```
#include <Adafruit_NeoPixel.h>
#include <FastLED.h>
#include <math.h>
#include <SoftwareSerial.h>
#define N_PIXELS  59// Number of pixels in strand (max 76 arduino uno)
#define N_PIXELS_HALF (N_PIXELS/2)
#define MIC_PIN   A5  // Microphone is attached to this analog pin
#define LED_PIN    6  // NeoPixel LED strand is connected to this pin
#define SAMPLE_WINDOW   10  // Sample window for average level
#define PEAK_HANG 24 //Time of pause before peak dot falls
#define PEAK_FALL 20 //Rate of falling peak dot
#define PEAK_FALL2 8 //Rate of falling peak dot
#define INPUT_FLOOR 10 //Lower range of analogRead input
#define INPUT_CEILING 300 //Max range of analogRead input, the lower the
value the more sensitive (1023 = max)300 (150)
#define DC_OFFSET  0  // DC offset in mic signal - if unusure, leave 0
#define NOISE     10  // Noise/hum/interference in mic signal
#define SAMPLES   60  // Length of buffer for dynamic level adjustment
#define TOP       (N_PIXELS + 2) // Allow dot to go slightly off scale
#define SPEED .20       // Amount to increment RGB color by each cycle
#define TOP2      (N_PIXELS + 1) // Allow dot to go slightly off scale
#define LAST_PIXEL_OFFSET N_PIXELS-1
#define PEAK_FALL_MILLIS 10  // Rate of peak falling dot
#define POT_PIN    4
#define BG 0
#define LAST_PIXEL_OFFSET N_PIXELS-1
#if FASTLED_VERSION < 3001000
#error "Requires FastLED 3.1 or later; check github for latest code."
#endif
#define BRIGHTNESS  255
#define LED_TYPE    WS2812B     // Only use the LED_PIN for WS2812's
#define COLOR_ORDER GRB
#define COLOR_MIN           0
#define COLOR_MAX         255
#define DRAW_MAX          100
#define SEGMENTS            4  // Number of segments to carve amplitude
bar into
#define COLOR_WAIT_CYCLES  10  // Loop cycles to wait between advancing
pixel origin
#define qsubd(x, b)  ((x>b)?b:0)
#define qsuba(x, b)  ((x>b)?x-
b:0)                                          // Analog Unsigned
subtraction macro. if result <0, then => 0.
#define ARRAY_SIZE(A) (sizeof(A) / sizeof((A)[0]))
#define INTERVALTIME 120 // intervaltijd

struct CRGB leds[N_PIXELS];
```

```
    Adafruit_NeoPixel strip = Adafruit_NeoPixel(N_PIXELS, LED_PIN, NEO_GRB +
NEO_KHZ800);

    static uint16_t dist;         // A random number for noise generator.
    uint16_t scale = 30;          // Wouldn't recommend changing this on the
fly, or the animation will be really blocky.
    uint8_t maxChanges = 48;      // Value for blending between palettes.

    CRGBPalette16 currentPalette(OceanColors_p);
    CRGBPalette16 targetPalette(CloudColors_p);


//new ripple vu
uint8_t timeval =
20;                                                    // Currently
'delay' value. No, I don't use delays, I use EVERY_N_MILLIS_I instead.
uint16_t loops =
0;                                                     // Our loops
per second counter.
bool     samplepeak =
0;                                                  // This sample is
well above the average, and is a 'peak'.
uint16_t oldsample =
0;                                                   // Previous sample
is used for peak detection and for 'on the fly' values.
bool thisdir = 0;
//new ripple vu

// Modes
enum
{
} MODE;
bool reverse = true;
int BRIGHTNESS_MAX = 80;
int brightness = 20;


byte
//  peak      = 0,      // Used for falling dot
//  dotCount  = 0,      // Frame counter for delaying dot-falling speed
  volCount  = 0;      // Frame counter for storing past volume data
int
  reading,
  vol[SAMPLES],       // Collection of prior volume samples
  lvl       = 10,      // Current "dampened" audio level
  minLvlAvg = 0,      // For dynamic adjustment of graph low & high
  maxLvlAvg = 512;
float
```

```
  greenOffset = 30,
  blueOffset = 150;
// cycle variables

int CYCLE_MIN_MILLIS = 2;
int CYCLE_MAX_MILLIS = 1000;
int cycleMillis = 20;
bool paused = false;
long lastTime = 0;
bool boring = true;
bool gReverseDirection = false;
int           myhue =   0;
//vu ripple
uint8_t colour;
uint8_t myfade = 255;                                    // Starting
brightness.
#define maxsteps 16                                      // Case
statement wouldn't allow a variable.
int peakspersec = 0;
int peakcount = 0;
uint8_t bgcol = 0;
int thisdelay = 20;
uint8_t max_bright = 255;


  unsigned int sample;

//Samples
#define NSAMPLES 64
unsigned int samplearray[NSAMPLES];
unsigned long samplesum = 0;
unsigned int sampleavg = 0;
int samplecount = 0;
//unsigned int sample = 0;
unsigned long oldtime = 0;
unsigned long newtime = 0;

//Ripple variables
int color;
int center = 0;
int step = -1;
int maxSteps = 16;
float fadeRate = 0.80;
int diff;

//vu 8 variables
int
  origin = 0,
  color_wait_count = 0,
```

```
  scroll_color = COLOR_MIN,
  last_intensity = 0,
  intensity_max = 0,
  origin_at_flip = 0;
uint32_t
    draw[DRAW_MAX];
boolean
  growing = false,
  fall_from_left = true;



//background color
uint32_t currentBg = random(256);
uint32_t nextBg = currentBg;
TBlendType    currentBlending;

const int buttonPin = 0;     // the number of the pushbutton pin

 //Variables will change:
int buttonPushCounter = 0;   // counter for the number of button presses
int buttonState = 0;         // current state of the button
int lastButtonState = 0;


    byte peak = 16;      // Peak level of column; used for falling dots
//    unsigned int sample;

    byte dotCount = 0;  //Frame counter for peak dot
    byte dotHangCount = 0; //Frame counter for holding peak dot

void setup() {

     //analogReference(EXTERNAL);

    pinMode(buttonPin, INPUT);
  //initialize the buttonPin as output
   digitalWrite(buttonPin, HIGH);

      // Serial.begin(9600);
      strip.begin();
      strip.show(); // all pixels to 'off'

      //Serial.begin(57600);
      delay(3000);

  LEDS.addLeds<LED_TYPE,LED_PIN,COLOR_ORDER>(leds,N_PIXELS).setCorrection(Typi
calLEDStrip);
  LEDS.setBrightness(BRIGHTNESS);
```

```
  dist = random16(12345);          // A semi-random number for our noise
generator

 }


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;
  boolean invFlag = 0;


  // condition curve parameter
  // limit range

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

  curve = (curve * -.1) ; // - invert and scale - this seems more intuitive -
postive numbers give more weight to high end on output
  curve = pow(10, curve); // convert linear scale into lograthimic exponent
for other pow function


  // Check for out of range inputValues
  if (inputValue < originalMin) {
    inputValue = originalMin;
  }
  if (inputValue > originalMax) {
    inputValue = originalMax;
  }

  // Zero Refference the values
  OriginalRange = originalMax - originalMin;

  if (newEnd > newBegin){
    NewRange = newEnd - newBegin;
  }
  else
  {
    NewRange = newBegin - newEnd;
    invFlag = 1;
  }
```

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


  // Check for originalMin > originalMax  - the math for all other cases i.e.
negative numbers seems to work out fine
  if (originalMin > originalMax ) {
    return 0;
  }

  if (invFlag == 0){
    rangedValue =  (pow(normalizedCurVal, curve) * NewRange) + newBegin;

  }
  else     // invert the ranges
  {
    rangedValue =  newBegin - (pow(normalizedCurVal, curve) * NewRange);
  }

  return rangedValue;

  }

  void loop() {

   //for mic
  uint8_t  i;
  uint16_t minLvl, maxLvl;
  int      n, height;
  // end mic
 // read the pushbutton input pin:
  buttonState = digitalRead(buttonPin);
    // compare the buttonState to its previous state
  if (buttonState != lastButtonState) {
    // if the state has changed, increment the counter
    if (buttonState == HIGH) {
      // if the current state is HIGH then the button
      // wend from off to on:
      buttonPushCounter++;
      //Serial.println("on");
      //Serial.print("number of button pushes:  ");
      //Serial.println(buttonPushCounter);
      if(buttonPushCounter==16) {
////////////////////////////////////////////////////////////////////////////
      buttonPushCounter=1;}
    }
    else {
      // if the current state is LOW then the button
```

```
      // wend from on to off:
      //Serial.println("off");
    }
  }
  // save the current state as the last state,
  //for next time through the loop
  lastButtonState = buttonState;


switch (buttonPushCounter){

    case 1:
     buttonPushCounter==1; {
     All2(); // NORMAL
      break;}

      case 2:
     buttonPushCounter==2; {
     vu(); // NORMAL
      break;}

     case 3:
     buttonPushCounter==3; {
       vu1(); // Centre out
      break;}

   case 4:
     buttonPushCounter==4; {
    vu2(); // Centre Inwards
      break;}

    case 5:
     buttonPushCounter==5; {
    Vu3(); // Normal Rainbow
      break;}

      case 6:
     buttonPushCounter==6; {
    Vu4(); // Centre rainbow
      break;}

       case 7:
     buttonPushCounter==7; {
    Vu5(); // Shooting Star
      break;}

         case 8:
     buttonPushCounter==8; {
    Vu6(); // Falling star
```

```
      break;}

          case 9:
     buttonPushCounter==9; {
    vu7(); // Ripple with background
      break;}

            case 10:
     buttonPushCounter==10; {
    vu8(); // Shatter
      break;}

            case 11:
     buttonPushCounter==11; {
    vu9(); // Pulse
      break;}

            case 12:
     buttonPushCounter==12; {
     vu10(); // stream
      break;}
           case 13:
     buttonPushCounter==13; {
     vu11(); // Ripple without Background
      break;}

            case 14:
     buttonPushCounter==14; {
     vu12(); // Ripple without Background
      break;}

                 case 15:
     buttonPushCounter==15; {
     vu13(); // Ripple without Background
      break;}


           case 16:
     buttonPushCounter==16; {
    colorWipe(strip.Color(0, 0, 0), 10); // Black
      break;}
}

}

void colorWipe(uint32_t c, uint8_t wait) {
  for(uint16_t i=0; i<strip.numPixels(); i++) {
      strip.setPixelColor(i, c);
      strip.show();
```

```
      if (digitalRead(buttonPin) != lastButtonState)  // <------------- add
this
       return;         // <------------ and this
      delay(wait);
}}


void vu() {

  uint8_t  i;
  uint16_t minLvl, maxLvl;
  int      n, height;

  n   = analogRead(MIC_PIN);                        // Raw reading from mic
  n   = abs(n - 512 - DC_OFFSET); // Center on zero
  n   = (n <= NOISE) ? 0 : (n - NOISE);             // Remove noise/hum
  lvl = ((lvl * 7) + n) >> 3;    // "Dampened" reading (else looks twitchy)

  // Calculate bar height based on dynamic min/max levels (fixed point):
  height = TOP * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

  if(height < 0L)       height = 0;      // Clip output
  else if(height > TOP) height = TOP;
  if(height > peak)     peak   = height; // Keep 'peak' dot at top


  // Color pixels based on rainbow gradient
  for(i=0; i<N_PIXELS; i++) {
    if(i >= height)               strip.setPixelColor(i,   0,   0, 0);
    else strip.setPixelColor(i,Wheel(map(i,0,strip.numPixels()-1,30,150)));

  }


  // Draw peak dot
  if(peak > 0 && peak <= N_PIXELS-1)
strip.setPixelColor(peak,Wheel(map(peak,0,strip.numPixels()-1,30,150)));

   strip.show(); // Update strip

// Every few frames, make the peak pixel drop by 1:

    if(++dotCount >= PEAK_FALL) { //fall rate

      if(peak > 0) peak--;
      dotCount = 0;
    }
```

```
  vol[volCount] = n;                      // Save sample for dynamic leveling
  if(++volCount >= SAMPLES) volCount = 0; // Advance/rollover sample counter

  // Get volume range of prior frames
  minLvl = maxLvl = vol[0];
  for(i=1; i<SAMPLES; i++) {
    if(vol[i] < minLvl)      minLvl = vol[i];
    else if(vol[i] > maxLvl) maxLvl = vol[i];
  }
  // minLvl and maxLvl indicate the volume range over prior frames, used
  // for vertically scaling the output graph (so it looks interesting
  // regardless of volume level).  If they're too close together though
  // (e.g. at very low volume levels) the graph becomes super coarse
  // and 'jumpy'...so keep some minimum distance between them (this
  // also lets the graph go to zero when no sound is playing):
  if((maxLvl - minLvl) < TOP) maxLvl = minLvl + TOP;
  minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Dampen min/max levels
  maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (fake rolling average)

}

// Input a value 0 to 255 to get a color value.
// The colors are a transition r - g - b - back to r.
uint32_t Wheel(byte WheelPos) {
  if(WheelPos < 85) {
   return strip.Color(WheelPos * 3, 255 - WheelPos * 3, 0);
  } else if(WheelPos < 170) {
   WheelPos -= 85;
   return strip.Color(255 - WheelPos * 3, 0, WheelPos * 3);
  } else {
   WheelPos -= 170;
   return strip.Color(0, WheelPos * 3, 255 - WheelPos * 3);
  }
}


void vu1() {

  uint8_t  i;
  uint16_t minLvl, maxLvl;
  int      n, height;



  n   = analogRead(MIC_PIN);                        // Raw reading from mic
  n   = abs(n - 512 - DC_OFFSET); // Center on zero
  n   = (n <= NOISE) ? 0 : (n - NOISE);             // Remove noise/hum
  lvl = ((lvl * 7) + n) >> 3;    // "Dampened" reading (else looks twitchy)
```

```
  // Calculate bar height based on dynamic min/max levels (fixed point):
  height = TOP * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

  if(height < 0L)       height = 0;      // Clip output
  else if(height > TOP) height = TOP;
  if(height > peak)     peak   = height; // Keep 'peak' dot at top


  // Color pixels based on rainbow gradient
  for(i=0; i<N_PIXELS_HALF; i++) {
    if(i >= height) {
      strip.setPixelColor(N_PIXELS_HALF-i-1,   0,   0, 0);
      strip.setPixelColor(N_PIXELS_HALF+i,   0,   0, 0);
    }
    else {
      uint32_t color = Wheel(map(i,0,N_PIXELS_HALF-1,30,150));
      strip.setPixelColor(N_PIXELS_HALF-i-1,color);
      strip.setPixelColor(N_PIXELS_HALF+i,color);
    }

  }

  // Draw peak dot
  if(peak > 0 && peak <= N_PIXELS_HALF-1) {
    uint32_t color = Wheel(map(peak,0,N_PIXELS_HALF-1,30,150));
    strip.setPixelColor(N_PIXELS_HALF-peak-1,color);
    strip.setPixelColor(N_PIXELS_HALF+peak,color);
  }

   strip.show(); // Update strip

// Every few frames, make the peak pixel drop by 1:

    if(++dotCount >= PEAK_FALL) { //fall rate

      if(peak > 0) peak--;
      dotCount = 0;
    }



  vol[volCount] = n;                      // Save sample for dynamic leveling
  if(++volCount >= SAMPLES) volCount = 0; // Advance/rollover sample counter

  // Get volume range of prior frames
  minLvl = maxLvl = vol[0];
  for(i=1; i<SAMPLES; i++) {
    if(vol[i] < minLvl)      minLvl = vol[i];
    else if(vol[i] > maxLvl) maxLvl = vol[i];
```

```
  }
  // minLvl and maxLvl indicate the volume range over prior frames, used
  // for vertically scaling the output graph (so it looks interesting
  // regardless of volume level).  If they're too close together though
  // (e.g. at very low volume levels) the graph becomes super coarse
  // and 'jumpy'...so keep some minimum distance between them (this
  // also lets the graph go to zero when no sound is playing):
  if((maxLvl - minLvl) < TOP) maxLvl = minLvl + TOP;
  minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Dampen min/max levels
  maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (fake rolling average)

}



void vu2()
    {
      unsigned long startMillis= millis();  // Start of sample window
      float peakToPeak = 0;   // peak-to-peak level

      unsigned int signalMax = 0;
      unsigned int signalMin = 1023;
      unsigned int c, y;

      while (millis() - startMillis < SAMPLE_WINDOW)
      {
        sample = analogRead(MIC_PIN);
        if (sample < 1024)
        {
          if (sample > signalMax)
          {
            signalMax = sample;
          }
          else if (sample < signalMin)
          {
            signalMin = sample;
          }
        }
      }
      peakToPeak = signalMax - signalMin;

      // Serial.println(peakToPeak);



      for (int i=0;i<=N_PIXELS_HALF-1;i++){
        uint32_t color = Wheel(map(i,0,N_PIXELS_HALF-1,30,150));
        strip.setPixelColor(N_PIXELS-i,color);
        strip.setPixelColor(0+i,color);
```

```
      }


      c = fscale(INPUT_FLOOR, INPUT_CEILING, N_PIXELS_HALF, 0, peakToPeak, 2);

      if(c < peak) {
        peak = c;        // Keep dot on top
        dotHangCount = 0;    // make the dot hang before falling
      }
      if (c <= strip.numPixels()) { // Fill partial column with off pixels
        drawLine(N_PIXELS_HALF, N_PIXELS_HALF-c, strip.Color(0, 0, 0));
        drawLine(N_PIXELS_HALF, N_PIXELS_HALF+c, strip.Color(0, 0, 0));
      }




      y = N_PIXELS_HALF - peak;
      uint32_t color1 = Wheel(map(y,0,N_PIXELS_HALF-1,30,150));
      strip.setPixelColor(y-1,color1);
      //strip.setPixelColor(y-1,Wheel(map(y,0,N_PIXELS_HALF-1,30,150)));

      y = N_PIXELS_HALF + peak;
      strip.setPixelColor(y,color1);
      //strip.setPixelColor(y+1,Wheel(map(y,0,N_PIXELS_HALF+1,30,150)));

      strip.show();

      // Frame based peak dot animation
      if(dotHangCount > PEAK_HANG) { //Peak pause length
        if(++dotCount >= PEAK_FALL2) { //Fall rate
          peak++;
          dotCount = 0;
        }
      }
      else {
        dotHangCount++;
      }
    }


void Vu3() {
  uint8_t i;
  uint16_t minLvl, maxLvl;
  int n, height;

  n = analogRead(MIC_PIN);              // Raw reading from mic
  n = abs(n - 512 - DC_OFFSET);         // Center on zero
  n = (n <= NOISE) ? 0 : (n - NOISE);  // Remove noise/hum
```

```
  lvl = ((lvl * 7) + n) >> 3;    // "Dampened" reading (else looks twitchy)

  // Calculate bar height based on dynamic min/max levels (fixed point):
  height = TOP * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

  if (height < 0L)       height = 0;      // Clip output
  else if (height > TOP) height = TOP;
  if (height > peak)     peak   = height; // Keep 'peak' dot at top

  greenOffset += SPEED;
  blueOffset += SPEED;
  if (greenOffset >= 255) greenOffset = 0;
  if (blueOffset >= 255) blueOffset = 0;

  // Color pixels based on rainbow gradient
  for (i = 0; i < N_PIXELS; i++) {
    if (i >= height) {
      strip.setPixelColor(i, 0, 0, 0);
    } else {
      strip.setPixelColor(i, Wheel(
        map(i, 0, strip.numPixels() - 1, (int)greenOffset, (int)blueOffset)
      ));
    }
  }
  // Draw peak dot
  if(peak > 0 && peak <= N_PIXELS-1)
strip.setPixelColor(peak,Wheel(map(peak,0,strip.numPixels()-1,30,150)));

   strip.show(); // Update strip

// Every few frames, make the peak pixel drop by 1:

    if(++dotCount >= PEAK_FALL) { //fall rate

      if(peak > 0) peak--;
      dotCount = 0;
    }
  strip.show();  // Update strip

  vol[volCount] = n;
  if (++volCount >= SAMPLES) {
    volCount = 0;
  }

  // Get volume range of prior frames
  minLvl = maxLvl = vol[0];
  for (i = 1; i < SAMPLES; i++) {
    if (vol[i] < minLvl) {
      minLvl = vol[i];
```

```
    } else if (vol[i] > maxLvl) {
      maxLvl = vol[i];
    }
  }

  // minLvl and maxLvl indicate the volume range over prior frames, used
  // for vertically scaling the output graph (so it looks interesting
  // regardless of volume level).  If they're too close together though
  // (e.g. at very low volume levels) the graph becomes super coarse
  // and 'jumpy'...so keep some minimum distance between them (this
  // also lets the graph go to zero when no sound is playing):
  if ((maxLvl - minLvl) < TOP) {
    maxLvl = minLvl + TOP;
  }
  minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Dampen min/max levels
  maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (fake rolling average)
}


void Vu4() {
    uint8_t  i;
  uint16_t minLvl, maxLvl;
  int      n, height;

  n   = analogRead(MIC_PIN);                        // Raw reading from mic
  n   = abs(n - 512 - DC_OFFSET); // Center on zero
  n   = (n <= NOISE) ? 0 : (n - NOISE);             // Remove noise/hum
  lvl = ((lvl * 7) + n) >> 3;    // "Dampened" reading (else looks twitchy)

  // Calculate bar height based on dynamic min/max levels (fixed point):
  height = TOP * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

  if(height < 0L)       height = 0;      // Clip output
  else if(height > TOP) height = TOP;
  if(height > peak)     peak   = height; // Keep 'peak' dot at top
  greenOffset += SPEED;
  blueOffset += SPEED;
  if (greenOffset >= 255) greenOffset = 0;
  if (blueOffset >= 255) blueOffset = 0;

  // Color pixels based on rainbow gradient
  for(i=0; i<N_PIXELS_HALF; i++) {
    if(i >= height) {
      strip.setPixelColor(N_PIXELS_HALF-i-1,   0,   0, 0);
      strip.setPixelColor(N_PIXELS_HALF+i,   0,   0, 0);
    }
    else {
      uint32_t color = Wheel(map(i,0,N_PIXELS_HALF-1,(int)greenOffset,
(int)blueOffset));
```

```
      strip.setPixelColor(N_PIXELS_HALF-i-1,color);
      strip.setPixelColor(N_PIXELS_HALF+i,color);
    }

  }

  // Draw peak dot
  if(peak > 0 && peak <= N_PIXELS_HALF-1) {
    uint32_t color = Wheel(map(peak,0,N_PIXELS_HALF-1,30,150));
    strip.setPixelColor(N_PIXELS_HALF-peak-1,color);
    strip.setPixelColor(N_PIXELS_HALF+peak,color);
  }

   strip.show(); // Update strip

// Every few frames, make the peak pixel drop by 1:

    if(++dotCount >= PEAK_FALL) { //fall rate

      if(peak > 0) peak--;
      dotCount = 0;
    }


  vol[volCount] = n;                      // Save sample for dynamic leveling
  if(++volCount >= SAMPLES) volCount = 0; // Advance/rollover sample counter

  // Get volume range of prior frames
  minLvl = maxLvl = vol[0];
  for(i=1; i<SAMPLES; i++) {
    if(vol[i] < minLvl)      minLvl = vol[i];
    else if(vol[i] > maxLvl) maxLvl = vol[i];
  }
  // minLvl and maxLvl indicate the volume range over prior frames, used
  // for vertically scaling the output graph (so it looks interesting
  // regardless of volume level).  If they're too close together though
  // (e.g. at very low volume levels) the graph becomes super coarse
  // and 'jumpy'...so keep some minimum distance between them (this
  // also lets the graph go to zero when no sound is playing):
  if((maxLvl - minLvl) < TOP) maxLvl = minLvl + TOP;
  minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Dampen min/max levels
  maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (fake rolling average)

}

void Vu5()
{
  uint8_t  i;
  uint16_t minLvl, maxLvl;
```

```
  int      n, height;

  n   = analogRead(MIC_PIN);                        // Raw reading from mic
  n   = abs(n - 512 - DC_OFFSET); // Center on zero
  n   = (n <= NOISE) ? 0 : (n - NOISE);             // Remove noise/hum
  lvl = ((lvl * 7) + n) >> 3;    // "Dampened" reading (else looks twitchy)

  // Calculate bar height based on dynamic min/max levels (fixed point):
  height = TOP2 * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

  if(height < 0L)       height = 0;      // Clip output
  else if(height > TOP2) height = TOP2;
  if(height > peak)     peak   = height; // Keep 'peak' dot at top


#ifdef CENTERED
 // Color pixels based on rainbow gradient
  for(i=0; i<(N_PIXELS/2); i++) {
    if(((N_PIXELS/2)+i) >= height)
    {
      strip.setPixelColor(((N_PIXELS/2) + i),   0,   0, 0);
      strip.setPixelColor(((N_PIXELS/2) - i),   0,   0, 0);
    }
    else
    {
      strip.setPixelColor(((N_PIXELS/2) + i),Wheel(map(((N_PIXELS/2) +
i),0,strip.numPixels()-1,30,150)));
      strip.setPixelColor(((N_PIXELS/2) - i),Wheel(map(((N_PIXELS/2) -
i),0,strip.numPixels()-1,30,150)));
    }
  }

  // Draw peak dot
  if(peak > 0 && peak <= LAST_PIXEL_OFFSET)
  {
    strip.setPixelColor(((N_PIXELS/2) + peak),255,255,255); //
(peak,Wheel(map(peak,0,strip.numPixels()-1,30,150)));
    strip.setPixelColor(((N_PIXELS/2) - peak),255,255,255); //
(peak,Wheel(map(peak,0,strip.numPixels()-1,30,150)));
  }
#else
  // Color pixels based on rainbow gradient
  for(i=0; i<N_PIXELS; i++)
  {
    if(i >= height)
    {
      strip.setPixelColor(i,   0,   0, 0);
    }
    else
```

```
    {
      strip.setPixelColor(i,Wheel(map(i,0,strip.numPixels()-1,30,150)));
    }
  }

  // Draw peak dot
  if(peak > 0 && peak <= LAST_PIXEL_OFFSET)
  {
    strip.setPixelColor(peak,255,255,255); //
(peak,Wheel(map(peak,0,strip.numPixels()-1,30,150)));
  }

#endif

  // Every few frames, make the peak pixel drop by 1:

  if (millis() - lastTime >= PEAK_FALL_MILLIS)
  {
    lastTime = millis();

    strip.show(); // Update strip

    //fall rate
    if(peak > 0) peak--;
    }

  vol[volCount] = n;                      // Save sample for dynamic leveling
  if(++volCount >= SAMPLES) volCount = 0; // Advance/rollover sample counter

  // Get volume range of prior frames
  minLvl = maxLvl = vol[0];
  for(i=1; i<SAMPLES; i++)
  {
    if(vol[i] < minLvl)      minLvl = vol[i];
    else if(vol[i] > maxLvl) maxLvl = vol[i];
  }
  // minLvl and maxLvl indicate the volume range over prior frames, used
  // for vertically scaling the output graph (so it looks interesting
  // regardless of volume level).  If they're too close together though
  // (e.g. at very low volume levels) the graph becomes super coarse
  // and 'jumpy'...so keep some minimum distance between them (this
  // also lets the graph go to zero when no sound is playing):
  if((maxLvl - minLvl) < TOP2) maxLvl = minLvl + TOP2;
  minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Dampen min/max levels
  maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (fake rolling average)
}

void Vu6()
{
```

```
  uint8_t  i;
  uint16_t minLvl, maxLvl;
  int      n, height;

  n   = analogRead(MIC_PIN);                        // Raw reading from mic
  n   = abs(n - 512 - DC_OFFSET); // Center on zero
  n   = (n <= NOISE) ? 0 : (n - NOISE);             // Remove noise/hum
  lvl = ((lvl * 7) + n) >> 3;    // "Dampened" reading (else looks twitchy)

  // Calculate bar height based on dynamic min/max levels (fixed point):
  height = TOP2 * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

  if(height < 0L)       height = 0;      // Clip output
  else if(height > TOP2) height = TOP2;
  if(height > peak)     peak   = height; // Keep 'peak' dot at top


#ifdef CENTERED
  // Draw peak dot
  if(peak > 0 && peak <= LAST_PIXEL_OFFSET)
  {
    strip.setPixelColor(((N_PIXELS/2) + peak),255,255,255); //
(peak,Wheel(map(peak,0,strip.numPixels()-1,30,150)));
    strip.setPixelColor(((N_PIXELS/2) - peak),255,255,255); //
(peak,Wheel(map(peak,0,strip.numPixels()-1,30,150)));
  }
#else
  // Color pixels based on rainbow gradient
  for(i=0; i<N_PIXELS; i++)
  {
    if(i >= height)
    {
      strip.setPixelColor(i,   0,   0, 0);
    }
    else
    {
     }
  }

  // Draw peak dot
  if(peak > 0 && peak <= LAST_PIXEL_OFFSET)
  {
    strip.setPixelColor(peak,0,0,255); //
(peak,Wheel(map(peak,0,strip.numPixels()-1,30,150)));
  }

#endif

  // Every few frames, make the peak pixel drop by 1:
```

```
  if (millis() - lastTime >= PEAK_FALL_MILLIS)
  {
    lastTime = millis();

    strip.show(); // Update strip

    //fall rate
    if(peak > 0) peak--;
    }

  vol[volCount] = n;                      // Save sample for dynamic leveling
  if(++volCount >= SAMPLES) volCount = 0; // Advance/rollover sample counter

  // Get volume range of prior frames
  minLvl = maxLvl = vol[0];
  for(i=1; i<SAMPLES; i++)
  {
    if(vol[i] < minLvl)      minLvl = vol[i];
    else if(vol[i] > maxLvl) maxLvl = vol[i];
  }
  // minLvl and maxLvl indicate the volume range over prior frames, used
  // for vertically scaling the output graph (so it looks interesting
  // regardless of volume level).  If they're too close together though
  // (e.g. at very low volume levels) the graph becomes super coarse
  // and 'jumpy'...so keep some minimum distance between them (this
  // also lets the graph go to zero when no sound is playing):
  if((maxLvl - minLvl) < TOP2) maxLvl = minLvl + TOP2;
  minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Dampen min/max levels
  maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (fake rolling average)
}

void vu7() {

  EVERY_N_MILLISECONDS(1000) {
    peakspersec = peakcount;                                          // Count
the peaks per second. This value will become the foreground hue.
    peakcount = 0;                                                    // Reset
the counter every second.
  }

  soundmems();

  EVERY_N_MILLISECONDS(20) {
   ripple3();
  }

   show_at_max_brightness_for_power();
```

```
} // loop()


void soundmems() {                                                  // Rolling
average counter - means we don't have to go through an array each time.
  newtime = millis();
  int tmp = analogRead(MIC_PIN) - 512;
  sample = abs(tmp);

  int potin = map(analogRead(POT_PIN), 0, 1023, 0, 60);

  samplesum = samplesum + sample - samplearray[samplecount];        // Add the
new sample and remove the oldest sample in the array
  sampleavg = samplesum / NSAMPLES;                                 // Get an
average
  samplearray[samplecount] = sample;                                // Update
oldest sample in the array with new sample
  samplecount = (samplecount + 1) % NSAMPLES;                       // Update
the counter for the array

  if (newtime > (oldtime + 200)) digitalWrite(13, LOW);             // Turn
the LED off 200ms after the last peak.

  if ((sample > (sampleavg + potin)) && (newtime > (oldtime + 60)) ) { //
Check for a peak, which is 30 > the average, but wait at least 60ms for
another.
    step = -1;
    peakcount++;
    digitalWrite(13, HIGH);
    oldtime = newtime;
  }
}  // soundmems()



void ripple3() {
  for (int i = 0; i < N_PIXELS; i++) leds[i] = CHSV(bgcol, 255,
sampleavg*2);  // Set the background colour.

  switch (step) {

    case -1:                                                          //
Initialize ripple variables.
      center = random(N_PIXELS);
      colour = (peakspersec*10) %
255;                                          // More peaks/s = higher the
hue colour.
      step = 0;
      bgcol = bgcol+8;
```

```
      break;

    case 0:
      leds[center] = CHSV(colour, 255, 255);                          //
Display the first pixel of the ripple.
      step ++;
      break;

    case maxsteps:                                                    // At
the end of the ripples.
      // step = -1;
      break;

    default:                                                            //
Middle of the ripples.
      leds[(center + step + N_PIXELS) % N_PIXELS] += CHSV(colour, 255,
myfade/step*2);       // Simple wrap from Marc Miller.
      leds[(center - step + N_PIXELS) % N_PIXELS] += CHSV(colour, 255,
myfade/step*2);
      step ++;                                                         // Next
step.
      break;
  } // switch step
} // ripple()


void vu8() {
  int intensity = calculateIntensity();
  updateOrigin(intensity);
  assignDrawValues(intensity);
  writeSegmented();
  updateGlobals();
}

int calculateIntensity() {
  int      intensity;

  reading   = analogRead(MIC_PIN);                        // Raw reading from
mic
  reading   = abs(reading - 512 - DC_OFFSET); // Center on zero
  reading   = (reading <= NOISE) ? 0 : (reading - NOISE);             //
Remove noise/hum
  lvl = ((lvl * 7) + reading) >> 3;    // "Dampened" reading (else looks
twitchy)

  // Calculate bar height based on dynamic min/max levels (fixed point):
  intensity = DRAW_MAX * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

  return constrain(intensity, 0, DRAW_MAX-1);
```

```
}

void updateOrigin(int intensity) {
  // detect peak change and save origin at curve vertex
  if (growing && intensity < last_intensity) {
    growing = false;
    intensity_max = last_intensity;
    fall_from_left = !fall_from_left;
    origin_at_flip = origin;
  } else if (intensity > last_intensity) {
    growing = true;
    origin_at_flip = origin;
  }
  last_intensity = intensity;

  // adjust origin if falling
  if (!growing) {
    if (fall_from_left) {
      origin = origin_at_flip + ((intensity_max - intensity) / 2);
    } else {
      origin = origin_at_flip - ((intensity_max - intensity) / 2);
    }
    // correct for origin out of bounds
    if (origin < 0) {
      origin = DRAW_MAX - abs(origin);
    } else if (origin > DRAW_MAX - 1) {
      origin = origin - DRAW_MAX - 1;
    }
  }
}

void assignDrawValues(int intensity) {
  // draw amplitue as 1/2 intensity both directions from origin
  int min_lit = origin - (intensity / 2);
  int max_lit = origin + (intensity / 2);
  if (min_lit < 0) {
    min_lit = min_lit + DRAW_MAX;
  }
  if (max_lit >= DRAW_MAX) {
    max_lit = max_lit - DRAW_MAX;
  }
  for (int i=0; i < DRAW_MAX; i++) {
    // if i is within origin +/- 1/2 intensity
    if (
      (min_lit < max_lit && min_lit < i && i < max_lit) // range is within
bounds and i is within range
      || (min_lit > max_lit && (i > min_lit || i < max_lit)) // range wraps
out of bounds and i is within that wrap
    ) {
```

```
      draw[i] = Wheel(scroll_color);
    } else {
      draw[i] = 0;
    }
  }
}

void writeSegmented() {
  int seg_len = N_PIXELS / SEGMENTS;

  for (int s = 0; s < SEGMENTS; s++) {
    for (int i = 0; i < seg_len; i++) {
      strip.setPixelColor(i + (s*seg_len), draw[map(i, 0, seg_len, 0,
DRAW_MAX)]);
    }
  }
  strip.show();
}

uint32_t * segmentAndResize(uint32_t* draw) {
  int seg_len = N_PIXELS / SEGMENTS;

  uint32_t segmented[N_PIXELS];
  for (int s = 0; s < SEGMENTS; s++) {
    for (int i = 0; i < seg_len; i++) {
      segmented[i + (s * seg_len) ] = draw[map(i, 0, seg_len, 0, DRAW_MAX)];
    }
  }

  return segmented;
}

void writeToStrip(uint32_t* draw) {
  for (int i = 0; i < N_PIXELS; i++) {
    strip.setPixelColor(i, draw[i]);
  }
  strip.show();
}

void updateGlobals() {
  uint16_t minLvl, maxLvl;

  //advance color wheel
  color_wait_count++;
  if (color_wait_count > COLOR_WAIT_CYCLES) {
    color_wait_count = 0;
    scroll_color++;
    if (scroll_color > COLOR_MAX) {
      scroll_color = COLOR_MIN;
```

```
    }
  }

  vol[volCount] = reading;                      // Save sample for dynamic
leveling
  if(++volCount >= SAMPLES) volCount = 0; // Advance/rollover sample counter

  // Get volume range of prior frames
  minLvl = maxLvl = vol[0];
  for(uint8_t i=1; i<SAMPLES; i++) {
    if(vol[i] < minLvl)      minLvl = vol[i];
    else if(vol[i] > maxLvl) maxLvl = vol[i];
  }
  // minLvl and maxLvl indicate the volume range over prior frames, used
  // for vertically scaling the output graph (so it looks interesting
  // regardless of volume level).  If they're too close together though
  // (e.g. at very low volume levels) the graph becomes super coarse
  // and 'jumpy'...so keep some minimum distance between them (this
  // also lets the graph go to zero when no sound is playing):
  if((maxLvl - minLvl) < N_PIXELS) maxLvl = minLvl + N_PIXELS;
  minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Dampen min/max levels
  maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (fake rolling average)
}




void vu9() {
  //currentBlending = LINEARBLEND;
  currentPalette = OceanColors_p;                                 // Initial
palette.
  currentBlending = LINEARBLEND;
  EVERY_N_SECONDS(5) {                                            // Change the
palette every 5 seconds.
    for (int i = 0; i < 16; i++) {
      targetPalette[i] = CHSV(random8(), 255, 255);
    }
  }

  EVERY_N_MILLISECONDS(100) {                                     // AWESOME
palette blending capability once they do change.
    uint8_t maxChanges = 24;
    nblendPaletteTowardPalette(currentPalette, targetPalette, maxChanges);
  }


  EVERY_N_MILLIS_I(thistimer,20) {                                // For fun,
let's make the animation have a variable rate.
```

```
    uint8_t timeval = beatsin8(10,20,50);                    // Use a
sinewave for the line below. Could also use peak/beat detection.
    thistimer.setPeriod(timeval);                            // Allows you to
change how often this routine runs.
    fadeToBlackBy(leds, N_PIXELS, 16);                       // 1 = slow, 255
= fast fade. Depending on the faderate, the LED's further away will fade out.
    sndwave();
    soundble();
  }
  FastLED.setBrightness(max_bright);
  FastLED.show();

} // loop()



void soundble() {                                          // Quick and
dirty sampling of the microphone.

  int tmp = analogRead(MIC_PIN) - 512 - DC_OFFSET;
  sample = abs(tmp);

}  // soundmems()



void sndwave() {

  leds[N_PIXELS/2] = ColorFromPalette(currentPalette, sample, sample*2,
currentBlending); // Put the sample into the center

  for (int i = N_PIXELS - 1; i > N_PIXELS/2; i--) {       //move to the
left      // Copy to the left, and let the fade do the rest.
    leds[i] = leds[i - 1];
  }

  for (int i = 0; i < N_PIXELS/2; i++) {                  // move to the
right    // Copy to the right, and let the fade to the rest.
    leds[i] = leds[i + 1];
  }
  addGlitter(sampleavg);
}

void vu10() {

EVERY_N_SECONDS(5) {                                       // Change the
target palette to a random one every 5 seconds.
    static uint8_t baseC = random8();                        // You can use
this as a baseline colour if you want similar hues in the next line.
```

```
    for (int i = 0; i < 16; i++) {
      targetPalette[i] = CHSV(random8(), 255, 255);
    }
  }

  EVERY_N_MILLISECONDS(100) {
    uint8_t maxChanges = 24;
    nblendPaletteTowardPalette(currentPalette, targetPalette,
maxChanges);   // AWESOME palette blending capability.
  }

  EVERY_N_MILLISECONDS(thisdelay) {                          // FastLED based
non-blocking delay to update/display the sequence.
    soundtun();
    FastLED.setBrightness(max_bright);
    FastLED.show();
  }
} // loop()



void soundtun() {

  int n;
  n = analogRead(MIC_PIN);                                   // Raw reading
from mic
  n = qsuba(abs(n-512), 10);                                 // Center on
zero and get rid of low level noise
  CRGB newcolour = ColorFromPalette(currentPalette, constrain(n,0,255),
constrain(n,0,255), currentBlending);
  nblend(leds[0], newcolour, 128);

  for (int i = N_PIXELS-1; i>0; i--) {
    leds[i] = leds[i-1];
  }

} // soundmems()


void vu11() {

  EVERY_N_MILLISECONDS(1000) {
    peakspersec = peakcount;                                 // Count the
peaks per second. This value will become the foreground hue.
    peakcount = 0;                                           // Reset the
counter every second.
  }
```

```
  soundrip();

  EVERY_N_MILLISECONDS(20) {
   rippled();
  }

   FastLED.show();

} // loop()


void soundrip() {                                       // Rolling
average counter - means we don't have to go through an array each time.

  newtime = millis();
  int tmp = analogRead(MIC_PIN) - 512;
  sample = abs(tmp);

  int potin = map(analogRead(POT_PIN), 0, 1023, 0, 60);

  samplesum = samplesum + sample - samplearray[samplecount];  // Add the new
sample and remove the oldest sample in the array
  sampleavg = samplesum / NSAMPLES;                         // Get an
average

  //Serial.println(sampleavg);


  samplearray[samplecount] = sample;                        // Update oldest
sample in the array with new sample
  samplecount = (samplecount + 1) % NSAMPLES;               // Update the
counter for the array

  if (newtime > (oldtime + 200)) digitalWrite(13, LOW);       // Turn the LED
off 200ms after the last peak.

  if ((sample > (sampleavg + potin)) && (newtime > (oldtime + 60)) ) { //
Check for a peak, which is 30 > the average, but wait at least 60ms for
another.
    step = -1;
    peakcount++;
    oldtime = newtime;
  }

}  // soundmems()



void rippled() {
```

```
  fadeToBlackBy(leds, N_PIXELS, 64);                          // 8 bit, 1 =
slow, 255 = fast

  switch (step) {

    case -1:                                                  // Initialize
ripple variables.
      center = random(N_PIXELS);
      colour = (peakspersec*10) % 255;                        // More peaks/s
= higher the hue colour.
      step = 0;
      break;

    case 0:
      leds[center] = CHSV(colour, 255, 255);                  // Display the
first pixel of the ripple.
      step ++;
      break;

    case maxsteps:                                            // At the end of
the ripples.
      // step = -1;
      break;

    default:                                                  // Middle of the
ripples.
      leds[(center + step + N_PIXELS) % N_PIXELS] += CHSV(colour, 255,
myfade/step*2);       // Simple wrap from Marc Miller.
      leds[(center - step + N_PIXELS) % N_PIXELS] += CHSV(colour, 255,
myfade/step*2);
      step ++;                                                // Next step.
      break;
  } // switch step

} // ripple()


 //Used to draw a line between two points of a given color
    void drawLine(uint8_t from, uint8_t to, uint32_t c) {
      uint8_t fromTemp;
      if (from > to) {
        fromTemp = from;
        from = to;
        to = fromTemp;
      }
      for(int i=from; i<=to; i++){
        strip.setPixelColor(i, c);
      }
```

```
    }

void setPixel(int Pixel, byte red, byte green, byte blue) {
    strip.setPixelColor(Pixel, strip.Color(red, green, blue));

}


void setAll(byte red, byte green, byte blue) {

  for(int i = 0; i < N_PIXELS; i++ ) {

    setPixel(i, red, green, blue);

  }

  strip.show();

}
void vu12() {

  EVERY_N_MILLISECONDS(1000) {
    peakspersec = peakcount;                                  // Count the
peaks per second. This value will become the foreground hue.
    peakcount = 0;                                            // Reset the
counter every second.
  }

  soundripped();

  EVERY_N_MILLISECONDS(20) {
  rippvu();
  }

   FastLED.show();

} // loop()


void soundripped() {                                            // Rolling
average counter - means we don't have to go through an array each time.

  newtime = millis();
  int tmp = analogRead(MIC_PIN) - 512;
  sample = abs(tmp);

  int potin = map(analogRead(POT_PIN), 0, 1023, 0, 60);
```

```
  samplesum = samplesum + sample - samplearray[samplecount];  // Add the new
sample and remove the oldest sample in the array
  sampleavg = samplesum / NSAMPLES;                           // Get an
average

  //Serial.println(sampleavg);


  samplearray[samplecount] = sample;                          // Update oldest
sample in the array with new sample
  samplecount = (samplecount + 1) % NSAMPLES;                 // Update the
counter for the array

  if (newtime > (oldtime + 200)) digitalWrite(13, LOW);       // Turn the LED
off 200ms after the last peak.

  if ((sample > (sampleavg + potin)) && (newtime > (oldtime + 60)) ) { //
Check for a peak, which is 30 > the average, but wait at least 60ms for
another.
    step = -1;
    peakcount++;
    oldtime = newtime;
  }

}  // soundmems()
void rippvu()
{                                                             // Display
ripples triggered by peaks.

fadeToBlackBy(leds, N_PIXELS, 64);                           // 8 bit, 1 =
slow, 255 = fast

  switch (step) {

    case -1:                                                  // Initialize
ripple variables.
      center = random(N_PIXELS);
      colour = (peakspersec*10) % 255;                        // More peaks/s
= higher the hue colour.
      step = 0;
      break;

    case 0:
      leds[center] = CHSV(colour, 255, 255);                  // Display the
first pixel of the ripple.
      step ++;
      break;
```

```
    case maxsteps:                                          // At the end of
the ripples.
      // step = -1;
      break;

    default:                                                // Middle of the
ripples.
      leds[(center + step + N_PIXELS) % N_PIXELS] += CHSV(colour, 255,
myfade/step*2);       // Simple wrap from Marc Miller.
      leds[(center - step + N_PIXELS) % N_PIXELS] += CHSV(colour, 255,
myfade/step*2);
      step ++;                                              // Next step.
      break;
  } // switch step
   addGlitter(sampleavg);
} // ripple()



void vu13()
{                                                                   // The
>>>>>>>>>> L-O-O-P <<<<<<<<<<<<<<<<<<<<<<<<<<  is buried here!!!11!1!


  EVERY_N_MILLISECONDS(1000) {
    peakspersec = peakcount;                                // Count the
peaks per second. This value will become the foreground hue.
    peakcount = 0;                                          // Reset the
counter every second.
  }

  soundripper();

  EVERY_N_MILLISECONDS(20) {
  jugglep();
  }

   FastLED.show();

} // loop()


void soundripper() {                                          // Rolling
average counter - means we don't have to go through an array each time.

  newtime = millis();
  int tmp = analogRead(MIC_PIN) - 512;
  sample = abs(tmp);
```

```
  int potin = map(analogRead(POT_PIN), 0, 1023, 0, 60);

  samplesum = samplesum + sample - samplearray[samplecount];  // Add the new
sample and remove the oldest sample in the array
  sampleavg = samplesum / NSAMPLES;                           // Get an
average

  //Serial.println(sampleavg);


  samplearray[samplecount] = sample;                          // Update oldest
sample in the array with new sample
  samplecount = (samplecount + 1) % NSAMPLES;                 // Update the
counter for the array

  if (newtime > (oldtime + 200)) digitalWrite(13, LOW);       // Turn the LED
off 200ms after the last peak.

  if ((sample > (sampleavg + potin)) && (newtime > (oldtime + 60)) ) { //
Check for a peak, which is 30 > the average, but wait at least 60ms for
another.
    step = -1;
    peakcount++;
    oldtime = newtime;
                                                  // Change the current
pattern function periodically.
    jugglep();
  }

} // loop()


void jugglep()
{                                                             // Use the
juggle routine, but adjust the timebase based on sampleavg for some
randomness.

// Persistent local variables
  static uint8_t thishue=0;

  timeval =
40;                                                             // Our
EVERY_N_MILLIS_I timer value.

  leds[0] = ColorFromPalette(currentPalette, thishue++, sampleavg,
LINEARBLEND);

  for (int i = N_PIXELS-1; i >0 ; i-- ) leds[i] = leds[i-1];
```

```
  addGlitter(sampleavg/2);
 // Add glitter based on sampleavg. By Andrew Tuline.

} // matrix()



// Input a value 0 to 255 to get a color value.
// The colours are a transition r - g - b - back to r.
uint32_t Wheel(byte WheelPos, float opacity) {

  if(WheelPos < 85) {
    return strip.Color((WheelPos * 3) * opacity, (255 - WheelPos * 3) *
opacity, 0);
  }
  else if(WheelPos < 170) {
    WheelPos -= 85;
    return strip.Color((255 - WheelPos * 3) * opacity, 0, (WheelPos * 3) *
opacity);
  }
  else {
    WheelPos -= 170;
    return strip.Color(0, (WheelPos * 3) * opacity, (255 - WheelPos * 3) *
opacity);
  }
}


void addGlitter( fract8 chanceOfGlitter)
{                                       // Let's add some glitter, thanks to
Mark

  if( random8() < chanceOfGlitter) {
    leds[random16(N_PIXELS)] += CRGB::White;
  }

} // addGlitter()
// List of patterns to cycle through.  Each is defined as a separate function
below.
typedef void (*SimplePatternList[])();
SimplePatternList qPatterns = {vu, vu1, vu2, Vu3, Vu4, Vu5, Vu6, vu7, vu8,
vu9, vu10, vu11, vu12, vu13};
uint8_t qCurrentPatternNumber = 0; // Index number of which pattern is current

void nextPattern2()
{
  // add one to the current pattern number, and wrap around at the end
```

```
  qCurrentPatternNumber = (qCurrentPatternNumber + 1) % ARRAY_SIZE(
qPatterns);
}
void All2()
{
  // Call the current pattern function once, updating the 'leds' array
  qPatterns[qCurrentPatternNumber]();
  EVERY_N_SECONDS( 120 ) { nextPattern2(); } // change patterns periodically
}
```