Gemma Firewalker Lite Sneakers

Created by Becky Stern

https://learn.adafruit.com/gemma-led-sneakers

Last updated on 2024-06-03 01:49:13 PM EDT

Table of Contents

Overview

Build your own flashy footwear! This project is a major upgrade to our popular Firewalker Sneakers (https://adafru.it/jMF) tutorial, but this time using GEMMA and a vibration sensor to trigger animations on the LED strip encircling the soles of your new favorite sneakers.

GEMMA didn't exist yet when we first made the Firewalkers!

Before you begin, you should read up on the following prerequisite guides:

- Introducing Gemma M0 guide (https://adafru.it/yeq) or Introducing GEMMA guide (https://adafru.it/cHH)
- NeoPixel Uberguide (https://adafru.it/dhw)
- Battery Powering Your Wearable Electronics (https://adafru.it/jNa)
- Adafruit Guide to Excellent Soldering (https://adafru.it/drl)

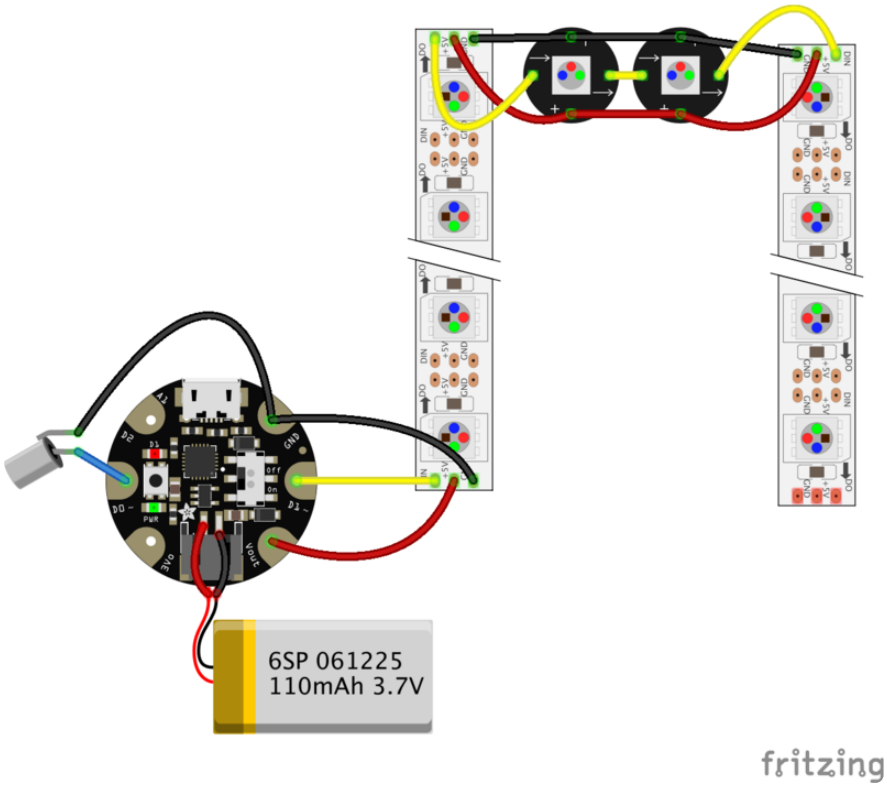
For this project you will need:

- 2x Gemma M0 (http://adafru.it/3501) or Gemma v2 (http://adafru.it/1222) Microcontrollers
- 2x Medium vibration sensor (http://adafru.it/2384) (you could try fast (http://adafru.it/1766) and slow (http://adafru.it/1767) varieties as well)
- 2x 500mAh lipoly batteries (http://adafru.it/1578) and at least one charger (http://adafru.it/1304)
- 4x Pack of mini NeoPixels (http://adafru.it/1612) (16-20 pixels for one pair of shoes)
- 2m of 60/m NeoPixel strip in white (http://adafru.it/1138) or black (http://adafru.it/1461)
- 26awg (http://adafru.it/1970) and 30awg (http://adafru.it/2051) silicone coated stranded wire in color(s) of your choice
- Sewing needle and thread
- access to a 3D printer and flexible filament for battery pockets (try 3D hubs (https://adafru.it/jNb) for local printers)
- Soldering tools and supplies (https://adafru.it/drl)
- Permatex 66B silicone adhesive (tube (https://adafru.it/jNc) or powerbead can (https://adafru.it/jNd))
- Latex or nitrile gloves
- Scissors
- Rubber bands

This guide was written for the Gemma v2 board, but can be done with either the Gemma v2 or M0 Gemma. We recommend the Gemma M0 as it is easier to use and is more compatible with modern computers!

Video for gifs by Mike Farino (https://adafru.it/iSE).

Circuit Diagram



This diagram uses the Gemma v2 but you can also use the Gemma M0 with the exact same wiring!

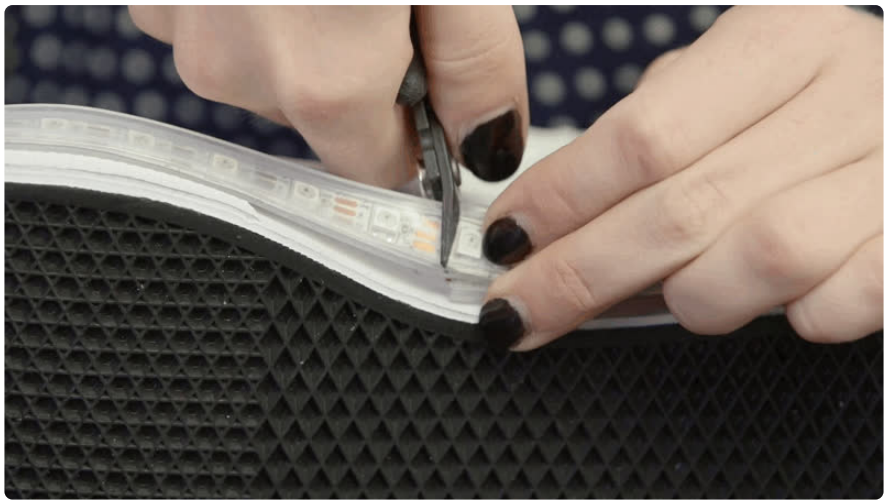
The vibration sensor connects to Gemma's **D0** and the remaining pin to GND.

The Neopixels are being controlled from Gemma's **D1** with and are also connected to Gemma's **Vout** (+) and **GND** (-).

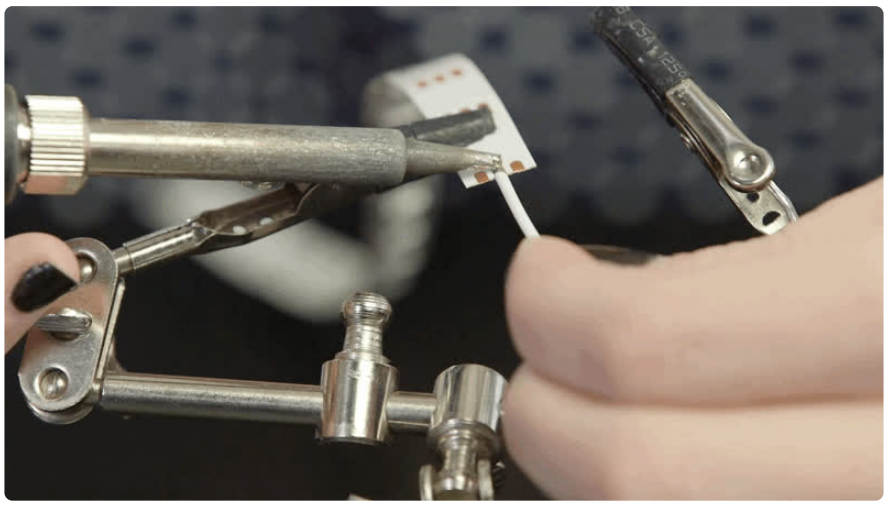
Prepare NeoPixel Strips

NeoPixel Strip is made of a flex PCB material that's meant to curve around surfaces, but it's not designed to bend laterally. On the original Firewalkers, the bend at the toe was vulnerable to breakage. They would break after a few times of wearing them, so we wanted to provide a way to mitigate that problem— the solution we came up with is to replace the bendiest parts of the strip with a homemade strip constructed of individual pixels and stranded wire, with extra slack for lateral bending. The result is much more labor-intensive than the original, so be warned! It requires delicate soldering of close-together parts and may not be suitable for beginners! You can always glue the strips on as in the original, then upgrade to this new method once the strip breaks...

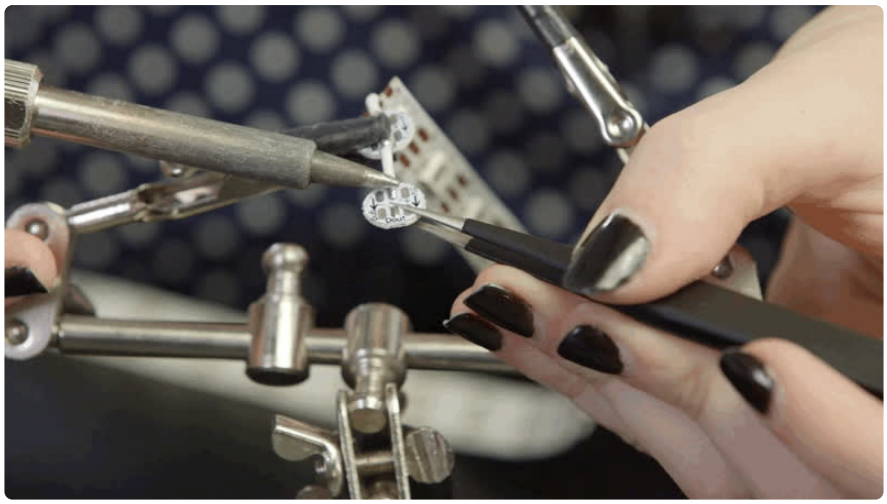
Wrap your NeoPixel strip around the sole of your shoe to determine the appropriate length, then cut it to size.



Preferably with your shoe on and laced up, bend the shoe at the toe and mark the bendiest parts— we'll be creating a more flexible section of NeoPixel strip to cover these areas. Hold the NeoPixel strip to the shoe sole again to see where your marks intersect with the strip, and mark down the pixel numbers for the sections to be replaced. For our mens' size 10.5 sneakers, these were pixels 19-23 and 35-39. Also note how many pixels make up the toe section (35-23=12 pixels around the toe).

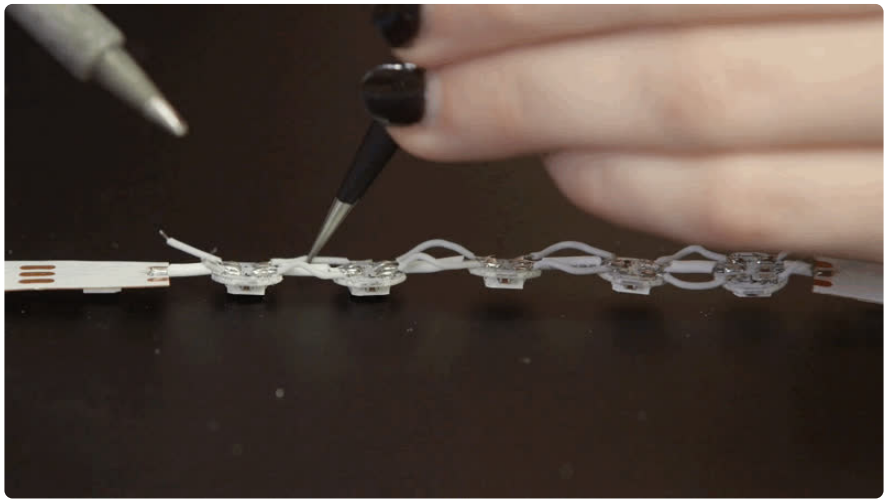


Remove the NeoPixels from their sheathing and set the sheathing aside for later. Being sure to start with the input end of your NeoPixel strip, count to the locations of the breaks and cut the strip into three pieces— you'll have extra strip at the end because the individual pixels will take their place. Solder a piece of 26 gauge to the center data line by tinning the pad and the wire, then reheating the solder to join the two.



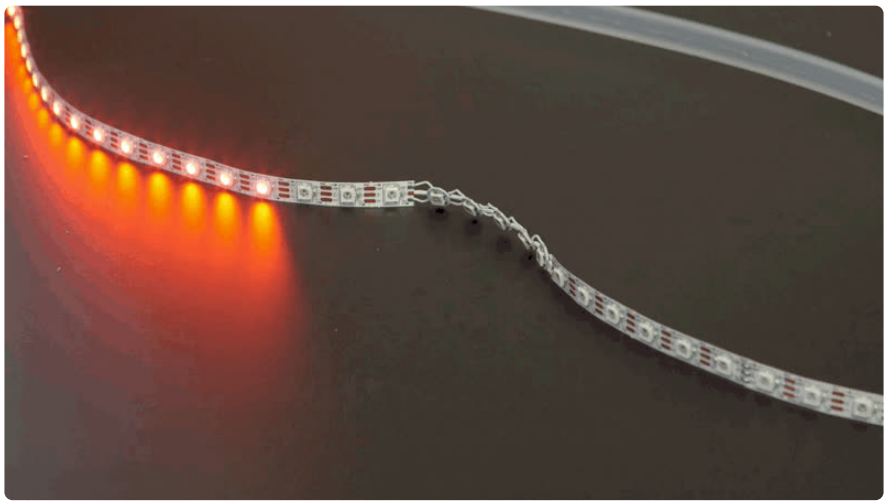
Trim and strip the wire and solder it to the input pad of an individual NeoPixel.

Continue soldering data output to data input lines down the chain until you've attached as many pixels as you like and have noted earlier (we found 4 or 5 to be a good amount), then chain up to the input of the toe section of the strip. Repeat this process at the end of the toe-section piece of NeoPixel strip.



With all the data lines connected, now it's time to wire up power and ground. Use 30 gauge wire for this, and make sure there is ample slack in these wires (they should bow out a bit from the strip), as this is what provides the lateral bending ability. If these wires are too short they WILL break when you bend the shoe at the toe! Tweezers really help get this job done.

Now the shoe can bend without fatiguing the flex PCB. Solder longer-than-nessessary wires to the input end of the strip and wire them up with alligator clips to GEMMA and run the strandtest NeoPixel example to ensure that your new custom strip is working. While it's on, bend it around a bit to see if you have any precarious connections. Inspect the back of your circuit and trim any stray strands or areas that could short out because of a long wire or blob of solder. Attention to detail at this step is critical, or your shoes will break sooner than you'd like!

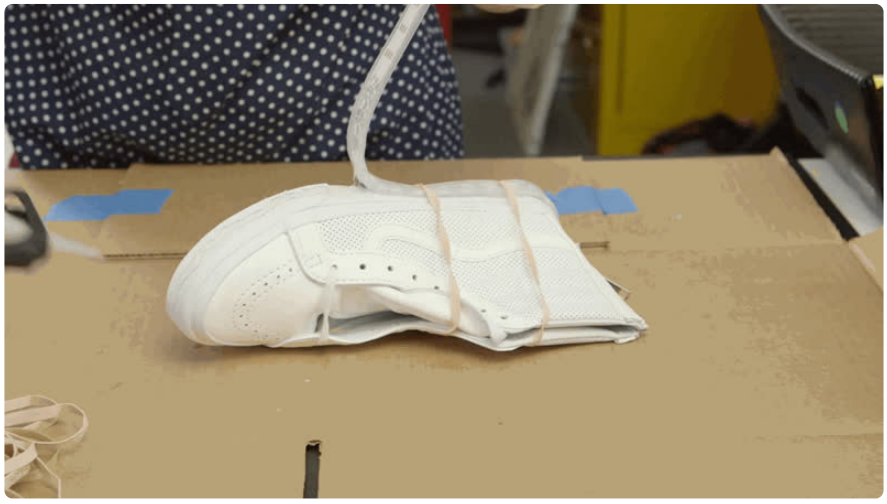


Cut the sheathing down the back to fit the strip back inside.

Glue to Shoes

Permatex 66B silicone adhesive is the only stuff we've found that sticks to the sheathing of the NeoPixel strips. This time around we found that the Powerbead can makes application a lot easier on the hands. You should do a test on your shoes to make sure it sticks to them too— wipe the surface of the soles with a alcohol first to remove any dust or residue. Likewise wipe the NeoPixel strip sheathing— clean surfaces adhere much better!

Wear gloves, protect your work surface, and work in a well-ventilated area— this stuff smells like salt & vinegar chips but after a while will make you dizzy from all the brain cells it's killing if you don't have proper ventilation.

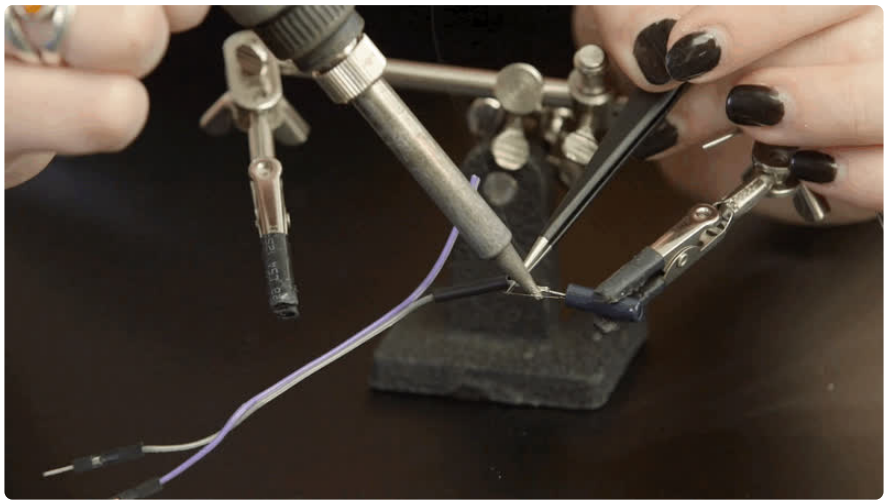


Start with the inststep (and the input end of your NeoPixel strip), and carefully apply glue one section at a time. Use rubber bands to secure the strip to the shoe.

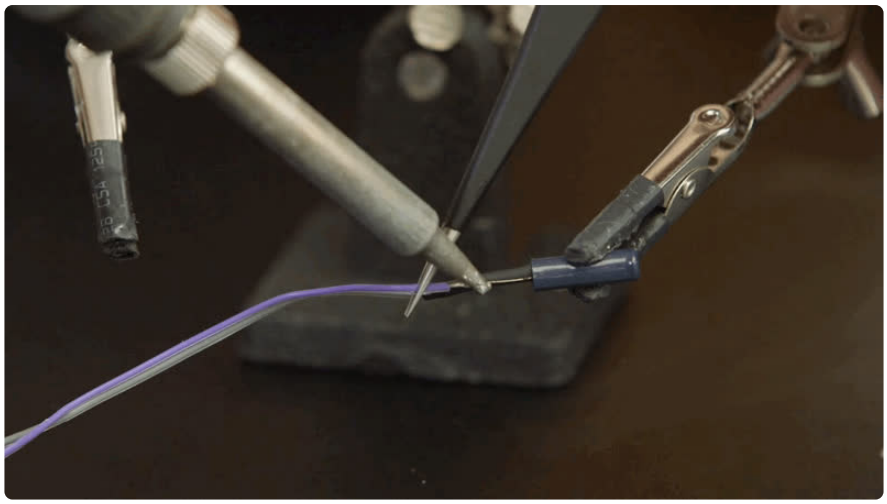
When you get to the end, fill the end of the sheathing with glue to keep moisture and dirt out.

Use a clean cloth and rubbing alcohol to clean up any excess glue, then let dry overnight.

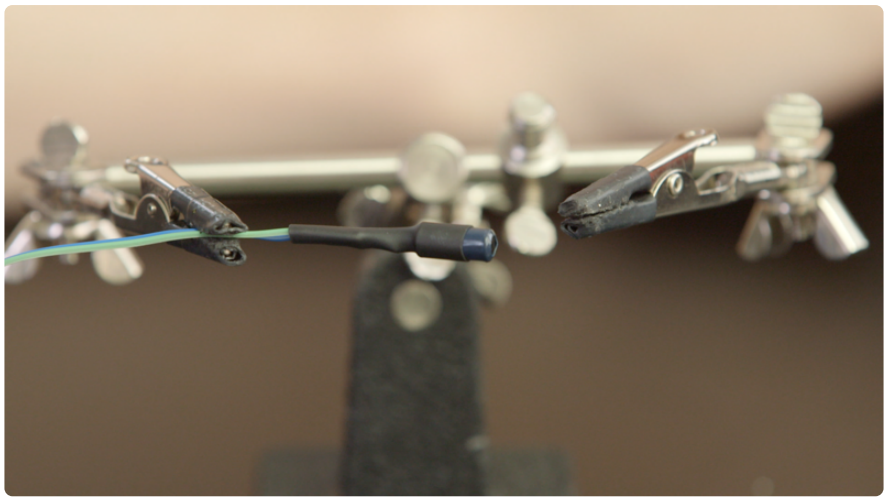
Solder GEMMA & Sensor



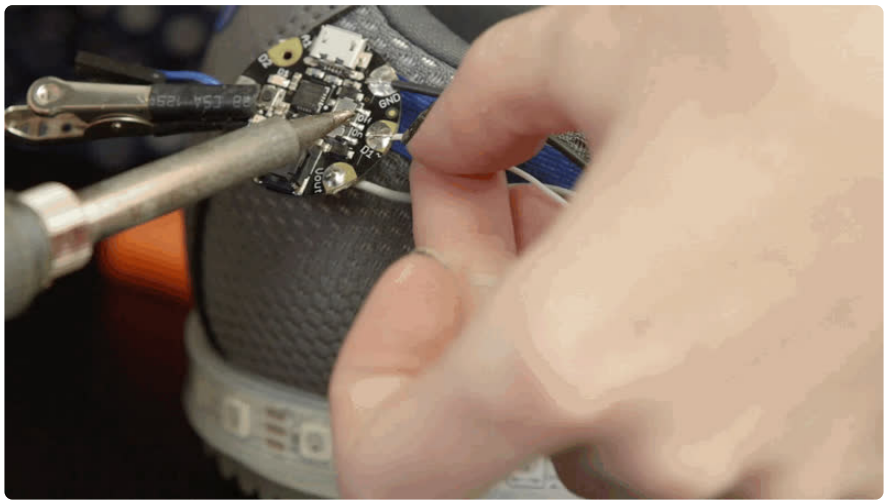
To prepare the vibration sensor, we'll solder wires to the leads and protect the whole thing with heat shrink tubing. If you want to try out different versions of the vibration sensor, use wires with connectors at the ends such as our premium breadboarding wires, or just use more silicone coated stranded wire.



First strip and tin one wire and slide a small piece of heat shrink tubing onto it. Tin the larger, center wire of the vibration sensor and then reheat the wire/sensor joint to melt the solder between the two. Slide the heat shrink tubing over the entirety of the exposed metal and shrink it with a heat gun (or lighter in a pinch).



Repeat with the outer wire, which is much skinnier, than slide a larger piece of heat shrink tubing over the whole sensor, extending down the wire a bit for added stability and moisture resistance.



Solder the sensors wires to D0 and GND on GEMMA (or use matching plugging wires if you're making swappable sensors). The sensor is not polar, meaning it doesn't matter which wire goes to ground and which to D0.

Position your GEMMA approximately where it will go on your shoe, and trim the NeoPixel strip input wires to length, then solder them up to GEMMA in the same configuration you used to test earlier:

- GEMMA Vout -> NeoPixel strip 5v
- GEMMA D1 -> NeoPixel strip data
- GEMMA GND -> NeoPixel strip ground

Stitch GEMMA to your shoe using a needle and thread, and pliers if necessary to pierce through tough materials.

Arduino Code

The Arduino code presented below works well on GEMMA: v2.. But if you have an M0 board you must use the CircuitPython code on the next page of this guide, no Arduino IDE required!

Plug in GEMMA to your computer with a USB cable and load up the following sensor-activated animation code written by Phillip Burgess (OG Firewalker creator extraordinaire):

```
// SPDX-FileCopyrightText: 2018 Mikey Sklar for Adafruit Industries
//
// SPDX-License-Identifier: MIT

/*-------------------------------------------------------------------------
  Gemma "Firewalker Lite" sneakers sketch.
  Uses the following Adafruit parts (X2 for two shoes):
```

```
  - Gemma 3V microcontroller (adafruit.com/product/1222 or 2470)
  - 150 mAh LiPoly battery (#1317) or larger
  - Medium vibration sensor switch (#2384)
  - 60/m NeoPixel RGB LED strip (#1138 or #1461)
  - LiPoly charger such as #1304 (only one is required, unless you want
    to charge both shoes at the same time)

  Needs Adafruit_NeoPixel library: github.com/adafruit/Adafruit_NeoPixel

  THIS CODE USES FEATURES SPECIFIC TO THE GEMMA MICROCONTROLLER BOARD AND
  WILL NOT COMPILE OR RUN ON MOST OTHERS.  OK on basic Trinket but NOT
  Pro Trinket nor anything else.  VERY specific to the Gemma!

  Adafruit invests time and resources providing this open source code,
  please support Adafruit and open-source hardware by purchasing
  products from Adafruit!

  Written by Phil Burgess / Paint Your Dragon for Adafruit Industries.
  MIT license, all text above must be included in any redistribution.
  See 'COPYING' file for additional notes.
  ------------------------------------------------------------------------*/

#include <Arduino.h>
#include <Adafruit_NeoPixel.h>
#include <avr/power.h>
#include <avr/sleep.h>

// CONFIGURABLE STUFF --------------------------------------------------------

#define NUM_LEDS      40 // Actual number of LEDs in NeoPixel strip
#define CIRCUMFERENCE 40 // Shoe circumference, in pixels, may be > NUM_LEDS
#define FPS           50 // Animation frames per second
#define LED_DATA_PIN   1 // NeoPixels are connected here
#define MOTION_PIN     0 // Vibration switch from here to GND
// CIRCUMFERENCE is distinct from NUM_LEDS to allow a gap (if desired) in
// LED strip around perimeter of shoe (might not want any on inside egde)
// so animation is spatially coherent (doesn't jump across gap, but moves
// as if pixels were in that space).  CIRCUMFERENCE can be equal or larger
// than NUM_LEDS, but not less.

// The following features are OPTIONAL and require ADDITIONAL COMPONENTS.
// Only ONE of these may be enabled due to limited pins on Gemma:

// BATTERY LEVEL graph on power-up: requires two resistors of same value,
// 10K or higher, connected to pin D2 -- one goes to Vout pin, other to GND.
//#define BATT_LVL_PIN  2 // Un-comment this line to enable battery level.
// Pin number cannot be changed, this is the only AREF pin on ATtiny85.
// Empty and full thresholds (millivolts) used for battery level display:
#define BATT_MIN_MV 3350 // Some headroom over battery cutoff near 2.9V
#define BATT_MAX_MV 4000 // And little below fresh-charged battery near 4.1V
// Graph works only on battery power; USB power will always show 100% full.

// POWER DOWN NeoPixels when idle to prolong battery: requires P-channel
// power MOSFET + 220 Ohm resistor, is a 'high side' switch to NeoPixel +V.
// DO NOT do this w/N-channel on GND side, could DESTROY strip!
//#define LED_PWR_PIN   2 // Un-comment this for NeoPixel power-down.
// Could be moved to other pin on Trinket to allow both options together.

// GLOBAL STUFF --------------------------------------------------------------

Adafruit_NeoPixel    strip(NUM_LEDS, LED_DATA_PIN);
void                 sleep(void); // Power-down function
extern const uint8_t gamma[];     // Table at the bottom of this code

// INTERMISSION explaining the animation code.  This works procedurally --
// using mathematical functions -- rather than moving discrete pixels left
// or right incrementally.  This sometimes confuses people because they go
// looking for some "setpixel(x+1, color)" type of code and can't find it.
// Also makes extensive use of fixed-point math, where discrete integer
```

```
// values are used to represent fractions (0.0 to 1.0) without relying on
// floating-point math, which just wouldn't even fit in Gemma's limited
// code space, RAM or speed.  The reason the animation's done this way is
// to produce smoother, more organic motion...when pixels are the smallest
// discrete unit, animation tends to have a stuttery, 1980s quality to it.
// We can do better!
// Picture the perimeter of the shoe in two different coordinate spaces:
// One is "pixel space," each pixel spans exactly one integer unit, from
// zero to N-1 when there are N pixels.  This is how NeoPixels are normally
// addressed.  Now, overlaid on this, imagine another coordinate space,
// spanning the same physical width (the perimeter of the shoe, or length
// of the LED strip), but this one has 256 discrete steps (8 bits)...finer
// resolution than the pixel steps...and we do most of the math using
// these units rather than pixel units.  It's then possible to move things
// by fractions of a pixel, but render each whole pixel by taking a sample
// at its approximate center in the alternate coordinate space.
// More explanation further in the code.
//
// |Pixel|Pixel|Pixel|    ...    |Pixel|Pixel|Pixel|<- end of strip
// |  0  |  1  |  2  |           |  3  |  4  | N-1 |
// |0...                 ...               ...255|<- fixed-point space
//
// So, inspired by the mothership in Close Encounters of the Third Kind,
// the animation in this sketch is a series of waves moving around the
// perimeter and interacting as they cross.  They're triangle waves,
// height proportional to LED brightness, determined by the time since
// motion was last detected.
//
//   <- /\       /\ -> <- /\          Pretend these are triangle waves
// ____/  \_____/  \_____/  \____  <- moving in 8-bit coordinate space.

struct {
  uint8_t center;    // Center point of wave in fixed-point space (0 - 255)
  int8_t  speed;     // Distance to move between frames (-128 - +127)
  uint8_t width;     // Width from peak to bottom of triangle wave (0 - 128)
  uint8_t hue;       // Current wave hue (color) see comments later
  uint8_t hueTarget; // Final hue we're aiming for
  uint8_t r, g, b;   // LED RGB color calculated from hue
} wave[] = {
  { 0,  3, 60 },     // Gemma can animate 3 of these on 40 LEDs at 50 FPS
  { 0, -5, 45 },     // More LEDs and/or more waves will need lower FPS
  { 0,  7, 30 }
};
// Note that the speeds of each wave are different prime numbers.  This
// avoids repetition as the waves move around the perimeter...if they were
// even numbers or multiples of each other, there'd be obvious repetition
// in the pattern of motion...beat frequencies.
#define N_WAVES (sizeof(wave) / sizeof(wave[0]))

// ONE-TIME INITIALIZATION --------------------------------------------------

void setup() {
#if defined(__AVR_ATtiny85__) && (F_CPU == 16000000L)
  clock_prescale_set(clock_div_1); // Allow 16 MHz Trinket too
#endif
#ifdef POWER_PIN
  pinMode(POWER_PIN, OUTPUT);
  digitalWrite(POWER_PIN, LOW);    // Power-on LED strip
#endif
  strip.begin();                   // Allocate NeoPixel buffer
  strip.clear();                   // Make sure strip is clear
  strip.show();                    // before measuring battery

#ifdef BATT_LVL_PIN
  // Battery monitoring code does some low-level Gemma-specific stuff...
  int      i, prev;
  uint8_t  count;
  uint16_t mV;
```

```
  pinMode(BATT_LVL_PIN, INPUT);    // No pullup

  // Select AREF (PB0) voltage reference + Bandgap (1.8V) input
  ADMUX  = _BV(REFS0) | _BV(MUX3) | _BV(MUX2);   // AREF, Bandgap input
  ADCSRA = _BV(ADEN)  |                          // Enable ADC
           _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0); // 1/128 prescale (125 KHz)
  delay(1); // Allow 1 ms settling time as per datasheet
  // Bandgap readings may still be settling.  Repeated readings are
  // taken until four concurrent readings stabilize within 5 mV.
  for(prev=9999, count=0; count<4; ) {
    for(ADCSRA |= _BV(ADSC); ADCSRA & _BV(ADSC); ); // Start, await ADC conv
    i  = ADC;                                       // Result
    mV = i ? (1100L * 1023 / i) : 0;                // Scale to millivolts
    if(abs((int)mV - prev) <= 5) count++;   // +1 stable reading
    else                         count = 0; // too much change, start over
    prev = mV;
  }
  ADCSRA = 0; // ADC off
  mV *= 2;    // Because 50% resistor voltage divider (to work w/3.3V MCU)

  uint8_t  lvl = (mV >= BATT_MAX_MV) ? NUM_LEDS : // Full (or nearly)
                 (mV <= BATT_MIN_MV) ?        1 : // Drained
                 1 + ((mV - BATT_MIN_MV) * NUM_LEDS + (NUM_LEDS / 2)) /
                 (BATT_MAX_MV - BATT_MIN_MV + 1); // # LEDs lit (1-NUM_LEDS)
  for(uint8_t i=0; i<lvl; i++) {                  // Each LED to batt level
    uint8_t g = (i * 5 + 2) / NUM_LEDS;           // Red to green
    strip.setPixelColor(i, 4-g, g, 0);
    strip.show();                                 // Animate a bit
    delay(250 / NUM_LEDS);
  }
  delay(1500);                                    // Hold last state a moment
  strip.clear();                                  // Then clear strip
  strip.show();
  randomSeed(mV);
#else
  randomSeed(analogRead(2));
#endif // BATT_LVL_PIN

  // Assign random starting colors to waves
  for(uint8_t w=0; w<N_WAVES; w++) {
    wave[w].hue = wave[w].hueTarget = 90 + random(90);
    wave[w].r = h2rgb(wave[w].hue - 30);
    wave[w].g = h2rgb(wave[w].hue);
    wave[w].b = h2rgb(wave[w].hue + 30);
  }

  // Configure motion pin for change detect & interrupt
  pinMode(MOTION_PIN, INPUT_PULLUP);
  PCMSK = _BV(MOTION_PIN); // Pin change interrupt mask
  GIFR  = 0xFF;            // Clear interrupt flags
  // Interrupt is not actually enabled yet, that's in sleep function...

  sleep(); // Sleep until motion is detected
}

// MAIN LOOP ---------------------------------------------------------------

uint32_t prevFrameTime = 0L;    // Used for animation timing
uint8_t  brightness    = 0;     // Current wave height
boolean  rampingUp     = false; // If true, brightness is increasing

void loop() {
  uint32_t t;
  uint16_t r, g, b, m, n;
  uint8_t  i, x, w, d1, d2, y;

  // Pause until suitable interval since prior frame has elapsed.
  // This is preferable to delay(), as the time to render each frame
  // can vary.
```

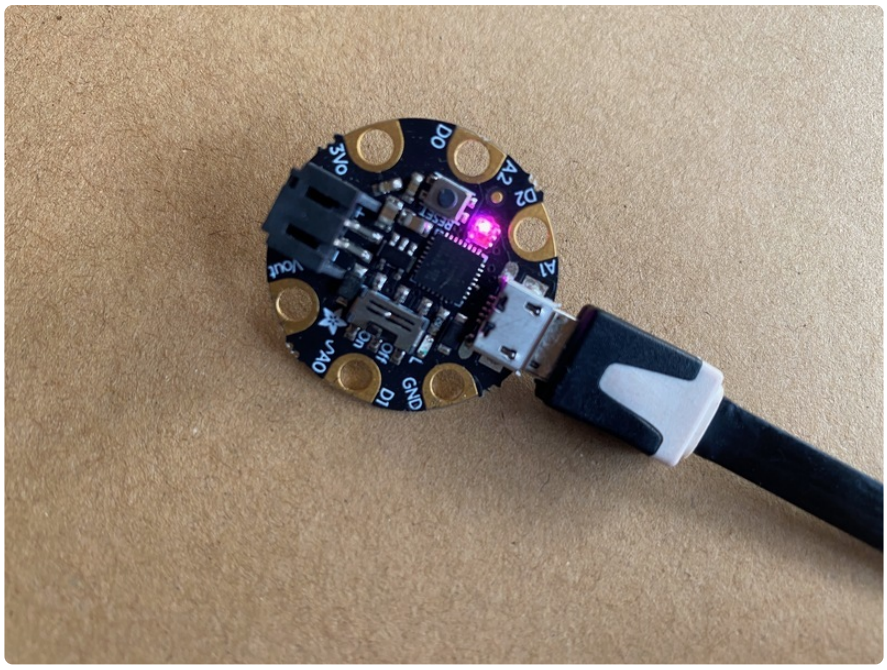
```
while(((t = micros()) - prevFrameTime) < (1000000L / FPS));

// Immediately show results calculated on -prior- pass,
// so frame-to-frame timing is consistent.  Then render next frame.
strip.show();
prevFrameTime = t;     // Save frame update time for next pass

if(GIFR & _BV(PCIF)) { // Pin change detected?
  rampingUp = true;    // Set brightness-ramping flag
  GIFR      = 0xFF;    // Clear interrupt masks
}

// Okay, here's where the animation starts to happen...

// First, check the 'rampingUp' flag.  If set, this indicates that
// the vibration switch was activated recently, and the LEDs should
// increase in brightness.  If clear, the LEDs ramp down.
if(rampingUp) {
  // But it's not just a straight shot that it ramps up.  This is a
  // low-pass filter...it makes the brightness value decelerate as it
  // approaches a target (200 in this case).  207 is used here because
  // integers round down on division and we'd never reach the target;
  // it's an ersatz ceil() function: ((199*7)+200+7)/8 = 200;
  brightness = ((brightness * 7) + 207) / 8;
  // Once max brightness is reached, switch off the rampingUp flag.
  if(brightness >= 200) rampingUp = false;
} else {
  // If not ramping up, we're ramping down.  This too uses a low-pass
  // filter so it eases out, but with different constants so it's a
  // slower fade.  Also, no addition here because we want it rounding
  // down toward zero...
  if(!(brightness = (brightness * 15) / 16)) { // Hit zero?
    sleep(); // Turn off animation
    return;  // Start over at top of loop() on wake
  }
}

// Wave positions and colors are updated...
for(w=0; w<N_WAVES; w++) {
  wave[w].center += wave[w].speed; // Move wave; wraps around ends, is OK!
  if(wave[w].hue == wave[w].hueTarget) { // Hue not currently changing?
    // There's a tiny random chance of picking a new hue...
    if(!random(FPS * 4)) {
      // Within 1/3 color wheel
      wave[w].hueTarget = random(wave[w].hue - 30, wave[w].hue + 30);
    }
  } else { // This wave's hue is currently shifting...
    if(wave[w].hue < wave[w].hueTarget) wave[w].hue++; // Move up or
    else                                wave[w].hue--; // down as needed
    if(wave[w].hue == wave[w].hueTarget) { // Reached destination?
      wave[w].hue = 90 + wave[w].hue % 90; // Clamp to 90-180 range
      wave[w].hueTarget = wave[w].hue;     // Copy to target
    }
    wave[w].r = h2rgb(wave[w].hue - 30);
    wave[w].g = h2rgb(wave[w].hue);
    wave[w].b = h2rgb(wave[w].hue + 30);
  }
}

// Now render the LED strip using the current brightness & wave states

for(i=0; i<NUM_LEDS; i++) { // Each LED in strip is visited just once...

  // Transform 'i' (LED number in pixel space) to the equivalent point
  // in 8-bit fixed-point space (0-255).  "* 256" because that would be
  // the start of the (N+1)th pixel.  "+ 127" to get pixel center.
  x = (i * 256 + 127) / CIRCUMFERENCE;

  r = g = b = 0; // LED assumed off, but wave colors will add up here
```

```
    for(w=0; w<N_WAVES; w++) { // For each item in wave[] array...

      // Calculate distance from pixel center to wave center point,
      // using both signed and unsigned 8-bit integers...
      d1 = abs((int8_t)x  - (int8_t)wave[w].center);
      d2 = abs((uint8_t)x - (uint8_t)wave[w].center);
      // Then take the lesser of the two, resulting in a distance (0-128)
      // that 'wraps around' the ends of the strip as necessary...it's a
      // contiguous ring, and waves can move smoothly across the gap.
      if(d2 < d1) d1 = d2;       // d1 is pixel-to-wave-center distance
      if(d1 < wave[w].width) {   // Is distance within wave's influence?
        d2 = wave[w].width - d1; // d2 is opposite; distance to wave's end

        // d2 distance, relative to wave width, is then proportional to the
        // wave's brightness at this pixel (basic linear y=mx+b stuff).
        // Normally this would require a fraction -- floating-point math --
        // but by reordering the operations we can get the same result with
        // integers -- fixed-point math -- that's why brightness is cast
        // here to a 16-bit type; the interim result of the multiplication
        // is a big integer that's then divided by wave width (back to an
        // 8-bit value) to yield the pixel's brightness.  This massive wall
        // of comments is basically to explain that fixed-point math is
        // faster and less resource-intensive on processors with limited
        // capabilities.  Topic for another Adafruit Learning System guide?
        y = (uint16_t)brightness * d2 / wave[w].width; // 0 to 200

        // y is a brightness scale value -- proportional to, but not
        // exactly equal to, the resulting RGB value.  Values from 0-127
        // represent a color ramp from black to the wave's assigned RGB
        // color.  Values from 128-255 ramp from the RGB color to white.
        // It's by design that y only goes to 200...white peaks then occur
        // only when certain waves overlap.
        if(y < 128) { // Fade black to RGB color
          // In HSV colorspace, this would be tweaking 'value'
          n  = (uint16_t)y * 2 + 1;  // 1-256
          r += (wave[w].r * n) >> 8; // More fixed-point math
          g += (wave[w].g * n) >> 8; // Wave color is scaled by 'n'
          b += (wave[w].b * n) >> 8; // >>8 is equiv to /256
        } else {      // Fade RGB color to white
          // In HSV colorspace, this would be tweaking 'saturation'
          n  = (uint16_t)(y - 128) * 2; // 0-255 affects white level
          m  = 256 * n;
          n  = 256 - n;                 // 1-256 affects RGB level
          r += (m + wave[w].r * n) >> 8;
          g += (m + wave[w].g * n) >> 8;
          b += (m + wave[w].b * n) >> 8;
        }
      }
    }

    // r,g,b are 16-bit types that accumulate brightness from all waves
    // that affect this pixel; may exceed 255.  Now clip to 0-255 range:
    if(r > 255) r = 255;
    if(g > 255) g = 255;
    if(b > 255) b = 255;
    // Store resulting RGB value and we're done with this pixel!
    strip.setPixelColor(i, r, g, b);
  }

  // Once rendering is complete, a second pass is made through pixel data
  // applying gamma correction, for more perceptually linear colors.
  // https://learn.adafruit.com/led-tricks-gamma-correction
  uint8_t *pixels = strip.getPixels(); // Pointer to LED strip buffer
  for(i=0; i<NUM_LEDS*3; i++) pixels[i] = pgm_read_byte(&gamma[pixels[i]]);
}

// SLEEP/WAKE CODE is very Gemma-specific  ---------------------------------
```

```
void sleep() {
  strip.clear();                        // Clear pixel buffer
#ifdef POWER_PIN
  pinMode(LED_DATA_PIN, INPUT);         // Avoid parasitic power to strip
  digitalWrite(POWER_PIN, HIGH);        // Cut power to pixels
#else
  strip.show();                         // Turn off LEDs
#endif // POWER_PIN
  power_all_disable();                  // Peripherals ALL OFF
  GIMSK = _BV(PCIE);                    // Allow pin-change interrupt only
  set_sleep_mode(SLEEP_MODE_PWR_DOWN); // Deepest sleep mode
  sleep_enable();
  interrupts();                         // Needed for pin-change wake
  sleep_mode();                         // Power down (stops here)
  //                                    ** RESUMES HERE ON WAKE **
  GIMSK = 0;                            // Clear global interrupt mask
  // Pin change when awake is done by polling GIFR register, not interrupt
#ifdef POWER_PIN
  digitalWrite(POWER_PIN, LOW);         // Power-up LEDs
  pinMode(LED_DATA_PIN, OUTPUT);
  strip.show();                         // Clear any startup garbage
#endif
  power_timer0_enable();                // Used by micros()
  // Remaining peripherals (ADC, Timer1, etc) are NOT re-enabled, as they're
  // not used elsewhere in the sketch.  If adding features, you might need
  // to re-enable some/all peripherals here.
  rampingUp = true;
}

EMPTY_INTERRUPT(PCINT0_vect); // Pin change (does nothing, but required)

// COLOR-HANDLING CODE -----------------------------------------------------

// A full HSV-to-RGB function wasn't required by sketch, just needed limited
// hue-to-RGB.  There are 90 distinct hues (0-89) around color wheel (to
// allow 4-bit table entries).  This function gets called three times (for
// R,G,B, with different offsets relative to hue) to produce a fully-
// saturated color.  Was a little more compact than a full HSV function.

static const uint8_t PROGMEM hueTable[45] = {
 0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xED,0xCB,0xA9,0x87,0x65,0x43,0x21,
 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
 0x12,0x34,0x56,0x78,0x9A,0xBC,0xDE,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF
};

uint8_t h2rgb(uint8_t hue) {
  hue %= 90;
  uint8_t h = pgm_read_byte(&hueTable[hue >> 1]);
  return ((hue & 1) ? (h & 15) : (h >> 4)) * 17;
}

// Gamma-correction table (see earlier comments).  It's big and ugly
// and would interrupt trying to read the code, so I put it down here.
const uint8_t gamma[] PROGMEM = {
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,
    1,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,
    2,  3,  3,  3,  3,  3,  3,  3,  4,  4,  4,  4,  4,  5,  5,  5,
    5,  6,  6,  6,  6,  7,  7,  7,  7,  8,  8,  8,  9,  9,  9, 10,
   10, 10, 11, 11, 11, 12, 12, 13, 13, 13, 14, 14, 15, 15, 16, 16,
   17, 17, 18, 18, 19, 19, 20, 20, 21, 21, 22, 22, 23, 24, 24, 25,
   25, 26, 27, 27, 28, 29, 29, 30, 31, 32, 32, 33, 34, 35, 35, 36,
   37, 38, 39, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 50,
   51, 52, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 66, 67, 68,
   69, 70, 72, 73, 74, 75, 77, 78, 79, 81, 82, 83, 85, 86, 87, 89,
   90, 92, 93, 95, 96, 98, 99,101,102,104,105,107,109,110,112,114,
  115,117,119,120,122,124,126,127,129,131,133,135,137,138,140,142,
  144,146,148,150,152,154,156,158,160,162,164,167,169,171,173,175,
```

```
  177,180,182,184,186,189,191,193,196,198,200,203,205,208,210,213,
  215,218,220,223,225,228,231,233,236,239,241,244,247,249,252,255 };
```

Arduino Code for Gemma M0



The Arduino code presented below works well on Gemma M0. If you have an older Gemma v2 board, use the code on the previous page.

Plug in your Gemma M0 into your computer with a USB cable and load up the following sensor-activated animation code written by Phillip Burgess (OG Firewalker creator extraordinaire):

```
// SPDX-FileCopyrightText: 2020 Phillip Burgess for Adafruit Industries
//
// SPDX-License-Identifier: MIT

// 'Firewalker' LED sneakers sketch for Adafruit NeoPixels by Phillip Burgess

#include <Adafruit_NeoPixel.h>

const uint8_t gamma1[] PROGMEM = { // Gamma correction table for LED brightness
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,
    1,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,
    2,  3,  3,  3,  3,  3,  3,  3,  4,  4,  4,  4,  4,  5,  5,  5,
    5,  6,  6,  6,  6,  7,  7,  7,  7,  8,  8,  8,  9,  9,  9, 10,
   10, 10, 11, 11, 11, 12, 12, 13, 13, 13, 14, 14, 15, 15, 16, 16,
   17, 17, 18, 18, 19, 19, 20, 20, 21, 21, 22, 22, 23, 24, 24, 25,
   25, 26, 27, 27, 28, 29, 29, 30, 31, 32, 32, 33, 34, 35, 35, 36,
   37, 38, 39, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 50,
   51, 52, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 66, 67, 68,
   69, 70, 72, 73, 74, 75, 77, 78, 79, 81, 82, 83, 85, 86, 87, 89,
   90, 92, 93, 95, 96, 98, 99,101,102,104,105,107,109,110,112,114,
  115,117,119,120,122,124,126,127,129,131,133,135,137,138,140,142,
  144,146,148,150,152,154,156,158,160,162,164,167,169,171,173,175,
```

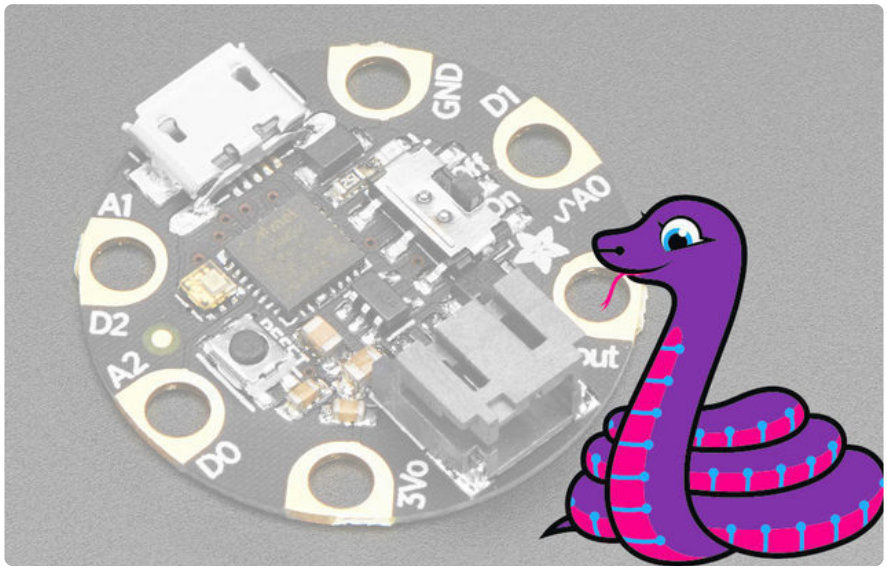
```
  177,180,182,184,186,189,191,193,196,198,200,203,205,208,210,213,
  215,218,220,223,225,228,231,233,236,239,241,244,247,249,252,255 };

// LEDs go around the full perimeter of the shoe sole, but the step animation
// is mirrored on both the inside and outside faces, while the strip doesn't
// necessarily start and end at the heel or toe.  These constants help configure
// the strip and shoe sizes, and the positions of the front- and rear-most LEDs.
// Becky's shoes: 39 LEDs total, 20 LEDs long, LED #5 at back.
// Phil's shoes: 43 LEDs total, 22 LEDs long, LED #6 at back.
#define N_LEDS        39 // TOTAL number of LEDs in strip
#define SHOE_LEN_LEDS 20 // Number of LEDs down ONE SIDE of shoe
#define SHOE_LED_BACK  5 // Index of REAR-MOST LED on shoe
#define STEP_PIN      A2 // Analog input for footstep
#define LED_PIN       A0 // NeoPixel strip is connected here
#define MAXSTEPS       3 // Process (up to) this many concurrent steps

Adafruit_NeoPixel strip = Adafruit_NeoPixel(N_LEDS, LED_PIN, NEO_GRB + NEO_KHZ800);

// The readings from the sensors are usually around 250-350 when not being pressed,
// then dip below 100 when the heel is standing on it (for Phil's shoes; Becky's
// don't dip quite as low because she's smaller).
#define STEP_TRIGGER    150  // Reading must be below this to trigger step
#define STEP_HYSTERESIS 200  // After trigger, must return to this level

int
  stepMag[MAXSTEPS],  // Magnitude of steps
  stepX[MAXSTEPS],    // Position of 'step wave' along strip
  mag[SHOE_LEN_LEDS], // Brightness buffer (one side of shoe)
  stepFiltered,       // Current filtered pressure reading
  stepCount,          // Number of 'frames' current step has lasted
  stepMin;            // Minimum reading during current step
uint8_t
  stepNum = 0,        // Current step number in stepMag/stepX tables
  dup[SHOE_LEN_LEDS]; // Inside/outside copy indexes
boolean
  stepping  = false;  // If set, step was triggered, waiting to release


void setup() {
  pinMode(9, INPUT_PULLUP); // Set internal pullup resistor for sensor pin
  // As previously mentioned, the step animation is mirrored on the inside and
  // outside faces of the shoe.  To avoid a bunch of math and offsets later, the
  // 'dup' array indicates where each pixel on the outside face of the shoe should
  // be copied on the inside.  (255 = don't copy, as on front- or rear-most LEDs).
  // Later, the colors for the outside face of the shoe are calculated and then get
  // copied to the appropriate positions on the inside face.
  memset(dup, 255, sizeof(dup));
  int8_t a, b;
  for(a=1               , b=SHOE_LED_BACK-1            ; b>=0    ;) dup[a++] = b--;
  for(a=SHOE_LEN_LEDS-2, b=SHOE_LED_BACK+SHOE_LEN_LEDS; b<N_LEDS;) dup[a--] = b++;

  // Clear step magnitude and position buffers
  memset(stepMag, 0, sizeof(stepMag));
  memset(stepX  , 0, sizeof(stepX));
  strip.begin();
  stepFiltered = analogRead(STEP_PIN); // Initial input
}

void loop() {
  uint8_t i, j;

  // Read analog input, with a little noise filtering
  //stepFiltered = ((stepFiltered * 3) + analogRead(STEP_PIN)) >> 2;
    stepFiltered = (((stepFiltered * 3) - 100) + analogRead(STEP_PIN)) >> 2;

  // The strip doesn't simply display the current pressure reading.  Instead,
  // there's a bit of an animated flourish from heel to toe.  This takes time,
  // and during quick foot-tapping there could be multiple step animations
  // 'in flight,' so a short list is kept.
```

```
if(stepping) { // If a step was previously triggered...
  if(stepFiltered >= STEP_HYSTERESIS) { // Has step let up?
    stepping = false;                   // Yep! Stop monitoring.
    // Add new step to the step list (may be multiple in flight)
    stepMag[stepNum] = (STEP_HYSTERESIS - stepMin) * 6; // Step intensity
    stepX[stepNum]   = -80; // Position starts behind heel, moves forward
    if(++stepNum >= MAXSTEPS) stepNum = 0; // If many, overwrite oldest
  } else if(stepFiltered < stepMin) stepMin = stepFiltered; // Track min val
} else if(stepFiltered < STEP_TRIGGER) { // No step yet; watch for trigger
  stepping = true;         // Got one!
  stepMin  = stepFiltered; // Note initial value
}

// Render a 'brightness map' for all steps in flight.  It's like
// a grayscale image; there's no color yet, just intensities.
int mx1, px1, px2, m;
memset(mag, 0, sizeof(mag));    // Clear magnitude buffer
for(i=0; i<MAXSTEPS; i++) {     // For each step...
  if(stepMag[i] <= 0) continue; // Skip if inactive
  for(j=0; j<SHOE_LEN_LEDS; j++) { // For each LED...
    // Each step has sort of a 'wave' that's part of the animation,
    // moving from heel to toe.  The wave position has sub-pixel
    // resolution (4X), and is up to 80 units (20 pixels) long.
    mx1 = (j << 2) - stepX[i]; // Position of LED along wave
    if((mx1 <= 0) || (mx1 >= 80)) continue; // Out of range
    if(mx1 > 64) { // Rising edge of wave; ramp up fast (4 px)
      m = ((long)stepMag[i] * (long)(80 - mx1)) >> 4;
    } else { // Falling edge of wave; fade slow (16 px)
      m = ((long)stepMag[i] * (long)mx1) >> 6;
    }
    mag[j] += m; // Add magnitude to buffered sum
  }
  stepX[i]++; // Update position of step wave
  if(stepX[i] >= (80 + (SHOE_LEN_LEDS << 2)))
    stepMag[i] = 0; // Off end; disable step wave
  else
    stepMag[i] = ((long)stepMag[i] * 127L) >> 7; // Fade
}

// For a little visual interest, some 'sparkle' is added.
// The cumulative step magnitude is added to one pixel at random.
long sum = 0;
for(i=0; i<MAXSTEPS; i++) sum += stepMag[i];
if(sum > 0) {
  i = random(SHOE_LEN_LEDS);
  mag[i] += sum / 4;
}

// Now the grayscale magnitude buffer is remapped to color for the LEDs.
// The code below uses a blackbody palette, which fades from white to yellow
// to red to black.  The goal here was specifically a "walking on fire"
// aesthetic, so the usual ostentatious rainbow of hues seen in most LED
// projects is purposefully skipped in favor of a more plain effect.
uint8_t r, g, b;
int     level;
for(i=0; i<SHOE_LEN_LEDS; i++) { // For each LED on one side...
  level = mag[i];                // Pixel magnitude (brightness)
  if(level < 255) {              // 0-254 = black to red-1
    r = pgm_read_byte(&gamma1[level]);
    g = b = 0;
  } else if(level < 510) {       // 255-509 = red to yellow-1
    r = 255;
    g = pgm_read_byte(&gamma1[level - 255]);
    b = 0;
  } else if(level < 765) {       // 510-764 = yellow to white-1
    r = g = 255;
    b = pgm_read_byte(&gamma1[level - 510]);
  } else {                       // 765+ = white
    r = g = b = 255;
```

```
    }
    // Set R/G/B color along outside of shoe
    strip.setPixelColor(i+SHOE_LED_BACK, r, g, b);
    // Pixels along inside are funny...
    j = dup[i];
    if(j < 255) strip.setPixelColor(j, r, g, b);
  }

  strip.show();
  delayMicroseconds(1500);
}
```

CircuitPython Code

The CircuitPython code below runs very slowly!



GEMMA M0 boards can run **CircuitPython** — a different approach to programming compared to Arduino sketches. In fact, **CircuitPython comes factory pre-loaded on GEMMA M0**. If you've overwritten it with an Arduino sketch, or just want to learn the basics of setting up and using CircuitPython, this is explained in the **Adafruit GEMMA M0 guide** (https://adafru.it/z1B).

These directions are specific to the "M0" GEMMA board. The original GEMMA with an 8-bit AVR microcontroller doesn't run CircuitPython...for those boards, use the Arduino sketch on the "Arduino code" page of this guide.

Below is CircuitPython code that works similarly (though not exactly the same) as the Arduino sketch shown on a prior page. To use this, plug the GEMMA M0 into USB...it should show up on your computer as a small **flash drive**...then edit the file "**main.py**" with your text editor of choice. Select and copy the code below and paste it into that file, **entirely replacing its contents** (don't mix it in with lingering bits of old code).

When you save the file, the code should **start running almost immediately** (if not, see notes at the bottom of this page).

If GEMMA M0 doesn't show up as a drive, follow the GEMMA M0 guide link above to prepare the board for CircuitPython.

```
# SPDX-FileCopyrightText: 2018 Phillip Burgess for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# Gemma "Firewalker Lite" sneakers
#  - Uses the following Adafruit parts (X2 for two shoes):
#    * Gemma M0 3V microcontroller (#3501)
#    * 150 mAh LiPoly battery (#1317) or larger
#    * Medium vibration sensor switch (#2384)
#    * 60/m NeoPixel RGB LED strip (#1138 or #1461)
#    * LiPoly charger such as #1304
#
# - originally written by Phil Burgess for Gemma using Arduino
#   * https://learn.adafruit.com/gemma-led-sneakers

import board
import digitalio
import neopixel

try:
    import urandom as random
except ImportError:
    import random

# Declare a NeoPixel object on led_pin with num_leds as pixels
# No auto-write.
led_pin = board.D1  # Which pin your pixels are connected to
num_leds = 40  # How many LEDs you have
circumference = 40  # Shoe circumference, in pixels, may be > NUM_LEDS
frames_per_second = 50  # Animation frames per second
brightness = 0  # Current wave height
strip = neopixel.NeoPixel(led_pin, num_leds, brightness=1, auto_write=False)
offset = 0

# vibration sensor
motion_pin = board.D0  # Pin where vibration switch is connected
pin = digitalio.DigitalInOut(motion_pin)
pin.direction = digitalio.Direction.INPUT
pin.pull = digitalio.Pull.UP
ramping_up = False

center = 0  # Center point of wave in fixed-point space (0 - 255)
speed = 1  # Distance to move between frames (-128 - +127)
width = 2  # Width from peak to bottom of triangle wave (0 - 128)
hue = 3  # Current wave hue (color) see comments later
hue_target = 4  # Final hue we're aiming for
red = 5  # LED RGB color calculated from hue
green = 6  # LED RGB color calculated from hue
blue = 7  # LED RGB color calculated from hue

y = 0
brightness = 0
count = 0

# Gemma can animate 3 of these on 40 LEDs at 50 FPS
# More LEDs and/or more waves will need lower
wave = [0] * 8, [0] * 8, [0] * 8

# Note that the speeds of each wave are different prime numbers.
```

```
# This avoids repetition as the waves move around the
# perimeter...if they were even numbers or multiples of each
# other, there'd be obvious repetition in the pattern of motion...
# beat frequencies.
n_waves = len(wave)

# 90 distinct hues (0-89) around color wheel
hue_table = [255, 255, 255, 255, 255, 255, 255, 255, 237, 203,
             169, 135, 101, 67, 33, 0, 0, 0, 0, 0, 0, 0, 0, 0,
             0, 0, 0, 0, 0, 0, 18, 52, 86, 120, 154, 188, 222,
             255, 255, 255, 255, 255, 255, 255, 255]

# Gamma-correction table
gammas = [
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2,
    2, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5, 5,
    5, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 9, 9, 9, 10,
    10, 10, 11, 11, 11, 12, 12, 13, 13, 13, 14, 14, 15, 15, 16, 16,
    17, 17, 18, 18, 19, 19, 20, 20, 21, 21, 22, 22, 23, 24, 24, 25,
    25, 26, 27, 27, 28, 29, 29, 30, 31, 32, 32, 33, 34, 35, 35, 36,
    37, 38, 39, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 50,
    51, 52, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 66, 67, 68,
    69, 70, 72, 73, 74, 75, 77, 78, 79, 81, 82, 83, 85, 86, 87, 89,
    90, 92, 93, 95, 96, 98, 99, 101, 102, 104, 105, 107, 109, 110,
    112, 114, 115, 117, 119, 120, 122, 124, 126, 127, 129, 131, 133,
    135, 137, 138, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158,
    160, 162, 164, 167, 169, 171, 173, 175, 177, 180, 182, 184, 186,
    189, 191, 193, 196, 198, 200, 203, 205, 208, 210, 213, 215, 218,
    220, 223, 225, 228, 231, 233, 236, 239, 241, 244, 247, 249, 252,
    255
]


def h2rgb(colour_hue):
    colour_hue %= 90
    h = hue_table[colour_hue >> 1]

    if colour_hue & 1:
        ret = h & 15
    else:
        ret = (h >> 4)

    return ret * 17


# pylint: disable=global-statement
def wave_setup():
    global wave

    wave = [[0, 3, 60, 0, 0, 0, 0, 0],
            [0, -5, 45, 0, 0, 0, 0, 0],
            [0, 7, 30, 0, 0, 0, 0, 0]]

    # assign random starting colors to waves
    for wave_index in range(n_waves):
        current_wave = wave[wave_index]
        random_offset = random.randint(0, 90)

        current_wave[hue] = current_wave[hue_target] = 90 + random_offset
        current_wave[red] = h2rgb(current_wave[hue] - 30)
        current_wave[green] = h2rgb(current_wave[hue])
        current_wave[blue] = h2rgb(current_wave[hue] + 30)


def vibration_detector():
    while True:
        if not pin.value:
```

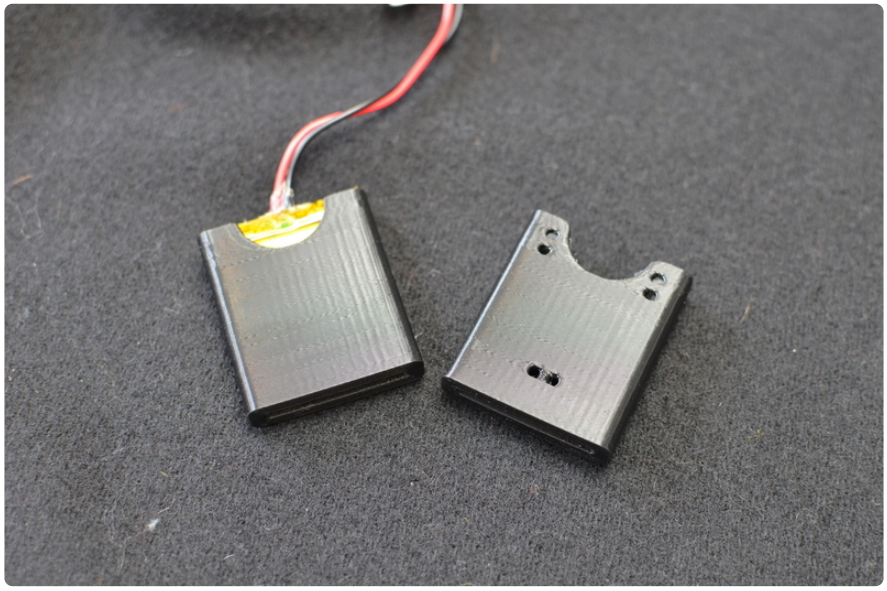
```
            return True


while True:

    # wait for vibration sensor to trigger
    if not ramping_up:
        ramping_up = vibration_detector()
        wave_setup()

    # But it's not just a straight shot that it ramps up.
    # This is a low-pass filter...it makes the brightness
    # value decelerate as it approaches a target (200 in
    # this case).  207 is used here because integers round
    # down on division and we'd never reach the target;
    # it's an ersatz ceil() function: ((199*7)+200+7)/8 = 200;
    brightness = int(((brightness * 7) + 207) / 8)
    count += 1

    if count == (circumference + num_leds + 5):
        ramping_up = False
        count = 0

    # Wave positions and colors are updated...
    for w in range(n_waves):
        # Move wave; wraps around ends, is OK!
        wave[w][center] += wave[w][speed]

        # Hue not currently changing?
        if wave[w][hue] == wave[w][hue_target]:

            # There's a tiny random chance of picking a new hue...
            if not random.randint(frames_per_second * 4, 255):
                # Within 1/3 color wheel
                wave[w][hue_target] = random.randint(
                    wave[w][hue] - 30, wave[w][hue] + 30)

        # This wave's hue is currently shifting...
        else:

            if wave[w][hue] < wave[w][hue_target]:
                wave[w][hue] += 1  # Move up or
            else:
                wave[w][hue] -= 1  # down as needed

            # Reached destination?
            if wave[w][hue] == wave[w][hue_target]:
                wave[w][hue] = 90 + wave[w][hue] % 90  # Clamp to 90-180 range
                wave[w][hue_target] = wave[w][hue]  # Copy to target

            wave[w][red] = h2rgb(wave[w][hue] - 30)
            wave[w][green] = h2rgb(wave[w][hue])
            wave[w][blue] = h2rgb(wave[w][hue] + 30)

        # Now render the LED strip using the current
        # brightness & wave states.
        # Each LED in strip is visited just once...
        for i in range(num_leds):

            # Transform 'i' (LED number in pixel space) to the
            # equivalent point in 8-bit fixed-point space (0-255)
            # "* 256" because that would be
            # the start of the (N+1)th pixel
            # "+ 127" to get pixel center.
            x = (i * 256 + 127) / circumference

            # LED assumed off, but wave colors will add up here
            r = g = b = 0
```

```
            # For each item in wave[] array...
            for w_index in range(n_waves):
                # Calculate distance from pixel center to wave
                # center point, using both signed and unsigned
                # 8-bit integers...
                d1 = int(abs(x - wave[w_index][center]))
                d2 = int(abs(x - wave[w_index][center]))

                # Then take the lesser of the two, resulting in
                # a distance (0-128)
                # that 'wraps around' the ends of the strip as
                # necessary...it's a contiguous ring, and waves
                # can move smoothly across the gap.
                if d2 < d1:
                    d1 = d2  # d1 is pixel-to-wave-center distance

                # d2 distance, relative to wave width, is then
                # proportional to the wave's brightness at this
                # pixel (basic linear y=mx+b stuff).
                # Is distance within wave's influence?
                # d2 is opposite; distance to wave's end
                if d1 < wave[w_index][width]:
                    d2 = wave[w_index][width] - d1
                    y = int(brightness * d2 / wave[w_index][width])  # 0 to 200

                    # y is a brightness scale value --
                    # proportional to, but not exactly equal
                    # to, the resulting RGB value.
                    if y < 128:  # Fade black to RGB color
                        # In HSV colorspace, this would be
                        # tweaking 'value'
                        n = int(y * 2 + 1)  # 1-256
                        r += (wave[w_index][red] * n) >> 8  # More fixed-point math
                        # Wave color is scaled by 'n'
                        g += (wave[w_index][green] * n) >> 8
                        b += (wave[w_index][blue] * n) >> 8  # >>8 is equiv to /256
                    else:  # Fade RGB color to white
                        # In HSV colorspace, this tweaks 'saturation'
                        n = int((y - 128) * 2)  # 0-255 affects white level
                        m = 256 * n
                        n = 256 - n  # 1-256 affects RGB level
                        r += (m + wave[w_index][red] * n) >> 8
                        g += (m + wave[w_index][green] * n) >> 8
                        b += (m + wave[w_index][blue] * n) >> 8

            # r,g,b are 16-bit types that accumulate brightness
            # from all waves that affect this pixel; may exceed
            # 255.  Now clip to 0-255 range:
            if r > 255:
                r = 255
            if g > 255:
                g = 255
            if b > 255:
                b = 255

            # Store resulting RGB value and we're done with
            # this pixel!
            strip[i] = (r, g, b)

        # Once rendering is complete, a second pass is made
        # through pixel data applying gamma correction, for
        # more perceptually linear colors.
        # https://learn.adafruit.com/led-tricks-gamma-correction
        for j in range(num_leds):
            (red_gamma, green_gamma, blue_gamma) = strip[j]
            red_gamma = gammas[red_gamma]
            green_gamma = gammas[green_gamma]
            blue_gamma = gammas[blue_gamma]
            strip[j] = (red_gamma, green_gamma, blue_gamma)
```

```
        strip.show()
```

This code requires the **neopixel.py** library. A factory-fresh board will have this already installed. If you've just reloaded the board with CircuitPython, create the "lib" directory and then download neopixel.py from Github (https://adafru.it/yew).

3D Printed Battery Pocket



Print a sew-on pocket for your lipoly battery! The pocket protects the battery from abuse and also makes it easy to remove the battery for charging. It's not strictly necessary, though, but bare lipoly batteries can be risky, so unless you protect it in some way, we recommend using a hard shell alkaline pack like the 3xAAA holder (http://adafru.it/727).

TPE Flexible Filament

The battery pocket works best when printed in flexible material like Ninjaflex or **Semiflex**. This material requires a direct-drive extruder system and can be challenging to print. We recommend **Semiflex** because it handles overhangs better than Ninjaflex, and has a shell hardness (98A). Follow the print settings below for best results.

Printing speed	30mm/sec	
Retraction	OFF	

Raft / Support Material	OFF	
Extruder Temperature	220-230c	
Heated Bed	20-50c (if applicable)	

The part should be centered on the print bed and ready to print "as-is". We recommend using CURA (https://adafru.it/iAQ), or Simplify3D (https://adafru.it/iAR) to slice the file.

Download STL & Source files

https://adafru.it/iAS

Wear 'em!

Get out there and get flashin'! These shoes are sure to turn heads wherever you go.

To protect the circuit further, you can coat GEMMA with a conformal coating or paint on some clear nail polish. See our Rugged-izing Wearables (https://adafru.it/e4d) video for more tips on protecting your cirtcuit from the elements.

If you make a pair, share them with us on our weekly Google+ show and tell hangout (), post them up in the forums (https://adafru.it/jlf), or submit a blog tip (https://adafru.it/dB4)!