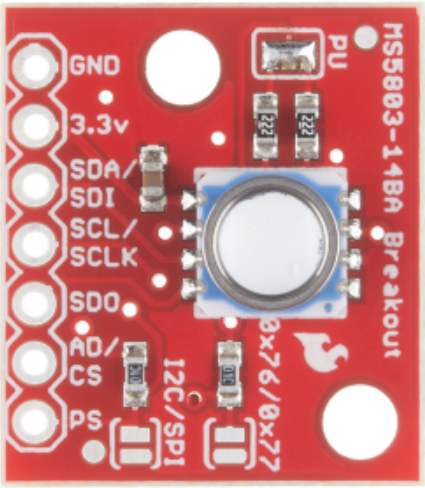
MS5803 Pressure Sensor



This is the MS5803-14BA Pressure Sensor Breakout, a high resolution pressure sensor with both an I^2C and SPI interface. This MEMS pressure sensor measures the absolute pressure of the fluid around it which includes air, water, and anything else that acts like a viscous fluid. Depending on how you interpret the data, you can determine altitude, water depth, or any other tasks that require an accurate pressure reading. What makes the MS5803-14BA unique is the the gel membrane and antimagnetic stainless steel cap that protects against 30 bar water pressure (**300 meters underwater**).

We have broken out all the pins you need including GND and 3.3V for power, SDA/SDI and SCL/SCLK for an I^2C interface and SD0, AD/CS, and PS for a SPI interface. The MS5803-14BA Breakout offers a resolution range of 1 / 0.6 / 0.4 / 0.3 / 0.2 mbar. Be aware that to switch between I^2C and SPI interfaces a little bit of soldering between solder pads will be required, check the hookup guide below for more information.

Heads up! While the IC is capable of outputting data via I2C and SPI, the Arduino Library and example was only written to output via I2C! You'll need to modify the jumpers on the board for SPI mode by removing solder from the pull-up resistors jumper pads and close the other two jumpers. Additionally, you will need to write a library to communicate with the MS5803 in SPI mode if you plan on using this with Arduino.

Code for Reading Sensors and Computing Depth

```
   // Read temp and pressure from the sensor
  // TODO: Consider reducing the read frequency to reduce power consumption however, I do not think the sensor
uses much power.
  D1 = ms5803_cmd_adc( SENSOR_CMD_ADC_D1 + SENSOR_CMD_ADC_4096);    // read uncompensated pressure
  D2 = ms5803_cmd_adc( SENSOR_CMD_ADC_D2 + SENSOR_CMD_ADC_4096);    // read uncompensated temperature
  // calculate 1st order pressure and temperature correction factors (MS5803 1st order algorithm)
  deltaTemp = D2 - sensorCoefficients[5] * pow( 2, 8 );
  sensorOffset = sensorCoefficients[2] * pow( 2, 16 ) + ( deltaTemp * sensorCoefficients[4] ) / pow( 2, 7 );
  sensitivity = sensorCoefficients[1] * pow( 2, 15 ) + ( deltaTemp * sensorCoefficients[3] ) / pow( 2, 8 );
  // calculate 2nd order pressure and temperature (MS5803 2st order algorithm)
  temperature = ( 2000 + (deltaTemp * sensorCoefficients[6] ) / pow( 2, 23 ) ) / 100;
  pressure = ( ( ( ( D1 * sensitivity ) / pow( 2, 21 ) - sensorOffset) / pow( 2, 15 ) ) / 10 );

  // The pressure at sea level is 1013.25 mbars so if the pressure is lower, then the user is not at sea level
and an altitude compensation factor is needed.
  // This needs to be improved to take the lowest reading since power up (not just the first reading above sea
level) for situations when the unit does not power up until the user is in the water.
  if ( pressure < 1013.25 && !altitudeCompSet ) {
    altitudeCompensation = 1013.25 - pressure;
    altitudeCompSet = true;
  }

  // calculate depth
  if ( meters ) depth = ( ( pressure + altitudeCompensation ) - 1013.25 ) / 100.52; // There are 100.52 mbars
per meter of depth.
  else depth = ( ( pressure + altitudeCompensation ) - 1013.25 ) / 30.64; // There are 30.64 mbars per foot of
depth.

  // If there is a pressure rebound, make sure the depth does not go negative.
  if ( depth < 0.0 ) depth = 0.0;

  // Convert the temp to ferinheight if the user has selected that unit.
  if ( ferinheight ) temperature = ( ( temperature * 9.0 ) / 5.0 ) + 32.0;
```

Full Code

Dive_Computer_Code.ino

```
/*****************************************************************
 * This is the Teensy 3.0 code for the DIY Dive Computer.
 *
 * Written by Victor Konshin.
 * BSD license, check license.txt for more information.
 * All text above must be included in any redistribution.
 *****************************************************************/

#include <Wire.h>
#include <EEPROM.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include <SD.h>
#include "RTClib.h"
#include <Time.h>
#include "PITimer.h"
#include <Bounce.h>
#include <SPI.h>
//#include "UILibrary.h"

// Sensor constants:
#define SENSOR_CMD_RESET      0x1E
#define SENSOR_CMD_ADC_READ   0x00
#define SENSOR_CMD_ADC_CONV   0x40
#define SENSOR_CMD_ADC_D1     0x00
#define SENSOR_CMD_ADC_D2     0x10
#define SENSOR_CMD_ADC_256    0x00
```

```
#define SENSOR_CMD_ADC_512    0x02
#define SENSOR_CMD_ADC_1024   0x04
#define SENSOR_CMD_ADC_2048   0x06
#define SENSOR_CMD_ADC_4096   0x08

static unsigned char PROGMEM dive_logo_bmp[] =
{
  B00011111, B11111111, B11111111,
  B00001111, B11111111, B11111111,
  B00000111, B11111111, B11111111,
  B10000011, B11111111, B11111111,
  B11000001, B11111111, B11111111,
  B11100000, B11111111, B11111111,
  B11110000, B01111111, B11111111,
  B11111000, B00111111, B11111111,
  B11111100, B00011111, B11111111,
  B11111110, B00001111, B11111111,
  B11111111, B00000111, B11111111,
  B11111111, B10000011, B11111111,
  B11111111, B11000001, B11111111,
  B11111111, B11100000, B11111111,
  B11111111, B11110000, B01111111,
  B11111111, B11111000, B00111111,
  B11111111, B11111100, B00011111,
  B11111111, B11111110, B00001111,
  B11111111, B11111111, B00000111,
  B11111111, B11111111, B10000011,
  B11111111, B11111111, B11000001,
  B11111111, B11111111, B11100000,
  B11111111, B11111111, B11110000,
  B11111111, B11111111, B11111000 };

static unsigned char PROGMEM log_logo_bmp[] =
{
  B00000100, B00000000, B00100000,
  B00000111, B00000000, B11100000,
  B00000111, B11000011, B11100000,
  B00110111, B11100111, B11101100,
  B00110111, B11100111, B11101100,
  B00110111, B11100111, B11101100,
  B10110111, B11100111, B11101101,
  B10110111, B11100111, B11101101,
  B10110111, B11100111, B11101101,
  B10110111, B11100111, B11101101,
  B10110111, B11100111, B11101101,
  B10110111, B11100111, B11101101,
  B10110111, B11100111, B11101101,
  B10110111, B11100111, B11101101,
  B10110111, B11100111, B11101101,
  B10110111, B11100111, B11101101,
  B10110011, B11100111, B11001101,
  B10111110, B11100111, B01111101,
  B10111111, B00100100, B11111101,
  B10111111, B10100101, B11111101,
  B11100001, B11000011, B10000111,
  B11111111, B00100100, B11111111,
  B11111111, B11000011, B11111111 };

static unsigned char PROGMEM settings_logo_bmp[] =
{
  B00000000, B00000000, B00000000,
  B00000000, B00000000, B00110000,
  B00000000, B00000000, B11111100,
  B00000000, B00000000, B11111100,
  B00000000, B00000001, B11001110,
  B00000000, B00000001, B11001110,
  B00000000, B11000000, B11111100,
  B00010001, B11100010, B11111100,
  B00111001, B11100111, B00110000,
  B01111111, B11111111, B10000000,
```

```
  B00111111, B11111111, B00000000,
  B00011111, B11111110, B00000000,
  B00011111, B00111110, B00000000,
  B01111110, B00011111, B10000000,
  B11111100, B00001111, B11000000,
  B11111100, B00001111, B11000000,
  B01111110, B00011111, B10000000,
  B00011111, B00111110, B00000000,
  B00011111, B11111110, B00000000,
  B00111111, B11111111, B00000000,
  B01111111, B11111111, B10000000,
  B00111001, B11100111, B00000000,
  B00010001, B11100010, B00000000,
  B00000000, B11000000, B00000000 };

static unsigned char PROGMEM up_arrow[] =
{
  B00011000,
  B00111100,
  B01111110,
  B11111111 };

static String PROGMEM months[] = {
  "???", "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };

static unsigned int accent_tone[] = { // Structure: # of tones then frequencies
  4, 1000, 1200, 1400, 1600
};

static unsigned int trill_tone[] = { // Structure: # of tones then frequencies
  4, 1000, 1400, 1000, 1400
};

//#define SHOW_LAYOUT // Uncomment this to see bounding boxes on UI elements - makes laying out items easier.

// ****************************************** Sensor Parameters *****
unsigned int      sensorCoefficients[8];           // calibration coefficients
unsigned long     D1                      = 0;    // Stores uncompensated pressure value
unsigned long     D2                      = 0;    // Stores uncompensated temperature value
float             pressure                = 0;    // Stores actual pressure in mbars
float             temperature             = 0;    // Stores actual temp in degrees C.
float             deltaTemp               = 0;
float             sensorOffset            = 0;
float             sensitivity             = 0;

// ****************************************** Hardware Parameters *****
const int         button1Pin              = 0;     // Pin for the right button.
const int         button2Pin              = 1;     // Pin for the left button.
const int         button3Pin              = 2;     // Pin for a third button - the inital design will not use
it but I will have parts on the board to support it.
const int         button4Pin              = 3;     // Pin for a fourth button - the inital design will not use
it but I will have parts on the board to support it.
const int         oledReset               = 4;     // Pin for the OLED reset.
const int         alertLEDPin             = 5;     // A blinking LED is attached to this pin.
const int         chargingPin             = 6;     // Pin for receiving the charging signal from the charging
circuit.
const int         sensorSelectPin         = 9;     // Sensor device select
const int         sdChipSelect            = 10;    // For the SD card breakout board

const int         batteryLevelPin         = A0;    // Pin for monitoring the battery voltage.
const int         wakeUpPin               = A1;    // Pin for water wake up.
const int         speakerPin              = A8;    // Pin for the piezo speaker (PWM out).
const int         fakeDepthPin            = A9;    // Connect a potentiameter to this pin for testing and
debugging

const int         profileSize             = 89;    // The number of depths that are saved to EEPROM for the
dive profile graph.
Sd2Card           card;
SdVolume          volume;
SdFile            root;
```

```
// ************************************** System Parameters *****
Adafruit_SSD1306  display(oledReset);               // Allocate for the OLED
unsigned long     currentTime              = 0;     // Stores the current date/time for use throughout the code.
int               batteryLevel             = 128;   // Current battery capacity remaining.
int               displayMode              = 0;     // Stores the current display mode: 0 = menu, 1 = dive, 2 =
log, 3 = settings.
int               currentMenuOption        = 0;     // Stores which menu option is being displayed. 0 = dive, 1
= log, 2 = settings.
int               diveModeDisplay          = 0;     // There are multiple dive mode displays. This is the one
currently being viewed.
float             altitudeCompensation     = 0.0f;  // The altitude compensation factor.  If the user is above
sea level, this stored the difference between air pressure and sea level pressure.
boolean           altitudeCompSet          = false; // Set this to true when the altitudeCompensation variable
gets set (locked in).
unsigned long     lLastDebounceTime        = 0;     // the last time the output pin was toggled
unsigned long     rLastDebounceTime        = 0;     // the last time the output pin was toggled
int               debounceDelay            = 300;   // the debounce time; increase if the output flickers
boolean           accentBlink              = false; // Store for the accent indicator blink state.
boolean           batteryBlink             = false; // Store for the battery indicator blink state.
boolean           safetyStopBlinkState     = false; // Store for the safety stop blink state.
boolean           inWater                  = false; // Goes true if the water sensor detects that the device is
submerged.
int               inWaterCounter           = 0;     // Counter used to determine if the device is in water.
boolean           alertShowing             = false; // true = there is an alert showing.
uint32_t          alertTime                = 0;     // Stores time the alert was displayed.
uint32_t          lastDataRecord           = 0;     // Stores the last time data was recorded in unixtime.
uint32_t          lastColonStateChange     = 0;     // Stores the last time the clock was updates to that the
blink will happen at 1Hz.
uint32_t          lastProfileRecord        = 0;     // The real time that the last profile record was saved.
uint32_t          safetyStopBlink          = 0;     // Stores the last time the safety stop banner was updates
to that the blink will happen at 1Hz
const int         numberOfDiveDisplays     = 3;     // The number of dive displays (zero based).
int               toneCounter              = 0;     // Store the current tone being played in the tone sequence.
boolean           toneActive               = false; // Set true to play a tone.
int               tonePlaying              = 0;     // The ID of the tone to play (0 = accent to fast, 1 =
trill).
const int         accentSamples            = 10;    // Number of depth samples to store for accent rate
calculations.
boolean           safetyStopNeeded         = false; // Goes true when the user's depth surpasses
requireSafetyStopAfter.
boolean           safetyStop               = false; // When true the safety stop screen is shown.
uint32_t          safetyStopTime           = 0;     // Stores the time the safety stop started.
char              currentDataFile[]        = "00000000.txt"; // This is the name of the current data file.

// ************************************** Dive parameter storage *****
boolean           diveMode                 = false; // Stores dive mode or surface mode state: 0 = surface
mode, 1 = dive mode.
uint32_t          diveStart                = 0;     // This stores the dive start time in unixtime.
int               diveTime                 = 0;     // This stores the current dive time in seconds
float             depth                    = 0.0f;  // Current Depth
int               maxDepth                 = 0;     // Max Depth
int               nitrogen                 = 0;     // Current nitrogen saturation level
int               oxygen                   = 0;     // Current oxygen saturation level.
float             depths[accentSamples];            // Stores the last 10 depth reading.
unsigned long     depthMicros[accentSamples];       // Stores the times of the last ten depth readings (to
calculate accent rate).
int               profilePointer           = 0;     // The computer saves depth data to EEPROM for the graph
function.  This is the pointer to the current location data is being saved.
int               profileSamples           = 0;     // Number of samples collected (when less than the max).
int               depthsPointer            = 0;     // Pointer for the depths and depthMicros variables.
float             accentRate               = 0;     // This will go away when accent rate is calculated from
recent pressure readings.

// ************************************** User Preferences *****
boolean           time12Hour               = true;  // true = 12 hour time, false = 24 hour time.
boolean           clockBlink               = true;  // true = blink the colon, false = don't blink the colon.
boolean           ferinheight              = true;  // true = Ferinheight, false = Celsius.
boolean           meters                   = false; // true = meters, false = feet.
boolean           doubleAccentBelowRec     = true;  // True = 60ft/min accent rate below recreational limits
(130ft). 30ft/min otherwise.
int               recordInterval           = 10;    // This is the inteval in which it will record data to the
```

```
SD card in minutes.
int               profileInterval          = 0;     // The interval that dive profile depths are recorded to
EEPROM (for the graphing funtion).
boolean           playAccentTooFastTone    = true;  // If this is true, then the accent too fast tone will play.
float             requireSafetyStopAfter   = 30.0;  // The depth the user needs to pass for the safety stop to
be enabled;
int               safetyStopLength         = 180;   // Length of the standard safety stop.

// ***************************************************************


// ******************************************************************************
//                                  Setup
// ******************************************************************************

void setup()   {

  // Read the preferenced from the EEPROM
  readPrefereces();

  // Start the serial ports.
  Serial.begin( 115200 );
  Wire.begin();

  // Display Setup
  display.begin( SSD1306_SWITCHCAPVCC, 0x3D );  // initialize with the I2C addr 0x3D
  display.display(); // show splashscreen
  delay( 1000 );
  display.clearDisplay();
  display.display();
  display.setTextColor( WHITE );

  // Set up the warning LED Pin
  pinMode( alertLEDPin, OUTPUT ); // Warning LED
  digitalWrite( alertLEDPin, HIGH );

  // Set up the SD card pin select
  pinMode( sdChipSelect, OUTPUT );
  digitalWrite( sdChipSelect, HIGH );

  // Set pins for charging detection
  pinMode( chargingPin, INPUT );

  // Set up the sensor SPI select Pin
  pinMode( sensorSelectPin, OUTPUT );
  digitalWrite( sensorSelectPin, HIGH );
  SPI.begin(); //see SPI library details on arduino.cc for details
  SPI.setBitOrder( MSBFIRST );
  SPI.setClockDivider( SPI_CLOCK_DIV2 );
  delay( 10 );

  // Read sensor coefficients - these will be used to convert sensor data into pressure and temp data
  ms5803_reset_sensor(); // resetting the sensor on startup is impoortant
  for (int i = 0; i < 8; i++ ){
    sensorCoefficients[ i ] = ms5803_read_coefficient( i );  // read coefficients
    delay(10);
  }

  // Check the CRC data returned from the sensor to ensure data integrity.
  unsigned char n_crc;
  unsigned char p_crc = sensorCoefficients[ 7 ];

  n_crc = ms5803_crc4( sensorCoefficients ); // calculate the CRC
  // If the calculated CRC does not match the returned CRC, then there is a data integrity issue.
  // Check the connections for bad solder joints or "flakey" cables. If this issue persists, you may have a bad
sensor.
  if ( p_crc != n_crc ) {
    showAlert ( 0, "The sensor CRC check failed. There is a data integrity issue with the sensor." );
  }

  //  Setup the user input buttons.
```

```
  pinMode( button1Pin, INPUT_PULLUP ); // right button
  pinMode( button2Pin, INPUT_PULLUP ); // left button
  attachInterrupt( button1Pin, rightPressed, FALLING );
  attachInterrupt( button2Pin, leftPressed, FALLING );

  // Wake up pin is also the "in water" pin
  pinMode( wakeUpPin, INPUT );

  // initialize access to the SD card
  Serial.print( "Initializing SD card..." );
  // make sure that the default chip select pin is set to
  // output, even if you don't use it:

  // see if the card is present and can be initialized:
  if ( !SD.begin( sdChipSelect ) ) {
    Serial.println( "Card failed, or not present" );
    // don't do anything more:
    showAlert( 0, "Failed to initialize the SD card.  Please make sure the card isinstalled correctly." );
  }
  else {
    Serial.println( "card initialized." );
  }
  //showAlert (0, "You are severly bent and about to die. Youshould surface while you still can.");

  // Was a nice idea but it seems that these timers interfere with other things happening...
  //  PITimer0.period(.1);
  //  PITimer0.start(cycleTone);

  // Give everything a half sec to settle
  delay( 500 );

  // Make sure the real time clock is set correctly and set some inital parameters
  Teensy3Clock.set( 0 );//DateTime( __DATE__, __TIME__ ).unixtime() ); // comment out if there is a backup
battery connected to the real time clock.
  currentTime = Teensy3Clock.get();
  lastDataRecord = currentTime;
  lastColonStateChange = currentTime;
  diveStart = currentTime;

}

// ******************************************************************************
// ******************************************************************************
//                                  Main Loop
// ******************************************************************************
// ******************************************************************************

void loop() {

  // If there is an active tone, then play the next in the sequence.
  cycleTone();

  // Fake values for debugging
  nitrogen++;
  oxygen++;
  if ( nitrogen > 255 ) nitrogen = 0;
  if ( oxygen > 255 ) oxygen = 0;

  // Read real time clock.
  currentTime = Teensy3Clock.get();

  // Read temp and pressure from the sensor
  // TODO: Consider reducing the read frequency to reduce power consumption however, I do not think the sensor
uses much power.
  D1 = ms5803_cmd_adc( SENSOR_CMD_ADC_D1 + SENSOR_CMD_ADC_4096);    // read uncompensated pressure
  D2 = ms5803_cmd_adc( SENSOR_CMD_ADC_D2 + SENSOR_CMD_ADC_4096);    // read uncompensated temperature
  // calculate 1st order pressure and temperature correction factors (MS5803 1st order algorithm)
  deltaTemp = D2 - sensorCoefficients[5] * pow( 2, 8 );
  sensorOffset = sensorCoefficients[2] * pow( 2, 16 ) + ( deltaTemp * sensorCoefficients[4] ) / pow( 2, 7 );
  sensitivity = sensorCoefficients[1] * pow( 2, 15 ) + ( deltaTemp * sensorCoefficients[3] ) / pow( 2, 8 );
  // calculate 2nd order pressure and temperature (MS5803 2st order algorithm)
```

```
  temperature = ( 2000 + (deltaTemp * sensorCoefficients[6] ) / pow( 2, 23 ) ) / 100;
  pressure = ( ( ( ( D1 * sensitivity ) / pow( 2, 21 ) - sensorOffset) / pow( 2, 15 ) ) / 10 );

  // The pressure at sea level is 1013.25 mbars so if the pressure is lower, then the user is not at sea level
and an altitude compensation factor is needed.
  // This needs to be improved to take the lowest reading since power up (not just the first reading above sea
level) for situations when the unit does not power up until the user is in the water.
  if ( pressure < 1013.25 && !altitudeCompSet ) {
    altitudeCompensation = 1013.25 - pressure;
    altitudeCompSet = true;
  }

  // calculate depth
  if ( meters ) depth = ( ( pressure + altitudeCompensation ) - 1013.25 ) / 100.52; // There are 100.52 mbars
per meter of depth.
  else depth = ( ( pressure + altitudeCompensation ) - 1013.25 ) / 30.64; // There are 30.64 mbars per foot of
depth.

  // If there is a pressure rebound, make sure the depth does not go negative.
  if ( depth < 0.0 ) depth = 0.0;

  // Convert the temp to ferinheight if the user has selected that unit.
  if ( ferinheight ) temperature = ( ( temperature * 9.0 ) / 5.0 ) + 32.0;

  // This code will read a potentiometer and over ride the depth reading of the sensor if it is larger than the
sensor's reading. This is used for testing and debugging.
  float fakeDepthReading = analogRead(fakeDepthPin) / 30.0f;
  if (fakeDepthReading > depth) depth = fakeDepthReading;

  // Records a round robin buffer of depth readings for accent rate calculations.
  depths[depthsPointer] = depth;
  depthMicros[depthsPointer] = micros();
  depthsPointer++;
  if (depthsPointer > accentSamples) depthsPointer = 0;

  // Set Max Depth
  if ( depth > maxDepth ) maxDepth = depth;

  // Enable safety stop.
  if ( depth > requireSafetyStopAfter ) {
    safetyStopNeeded = true;
    safetyStop = false;
  }

  // If the user accents to safety stop levels, then start the safety stop.
  if ( safetyStopNeeded && depth < ( requireSafetyStopAfter - 10.0 ) && !safetyStop ) {
    safetyStop = true;
    safetyStopTime = now() + safetyStopLength; // add 3 minutes.
  }

  if ( depth < requireSafetyStopAfter && safetyStop && safetyStopTime < now() ) {
    safetyStopNeeded = false;
    safetyStop = false;
  }

  // Fake battery level for testing.
  //  batteryLevel -= 1;
  //  if (batteryLevel < 0) batteryLevel = 255;

  // In water detection.  This will tell if the user is in the water. Not exactly sure what it will be used for
at this point but the code is here.
  if ( analogRead(A1) == 0 ) inWaterCounter++;
  else inWaterCounter--;

  if ( inWaterCounter > 100 ) {
    inWater = true;
    if ( inWaterCounter > 150 ) inWaterCounter = 150;
  }

  if ( inWaterCounter < 0 ) {
    inWater = false;
```

```
    inWaterCounter = 0;
  }

  // Dive start detection.  When the user decends below 4 feet, then the dive will start.
  if ( diveMode == false && depth > 4.0 ) {
    diveMode = true; // Set global "dive/surface" mode flag
    diveStart = now(); // reset the dive timer.
    diveTime = 0;
    maxDepth = 0; // reset the max depth.
    inWater = true;  // If you are 4 feet under water, it's safe to assume that you are in the water, if this
flag has not already been set...
    inWaterCounter = 150;
    displayMode = 1; //  Since this conditional is only called when a dive starts, put the display into dive
mode if it isn't already.
    clearDiveProfile();
    setupNewDataFile();
  }

  // TODO: Probably need something similar to the inWaterCounter to prevent multiple dive start records if the
user is hanging out between 3 and 4 feet.
  if ( diveMode == true && depth < 4.0 ) {
    diveMode = false;
    safetyStopNeeded = false;
    safetyStop = false;
  }
  Serial.print("Depth = ");
  Serial.print(depth);
    Serial.print(" diveMode = ");
  Serial.println(diveMode);

  // Things to do when in dive mode.
  if ( diveMode ) {
    diveTime = now() - diveStart;
    logData();  // Data will only log when (diveTime % recordInterval == 0)
    recordDiveProfileValue();
    // TODO: Calculate nitrogen
    // TODO: Calculate oxygen
  }

  // Calculate the accent rate.  This is where the LED gets activated and alert tone gets initiated.
  calculateAccentRate();

  // if there is no alert condition then show the normal display.
  // TODO: The way this is currently written, the display is constantly being updated. This is not efficient.
  // TODO: Need to change this so only areas of the screen that need updating are updated.  This will likely
improve battery performance.
  if ( !alertShowing ) {
    display.clearDisplay();
    display.setTextColor( WHITE );

    // 0 = menu, 1 = dive, 2 = log, 3 = settings
    if ( displayMode == 0 ) { // Menu Display
      drawMenu();
    }
    else if ( displayMode == 1 ) { // Dive Display
      // TODO: Make it easier to add dive display modes.
      if ( safetyStop ) {
        drawSafetyStopScreen();
      }
      else {
        if ( diveModeDisplay == 0 ) {
          drawDiveDisplayA();
        }
        else if ( diveModeDisplay == 1 ) {
          drawDiveDisplayB();
        }
        else if ( diveModeDisplay == 2 ) {
          drawDiveDisplayC(); // TODO: add a low power view - a view that has the minimum number of "on"
pixels.  OLED displays only power on pixels.  Consider reducing sensor read freq for this view.
        }
        else {
```

```
          drawDiveDisplayD();
        }
      }
    }
    else if ( displayMode == 2 ) { // Log Book Display
      drawLogBook();
    }
    else if ( displayMode == 3 ) { // Settings Display
      drawSettings();
    }
    display.display();
  }
}

// ****************************************************************************
//                                 Main Menu
// ****************************************************************************


void drawMenu() {

  display.setTextColor( WHITE );

  display.setTextSize( 1 );
  if ( currentMenuOption == 0 ) {
    display.setCursor( 37, 44 );
    display.println( "Dive Mode" );
    display.drawBitmap( 52, 16, dive_logo_bmp, 24, 24, WHITE );
  }
  else if ( currentMenuOption == 1 ) {
    display.setCursor( 40, 44 );
    display.println( "View Log" );
    display.drawBitmap( 52, 16, log_logo_bmp, 24, 24, WHITE );
  }
  else if ( currentMenuOption == 2 ) {
    display.setCursor( 40, 44 );
    display.println( "Settings" );
    display.drawBitmap( 52, 16, settings_logo_bmp, 24, 24, WHITE );
  }

  display.fillRect( 0, 0, 128, 9, WHITE );
  display.setTextColor( BLACK );
  display.setCursor( 36, 1 );
  display.setTextSize( 1 );
  display.println( "MAIN MENU:" );

  drawButtonOptions( "SEL", ">>", true, true );

  display.setTextColor( WHITE );

  // Show essensial data if the user is in dive mode.
  if ( diveMode ) {
    drawDepth( depth, 0, 26, 9, 1, true, "Depth:" ); // Parameters: depth, x, y, size, bold, header string
    drawAccentRateArrows( 120, 20 ); // Parameters: x, y
  }
}

// ****************************************************************************
//                              User Interactions
// ****************************************************************************

void rightPressed() {

  // TODO: Replace debounce code with Debounce library since it provides more functionality.
  if ( ( millis() - rLastDebounceTime ) < debounceDelay ) {
    return;
  }

  rLastDebounceTime = millis();

  if ( alertShowing ) {
```

```
    killAlert();
    return;
  }
  if ( displayMode == 0 ) { // Menu
    currentMenuOption ++;
    if ( currentMenuOption > 2 ) currentMenuOption = 0;
  }

  if ( displayMode == 1 ) { // Dive
    diveModeDisplay++;
    if ( diveModeDisplay > numberOfDiveDisplays ) diveModeDisplay = 0;
  }
}

void leftPressed() {

  // TODO: Replace debounce code with Debounce library since it provides more functionality.

  if ( ( millis() - lLastDebounceTime ) < debounceDelay ) {
    return;
  }

  lLastDebounceTime = millis();

  if ( displayMode == 0 ) { // Menu
    displayMode = currentMenuOption + 1;
    return;
  }

  if ( displayMode == 1 ) { // Dive
    displayMode = 0;
    return;
  }

  if ( displayMode == 2 ) { // Log
    displayMode = 0;
    return;
  }

  if ( displayMode == 3 ) { // Settings
    displayMode = 0;
    writePreferences(); // save the preferences to the EEPROM
    return;
  }
}

// ****************************************************************************
//                              Dive Mode Displays
// ****************************************************************************

void drawDiveDisplayA() {

  drawDiveTime( diveTime, 62, 0, 9, 2, false, "Dive Time:" ); // Parameters: Time in seconds, x, y, size, bold
  drawDepth( maxDepth, 14, 31, 9, 2, false, "Max:" ); // Parameters: depth, x, y, size, bold, header string
  drawDepth( depth, 6, 0, 9, 3, true, "Depth:" ); // Parameters: depth, x, y, size, bold, header string
  drawAccentRateBars( 71, 24 );
  drawAccentRate( 70, 45, 1, true, true );
  drawSaturation( nitrogen, true );
  drawSaturation( oxygen, false );
  drawButtonOptions( "^^", ">>", true, true );
}

void drawDiveDisplayB() {

  drawDiveTime( diveTime, 56, 28, 9, 2, false, "Dive Time:" ); // Parameters: Time in seconds, x, y, heading
offset, size, bold
  drawDepth( maxDepth, 10, 28, 9, 2, false, "Max:" ); // Parameters: depth, x, y, heading offset, size, bold,
header string
  drawDepth( depth, 10, 2, 9, 2, true, "Depth:" ); // Parameters: depth, x, y, heading offset, size, bold,
header string
  drawAccentRateArrows( 96, 0 ); // Parameters: x, y
```

```
  drawSaturation( nitrogen, true ); // Parameters: value and a boolean to indicate nitrogen or oxygen (true =
nitrogen, false = oxygen)
  drawSaturation( oxygen, false ); // Parameters: value and a boolean to indicate nitrogen or oxygen (true =
nitrogen, false = oxygen)
  drawButtonOptions( "^^", ">>", true, true ); // Draws the button bar at the bottom of the screen. Parameters:
Left button title, right button title, show time/temp bar, show battery.

}

// ++++++ DEPTH ONLY ++++++
void drawDiveDisplayC() {

  drawDepth( depth, 20, 0, 13, 5, true, "Depth:" ); // Parameters: depth, x, y, heading offset, size, bold,
header string
  drawAccentRateArrows( 110, 16 ); // Parameters: x, y
  drawSaturation( nitrogen, true ); // Parameters: value and a boolean to indicate nitrogen or oxygen (true =
nitrogen, false = oxygen)
  drawSaturation( oxygen, false ); // Parameters: value and a boolean to indicate nitrogen or oxygen (true =
nitrogen, false = oxygen)
  drawButtonOptions( "^^", ">>", false, true ); // Draws the button bar at the bottom of the screen.
Parameters: Left button title, right button title, show time/temp bar, show battery.
}

// ++++++ Dive Profile View ++++++
void drawDiveDisplayD() {
  drawDiveProfileGraph( 0, 0 );
  drawDepth( depth, 91, 0, 9, 1, true, "Depth:" ); // Parameters: depth, x, y, heading offset, size, bold,
header string
  drawAccentRateArrows( 120, 22 ); // Parameters: x, y
  drawButtonOptions( "^^", ">>", true, true ); // Draws the button bar at the bottom of the screen. Parameters:
Left button title, right button title, show time/temp bar, show battery.
}

// ******************************************************************************
//                                  Record Data
// ******************************************************************************

void createFileName( int diveNumber ) {

  int currentYear = year();
  if (currentYear < 2000) currentYear = 2013; // If the clock is not set, then use 2013.

  currentDataFile[0] = (currentYear - 2000 ) / 10 + '0'; // File name format: YYMMDD##.txt
  currentDataFile[1] = currentYear % 10 + '0';
  currentDataFile[2] = month() / 10 + '0';
  currentDataFile[3] = month() % 10 + '0';
  currentDataFile[4] = day() / 10 + '0';
  currentDataFile[5] = day() % 10 + '0';
  currentDataFile[6] = diveNumber / 10;
  currentDataFile[7] = diveNumber % 10;
  //    currentDataFile[8] = '.';
  //    currentDataFile[9] = 't';
  //    currentDataFile[10] = 'x';
  //    currentDataFile[11] = 't';

  // Debug
  Serial.println( currentDataFile );
}

void setupNewDataFile() {

  int diveNumber = 1;

  createFileName( diveNumber );

//  for ( int x = 0; x < nameLength; x++ ) {
//    fileName[x] = currentDataFile[x];
//  }

  while ( SD.exists( currentDataFile ) ) {
    diveNumber++;
```

```
    createFileName( diveNumber );
  }

}

void logData() {
  //  ************* Data logging code *************
//Serial.print("lastDataRecord = ");
//Serial.println(lastDataRecord);
//
//Serial.print("now = ");
//Serial.println(now());

  if ( now() > ( lastDataRecord + recordInterval ) ) {

    lastDataRecord = now();
    // make a string of data to log:
    String dataString = generateRecordDataString();

    // open the file. note that only one file can be open at a time,
    // so you have to close this one before opening another.
    File dataFile = SD.open( currentDataFile, FILE_WRITE );

    // if the file is available, write to it:
    if ( dataFile ) {
      dataFile.println( dataString );
      dataFile.close();
      // print to the serial port:
      //Serial.print("Logging Data: ");
      //Serial.println(dataString);

    }
    else {
      Serial.println( "error opening datalog.txt" );
      showAlert( 0, "Error opening data log file. Check the SD card." );
    }
  }
}

String generateRecordDataString() {

  String returnString = "Nitrogen: ";
  returnString += String( nitrogen );
  return returnString;
}

void drawButtonOptions( String left, String right, boolean showTimeTemp, boolean showBattery ) {

  int leftWidth = ( left.length() * 6 ) - 1;
  int rightWidth = ( right.length() * 6 ) - 1;

  display.fillRect( 0, 55, leftWidth + 2, 9, WHITE );
  display.setTextColor( BLACK );
  display.setCursor( 1, 56 );
  display.setTextSize( 1 );
  display.println( left );

  display.fillRect( 128 - ( rightWidth + 2 ), 55, rightWidth + 2, 9, WHITE );
  display.setTextColor( BLACK );
  display.setCursor( 128 - ( rightWidth + 1 ), 56 );
  display.setTextSize( 1 );
  display.println( right );

  // (leftWidth + rightWidth + 4);
  if ( showTimeTemp ) {
    drawTimeTempBar( leftWidth + 3, 55, 128 - ( ( leftWidth + 3 ) + ( rightWidth + 3 ) ) , showBattery );
  }
  else if ( showBattery ) {
    // TODO: This this appropiately dynamic.
    drawBattery( 54, 56, 20, false ); // if color = false, battey is white, true = black battery.
  }
```

```
}

// ****************************************************************************
//                                Draw Saturation
// ****************************************************************************

void drawSaturation( int value, boolean nitrogen ) {

  float level = ( 47.0 / 255.0 ) * value;

  display.setTextSize( 1 );
  int x = 0;
  display.setTextColor( WHITE );
  if ( nitrogen ) {
    display.setCursor( 0, 47 );

    display.println( "N" );
  }
  else {
    display.setCursor( 122, 47 );

    display.println( "O" );
    x = 122;
  }
  display.fillRect( x, 47 - level, 5, level, WHITE );
}

// ****************************************************************************
//                              Draw Dive Time
// ****************************************************************************

// Good up to 99 min and 99 sec. Font sizes 1-4 are valid
void drawDiveTime(unsigned long seconds, int x, int y, int headingGap, int size, boolean bold, String heading) {

  String timeString = secondsToString( seconds );

  int width = ( ( 6 * 5 ) * size ) - size;
  int height = ( 7 * size ) + headingGap;
  if ( width < 59 ) width = 59;

  display.setCursor( ( ( width - ( heading.length() * 6 ) ) / 2) + x, y );
  display.setTextSize( 1 );
  display.println( heading );

  display.setTextSize( size );
  if ( width > 59 ) {
    display.setCursor( x, y + headingGap );
  }
  else {
    display.setCursor( ( ( width - ( ( 6 * 5 ) * size ) ) / 2 ) + x, y + headingGap);
  }

  display.print( timeString );

  if ( bold ) {
    if ( width > 59 ) {
      display.setCursor( x + 1, y + headingGap );
    }
    else {
      display.setCursor( ( ( width - ( ( 6 * 5 ) * size ) ) / 2 ) + x + 1, y + headingGap);
    }
    display.print( timeString );
  }

#ifdef SHOW_LAYOUT
  display.drawRect( x, y, width, height, WHITE );
#endif
}

// ****************************************************************************
//                                  Draw Clock
```

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

void drawClock( int x, int y, int headingGap, int size, boolean bold ) {

  String timeString = createTimeString( false );

  int width = ( ( 6 * timeString.length()) * size) - size;
  int height = ( 7 * size ) + headingGap;
  if ( width < 29 ) width = 29;

  display.setCursor( ( ( width - 29 ) / 2) + x, y );
  display.setTextSize(1);
  display.println("Time:");

  display.setTextSize( size );
  if ( width > 29 ) {
    display.setCursor( x, y + headingGap );
  }
  else {
    display.setCursor( ( ( width - ( ( 6 * 5 ) * size ) ) / 2 ) + x, y + headingGap );
  }

  display.print( timeString );

  if ( bold ) {
    if ( width > 29 ) {
      display.setCursor( x + 1, y + headingGap );
    }
    else {
      display.setCursor( ( ( width - ( ( 6 * 5 ) * size ) ) / 2 ) + x + 1, y + headingGap );
    }
    display.print( timeString );
  }

#ifdef SHOW_LAYOUT
  display.drawRect( x, y, width, height, WHITE );
#endif

}

String createTimeString(boolean amPm) {

  String timeString;
  int hours = hour();

  if ( time12Hour ) {
    if ( hour() > 12 ) {
      hours -= 12;
    }
    if (hours == 0) {
      hours = 12;
    }
  }

  if ( hours < 10 ) {
    timeString = " ";
    timeString = String( timeString + hours );
  }
  else {
    timeString = String( hours );
  }

  if ( now() > lastColonStateChange ) {
    lastColonStateChange = now();
    clockBlink = !clockBlink;
  }

  if ( clockBlink ) {
    timeString = String( timeString + ':' );
  }
  else {
```

```
    timeString = String( timeString + ' ' );
  }

  int minutes = minute();

  if ( minutes < 10 ) {
    timeString = String( timeString + '0' );
  }


  timeString = String( timeString + minutes );

  if ( time12Hour ) {
    if ( hour() > 11 ) {
      timeString = String( timeString + "PM" );
    }
    else {
      timeString = String( timeString + "AM" );
    }
  }

  timeString.replace( '0', 'O' );

  return timeString;
}

// ******************************************************************************
//                                    Draw Temp
// ******************************************************************************

// Good up to three digits. Sizes 1-4 are valid
void drawTemp( int x, int y, int headingGap, int size, boolean bold ) {

  int headerLength = 5; // "Temp:"
  int headerWidth = ( 6 * headerLength );

  int width = ( ( 6 * 4 ) * size) - size + 1;
  int height = ( 7 * size ) + headingGap;

  if ( width < headerWidth ) width = headerWidth;

  int tempWidth = 10 * size;

  int tempInt = temperature;

  if (tempInt > 9) {
    tempWidth = 16 * size;
  }
  if (tempInt > 99) {
    tempWidth = 21 * size;
  }

  display.setCursor( ( ( width - headerWidth ) / 2) + x + 1, y );
  display.setTextSize( 1 );
  display.println( "Temp:" );

  display.setTextSize( size );
  display.setCursor( ( ( width - tempWidth ) / 2 ) + x + 1, y + headingGap );
  String depthString =  String( tempInt );
  depthString.replace( '0', 'O' );

  if ( ferinheight ) {
    depthString += String( 'F' );
  }
  else {
    depthString += String( 'C' );
  }
  display.print( depthString );

  if ( bold ) {
    display.setCursor( ( ( ( width - tempWidth ) / 2 ) + x) + 2, y + headingGap );
```

```
    display.print( depthString );
  }

#ifdef SHOW_LAYOUT
  display.drawRect( x, y, width, height, WHITE );
#endif
}

// ****************************************************************************
//                                Draw Battery
// ****************************************************************************

void drawBattery( int x, int y, float width, boolean color ) { // if color = false, battey is white, true =
black battery.

  uint16_t colorValue = WHITE;
  if ( color ) colorValue = BLACK;

  if ( batteryLevel < 64 ) {
    if ( batteryBlink ) {
      colorValue = BLACK;
      if ( color ) colorValue = WHITE;
      batteryBlink = false;
    }
    else {
      batteryBlink = true;
    }
  }

  int levelWidth = ( width / 255.0f ) * batteryLevel;
  display.drawRect( x, y, width, 7, colorValue );
  display.fillRect( x + 1, y + 1, levelWidth - 2, 5, colorValue );
  display.drawLine( x + width, y + 2, x + width, y + 4, colorValue );
}

// ****************************************************************************
//                            Draw Time Temp Bar
// ****************************************************************************

// Bar has a fixed height of 9 pixels.
void drawTimeTempBar( int x, int y, int w, boolean showBattery ) {

  // Draw Bar
  display.fillRect( x, y, w, 9, WHITE );

  // DrawTime
  display.setTextColor( BLACK );
  display.setTextSize( 1 );

  int tempInt = temperature;
  display.setCursor( x + 1, y + 1 );
  String timeString = createTimeString( true );
  int timeWidth = ( timeString.length() * 6 ) - 1;
  display.print( timeString );

  // Draw Temp
  String tempString = String( tempInt );
  tempString.replace( '0', 'O' );

  if (ferinheight ) {
    tempString += String( " F" );
  }
  else {
    tempString += String( " C" );
  }

  int tempWidth = ( tempString.length() * 6 ) - 1;

  display.setCursor( x + w - tempWidth - 1, y + 1 );
  display.print(tempString);
```

```
  display.setCursor( ( x + w - 13 ), y - 1 );
  display.write(9);

  // Draw Battery
  if (showBattery) {
    int batteryWidth = w - ( timeWidth + tempWidth + 16);
    drawBattery( x + timeWidth + 8, y + 1, batteryWidth, true);
  }
}

// ******************************************************************************
//                                  Draw Depth
// ******************************************************************************

// Good up to three digits. Sizes 1-4 are valid
void drawDepth(int value, int x, int y, int headingGap, int size, boolean bold, String header) {

  int headerLength = header.length();
  int headerWidth = ( 6 * headerLength);

  int width = ( ( 6 * 3 ) * size) - size + 1;
  int height = (7 * size) + headingGap;

  if ( width < headerWidth ) width = headerWidth;

  int depthWidth = 5 * size;
  if (value > 9) {
    depthWidth = 11 * size;
  }
  if (value > 99) {
    depthWidth = 17 * size;
  }

  display.setCursor( ( ( width - headerWidth ) / 2) + x + 1, y );
  display.setTextSize(1);
  display.println(header);

  display.setTextSize(size);
  display.setCursor( ( ( width - depthWidth ) / 2 ) + x + 1, y + headingGap);
  String depthString =  String(value);
  depthString.replace( '0', 'O' );
  display.print(depthString);

  if ( bold ) {
    display.setCursor( ( ( ( width - depthWidth ) / 2 ) + x) + 2, y + headingGap );
    display.print( depthString );
  }

#ifdef SHOW_LAYOUT
  display.drawRect( x, y, width, height, WHITE );
#endif
}

// ******************************************************************************
//                              Draw Accent Rate
// ******************************************************************************

void calculateAccentRate() {

  int           samples     = accentSamples;
  int           pointer     = depthsPointer;
  unsigned long totalTime   = 0;
  unsigned long lastTime    = 0;
  float         totalDepth  = 0;
  float         lastDepth   = 0;

  for ( int x = 0; x < accentSamples; x++ ) {
    if ( depthMicros[ pointer ] == 0 ) {
      samples--;
    }
    else {
```

```
      if ( lastTime == 0.0 ) {
        lastTime = depthMicros[ pointer ];
      }
      else {
        totalTime += ( depthMicros[ pointer ] - lastTime ) ;
        lastTime = depthMicros[ pointer ];
      }
      if ( lastDepth == 0.0 ) {
        lastDepth = depths[ pointer ];
      }
      else {
        totalDepth += ( lastDepth - depths[ pointer ] );
        lastDepth = depths[ pointer ];
      }
    }
    pointer++;
    if ( pointer > accentSamples ) pointer = 0;
  }

  if (samples == 0) {
    accentRate = 0;
    digitalWrite(alertLEDPin, HIGH);
    return;
  }

  accentRate = (totalDepth / ( totalTime / 1000000.0f ) ) * 60.0;

  int accentRateMultiplier = 1;

  if ( depth > 130 && doubleAccentBelowRec ) accentRateMultiplier = 2;

  // Activate LED and play tone
  if ( accentRate > ( 30.0 * accentRateMultiplier ) ) {
    if ( diveMode ) {
      digitalWrite( alertLEDPin, LOW );
      if ( !toneActive ) {
        if ( playAccentTooFastTone ) {
          tonePlaying = 0;
          toneActive = true;
        }
      }
    }
  }
  else {
    digitalWrite( alertLEDPin, HIGH );
  }
}

void drawAccentRate(int x, int y, int textSize, boolean background, boolean black) {

  uint8_t color = WHITE;
  if ( black ) color = BLACK;

  if ( background ) {
    display.fillRect( x, y, 49, 9, WHITE );
  }

  display.setCursor( x + 1, y + 1 );
  display.setTextColor( color );
  display.setTextSize( textSize );

  if ( accentRate < 10.0 && accentRate > -10.0 ) {
    display.print( " " );
  }
  if ( accentRate < 100.0 && accentRate > -100.0 ) {
    display.print( " " );
  }
  if ( accentRate < 1.0 && accentRate > -1.0 ) {
    display.print( " " );
  }
  else if (accentRate > 0.0 ) {
```

```
    display.print( "+" );
  }

  int accentInt = accentRate;
  String accentString = String( accentInt );
  accentString.replace( '0', 'O' );

  display.print( accentString );
  display.print( "ft/m" );
}

void drawAccentRateArrows(int x, int y) {

  int accentRateMultiplier = 1;
  if ( depth > 130 && doubleAccentBelowRec ) accentRateMultiplier = 2;

  uint16_t colorValue = WHITE;

  if ( accentRate > ( 30 * accentRateMultiplier ) ) {
    if ( accentBlink ) {
      colorValue = BLACK;
      accentBlink = false;
    }
    else {
      accentBlink = true;
    }
    display.drawBitmap( x, y, up_arrow, 8, 4, colorValue );
  }
  if ( accentRate > ( 24 * accentRateMultiplier ) ) {
    display.drawBitmap( x, y + 4, up_arrow, 8, 4, colorValue );
  }
  if ( accentRate > ( 18 * accentRateMultiplier ) ) {
    display.drawBitmap( x, y + 8, up_arrow, 8, 4, colorValue );
  }
  if ( accentRate > ( 12 * accentRateMultiplier ) ) {
    display.drawBitmap( x, y + 12, up_arrow, 8, 4, colorValue );
  }
  if ( accentRate > ( 6 * accentRateMultiplier ) ) {
    display.drawBitmap( x, y + 16, up_arrow, 8, 4, colorValue );
  }
  if ( accentRate > 0) {
    display.drawBitmap( x, y + 20, up_arrow, 8, 4, colorValue );
  }
}

void drawAccentRateBars(int x, int y) {

  int accentRateMultiplier = 1;
  if ( depth > 130 && doubleAccentBelowRec ) accentRateMultiplier = 2;

  uint16_t colorValue = WHITE;

  if ( accentRate > ( 30 * accentRateMultiplier ) ) {
    if ( accentBlink ) {
      colorValue = BLACK;
      accentBlink = false;
    }
    else {
      accentBlink = true;
    }
    display.fillRect( x + 40, y, 7, 20, colorValue );
  }

  if (accentRate > ( 24 * accentRateMultiplier) ) {
    display.fillRect( x + 32, y + 8, 7, 12, colorValue );
  }

  if (accentRate > ( 18 * accentRateMultiplier) ) {
    display.fillRect( x + 24, y + 12, 7, 8, colorValue );
  }
```

```
  if (accentRate > ( 12 * accentRateMultiplier) ) {
    display.fillRect( x + 16, y + 16, 7, 4, colorValue );
  }

  if (accentRate > ( 6 * accentRateMultiplier) ) {
    display.fillRect( x + 8, y + 18, 7, 2, colorValue );
  }

  if (accentRate > 0) {
    display.fillRect( x, y + 19, 7, 1, colorValue );
  }
}

// ****************************************************************************
//                                Utility Methods
// ****************************************************************************

String secondsToString( unsigned long diveSeconds ) {

  int minutes = diveSeconds / 60;
  int seconds = diveSeconds % 60;

  // In case of overflow
  if (minutes > 99) minutes = minutes % 99;

  String timeString;

  if ( minutes < 10 ) {
    timeString = String( '0' );
    timeString = String( timeString + minutes );
  }
  else {
    timeString = String( minutes );
  }

  timeString = String( timeString + ':' );

  if ( seconds < 10 ) {
    timeString = String( timeString + '0' + seconds );
  }
  else {
    timeString = String( timeString + seconds );
  }

  timeString.replace( '0', 'O' );

  return timeString;
}

void showAlert(int time, String message) { // time = 0 will ask the user to dismiss. Time is in seconds

  if ( !alertShowing ) {
    alertShowing = true;
    alertTime = now();
    display.clearDisplay();
    display.display();
    digitalWrite( alertLEDPin, LOW );

    display.setTextColor( WHITE );
    display.setTextSize( 1 );
    display.setCursor( 0, 20 );
    display.print( message );

    if (time == 0) {
      display.fillRect( 115, 55, 13, 9, WHITE );
      display.setTextColor( BLACK );
      display.setCursor( 116, 56 );
      display.setTextSize( 1 );
      display.println( "OK" );
    }
```

```
    clockBlink = false;
    if ( clockBlink ) {
      display.fillRect( 0, 0, 128, 18, BLACK );
      display.setTextColor( WHITE );
      display.setCursor( 35, 2 );
      display.setTextSize( 2 );
      display.println( "ALERT:" );
    }
    else {
      display.fillRect( 0, 0, 128, 18, WHITE );
      display.setTextColor( BLACK );
      display.setCursor( 35, 2 );
      display.setTextSize( 2 );
      display.println( "ALERT:" );
    }
    display.display();
  }

  PITimer1.period( 1 );
  PITimer1.start( blinkAlert );

  if ( time > 0 ) {
    PITimer2.period( time );
    PITimer2.start( killAlert );
  }
}

void killAlert() {
  alertShowing = false;
  digitalWrite( alertLEDPin, HIGH );
  PITimer1.stop();
  PITimer2.stop();
  display.clearDisplay();
  display.display();
}


void blinkAlert() {
  //PITimer1.clear();
  if ( now() > lastColonStateChange ) {
    lastColonStateChange = now();
    clockBlink = !clockBlink;
  }

  if (clockBlink) {
    display.fillRect( 0, 0, 128, 18, BLACK );
    display.setTextColor( WHITE );
    display.setCursor( 35, 2 );
    display.setTextSize( 2 );
    display.println( "ALERT:" );
  }
  else {
    display.fillRect( 0, 0, 128, 18, WHITE );
    display.setTextColor( BLACK );
    display.setCursor( 35, 2 );
    display.setTextSize( 2 );
    display.println( "ALERT:" );
  }
  display.display();
}

// ******************************************************************************
//                           MS5803 Sensor Methods
// ******************************************************************************

// Sends a power on reset command to the sensor. Should be done at powerup and maybe on a periodic basis (needs
to confirm with testing).
void ms5803_reset_sensor() {
  digitalWrite( sensorSelectPin, LOW );
  SPI.setDataMode( SPI_MODE3 );
  SPI.transfer( SENSOR_CMD_RESET );
```

```
  delay( 10 );
  digitalWrite( sensorSelectPin, HIGH );
  delay( 5 );
}

// These sensors have coefficient values stored in ROM that are used to convert the raw temp/pressure data into
degrees and mbars.
// This method reads the coefficient at the index value passed.  Valid values are 0-7. See datasheet for more
info.
unsigned int ms5803_read_coefficient(uint8_t index) {
  unsigned int result = 0;   // result to return

  digitalWrite( sensorSelectPin, LOW );
  SPI.setDataMode( SPI_MODE3 );

  // send the device the coefficient you want to read:
  SPI.transfer( 0xA0 + ( index * 2 ) );

  // send a value of 0 to read the first byte returned:
  result = SPI.transfer( 0x00 );
  result = result << 8;
  result |= SPI.transfer( 0x00 ); // and the second byte

  // take the chip select high to de-select:
  digitalWrite( sensorSelectPin, HIGH );
  //Serial.println (result);
  return( result );
}

// Coefficient at index 7 is a four bit CRC value for verifying the validity of the other coefficients.
// The value returned by this method should match the coefficient at index 7. If not there is something works
with the sensor or the connection.
unsigned char ms5803_crc4(unsigned int n_prom[]) {
  int cnt;
  unsigned int n_rem;
  unsigned int crc_read;
  unsigned char  n_bit;
  n_rem = 0x00;
  crc_read = sensorCoefficients[7];
  sensorCoefficients[7] = ( 0xFF00 & ( sensorCoefficients[7] ) );

  for (cnt = 0; cnt < 16; cnt++)
  { // choose LSB or MSB
    if ( cnt%2 == 1 ) n_rem ^= (unsigned short) ( ( sensorCoefficients[cnt>>1] ) & 0x00FF );
    else n_rem ^= (unsigned short) ( sensorCoefficients[cnt>>1] >> 8 );
    for ( n_bit = 8; n_bit > 0; n_bit-- )
    {
      if ( n_rem & ( 0x8000 ) )
      {
        n_rem = ( n_rem << 1 ) ^ 0x3000;
      }
      else {
        n_rem = ( n_rem << 1 );
      }
    }
  }

  n_rem = ( 0x000F & ( n_rem >> 12 ) );// // final 4-bit reminder is CRC code
  sensorCoefficients[7] = crc_read; // restore the crc_read to its original place
  return ( n_rem ^ 0x00 ); // The calculated CRC should match what the device initally returned.
}

// Use this method to send commands to the sensor.  Pretty much just used to read the pressure and temp data.
unsigned long ms5803_cmd_adc(char cmd) {
  unsigned int result = 0;
  unsigned long returnedData = 0;
  digitalWrite( sensorSelectPin, LOW );

  SPI.transfer( SENSOR_CMD_ADC_CONV + cmd );
  switch ( cmd & 0x0f )
  {
```

```
  case SENSOR_CMD_ADC_256 :
    delay( 1 );
    break;
  case SENSOR_CMD_ADC_512 :
    delay( 3 );
    break;
  case SENSOR_CMD_ADC_1024:
    delay( 4 );
    break;
  case SENSOR_CMD_ADC_2048:
    delay( 6 );
    break;
  case SENSOR_CMD_ADC_4096:
    delay( 10 );
    break;
  }
  digitalWrite( sensorSelectPin, HIGH );
  delay(3);
  digitalWrite( sensorSelectPin, LOW );
  SPI.transfer( SENSOR_CMD_ADC_READ );
  returnedData = SPI.transfer( 0x00 );
  result = 65536 * returnedData;
  returnedData = SPI.transfer( 0x00 );
  result = result + 256 * returnedData;
  returnedData = SPI.transfer( 0x00 );
  result = result + returnedData;
  digitalWrite( sensorSelectPin, HIGH );
  return result;
}

// ******************************************************************************
//                                   Log Book
// ******************************************************************************

void drawLogBook() {

  display.fillRect(0, 0, 128, 9, WHITE);

  display.setTextColor(BLACK);
  display.setTextSize(1);
  display.setCursor(40, 1);
  display.print("Log Book");

  for (int i = 0; i < 5; i++ ) {
    drawLogEntry( 10, i, currentTime, i );
  }

  drawButtonOptions("^^", ">>", true, true);
}

void drawLogEntry(int y, int listPosition, unsigned long date, int diveNumber) { // Parameters: y is the very
top of the list, listPosition is where the line will be drawn relative to y, date is the date/time of the entry
and diveNumber is the # for that date.

  String text = String( "Mar 28, 2013 Dive #" ); // 20 characters
  text += String( diveNumber + 1 );
  display.setTextColor( WHITE );
  display.setTextSize( 1 );
  display.setCursor( 0, y + ( 9 * listPosition ) );
  display.print( text );


}



// ******************************************************************************
//                                   Settings
// ******************************************************************************

void drawSettings() {
```

```
  display.fillRect(0, 0, 128, 9, WHITE);

  display.setTextColor(BLACK);
  display.setTextSize(1);
  display.setCursor(40, 1);
  display.print("Settings");

  display.setTextColor(WHITE);
  display.setCursor(0, 10);
  display.print("Under construction");

  drawButtonOptions("MENU", ">>", true, false);
}

// ****************************************************************************
//                               EEPROM Methods
// ****************************************************************************

void readPrefereces() {

  int value           = EEPROM.read( 0 );
  int value2          = EEPROM.read( 1 );

  // if the values stored in locations 0 and 1 are not 170, then assume this is a first run.
  // Clear the EEPROM and store the 170s to indicate first run has happened.
  if (value != 170 || value2 != 170) {  // First run...

    // Wipe the EEPROM
    for (int i = 0; i < 2048; i++ ) { // The Teensy has 2K of EEPROM
      EEPROM.write( i, 0 );
    }

    EEPROM.write( 0, 170 );
    EEPROM.write( 1, 170 );

    // Write the default preference.
    writePreferences();
  }
  else { // Not a first run...
    // read preferences
    time12Hour            = EEPROM.read( 2 ); // true = 12 hour time, false = 24 hour time.
    clockBlink            = EEPROM.read( 3 ); // true = blink the colon, false = don't blink the colon.
    ferinheight           = EEPROM.read( 4 ); // true = Ferinheight, false = Celsius.
    meters                = EEPROM.read( 5 ); // true = meters, false = feet.
    doubleAccentBelowRec  = EEPROM.read( 6 ); // True = 60ft/min accent rate below recreational limits (130ft).
30ft/min otherwise.
    recordInterval        = EEPROM.read( 7 ); // This is the inteval in which it will record data to the SD
card in minutes.
    profileInterval       = EEPROM.read( 8 ); // The interval that dive profile depths are recorded to EEPROM
(for the graphing funtion).
    playAccentTooFastTone = EEPROM.read( 9 ); // Determines if the accent rate tone is played.
  }
}

void writePreferences() {
  EEPROM.write( 2, time12Hour );
  EEPROM.write( 3, clockBlink );
  EEPROM.write( 4, ferinheight );
  EEPROM.write( 5, meters );
  EEPROM.write( 6, doubleAccentBelowRec );
  EEPROM.write( 7, recordInterval );
  EEPROM.write( 8, profileInterval );
  EEPROM.write( 9, playAccentTooFastTone );
}

// ****************************************************************************
//                            Dive Profile Methods
// ****************************************************************************

// TODO: Fix this so that it saves an integer (uint_16) not a byte (uint_8).  This will allow it to exceed 255
feet for those crazy enough to go that deep.
```

```
void recordDiveProfileValue() {

  if ( now() > ( lastProfileRecord + profileInterval ) ) {

    lastProfileRecord = now();
    byte profileDepth = depth;
    int pointer = 2047 - profileSize + profilePointer; // Put the profile at the end of the EEPROM.
    EEPROM.write( pointer, profileDepth );
    profilePointer++;
    profileSamples++;
    if ( profilePointer > profileSize ) profilePointer = 0;
    if ( profileSamples > profileSize ) profileSamples = profileSize;
  }
}

void clearDiveProfile() {
  byte profileDepth = 0;
  int pointer = 0;
  for (int x = 0; x < profileSize; x++) {
    pointer = ( 2047 - profileSize ) + x;
    EEPROM.write( x, profileDepth );
  }
  profilePointer = 0;
  profileSamples = 0;
}

// Graph has a fixed size of 100 pixels wide and 50 pixles high.
void drawDiveProfileGraph(int x, int y) {

  int maxRange = ( maxDepth / 10 ) + 1;
  maxRange = maxRange * 10;
  float multiplier = 50.0f / maxRange;
  float yVal = 0.0f;
  int yInt = 0;
  int pointer = ( 2047 - profileSize ) + profilePointer - 1;
  int storedValue = 0;
  float storedFloat = 0.0f;

  for ( int c = 0; c < profileSamples; c++ ) {
    storedValue = EEPROM.read( pointer );
    storedFloat = storedValue;
    yVal = ( storedFloat * multiplier ) * 1.0f;
    yInt = yVal;
    display.drawPixel( ( x + profileSize ) - c, yInt, WHITE);
    pointer--;
    if ( pointer < (2047 - profileSize) ) pointer = 2047;
  }

  display.setTextColor(WHITE);
  display.setTextSize(1);
  display.setCursor (x + 2, y + 43);
  display.print(maxRange);

  display.drawLine(x, y + 51, x + profileSize, y + 51, WHITE);
  display.drawLine(x, y, x, y + 51, WHITE);

}

void cycleTone() {

  noTone( speakerPin );

  if ( !toneActive ) return;
  int numberOfTones = 0;
  if ( tonePlaying == 0 ) numberOfTones = accent_tone[ 0 ];
  if ( tonePlaying == 1 ) numberOfTones = trill_tone[ 0 ];

  if ( toneCounter > numberOfTones - 1 ) {
    toneCounter = 0;
    toneActive = false;
    return;
```

```
  }

  toneCounter++;
  if ( tonePlaying == 0 ) tone( speakerPin, accent_tone[ toneCounter ] );
  if ( tonePlaying == 1 ) tone( speakerPin, trill_tone[ toneCounter ] );

}

// ***************************************************************************
//                            Safety Stop Methods
// ***************************************************************************

void drawSafetyStopScreen() {

  uint16_t textColorValue = WHITE;
  uint16_t barColorValue = BLACK;

  if ( now() > safetyStopBlink ) {
    safetyStopBlink = now();
    safetyStopBlinkState = !safetyStopBlinkState;
  }

  if ( safetyStopBlinkState ) {
    textColorValue = BLACK;
    barColorValue = WHITE;
  }

  display.fillRect( 0, 0, 128, 9, barColorValue );
  display.setTextColor( textColorValue );
  display.setCursor( 31, 1 );
  display.setTextSize( 1 );
  display.println( "SAFETY STOP" );

  display.setTextColor( WHITE );
  drawDiveTime( ( safetyStopTime - now() ), 60, 10, 9, 2, false, "Stop Time:" ); // Parameters: Time in
seconds, x, y, size, bold
  drawDepth( depth, 6, 16, 9, 3, true, "Depth:" ); // Parameters: depth, x, y, size, bold, header string
  drawAccentRateBars( 71, 34 );
  drawSaturation( nitrogen, true );
  drawSaturation( oxygen, false );
  drawButtonOptions( "^^", ">>", true, true );

}

// ***************************************************************************
//                             Methods
// ***************************************************************************
```

Reference	URL
Tutorial: Using an MS5803 pressure sensor with Arduino	https://thecavepearlproject.org/2014/03/27/adding-a-ms5803-02-high-resolution-pressure-sensor/
Fundamentals of Pressure Sensor Technology	https://www.fierceelectronics.com/components/fundamentals-pressure-sensor-technology
SparkFun Pressure Sensor Breakout	https://www.sparkfun.com/products/12909

DIY-Dive-Computer/DIY-Dive-Computer (CODE)	https://github.com/DIY-Dive-Computer/DIY-Dive-Computer/blob/v0.1/Dive_Computer_Code/Dive_Computer_Code.ino