```
//============================================================
//=====  CHS Smokey General Library Header - common code  =====
//=====       for Mega Arduino used in Smokey                 =====
//=====                    Code includes the setup() section
=====
//============================================================
//===== Global definitions ===================================
#define TRUE 1
#define FALSE 0
//===== Definitions for Bluefruit & UART functions
#define BUFSIZE                   128       // Size of the read buffer for incoming data
#define VERBOSE_MODE              true      // If set to 'true' enables debug output
#define BLUEFRUIT_UART_MODE_PIN       4     // Set to -1 if unused
#define FACTORYRESET_ENABLE       0
#define MINIMUM_FIRMWARE_VERSION    "0.6.6"
#define MODE_LED_BEHAVIOUR          "MODE"
#define BLUEFRUIT_HWSERIAL_NAME      Serial1  // Points to first Serial Port on MEGA2560

// Definitions and instantiation for DHT11 temp sensor
#define DHTPIN 2
#define DHTTYPE DHT11

//===== Pan Angle motor constants
#define Steps180 1024.0   // Adjust these and/or the myStepper.Speed() call in setup
#define Steps90 512.0     //   to set motor speeds

//===== Flame definitions
#define AngRst_pin 5      // This is where angle reset pushbutton is connected
#define AngRef_pin 6      // This is gnd ref (set to zero) for angle reset
#define AngAuto_pin 7     // This is where auto switch is connected
#define AutoRef_pin 8     // This is gnd ref (set to zero) for auto switch.
#define Flm_pin A1        // Analog input from flame detector pcb.

//=====  Library files for General Mega Arduino
#include <Arduino.h>  // Basic Arduino library
#include <Wire.h>                 // I2C library
#include <EEPROM.h>                   // EEPROM library for non-volatile storage
#include <DHT.h>      // Temp/Humidity module library
DHT dht(DHTPIN,DHTTYPE); // declare a DHT object named dht
#include <LiquidCrystal_I2C.h> // Liquid crystal display module library
LiquidCrystal_I2C lcd = LiquidCrystal_I2C(0x27,20,4); // declare lcd module named lcd
#include <Stepper.h>

//=====  Library files for Bluefruit
```

```
#include "Adafruit_BLE.h"              // BLE is BluefruitLowEnergy- which is our module
#include "Adafruit_BluefruitLE_UART.h"  // Functions to handle UART: HW or SW
#include "Adafruit_BluefruitLE_SPI.h"   // Try using the Adafruit Spi library
Adafruit_BluefruitLE_UART ble(BLUEFRUIT_HWSERIAL_NAME,
BLUEFRUIT_UART_MODE_PIN);
#include "Adafruit_PM25AQI.h"  // Basic PMSA0003 library
Adafruit_PM25AQI pmsA003 = Adafruit_PM25AQI(); // declare pms object named pmsA003
PM25_AQI_Data pmsData;                        // declare pms data as pmsData structure

//=====  Library files for DS3231 Rtc
#include <DS3231.h>
DS3231 rtcClock; // Define an object that is the real time clock
bool century = false; // Tells us we're still in 2000's era
bool h12Flag;  // 12 hour flag, as opposed to 24 hr flag
bool pmFlag;  // after noon flag for 12 hr application

//=====  Library files for Adafruit PM25AQI
#include "Adafruit_SGP30.h"  // Basic PMSA0003 library
Adafruit_SGP30 sgp30;

//===== Sgp CO2/TVOC sensor variables
int sgp30Found = FALSE; // Found flag for sensor
char sgpString[80]="No Data yet.\n";      // String for returned values of TVOC and eCO2
unsigned int TVOC_base, eCO2_base;  // Bases are 16 bits, so we use unsigned ints
byte TVOC_msb,TVOC_lsb;  // We write to EEPROM using bytes,
byte eCO2_msb,eCO2_lsb;  // so we separate msb (most sig byte) and lsb (least sig byte).
unsigned int EEeCO2_base, EETVOC_base;  // Recorded bases for fast restart (i hope)
                                        // each of these has both bytes
byte EEeCO2_msb,EEeCO2_lsb,EETVOC_msb,EETVOC_lsb; // msb and lsb same as TVOC
and eCO2
int getBaseFlag = FALSE;
int oldBaseFlag = FALSE;
int baselineValid = FALSE;

//===== Stepper Motor definitions and variables
const int IN0_1=22,IN0_2=24,IN0_3=26,IN0_4=28;  // Motor1 uses pins 22,24,26,28 for IN0 -
IN4
const int IN1_1=23,IN1_2=25,IN1_3=27,IN1_4=29;  // Motor2 uses pins 23,25,27.29 for IN0 -
IN4
const int stepsPerRev_tilt = 400;      // for tilt stepper
const int stepsPerRev_pan = 100;     // for pan stepper
Stepper tiltStepper(stepsPerRev_tilt, IN1_1,IN1_2,IN1_3,IN1_4); //1,2,3,4 for bipolar stepper
Stepper panStepper(stepsPerRev_pan, IN0_1,IN0_3,IN0_2,IN0_4);   //1,3,2,4 for unipolar
stepper
```

```
//================================================================
//  SD card datalogger library for Smokey
//
//  SPI pins for Smokey, Mega
//===== Mega 2560 SPI connections
// .  ** MISO - pin 50
// .  ** MOSI - pin 51
// .  ** SCLK - pin 52
// .  ** CS - pin 53.
//================================================================
#include <SD.h>     // This is for the SD (Secure Digital) card routines
const int chipSelect = 53;  // CS: chip select enbles the SD module.
                      //  There can be more than one SPI module
                      //  on one Arduino, they just need to use
                      //  different CS pins.  But MISO, MOSI, etc
                      //  can all be the same.
int dataCount;  // Counts data lines in each output
//=====  Global declarations ========================

//===== Integers
int i,j,k;              // Use these for loops when convenient.
const float batLowThresh =7.0;
float batteryVoltage=0.0;
int batteryWhole, batteryFrac, batteryLow;
int SettingHome = 0; // True when setting home position
char batteryString[40] ;
char SDString[100];
const int Verbose = 0;
const int PinReset = 12;
int SDenabled = TRUE;       // This starts out TRUE, but if the setup
                      //  function can't find the SD module, it
                      //  it will be set to FALSE.  This will happen
                      //  if a card/chip isnt inserted in the the
                      //  module.

// Interrupt variables
const int EEsaveSet = 12;   // How many repeats b4 we save baselines

// volatile variables for interrupt use
volatile int Flag05sec =0;         // 0.5sec Flag for smoother Flame Det motion
volatile int Flag1sec =0;          // 1 sec Flag to communicate from interrupt to loop()
volatile int Flag10sec =0;         // 10 sec Flag to communicate from interrupt to loop()
volatile int Flag1min = 0;
```

```
volatile int FlagDisplay =0;       // 5 sec Flag to communicate from interrupt to loop()
volatile int WatchDog =0;          // WatchDog timer increments & trips if loop stops
volatile int WatchDogDly = 10;     // How many seconds before watchdog resets.
volatile int Count10sec =0;        // Ten second counter/timer.
volatile int Count1min = 0;        // 1min timer flag
volatile int CountDisplay =0;      // Count seconds to enable 5 sec trip point
// normal ints for interrupt setup
int timer5_counter = 34286; // Timer counter reset value
int timer5_repeat = 0;      // Flips zero to 1 each interrupt: interrupts happen
                            //   every half second so we ignore every other one.
int EEsaveCount = 0;        // Counter for SGP baseline save.
int seconds, oldsec;        // Variables for data interval timing
int minutes, oldmin;        // Varaibles to process RTC data
const int DisplayTime = 5;  // Constant for display interval = (5) seconds
int currentDisplay = 2;     // Variable with current display type (2= TVOC/CO2)
int smoke,pollen,dust;      // Variables for calculated air quality results

//Variables for Pms particulate sensor
char pmsStandardString[80];
   //other variables are define in the library in instantiation step above.

// Variables for GetCommand
int AppShift = 0;
int Command, cStart, cBee, cButton;     //Variables for GetCommand
int ActiveCmd,cPress, cCSum, cCalcSum;  // If button pressed, return a postive number for
button pressed.
int newCmd=0, currentCmd=0;             // We will get new commands and keep current
int Active;               // Active commands keep running while button pushed
int Auto,oldAuto;         // Auto is for automatic pan/tilt and data logging
char sCmd[20] = "";       // we will receive commands from APP as strings
char SmokeyCommand[80] = "";
char  displayString1[60], displayString2[60]; // Strings for App display
int AngRst_valid = 0; // Used to track validity of Pan/Tilt home calibration


// Variables for DHT11
float dht_h, dht_t, dht_f,dht_hif, dht_hic;  // These come from module call as floats
int Hum_wh, Hum_fr, fTemp_wh, fTemp_fr;      // wh: whole and fr: fractional parts for printing

// Variables to compute average pm10 = smoke.
int pm10_avg = 0; // Computes smoke average
int pm10_count = 0;
int pm10_sum = 0;
```

```
// Variables to compute average temp
float temp_avg = 0; // Computes temperature average
float temp_count = 0;
float temp_sum = 0;

// Variables to compute average humidity
float hum_avg = 0; // Computes humidity average
float hum_count = 0;
float hum_sum = 0;

// Thresholds to turn on LEDs
const int smokeGrnlimit = 10;
const int smokeYellimit = 30;
const int smokeRedlimit = 100;

const float tempGrnlimit = 85.0;
const float tempYellimit = 90.0;
const float tempRedlimit = 95.0;

const float humGrnlimit = 40.0;
const float humYellimit = 30.0;
const float humRedlimit = 20.0;


//===== Calendar variables
int rtcYear, rtcMonth, rtcDay, rtcDoW;
char dateSimple[12];

//===== Clock variables
int rtcHour, rtcMin, rtcSec;

//===== LED pins ... may be swapped in different units
//=====    Check using serial input and see if correct.
const int PinSmokeGrnLed = 45;
const int PinSmokeYelLed = 47;
const int PinSmokeRedLed = 49;

const int PinFweaGrnLed = 44;
const int PinFweaYelLed = 46;
const int PinFweaRedLed = 48;

//===== Serial Input Parser variables and parameter
const int MAX_MESSAGE=30;
  char inString[MAX_MESSAGE + 1]; // This is a string (character array) for
```

```
                          //   for whatever we read from monitor
  char * outString;               // This is just a pointer that points to
                          //   specific words in the input.
  int inSize;                    // Size of input: how many characters
  int index = 0;                  // Points to characters in buffer
  int words[4], wordCount;        // Holds index value of each 'word' in input
  float tempserial=0.0, humserial;    // The variables we want to load with input data
  int smokeserial, dustserial;    //      ditto

// Flame detector variables
int Flm_read; // Value read directly from A1, the Flame detector analog channel
int Flm_det;  // Integerized value (0 -99 at the moment) scaled for detection
int Flm_max = 0;      // Max value observed since last angle reset
int Flm_ang_max = 0;  // Angle of max value observed since last angle reset.
int Flm_mag[20],Flm_ang[20];

// Pan motion variables
int Pan_steps=0; // Step counts for pan motor
int Pan_ang_wh, Pan_ang_fr; // Step counts converted to degrees, whole and fraction
float Pan_ang = 0.0; // Full floating point version of pan angle
int Pan_direction =1; // 1 means forward, positive angle, -1 means negative.
int panCount = 0, panAccel = 1; // pan variables for accelerated motion in manual
int AngCalibrate = 1; // logic 1 means we are in angle calibrate mode, 0 is normal

// Tilt motion variables
int Tilt_steps = 40; // Step counts for tilt motor
int oldTilt_steps = 40; // Variable to save old value in case angle needs resetting
int Tilt_ang_wh = 0, Tilt_ang_fr = 0; // Step counts converted to degrees, whole and fraction
float Tilt_ang = 0.0; // floating point version of Tilt angle
int Tilt_direction = -1; // 1 means up, -1 means down

// Lcd control and data variables
int lcdTopToggle = 0; // toggles ever 5 sec to change top line of LCD
char lcdString0[30],lcdString1[30],lcdString2[30];   // scratchpad strings to print to LCD
byte angleChar[8]  = {
  B00011,
  B00010,
  B00110,
  B00100,
  B01100,
  B01000,
  B11000,
  B11111
};
```

```
char angle = 0x01;


// Rtc (real time clock) buffers and variables
char rtcDateString[40]=""; // String to hold entire date, YY/MM/DD DOW MONTH
char rtcTimeString[40]=""; // String to hold entire time, HH:MM:SS AM/PM
char rtcDaysOfWeek[8][5] = { "","Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
char rtcMonthsOfYear[13][5] =
{"","Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"};
char AmPm[3],strHour[4],strMin[4],strSec[4];
float Temp; // Rtc chip temperature.  Should track ambient temperature.

//=============================================================
//===== Function prototypes =====
void TiltUp(int);
void TiltDown(int);
void computeTilt();
void DO_DisplayFunctions();
void MotorsOff();
void labelLCD();
int DO_command(); // For manual 'set pan/tilt angle' section of startup
void DO_1secFunctions(); // Ditto - to keep SGP and watchdog happy
void computeFlm();
void Wait4_1secFlag();
void updateFlm0Dsply();
void updateFlm1Dsply();

//===== Special functions
void (* resetFunc)(void)=0 ; // Set the resetFunc() address to zero to force reset of processor.

void error(const __FlashStringHelper*err) {  // Sample code had this function for handling errors.
  Serial.println(err);
  while (1);
}

void Halt() {   // This is a halt.  There is no way to stop
  while (TRUE); //  the Arduino, so we put it in infinite
}               //  loop that does nothing.



//===== Setup Functions: called in master setup =====
void SmokeyGenSetup() {           // Setup code is only executed once.
  noInterrupts();           // disable all interrupts
```

```
  TCCR5A = 0;
  TCCR5B = 0;
  timer5_counter = 34286; //34286;   // preload timer 65536-16MHz/256/2Hz

  TCNT5 = timer5_counter;   // preload timer
  TCCR5B |= (1 << CS52);    // 256 prescaler
  TIMSK5 |= (1 << TOIE5);   // enable timer overflow interrupt
  interrupts();             // enable all interrupts

  //===== I2C Startup
  Wire.begin();
  //===== Serial Startup
  Serial.begin(9600);           // Start serial port in case we want to print
  while(! Serial) delay(100);   // Wait until serial starts up
  Serial.println(" ");          // Print message to show we restarted.
  Serial.println("=====   RESTART   =====");
  Serial.println("Welcome to Smokey Setup for MEGA2560");
  // Changed this so you need to open unit and place the jumper to put
  //    unit in calibration mode.  In calibration mode it needs to be
  //    in clean air (CO2=400, TVOC=0) for few hours and it will adjust
  //    the air quality calibration in EE Prom.

//  UpdateEEBLine = (digitalRead(PinUpdateEE) == 1);
//  Removed the jumper control for calibration.  Using logical define.
    if(EECalibrate){
      Serial.println("EEPROM will be updated with new values.");
      Serial.println(" this re-establishes new baseline for this cycle");
    }
}

 void SmokeyLedSetup(){  // Called in master setup for Led pin definitions
  pinMode(PinSmokeGrnLed, OUTPUT);
  pinMode(PinSmokeYelLed, OUTPUT);
  pinMode(PinSmokeRedLed, OUTPUT);
  pinMode(PinFweaGrnLed, OUTPUT);
  pinMode(PinFweaYelLed, OUTPUT);
  pinMode(PinFweaRedLed, OUTPUT);

}

 void SmokeyFlmSetup(){  // Called in master setup for flame detector setup.
  pinMode(AngRst_pin, INPUT_PULLUP);
  pinMode(AutoRef_pin, OUTPUT);
```

```
  pinMode(AngAuto_pin, INPUT_PULLUP); // Auto - when low angle is swept automatically
  pinMode(AngRef_pin, OUTPUT);

  digitalWrite(AutoRef_pin, 0);  // Puts a GND reference right next to Auto pin.
  digitalWrite(AngRef_pin, 0);  // Puts a GND reference right next to Angle Reset pin
}

void SmokeyStepperSetup() { // Setup code is only executed once.
  pinMode( IN0_1, OUTPUT);  // Set motor controll pins to OUTPUTS
  pinMode(IN0_2, OUTPUT);
  pinMode(IN0_3, OUTPUT);
  pinMode(IN0_4, OUTPUT);

  pinMode( IN1_1, OUTPUT);
  pinMode(IN1_2, OUTPUT);
  pinMode(IN1_3, OUTPUT);
  pinMode(IN1_4, OUTPUT);
  tiltStepper.setSpeed(200);
  panStepper.setSpeed(25);

  MotorsOff();    // Start with the Motors OFF.
}

void SmokeyBluefruitSetup() {           // Setup code is only executed once.
  if ( !ble.begin(VERBOSE_MODE) )
  { error(F("Couldn't find Bluefruit, make sure it's in CoMmanD mode & check wiring?")); }
  Serial.println( F("OK!") );

  if ( FACTORYRESET_ENABLE )
  { /* Perform a factory reset to make sure everything is in a known state */
    Serial.println(F("Performing a factory reset: "));
    if ( ! ble.factoryReset() ){
      error(F("Couldn't factory reset")); }
  }

  /* Disable command echo from Bluefruit */
  ble.echo(false);

  Serial.println("Requesting Bluefruit info:");
  /* Print Bluefruit information */
  ble.info();

  Serial.println(F("Please use Adafruit Bluefruit LE Controller"));
  Serial.println(F("Then Enter characters to send to Bluefruit"));
```

```
  Serial.println(F("Connect > Controller > Control Pad"));

  Serial.println("");
  Serial.println("");

#ifdef changeName
  Serial.print("Change name of device to ");
  Serial.print(mySmokeyName);
  Serial.println(" for easy identification");
  strcat(SmokeyCommand,"AT+GAPDEVNAME=");
  strcat(SmokeyCommand,mySmokeyName);
  Serial.print("AT command = ");
  Serial.println(SmokeyCommand);

  // Rename module to identify individual Snoopies
  ble.sendCommandCheckOK(SmokeyCommand);
#endif


  ble.verbose(false);  // debug info is a little annoying after this point!


  if(! ble.isConnected()) Serial.print("BLE not connected yet");
  i=0;
  while (! ble.isConnected()) {
      i = i+1;
      delay(1000);
      if(i < 10) Serial.print("*");
      if(i==10) Serial.print("STILL WAITING !!!");
      WatchDog = 0; // Keep watchdog happy while waiting
  }
  Serial.println(" ");
  Serial.println("BLE is now connected");
  // Rename module to identify individual Snoopies

  Serial.println(F("******************************"));

  // LED Activity command is only supported from 0.6.6
  if ( ble.isVersionAtLeast(MINIMUM_FIRMWARE_VERSION) )
  {
    // Change Mode LED Activity
    Serial.println(F("Change LED activity to " MODE_LED_BEHAVIOUR));
    ble.sendCommandCheckOK("AT+HWModeLED=" MODE_LED_BEHAVIOUR);
  }
```

```
  // Set module to DATA mode
  Serial.println( F("Switching to DATA mode!") );
  ble.setMode(BLUEFRUIT_MODE_DATA);



  Serial.println(F("********************************"));
}

void SmokeyPmsA003Setup() {     // Setup code is only executed once.
  if(! pmsA003.begin_I2C()) {   // Try to connect over I2C
    Serial.println("Could not find PmsA003");
        Halt();
  }
  Serial.println(" ----------------- ");          // Print blank line.
  Serial.println("PmsA003 connected.");
  Serial.println(" ----------------- ");          // Print blank line.
}

void SmokeySDSetup() {
    Serial.println("-----------------------------------------");
    Serial.println("-----         SD Module Setup       -----");
  if (!SD.begin(chipSelect)) { // Check for init failure
    Serial.println("SD initialization failed. Things to check:");
    Serial.println("Is a card inserted?");
    Serial.println("Disabling SD Module Access.");
    SDenabled = FALSE;
  }
  Serial.println("SD initialization done.");  // Init worked!!
  if (SDenabled) {
    Serial.println("SD Card is enabled");
    dataCount = 0;

    File dataFile = SD.open(myFileName, FILE_WRITE); //Open the file
      Serial.println("--------------------------------"); // New data header
      Serial.print("  Start New Data Record for ");
      Serial.println(mySmokeyName);
      Serial.println(" Elapsed Time, MM:DD:YYYY, HH:MM:SS, PM1.0, PM2.5, PM10.0,
TVOC(ppb), CO2(ppm), Humidity(%), Temp(F), FlmAngle, FlmDet");

    if (dataFile) {   // If it opens, write the string to it.
      dataFile.println("--------------------------------"); // New data header
      dataFile.print("  Start New Data Record for ");
```

```
      dataFile.println(mySmokeyName);
      dataFile.println(" Elapsed Time (min), MM:DD:YYYY, HH:MM:SS, PM1.0, PM2.5, PM10.0,
TVOC(ppb), CO2(ppm), Humidity(%), Temp(F), FlmAng, FlmValue");
      dataFile.close();  // close file for every data point.
    } else {
      Serial.print("error opening "); Serial.println(myFileName); //Print error if open fails
      SDenabled = FALSE;
    }
  }
}

void SmokeyLCDSetup(){  // LCD setup function
  lcd.init();

//===== Load special character (may not have been properly initialized)
//=====   this is an "angle symbol" for magnitude & angle of a vector
  angleChar[0]  = B00011;
  angleChar[1]  = B00010;
  angleChar[2]  = B00110;
  angleChar[3]  = B00100;
  angleChar[4]  = B01100;
  angleChar[5]  = B01000;
  angleChar[6]  = B11000;
  angleChar[7]  = B11111;

  lcd.createChar(1,angleChar);
  lcd.home();

  lcd.backlight();

  labelLCD();
}
//===== Functions ================================================
void nextTiltMove(){    // Move to next Tilt position. Reverses at end of travel.
  if( Tilt_direction > 0) TiltUp(20); // Positive direction we keep moving up
  else TiltDown(20);                  // Otherwise move down.
  computeTilt();                      // Calculate tilt angle etc.
  if( Tilt_ang >= 4.9 ) Tilt_direction = -1;  // if above 5deg reverse to neg direction
  if( Tilt_ang <= -4.9 ) Tilt_direction = 1;  // if below -5deg reverse to pos direction
}

int limit(int x, int lo, int hi){  // limit input to range: lo <= x <= hi.
  if(x<lo) return lo;
  else if (x>hi) return hi;
```

```
  else return x;
}


void readPmsData(){ // Grabs whole data set from PMSA003 module: particulates
  if(! pmsA003.read(&pmsData)) {
    Serial.println("Could not read from pmsA003");
    Halt();
  }
  Serial.println(" ----------------- ");          // Print blank line.
  Serial.println("PmsA003 data is aquired");
  Serial.println(" ----------------- ");          // Print blank line.
}

void printLCD(int row, int col, char pstring[]){ // Print string to LCD
  int i;
  i=0;
  pstring[20]=0;  //No more than 20 characters
  lcd.setCursor(col,row);
  while (pstring[i] != 0) {
    lcd.print(pstring[i]);
    i++;
  }
  return;
}

void getSerialInput() {  // Reads a string from monitor and parses to values to Temp, Hum,
Smoke, Dust
  if(Serial.available() == 0){  // If there's no input just return
    return ;                    //  - quick return if no input for polling
  } else {                      // Come here if there is input
    inSize = Serial.readBytes(inString, MAX_MESSAGE); // Read all characters
                                            //  and get number of characters
    inString[inSize] = 0;                           // Add the 0 to end the string
                                            // Strings in C end with a 0 value
                                            // These are called 'null terminated strings'
    delay(5); // Allow time to get next character     // We need to allow some time to get next
character
  }
    if(inSize > 0) {              // Come here if the message has some content
      words[0] = 0;               // First word pointer is at start of buffer
      wordCount = 0;                // We start with no words
      index = 0;                  //  and point index to start of buffer
```

```
     while( (index<inSize) && (wordCount<4) ){     // Make a loop that stopsat end of buffer or
words = 4.
       if(inString[index]==' ') {               // Check if the character is a blank (separates words)
         inString[index]=0;                     // Make the blanks zeros, so it terminates that word
         wordCount += 1;                         // Each blank means we found another word
         words[wordCount] = index+1;             // We want to point to next word
       }
     index += 1;                              // Increment the index to scan through buffer
   }
   tempserial = atof(&inString[words[0]]);        // atof(),atoi() convert ascii to floats or ints
   humserial = atof(&inString[words[1]]);         // words[#] points to a word in the buffer
   smokeserial = atoi(&inString[words[2]]);       // inString[words[#]] is the begining of the word
   dustserial = atoi(&inString[words[3]]);        // & gets the address - makes it look like a string
name
   Serial.print("Temp ="); Serial.println(tempserial);   // Print out values for checking
   Serial.print("Hum  ="); Serial.println(humserial);
   Serial.print("Smoke="); Serial.println(smokeserial);
   Serial.print("Dust ="); Serial.println(dustserial);

  }

}


void computeTilt(){ // calculate tilt angle from steps, make whole and fractional parts
  Tilt_ang = ((float) Tilt_steps)*0.125; // Need to measure degrees per step and fix this
  Tilt_ang_wh = int(Tilt_ang);
  Tilt_ang_fr = abs(int(10.0*(Tilt_ang - float(Tilt_ang_wh))));
}

void computePan(){ // calculate pan angle from steps, make whole and fractional parts
  Pan_ang = (((float)Pan_steps)/Steps180)*180.0;
  Pan_ang_wh = int(Pan_ang);
  Pan_ang_fr = abs(int(10.0*(Pan_ang - float(Pan_ang_wh))));

}

void clearLeds(){  // Turn off all leds before we decode which to turn on.
        digitalWrite(PinSmokeGrnLed, LOW);
        digitalWrite(PinSmokeYelLed, LOW);
        digitalWrite(PinSmokeRedLed, LOW);
        digitalWrite(PinFweaGrnLed, LOW);
        digitalWrite(PinFweaYelLed, LOW);
        digitalWrite(PinFweaRedLed, LOW);
```

```
}

void writeLeds(){ // update Led display. Use serial inputs if defined (for test)
	clearLeds();
	if(tempserial <= 0.0) {
	  if(pm10_avg > smokeRedlimit) digitalWrite(PinSmokeRedLed, HIGH);
	    else if(pm10_avg > smokeYellimit) digitalWrite(PinSmokeYelLed, HIGH);
	    else if(pm10_avg > smokeGrnlimit) digitalWrite(PinSmokeGrnLed, HIGH);

	  if( (temp_avg >= tempRedlimit) && (hum_avg <= humRedlimit) )
digitalWrite(PinFweaRedLed, HIGH);
	    else if( (temp_avg >= tempYellimit) && (hum_avg <= humYellimit) )
digitalWrite(PinFweaYelLed, HIGH);
	    else if( (temp_avg >= tempGrnlimit) && (hum_avg <= humGrnlimit) )
digitalWrite(PinFweaGrnLed, HIGH);
	} else {
	  if(smokeserial > smokeRedlimit) digitalWrite(PinSmokeRedLed, HIGH);
	    else if(smokeserial > smokeYellimit) digitalWrite(PinSmokeYelLed, HIGH);
	    else if(smokeserial > smokeGrnlimit) digitalWrite(PinSmokeGrnLed, HIGH);

	  if( (tempserial >= tempRedlimit) && (humserial <= humRedlimit) )
digitalWrite(PinFweaRedLed, HIGH);
	    else if( (tempserial >= tempYellimit) && (humserial <= humYellimit) )
digitalWrite(PinFweaYelLed, HIGH);
	    else if( (tempserial >= tempGrnlimit) && (humserial <= humGrnlimit) )
digitalWrite(PinFweaGrnLed, HIGH);
	}
}

void getDhtData(){ // read temp/humidity from DHT module
	dht_h = 0.0; dht_t = 0.0; dht_f = 0.0;
	dht_h = dht.readHumidity();
	Hum_wh = (int) dht_h;
	Hum_fr = (int) ((dht_h - (float) Hum_wh )*100.0);
	// Read temperature as Celsius (the default)
	dht_t = dht.readTemperature();
	// Read temperature as Fahrenheit (isFahrenheit = true)
	dht_f = (dht_t *1.8) + 32.0;
	fTemp_wh = (int) dht_f;
	fTemp_fr = (int) ((dht_f - (float) fTemp_wh )*100.0);
}

void printDhtData(){  // write dht data to serial output
  Serial.print(dht_h);
```

```
  Serial.print("%  ");
  Serial.print(dht_t);
  Serial.print("C  ");
  Serial.print(dht_f);
  Serial.println("F  ");
}

void MotorsOff(){           // Turn OFF all coils when we
  digitalWrite(IN0_1, LOW); // .  aren't moving.
  digitalWrite(IN0_2, LOW);
  digitalWrite(IN0_3, LOW);
  digitalWrite(IN0_4, LOW);

  digitalWrite(IN1_1, LOW);
  digitalWrite(IN1_2, LOW);
  digitalWrite(IN1_3, LOW);
  digitalWrite(IN1_4, LOW);
}

void Pan(int npan){          // Pan right on motor 0.
  panCount += 1;
  panAccel = npan*limit((panCount/10) + 1,1,10);
  panStepper.step( panAccel );
  Pan_steps += panAccel;
  computePan();   // Get data to update display
  computeFlm();   // Read and compute flame parameters
  if(SettingHome == 0){
  sprintf(lcdString0, "Pan:%2d%c%-3d Pk:%2d%c%-3d",
Flm_det,angle,Pan_ang_wh,Flm_max,angle,Flm_ang_max);
  printLCD(3,0,lcdString0);
  }
}

void PanAuto(int npan){          // Pan right on motor 0.
  panStepper.step( npan );
  Pan_steps += npan;
  computePan();   // Get data to update display
  computeFlm();   // Read and compute flame parameters
  sprintf(lcdString0, "Pan:%2d%c%-3d Pk:%2d%c%-3d",
Flm_det,angle,Pan_ang_wh,Flm_max,angle,Flm_ang_max);
  printLCD(3,0,lcdString0);

}
void TiltUp(int tsteps){          // Tilt up on motor 1.
```

```
  oldTilt_steps = Tilt_steps; // Save current postion in case of reset
  Tilt_steps += tsteps;
  computeTilt();   // Get data to update display
  if( (Tilt_ang > 5.0) || (Tilt_ang< -5.0)){
    Tilt_steps = oldTilt_steps;
    computeTilt();
    //return;
  }
  tiltStepper.step(10*tsteps);
  if(SettingHome == 0){
    sprintf(lcdString0, "Tilt:%3d.%1d", Tilt_ang_wh, Tilt_ang_fr);
    printLCD(2,0,lcdString0);
    computeFlm();   // Read and compute flame parameters
    sprintf(lcdString0, "Pan:%2d%c%-3d Pk:%2d%c%-3d",
Flm_det,angle,Pan_ang_wh,Flm_max,angle,Flm_ang_max);
    printLCD(3,0,lcdString0);

  }
}
void TiltDown(int tsteps){         // Tilt down on motor 1.
  oldTilt_steps = Tilt_steps; // Save in case overrange and we have to reset
  Tilt_steps -= tsteps;
  computeTilt();   // Get data to update display
  if( (Tilt_ang > 5.0) || (Tilt_ang< -5.0)){
    Tilt_steps = oldTilt_steps;
    computeTilt();
    //return;
  }
  tiltStepper.step(-10*tsteps);
  if(SettingHome == 0){
    sprintf(lcdString0, "Tilt:%3d.%1d", Tilt_ang_wh, Tilt_ang_fr);
    printLCD(2,0,lcdString0);
    computeFlm();   // Read and compute flame parameters
    sprintf(lcdString0, "Pan:%2d%c%-3d Pk:%2d%c%-3d",
Flm_det,angle,Pan_ang_wh,Flm_max,angle,Flm_ang_max);
    printLCD(3,0,lcdString0);
  }
}

//=====   Command processing: turn the command codes into motion commands.
int GetCommand(){   // Check to see if we've gotten command from APP.  If not, return a zero.
  Command = 0;      // If button released, return a -1 to end command.
    cStart = ' ';
    cBee = ' ';
```

```
    cButton = ' ';
    cPress = '0';
    cCSum = ' ';
// We need to add TimeOut code so that we don't get stuck if there's a glitch.
    if ( ble.available() ) // Execute this code as long as there's input from the App
     {                     // Read 5 characters from APP
       Serial.println("Reading Cmd Start");
     delay(10); cStart = ble.read();
     delay(10); cBee = ble.read();          // 2nd character is "B" - ignore
     delay(10); cButton = ble.read();       // 3rd character is button number
     delay(10); cPress = ble.read();        // 4th character is '1'= pushed, '0' = released
     delay(10); cCSum = ble.read();         // 5th character is checksum - ignore

     sCmd[0]=(char)cStart;          // Save button number in string
     sCmd[1]=(char)cBee;          // Save button number in string
     sCmd[2]=(char)cButton;          // Save button number in string
     sCmd[3]=(char)cPress;          // Save button number in string
     sCmd[4]=(char)cCSum;           // Save press or release in string
     Serial.print("Raw Cmd="); Serial.println(sCmd);
     Command = sCmd[2] - '0';           // Create a number for the button
     if((Command < 1) || (Command > 8)) {
       Command = -1;
       panCount = 0;
     }
     if(sCmd[3]=='0') {
       Command = -1;  // Make it -1 if released, else positive
       Serial.println("Released");
       panCount = 0;
       AppShift = FALSE;
     }
     if(sCmd[3]=='1') {
       Serial.println("Pressed");
     }

     cCalcSum=((cStart+cBee+cButton+cPress)&255)^255;
     if(cCalcSum != cCSum) {
       Command = -1; // if theres error just act like unpress
       AppShift = FALSE;
     }
     Serial.print(sCmd);             // Print string for debug
     Serial.print(" =>");            // Print "=>" to show 'maps to'
     Serial.print(Command,DEC);    // Print the decimal value of the command.
     Serial.print("  CSum calc:");
     Serial.print(cCalcSum, DEC);
```

```
      Serial.print("  CSum:");
      Serial.println(cCSum,DEC);
    }
// Loop back if more characters available.
  return Command;
}

void updateDisplay() {
          ble.write( displayString1); // displayString1 and 2 are written in specifice
          ble.write( displayString2); //   Time/Date/AirQual/etc update functions.
//  Serial.println("Update Display");  // This is for debugging6
}

void clearDisplay(){
          ble.write("\n\n\n");
}



//===== Setup code: runs once on power up only =====




void SmokeyRtcSetup() {
          // Start the I2C interface
//        Wire.begin();

          // Start the serial interface
//        Serial.begin(9600);

  h12Flag = TRUE;
}


int readMonth()         {rtcMonth = rtcClock.getMonth(century); return rtcMonth;}
int readDay()           {rtcDay = rtcClock.getDate(); return rtcDay;}
int readYear()  {rtcYear = (2000 + rtcClock.getYear()); return rtcYear;}

int readDoW()           {rtcDoW = (rtcClock.getDoW()); return rtcDoW;}

void getDateString(){
```

```
        readYear();  // Load up the Data variables with latest values
        readMonth();
        readDay();
        readDoW();

        rtcDateString[0] = 0;   //      Reset WholeData string to empty string.
                                                                                //
        C strings are char arrays with last char = zero.
                                                                                //
        If we make first char=0, its an empty string.

        sprintf(rtcDateString, "%s %s %2u, %4u
\n",rtcDaysOfWeek[rtcDoW],rtcMonthsOfYear[rtcMonth],rtcDay,rtcYear );
}

int readHour() {
        rtcHour = rtcClock.getHour(h12Flag, pmFlag);
  h12Flag = TRUE;
        pmFlag = FALSE;
  if(rtcHour > 12) {
    rtcHour -= 12;
    pmFlag = TRUE;
  }
        if(rtcHour == 0) rtcHour = 12;
        return rtcHour;
}
int readMin() {rtcMin = rtcClock.getMinute(); return rtcMin;}
int readSec() {rtcSec = rtcClock.getSecond(); return rtcSec;}

void getTimeString(){
        readHour();
        if(rtcHour<10) {        strHour[0]='0'; strHour[1]='0'+rtcHour; strHour[2]=0;}
        else                                    {       sprintf(strHour,"%2u",rtcHour);}

        readMin();
        if(rtcMin<10)   {       strMin[0]='0'; strMin[1]='0'+rtcMin; strHour[2]=0;}
        else                                    {       sprintf(strMin,"%2u",rtcMin);}

        readSec();
        if(rtcSec<10)   {       strSec[0]='0'; strSec[1]='0'+rtcSec; strSec[2]=0;}
        else                                    {       sprintf(strSec,"%2u",rtcSec);}

        if(pmFlag)  strcpy(AmPm,"PM");
        else                    strcpy(AmPm,"AM");
```

```
        sprintf(rtcTimeString, "%2s:%2s:%2s  %s \n",strHour,strMin,strSec,AmPm );
}



//===== Setup code: runs once on power up only =====

void getPmsA003Data(){
if (! pmsA003.read(&pmsData)) {
  Serial.println("Could not read from AQI");
  delay(500);  // try again in a bit!
  return;
}
if(Verbose) Serial.println("Got data from PmsA003");
}

void getPmsStandardString(){
        getPmsA003Data();
        sprintf(pmsStandardString, "Sm: %4d,%4d | Dst: %4d \n",
                        pmsData.pm10_standard,pm10_avg,pmsData.pm100_standard -
pmsData.pm10_standard);
}




/* return absolute humidity [mg/m^3] with approximation formula
* @param temperature [°C]
* @param humidity [%RH]
*/
uint32_t getAbsoluteHumidity(float temperature, float humidity) {
    // approximation formula from Sensirion SGP30 Driver Integration chapter 3.15
    const float absoluteHumidity = 216.7f * ((humidity / 100.0f) * 6.112f * exp((17.62f *
temperature) / (243.12f + temperature)) / (273.15f + temperature)); // [g/m^3]
    const uint32_t absoluteHumidityScaled = static_cast<uint32_t>(1000.0f * absoluteHumidity);
// [mg/m^3]
    return absoluteHumidityScaled;
}

void putSgpBLines(){
        if(Verbose){
        Serial.println(" ");
```

```
        Serial.println("Writing SGP Baselines to SGP---------------- ");
        Serial.print("eCO2: "); Serial.print(eCO2_base, HEX);
        Serial.print("  TVOC: "); Serial.println(TVOC_base, HEX);
        }
//      if (! sgp30.setIAQBaseline( TVOC_base, eCO2_base)) { // I think this function has args
backwards

                if (! sgp30.setIAQBaseline(eCO2_base, TVOC_base)) {
         Serial.println("Failed to send baseline readings");
        return;
 }
}

void getSgpBLines(){
        if (! sgp30.IAQmeasure()) {
   Serial.println("SGP30 Measurement failed");
   return "SGP30 Measurement failed";
        }
  if (! sgp30.getIAQBaseline(&eCO2_base, &TVOC_base)) {
    Serial.println("Failed to get baseline readings");
                baselineValid = FALSE;
    return;
  }
        if(Verbose) {
        Serial.println(" ");
        Serial.print("Reading baseline from SGP30: eCO2: 0x"); Serial.print(eCO2_base, HEX);
  Serial.print(" & TVOC: 0x"); Serial.println(TVOC_base, HEX);
        }
        baselineValid = TRUE;
        TVOC_lsb = (byte) (TVOC_base & 0xff);
        TVOC_msb = (byte) ((TVOC_base >> 8) & 0xff);
        eCO2_lsb = (byte) (eCO2_base & 0xff);
        eCO2_msb = (byte) ((eCO2_base >> 8) & 0xff);

        return;
}


void printEEBLines(){
        Serial.println(" ");
  Serial.print("EEPROM eCO2 baseline =");              // Print it out for debug purposes
  Serial.print(EEeCO2_base,HEX); Serial.print(" bytes msb,lsb => ");
Serial.print(EEeCO2_msb,HEX);       Serial.println(EEeCO2_lsb,HEX);
  Serial.print("EEPROM TVOC baseline =");
```

```
  Serial.print(EETVOC_base,HEX);  Serial.print(" bytes msb,lsb => ");
Serial.print(EETVOC_msb,HEX);       Serial.println(EETVOC_lsb,HEX);
}

void printSgpBLines(){
        Serial.println(" ");
  Serial.print("SGP eCO2 baseline =");              // Print it out for debug purposes
  Serial.print(eCO2_base,HEX); Serial.print(" bytes msb,lsb => "); Serial.print(eCO2_msb,HEX);
                if(eCO2_lsb < 10) Serial.print("0");
                Serial.println(eCO2_lsb,HEX);
  Serial.print("SGP TVOC baseline =");
  Serial.print(TVOC_base,HEX);  Serial.print(" bytes msb,lsb => ");
Serial.print(TVOC_msb,HEX);
                if(TVOC_lsb < 10) Serial.print("0");
                Serial.println(TVOC_lsb,HEX);
}



void getEEBLines(){
        EEeCO2_msb = EEPROM.read(0);
        EEeCO2_lsb = EEPROM.read(1);
        EETVOC_msb = EEPROM.read(2);
        EETVOC_lsb = EEPROM.read(3);

        EEeCO2_base = ((int)EEeCO2_msb << 8) | ((int)EEeCO2_lsb & 0xff);
        EETVOC_base = ((int)EETVOC_msb << 8) | ((int)EETVOC_lsb & 0xff);

        return;
}

void putEEBLines(){
        EEeCO2_msb = EEeCO2_base >> 8; EEeCO2_lsb = EEeCO2_base & 0xff;
        EETVOC_msb = EETVOC_base >> 8; EETVOC_lsb = EETVOC_base & 0xff;
        Serial.println(" "); Serial.println("Sending baselines to EEPROM----------");
        if(Verbose){
        Serial.print("EEeCO2_msb:"); Serial.print(EEeCO2_msb,HEX);
        Serial.print("  EEeCO2_lsb:"); Serial.print(EEeCO2_lsb,HEX);
        Serial.print("  EETVOC_msb:"); Serial.print(EETVOC_msb,HEX);
        Serial.print("  EETVOC_lsb:"); Serial.println(EETVOC_lsb,HEX);
        }

        EEPROM.write(0,EEeCO2_msb);
        EEPROM.write(1,EEeCO2_lsb);
```

```
        EEPROM.write(2,EETVOC_msb);
        EEPROM.write(3,EETVOC_lsb);
        return;
}

void SmokeySgp30Setup() {
        Serial.println("----------------------------");
  Serial.println("SGP30 test");

        sgp30Found = sgp30.begin();
  if (! sgp30Found){
    Serial.println("Sensor not found :(");
  }
  if(sgp30Found){
                Serial.print("Found SGP30 serial #");
        Serial.print(sgp30.serialnumber[0], HEX);
        Serial.print(sgp30.serialnumber[1], HEX);
        Serial.println(sgp30.serialnumber[2], HEX);

                if(Verbose) Serial.println("Get baseline from SGP30 ----------------");
                getSgpBLines();


                if(Verbose) Serial.println("Get baseline from EEPROM ----------------");
                getEEBLines();

                if(EECalibrate){
                        Serial.println("Loading EEBaseline into SGP30");
                        sgp30.setIAQBaseline(EEeCO2_base, EETVOC_base);
                }
        }
}

void readSgpData() {
        if (! sgp30.IAQmeasure()) {
    Serial.println("SGP30 Measurement failed");
        }
}


void getSgpString(){
//      readSgpData();  // We need to read this every sec for baseline update
/*      if(sgp30.TVOC==0 && sgp30.eCO2==400)
                sprintf(sgpString, "SGP: sgp30 No Data Yet \n");
```

```
          else
*/                    sprintf(sgpString, "     TVOC:%3dppb eCO2:%3dppm\n", sgp30.TVOC,
sgp30.eCO2);
}

//===== print welcome statement to LCD
void promptAngLCD(){ // Prints instructions to calibrate Pan/Tilt angles
// Top Line - Date time
  printLCD(0,0," Calibrate Position ");
  printLCD(1,0,"1)Pan to 0 degrees  ");
// Particulates
  printLCD(2,0,"2)Tilt to full up   ");
// Flame detector
  printLCD(3,0,"3)Press Angle Reset ");
  AngCalibrate = TRUE;

}

void labelLCD(){  // Opening screen - may not be used in this version
  getDateString();
  getTimeString();
// Top Line - Date time
  sprintf(lcdString1,"%2d/%2d/%4d %2d:%02d %s ",
rtcMonth,rtcDay,rtcYear,rtcHour,rtcMin,AmPm);
  printLCD(0,0,lcdString1);
  printLCD(1,0,"Welcome to Smokey2.0");
// Particulates
  printLCD(2,0," Detects fire in air");
// Flame detector
  printLCD(3,0," & looks for flames!");
}

// update data on LCD
void updateLCD(){
    smoke = pmsData.pm10_standard; // calculate smoke: same as 1.0u particulate
//==== 1st Line - Date/Time or Smokey Name
  if(lcdTopToggle == 0){  // top line toggles every other display
    getDateString();      //   between Date/Time and Smokey name
    getTimeString();
    sprintf(lcdString1,"%2d/%2d/%4d %2d:%02d %s ",
rtcMonth,rtcDay,rtcYear,rtcHour,rtcMin,AmPm);
    printLCD(0,0,lcdString1);
    lcdTopToggle = 1;
   } else {
```

```
    printLCD(0,0,"                    ");
    sprintf(lcdString1,"%s",mySmokeyName);
    printLCD(0,0,lcdString1);
    lcdTopToggle = 0;
   }
//==== 2nd Line - Tp means Temperature, Hm means Humidity.
  printLCD(1,0,"                    ");
  sprintf(lcdString1, "Tp:%3d Hm:%2d CO2:%3d", fTemp_wh, Hum_wh, limit(sgp30.eCO2, 0,
999));
  printLCD(1,0,lcdString1);

//==== 3rd line - Smoke/Tilt
  printLCD(2,0,"                    ");
  computeTilt();
  sprintf(lcdString1, "Tilt:%3d.%01d Smoke:%3d", Tilt_ang_wh, Tilt_ang_fr, limit(smoke,0,999));
  printLCD(2,0,lcdString1);

//==== 4th line - Flame Detector - Angle/FlameLevel/Alarm
  printLCD(3,0,"                    ");
  computePan();
  sprintf(lcdString0, "Pan:%2d%c%-3d Pk:%2d%c%-3d",
Flm_det,angle,Pan_ang_wh,Flm_max,angle,Flm_ang_max);
  printLCD(3,0,lcdString0);
}

void updateTimeDateDsply(){
    sprintf(displayString1, "[DATE] - %s", rtcDateString);
    sprintf(displayString2, "[TIME] - %s", rtcTimeString);
    clearDisplay();
//    updateDisplay();
}

void updateFlm0Dsply(){
    sprintf(displayString1, "<%2d/%-3d> %2d/%-3d %2d/%-3d %2d/%-3d::4\n",

Flm_mag[0],Flm_ang[0],Flm_mag[1],Flm_ang[1],Flm_mag[2],Flm_ang[2],Flm_mag[3],Flm_ang[
3]);
    sprintf(displayString2, " %2d/%-3d %2d/%-3d %2d/%-3d %2d/%-3d::8\n",

Flm_mag[4],Flm_ang[4],Flm_mag[5],Flm_ang[5],Flm_mag[6],Flm_ang[6],Flm_mag[7],Flm_ang[
7]);
   clearDisplay();
//    updateDisplay();
}
```

```
void updateFlm1Dsply(){
   sprintf(displayString1, " %2d/%-3d %2d/%-3d %2d/%-3d %2d/%-3d::12\n",

Flm_mag[8],Flm_ang[8],Flm_mag[9],Flm_ang[9],Flm_mag[10],Flm_ang[10],Flm_mag[11],Flm_a
ng[11]);
   sprintf(displayString2, " %2d/%-3d %2d/%-3d %2d/%-3d %2d/%-3d::16\n",

Flm_mag[12],Flm_ang[12],Flm_mag[13],Flm_ang[13],Flm_mag[14],Flm_ang[14],Flm_mag[15],F
lm_ang[15]);
  clearDisplay();
//    updateDisplay();
}


void updateNameDsply(){
 if(SDenabled) sprintf(displayString1,"%s >>> %s\n",mySmokeyName,myFileName); // print
name on line 1
 else         sprintf(displayString1,"%s >>> NO FILE <<<\n",mySmokeyName); // print name on
line 1

 batteryVoltage = (( analogRead(A0)) / 1023.0)*10.0; // battery voltage on line 2

        if(batteryVoltage < batLowThresh) batteryLow = TRUE;
        else
                        batteryLow = FALSE;

 batteryWhole = batteryVoltage;       // bat voltage is separated into whole and fraction
 batteryFrac = (int) ((batteryVoltage - (float) batteryWhole)*100.0);
 getSgpBLines();                    // we also add the current baseline values

 if(batteryLow){
                sprintf(batteryString,"Battery=%2d.%-2d  BATTERY
LOW!!!\n",batteryWhole,batteryFrac);
 } else {
                sprintf(batteryString,"Battery=%2d.%-2d
Base=%X,%X\n",batteryWhole,batteryFrac,eCO2_base,TVOC_base);
 }
 strcpy(displayString2, batteryString);
        clearDisplay();
//  updateDisplay();
}

void updateAirQualDsply(){
```

```
  getSgpString();
  sprintf(displayString1, "Air Qual:  Hum: %d.%d Temp:%d.%d\n", Hum_wh,Hum_fr,
fTemp_wh,fTemp_fr);
  strcpy(displayString2, sgpString);  // latest data string from SGP30 sensor
          clearDisplay();
//  updateDisplay();
}

void updatePartDsply(){
  getPmsA003Data();
  strcpy(displayString1, "Particulates: Smoke & Dust\n");
  strcpy(displayString2, pmsStandardString);  // latest data string from SGP30 sensor
          clearDisplay();
//  updateDisplay();
}

void writeSDdata() {
  File dataFile = SD.open(myFileName, FILE_WRITE); //Open the file
  if (dataFile) {   // If it opens, write the string to it.
    dataCount += 1; // Keep track of data records.
    sprintf(SDString, " %5d, %02d/%02d/%4d, %2d:%2d:%2d %s, %4d, %4d, %4d, %5d, %5d,
%2d.%02d, %2d.%02d, %4d, %2d",
        dataCount, rtcMonth, rtcDay, rtcYear, rtcHour, rtcMin, rtcSec, AmPm,
        pmsData.pm10_standard, pmsData.pm25_standard, pmsData.pm100_standard,
        sgp30.TVOC, sgp30.eCO2, Hum_wh ,Hum_fr,fTemp_wh,fTemp_fr,Flm_max,Flm_ang_max
);
    for(i=15; i>0; i--){
      Flm_mag[i] = Flm_mag[i-1];
      Flm_ang[i] = Flm_ang[i-1];
    }
    Flm_mag[0] = Flm_max;       // Save peak Flm magnitude
    Flm_ang[0] = Flm_ang_max;   // Save peak Flm angle

    Flm_max = 0;
    Flm_ang_max = 0;

    dataFile.println(SDString);
    dataFile.close();
    Serial.println(SDString);
  }
  else {
    Serial.print("error opening "); Serial.println(myFileName);
    SDenabled = FALSE;
  } //Print error if open fails
```

```
}

void forceManual(){  // Prompt for initial pan/tilt angle setup and go into manual
                     //     mode until angle reset button is pushed.
     if(digitalRead(AngAuto_pin == 0) ) {  // If we are in Auto mode, print Error msg
       printLCD(1,0,"  Switch to Manual  ");
       printLCD(2,0,"  for startup ..... ");
       printLCD(3,0,"  Then cal pan/tilt ");
     }
     while( digitalRead(AngAuto_pin) == 0 )  { // Loop until switch is flipped to manual
       WatchDog = 0;  // Keep watchdog happy
       printLCD(0,0," *ERROR ERROR ERROR*");
       delay(500);
       printLCD(0,0,"!Switch to MAN-ual! ");
       delay(500);
     }
}

void calPanTilt(){    // Prompt user to put Flame det in home position
                    // HOME: Pan is centered, Tilt is fully forward
  promptAngLCD();           //  Prompt for angle calibration
  while ( digitalRead(AngRst_pin)!= 0){ // We stop manual cal when reset is pressed
    do {                  // We are effectively in manual mode
      DO_1secFunctions();   // 1 sec functions - CO2 and TVOC module & watchdog reset
      Active = DO_command();// Motion commands are active - they persist
                          //   while button is pressed, so we keep looping
                          //   here until we get Active= FALSE.
      delay(10);            // Delay to let Bluetooth get a command
    } while ( Active );     // "do while" loops are tested at end, always
  AngRst_valid = TRUE;
  }
// We're positioned at home, now reset angles
  Tilt_steps = 40;                  //   reset pan & tilt angles
  Tilt_ang_wh = 0;
  Tilt_ang_fr = 0;
  Tilt_ang = 0.0;

  Pan_steps = 0; // reset Flame angle position (counter in motor control)
  Pan_ang = 0.0; Pan_ang_wh = 0;
  Pan_ang_fr = 0;

  DO_DisplayFunctions();    // Update display

  Flm_max = 0; // Reset flame parameters.
```

```
  Flm_ang_max = 0;

  Auto = (digitalRead(AngAuto_pin)==0);
  oldAuto = Auto;
  Serial.println("Pan/Tilt Cal Complete"); // Msg to serial - we're done pan/tilt cal
}

void computeFlm(){
  Flm_read = analogRead(Flm_pin);                 // Get current output from flame detector
//+++++ changed numbers here to make it go to zero when no flame
  Flm_det = limit( ((800 - Flm_read) / 7),0,99);// Scale it to 0 to 99
  if(Flm_det > Flm_max) {     // If its the biggest
    Flm_max = Flm_det;        //   save the value
    Flm_ang_max = Pan_ang_wh; //   and the angle.
  }
}

void Wait4_1secFlag(){
  while(Flag1sec == FALSE) delay(1);
}
```

```
//=============================================================
//=====        Smokey Control Program ver 2.02          =====
//=====                                                 =====
//===== This program operates the motors so that Smokey   =====
//=====   point its flame detector in azimuth and         =====
//=====   elevation using pan / tilt controls.           =====
//=====                                                 =====
//===== The ^ button causes sensor to tilt up*,          =====
//=====   v button causes sensor to tilt down.           =====
//=====   > button pans right.                          =====
//=====   < button pans left.                           =====
//===== Smokey knows what time it is via the RTC: real    =====
//=====   time clock.  Smokey can sniff the air and       =====
//=====   tell you how many dust particles there are      =====
//=====   and how much CO2 and other chemicals are        =====
//=====   there. Fine dust particles, 1u, are registered  =====
//=====   as smoke, others are ignored.                   =====
//===== If there is an SD card/chip in the module, Smokey =====
//=====   will write this data to the drive with time and =====
//=====   date for each data point, recorded every minute.=====
//=====   Revision History:                              =====
//=====     Source version: smokey 1v8                    =====
//=====     Smokey_2v01:                                  =====
//=====                 Fixed comments                    =====
//=====                 Created Do_Man();                 =====
//=====     Smokey_2v02:                                  =====
//=====                Added Shift key                  =====
//=====                Added debounce on reset          =====
//=====                Added app code for flm history    =====
//=====                Recalibrated Flm det for noise    =====
//=============================================================

// String to name your Smokey on bluetooth
// mySmokeyName will be broadcast as the BlueTooth name of your Smokey
 #define mySmokeyName "Smokey.Flame1"
 #define changeName // comment out if NOT changing name for Bluetooth
 #define myFileName "DataLog.txt"
 // verbose = 1; // if you want more printouts
 #define EECalibrate FALSE

//=====  Library files to include
#include "CHS_SmokeyLib_1v31.h" // Library for general Arduino Mega stuff.
```

```
//=====================================================================
==
//=====    DO_functions list: code snippets for major functions       =====
//=====        Used to simplify main code to make logic flow obvious  =====
//=====================================================================
==
//    DO_1secFunctions();   Stuff to do once per second
//    DO_10secFunctions();  Stuff to do once per 10 secs
//    DO_1minFunctions();   Stuff to do once per 1 min
//    DO_Auto();         Test Auto/Man switch and pan/tilt accordingly
//    DO_command();         Get a command and execute it
//    DO_DisplayFunctions();Get data and write to App and LCD
#include "SmokeyDO_functions.h" // Link to the DO functions - this is right before executable
                          //   code because we need all the variables etc defined first.

//=====================================================================
//===== Overall setup function, calls specific setups for modules  =====
void setup(){
//===== SmokeyLib has setup functions for each module and misc stuff
  SmokeyGenSetup();         // Starts Serial monitor and prints a RESTART message
                        // ALSO: starts watchdog timer
  SmokeyStepperSetup();     // Defines the motor pins and turns off motors
  SmokeyRtcSetup();         // Starts any special functions for RTC
  SmokeyPmsA003Setup();     // Starts I2C communication with PmsA003.
  SmokeySgp30Setup();       // Starts SGP30 - figures out baseline update
  SmokeySDSetup();          // Starts the SD card module
  SmokeyLCDSetup();         // Starts LCD and writes labels
  SmokeyFlmSetup();         // Sets angle to zero
  SmokeyLedSetup();
  SmokeyBluefruitSetup();   // Goes through bluetooth startup procedure,
                        // .  sets to DATA mode and prints out results.
  delay(50);                // Execution more stable with delays -
                        //   here we are letting modules complete their
                        //   setups before we ask them for data
//=====  Get data from all modules to start with
  getPmsStandardString();   // Get PMSA003 and SGP30 data
  getTimeString();          // Also get the Time data
  getDateString();          //   and Date
  getSgpBLines();           // Get baselines from SGP & loads integers and byte versions
  getEEBLines();            // Get baselines from EEPROM & loads integer and byte versions
  getSgpString();           // Get new chemical data every second

//===== Make sure we are executing no commands to start with
    Command =0; newCmd=0; currentCmd=0; // Initialize commands
```

```
//===== Start up the DHT module.  It needs to figure out the timing on its interface here.
  dht.begin();    // Start DHT module - establish communication
  labelLCD();     // Print welcome screen to LCD
  delay(1000); WatchDog = 0;  // Leave welcome up for 2 secs
  delay(1000); WatchDog = 0;  //  while keeping watchdog happy
  forceManual();    // Check Auto/Man switch:if Auto, prompt for Man
                    //   return when switch is set to Man
  SettingHome = 1;  // Set to homing state to get display right
  calPanTilt();     // Go into manual mode one time during startup
                    //    to prompt user to put Flame det to home position
  SettingHome = 0;  // Back to normal

//++++++ Add this stuff to stop re-entry of ang reset in first time through loop
  delay(100);       // Wait a little bit
  while( digitalRead(AngRst_pin) == 0 ) delay(50); //Wait for reset button release
  Wait4_1secFlag(); // Synchronize to 1sec flag

}


//===== Main Loop: this loop repeats indefinitely until power down =====
void loop() {             // Main loop. Repeat forever.
    WatchDogDly = 60;     // Loop is running, so bump WD to 60 seconds.
    DO_AngRst();          // Check angle reset pin, if pushed go through AngRst function.
    DO_1secFunctions();   // 1 sec functions - mostly the CO2 and TVOC module
    DO_10secFunctions();  // 10 sec functions - mostly the DHT module
    DO_1minFunctions();   // 1min functions - mostly the SD drive data log
    DO_Auto();            // Check Auto/Man switch everytime through loop
    DO_Man();             // if Manual, do manual command
  }


ISR(TIMER5_OVF_vect){   // interrupt service routine every half second
  TCNT5 = timer5_counter;   // preload timer (from arduino app notes)
  Flag05sec = TRUE;
  if(timer5_repeat == 0) {  // need to skip every other call because
      timer5_repeat =1;     //  this timer runs every half second.
    } else {                // Only come here once per second.
      timer5_repeat = 0;    // Clear "repeat" flag.
      Flag1sec = TRUE;      // Set the 1sec flag.
      Count10sec = (Count10sec + 1)%10;       // Update 10 sec counter
      if (Count10sec == 0) Flag10sec = TRUE;  // Set 10sec flag on overflow
      Count1min = (Count1min +1)%60;          // Update 1min counter
```

```
    if (Count1min ==0) Flag1min = TRUE;     // Set 1min flag on overflow
    CountDisplay = (CountDisplay +1)%DisplayTime; // Update Dispaly counter
    if(CountDisplay == 0) FlagDisplay = TRUE;     // Set display on overflow
    WatchDog += 1;                        // Update watchdog counter
    if(WatchDog >= WatchDogDly) {           // If watchdog hits delay, reset processor
                                  // delay set to 10 at start, up to 300 at setup
      Serial.println("WDog Rst in 1 sec");  // Print watchdog warning
      delay(1000);                        // Pause for printint
      resetFunc(); // if WatchDog not reset in loop for 5min then reboot machine.
    }
   }
}
```

```
int DO_command(){
//=====================================================================
// Get new command, error check and execute
//====== Pushbutton Codes: codes received from BLuetooth App   =======
//  Pushing a button sends a 5 character code.
// . like: pushing 2 gives !B219
// . These patterns are decoded in GetCommand() routine which returns
// . the number of the pressed button (1 thru 8) or a -1 (release of
// . any button).
//=====================================================================

//    Serial.print("Start Cmd Proc, Current Cmd =");
//    Serial.println(currentCmd);
    newCmd = GetCommand();        // Always read from the App input on bluetooth
    if(newCmd < 0 || newCmd >8) { // Illegal command - cancel it
      currentCmd = 0;             // Zero out the command variables
      newCmd =0;
      MotorsOff();                // Make sure motors are off
      }
    ActiveCmd = (currentCmd<=8) && (currentCmd>=5); // 5<=Cmd<=8 is active command
    if(ActiveCmd == 0)  currentCmd = newCmd; // If its not active cmd, update it
// Active commands implemented
    if((currentCmd == 8) && (Auto == 0))  Pan(1);      // Pan right
    else if((currentCmd == 7) && (Auto == 0)) Pan(-1);   // Pan Left
    else if((currentCmd == 6) && (Auto == 0)) TiltDown(1);  // Tilt Down - not used at present
    else if((currentCmd == 5) && (Auto == 0)) TiltUp(1);    // Tilt Up - not used at present
    else if(currentCmd == 4){         // Pressed button 4 ... Date and Time
      AppShift = TRUE;
    } else if(currentCmd == 3){   // Pressed button 3 ... Battery & baselines
      if(AppShift != FALSE) {     // For App Display #n6
        currentDisplay = 6;
        updateFlm1Dsply();
        updateDisplay();
      } else {              // For App Display #3
      currentDisplay = 3; // Each button goes through same sequence
      updateNameDsply();
      delay(100);
      updateDisplay();
      currentCmd = 0;      // Commands 4,3,2,1 are button presses and get cancelled
      newCmd =0;           //   once implemented.
      delay(100);
      }
    } else if(currentCmd == 2){   // Pressed button 2 ... Chemical data CO2 and TVOC
      if(AppShift != FALSE){      // For App Display #4 & # 1
```

```
      currentDisplay = 5;
      updateFlm0Dsply();
      updateDisplay();
    } else {
      currentDisplay = 2;
      updateAirQualDsply();
      delay(100);
      updateDisplay();
      currentCmd = 0;      // Commands 4,3,2,1 are button presses and get cancelled
      newCmd =0;           //   once implemented.
      delay(100);
    }
  } else if(currentCmd == 1){   // Pressed button 1 ... Particulate data
    if(AppShift != FALSE){      // For App Display #4
      currentDisplay = 4;       // Button number is same as display number - stopped here
      updateTimeDateDsply();    // Format Time/Date for display
      delay(100);               // Give it some time to settle
      updateDisplay();          // Write out to display
      currentCmd = 0;     // Commands 4,3,2,1 are button presses and get cancelled
      newCmd =0;          //   once implemented.
      delay(100);         // Wait for settling again
    } else {                  // For App Display #1
      currentDisplay = 1;
      updatePartDsply();
      delay(100);
      updateDisplay();
      currentCmd = 0;      // Commands 4,3,2,1 are button presses and get cancelled
      newCmd =0;           //   once implemented.
      delay(100);
    }
  }
  return ActiveCmd; // Return ActiveCmd for Loop contrl
}
void DO_Man(){
  do {
    WatchDog = 0;   // User could hold button for a while, triggering watchdog so
                    //   we reset here in the command loop.
    Active = DO_command();  // Motion commands are active - they persist
                            //   while button is pressed, so we keep looping
                            //   here until we get Active= FALSE.
    delay(10);              // Delay to let Bluetooth get a command
  } while ( Active );       // "do while" loops are tested at end, always
                            //    execute at least once.
  if(Auto == 0) DO_DisplayFunctions();    // Now we update App and LCD only in manual
```

```
                                    // Auto mode updates displays when it reverses
                                    //   at end of a sweep.
  }



void DO_Auto(){
  Auto = (digitalRead(AngAuto_pin)==0); // Update value of Auto
  if(Auto == 0) return; // If not in Auto mode, return, we don't do anything.
//===== from here on we are in Auto mode, (Auto != 0)
  if(Pan_direction > 0)  {  // If panning in positive direction
    PanAuto(5);             // Pan +5 steps
    if(Pan_ang >= 90.0 ) {  // If angle > 90 degrees, reverse pan direction
      Pan_direction = -1;   // Reverse
      nextTiltMove();       // Update Tilt at the reverse point
      DO_DisplayFunctions();// Update display here (not every 5 secs - too jerky)
    }
  } else {                  // If panning in negative direction
    PanAuto(-5);            // Pan -5 steps
    if(Pan_ang <= -90.0 ) { // If angle < 90 degress, reverse pan direction
      Pan_direction = 1;    // Reverse
      nextTiltMove();       // Update Tilt at reverse point
      DO_DisplayFunctions();// Update display at reverse point
    }
  }
}

void DO_AngRst(){ // If angle reset button is pressed, do an angle reset
  if(digitalRead(AngRst_pin) == 0)  {  // Is reset button pressed?
    while(digitalRead(AngRst_pin) == 0) delay(100);
    forceManual();    // Check Auto/Man switch:if Auto, prompt for Man
                      //   return when switch is set to Man
    SettingHome = 1;  // Set to homing state to get display right
    calPanTilt();     // Calibrate auto manual mode.
                      //    to prompt user to put Flame det to home position
    SettingHome = 0;  // Back to normal
    AngRst_valid =  TRUE;
    delay(100);
    while(digitalRead(AngRst_pin) == 0) delay(50);
  }
  return;
}

void DO_1secFunctions(){
```

```
  if(Flag1sec == TRUE){
    readSgpData();
    Flag1sec = FALSE;
    WatchDog = 0;   // Reset watchdog when we get here - if we don't we'll need to reboot
  }
}

void DO_10secFunctions(){
  if (Flag10sec == TRUE){
    getPmsStandardString();   // Get PMSA003 and SGP30 data on every iteration
    getDhtData();             // Get Humidity/Temp data from DHT11 module
    pm10_sum += pmsData.pm10_standard; // Add up to compute average
    temp_sum += dht_f;
    hum_sum += dht_h;
    pm10_count += 1;                  // Count how many we've added
    if (pm10_count >= 6){
      pm10_avg = pm10_sum / 6;          // Div by 6 at 10 secs means one minute overall
      pm10_count = 1;
      pm10_sum = 0;
      temp_avg = temp_sum / 6.0;
      temp_sum = 0.0;
      hum_avg = hum_sum / 6.0;
      hum_sum = 0.0;

    Flag10sec = FALSE;
    }
  }
}

void DO_1minFunctions(){
  if(Flag1min){
  writeSDdata();  // Write out date, time, particulates, and CO2/TVOC per min.
  printDhtData(); // Print humidity temp data every minute.
  Flag1min = FALSE;
  }
}

void DO_DisplayFunctions(){
  if(FlagDisplay) {   // Read eCO2 and TVOC every DisplayTime seconds (like the demo)
    FlagDisplay = FALSE;
    EEsaveCount = (1 + EEsaveCount) % EEsaveSet;    // Increment and reset on overflow
EEsaveCount

    if(EECalibrate){
```

```
      if( (EEsaveCount == 0)  ) {
        EETVOC_base = TVOC_base;  EEeCO2_base = eCO2_base;
        putEEBLines();
      }
    } else {
      getEEBLines();            // Get baselines from EEPROM & loads integer and byte versions
      TVOC_base = EETVOC_base;    // Put EE base numbers into SGP variables.
      eCO2_base = EEeCO2_base;
      putSgpBLines();           // Output to SGP chip
    }
    getTimeString();          // Also get the Time data
    getDateString();
    getSgpBLines();           // Get baselines from SGP & loads integers and byte versions
    getEEBLines();            // Get baselines from EEPROM & loads integer and byte versions
    getSgpString();           // Get new chemical data every second
    getSerialInput();         // Check if theres override values for LED limits
    updateLCD();
    if(ble.isConnected() ){   // If Bluetooth is connected, update the display on App
      if(currentDisplay == 6) { updateFlm1Dsply(); updateDisplay();  delay(80); }
      else if(currentDisplay == 5) { updateFlm0Dsply(); updateDisplay();  delay(80); }
      else if(currentDisplay == 4) { updateTimeDateDsply(); updateDisplay();  delay(80); }
      else if(currentDisplay == 3) { updateNameDsply();     updateDisplay();  delay(80); }
      else if(currentDisplay == 2) { updateAirQualDsply();  updateDisplay();  delay(80); }
      else if(currentDisplay == 1) { updatePartDsply();     updateDisplay();  delay(80); }
    }
    writeLeds(); // Update Led displays
  }
}
```