ME102B Fall 2020

ESP32 VU Meter by Jonathan de Laine

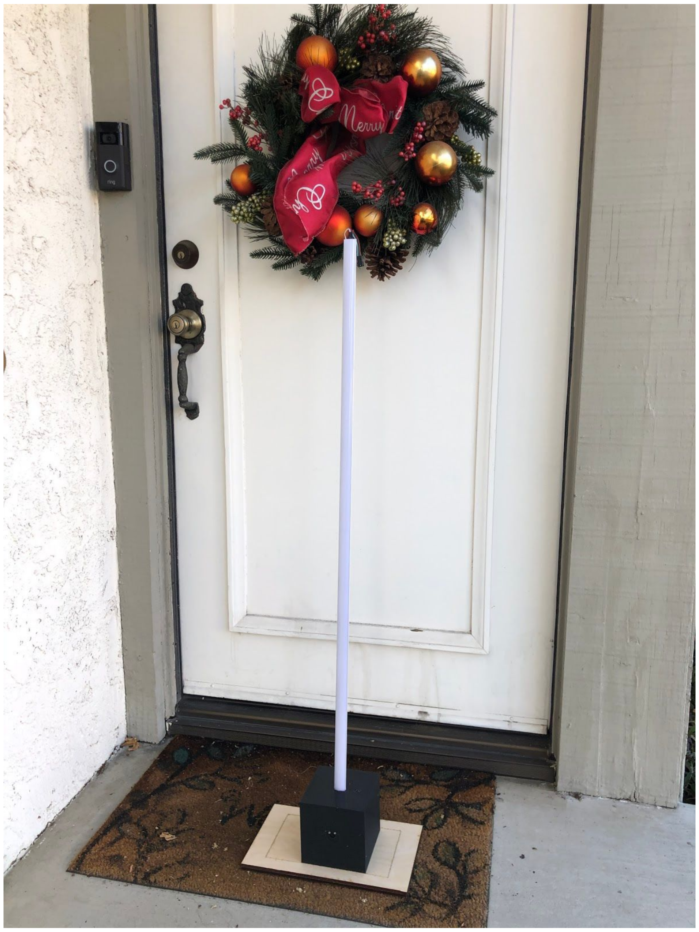

Figure 1. Festive Photo of Completed Project Resting on Wood Baseboard.

Description of the product:
I love electronic music. Whether studying, driving, walking, or on a run, I spend most of my time listening to electronic/dance music and I enjoy going to music festivals like Electric Daisy Carnival (EDC) in my free time. A large element of electronic music festivals revolves around performance stages with complex lighting using LEDs, lasers, and more. As I am unable to attend any shows this year due to COVID restrictions, for my project I wanted to create a device that could recapture some of the atmosphere I was missing out on. By using a microphone to sample audio, I wanted to scale both the number of LEDs illuminated and the color displayed. I designed this system to be placed in the corner of a room, powered off a single wall outlet, and dynamically responsive to music (or other sounds!) played in the room.

Video Link: https://bit.ly/37CWPHy

Electromechanical Details:

3D Housing: An LED strip, 5V fan, sensor and power plug are designed to interface with this 3D printed housing. A microphone faces out from the front side of the container, LED strip and aluminum channel insert into the top of the box and power and the cooling fan face out from the rear of the box.

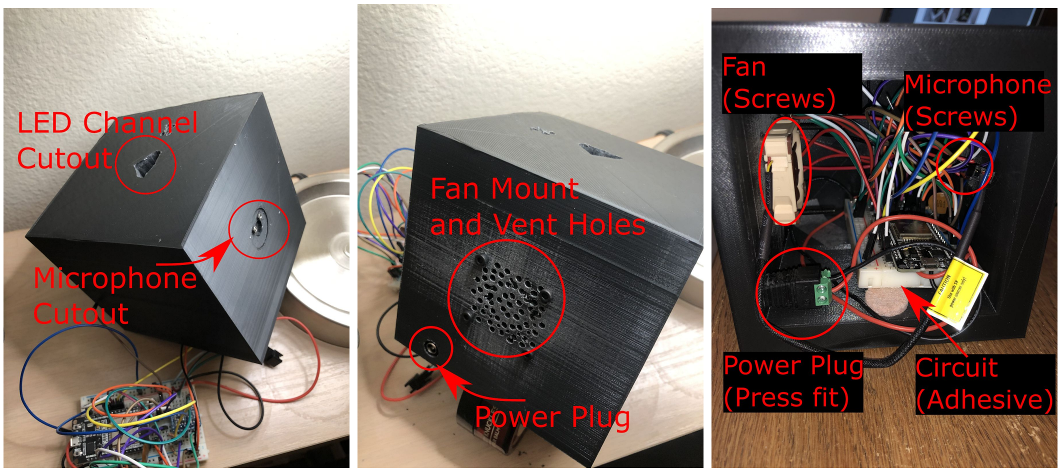

Figure 2. Images depicting how components integrate into the housing. Image furthest to the right indicates specific methods of securing components to the housing.

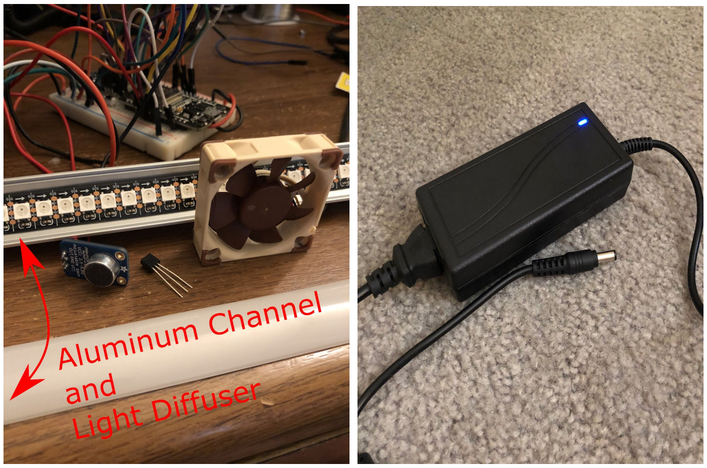

Figure 3. Circuit at a glance including the final mechanical component: an aluminum LED channel and light diffuser. Right image shows 40W power supplied used to power LEDs

Circuit:

The main component of this project revolves around supplying data to the WS2812B LED strip by sampling from an Adafruit MAX4466 Microphone. Additionally, as the circuit demands large amounts of current, dedicated power, a temperature sensor, and a cooling fan were included.

(1) WS218B: The heart of the circuit is this strip of LEDs. The chain I used contains 144 parallel, individual LED modules consisting of 3 LEDs (R,G,B) and a tiny microcontroller per module. This strip requires 5V power, ground, and logic PWM input to individually address each module. This LED strip was chosen as it is compliant with the Arduino FastLED library which allowed for speedy implementation. 5V logic was supplied and controlled using a BSS138 MOSFET found in Adafruit's BSS138 Logic Converter, chosen due to its compact form and integrated 10K Ohm current limiting resistor.
LEDs: https://amzn.to/3pWtRe1 (~$20) | *Logic Converter*: https://bit.ly/3nSmVMU (~$6)

(2) MAX4466 Microphone: This microphone and amplifier is the main method to detect audio. Adafruit claims this microphone to have excellent supply noise rejection so I forewent a capacitor. Calibration of the microphone can be seen in **Figure X.** Using MATLAB, I determined the 'baseline' volume for my room to detrend the output to make further processing easier.
Microphone: https://bit.ly/3fzm4xM (~$7)

(3) ALITOVE 5V 8A Power Supply: According to the WS218B datasheet, each LED in a module draws 20mA under max brightness for a total of 60mA per module. As there are 144 LEDs, this totals 8640mA under max load. As I am using a 8A power supply, the maximum brightness is limited through code to keep within safe limits. Additionally, I am using heavier, 18AWG wire for current-bearing wires.
Power Supply: https://amzn.to/39iD6zo (~$16)

(4) Analog Temperature Sensor (TMP36): This solid-state sensor detects the difference in voltage across a diode as temperature increases and outputs voltage proportional to temperature. Upon implementation, it was found this sensor does not cooperate well with the ESP32 ADC. I used a 100nF '104' Capacitor to ground on the VCC input and a pulldown resistor on the output to help eliminate noise. This had an effect of reducing 2C swings down to .5C. An additional offset of

10C was necessary after conversion to match temperature shown on a desk thermometer. I believe this is due to the ESP32's ADC Nonlinearity.
TMP36: https://bit.ly/2V5jaaA (~$1.50)

(5) Noctua NF-A4x10 5V Fan: I am familiar with this company through my PC building hobby. This fan has excellent noise levels and does not interfere with the microphone. This fan has an integrated commutation controller thus I only need to provide power and ground. I am using an STP16NF06L N-Channel MOSFET to command the fan with a control signal. Prior to the gate, I am using a 2.2K Ohm current limiting resistor and a 4.7K Ohm resistor to ground to act as a voltage divider.
Fan: https://amzn.to/3fCNqmB (~$14)

As I wanted the product to run off a single plug, I also purchased an ESP32 Devkit1 with an integrated Vin pin. Checking the datasheet, this pin contains an AMS1117 voltage regulator that accepts up to 15 volts. In addition, a switch is used to manually cycle between VU modes.

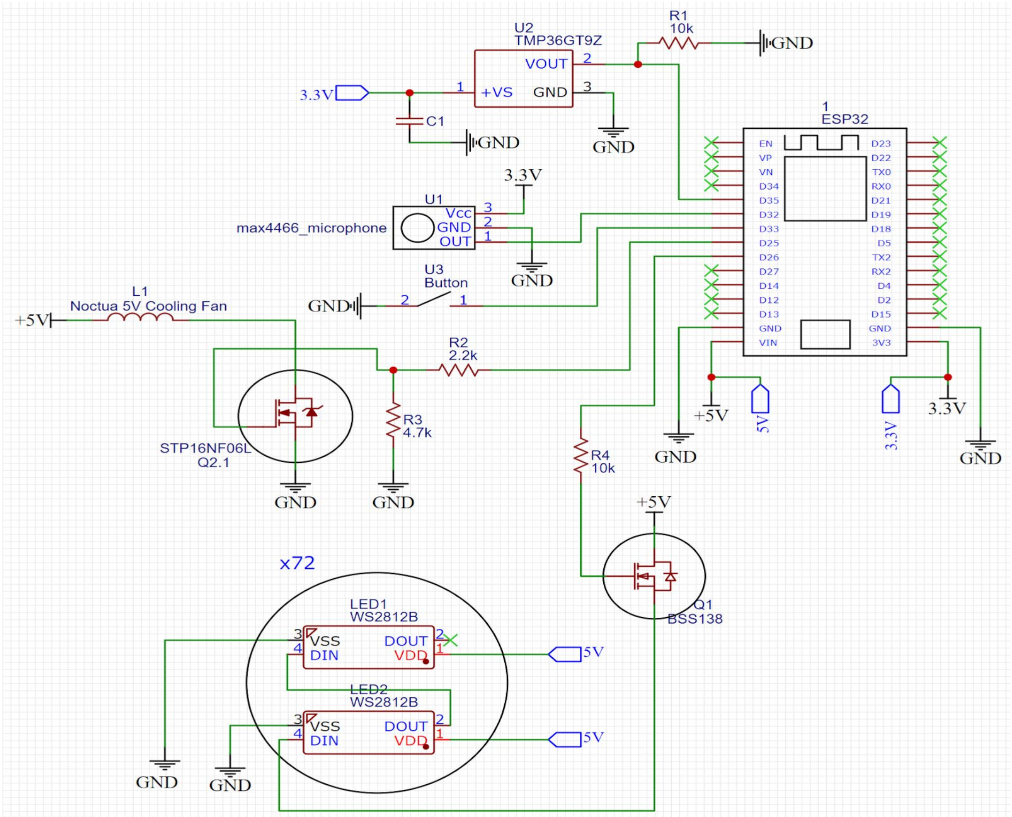

Figure 4. Complete circuit diagram. Designed using EasyEDA software.

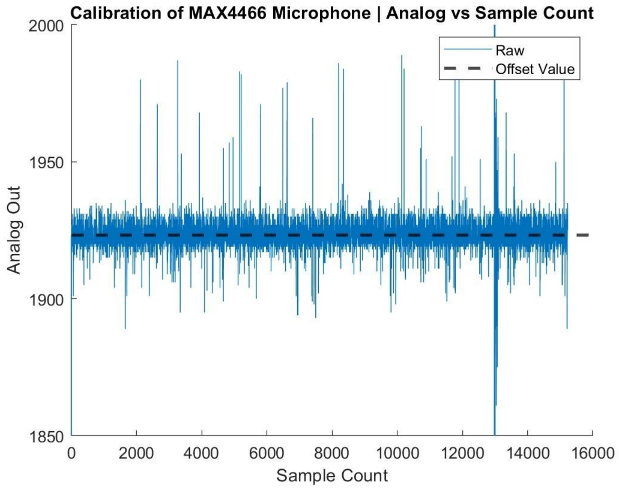

Figure 5. Calibration of MAX4466 microphone. This figure shows raw recording used to find “no volume” offset. Offset = 1922

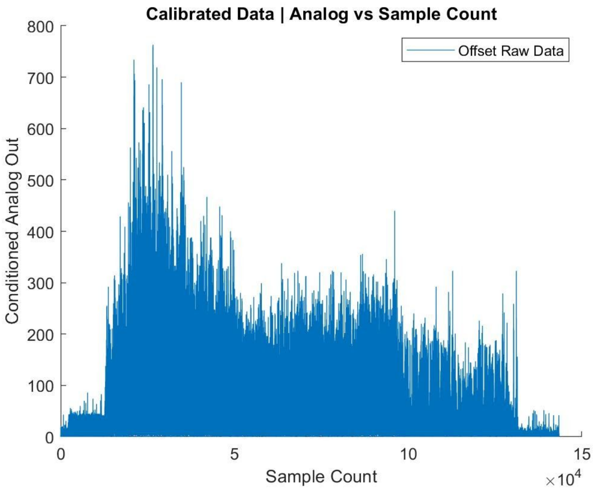

Figure 6. Calibrated data from MAX4466 microphone. This figure shows the absolute value of outputs with offset removed while speaking into the microphone.

Finite State Machine:

The VU meter can be cycled through various display patterns by pressing the button integrated in the circuit. Currently the microcontroller is programmed to have 1 dynamic state, 2 passive, and a standby state. Additionally, the FSD shows an additional state for each, corresponding to the fan being in either “ON” or “OFF” state depending on temperature inside the housing. The fan runs on a simple bang-bang controller. As the temperature is sampled and the fan toggled once every 30 seconds, I did not feel the need to include hysteresis.

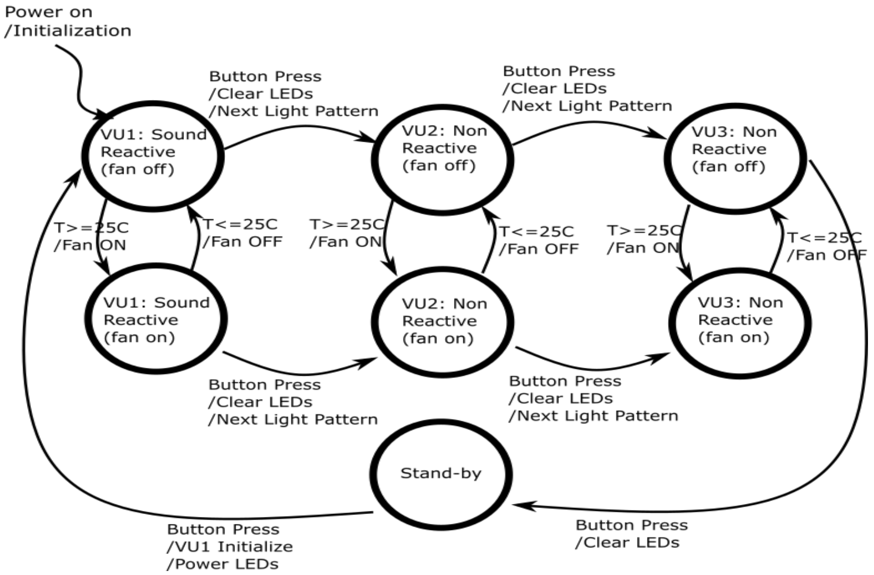

Figure 7. Finite state diagram showing 3 programmed light shows, although more can be easily added. VU1 is the sound reactive VU Mode demonstrated in the final portion of my video linked above.

Full code can be found in Appendix 1. Note that the sound reactive functionality also requires the use of a header file, included in Appendix 2.

APPENDIX 1: ARDUINO CODE (MAIN CODE)

ME102B VuMeter - Final Project

Jonathan de Laine

```
/*Jonathan de Laine
   ME102B Final Project
   VU Meter
      Features:
          -Three States + Stanby
          -Sound Reactive Mode using MAX4466 Microphone
          -Brushless DC fan control
          -TMP36 Temperature Sensor Implementation

          -Timer Interrupt Temperature Check/Fan Toggle
          -Button Interrupt State Switching with Debounce

    Copyright © Jonathan de Laine, 2020
*/

#include "FastLED.h"
#include "averagesContainer.h"

//Pin Definitions
#define MIC 32
#define BUTTON 33
#define TMP 35
#define FAN 25
#define LED_DATA 26

//LED Strip Specific Params (FastLED)
#define NUM_LEDS 144
#define LED_TYPE WS2812B
#define COLOR_ORDER GRB //For WS2812B strip
uint8_t max_bright = 200; //255 is max


//Class object defs
#define NUMSAMPLES 20
#define NUMLONGSAMPLES 250
#define BUFFER_DEV 400
#define BUFFER_SIZE 3
//end class object defs

//Microphone Calibration Defines
#define s_MIC_HIGH 600
```

```
#define s_MIC_LOW 0
int MIC_HIGH; //Non-static version (hopefully no calibration)
int MIC_LOW; //(Found through header file functions
//

//LED Color Vars
float gHue = 0;
float gBright = 250;
int hueOffset = 120;
float fadeSc = 1.3;
float hueInc = 0.7;

// end LED

//Mic
int raw;
int condraw;
uint16_t minLVL;
uint16_t maxLVL;
int micoffset = 1923; //From MATLAB processing
//end mic

//Button
volatile bool buttonflag; //Volatile flag -> change in and out of interrupt
const int debounce = 200; //milliseconds
int currentflagtime, lastflagtime;
//end button

//State Variables
int vumode = 1;
//end state

//Timer Variables
float temp = 0; //only for initialization/debug
float in;
const int templimit = 25; //Celsius ~= 77F
const int tempoffset = 10; //TMP sensor not cooperating
volatile bool tempcheckFlag;
bool fanFlag;
//const int timer_speed = 30000000; //30seconds
const int timer_speed = 1000000; //1 second
hw_timer_t * timer1 = NULL;
portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED;
//end timer

//Class Objects for Storing Data *requires intialization in setup
struct averagesContainer *samples;
struct averagesContainer *longsamples;
struct averagesContainer *buffersamples;
```

```
//end class objects

CRGB leds[NUM_LEDS];

void Press() {
  currentflagtime = millis();
  if (currentflagtime - lastflagtime > debounce) {
    buttonflag = true;
    lastflagtime = currentflagtime;
  }
}
void IRAM_ATTR tempCheck() {
  portENTER_CRITICAL_ISR(&timerMux);
  tempcheckFlag = true;
  portEXIT_CRITICAL_ISR(&timerMux);
}

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  delay(10);
  Serial.println('\n');
  pinMode(MIC, INPUT);
  pinMode(BUTTON, INPUT_PULLUP);
  pinMode(TMP, INPUT);
  pinMode(FAN, OUTPUT);
  attachInterrupt(BUTTON, Press, FALLING);

  //Initialize class objects
  samples = new averagesContainer(NUMSAMPLES);
  longsamples = new averagesContainer(NUMLONGSAMPLES);
  buffersamples = new averagesContainer(BUFFER_SIZE);

  //Use while loops to fill sample containers w/ placeholders | Note: setSamples returns true until full
  while (buffersamples->setSample(250) == true) {}
  while (longsamples->setSample(200) == true) {}

  //Define LED Setup
  FastLED.addLeds<LED_TYPE, LED_DATA, COLOR_ORDER>(leds, NUM_LEDS);

  //Timer setup
  timer1 = timerBegin(0, 80, true);
  timerAttachInterrupt(timer1, &tempCheck, true);
  timerAlarmWrite(timer1, timer_speed, true); //Edge trigger
  timerAlarmEnable(timer1); //Begin timer


}

```

```
void loop() {
  // put your main code here, to run repeatedly:
  raw = analogRead(MIC);
  condraw = abs(raw - micoffset); //Centers reading around zero, removes negative range
  //Do stuff with Raw to change colors, etc. Map?

  if (buttonflag == true) {
    vumode += 1;
    buttonflag = false;
  }
  if (tempcheckFlag == true) {
    in = analogRead(TMP);
    temp = in * 3.3 / 4096.0 * 100 - 50 + tempoffset;
    if (temp >= templimit) {
      fanFlag = true;
    }
    else {
      fanFlag = false;
    }
    portENTER_CRITICAL(&timerMux);
    tempcheckFlag = false;
    portEXIT_CRITICAL(&timerMux);
  }
  switch (fanFlag) {
    case true:
      digitalWrite(FAN, HIGH);
      break;
    case false:
      digitalWrite(FAN, LOW);
      break;
    default:
      Serial.println("Something goofy going on");
      break;
  }

  //States
  switch (vumode) {
    case 1:
      Vu1();
      break;
    case 2:
      Vu2();
      break;
    case 3:
      Vu3();
      break;
    case 4:
      Standby();
      break;
    default: //Catch case / used to reset Vumode to 1
      vumode = 1;
      Serial.println("Change");
      break;
  }
}
void Vu1() { //Sound Reactive Vu Meter

  //  Serial.println(condraw);
  //  Serial.println(temp);

  //Attempt to use static calibration instead of dynamic
  int bufferval = buffersamples->findAverage();
  if (!(abs(condraw - bufferval) > BUFFER_DEV)) {
    buffersamples->setSample(condraw);
  }
  //Scale conditioned signal to Log Scale with .4 scalar
  condraw = fscale(s_MIC_LOW, s_MIC_HIGH, s_MIC_LOW, s_MIC_HIGH, condraw, 0.4);

  if (samples->setSample(condraw))
    return; //continue adding until full

  uint16_t longsamplesAvg = longsamples->findAverage();
  uint16_t inputVal = samples->findAverage();

  longsamples->setSample(inputVal);

  //Change hue of colors based on long term averages
  int diff = (inputVal - longsamplesAvg);
  if (diff > 5)
  {
    if (gHue < 235)
    {
      gHue += hueInc;
    }
  }
  else if (diff < -5)
  {
    if (gHue > 2)
    {
      gHue -= hueInc;
    }
  }

  int height = fscale(s_MIC_LOW, s_MIC_HIGH, 0.0, (float)NUM_LEDS, (float)inputVal, 0);

  for (int i = 0; i < NUM_LEDS; i++)
  {
    if (i < height)
    {
      leds[i] = CHSV(gHue + hueOffset + (i * 2), 255, max_bright);
    }
    else
    {
      leds[i] = CRGB(leds[i].r / fadeSc, leds[i].g / fadeSc, leds[i].b / fadeSc);
    }
  }
  delay(5);
  FastLED.show();
  //Serial.println(height);
  //Serial.println(raw);
  //Serial.println(temp);

}
void Vu2 () { //Non Sound Reactive - static color climb+descent
  for (int i = 0; i < NUM_LEDS; i++) {
    leds[i] = CRGB::Magenta;
    FastLED.show();
    delay(10);
  }
  for (int i = NUM_LEDS; i >= 0; i--) {
    leds[i] = CRGB::Black;
    FastLED.show();
    delay(10);
  }
}
void Vu3 () { //Non reactive - rainbow color wheel
  uint8_t initialHue = 0; //starting color
  const uint8_t deltaHue = 2; //Change in color from 1 led to another
  const uint8_t initialHueIncrement = 4; //This increments the initial color each iteration of loop -> scrolls rainbow

  fill_rainbow(leds, NUM_LEDS, initialHue += initialHueIncrement, deltaHue);
  FastLED.show();
}
void Standby() { //No color.
  FastLED.clear();
  FastLED.show();
}
//Code implemented courtesy of Cine-lights via GitHub. Filters Microphone Raw data to fit acceptable ranges
float fscale(float originalMin, float originalMax, float newBegin, float newEnd, float inputValue, float curve)
{
  float OriginalRange = 0;
  float NewRange = 0;
  float zeroRefCurVal = 0;
  float normalizedCurVal = 0;
  float rangedValue = 0;
  bool invFlag = 0; //Invert Flag

```

```
if (curve > 10)
  curve = 10;
if (curve < -10)
  curve = -10;

curve = curve * (-.1);
curve = pow(10, curve);

if (inputValue < originalMin)
{
  inputValue = originalMin;
}
if (inputValue > originalMax)
{
  inputValue = originalMax;
}

//Zero reference the values
OriginalRange = originalMax - originalMin;
if (newEnd > newBegin)
{
  NewRange = newEnd - newBegin;
}
else
{
  NewRange = newBegin - newEnd;
  invFlag = 1; //Invert Flag
}

zeroRefCurVal = inputValue - originalMin;
normalizedCurVal = zeroRefCurVal / OriginalRange; //Normalize to 0-1 float

//Check for originalMin > orignalMax
if (originalMin > originalMax)
{
  return 0;
}

if (invFlag == 0)
{
  rangedValue = (pow(normalizedCurVal, curve) * NewRange) + newBegin;
}
else //invert range
{
  rangedValue = newBegin - (pow(normalizedCurVal, curve) * NewRange);
}

return rangedValue;
```

```

}
```

APPENDIX 2: Header File

averagesContainer Header File

Jonathan de Laine

```
struct averagesContainer{
  uint16_t *samples; //create pointer to samples for memory consideration
  uint16_t container_size;
  uint8_t counter;
  uint16_t minLvl;
  uint16_t maxLvl;

    //Constructor (Class Object Initializer)
    averagesContainer(uint16_t datapoints){
      counter = 0;
      container_size = datapoints;
      samples = (uint16_t*) malloc(sizeof(uint16_t)*container_size); //Creates
    }

    //Define class functions for each object -> allows to generate averages, etc.
    bool setSample(uint16_t value){
      if(counter < container_size) { //if we haven't hit max size of container
        samples[counter++] = value; //Save the value we call function with
        return true; //true means allow to continue adding
      }
      else { //Container is full. Reset counter to allow roll-over/rewriting of container
        counter = 0;
        return false; //Flag that container is full
      }
    }

    int findAverage(){
      int sum = 0;
      for (int i = 0; i<container_size; i++){ //Loop through container, add values
        sum += samples[i];
      }
      return (int)(sum/container_size);
    }
    void MinMax(){
      minLvl = maxLvl = samples[0];
      for(int i = 1; i<container_size; i++){
        if(samples[i]<minLvl) minLvl = samples[i];
        else if(samples[i]>maxLvl) maxLvl = samples[i];
      }
    }
    uint16_t getMin(){
      return minLvl;
    }
    uint16_t getMax(){
      return maxLvl;
    }
};
```