                 uint8_t HEATER_SUPPLY: 1;
239
240
                 void set(uint8 t data){
241
                    WRITE REG INVALID = data & 0 \times 01;
2.42
                    READ_REG_INVALID = (data & 0x02) >> 1;
243
244
                    MEASMODE INVALID = (data & 0 \times 04) >> 2;
245
                    MAX RESISTANCE = (data & 0 \times 08) >> 3;
                    HEATER FAULT = (data & 0 \times 10) >> 4;
246
                    HEATER SUPPLY = (data & 0x20) >> 5;
247
248
                 - }
249
               error_id _error_id;
251
252
253
      };
254
      #endif
256
```

```
* 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 45% 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.
10
11
     * Part 1 begins...
13
14
     #include "CCS821.h"
15
16
     17
18
19
        @brief Setups the I2C interface and hardware and checks for communication.
        {\tt @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
22
23
24
    bool Adafruit_CCS811::begin(uint8_t addr)
25
      i2caddr = addr;
26
27
28
      _i2c_init();
29
      SWReset();
30
      delay(100);
      //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
      this->write (CCS811 BOOTLOADER APP START, NULL, 0);
39
      delay(100);
40
41
      //\text{make} sure there are no errors and we have entered application mode
42
      if(checkError()) return false;
      if(!_status.FW_MODE) return false;
43
44
45
      disableInterrupt();
46
47
      //default to read every second
48
      setDriveMode (CCS811 DRIVE MODE 1SEC);
49
50
      return true:
51
    }
52
     53
54
    /*!
        @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.
5.5
56
57
58
    void Adafruit CCS811::setDriveMode(uint8 t mode)
59
60
       meas mode.DRIVE MODE = mode;
61
      this->write8(CCS811_MEAS_MODE, _meas_mode.get());
62
63
64
    /*!
65
66
        @brief enable the data ready interrupt pin on the device.
67
```

```
69
    void Adafruit_CCS811::enableInterrupt()
71
       meas mode.INT DATARDY = 1;
72
      this->write8(CCS811_MEAS_MODE, _meas_mode.get());
74
     /*****************************
75
76
77
        @brief disable the data ready interrupt pin on the device
78
79
80
     void Adafruit CCS811::disableInterrupt()
81
     {
       meas mode.INT DATARDY = 0;
82
      this->write8(CCS811_MEAS_MODE, _meas_mode.get());
83
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
    1
99
     @brief read and store the sensor data. This data can be accessed with getTVOC()
        and geteCO2()
        @returns 0 if no error, error code otherwise.
104
     105
106
     uint8 t Adafruit CCS811::readData()
108
      if(!available())
109
        return false;
      else{
        uint8 t buf[8];
        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
       if ( status.ERROR)
118
        return buf[5];
119
        else return 0;
121
      }
    }
124
125
126
        @brief set the humidity and temperature compensation for the sensor.
        @param humidity the humidity data as a percentage. For 55% humidity, pass in
        integer 55.
128
        @param temperature the temperature in degrees C as a decimal number. For 25.5
        degrees C, pass in 25.5
129
     void Adafruit CCS811::setEnvironmentalData(uint8 t humidity, double temperature)
     {
      /* 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
       degrees; there is an offset: 0 maps to -25\Box C. The default value is
138
139
       25\Box 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
       not set by the application) to compensate for changes in
142
       relative humidity and ambient temperature.*/
143
144
145
       uint8 t hum perc = humidity << 1;</pre>
146
147
       float fractional = modf(temperature, &temperature);
       uint16_t temp_high = (((uint16_t)temperature + 25) << 9);
uint16_t temp_low = ((uint16_t)(fractional / 0.001953125) & 0x1FF);</pre>
148
149
151
       uint16_t temp_conv = (temp_high | temp_low);
152
       uint8 t buf[] = {hum_perc, 0x00,
153
154
         (uint8_t)((temp_conv >> 8) & 0xff), (uint8_t)(temp_conv & 0xff));
155
156
       this->write(CCS811 ENV DATA, buf, 4);
157
158
     1
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);
       uint32_t vref = ((uint32_t)buf[0] << 8) | buf[1];
uint32 t vntc = ((uint32_t)buf[2] << 8) | buf[3];</pre>
171
172
173
       //from ams ccs811 app note
174
175
       uint32 t rntc = vntc * CCS811 REF RESISTOR / vref;
176
177
       double ntc temp;
178
       ntc temp = log((double)rntc / CCS811 REF RESISTOR); // 1
       ntc_temp /= 3380; // 2
179
       ntc_{temp} += 1.0 / (25 + 273.15); // 3
180
181
       ntc_temp = 1.0 / ntc_temp; // 4
182
       ntc temp -= 273.15; // 5
       return ntc_temp - _tempOffset;
183
184
185
186
      187
188
     /*!
         \textbf{@brief} \quad \texttt{set interrupt thresholds}
189
         @param low_med the level below which an interrupt will be triggered.
190
191
         @param med high the level above which the interrupt will ge triggered.
         @param hysteresis optional histeresis level. Defaults to 50
192
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 \gg 8) & 0xF), (uint8 t)(low med & 0xF),
198
       (uint8_t)((med_high \gg 8) & 0xF), (uint8_t)(med_high & 0xF), hysteresis);
       this->write(CCS811_THRESHOLDS, buf, 5);
```

```
204
205
       @brief trigger a software reset of the device
206
208
    void Adafruit CCS811::SWReset()
209
    {
     //reset sequence from the datasheet
      uint8_t seq[] = \{0x11, 0xE5, 0x72, 0x8A\};
     this->write(CCS811 SW RESET, seq, 4);
    }
213
214
    215
    /*!
216
       \ensuremath{\mathfrak{Q}\mathbf{brief}} \; read the status register and store any errors.
       @returns the error bits from the status register of the device.
218
219
    220
221
    bool Adafruit CCS811::checkError()
      _status.set(read8(CCS811_STATUS));
224
     return _status.ERROR;
225
226
    228
229
       @brief write one byte of data to the specified register
       @param reg the register to write to
       @param value the value to write
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
       @param reg the register to read
2.42
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);
251
     return ret;
252
254
    void Adafruit_CCS811::_i2c_init()
255
    {
256
     Wire.begin();
2.57
2.58
259
    void Adafruit CCS811::read(uint8 t reg, uint8 t *buf, uint8 t num)
260
    {
261
      uint8 t value;
      uint8_t pos = 0;
2.62
263
264
      //on arduino we need to read in 32 byte chunks
265
      while(pos < num) {</pre>
266
       uint8_t read_now = min((uint8_t)32, (uint8_t)(num - pos));
267
       Wire.beginTransmission((uint8_t)_i2caddr);
268
269
       Wire.write((uint8_t)reg + pos);
270
       Wire.endTransmission();
271
       Wire.requestFrom((uint8 t) i2caddr, read now);
```

```
273
          for(int i=0; i<read_now; i++){</pre>
274
            buf[pos] = Wire.read();
275
            pos++;
2.76
          }
277
       }
278
      }
279
280
      void Adafruit_CCS811::write(uint8_t reg, uint8_t *buf, uint8_t num)
281
       Wire.beginTransmission((uint8_t)_i2caddr);
282
        Wire.write((uint8_t)reg);
Wire.write((uint8_t *)buf, num);
283
284
285
        Wire.endTransmission();
286
287
288
      * Part 2: code written by team 26
289
      * This code was written by Team 26
290
       \star This code is properly commented
291
2.92
293
294
      bool Adafruit CCS811::start voc(void) {
295
           * Start voc sensor using the library's begin() function
296
           * If sensor is started, calibrate temperature
297
298
299
          Serial.println("Trying to start VOC Sensor...");
          if(!begin()){
              Serial.println("Failed to start CC2821 VOC sensor! Wiring is likely incorrect.");
301
302
              return false;
304
          else {
305
              Serial.println("Successfully started VOC Sensor!");
306
              delay (5000);
              return true;
308
309
     }
      bool Adafruit_CCS811::run_voc(void) {
312
           * Run the VOC sensor
           ^{\star} Take measurements until enough measurements have been taken to calculate the
314
           average
           * use read_voc() to read from sensor
315
           */
316
          is_average_taken = false;
318
          read count = 1;
319
          error count = 0;
320
          while(is_average_taken == false && error_count < MAX_ERROR_COUNT) {read_voc();}</pre>
          if(is_average_taken) {return true;}
          else if(error count >= MAX ERROR COUNT) {return false;}
324
      1
325
      void Adafruit_CCS811::read_voc(void) {
326
327
328
           * Read values from <a href="voc">voc</a> sensor
329
           * IF data is read and max read count has not been exceed
           * THEN fill_buffer and print_readings and read_count ++
330
           * calculate_average_reading
331
332
           * print_average_reading
334
          if(available()){
335
              float temp = calculateTemperature();
              if(!readData() && read count <= MAX READ COUNT){</pre>
336
                   fill_buffer();
338
                  print readings();
339
                  read_count += 1;
```

```
340
                  error_count = 0;
341
              }
342
              else {
343
                error_count ++;
                Serial.print("ERROR #");
344
345
                Serial.println(error count);
346
                delay(500);
347
348
          1
349
          calculate average reading();
          print_average_reading();
351
      void Adafruit CCS811::fill_buffer(void) {
354
          \star add new values to buffers
355
356
          eCO2 buf[read count-1] = geteCO2();
358
          TVOC_buf[read_count-1] = getTVOC();
359
361
      void Adafruit CCS811::print readings(void) {
362
           * Print readings
363
364
          Serial.print("VOC Reading #:");
366
          Serial.print(read count);
367
          Serial.print(", CO2: ");
368
          Serial.print(geteCO2());
          Serial.print("ppm, TVOC: ");
369
          Serial.print(getTVOC());
371
          Serial.println("pph");
372
373
374
     void Adafruit_CCS811::calculate_average_reading(void) {
375
          * Calculate the average reading if enough readings have been taken
376
377
378
          if(read count > MAX READ COUNT) {
              eCO\overline{2}_ave = 0;
379
380
              TVOC ave = 0;
381
              for(int k = 0; k < MAX READ COUNT; k++) {</pre>
                 eCO2_ave += eCO2_buf[k];
382
                  TVOC_ave += TVOC_buf[k];
383
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
          * print average reading values
395
396
397
          if(is average taken) {
              Serial.println("----");
398
              Serial.println("VOC Sensor Average Readings:");
399
              Serial.println("----");
400
              Serial.print("CCS eCO2 Average: ");
401
              Serial.println(eCO2_ave);
Serial.print("CCS_TVOC_Average: ");
402
403
404
              Serial.println(TVOC_ave);
405
          }
406
407
     // 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

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

```
/*
 * .h file for SHT35D
       #ifndef SHT35D
       #define SHT35D
       #define MAX READ COUNT 5
       #define MAX ERROR COUNT 5
       #define ADDR_SHT 0x45
10
11
       #include <Arduino.h>
13
       //List of Commands for SHT35D Sensor:
       typedef enum {
14
             SHT3XD_CMD_READ_SERIAL_NUMBER = 0x3780,
15
16
17
             SHT3XD_CMD_READ_STATUS = 0xF32D,
             SHT3XD CMD CLEAR STATUS = 0 \times 3041,
18
19
             SHT3XD_CMD_HEATER_ENABLE = 0 \times 306D,
             SHT3XD_CMD_HEATER_DISABLE = 0 \times 3066,
23
             SHT3XD CMD SOFT RESET = 0x30A2,
24
25
             SHT3XD\_CMD\_CLOCK\_STRETCH\_H = 0x2C06,
26
             SHT3XD\_CMD\_CLOCK\_STRETCH\_M = 0x2COD,
27
             SHT3XD CMD CLOCK STRETCH L = 0x2C10,
28
             \begin{array}{lll} \text{SHT3XD\_CMD\_POLLING\_H} &=& 0 \times 2400 \,, \\ \text{SHT3XD\_CMD\_POLLING\_M} &=& 0 \times 2408 \,, \end{array}
29
             SHT3XD CMD POLLING L = 0x2416,
             SHT3XD CMD ART = 0x2B32,
34
3.5
             SHT3XD_CMD_PERIODIC_HALF_H = 0x2032,
            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,
37
39
40
41
             SHT3XD CMD PERIODIC 2 M = 0x2220,
42
             SHT3XD_CMD_PERIODIC_2_L = 0x222B,
43
             SHT3XD_CMD_PERIODIC_4_H = 0x2334,
SHT3XD_CMD_PERIODIC_4_M = 0x2322,
SHT3XD_CMD_PERIODIC_4_L = 0x2329,
44
45
46
             SHT3XD CMD PERIODIC 10 H = 0x2737,
SHT3XD CMD PERIODIC 10 M = 0x2721,
47
49
             SHT3XD CMD PERIODIC 10 L = 0x272A,
50
             SHT3XD_CMD_FETCH_DATA = 0xE000,
SHT3XD_CMD_STOP_PERIODIC = 0x3093,
51
52
             SHT3XD_CMD_READ_ALR_LIMIT_LS = 0xE102,
SHT3XD_CMD_READ_ALR_LIMIT_LC = 0xE109,
SHT3XD_CMD_READ_ALR_LIMIT_HS = 0xE11F,
54
5.5
56
57
             SHT3XD CMD READ ALR LIMIT HC = 0xE114,
             SHT3XD CMD WRITE ALR LIMIT HS = 0 \times 611D,
             SHT3XD CMD WRITE ALR LIMIT HC = 0x6116, SHT3XD CMD WRITE ALR LIMIT LC = 0x610B,
59
60
61
             SHT3XD CMD WRITE ALR LIMIT LS = 0 \times 6100,
62
63
             SHT3XD CMD NO SLEEP = 0 \times 303E,
64
       } SHT31D_Commands;
65
       // List of repeatability options for SHT35D:
66
67
       typedef enum {
68
          SHT3XD REPEATABILITY HIGH,
          SHT3XD REPEATABILITY MEDIUM,
```

```
SHT3XD_REPEATABILITY_LOW,
 71
       } SHT31D_Repeatability;
 72
 73
       // List of modes:
typedef enum {
 74
 75
          SHT3XD MODE CLOCK STRETCH,
 76
          SHT3XD MODE POLLING,
 77
       } SHT31D Mode;
 78
 79
       // List of frequency choices
 80
       typedef enum {
         SHT3XD FREQUENCY HZ5,
SHT3XD FREQUENCY 1HZ,
SHT3XD FREQUENCY 2HZ,
SHT3XD FREQUENCY 4HZ,
 81
 82
 83
 84
       SHT3XD_FREQUENCY_10HZ
} SHT31D_Frequency;
 8.5
 86
 87
 88
       // List of errors:
       typedef enum {
 89
 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,
         SHT3XD PARAM_WRONG_FREQUENCY = -503,
SHT3XD_PARAM_WRONG_ALERT = -504,
 97
 98
 99
         // 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,
103
104
       SHT3XD_WIRE_I2C_UNKNOW_ERROR = -40
} SHT31D_ErrorCode;
106
107
108
       // List of statuses:
109
       typedef union {
110
            uint16 t rawData;
111
            struct {
                uint8 t WriteDataChecksumStatus : 1;
                 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;
                 uint8_t RH_TrackingAlert : 1;
uint8_t Reserved2 : 1;
118
119
                 uint8_t HeaterStatus : 1;
uint8_t Reserved3 : 1;
uint8_t AlertPending : 1;
122
123
            1:
124
       } SHT31D_RegisterStatus;
125
126
       struct SHT31D {
127
          /*
             * Structure for SHT31D
128
             * t - temperature
129
             * rh - relative humidity
             * error - error of type SHT31D_ErrorCode
131
             */
            float t;
float rh;
134
            SHT31D ErrorCode error;
135
136
137
       class ClosedCube SHT31D {
```

```
139
      * Class definition for ClosedCube_SHT31D
140
141
      public:
142
143
          ClosedCube_SHT31D();
144
145
          bool start sht(void);
          bool run_sht(void);
146
          float get_t_ave(void);
147
          float get rh ave (void);
148
149
150
151
          SHT31D_ErrorCode begin(uint8_t address);
          SHT31D ErrorCode clearAll();
152
153
          SHT31D RegisterStatus readStatusRegister();
154
155
          SHT31D_ErrorCode heaterEnable();
          SHT31D ErrorCode heaterDisable();
156
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);
          SHT31D readTempAndHumidity (SHT31D Repeatability repeatability, SHT31D Mode mode,
166
          uint8 t timeout);
167
          SHT31D readTempAndHumidityClockStretch(SHT31D Repeatability repeatability);
168
          SHT31D readTempAndHumidityPolling(SHT31D Repeatability repeatability, uint8 t
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);
          SHT31D readAlertLowSet();
179
180
          SHT31D readAlertLowClear();
181
182
      private:
183
          float t buf[MAX READ COUNT];
          float rh_buf[MAX_READ_COUNT];
184
185
          bool is_average_taken;
186
          int read count;
187
          int error count;
188
          float t_average;
189
          float rh_average;
190
191
          SHT31D save to buffer (SHT31D result);
192
          SHT31D read sht (void);
193
          void calculate_average(void);
194
195
          uint8_t _address;
196
          SHT31D RegisterStatus status;
197
          SHT31D ErrorCode writeCommand(SHT31D Commands command);
198
          SHT31D ErrorCode writeAlertData(SHT31D Commands command, float temperature, float
          humidity);
201
          uint8 t checkCrc(uint8 t data[], uint8 t checksum);
```

```
202
203
             uint8_t calculateCrc(uint8_t data[]);
204
             float calculateHumidity(uint16 t rawValue);
205
             float calculateTemperature(uint16_t rawValue);
             uint16_t calculateRawHumidity(float value);
uint16_t calculateRawTemperature(float value);
207
208
209
             SHT31D readTemperatureAndHumidity();
SHT31D readAlertData(SHT31D_Commands command);
SHT31D_ErrorCode read(uint16_t* data, uint8_t numOfPair);
211
212
213
214
             SHT31D returnError(SHT31D_ErrorCode command);
215
        };
216
217
        #endif
```

```
* This is the .cpp file for the SHT35D Temperature
     * and relative humidity sensor.
     * Part 1 of this code was retrieved online:
     * https://github.com/closedcube/ClosedCube SHT31D Arduino
     * Part 2 was written by MECH 45X Team 26
     * Part 1 begins...
10
11
     #include <Wire.h>
13
     #include "SHT35D.h"
14
15
16
    ClosedCube_SHT31D::ClosedCube_SHT31D()
17
18
19
     SHT31D_ErrorCode ClosedCube_SHT31D::begin(uint8_t address) {
         SHT31D ErrorCode error = SHT3XD NO ERROR;
22
         address = address;
23
         return error;
24
25
     SHT31D ClosedCube SHT31D::periodicFetchData()
26
27
28
         SHT31D ErrorCode error = writeCommand(SHT3XD CMD FETCH DATA);
         if (error == SHT3XD NO ERROR)
29
30
             return readTemperatureAndHumidity();
         else
31
32
             returnError(error);
34
35
     SHT31D ErrorCode ClosedCube SHT31D::periodicStop() {
36
         return writeCommand(SHT3XD_CMD_STOP_PERIODIC);
37
     SHT31D ErrorCode ClosedCube SHT31D::periodicStart(SHT31D Repeatability repeatability,
     SHT31D_Frequency frequency)
40
41
         SHT31D ErrorCode error;
42
43
         switch (repeatability)
44
45
         case SHT3XD REPEATABILITY LOW:
46
             switch (frequency)
47
48
             case SHT3XD FREQUENCY HZ5:
                 error = writeCommand(SHT3XD CMD PERIODIC HALF L);
49
50
                break;
51
             case SHT3XD_FREQUENCY_1HZ:
                error = writeCommand(SHT3XD CMD PERIODIC 1 L);
53
                break:
             case SHT3XD FREQUENCY 2HZ:
54
5.5
                 error = writeCommand(SHT3XD_CMD_PERIODIC_2_L);
56
                 break;
57
             case SHT3XD FREQUENCY 4HZ:
58
                 error = writeCommand(SHT3XD CMD PERIODIC 4 L);
59
                 break;
60
             case SHT3XD FREQUENCY 10HZ:
                 error = writeCommand(SHT3XD_CMD_PERIODIC_10_L);
61
62
                 break;
63
             default:
64
                 error = SHT3XD PARAM WRONG FREQUENCY;
65
                 break;
66
         case SHT3XD REPEATABILITY MEDIUM:
```

```
switch (frequency)
 70
              case SHT3XD FREQUENCY HZ5:
                  error = writeCommand(SHT3XD_CMD_PERIODIC_HALF_M);
                  break:
 74
              case SHT3XD FREQUENCY 1HZ:
 75
                  error = writeCommand(SHT3XD CMD PERIODIC 1 M);
                  break;
 77
              case SHT3XD_FREQUENCY_2HZ:
 78
                  error = writeCommand (SHT3XD CMD PERIODIC 2 M);
 79
                  break;
              case SHT3XD FREQUENCY 4HZ:
                  error = writeCommand(SHT3XD_CMD_PERIODIC 4 M);
 82
                  break:
              case SHT3XD FREQUENCY 10HZ:
 83
 84
                  error = writeCommand(SHT3XD_CMD_PERIODIC_10_M);
 85
                  break;
 86
              default:
 87
                  error = SHT3XD PARAM WRONG FREQUENCY;
 88
                  break;
 89
 90
              break:
 92
         case SHT3XD_REPEATABILITY_HIGH:
 93
              switch (frequency)
 94
 95
              case SHT3XD FREQUENCY HZ5:
                  error = writeCommand(SHT3XD_CMD_PERIODIC_HALF_H);
 97
                  break;
              case SHT3XD FREQUENCY 1HZ:
 98
99
                  error = writeCommand (SHT3XD CMD PERIODIC 1 H);
                  break;
              case SHT3XD FREQUENCY 2HZ:
                  error = writeCommand(SHT3XD CMD PERIODIC 2 H);
102
103
                 break:
104
              case SHT3XD FREQUENCY 4HZ:
105
                  error = writeCommand(SHT3XD_CMD_PERIODIC_4_H);
106
                  break;
107
              case SHT3XD FREQUENCY 10HZ:
                  error = writeCommand(SHT3XD_CMD_PERIODIC_10_H);
108
109
                  break;
110
              default:
111
                  error = SHT3XD PARAM WRONG FREQUENCY;
                  break:
114
              break:
115
116
              error = SHT3XD PARAM WRONG REPEATABILITY;
117
118
              break;
119
          delay(100);
121
         return error;
122
      SHT31D ClosedCube_SHT31D::readTempAndHumidity(SHT31D_Repeatability repeatability,
124
      SHT31D Mode mode, uint8 t timeout)
125
126
          SHT31D result;
127
128
          switch (mode) {
129
          case SHT3XD_MODE_CLOCK_STRETCH:
              result = readTempAndHumidityClockStretch(repeatability);
              break;
          case SHT3XD MODE POLLING:
              result = readTempAndHumidityPolling(repeatability, timeout);
134
              break;
135
          default:
136
              result = returnError(SHT3XD PARAM WRONG MODE);
```

```
break;
138
139
          return result;
140
      }
141
142
143
      SHT31D ClosedCube SHT31D::readTempAndHumidityClockStretch(SHT31D Repeatability
      repeatability)
144
          SHT31D_ErrorCode error = SHT3XD NO ERROR;
145
146
          SHT31D Commands command;
147
148
          switch (repeatability)
149
          case SHT3XD_REPEATABILITY_LOW:
              error = writeCommand(SHT3XD_CMD_CLOCK_STRETCH_L);
152
              break;
          case SHT3XD REPEATABILITY MEDIUM:
153
154
              error = writeCommand(SHT3XD CMD CLOCK STRETCH M);
155
              break:
          case SHT3XD_REPEATABILITY_HIGH:
156
157
              error = writeCommand(SHT3XD CMD CLOCK STRETCH H);
              break;
159
          default:
160
              error = SHT3XD_PARAM_WRONG_REPEATABILITY;
161
              break;
162
          }
163
164
          delay(50);
165
          if (error == SHT3XD NO ERROR) {
166
167
              return readTemperatureAndHumidity();
          } else {
168
169
              return returnError(error);
170
171
172
173
174
      SHT31D ClosedCube SHT31D::readTempAndHumidityPolling(SHT31D Repeatability repeatability,
175
      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:
              error = writeCommand(SHT3XD_CMD_POLLING_H);
189
190
              break;
191
          default:
192
              error = SHT3XD PARAM WRONG REPEATABILITY;
193
              break;
194
          }
195
196
          delay(50);
197
          if (error == SHT3XD NO ERROR) {
198
199
              return readTemperatureAndHumidity();
          } else {
              return returnError(error);
203
```

```
204
205
206
      SHT31D ClosedCube SHT31D::readAlertHighSet() {
          return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_HS);
208
209
210
      SHT31D ClosedCube SHT31D::readAlertHighClear() {
211
          return readAlertData(SHT3XD CMD READ ALR LIMIT HC);
      SHT31D ClosedCube SHT31D::readAlertLowSet() {
214
          return readAlertData(SHT3XD CMD READ ALR LIMIT LS);
215
216
217
      SHT31D ClosedCube SHT31D::readAlertLowClear() {
218
          return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_LC);
219
221
222
      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);
          if (error == SHT3XD_NO_ERROR)
    error = writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_HC, temperatureClear,
225
226
              humidityClear);
227
228
          return error;
229
     }
231
      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);
233
          if (error == SHT3XD NO ERROR)
234
              writeAlertData(SHT3XD CMD WRITE ALR LIMIT LC, temperatureClear, humidityClear);
235
236
          return error;
237
     1
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
          1
247
          else {
248
              uint16_t rawTemperature = calculateRaWTemperature(temperature);
249
              uint16 t rawHumidity = calculateRawHumidity(humidity);
250
              uint16 t data = (rawHumidity & 0xFE00) | ((rawTemperature >> 7) & 0x001FF);
2.51
              uint8_t buf[2];
253
              buf[0] = data >> 8;
254
              buf[1] = data & 0xFF;
255
256
              uint8_t checksum = calculateCrc(buf);
258
              Wire.beginTransmission( address);
259
              Wire.write(command >> 8);
260
              Wire.write(command & OxFF);
261
              Wire.write(buf[0]);
              Wire.write(buf[1]);
263
              Wire.write(checksum);
264
              return (SHT31D ErrorCode) (-10 * Wire.endTransmission());
265
          1
```

```
266
267
          return error;
268
     1
269
271
      SHT31D ErrorCode ClosedCube SHT31D::writeCommand(SHT31D Commands command)
272
273
          Wire.beginTransmission( address);
274
          Wire.write(command >> 8);
          Wire.write(command & 0xFF);
275
276
          return (SHT31D ErrorCode) (-10 * Wire.endTransmission());
277
278
279
      SHT31D ErrorCode ClosedCube SHT31D::softReset() {
280
          return writeCommand(SHT3XD_CMD_SOFT_RESET);
281
282
      SHT31D ErrorCode ClosedCube SHT31D::generalCallReset() {
283
284
          Wire.beginTransmission(\overline{0}x0);
285
          Wire.write (0 \times 06);
286
          return (SHT31D_ErrorCode) (-10 * Wire.endTransmission());
287
288
289
      SHT31D_ErrorCode ClosedCube_SHT31D::heaterEnable() {
2.90
          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
      uint32_t ClosedCube_SHT31D::readSerialNumber()
304
          uint32 t result = SHT3XD NO ERROR;
          uint16_t buf[2];
305
306
          if (writeCommand(SHT3XD CMD READ SERIAL NUMBER) == SHT3XD NO ERROR) {
              if (read(buf, 2) == SHT3XD_NO ERROR) {
308
309
                result = (buf[0] \ll 16) | buf[1];
310
311
          1
          return result;
314
     1
315
316
      SHT31D RegisterStatus ClosedCube SHT31D::readStatusRegister()
318
          SHT31D RegisterStatus result;
319
          SHT31D_ErrorCode error = writeCommand(SHT3XD_CMD_READ_STATUS);
          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
      SHT31D ClosedCube_SHT31D::readTemperatureAndHumidity()
334
          SHT31D result;
```

```
335
336
          result.t = 0;
          result.rh = 0;
338
339
          SHT31D_ErrorCode error;
340
          uint16 t buf[2];
341
342
          if (error == SHT3XD NO ERROR)
343
              error = read(buf, \frac{1}{2});
344
          if (error == SHT3XD_NO_ERROR) {
345
346
              result.t = calculateTemperature(buf[0]);
347
              result.rh = calculateHumidity(buf[1]);
348
349
          result.error = error;
351
          return result;
352
     1
353
354
      SHT31D ClosedCube_SHT31D::readAlertData(SHT31D_Commands command)
356
          SHT31D result;
357
358
          result.t = 0;
359
          result.rh = 0;
361
          SHT31D ErrorCode error;
362
          uint16_t buf[1];
363
          error = writeCommand(command);
364
366
          if (error == SHT3XD_NO_ERROR)
367
              error = read(bu\overline{f}, \overline{1});
368
369
          if (error == SHT3XD NO ERROR) {
              result.rh = calculateHumidity(buf[0] << 7);</pre>
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;
391
          for (counter = 0; counter < numOfPair; counter++) {</pre>
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] \iff 8) \mid 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
     float ClosedCube SHT31D::calculateTemperature(uint16 t rawValue)
413
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
          return (value + 45.0f) / 175.0f * 65535.0f;
426
427
428
429
     uint16 t ClosedCube SHT31D::calculateRawHumidity(float value)
430
          return value / 100.0f * 65535.0f;
431
432
433
434
     uint8 t ClosedCube SHT31D::calculateCrc(uint8 t data[])
435
          uint8 t bit;
436
         uint8_t crc = 0xFF;
uint8_t dataCounter = 0;
437
438
439
440
          for (; dataCounter < 2; dataCounter++) {</pre>
              crc ^= (data[dataCounter]);
441
              for (bit = 8; bit > 0; --bit) {
   if (crc & 0x80){crc = (crc << 1) ^ 0x131;}</pre>
442
443
444
                  else {crc = (crc << 1);}</pre>
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
     1
458
      459
460
      // Part 2: Code Written by team 26
     // Team 26 understands this code
461
     // Therefore it is properly commented
462
463
464
     bool ClosedCube_SHT31D::start_sht(void) {
465
466
          * Start sequence for SHT35D
          * Return true: sensor was successfully started
467
          * Return false: sensor was not started
468
          \star Try to read from sensor
469
470
          * If no error, return true
          * Else return false
471
472
```

```
473
          Serial.println("Trying to start SHT sensor...");
474
          delay(500);
475
          begin (ADDR SHT); // I2C address: 0x44 or 0x45
          Serial.print("Serial #");
476
477
          Serial.println(readSerialNumber());
478
          delay(500);
479
          if (periodicStart(SHT3XD REPEATABILITY HIGH, SHT3XD FREQUENCY 10HZ) !=
480
          SHT3XD_NO_ERROR) {
              Serial.println("[ERROR] Cannot start periodic mode");
481
482
              return false;
483
484
          else {
              Serial.println("Successfully started SHT sensor!");
485
486
              return true;
487
488
      }
489
490
      bool ClosedCube SHT31D::run sht(void) {
491
          * Run SHT sensor
492
493
          * start read count from 1
           * is average taken is false until average is taken
494
495
           * take reading from <a href="mailto:sht">sht</a> until enough values are read to take an average
496
497
          is_average_taken = false;
498
          error count = 1;
          read count = 1;
499
500
          while(read count <= MAX READ COUNT && error count <= MAX ERROR COUNT) {</pre>
501
              read_sht();
502
503
504
          return(is_average_taken);
505
     1
506
      SHT31D ClosedCube_SHT31D::read_sht(void) {
508
509
           * Read from SHT35D, and assign values to my result
           * print results
510
           * save results to buffer
511
           * calculate average if enough values have been read
512
514
          SHT31D my result = periodicFetchData();
          printResult("Periodic Mode", my_result);
515
516
          save to buffer (my result);
517
          calculate average();
518
          delay(250);
519
521
      SHT31D ClosedCube SHT31D::printResult(String text, SHT31D result) {
          * Prints current reading if no error and not exceeded max count
523
524
           * else print error message
525
          if (result.error == SHT3XD_NO_ERROR && read_count <= MAX_READ_COUNT ) {</pre>
526
              float current_t = result.t;
527
528
              float current rh = result.rh;
529
              if(current_t > 0 && current_rh > 0) {
530
531
                   //Serial.print(text);
532
                  Serial.print("SHT Reading #");
                  Serial.print(read_count);
534
                  Serial.print(": T=");
                  Serial.print(current_t);
Serial.print("C, RH=");
535
536
                  Serial.print(current rh);
538
                  Serial.println("%");
539
              }
540
          }
```

```
541
542
      SHT31D ClosedCube SHT31D::save to buffer(SHT31D result) {
543
544
           \,^\star Save current t and \underline{rh} readings to their respective buffers
545
546
547
           {}^{\star} if no error and the number of readings is less than the max
           * then save values
548
549
           \star 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;
              float current_rh = result.rh;
554
555
556
              if(current_t > 0 && current_rh > 0) {
                  t_buf[read_count - 1] = current_t;
rh buf[read_count - 1] = current_rh;
557
558
559
                   read count++;
560
                   error_count = 1;
              } else {
562
                  Serial.print("SHT Error count: ");
                   Serial.println(error_count);
564
                  error_count ++;
              }
566
          } else if (result.error != SHT3XD_NO_ERROR) {
567
              Serial.print("[ERROR] Code #");
              Serial.println(result.error);
568
              Serial.print("SHT Error count: ");
569
570
              Serial.println(error_count);
571
              error count ++;
572
573
      }
574
575
      void ClosedCube SHT31D::calculate average(void) {
576
           * Calculate average if enough values have been read
577
578
           * assign t ave to t average
579
           * assign rh ave to rh average
           * change is_average_taken to true so that while loop will exit
580
581
582
          if( read count > MAX READ COUNT ) {
583
              t average = 0.00;
              \overline{\text{rh}} average = 0.00;
584
              for(int k = 0; k < MAX_READ_COUNT; k++) {</pre>
585
586
                   t_average += t_buf[k];
587
                   rh_average += rh_buf[k];
588
              t average = t average / MAX READ COUNT;
589
590
              rh_average = rh_average / MAX_READ_COUNT;
591
592
              delay(500);
              Serial.println("----");
594
              Serial.println("SHT Sensor Average Readings");
              Serial.println("----");
595
              Serial.print("SHT T Average: ");
597
              Serial.println(t average);
598
              Serial.print("SHT RH Average: ");
              Serial.println(rh_average);
is_average_taken = true;
599
600
601
          }
602
603
604
      // getter function to get average temperature reading
      float ClosedCube_SHT31D::get_t_ave(void) {
605
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

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
 4 5
     #ifndef CLOSEDCUBE SI7051 h
     #define _CLOSEDCUBE_SI7051_h
#define MAX_READ_COUNT 5
#define MAX_ERROR_COUNT 40
10
11
      #define ADDR_MRT 0x40
      #define DEFAULT_AVERAGE 128
#include <Arduino.h>
13
14
15
      class ClosedCube_Si7051 {
      public:
16
17
           ClosedCube_Si7051();
18
19
           float readT(); // short-cut for readTemperature
20
           bool run_mrt(void);
            bool start_mrt(void);
22
            float get_MRT_ave(void);
23
24
25
      private:
           uint8_t _address;
void begin(uint8_t address);
26
27
            float readTemperature();
           float T_buf[MAX_READ_COUNT];
float T_ave;
int read_count;
28
29
30
31
            int error_count;
32
34
      #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
11
     * Team 26 commented Part 2 as they wrote Part 2
13
     * and understand how the code in Part 2 works
14
1.5
     ^{\star} Please note that the Globe Thermometer does not
16
     * measure Mean Radiant Temperature (MRT), it
     * actually measures the globe temperature.
18
     * MRT is calculate later using air temperature and
19
     ^{\star} globe temperature.
22
    #include <Wire.h>
24
    #include "MRT.h"
2.5
    ClosedCube Si7051::ClosedCube Si7051()
26
27
28
29
     void ClosedCube_Si7051::begin(uint8_t address) {
30
31
         address = address;
         Wire.begin();
34
         Wire.beginTransmission(_address);
35
         Wire.write(0xE6);
36
         Wire.write (0x0);
37
         Wire.endTransmission();
39
     1
40
41
    float ClosedCube Si7051::readT() {
42
         return readTemperature();
43
44
     float ClosedCube Si7051::readTemperature() {
45
46
         Wire.beginTransmission( address);
47
         Wire.write(0xF3);
48
         Wire.endTransmission();
49
50
         delay(15);
51
52
         Wire.requestFrom(_address, (uint8_t)2);
         delay(25);
         byte msb = Wire.read();
byte lsb = Wire.read();
54
5.5
56
57
         uint16 t val = msb << 8 | lsb;</pre>
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
    // It is properly commented
65
66
67
     bool ClosedCube Si7051::start mrt(void) {
```

```
/*
* Start MRT sensor
          ^{\star} The code will read a value of 128 or greater
          ^{\star} if the sensor is broken or disconnected
75
76
         * The start sequence returns false (sensor does not work)
         * if a value of 128 is read
78
79
         \,^{\star} If the value is less than 128, it returns true
         * (sensor works)
80
81
         * The code retrieved from the online library should be improved
          * to fix this.
83
84
         begin (ADDR MRT);
8.5
86
         delay(500);
87
         return(run mrt());
88
     - }
89
90
     bool ClosedCube_Si7051::run_mrt(void) {
91
          * 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;
98
         while(read_count <= MAX_READ_COUNT && error_count <= MAX_ERROR_COUNT) {</pre>
99
             float current_T = readTemperature();
             if(current_T >= DEFAULT_AVERAGE) {
                 Serial.println("----
103
                 Serial.print("Error reading from Globe Thermometer, Tg: ");
104
                 Serial.println(current_T);
                 Serial.println("-----");
106
                 error_count ++;
107
                 delay(1000);
108
             } else{
109
                 T_buf[read_count - 1] = readTemperature();
                 Serial.print("Globe Thermometer Reading #");
110
111
                 Serial.print(read count);
                 Serial.print(": Tg is: ");
                 Serial.println(T_buf[read_count - 1]);
114
                 read count ++;
115
                 error count = 1;
116
                 delay(250);
117
             }
118
         1
119
         if(read_count > MAX_READ_COUNT) {
             T_{ave} = 0;
             \overline{for} (int k = 0; k < MAX READ COUNT; k++) {
123
                 T_ave = T_ave + T_buf[k];
124
             T_ave = T_ave / MAX_READ_COUNT;
126
             Serial.println("-----
127
             Serial.print("Average Tg is: ");
             Serial.println(T_ave);
Serial.println("----");
128
129
             return(true);
131
         else if(error_count > MAX_ERROR COUNT) {
             T_ave = -\overline{1};
             Serial.println("-----");
134
             Serial.println("Error reading from Globe Thermometer, no average Tg calculated");
136
             Serial.println("-----");
137
             return(false);
138
         1
```