Ukulele Glove Tutorial

Introduction

In this tutorial we are going to be making a Music Glove! This glove makes different sounds when the tip of one of the four main fingers is touched to the tip of the thumb, completing a circuit. This can be tuned to any sound you would like. I have decided to tune the fingers to the open tuning of a ukulele. This tutorial requires you to have some basic knowledge of sewing, C++ code, and some hand dexterity to sew the finer components.

This glove is designed to be used as an entertainment device, specifically one that can be used to make simple melodies! In later versions, there is potential to turn this into a full instrument with some additional hardware and software.

This was inspired by the Theremin Glove in the book *Make: Wearable Electronics 2nd edition*.

Note: This tutorial previously used LEDs in-line for the circuit for each finger. This creates an issue where too much resistance prevents the LED from lighting up and the board from recognizing the completion of the circuit to play the music. Instead, I have opted to change the color of the built in NeoPixel on the Flora V2 as a replacement. This required additional code, but no further changes. The changes to the code have been marked with "// <<< CHANGED" to indicate where those changes are.

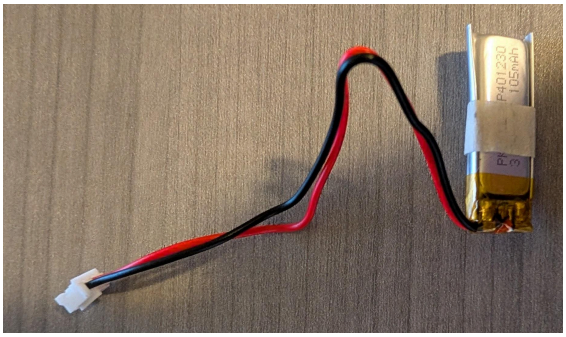
Currently, this tutorial only describes how to make a glove which allows for one finger-thumb connection at a time. Future upgrades to this glove could include: multiple finger-thumb connections registering as a combination of two sounds, variable tones per finger based on pinch force used, light sensor for variable tones (imitating a fretboard) and more!

An important note: When speaking about "Fingers" in this tutorial, it should be assumed that we are describing the four main digits on the hand (Index, Middle, Ring, Pinkie) and not the thumb, the thumb will be used as the "Connection Finger" (or referred to as just "thumb" for simplicity) to which each of the other fingers touch in order to complete a circuit and produce a sound.

The code for this project can be found in my Github Repo! See the Code section for more information on how this code was written and what it does.

Materials

The required materials for this project are:

- **Electronics:**
  - Adafruit Flora V2 Board
  - Conductive Thread (any gauge)
  - Conductive Tape
  - Buzzer (Piezobuzzer from Lilypad Protosnap Board used in this tutorial)
  - Button (Button from Lilypad Protosnap Board used in this tutorial)
  - Alligator Clips (For testing code and circuits before final stitching)
  - Small 3.3v battery (to power the Flora V2 when not connected to a computer)
  - 
  - 6 conductive snaps of small size (or 5 conductive snaps and 1 non-conductive snap) (6th snap is optional, it's used to stabilize the piece but not required for functionality!)
    

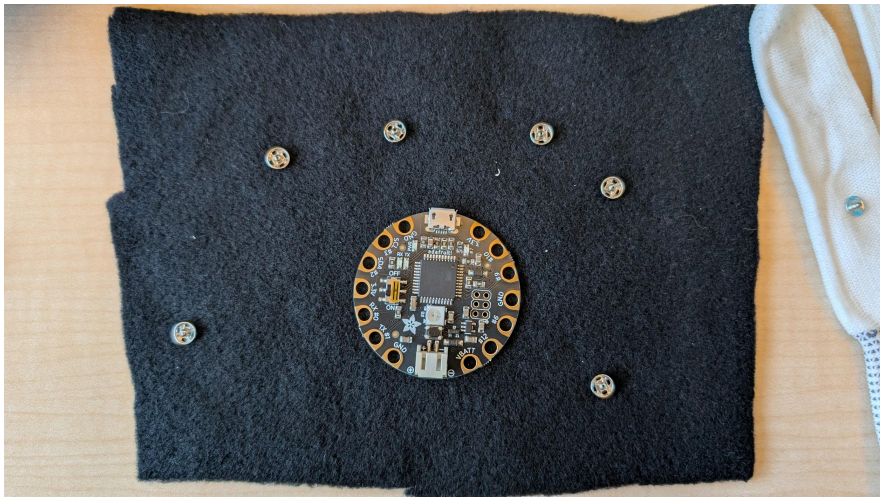
    (Gondola brand 5mm snaps used for this tutorial)
  - Micro USB to USB A Cable (For loading code onto Flora Board)
  - Computer with Arduino IDE installed and Flora Board set up (Follow this tutorial for setting up the Flora V2 Board in your Arduino IDE and loading your first program)
- **Textiles/Sewing Materials**
  - Sewing needle
  - Tailor's chalk (or other method to mark felt)
  - Felt of at least full hand size (any color, black is used in this tutorial)
  - Pins ***Essential!** (Can be any fabric pins, clothespins. Used to hold fabric in place during measurement and hold location of items sewn onto glove/back piece)
  - Cutting device (Rotary Cutter, Fabric Scissors preferred)
    - Cutting surface (Essential for Rotary Cutter, not required for others)

- Sewing thread (non-conductive, any color)
- Glove (Any non-stretchy glove works, this tutorial uses a $5 pair of work gloves from Home Depot [Milwaukee X-Large Red Nitrile Level 1 Cut Resistant Dipped Work Gloves 48-22-8903 - The Home Depot])

Design and Setup (Part 1 - The Backpiece)

For this glove, we need to start with the felt backpiece. This will be where all of your electronics are mounted, and will be removable so the glove portion can be washed or swapped with another glove.
This will be connected to the back of the glove via Conductive Snaps (in materials) which will connect the conductive thread running up each finger to the electronics and controller board on the felt backpiece.

Example of the snaps layout, (FOR DEMONSTRATION ONLY: this has snaps on the wrong side of the felt piece, this is to demonstrate rough location of these snaps not to be used as a guide for later installation. Additionally, this white glove has been chosen to show location of snaps and will not be the same as used in the final project.)



Example board placement, (FOR DEMONSTRATION ONLY: this will be on the opposite side of the conductive snaps attached to the felt backpiece.)

1. Take your felt and wrap it around the hand your glove is for (in my case, my right hand) to get an idea of the measurements that we'll cut the felt down to. Don't worry too much about the felt running up your fingers or down onto your wrist, this will be adjusted before we cut the piece to size.

2. Once you have your felt on the back of your hand, fold the sides in to match the edges of the back of your hand. Don't worry if the top and bottom are too high on your knuckles or

low on your wrist, this will be fixed in the next step.

3. Take some of the pins (in the materials list) and begin to pin the sides down to each other.
   Once you have the sides pinned, then fold about a quarter to a half inch of the top and bottom of the felt piece.
   This is to bring the felt down to the size of the back of your hand (removing those sections going over your knuckles and down onto your wrist) so if your felt is cut larger or smaller than the piece here, adjust the sizing accordingly.
   Put the felt piece on the back of your hand after each pin to ensure the sizing is correct.

Once complete, it should look something like this:

4. Once you have your pinned piece of felt, use tailor's chalk (from materials) to mark each of the edges on the outside along the crease. This is so we can remove the pins and still have an edge to cut along in our next step. (Don't worry, the chalk comes off if you brush it forcefully with your hand, or wash the felt)

5. Once you've marked all of your edges, take out all of the pins from your felt piece, lay it on a cutting mat with chalk marks facing up (do not cut this on a table if using a rotary cutter, use a mat designed for this! It will damage your cutting surface otherwise) and using your rotary cutter, cut each edge along the chalk marks.
   If you don't have a rotary cutter, using fabric scissors is a good alternative.
   If you don't have those, pick a pair of sharp scissors and try it out on some scrap felt to ensure it cuts cleanly before cutting your final back piece.

Once done, you should have a felt back piece that looks something like this!

Now that we've cut out our backpiece, let's move on to the glove!

Design and Setup (Part 2 - The Glove)

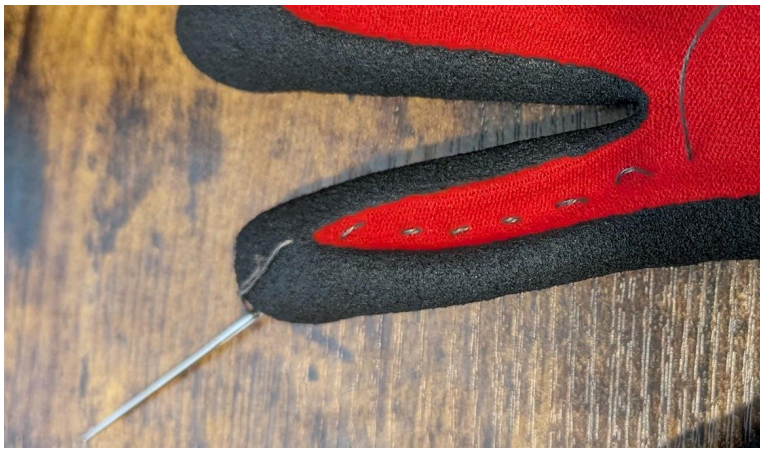
The glove is the most important part of this whole setup, so it's important to understand how it works in conjunction with the back piece!

The work we need to do on the glove itself is relatively minimal. We will just need to attach the conductive snaps, and run conductive thread up each of the fingers from those snaps. While that sounds easy, there is some complexity involved and a few things we need to make sure we do carefully so the glove can continue to be used!

The first and most important thing to consider is that conductive thread is not very stretchy, and the fingers of gloves are designed to stretch as the wearer moves their fingers. If you want to see just how much your glove stretches, put it on and make a fist with your hand. If you're using a woven glove, you'll likely see the threads of the fingers separate and stretch slightly. This is great, but since our conductive thread won't stretch as easily and is prone to snapping, we need to be careful with how we sew the fingers.

How to Sew the Fingers

When sewing the fingers of your glove, as mentioned above, you will need to be careful to give your thread enough slack to bend comfortably with the fabric of the glove fingers.



Example of a finger sewn with some light slack, increasing near the knuckle where most of the stretching occurs.

To be certain that you have given your fingers enough slack to prevent breaking your conductive thread, before you tie off the ends of the thread, it's a good idea to put the glove on then close your fist as shown below. When you open your hand the thread will have moved and given itself some slack naturally. However much there is, that should be your rough guide on how much slack to give that finger.

Put the glove on with the thread not tied off on either end. Make sure to not stab yourself with the needle while doing this. It should be outside of the glove.

Make a fist with your hand to stretch the conductive thread to the maximum required extent

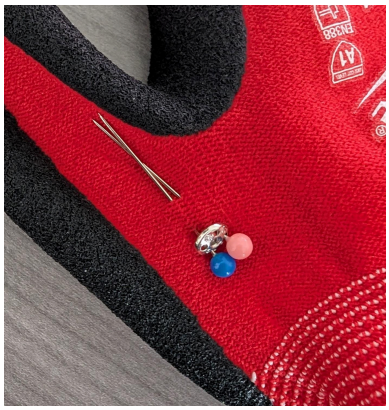
Unclench your fist and check the slack left behind in the conductive thread. This should be a rough target for each finger. Remember, each finger stretches a bit differently, so it's always best to check as many as possible!

Once you've got a good idea of how much slack you'll need in the fingers of your glove, we can start attaching the conductive snaps at the base of each knuckle!

Attaching the Conductive Snaps

For this process we will start by putting on the glove and bending your fingers. You want to put the snap BELOW the bend of your knuckles, this, while extending the length of the conductive thread and requiring more slack, will keep the back piece connection points from shifting during use, which is essential.

See how the snap goes below the knuckle? This is what we're looking for! This keeps the connection to the back piece in a stable location, preventing the back piece from stretching or deforming too much.



Use pins to keep the snap where you want it to go, then use conductive thread to go through the holes of the snap. Don't worry if it isn't fully secure now. We can go back over this with a non-conductive thread later to strengthen it.

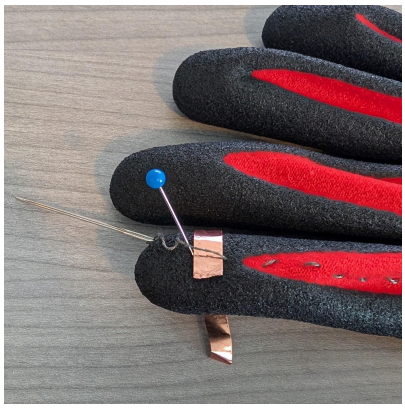
Once you have the snap attached, tie off the end of the thread. This can be left for the time being, or cut now. We will clean these up before we finish!

Now that we've got our first conductive snap on, it's time to sew the finger tip!

Sewing the Fingertips (Connection Points)

To make the glove functional as a music device, we need to have something conductive that touches between the fingers and the thumb to complete a circuit! To do this, we are using conductive tape secured by non-conductive thread.

First, you want to measure out roughly the circumference of your fingertip worth of conductive tape.

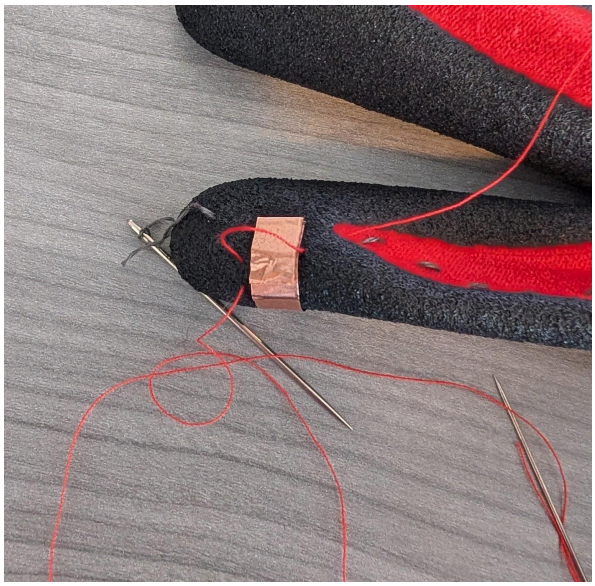


Once we've done this, as shown in the image above, we need to bring the conductive thread over the top of the tape to make the connection. I have put a pin where I will be putting the conductive thread back through the glove.

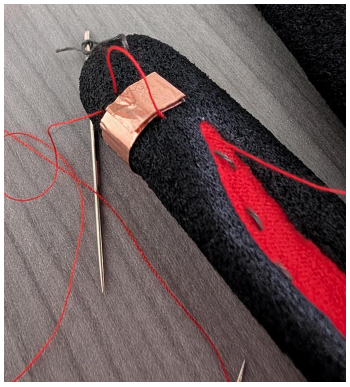
Before we sew it down, we want to make sure we orient the tape properly. For this glove, since it's the right hand, I am putting the tape on the left side of the fingers. This is so that the thumb-side of the fingertips are all conductive, ensuring an easier connection.

Orient the tape so there's about a quarter-inch of tape on the right of the conductive thread, and the rest wraps around the finger tip. This is so we can fold the tape over the conductive thread to ensure a solid connection.

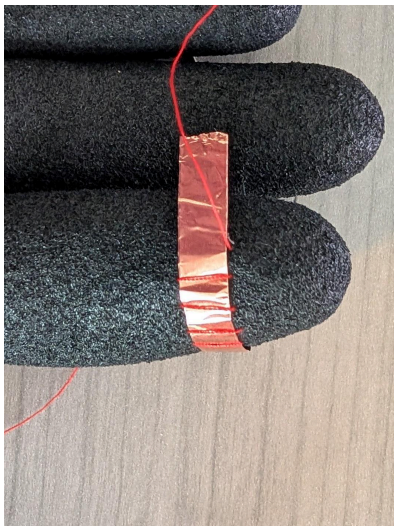
Once we've folded the tape over the conductive thread, the next step is to sew the conductive tape down! The tape won't stay forever on its own, so we have to secure it. For this process I've chosen red non-conductive thread. Red is a good accent color for this glove, but pick whatever color looks best with yours!



Now that we've folded the tape over, make sure to sew across the tape the whole way along it. We don't want to leave it loose or there's a chance it will break.



Starting the stitch along the conductive tape, with the fold over the conductive thread. Make sure this first stitch is tight!



As you move along the tape, try to keep your stitches even. This will help with look, but also evenly secure the tape to prevent breakage.

Once done, run the non-conductive thread back to your starting point and tie it off! Then cut the excess and there you go, a conductive fingertip!

Lastly, you'll want to tie the conductive thread at the tip of the finger. For this, I did a square knot. It's large enough to prevent the thread from coming back through the hole, but gives a bit of room to move beneath it to avoid breaking the conductive thread on the finger when you bend it during use.

Glove with one finished finger!

Now that you've done one finger, repeat this process for all other fingers, including the thumb!

Once we've finished up our glove base, it's time to move on to the code!

Code

Writing the Code!

Alright, let's get into the brains of our Music Glove: the code! This is what will tell the Flora board how to react when you touch your fingers together and press the octave button.

First thing's first, before we start writing any code, make sure you have your Arduino IDE set up. For this tutorial, the version used is 2.3.6, but newer versions should work fine. You'll also need to have gone through the Adafruit Flora V2 Tutorial to get your board configured correctly with the IDE and to understand how to upload code.

Once you've got your IDE set up, plug your Flora into your computer using the Micro USB to USB A cable, and let's get started!

1. Identifying and Defining Finger Pins

The very first thing our code needs to know is *where* our electronic components are connected on the Flora board. Each connection point on the Flora is called a **pin**. We need to tell the code which pins will be used for sensing when each of our four main fingers (Index, Middle, Ring, Pinkie) makes a connection with the thumb.

Looking at a pinout diagram for the Flora V2 (you can find this on the Adafruit website here), we'll choose specific digital pins. Digital pins can be either "on" (HIGH) or "off" (LOW). For this project, we'll use:

- **D10** for the Index Finger
- **D9** for the Middle Finger
- **D6** for the Ring Finger
- **D12** for the Pinkie Finger

In C++ (the language Arduino uses), we declare variables to store these pin numbers. We use `const int` which means "constant integer." "Constant" means the value won't change once set, and "integer" means it's a whole number.

```
// Written by: Alex Hines
// 'Ukulele Synth Glove' project
// Date: 5/26/2025

// ====================== Finger settings ======================
// Define the pins for the fingers
const int fingerPin1 = 10;  // Pin for G (Index Finger)
const int fingerPin2 = 9;   // Pin for C (Middle Finger)
const int fingerPin3 = 6;   // Pin for E (Ring Finger)
const int fingerPin4 = 12;  // Pin for A (Pinkie Finger)
```

To make it easier to work with all these finger pins together (for example, when we want to check all of them), we can put them into an **array**. An array is like a list that holds multiple values of the same type. We'll call this array `fingerPins`.

We also define `numFingers` to hold the number 4. This makes our code more readable and easier to update if we ever wanted to change the number of fingers.

```
const int fingerPins[] = {fingerPin1, fingerPin2, fingerPin3, fingerPin4}; // Array
of finger pins
const int numFingers = 4; // Number of fingers we are using
```

Finally, for our finger settings, we need a variable to keep track of which finger's note is currently being played. We'll call it `currentlyPlayingFinger`. We initialize it to `-1`. This is a common trick: since finger numbers (or array indices) usually start from 0, `-1` is a good way to signify "nothing is playing" or "no finger is active." This glove is designed to play one note at a time.

```
// Variable to track which finger's note is currently playing
// -1 means no note is playing for that finger. This system plays one note at a //
// time.
// [Area of potential future improvement - multiple notes at one time]
int currentlyPlayingFinger = -1;
```

1.5 Setting up the NeoPixel

```
// ====================== NeoPixel settings ==================== // <<< CHANGED
// Pin for the onboard NeoPixel.
const int NEOPIXEL_PIN = 8;
// Create the NeoPixel object globally so it can be used in any function.
Adafruit_NeoPixel strip = Adafruit_NeoPixel(1, NEOPIXEL_PIN, NEO_GRB +
NEO_KHZ800); // <<< CHANGED

// Define colors for each finger (G, C, E, A) - Red, Green, Blue, Yellow
uint32_t fingerColors[] = { // <<< CHANGED
  strip.Color(255, 0, 0),    // Index finger (G) -> Red
  strip.Color(0, 255, 0),    // Middle finger (C) -> Green
  strip.Color(0, 0, 255),    // Ring finger (E) -> Blue
  strip.Color(255, 255, 0)   // Pinky finger (A) -> Yellow
};
```

We will define here the settings we need to have the NeoPixel on the Flora light up a different color for each finger when connected.

- **NeoPixel Pin:** We need to tell the code which pin is for the NeoPixel. Since this is built in, it isn't necessarily visible on the board or the pinout diagram. The pin for the built-in NeoPixel is pin 8, so we'll use that.
- **Finger Colors:** For each finger to light up a different color, we are going to set up a fingerColors list variable which stores the RGB color value associated with each finger. Then, when we trigger the sounds from the buzzer we will trigger the finger colors in the same way.

2. Setting Up the Sound: Buzzer Pin and Note Frequencies

Next, we need to define how our glove will make sound.

- **Buzzer Pin**: We need to tell the code which pin the buzzer (our sound-making component) is connected to. We're using pin **D2** (which is also labeled SDA on the Flora, but it can be used as a general digital pin too).
- **Note Frequencies**: Musical notes are essentially sound waves vibrating at specific speeds, called frequencies (measured in Hertz, or Hz). I've chosen to tune the glove like a ukulele in its standard GCEA tuning. We'll define these frequencies. Since frequencies can have decimal points, we use `const float` (a "floating-point number" is a number that can have a decimal).

```
// ======================= Note settings =======================
// Define the pin for the buzzer (SDA pin, usable as general I/O)
const int buzzerPin = 2;

// Ukulele base note frequencies (G4, C4, E4, A4)
const float NOTE_G4 = 392.00; // G note, 4th octave
const float NOTE_C4 = 261.63; // C note, 4th octave
const float NOTE_E4 = 329.63; // E note, 4th octave
const float NOTE_A4 = 440.00; // A note, 4th octave
```

Just like with the finger pins, we'll store these note frequencies in an array called `baseNotes` for easy access. The order in this array should correspond to the order of your `fingerPins` (e.g., fingerPin1 plays NOTE_G4, fingerPin2 plays NOTE_C4, and so on).

```
// Array of base notes, corresponding to fingerPins
const float baseNotes[] = {NOTE_G4, NOTE_C4, NOTE_E4, NOTE_A4};
```

3. Adding Octaves: Button and Multipliers

To make our musical glove more versatile, we're adding an octave button. Pressing this button will shift all the notes up to a higher pitch.

- **Octave Button Pin**: The button used to change octaves will be connected to pin **D3** (also labeled SCL on the Flora, usable as a general digital pin).
- **Octave Tracking**:

    - `currentOctave`: An integer variable to remember which octave setting is currently active. It will act as an index to pick a multiplier from the octaveMultipliers array. We start at 0, which will be our normal octave.
    - `numOctaveSettings`: A constant telling the code how many different octave settings we have (e.g., normal, one octave up, two octaves up). For this tutorial, we have 3 settings.
- **Octave Multipliers**: An array of float values. To change an octave, you multiply a note's frequency.
    - 1.0: Plays the note at its original pitch (no change).
    - 2.0: Plays the note one octave higher (frequency doubles).
    - 4.0: Plays the note two octaves higher (frequency quadruples).
      The comments also show an alternative {0.5, 1.0, 2.0} if you wanted low, middle, and high octaves.

```
// ====================== Octave settings ======================
// Define the pin for the octave button (SCL Pin, but works for standard IO)
const int octaveButtonPin = 3;

// Index for the octaveMultipliers array, starts at the first setting (0)
int currentOctave = 0;

// Total number of octave settings to cycle through
const int numOctaveSettings = 3;

// Octave multipliers: 0 = Middle, 1 = Up one octave, 2 = Up two octaves
float octaveMultipliers[] = {1.0, 2.0, 4.0};
// Alternative for Low, Middle, High with different octaves in-range:
// float octaveMultipliers[] = {0.5, 1.0, 2.0};
```

4. Handling Connection "Bounces" (Debouncing)

When you press a physical button, it doesn't just make one clean electrical connection. For a tiny fraction of a second, the metal contacts inside can "bounce," opening and closing the circuit multiple times. A microcontroller like the Flora is so fast it might see these bounces as many separate presses! This is also true for the finger connection points, which act similar to how buttons act internally, so we must do this for both!

Debouncing is a technique to make sure we only register one action for each actual physical press. We need some variables to help us with this, both for the octave button and for the finger "presses" (when a finger touches the thumb, it's like a button press).

For the Octave Button:

- lastButtonState: Stores if the button was pressed (LOW) or not (HIGH) the last time we checked. We'll initialize it to HIGH (not pressed).

- lastDebounceTime: Records the time (in milliseconds) when the button last seemed to change state.
- debounceDelay: A short period (e.g., 50 milliseconds) we'll wait for the button signal to become stable before we trust its state.
- debouncedButtonState: Stores the confirmed, stable state of the button.

```
// ====================== Button settings ======================
// Variables to track button state for debouncing and single press detection
int lastButtonState = HIGH;  // Assuming button pulls to LOW when pressed (internal pull-up)
unsigned long lastDebounceTime = 0; // Stores the last time the output pin was toggled
unsigned long debounceDelay = 50;  // 50 milliseconds for debounce
int debouncedButtonState = HIGH;   // Stores the debounced state of the button
```

Note: unsigned long is used for time variables because the time in milliseconds can become a very large number, and unsigned long can hold larger positive values than int.

For the Finger "Presses":
Each finger completing a circuit with the thumb also needs debouncing. Since we have multiple fingers, we'll use arrays for these variables. Each finger gets its own set of debouncing timers and state trackers.

- `fingerDebouncedState[numFingers]`: An array to store the confirmed, stable state for each finger (LOW means the circuit is complete/pressed).
- `lastFingerRawState[numFingers]`: An array storing the last raw input state we read from each finger pin before debouncing.
- `lastFingerDebounceTime[numFingers]`: An array storing the time of the last potential state change for each finger.

```
// ====================== Debouncing Vars ======================
// Finger Debouncing Variables
int fingerDebouncedState[numFingers];     // Stores the current debounced state (LOW = pressed)
int lastFingerRawState[numFingers];       // Stores the last raw input state read
unsigned long lastFingerDebounceTime[numFingers];  // Stores time of last raw state change for each finger
```

5. The setup() Function: One-Time Preparations

In every Arduino program, there are two main functions: setup() and loop().
The setup() function runs only once when the Flora board first powers on or when it's reset. This is where we do all our initial preparations.

- `Serial.begin(9600);`: This line initializes serial communication. "Serial" is a way for the Flora to send messages back to your computer, which appear in the Arduino IDE's "Serial Monitor." This is incredibly useful for debugging or just seeing what your code is doing.
- **NeoPixel Setup**:
  - We will use the NeoPixel library (Please install it using the IDE by searching for it in the Libraries tab on the left. It will be called Adafruit NeoPixel. I am using version 1.15.1, so any version this one or later should work!
- **Finger Pin Initialization (Loop)**:
  - `for (int i = 0; i < numFingers; i++) { ... }`: This is a `for` loop. It repeats the code inside its curly braces `{}` a specific number of times. Here, it runs `numFingers` times (so, 4 times). The variable `i` starts at 0 and increases by 1 each time, up to 3. This `i` is used as an index to access each element in our `fingerPins` array and the finger debouncing arrays.
  - `pinMode(fingerPins[i], INPUT_PULLUP);`: This is a very important command. `pinMode()` tells the Flora whether a specific pin should be an INPUT (listening for a signal) or an OUTPUT (sending a signal).
    - We set each finger pin to `INPUT_PULLUP`. This means:
      - It's an `INPUT`: The Flora will "listen" to this pin.
      - `_PULLUP`: This activates a tiny internal "pull-up resistor" inside the Flora's chip. This resistor connects the pin to a positive voltage (HIGH). So, when nothing is touching and the circuit is open, the pin reads as HIGH. When you touch the conductive material on a finger to the conductive material on the thumb (which should be connected to Ground/GND), you complete the circuit, and the pin's voltage is pulled LOW (to 0V). This change from HIGH to LOW is how we detect a "press."
  - The lines `lastFingerRawState[i] = HIGH;`, etc., initialize the debouncing variables for each finger to their default "not pressed" state.
- **Octave Button Initialization**:
  - `pinMode(octaveButtonPin, INPUT_PULLUP);`: Just like the finger pins, the octave button pin is set as an `INPUT_PULLUP`. When the button isn't pressed, it's HIGH. When pressed, it connects to Ground and goes LOW.
  - `lastButtonState = digitalRead(octaveButtonPin);`: We read the initial state of the button. `digitalRead()` checks if a pin is HIGH or LOW.
  - `debouncedButtonState = lastButtonState;`: We set the initial debounced state to match.
- **Buzzer Pin Initialization**:
  - `pinMode(buzzerPin, OUTPUT);`: The buzzer pin is set as an `OUTPUT` because the Flora will send electrical signals *to* the buzzer to make it produce sound.
- **Serial Messages**:
  - `Serial.println(...)` and `Serial.print(...)`: These commands send text to the Serial Monitor. `println` adds a new line after the text, while `print` does not. This is helpful to confirm that the setup is complete and to show the initial octave.

```
// ================== Setup & Loop Methods ====================
void setup() {
  Serial.begin(9600);

  // Initialize each finger pin
  for (int i = 0; i < numFingers; i++) {
    pinMode(fingerPins[i], INPUT_PULLUP); // Set as input with internal pull-up resistor

    // Set up the NeoPixel and default to off.
    strip.begin();                                  // <<< CHANGED
    strip.setBrightness(50);                        // <<< CHANGED
    strip.show(); // Initialize all pixels to 'off' // <<< CHANGED

    // Initialize finger debouncing variables to 'not pressed' state
    lastFingerRawState[i] = HIGH;     // Assume not pressed initially
    fingerDebouncedState[i] = HIGH;   // Assume not pressed initially
    lastFingerDebounceTime[i] = 0;    // Reset debounce timer
  }

  // Set octave button pin as input with internal pull-up resistor
  pinMode(octaveButtonPin, INPUT_PULLUP);
  lastButtonState = digitalRead(octaveButtonPin); // Read initial state
  debouncedButtonState = lastButtonState;         // Set initial debounced state

  // Set buzzer pin as output
  pinMode(buzzerPin, OUTPUT);

  // Print messages to Serial Monitor to show the program has started
  Serial.println("Ukulele Glove Synthesizer ready");
  Serial.print("Octave Setting: ");
  Serial.print(currentOctave); // Print the initial octave index
  printOctaveName();           // Call a helper function to print the octave name
}
```

6. The `loop()` Function:

After `setup()` runs once, the `loop()` function takes over. The code inside `loop()` runs over and over again, continuously, as long as the Flora has power. This is where the main, repetitive actions of our program happen – checking for finger touches and button presses.

Our `loop()` is kept very simple and readable by calling other functions that do the detailed work:

- `handleOctaveButton();`: This calls a custom function (which we'll define below) that checks if the octave button has been pressed and updates the octave if needed.
- `handleFingerPresses();`: This calls another custom function that checks all the finger sensors, figures out if a note should be played or stopped, and controls the buzzer.

Using functions like this is good practice. It's called "modularization" and helps keep your code organized and easier to understand and debug.

```
void loop() {
  handleOctaveButton();   // Check and process octave button presses
  handleFingerPresses();  // Check and process finger touches to play notes
}
```

7. Helper Functions: Doing Specific Jobs

Helper functions are smaller pieces of code that perform specific tasks. We've already seen printOctaveName() called in setup(), and loop() calls handleOctaveButton() and handleFingerPresses(). Let's look at how these are written.

printOctaveName()

This small utility function's only job is to print a more human-readable name for the current octave to the Serial Monitor (e.g., "(Middle)", "(High)"). It looks at the currentOctave variable (which is an index like 0, 1, or 2) and checks the corresponding value in the octaveMultipliers array to decide what name to print.

```
// ====================== Helper Methods ======================
void printOctaveName() {
  // Check the multiplier for the current octave and print a descriptive name
  if (octaveMultipliers[currentOctave] == 0.5) Serial.println(" (Low)");
  else if (octaveMultipliers[currentOctave] == 1.0) Serial.println(" (Middle)");
  else if (octaveMultipliers[currentOctave] == 2.0) Serial.println(" (High)");
  else if (octaveMultipliers[currentOctave] == 4.0) Serial.println(" (Very High)");
  else Serial.println(""); // If no match, print an empty line
}
```

handleOctaveButton()

This function manages everything related to the octave button. It's designed to be "non-blocking" which means it checks the button quickly without using delay() commands. delay() would pause the entire program, making the glove unresponsive (which was an issue in early versions of this project!).

1. **Read Button State**:
   - `int currentRawReading = digitalRead(octaveButtonPin);`: It reads the current electrical state of the octave button pin (HIGH if not pressed, LOW if pressed, thanks to INPUT_PULLUP).
2. **Debounce Logic**: This is the core of making sure we only react once per press.
   - `if (currentRawReading != lastButtonState)`: If the button's current raw state is different from what it was the last time we checked, it *might* be changing. So, we record the current time by calling millis() and storing it in lastDebounceTime. millis() gives the number of milliseconds since the Flora started running.
   - `if ((millis() - lastDebounceTime) > debounceDelay)`: This checks if enough time (our debounceDelay, e.g., 50ms) has passed since the last time the button state

seemed to flicker. If it has been stable for this delay period:

- `if (currentRawReading != debouncedButtonState)`: And if this stable currentRawReading is different from our official debouncedButtonState, then we have a confirmed change!
  - `debouncedButtonState = currentRawReading;`: We update the official debouncedButtonState.
  - `if (debouncedButtonState == LOW)`: If this confirmed new state is LOW, it means the button was definitely pressed!
    - Serial.println(...): We print a message to the Serial Monitor for debugging.
    - `currentOctave = (currentOctave + 1) % numOctaveSettings;`: This is a trick to cycle through octaves. It increments currentOctave. The % is the "modulo" operator. It gives the remainder of a division. So, (currentOctave + 1) % 3 will result in:
      - If currentOctave was 0, it becomes (0+1)%3 = 1.
      - If currentOctave was 1, it becomes (1+1)%3 = 2.
      - If currentOctave was 2, it becomes (2+1)%3 = 0 (because 3 divided by 3 is 1 with a remainder of 0). This makes it cycle: 0, 1, 2, 0, 1, 2...
    - Serial.print(...) and printOctaveName();: We announce the new octave.
    - `if (currentlyPlayingFinger != -1)`: If a note is currently playing when the octave changes, we should stop it to avoid confusion.
      - `noTone(buzzerPin);`: The noTone() command stops any sound being played on the buzzerPin.
      - `currentlyPlayingFinger = -1;`: We reset this to indicate no note is (or should be) playing. The note will restart in the new octave if the finger is still held down, because handleFingerPresses() will detect it again in the next loop.

3. **Update Last State**:
   - `lastButtonState = currentRawReading;`: Finally, we save the currentRawReading into lastButtonState so that in the next run of the loop(), we can compare against it to detect further changes.

```
// Method to handle button interactions while not blocking loop (No delay() calls)
void handleOctaveButton() {
  int currentRawReading = digitalRead(octaveButtonPin); // Read the raw state of the button

  // --- Debug Prints (Optional: uncomment to see button states in Serial Monitor) ---
  // Serial.print("Octave RAW: "); Serial.print(currentRawReading);
  // Serial.print(" | LastRAW: "); Serial.print(lastButtonState);
  // Serial.print(" | DebouncedState: "); Serial.print(debouncedButtonState);
  // Serial.print(" | Debounce Timer: "); Serial.println(millis() - lastDebounceTime);
  // --- End of Debug Prints ---

  // Debounce logic for the button
  // Source for idea (Also used in handleFingerPresses()):
https://dev.to/aneeqakhan/throttling-and-debouncing-explained-1ocb#
```

```
  // If the raw input has changed since last time, reset the debounce timer
  if (currentRawReading != lastButtonState) {
    lastDebounceTime = millis(); // Record the time of this change
  }

  // If enough time has passed since the last raw input change (i.e., signal is stable)
  if ((millis() - lastDebounceTime) > debounceDelay) {
    // If the stable reading is different from our confirmed debounced state
    if (currentRawReading != debouncedButtonState) {
      debouncedButtonState = currentRawReading; // Update the debounced state

      // If the confirmed debounced state is LOW, the button was pressed
      if (debouncedButtonState == LOW) {
        Serial.println(">>> Octave Button PRESS Detected (Debounced) <<<");
        currentOctave = (currentOctave + 1) % numOctaveSettings; // Cycle to the next octave
        Serial.print("Octave Setting changed to: ");
        Serial.print(currentOctave);
        printOctaveName(); // Print the friendly name of the new octave

        // If a note is playing when octave changes, stop it.
        // It will restart in the new octave if the finger is still held.
        if (currentlyPlayingFinger != -1) {
          noTone(buzzerPin);             // Stop the current sound
          currentlyPlayingFinger = -1;   // Mark that no note is playing
          Serial.println("Note stopped for octave change, will restart if finger held.");
        }
      }
    }
  }

  // Store the current raw reading for the next comparison
  lastButtonState = currentRawReading;
}
```

handleFingerPresses()

This is where making music happens! This function checks each finger, debounces its input, and decides whether to play a new note, continue an old one, or stop the sound.

1. **Debounce Each Finger**:
   - `for (int i = 0; i < numFingers; i++) { ... }`: This loop goes through each finger (index `i` from 0 to 3).
   - Inside, it performs the exact same debouncing logic we used for `handleOctaveButton()`, but this time it's applied to `fingerPins[i]` and uses the corresponding `lastFingerRawState[i]`, `lastFingerDebounceTime[i]`, and `fingerDebouncedState[i]` for that specific finger. This ensures each finger's "press" (circuit completion) is reliably detected.
2. **Determine Which Finger to Play**:
   - `int fingerToPlay = -1;`: We start by assuming no finger is active enough to play a note.

- `for (int i = 0; i < numFingers; i++) { ... }`: We loop through the fingers again.
- `if (fingerDebouncedState[i] == LOW)`: If a finger's confirmed, debounced state is LOW (meaning its circuit with the thumb is complete)...
  - `fingerToPlay = i;`: We set fingerToPlay to the index (i) of this active finger.
  - `break;`: Since this glove is designed to play only one note at a time, as soon as we find *any* pressed finger, we break out of this loop. This means if you press multiple fingers, the one with the lowest index (e.g., Index finger if D10 is fingerPins[0]) will take priority.

3. **Play or Stop the Tone**:
   - `if (fingerToPlay != -1)`: If fingerToPlay is *not* -1 (meaning a finger *is* confirmed to be pressed):
     - `if (currentlyPlayingFinger != fingerToPlay)`: We check if this fingerToPlay is different from the currentlyPlayingFinger. This condition is true if:
       - A new finger is pressed.
       - A finger was held, the octave changed (which set currentlyPlayingFinger to -1), and the finger is *still* held.
       - No finger was playing, and now one is.
       If so, it's time to play or change the note:
       - `float frequency = baseNotes[fingerToPlay] * octaveMultipliers[currentOctave];`: We calculate the desired frequency. We take the baseNotes for the active fingerToPlay and multiply it by the octaveMultipliers for the currentOctave.
       - `tone(buzzerPin, frequency);`: This is the Arduino command that makes sound. It tells the buzzerPin to generate a sound wave at the calculated frequency. The sound will continue until noTone() is called.
       - `currentlyPlayingFinger = fingerToPlay;`: We update currentlyPlayingFinger to remember which finger is now making the sound.
       - `Serial.print(...)`: We print details about the note being played to the Serial Monitor for feedback.
   - `else`: If fingerToPlay *is* -1 (meaning no finger is currently confirmed to be pressed):
     - `if (currentlyPlayingFinger != -1)`: And if a note *was* playing before (i.e., currentlyPlayingFinger is not -1)...
       - `noTone(buzzerPin);`: We stop the sound on the buzzerPin.
       - `Serial.println("Stopping tone");`: Print a message.
       - `currentlyPlayingFinger = -1;`: We set currentlyPlayingFinger back to -1 to indicate that nothing is playing.

```
// Method to handle registering which finger is currently 'pressed'
// (in contact with the Thumb to complete circuit) and play notes
void handleFingerPresses() {
  // --- Debounce each finger input ---
  for (int i = 0; i < numFingers; i++) {
    int rawState = digitalRead(fingerPins[i]); // Read the raw state of the finger pin
```

```
    if (rawState != lastFingerRawState[i]) {
      // If state has changed, reset debounce timer for this finger
      lastFingerDebounceTime[i] = millis();
    }
    if ((millis() - lastFingerDebounceTime[i]) > debounceDelay) {
      // If the signal has been stable longer than the debounce delay,
      // and it's different from the current debounced state, update it.
      if (rawState != fingerDebouncedState[i]) {
        fingerDebouncedState[i] = rawState;
      }
    }
    lastFingerRawState[i] = rawState; // Save the current raw state for next time
  }

  // --- Determine which finger (if any) should be played ---
  int fingerToPlay = -1; // Assume no finger is pressed initially

  // Find the first (lowest index) pressed finger
  // This loop gives priority to fingerPins[0], then fingerPins[1], etc.
  for (int i = 0; i < numFingers; i++) {
    if (fingerDebouncedState[i] == LOW) { // LOW means pressed (after debouncing)
      fingerToPlay = i; // This finger should be played
      break;            // Since we only play one note, stop checking other fingers
    }
  }

  // --- Play or stop the tone based on which finger is active ---
  if (fingerToPlay != -1) { // If a finger is determined to be pressed
    if (currentlyPlayingFinger != fingerToPlay) {
      // If this is a new finger press, or if the note was stopped (e.g., by octave change)
      // and the finger is still held, then start a new tone.
      float frequency = baseNotes[fingerToPlay] * octaveMultipliers[currentOctave];
      tone(buzzerPin, frequency); // Play the calculated frequency on the buzzer
      currentlyPlayingFinger = fingerToPlay; // Remember which finger is now playing

      // Print information to Serial Monitor
      Serial.print("Playing Finger ");
      Serial.print(fingerToPlay + 1); // Adding 1 for human-readable finger number (1-4)
      Serial.print(" (Note: ");

      // Set the NeoPixel color based on which finger is playing
      strip.setPixelColor(0, fingerColors[fingerToPlay]); // <<< CHANGED
      strip.show();                                       // <<< CHANGED

      if(fingerToPlay == 0) Serial.print("G");
      else if(fingerToPlay == 1) Serial.print("C");
      else if(fingerToPlay == 2) Serial.print("E");
      else if(fingerToPlay == 3) Serial.print("A");
      Serial.print(") at Freq: ");
      Serial.println(frequency);
    }
    // If currentlyPlayingFinger == fingerToPlay, the note is already playing, so do
```

```
nothing.
  } else { // No finger is determined to be pressed
    if (currentlyPlayingFinger != -1) {
      // If a note was playing, but no finger is pressed now, stop the tone.
      noTone(buzzerPin); // Stop the sound
      Serial.println("Stopping tone");
      // Turn the NeoPixel off
      strip.clear();                    // <<< CHANGED
      strip.show();                     // <<< CHANGED
      currentlyPlayingFinger = -1; // Mark that no note is playing
    }
  }
}
```

And that's the complete walkthrough of the Music Glove code! To see it all in one file, please go to my Github Repo!

Testing the Code

Now the most important step of writing the code, testing! This is an essential step to ensure everything is working as expected before we put the electronics into the back piece permanently.

Without testing first, you won't know if any issues that arise later are related to the code, the circuit, or something else. It's very important in projects with e-textiles to test early and test often, which is what we'll be doing throughout this tutorial. There are many potential points of failure, so we should do our best to isolate them!

Setup for Testing

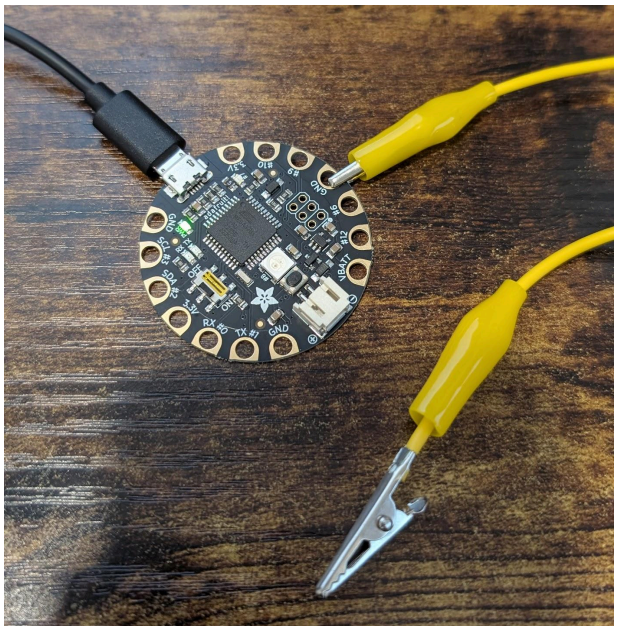
For this step, we'll need some Alligator Clips and the Flora V2 with the code pushed onto it.

Alligator Clips used in this tutorial - came in a bag with the Flora Budget Pack from Adafruit.

1. Testing the Finger Pins

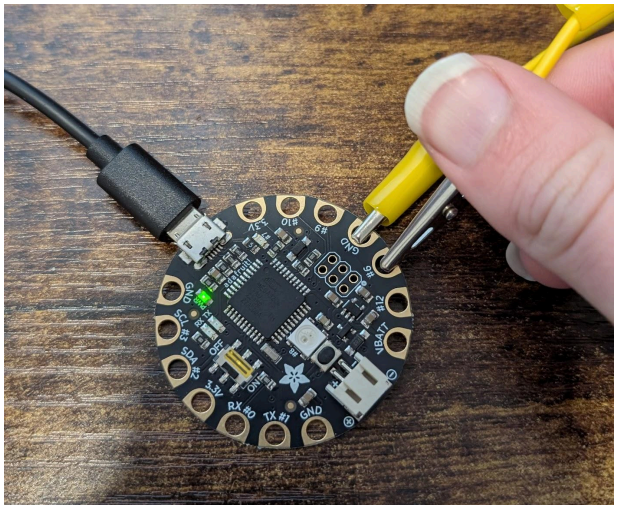
Once you have your alligator clips, pick one and connect one end to GND (Ground pin) on the Flora V2. There are multiple ground pins, you can select whichever one you would like. For this

example, I have chosen to go with the GND pin between the pins that'll be used for each of the Fingers on the glove.



Alligator clip connected to one of the GND pins on the Flora V2

Once you have your alligator clip connected to ground, use the other end to 'press' the pin for one of the fingers (In this case pin #6, the Ring Finger). Then remove the alligator clip from the pin and check your Serial Monitor in the Arduino IDE.



Pressing the alligator clip to the #6 pin (the Ring Finger pin)

You should see a these messages come up in the Serial Monitor:

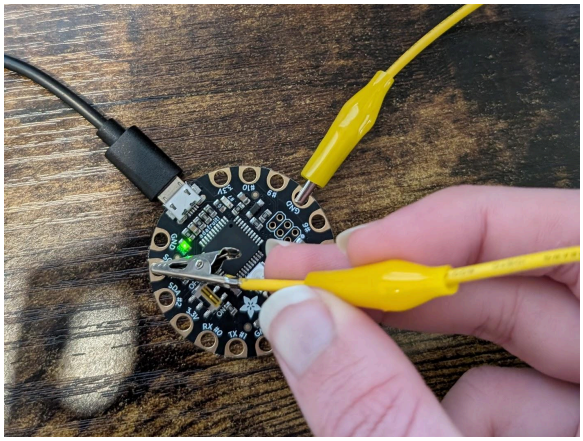
```
Playing Finger 3 (Note: E) at Freq: 329.63
Stopping tone
```

The first message should be from when you pressed the pin, and the second from when you release.

Now go through and check each finger pin to ensure they all are working as expected. Be sure to check that you only get the “Stopping tone” output when you release the connection! If you are seeing it before you release, up your Debouncing Delay by 10. If this keeps happening, a more in-depth debug may be required.

2. Testing the Button

Now, once you have confirmed that your Finger pins are working (they output the correct value to the Serial Monitor) it’s time to check the Button Pin (the pin that changes the Octave).
For this, take that same alligator clip, still connected to ground on one end, and press the other end against the “SDA #2” pin on the Flora. This is the pin for the button.



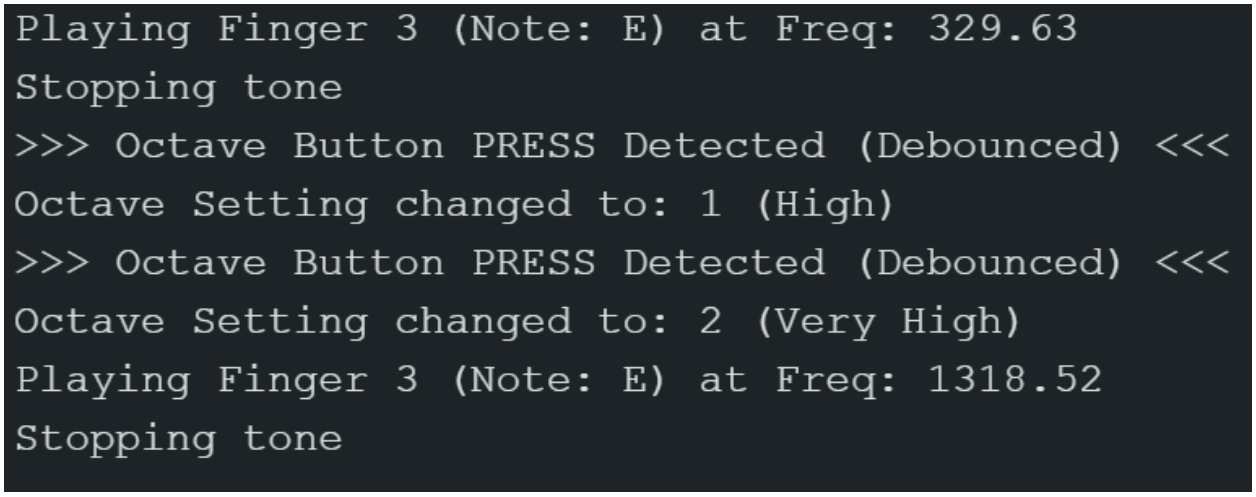
Pressing the alligator clip to the SDA #2 pin (button pin) to check functionality

Do this at least 3 times and check the Serial Monitor output. You should see the Octave Setting change from Middle, to High, to Very High, then back to Middle again.

```
>>> Octave Button PRESS Detected (Debounced) <<<
Octave Setting changed to: 1 (High)
>>> Octave Button PRESS Detected (Debounced) <<<
Octave Setting changed to: 2 (Very High)
```

Serial Monitor Output for switching octaves using the simulated button

Once you have done this and confirmed that the octaves are switching as expected, the next thing to check is that your fingers actually emit different sounds when set to different octaves! For this, all you need to do is change the octave to one that isn’t Middle (the default) then press the alligator clip connected to Ground to one of the finger pins. You should then see an output in your Serial Monitor which shows the expected tone output to the speaker (not connected yet).



Serial Monitor Output showing higher note played when octave is set to Very High

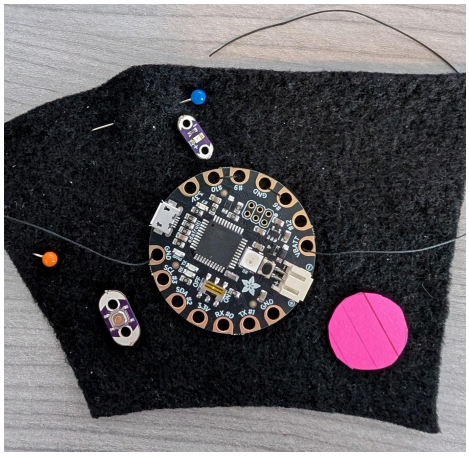
Once you've confirmed that all of your connections are working as expected and producing the Serial Monitor outputs that we expect, it's time to move on to the next step!

Putting it together (The Back Piece)

Now that we've cut our pieces and tested our code it's time to start putting the Back Piece together! For this, we'll need almost all of our electronic components (Flora V2, 4 LEDs, at least 5 Conductive Snaps [6th snap can be non-conductive], alligator clips, and conductive thread) as well as our sewing needle, non-conductive thread, and our felt Back Piece cut to size in the Design and Setup portion.

Laying out the Components

The first thing we're going to do is lay out a rough outline of where we want the components to go on the backpiece. We want to make sure we orient the Flora with the pins for the fingers oriented towards those finger locations, as well as give enough space around the Flora for the button, LED lights, and the Buzzer.



The Back Piece with the Flora V2 Board, Button, Buzzer (Pink cutout) and one of the LEDs laid out (NOTE: LEDs are no longer a part of this tutorial, please ignore). Pins mark conductive snap locations for index (blue) and thumb (orange)

Connecting the Components to the Backpiece

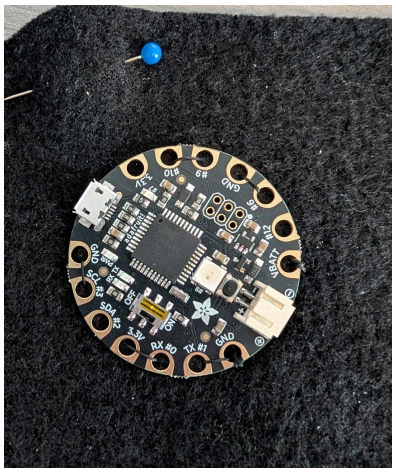
Once we have our parts laid out, we need to first attach the Flora to the backpiece. We will have the Flora visible in the final project, so make sure to sew it on the opposite side of the felt from where you will sew the conductive snaps.

Importantly, we will use a non-conductive thread for this part. We will be using the pin holes of the Flora to keep it sewn down, and attaching conductive thread between these could damage or destroy the Flora.

Back of the Flora, showing the thread used to attach it to the back piece.

This should be done through many or all of the holes in the Flora. This is critical for the stability of the Flora (and by extension the electronics we're attaching later!). In this case, I used most of

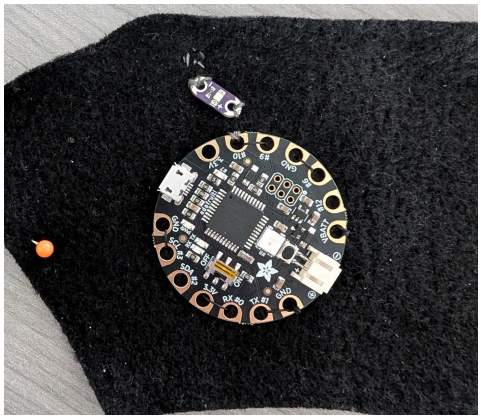
the holes for the Flora, but feel free to use whatever method works best for your back piece material.



Flora from the top down, showing the black non-conductive thread used to sew it down.

Once we've secured the Flora, our next step is to sew on the conductive snaps!
PIVOT:
No longer using in-line LEDs. Instead, moved to using the built-in NeoPixel on board on the Flora!
Now attach it to the snap! For this, start from the Flora and wrap the conductive thread around the pin (in this case pin #10) 3 times to ensure a solid connection (you don't want this to disconnect! More is better than less!) then go through the felt and come back up through the top through the holes on the conductive snap enough times to create a solid connection.

Now wrap the conductive thread another few times, then go back through the felt to the underside. Once on the underside, make sure both ends of that conductive thread are back there (If the other end isn't, put it through the back from inside the Flora pin hole again!) then tie a knot to secure it.

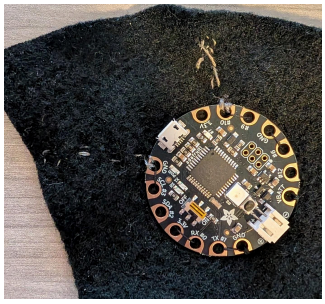


Flora with LED and conductive snap sewed down from the top down. NOTE: LEDs are no longer a part of this tutorial

Flora with conductive snap sewed down from the underside. There's a lot of slack in the conductive thread here, which should be cut down before moving on!

This process is the same for each of the other fingers! Make sure when doing this with the thumb to connect it to the GND pin, not another pin! This is how we complete the circuit and get our code to register that a sound should be played.

Underside of the backpiece showing the thumb and index finger connection snaps



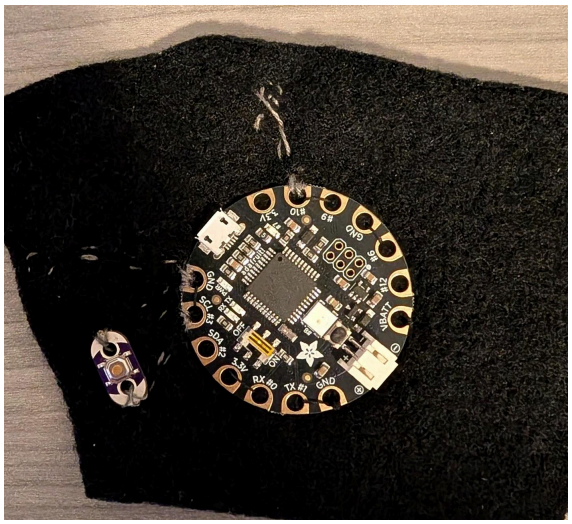
Top of the backpiece showing the connected index finger and the thumb connected to GND (ground)

Connecting the Button

Next we're going to connect the button! This is how the glove will change octaves when in use.

To do this, we need to place the button on the same side of the backpiece as the Flora, then we need to connect it to the Ground pin and to the SCL #3 pin (Pin 3) using conductive threads. We need to make sure to use different threads for each side of the button to ensure we don't short the circuit.

Please make sure to also cut the knots close to the knot to keep the stray threads from touching each other on the underside of the backpiece. This can cause a short circuit! It's getting a bit tight near the button now, so we need to be intentional with our stitches. Below is an image of the backpiece from the top once the button is added!

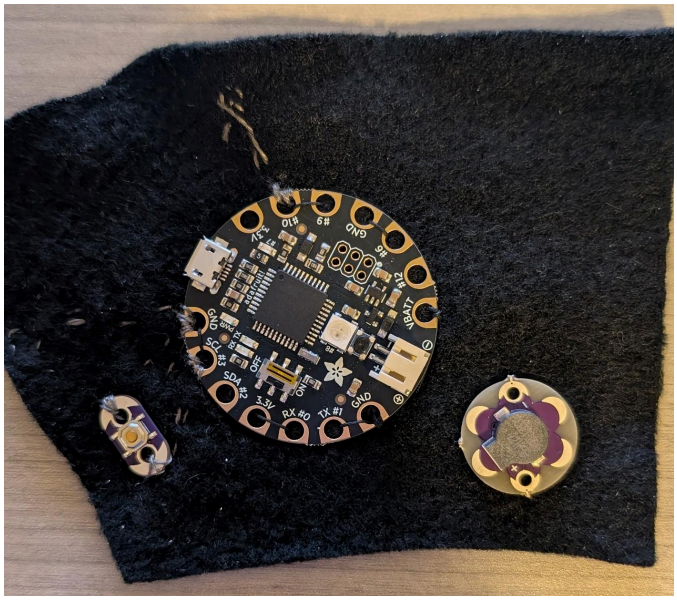


Top down view of the backpiece with the button added.

Connecting the Buzzer

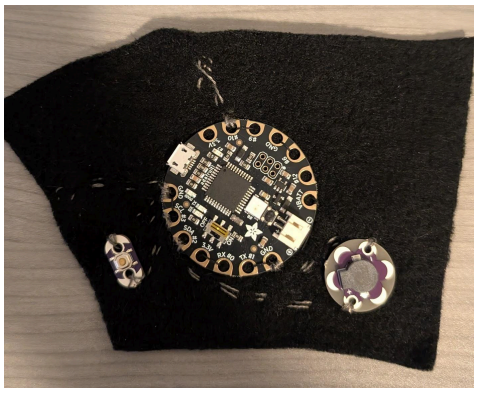
Now we need to connect the buzzer! This is how the glove will make sound, so it's essential that this is done right! For this, similar to the button, we will be connecting the '-' end to ground (GND) and the + end to SDA #2 (Pin 2) on the Flora. For this piece, we will be using a different GND pin than for the thumb and button since the buzzer is quite far away from that ground pin, and we don't want to deal with crossing conductive threads if we don't have to!

First thing's first, we need to figure out where the buzzer is going to go. Below, there is an image of how I chose to position my buzzer, you can pick the same place or somewhere different! It all depends on how big your glove is. For this, you just want to make sure the buzzer is out of the way and not interfering with any of the other components while still close enough to the #2 pin to not require long conductive thread runs.



Placement of the Buzzer on the backpiece before sewing it down.

You will want to make sure that before you sew down the buzzer it won't interfere with plugging in/taking our the battery! This is near the battery port, so move it as needed to prevent issues.



Buzzer sewn onto the backpiece, attached to new GND pin and SDA #2 (Pin 2)

Now that we've finished adding all of the electronic components, go ahead and add the remaining conductive snaps for each of the fingers the same way we did the index finger at the start of Putting It Together (The Back Piece)

Once you've finished that, it's time to finish up the glove, then we're done!

Putting it together (The Glove!)

This is where we'll put the glove component together! We already set up the index finger and the thumb earlier in this tutorial, so now it's time to do the rest of the fingers!

Since we have all of the locations for the snaps already set out (We added them to the backpiece) we just need to attach the backpiece to the glove and put pins where the snaps are!

From there, all we need to do is attach the snaps and run conductive thread up to the fingertips just like we did for the index finger.

Add the rest of the snaps the same way we did earlier, by pinning the locations below each finger knuckle and measuring them through the backpiece!

Pins used to mark the remaining conductive snap locations through the backpiece. From here, repeat the steps from the index finger above to complete the glove!

Wrap up

Now we're ready to wrap up! By this point, you should have 4 fingers and the thumb connected to the backpiece with conductive snaps, each one at the base of a knuckle. At this point, you can optionally add a snap (conductive or non-conductive) at the base of the right side (for a right

handed glove) near or under the buzzer to help keep the back piece stable and attached during use!

If everything worked as expected, you should now be able to play different sounds with each finger, which come through the buzzer on the backpiece, and be able to change the octave between 3 presets using the button!

Thank you for reading this tutorial! If you liked it, please feel free to use it as inspiration for a bigger, better project (I'd appreciate a citation for the inspiration 🙂!)

Future Upgrades/Next Steps for You!

There are many ways to improve this project, one of which is being able to play multiple notes at the same time. For this, you shouldn't need any additional hardware, just some changes to the code to account for multiple complete circuits. How you handle multiple notes is up to you! A good start could be averaging the pitch between them and seeing how that sounds, then go from there!

If you take this project and build onto it, best of luck in your endeavor!