MECH 45X
Dossier 11 - Code

Team 26

April 7, 2018

The code for running for running the sensor package is:

`'_all.ino'`.

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

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

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

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

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

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

bool read_from_co2 = true;
bool read_from_pm = false;

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

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

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

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

void setup() {
    /*
     * Start Serial and Wire connections
```

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

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

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

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

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

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

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

`'calculate_MRT.cpp' and 'calculate_MRT.h'`

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

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

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

class mrt_and_ot {
    public:
        mrt_and_ot(void);

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

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

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

```

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

mrt_and_ot::mrt_and_ot(void)
{
}

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

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

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


```

The code for the Time library, which is a library for low level time and date functions.

`'Time.cpp' and 'Time.h'`

This code was not written by Team 26, instead it was written by Michael Margolis in 2009. The .h file is presented first, followed by the .cpp file.

```
/*
  This is Time.h, the .h file for the Time library
  This library is implements low level time and date functions

  ----------------------------------------------------------
  This code is found online. It was not written by team 26
  ----------------------------------------------------------

  July 3 2011 - fixed elapsedSecsThisWeek macro (thanks Vincent Valdy for this)
              - fixed  daysToTime_t macro (thanks maniacbug)
*/

#ifndef _Time_h
#ifdef __cplusplus
#define _Time_h

#include <inttypes.h>
#ifndef __AVR__
#include <sys/types.h> // for __time_t_defined, but avr libc lacks sys/types.h
#endif


#if !defined(__time_t_defined) // avoid conflict with newlib or other posix libc
typedef unsigned long time_t;
#endif


// This ugly hack allows us to define C++ overloaded functions, when included
// from within an extern "C", as newlib's sys/stat.h does.  Actually it is
// intended to include "time.h" from the C library (on ARM, but AVR does not
// have that file at all).  On Mac and Windows, the compiler will find this
// "Time.h" instead of the C library "time.h", so we may cause other weird
// and unpredictable effects by conflicting with the C library header "time.h",
// but at least this hack lets us define C++ functions as intended.  Hopefully
// nothing too terrible will result from overriding the C library header?!
extern "C++" {
typedef enum {timeNotSet, timeNeedsSync, timeSet
}  timeStatus_t ;

typedef enum {
    dowInvalid, dowSunday, dowMonday, dowTuesday, dowWednesday, dowThursday, dowFriday,
    dowSaturday
} timeDayOfWeek_t;

typedef enum {
    tmSecond, tmMinute, tmHour, tmWday, tmDay,tmMonth, tmYear, tmNbrFields
} tmByteFields;

typedef struct  {
  uint8_t Second;
  uint8_t Minute;
  uint8_t Hour;
  uint8_t Wday;   // day of week, sunday is day 1
  uint8_t Day;
  uint8_t Month;
  uint8_t Year;   // offset from 1970;
}   tmElements_t, TimeElements, *tmElementsPtr_t;

//convenience macros to convert to and from tm years
#define  tmYearToCalendar(Y) ((Y) + 1970)  // full four digit year
#define  CalendarYrToTm(Y)   ((Y) - 1970)
#define  tmYearToY2k(Y)      ((Y) - 30)    // offset is from 2000
#define  y2kYearToTm(Y)      ((Y) + 30)

typedef time_t(*getExternalTime)();
//typedef void  (*setExternalTime)(const time_t); // not used in this version


/*=====================================================================================*/
```

```
/* Useful Constants */
#define SECS_PER_MIN  ((time_t)(60UL))
#define SECS_PER_HOUR ((time_t)(3600UL))
#define SECS_PER_DAY  ((time_t)(SECS_PER_HOUR * 24UL))
#define DAYS_PER_WEEK ((time_t)(7UL))
#define SECS_PER_WEEK ((time_t)(SECS_PER_DAY * DAYS_PER_WEEK))
#define SECS_PER_YEAR ((time_t)(SECS_PER_DAY * 365UL)) // TODO: ought to handle leap years
#define SECS_YR_2000  ((time_t)(946684800UL)) // the time at the start of y2k

/* Useful Macros for getting elapsed time */
#define numberOfSeconds(_time_) ((_time_) % SECS_PER_MIN)
#define numberOfMinutes(_time_) (((_time_) / SECS_PER_MIN) % SECS_PER_MIN)
#define numberOfHours(_time_) (((_time_) % SECS_PER_DAY) / SECS_PER_HOUR)
#define dayOfWeek(_time_) ((((_time_) / SECS_PER_DAY + 4)  % DAYS_PER_WEEK)+1) // 1 = Sunday
#define elapsedDays(_time_) ((_time_) / SECS_PER_DAY)  // this is number of days since Jan 1 1970
#define elapsedSecsToday(_time_) ((_time_) % SECS_PER_DAY)   // the number of seconds since last midnight
// The following macros are used in calculating alarms and assume the clock is set to a date later than Jan 1 1971
// Always set the correct time before settting alarms
#define previousMidnight(_time_) (((_time_) / SECS_PER_DAY) * SECS_PER_DAY)  // time at the start of the given day
#define nextMidnight(_time_) (previousMidnight(_time_)  + SECS_PER_DAY)   // time at the end of the given day
#define elapsedSecsThisWeek(_time_) (elapsedSecsToday(_time_) +  ((dayOfWeek(_time_)-1) * SECS_PER_DAY))   // note that week starts on day 1
#define previousSunday(_time_) ((_time_) - elapsedSecsThisWeek(_time_))      // time at the start of the week for the given time
#define nextSunday(_time_) (previousSunday(_time_)+SECS_PER_WEEK)          // time at the end of the week for the given time


/* Useful Macros for converting elapsed time to a time_t */
#define minutesToTime_t ((M)) ( (M) * SECS_PER_MIN)
#define hoursToTime_t   ((H)) ( (H) * SECS_PER_HOUR)
#define daysToTime_t    ((D)) ( (D) * SECS_PER_DAY) // fixed on Jul 22 2011
#define weeksToTime_t   ((W)) ( (W) * SECS_PER_WEEK)

/*============================================================================*/
/*  time and date functions   */
int     hour();            // the hour now
int     hour(time_t t);    // the hour for the given time
int     hourFormat12();    // the hour now in 12 hour format
int     hourFormat12(time_t t); // the hour for the given time in 12 hour format
uint8_t isAM();            // returns true if time now is AM
uint8_t isAM(time_t t);    // returns true the given time is AM
uint8_t isPM();            // returns true if time now is PM
uint8_t isPM(time_t t);    // returns true the given time is PM
int     minute();          // the minute now
int     minute(time_t t);  // the minute for the given time
int     second();          // the second now
int     second(time_t t);  // the second for the given time
int     day();             // the day now
int     day(time_t t);     // the day for the given time
int     weekday();         // the weekday now (Sunday is day 1)
int     weekday(time_t t); // the weekday for the given time
int     month();           // the month now  (Jan is month 1)
int     month(time_t t);   // the month for the given time
int     year();            // the full four digit year: (2009, 2010 etc)
int     year(time_t t);    // the year for the given time

time_t now();              // return the current time as seconds since Jan 1 1970
void    setTime(time_t t);
void    setTime(int hr,int min,int sec,int day, int month, int yr);
void    adjustTime(long adjustment);

```

```
128  /* date strings */
129  #define dt_MAX_STRING_LEN 9 // length of longest date string (excluding terminating null)
130  char* monthStr(uint8_t month);
131  char* dayStr(uint8_t day);
132  char* monthShortStr(uint8_t month);
133  char* dayShortStr(uint8_t day);
134
135  /* time sync functions  */
136  timeStatus_t timeStatus(); // indicates if time has been set and recently synchronized
137  void    setSyncProvider( getExternalTime getTimeFunction); // identify the external
     time provider
138  void    setSyncInterval(time_t interval); // set the number of seconds between re-sync
139
140  /* low level functions to convert to and from system time                     */
141  void breakTime(time_t time, tmElements_t &tm);  // break time_t into elements
142  time_t makeTime(const tmElements_t &tm);  // convert time elements into time_t
143
144  } // extern "C++"
145  #endif // __cplusplus
146  #endif /* _Time_h */
147
```

```
/*
  This is Time.cpp, the .cpp file for the Time library

  ----------------------------------------------------------
  This code is found online. It was not written by team 26
  ----------------------------------------------------------

  time.c - low level time and date functions
  Copyright (c) Michael Margolis 2009-2014


  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public
  License along with this library; if not, write to the Free Software
  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA

  1.0  6  Jan 2010 - initial release
  1.1  12 Feb 2010 - fixed leap year calculation error
  1.2  1  Nov 2010 - fixed setTime bug (thanks to Korman for this)
  1.3  24 Mar 2012 - many edits by Paul Stoffregen: fixed timeStatus() to update
                     status, updated examples for Arduino 1.0, fixed ARM
                     compatibility issues, added TimeArduinoDue and TimeTeensy3
                     examples, add error checking and messages to RTC examples,
                     add examples to DS1307RTC library.
  1.4  5  Sep 2014 - compatibility with Arduino 1.5.7
*/

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

#include "Time.h"

static tmElements_t tm;          // a cache of time elements
static time_t cacheTime;   // the time the cache was updated
static uint32_t syncInterval = 300;  // time sync will be attempted after this many
seconds

void refreshCache(time_t t) {
  if (t != cacheTime) {
    breakTime(t, tm);
    cacheTime = t;
  }
}

int hour() { // the hour now
  return hour(now());
}

int hour(time_t t) { // the hour for the given time
  refreshCache(t);
  return tm.Hour;
}

int hourFormat12() { // the hour now in 12 hour format
  return hourFormat12(now());
}

```

```
69  int hourFormat12(time_t t) { // the hour for the given time in 12 hour format
70    refreshCache(t);
71    if( tm.Hour == 0 )
72      return 12; // 12 midnight
73    else if( tm.Hour  > 12)
74      return tm.Hour - 12 ;
75    else
76      return tm.Hour ;
77  }
78
79  uint8_t isAM() { // returns true if time now is AM
80    return !isPM(now());
81  }
82
83  uint8_t isAM(time_t t) { // returns true if given time is AM
84    return !isPM(t);
85  }
86
87  uint8_t isPM() { // returns true if PM
88    return isPM(now());
89  }
90
91  uint8_t isPM(time_t t) { // returns true if PM
92    return (hour(t) >= 12);
93  }
94
95  int minute() {
96    return minute(now());
97  }
98
99  int minute(time_t t) { // the minute for the given time
100   refreshCache(t);
101   return tm.Minute;
102 }
103
104 int second() {
105   return second(now());
106 }
107
108 int second(time_t t) {  // the second for the given time
109   refreshCache(t);
110   return tm.Second;
111 }
112
113 int day(){
114   return(day(now()));
115 }
116
117 int day(time_t t) { // the day for the given time (0-6)
118   refreshCache(t);
119   return tm.Day;
120 }
121
122 int weekday() {   // Sunday is day 1
123   return  weekday(now());
124 }
125
126 int weekday(time_t t) {
127   refreshCache(t);
128   return tm.Wday;
129 }
130
131 int month(){
132   return month(now());
133 }
134
135 int month(time_t t) {  // the month for the given time
136   refreshCache(t);
137   return tm.Month;
```

```
}

int year() {  // as in Processing, the full four digit year: (2009, 2010 etc)
  return year(now());
}

int year(time_t t) { // the year for the given time
  refreshCache(t);
  return tmYearToCalendar(tm.Year);
}

/*=====================================================================*/
/* functions to convert to and from system time */
/* These are for interfacing with time serivces and are not normally needed in a sketch
*/

// leap year calulator expects year argument as years offset from 1970
#define LEAP_YEAR(Y)     ( ((1970+(Y))>0) && !((1970+(Y))%4) && ( ((1970+(Y))%100) || !((1970+(Y))%400) ) )

static  const uint8_t monthDays[]={31,28,31,30,31,30,31,31,30,31,30,31}; // API starts months from 1, this array starts from 0

void breakTime(time_t timeInput, tmElements_t &tm){
// break the given time_t into time components
// this is a more compact version of the C library localtime function
// note that year is offset from 1970 !!!

  uint8_t year;
  uint8_t month, monthLength;
  uint32_t time;
  unsigned long days;

  time = (uint32_t)timeInput;
  tm.Second = time % 60;
  time /= 60; // now it is minutes
  tm.Minute = time % 60;
  time /= 60; // now it is hours
  tm.Hour = time % 24;
  time /= 24; // now it is days
  tm.Wday = ((time + 4) % 7) + 1;  // Sunday is day 1

  year = 0;
  days = 0;
  while((unsigned)(days += (LEAP_YEAR(year) ? 366 : 365)) <= time) {
    year++;
  }
  tm.Year = year; // year is offset from 1970

  days -= LEAP_YEAR(year) ? 366 : 365;
  time  -= days; // now it is days in this year, starting at 0

  days=0;
  month=0;
  monthLength=0;
  for (month=0; month<12; month++) {
    if (month==1) { // february
      if (LEAP_YEAR(year)) {
        monthLength=29;
      } else {
        monthLength=28;
      }
    } else {
      monthLength = monthDays[month];
    }

    if (time >= monthLength) {
      time -= monthLength;
    } else {
```

```
204           break;
205       }
206     }
207     tm.Month = month + 1;  // jan is month 1
208     tm.Day = time + 1;     // day of month
209   }
210
211   time_t makeTime(const tmElements_t &tm){
212   // assemble time elements into time_t
213   // note year argument is offset from 1970 (see macros in time.h to convert to other
      formats)
214   // previous version used full four digit year (or digits since 2000),i.e. 2009 was 2009
      or 9
215
216     int i;
217     uint32_t seconds;
218
219     // seconds from 1970 till 1 jan 00:00:00 of the given year
220     seconds= tm.Year*(SECS_PER_DAY * 365);
221     for (i = 0; i < tm.Year; i++) {
222       if (LEAP_YEAR(i)) {
223         seconds +=  SECS_PER_DAY;   // add extra days for leap years
224       }
225     }
226
227     // add days for this year, months start from 1
228     for (i = 1; i < tm.Month; i++) {
229       if ( (i == 2) && LEAP_YEAR(tm.Year)) {
230         seconds += SECS_PER_DAY * 29;
231       } else {
232         seconds += SECS_PER_DAY * monthDays[i-1];  //monthDay array starts from 0
233       }
234     }
235     seconds+= (tm.Day-1) * SECS_PER_DAY;
236     seconds+= tm.Hour * SECS_PER_HOUR;
237     seconds+= tm.Minute * SECS_PER_MIN;
238     seconds+= tm.Second;
239     return (time_t)seconds;
240   }
241   /*=====================================================*/
242   /* Low level system time functions  */
243
244   static uint32_t sysTime = 0;
245   static uint32_t prevMillis = 0;
246   static uint32_t nextSyncTime = 0;
247   static timeStatus_t Status = timeNotSet;
248
249   getExternalTime getTimePtr;  // pointer to external sync function
250   //setExternalTime setTimePtr; // not used in this version
251
252   #ifdef TIME_DRIFT_INFO   // define this to get drift data
253   time_t sysUnsyncedTime = 0; // the time sysTime unadjusted by sync
254   #endif
255
256
257   time_t now() {
258     // calculate number of seconds passed since last call to now()
259     while (millis() - prevMillis >= 1000) {
260       // millis() and prevMillis are both unsigned ints thus the subtraction will always
        be the absolute value of the difference
261       sysTime++;
262       prevMillis += 1000;
263   #ifdef TIME_DRIFT_INFO
264       sysUnsyncedTime++; // this can be compared to the synced time to measure long term
        drift
265   #endif
266     }
267     if (nextSyncTime <= sysTime) {
268       if (getTimePtr != 0) {
```

```
269         time_t t = getTimePtr();
270         if (t != 0) {
271           setTime(t);
272         } else {
273           nextSyncTime = sysTime + syncInterval;
274           Status = (Status == timeNotSet) ?  timeNotSet : timeNeedsSync;
275         }
276       }
277     }
278     return (time_t)sysTime;
279   }
280
281   void setTime(time_t t) {
282   #ifdef TIME_DRIFT_INFO
283    if(sysUnsyncedTime == 0)
284      sysUnsyncedTime = t;   // store the time of the first call to set a valid Time
285   #endif
286
287     sysTime = (uint32_t)t;
288     nextSyncTime = (uint32_t)t + syncInterval;
289     Status = timeSet;
290     prevMillis = millis();  // restart counting from now (thanks to Korman for this fix)
291   }
292
293   void setTime(int hr,int min,int sec,int dy, int mnth, int yr){
294    // year can be given as full four digit year or two digts (2010 or 10 for 2010);
295    //it is converted to years since 1970
296     if( yr > 99)
297         yr = yr - 1970;
298     else
299         yr += 30;
300     tm.Year = yr;
301     tm.Month = mnth;
302     tm.Day = dy;
303     tm.Hour = hr;
304     tm.Minute = min;
305     tm.Second = sec;
306     setTime(makeTime(tm));
307   }
308
309   void adjustTime(long adjustment) {
310     sysTime += adjustment;
311   }
312
313   // indicates if time has been set and recently synchronized
314   timeStatus_t timeStatus() {
315     now(); // required to actually update the status
316     return Status;
317   }
318
319   void setSyncProvider( getExternalTime getTimeFunction){
320     getTimePtr = getTimeFunction;
321     nextSyncTime = sysTime;
322     now(); // this will sync the clock
323   }
324
325   void setSyncInterval(time_t interval){ // set the number of seconds between re-sync
326     syncInterval = (uint32_t)interval;
327     nextSyncTime = sysTime + syncInterval;
328   }
```

The code for running the PMS7003 Particulate Matter sensor is:

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

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

```
/*
 * This is the .h file for the PMS7003 sensor
 * This code was written exclusively by MECH 45X Team 26
 */
#include <stdint.h>
#include "WProgram.h"
#include "Time.h"

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

#define START_TIME 6000
#define SAMPLING_TIME 280
#define SLEEP_TIME 912
#define MAX_READ_COUNT 5
#define MAX_FRAME_SYNC_COUNT 40
#define PMS_START_UP_TIME 120
#define MAX_FUNCTION_CALL_COUNT 3

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

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

    bool debug = false;

    float pm_avgpm2_5;
    float pm_avgpm1_75;
    float pm_avgpm0_75;
    float pm_avgpm0_4;
    float pm1_75_buf[MAX_READ_COUNT];
    float pm0_75_buf[MAX_READ_COUNT];
    float pm0_4_buf[MAX_READ_COUNT];

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

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

    //time
```

```
70        void begin_timer(void);
71        bool check_begin_reading(void);
72        time_t start_time;
73        time_t current_time;
74        time_t duration;
75
76
77        struct PMS7003data {
78            uint8_t  start_frame[2];
79            uint16_t frame_length;
80            uint16_t concPM1_0_factory;
81            uint16_t concPM2_5_factory;
82            uint16_t concPM10_0_factory;
83            uint16_t concPM1_0_ambient;
84            uint16_t concPM2_5_ambient;
85            uint16_t concPM10_0_ambient;
86            uint16_t countPM0_3um;
87            uint16_t countPM0_5um;
88            uint16_t countPM1_0um;
89            uint16_t countPM2_5um;
90            uint16_t countPM5_0um;
91            uint16_t countPM10_0um;
92            uint8_t  version;
93            uint8_t  error;
94            uint16_t checksum;
95        } packetdata;
96    };
97
```

```cpp
/*
 * This is the .cpp file for the PMS7003 sensor
 * This code was written exclusively by MECH 45X Team 26
 */
PM_7003::PM_7003() {
    current_byte = 0;
    packetdata.frame_length = MAX_FRAME_LENGTH;
    frame_length = MAX_FRAME_LENGTH;
    first_time = true;
    pm_avgpm2_5 = -1;
}

PM_7003::~PM_7003() {
}

void PM_7003::set_transistor(int ground_pin, int tx_pin) {
    /*
     * Set transistor and set pin mode for transistors
     * tx_pin turns tx transistor on and off
     * ground_pin turns power to sensor on and off (transistor goes to goround)
     */
    pm_ground_control = ground_pin;
    pm_tx_control = tx_pin;
    pinMode(pm_ground_control,OUTPUT);
    pinMode(pm_tx_control,OUTPUT);
}

void PM_7003::begin_timer(void) {
    /*
     * Turn sensor on and start timer
     * (time how long sensor has been on)
     */
    digitalWrite(pm_ground_control, HIGH);
    digitalWrite(pm_tx_control, HIGH);
    start_time = now();
    Serial.println("--------------------");
    Serial.print("PMS Start time: ");
    Serial.println(start_time);
    Serial.println("--------------------");
    pm_avgpm2_5 = -1;
    first_time = false;
}

bool PM_7003::check_begin_reading(void) {
    /*
     * Check if the sensor has been on long enough to begin reading
     * duration >= PMS_START_UP_TIME
     */
    current_time = now();
    duration = current_time - start_time;
    Serial.println("-----------------");
    Serial.print("PMS Duration: ");
    Serial.println(duration);
    Serial.println("-----------------");

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

bool PM_7003::make_sensor_read(void) {
    /*
     * Get senor to read
     * Start timer if necessary
     * Check if timer has been on long enough to rad from sensor
     * If sensor has been on long enough, start reading
     * If enough readings have been taken, turn sensor off
     */
```

```
    if(first_time) {
        function_call_count = 0;
        begin_timer();
        return(false);
    }
    else if(function_call_count < MAX_FUNCTION_CALL_COUNT) {
        if(check_begin_reading()) {
            Serial.println("------------------------");
            Serial.print("PMS Function Call Count: ");
            Serial.println(function_call_count);
            Serial.println("------------------------");
            run_PM_sensor();
            function_call_count ++;
        } else {return(false);}
    }

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

void PM_7003::calibrate_sensor(void) {
    /*
     * Start timer, if necessary
     * Wait until the sensor has been on long enough before reading
     * Once sensor has been on long enough, read forever
     */
    if(first_time) {
        function_call_count = 0;
        begin_timer();
    }

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

bool PM_7003::run_PM_sensor(void) {
    /*
     * Start serial connection
     * Initialize variables
     * drain_serial() and read_sensor() until enough values have been read
     * Take average
     * end serial connection
     */
    Serial1.begin(9600);
    read_count = 1;
    done_reading = false;
    frame_sync_count = 0;
    while(!done_reading && frame_sync_count < MAX_FRAME_SYNC_COUNT) {
        drain_serial();
        delay(750);
        read_sensor();
    }

    Serial1.end();

    if(done_reading) {
        Serial.println("--------------------------");
        Serial.print("PM 2.5 Average Reading: ");
        Serial.println(pm_avgpm2_5);
```

```
        Serial.println("--------------------------");
        return true;
    }
    else if(!done_reading && frame_sync_count >= MAX_FRAME_SYNC_COUNT){return false;}
}

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

void PM_7003::frame_sync(void) {
  /*
   * syncs frames for PM sensor
   * checks that frames are being read in correct order
   * exits when it confirms that frames are being read correctly
  */
    sync_state = false;
    frame_count = 0;
    byte_sum = 0;

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

        if(current_byte == FIRST_BYTE && frame_count == 0) {
            frame_buffer[frame_count] = current_byte;
            packetdata.start_frame[0] = current_byte;
            byte_sum = current_byte;
            frame_sync_count = 1;
            frame_count = 1;
        }
        else if(current_byte == SECOND_BYTE && frame_count == 1){
            frame_buffer[frame_count] = current_byte;
            packetdata.start_frame[1] = current_byte;
            byte_sum = byte_sum + current_byte;
            frame_count = 2;
            frame_sync_count = 1;
            sync_state = true;
        }
        else{
            frame_sync_count++;

            if(frame_sync_count >= 10) {
                Serial.print("frame count: ");
                Serial.println(frame_sync_count);
            }

            if(debug) {
                Serial.println("frame is syncing");
                Serial.print("Current character: ");
                Serial.println(current_byte, HEX);
                Serial.print("frame count: ");
                Serial.println(frame_sync_count);
            }

            delay(750);

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

```
208             }
209
210         }
211     }
212 }
213
214 void PM_7003::read_sensor(void) {
215     /*
216      * Sync the frames
217      * read bytes and fill frame_buffer
218      * use data_switch to calculate different parameters
219      * print_messages once all values have been read.
220      * done_reading = true if enough values have been read
221      */
222     frame_sync();
223
224     while(sync_state == true && Serial1.available() > 0) {
225         current_byte = Serial1.read();
226         frame_buffer[frame_count] = current_byte;
227         byte_sum = byte_sum + current_byte;
228         frame_count++;
229         uint16_t current_data = frame_buffer[frame_count-1]+(frame_buffer[frame_count-2
          ]<<8);
230         data_switch(current_data);
231
232         if (frame_count >= frame_length && read_count <= MAX_READ_COUNT) {
233             print_messages();
234             read_count++;
235             break;
236         }
237     }
238
239     if (read_count > MAX_READ_COUNT) {
240         pm_avgpm2_5 = 0;
241         pm_avgpm1_75 = 0;
242         pm_avgpm0_75 = 0;
243         pm_avgpm0_4 = 0;
244         for(int k = 0; k < MAX_READ_COUNT; k++) {pm_avgpm1_75 += pm1_75_buf[k];}
245         for(int k = 0; k < MAX_READ_COUNT; k++) {pm_avgpm0_75 += pm0_75_buf[k];}
246         for(int k = 0; k < MAX_READ_COUNT; k++) {pm_avgpm0_4 += pm0_4_buf[k];}
247         float pm_avg04_f = 3668*exp(-2.265*pow(10,-6) * (pm_avgpm0_4/MAX_READ_COUNT)) +
          25.63*exp(0.0001089*(pm_avgpm0_4/MAX_READ_COUNT));
248         float pm_avg075_f = 329.9*exp(5.122*pow(10,-5) * (pm_avgpm0_75/MAX_READ_COUNT))
          + 21.26*exp(0.0002764*(pm_avgpm0_75/MAX_READ_COUNT));
249         float pm_avg175_f = 1.941*pow(10,-12)*pow((pm_avgpm0_75/MAX_READ_COUNT),4) +-
          2.409*pow(10,-8)*pow((pm_avgpm0_75/MAX_READ_COUNT),3) + 0.0001295*pow((
          pm_avgpm0_75/MAX_READ_COUNT),2)+ -0.02592*(pm_avgpm0_75/MAX_READ_COUNT)+ 30.16;
250         float pm_avg_fvol = pm_avg04_f*4/3*3.14159265359*pow((400/2*pow(10,-9)),3)+
          pm_avg075_f*4/3*3.14159265359*pow((750/2*pow(10,-9)),3)+pm_avg175_f*4/3*
          3.14159265359*pow((1750/2*pow(10,-9)),3);
251         float pm_avg_fmass = pm_avg_fvol*1.65*pow(100,3)*10*1000*1000000;
252
253         pm_avgpm2_5 = pm_avg_fmass;
254         done_reading = true;
255     }
256 }
257
258 void PM_7003::data_switch(uint16_t current_data) {
259     /*
260      * data_switch uses current data and frame_count
261      * to assign values to parameters
262      */
263     switch (frame_count) {
264     case 4:
265         packetdata.frame_length = current_data;
266         frame_length = current_data + frame_count;
267         break;
268     case 6:
269         packetdata.concPM1_0_factory = current_data;
```

```
270             break;
271         case 8:
272             packetdata.concPM2_5_factory = current_data;
273             break;
274         case 10:
275             packetdata.concPM10_0_factory = current_data;
276             break;
277         case 12:
278             packetdata.concPM1_0_ambient = current_data;
279             break;
280         case 14:
281             packetdata.concPM2_5_ambient = current_data;
282             break;
283         case 16:
284             packetdata.concPM10_0_ambient = current_data;
285             break;
286         case 18:
287             packetdata.countPM0_3um = current_data;
288             break;
289         case 20:
290             packetdata.countPM0_5um = current_data;
291             break;
292         case 22:
293             packetdata.countPM1_0um = current_data;
294             break;
295         case 24:
296             packetdata.countPM2_5um = current_data;
297             break;
298         case 26:
299             packetdata.countPM5_0um = current_data;
300             break;
301         case 28:
302             packetdata.countPM10_0um = current_data;
303             break;
304         case 29:
305             current_data = frame_buffer[frame_count-1];
306             packetdata.version = current_data;
307           break;
308         case 30:
309             current_data = frame_buffer[frame_count-1];
310             packetdata.error = current_data;
311             break;
312         case 32:
313             packetdata.checksum = current_data;
314             byte_sum -= ((current_data>>8)+(current_data&0xFF));
315             break;
316         default:
317             break;
318         }
319     }
320
321     void PM_7003::print_messages(void){
322         /*
323          * Print messages to string and Serial screen
324          */
325         sprintf(print_buffer, ", %02x, %02x, %04x, ",
326             packetdata.start_frame[0], packetdata.start_frame[1], packetdata.frame_length);
327         sprintf(print_buffer, "%s%04d, %04d, %04d, ", print_buffer,
328             packetdata.concPM1_0_factory, packetdata.concPM2_5_factory, packetdata.
                concPM10_0_factory);
329         sprintf(print_buffer, "%s%04d, %04d, %04d, ", print_buffer,
330             packetdata.concPM1_0_ambient, packetdata.concPM2_5_ambient, packetdata.
                concPM10_0_ambient);
331         sprintf(print_buffer, "%s%04d, %04d, %04d, %04d, %04d, %04d, ", print_buffer,
332             packetdata.countPM0_3um, packetdata.countPM0_5um, packetdata.countPM1_0um,
333             packetdata.countPM2_5um, packetdata.countPM5_0um, packetdata.countPM10_0um);
334         sprintf(print_buffer, "%s%02d, %02d, ", print_buffer,
335             packetdata.version, packetdata.error);
336
```

```
337       float pm0_4_f = packetdata.countPM0_3um - packetdata.countPM0_5um;
338       float pm0_75_f = packetdata.countPM0_5um - packetdata.countPM1_0um;
339       float pm1_75_f = packetdata.countPM1_0um - packetdata.countPM2_5um;
340       pm1_75_buf[read_count-1] = pm1_75_f;
341       pm0_75_buf[read_count-1] = pm0_75_f;
342       pm0_4_buf[read_count-1] = pm0_4_f;
343
344       if(debug) {
345           Serial.println(print_buffer);
346       }
347
348       Serial.print("PM 2.5 Reading #");
349       Serial.print(read_count);
350       Serial.print(": ");
351       Serial.println(pm1_75_buf[read_count-1]);
352   }
353
354   float PM_7003::get_pm_ave(void) {
355       return pm_avgpm2_5;
356   }
357
358   void PM_7003::reset_pm_ave(void) {
359       pm_avgpm2_5 = -1.0;
360   }
361
```

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

`'MHZ19.cpp' and 'MHZ19.h'`

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

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

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


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

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

        bool debug = false;

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

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

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

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

```

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

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

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

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

MHZ19::~MHZ19() {
}

void MHZ19::set_transistor(int pin) {
    /*
     * Set pin mode of co2 transistor pin
     */
    co2_transistor_control = pin;
    pinMode(co2_transistor_control,OUTPUT);
}

void MHZ19::begin_timer(void) {
    /*
     * Function to start timer
     * Only called if timer has not been started
     * Turns CO2 sensor on and starts timer
     * Prints start time
     */
    co2_ppm_average_uncalibrated = -1;
    co2_ppm_average_calibrated = -1;
    digitalWrite(co2_transistor_control, HIGH);
    start_time = now();
    Serial.println("--------------");
    Serial.print("CO2 start time: ");
    Serial.println(start_time);
    Serial.println("--------------");
    first_time = false;
}

bool MHZ19::check_begin_reading(void) {
    /*
     * Check if sensor has been on long enough to start reading
     * Print how long the sensor has been on
     * Return true if sensor is on long enough
     * else return false
     */
    current_time = now();
    duration = current_time - start_time;
    Serial.println("-----------------");
    Serial.print("CO2 Duration: ");
    Serial.println(duration);
    Serial.println("-----------------");

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

bool MHZ19::make_sensor_read(void) {
    /*
     * Method to make CO2 sensor read
     * Sensor turns off every time enough readings have been taken
     * Sensor turns back on again to take more readings
     *
     * IF sensor off (first_time == true), call begin_timer()
     * ELSE IF not enough readings have been taken
```

```
70          *       IF sensor on long enough, take a reading
71          *       ELSE return false
72          * IF enough readings have been taken
73          *       turn sensor off, first_time = true
74          *       return true
75          * ELSE return false
76          */
77      if(first_time) {
78          function_call_count = 0;
79          begin_timer();
80          return(false);
81      }
82      else if(function_call_count < MAX_FUNCTION_CALL_COUNT) {
83          if(check_begin_reading()) {
84              Serial.println("----------------------");
85              Serial.print("Function Call Count: ");
86              Serial.println(function_call_count);
87              Serial.println("----------------------");
88              run_sensor();
89              function_call_count ++;
90          } else {return(false);}
91      }
92
93
94      if(function_call_count >= MAX_FUNCTION_CALL_COUNT) {
95          first_time = true;
96          digitalWrite(co2_transistor_control, LOW);
97          return(true);
98      } else{return(false);}
99  }
100
101 void MHZ19::calibrate_sensor(void) {
102     /*
103      * Method to make CO2 sensor read without turning off
104      * Turn sensor on and wait until it warms upper_bound
105      * Take sensor readings forever
106      */
107     if(first_time) {
108         function_call_count = 0;
109         begin_timer();
110     }
111
112     if(check_begin_reading()) {
113         Serial.println("----------------------");
114         Serial.print("Function Call Count: ");
115         Serial.println(function_call_count);
116         Serial.println("----------------------");
117         run_sensor();
118         function_call_count ++;
119     }
120 }
121
122 bool MHZ19::run_sensor(void) {
123     /*
124      * Run the MHZ19 sensor
125      * Set ppm to zero
126      * clear the frame_buffer
127      * drain the serial buffer
128      * read from the sensor
129      * print reading
130      * add the reading to the average value buffer
131      * calculate average value
132      */
133     co2_ppm = -1;
134     co2_ppm_average_uncalibrated = 0;
135     co2_ppm_average_calibrated = 0;
136     is_average_taken = false;
137     does_sensor_work = true;
138     reading_count = 1;
```

```
139
140         serial_drain();
141
142         while(is_average_taken == false && does_sensor_work == true) {
143             memset(frame_buffer, 0, 9);
144             read_sensor();
145             print_current_reading();
146             add_to_ave_buf();
147             calculate_average_reading();
148             print_average_reading();
149             co2_ppm = -1;
150         }
151         if(is_average_taken == true) {return(true);}
152         else {return(false);}
153     }
154
155     void MHZ19::print_current_reading(void) {
156         /*
157          * Prints current reading if reading is valid (i.e. co2_ppm > 0)
158          * and if the maximum number of readings haven't been exceeded
159          */
160         if(co2_ppm > 0) {
161           Serial.print("MHZ19 CO2 PPM Reading ");
162           Serial.print(reading_count);
163           Serial.print(": ");
164           Serial.println(co2_ppm);
165         }
166         else {
167           Serial.println("Error reading CO2 PPM from MHZ19");
168         }
169     }
170
171     void MHZ19::add_to_ave_buf(void) {
172         /*
173          * IF a valid value of co2 is read and the number of reading is less than the max,
174          * THEN add current value to buffer
175          */
176         if(co2_ppm > 0 && reading_count <= NUMBER_OF_VALUES) {
177             mhz19_buffer[reading_count - 1] = co2_ppm;
178             reading_count += 1;
179         }
180     }
181
182     void MHZ19::calculate_average_reading(void) {
183         /*
184          * IF the number of readings exceeds the number of values to be read,
185          * THEN calculate the average
186          */
187         if(reading_count > NUMBER_OF_VALUES) {
188             for(int k = 0; k < NUMBER_OF_VALUES; k++) {co2_ppm_average_uncalibrated +=
                mhz19_buffer[k];}
189
190             co2_ppm_average_uncalibrated = co2_ppm_average_uncalibrated / ( NUMBER_OF_VALUES
                 );
191             is_average_taken = true;
192         }
193     }
194
195     void MHZ19::apply_calibration_curve(void) {
196         /*
197          * Method to apply calibration curve
198          * calibrated_value = a0 + uncalibrated_value * a1
199          */
200         co2_ppm_average_calibrated = calib_a0 + co2_ppm_average_uncalibrated * calib_a1;
201     }
202
203     void MHZ19::print_average_reading(void) {
204         /*
205          * IF the average has been taken (co2_ppm_average > 0)
```

```
206         * THEN print the average
207         */
208        if(co2_ppm_average_uncalibrated > 0) {
209            Serial.println("-----------------------------");
210            Serial.print("CO2 PPM Average Reading (Uncalibrated): ");
211            Serial.println(co2_ppm_average_uncalibrated);
212            apply_calibration_curve();
213            Serial.print("CO2 PPM Average Reading (Calibrated): ");
214            Serial.println(co2_ppm_average_calibrated);
215            Serial.println("-----------------------------");
216        }
217    }
218
219    void MHZ19::read_sensor(void) {
220        /*
221         *  Start Serial1 connection
222         *  Send command to read from sensor to the sensor
223         *  Read from the sensor (fill_from_buffer();)
224         *  Calculate PPM for CO2
225         *  End Serial connection
226        */
227
228        Serial1.begin(9600);
229        Serial1.write(mhz19_read_command, 9);
230        delay(1000);
231        fill_frame_buffer();
232        co2_ppm = 256*frame_buffer[2] + frame_buffer[3];
233        Serial1.end();
234    }
235
236    void MHZ19::serial_drain(void) {
237      /*
238       * Drains serial buffer when sensor is turned on
239       */
240        while (Serial1.available() > 0) {
241            drain = Serial1.available();
242            Serial.print("-- Draining buffer: ");
243        }
244    }
245
246    void MHZ19::frame_sync(void) {
247        /*
248         * Sync frames so that frames are added to the frame_buffer in the correct order
249         * IF correct byte is read, THEN add to buffer and move on to next byte
250         * ELSE read byte and discard
251         * IF no bytes are available to read and the frames have not been synced, THEN send
          command to read from sensor again
252         *
253         * frame_sync_count keeps track of how many frames are added to frame_buffer
254         * frame_read_count keeps track of how many frames are read but not added to buffer
          (fails if too many frames read)
255         */
256        sync_state = false;
257        frame_sync_count = 0;
258        frame_read_count = 0;
259        byte_sum = 0;
260
261        while (!sync_state && Serial1.available() > 0 && frame_read_count <
          MAX_FRAME_READ_COUNT) {
262            current_byte = Serial1.read();
263
264            if (current_byte == MHZ19_ZEROTH_BYTE && frame_sync_count == 0) {
265                frame_buffer[frame_sync_count] = current_byte;
266                byte_sum = current_byte;
267                frame_sync_count = 1;
268            }
269            else if (current_byte == MHZ19_FIRST_BYTE && frame_sync_count == 1) {
270                frame_buffer[frame_sync_count] = current_byte;
271                byte_sum += current_byte;
```

```
272             sync_state = true;
273             frame_sync_count = 2;
274         }
275         else {
276             if(debug) {
277                 Serial.print("-- Frame syncing... ");
278                 Serial.println(current_byte, HEX);
279             }
280 
281             frame_read_count ++;
282         }
283 
284         if (!sync_state && !(Serial1.available() > 0) && frame_read_count <
            MAX_FRAME_READ_COUNT) {
285             Serial1.write(mhz19_read_command, 9);
286 
287             if(debug) {
288                 Serial.println("----------------------------------------");
289                 Serial.println("Read command has been sent to CO2 sensor");
290                 Serial.println("----------------------------------------");
291             }
292 
293             delay(500);
294         }
295     }
296 }
297 
298 void MHZ19::fill_frame_buffer(void) {
299   /*
300    * Sync frames
301    * Read byte into frame_buffer
302    */
303     frame_sync();
304 
305     while(sync_state && Serial1.available() > 0 && frame_sync_count < MAX_FRAME_LEN) {
306         current_byte = Serial1.read();
307         frame_buffer[frame_sync_count] = current_byte;
308         byte_sum += current_byte;
309         frame_sync_count++;
310     }
311 }
312 
313 // getter functions
314 int MHZ19::get_co2_ave_uncalibrated(void) {
315     return co2_ppm_average_uncalibrated;
316 }
317 
318 int MHZ19::get_co2_ave_calibrated(void) {
319     return co2_ppm_average_calibrated;
320 }
321 
322 int MHZ19::get_co2_reading(void) {
323     return co2_ppm;
324 }
325 
326 void MHZ19::reset_co2_ave(void) {
327     co2_ppm_average_uncalibrated = -1;
328     co2_ppm_average_calibrated = -1;
329 }
330 
```

The code for running the CCS821 VOC sensor is:

'ccs821.cpp' and 'ccs821.h'

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

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

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

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

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

#include <Wire.h>

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

    #define MAX_READ_COUNT 5
    #define MAX_ERROR_COUNT 5

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

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

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

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

#define CCS811_HW_ID_CODE     0x81

#define CCS811_REF_RESISTOR     100000

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

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

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

    bool begin(uint8_t addr = CCS811_ADDRESS);

    void setEnvironmentalData(uint8_t humidity, double temperature);

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

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

    void SWReset();

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

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

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

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

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

    bool checkError();

  private:
    float eCO2_buf[MAX_READ_COUNT];
    float TVOC_buf[MAX_READ_COUNT];
    float eCO2_ave;
    float TVOC_ave;
```

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

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

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

 #include "CCS821.h"

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

  _i2c_init();

  SWReset();
  delay(100);

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

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

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

  disableInterrupt();

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

  return true;
}

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

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

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

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

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

```
272
273         for(int i=0; i<read_now; i++){
274           buf[pos] = Wire.read();
275           pos++;
276         }
277       }
278     }
279
280     void Adafruit_CCS811::write(uint8_t reg, uint8_t *buf, uint8_t num)
281     {
282       Wire.beginTransmission((uint8_t)_i2caddr);
283       Wire.write((uint8_t)reg);
284       Wire.write((uint8_t *)buf, num);
285       Wire.endTransmission();
286     }
287
288     /*
289      * Part 2: code written by team 26
290      * This code was written by Team 26
291      * This code is properly commented
292      */
293
294     bool Adafruit_CCS811::start_voc(void) {
295         /*
296          * Start voc sensor using the library's begin() function
297          * If sensor is started, calibrate temperature
298          */
299         Serial.println("Trying to start VOC Sensor...");
300         if(!begin()){
301             Serial.println("Failed to start CC2821 VOC sensor! Wiring is likely incorrect.");
302             return false;
303         }
304         else {
305             Serial.println("Successfully started VOC Sensor!");
306             delay(5000);
307             return true;
308         }
309     }
310
311     bool Adafruit_CCS811::run_voc(void) {
312         /*
313          * Run the VOC sensor
314          * Take measurements until enough measurements have been taken to calculate the
           average
315          * use read_voc() to read from sensor
316          */
317         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();}
321
322         if(is_average_taken) {return true;}
323         else if(error_count >= MAX_ERROR_COUNT) {return false;}
324     }
325
326     void Adafruit_CCS811::read_voc(void) {
327         /*
328          * Read values from voc sensor
329          * IF data is read and max read count has not been exceed
330          * THEN fill_buffer and print_readings and read_count ++
331          * calculate_average_reading
332          * print_average_reading
333          */
334         if(available()){
335             float temp = calculateTemperature();
336             if(!readData() && read_count <= MAX_READ_COUNT){
337                 fill_buffer();
338                 print_readings();
339                 read_count += 1;
```

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

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

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

`'SHT35D.cpp' and 'SHT35D.h'`

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

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

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

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

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

#include <Arduino.h>

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

    SHT3XD_CMD_READ_STATUS = 0xF32D,
    SHT3XD_CMD_CLEAR_STATUS = 0x3041,

    SHT3XD_CMD_HEATER_ENABLE = 0x306D,
    SHT3XD_CMD_HEATER_DISABLE = 0x3066,

    SHT3XD_CMD_SOFT_RESET = 0x30A2,

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

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

    SHT3XD_CMD_ART = 0x2B32,

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

    SHT3XD_CMD_FETCH_DATA = 0xE000,
    SHT3XD_CMD_STOP_PERIODIC = 0x3093,

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

    SHT3XD_CMD_NO_SLEEP = 0x303E,
} SHT31D_Commands;

// List of repeatability options for SHT35D:
typedef enum {
  SHT3XD_REPEATABILITY_HIGH,
  SHT3XD_REPEATABILITY_MEDIUM,
```

```
  SHT3XD_REPEATABILITY_LOW,
} SHT31D_Repeatability;

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

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

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

  SHT3XD_CRC_ERROR = -101,
  SHT3XD_TIMEOUT_ERROR = -102,

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

  // Wire I2C translated error codes

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

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

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

class ClosedCube_SHT31D {
```

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

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

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

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

ClosedCube_SHT31D::ClosedCube_SHT31D()
{
}

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

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

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

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

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

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

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

```
}

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

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

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

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


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

    return error;
}

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

    return error;
}

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

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

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

        uint8_t checksum = calculateCrc(buf);

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

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

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

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

```
473       Serial.println("Trying to start SHT sensor...");
474       delay(500);
475       begin(ADDR_SHT); // I2C address: 0x44 or 0x45
476       Serial.print("Serial #");
477       Serial.println(readSerialNumber());
478       delay(500);
479
480       if (periodicStart(SHT3XD_REPEATABILITY_HIGH, SHT3XD_FREQUENCY_10HZ) !=
          SHT3XD_NO_ERROR) {
481           Serial.println("[ERROR] Cannot start periodic mode");
482           return false;
483       }
484       else {
485           Serial.println("Successfully started SHT sensor!");
486           return true;
487       }
488   }
489
490   bool ClosedCube_SHT31D::run_sht(void) {
491       /*
492        * Run SHT sensor
493        * start read_count from 1
494        * is_average_taken is false until average is taken
495        * take reading from sht until enough values are read to take an average
496        */
497       is_average_taken = false;
498       error_count = 1;
499       read_count = 1;
500       while(read_count <= MAX_READ_COUNT && error_count <= MAX_ERROR_COUNT) {
501           read_sht();
502       }
503
504       return(is_average_taken);
505   }
506
507   SHT31D ClosedCube_SHT31D::read_sht(void) {
508       /*
509        * Read from SHT35D, and assign values to my_result
510        * print results
511        * save results to buffer
512        * calculate average if enough values have been read
513        */
514       SHT31D my_result = periodicFetchData();
515       printResult("Periodic Mode", my_result);
516       save_to_buffer(my_result);
517       calculate_average();
518       delay(250);
519   }
520
521   SHT31D ClosedCube_SHT31D::printResult(String text, SHT31D result) {
522       /*
523        * Prints current reading if no error and not exceeded max count
524        * else print error message
525        */
526       if (result.error == SHT3XD_NO_ERROR && read_count <= MAX_READ_COUNT ) {
527           float current_t = result.t;
528           float current_rh = result.rh;
529
530           if(current_t > 0 && current_rh > 0) {
531               //Serial.print(text);
532               Serial.print("SHT Reading #");
533               Serial.print(read_count);
534               Serial.print(": T=");
535               Serial.print(current_t);
536               Serial.print("C, RH=");
537               Serial.print(current_rh);
538               Serial.println("%");
539           }
540       }
```

```
}

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

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

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

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

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

```
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
 */

#ifndef _CLOSEDCUBE_SI7051_h

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

class ClosedCube_Si7051 {
public:
    ClosedCube_Si7051();

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

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

#endif

```

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

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

ClosedCube_Si7051::ClosedCube_Si7051()
{
}

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

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

}

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

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

    delay(15);

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

    uint16_t val = msb << 8 | lsb;

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

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


bool ClosedCube_Si7051::start_mrt(void) {
```

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

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