Creating and Modifying an Electric Violin Signal

by

Leann Mayberry

Progress Design Report

VIP Team: Retrofuturistic Hardware

Georgia Institute of Technology

April 2019

ABSTRACT

The goal of this project is to create an electronic system that can be added to a violin without damaging or permanently modifying the violin body or bow. This electronic system will be designed for two purposes; to modify a live performance and as a teaching tool for violin students.

This document details the design process of the Violin electronic system including devices researched over the semester, potential design concepts, and early design decisions. This is a progress design report, as the final project is not complete. Therefore, this report includes work done to this point as well as discussion of how the team plans to continue work in the future. This document is intended for use by future members of the Retrofuturistic Hardware Violin VIP team as a foundation of information to continue the project.

CONTENTS

1. Introduction

The idea for this design project came from a desire to learn violin combined with the motivation to create a project that combines vintage equipment with modern technology. The Retrofuturistic Hardware VIP team is inspired by retrofuturistic aesthetic, which can be seen in this project as it encompasses the traditional violin alongside potential futuristic soundscapes and learning opportunities.

Current technology to create and modify electric Violin sound includes piezo-electric pickups, pedal boards, standard electric violins, MIDI electric violins, and multi-instrument processing tools. The drawback of this current technology is that it requires significant user research to combine the available tools and is expensive to set up. For this project we are investigating methods of creating effects similar to those produced by current technology by placing sensors and keys directly on the violin, allowing the system to be more tightly integrated and cost effective.

Current technology to help students improve their violin technique without the aid of a teacher includes apps that listen to a student practicing to give feedback and correct intonation, ear-training apps, and visual and audio tuners. These existing technologies rely on the sound the violin produces, but for the violin project we plan to use sensor data to determine if correct technique is being used.

2. Hardware Requirements

2.1 Adafruit Feather 32u4 Microcontroller

The Adafruit Feather 32u4 is the primary device used in this project to collect signals from the violin. The 32u4 uses Bluetooth Low Energy to transmit information and can also act like a MIDI device by sending MIDI packages over Bluetooth. The board is Arduino-compatible, so Arduino and Adafruit libraries are used to set the operation of the sensors as well as to collect and process data from the board.

The 32u4 operates at 3.3V with an 8MHz clocking rate. This is adequate for the simple testing being done at this stage of the project, but in the future a 16MHz clocking rate may be desired in order to improve the speed at which data is transferred if we use it to process live sound.

The 32u4 also transmits sensor data collected by the MRP121 Capacitive Touch Sensor, the APDS 9960 Gesture Sensor, and the LSM9DS1. These sensors are described in further detail in following sections.

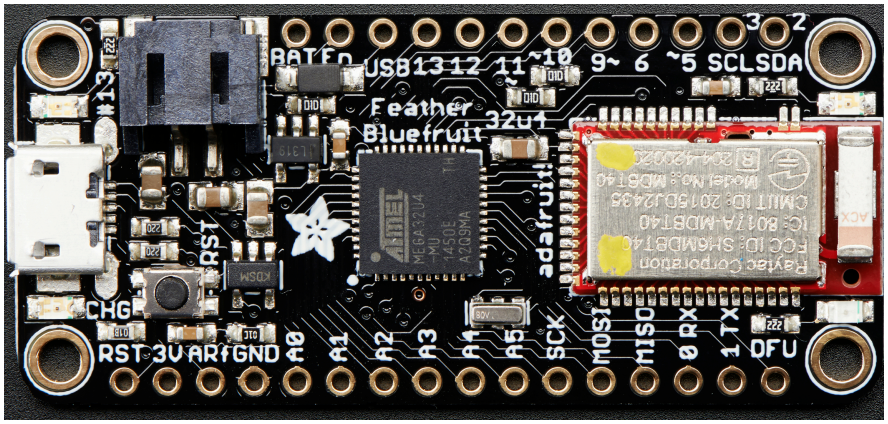

Figure 1. Adafruit Feather 32u4 [1].

2.1.1 Connecting the Adafruit 32u4 as a MIDI Bluetooth Device Using MIDI Studio on Mac OS

To set up the 32u4 as a MIDI Bluetooth Device it needs to have an MIDI support uploaded. The example used to test the Bluetooth set up process is code used in Adafruit's "Capacitive Touch Drum Machine" tutorial [2]. To load this example the SimpleSynth, FifteenStep, and Adafruit NeoPixel Arduino libraries are required. The specific sketch used is called ble_neopixel_mpr121, which is supplied by Adafruit. This code can be found in Appendix A and has not been modified from the original version.

After the code has been uploaded to the 32u4 MIDI Studio is opened and "Configure Bluetooth" is selected. The device now appears as a Bluetooth MIDI Input/Output and can be used in any DAW that connects to devices through MIDI Studio. To test this the 32u4 was used as a MIDI Input in Logic Pro. The process to connect the Adafruit Bluefruit LE as a Bluetooth device on Mac can be seen in Figures 2-3 and setting this device as an input in Logic Pro X can be seen in Figure 4.

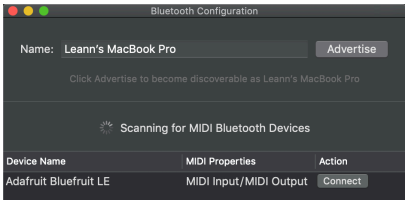

Figure 2. Bluetooth Configuration Menu Displaying Adafruit Bluefruit LE (32u4) as a Connectable Device.

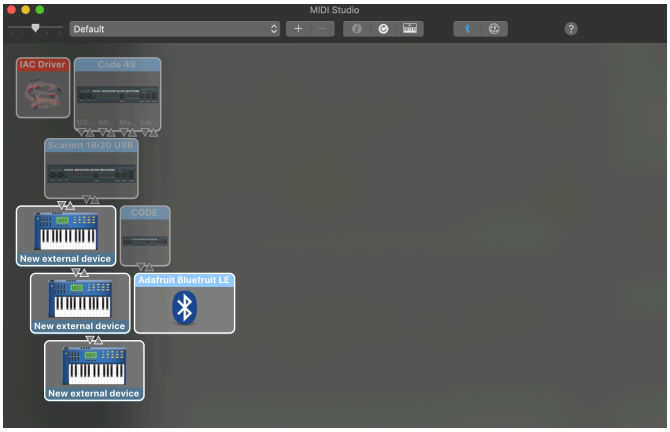

Figure 3. MIDI Studio on Mac OS Displaying Adafruit Bluefruit LE (32u4) as a Connected Device.

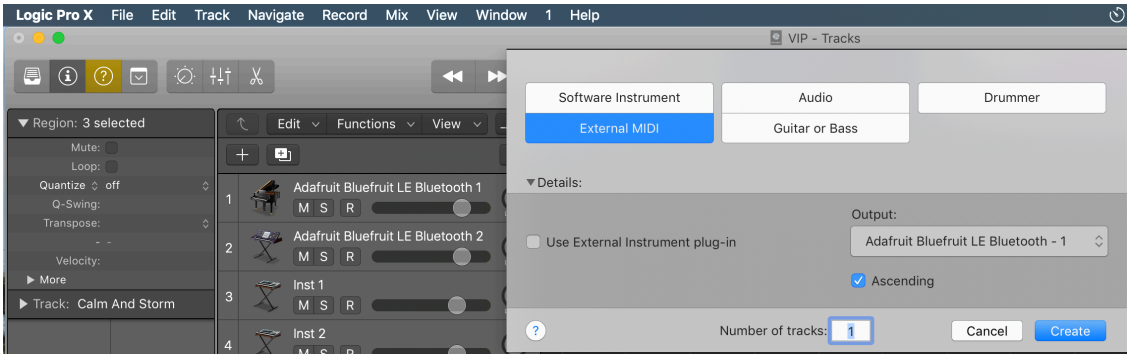

Figure 4. Logic Pro X Track Creator Confirming the Successful Setup of the 32u4 as a Bluetooth MIDI Device.

2.2 Adafruit MPR121 Capacitive Touch Sensor

The MPR121 Capacitive Touch Sensor is the primary sensor used for testing the Bluetooth MIDI program as it is the same sensor used for Adafruit's "Capacitive Touch Drum Machine" tutorial.

The sensor consists of 12 capacitive touch channels and can be configured for more or less sensitivity than the default setting. It operates at 3V but includes a regulator and 12C level shifting so it can be used with a 3V or 5V microcontroller.

To test the functionality of the MPR121 the Adafruit MPR121 library can be used, in particular the example sketch to test this sensor is MPR121test supplied by Adafruit. This code can be found in in Appendix B and has not been modified from the original version. When this sketch is uploaded the Arduino Serial Monitor displays which capacitive sensors have been touched/released as seen in Figure 6.

Using the MPR121 to create MIDI data using the sketch described in Section 2.1.1 assigns the functionality of the capacitive sensors described in the "Capacitive Touch Drum Machine" tutorial to sensors 0-5 on the board. This can be used to produce sounds in Logic Pro X or similar DAWs. An example of notes being produced in Logic Pro X using the capacitive touch sensor can be seen in Figure 7. The current functionality of the capacitive touch sensor as a Drum Machine will likely not be used in the final violin project, but it establishes the potential to successfully send MIDI data and opens up the option to assign more useful functions to the capacitive touch sensors later.

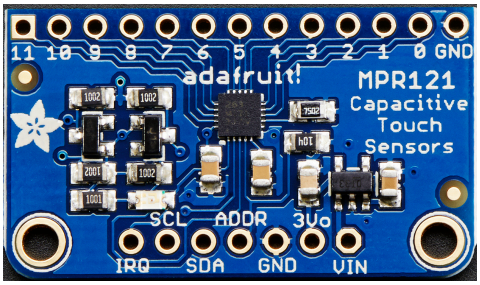

Figure 5. MPR121 Board [3].

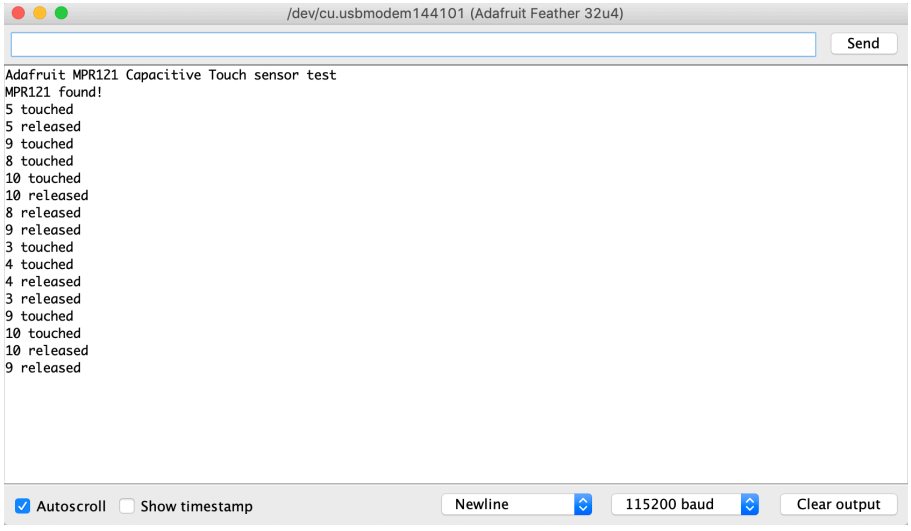

Figure 6. Arduino Serial Monitor Output Showing Capacitive Touch Sensor Data.

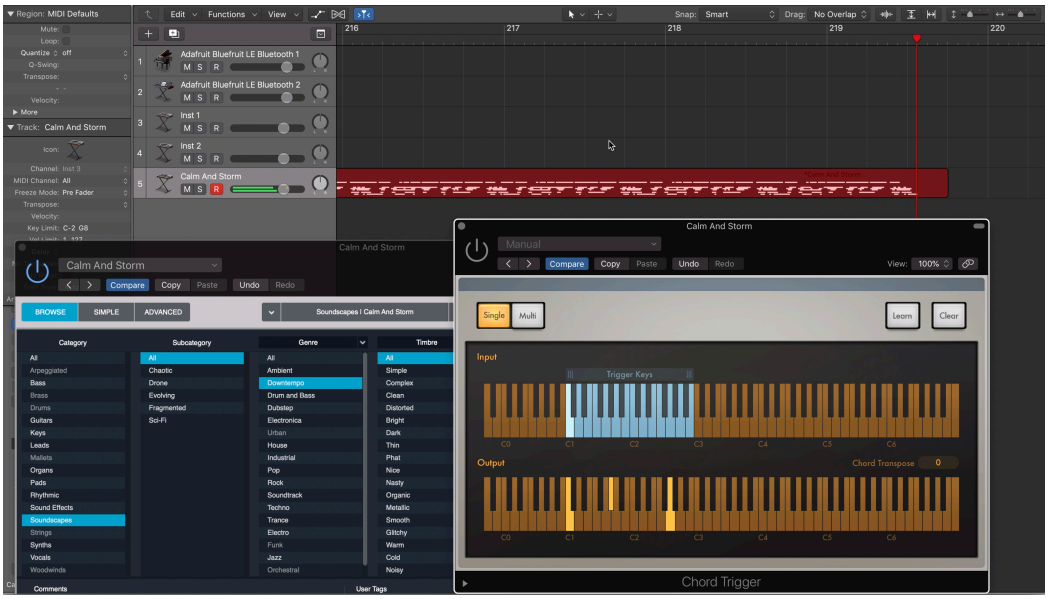

Figure 7. MIDI Data Created Using Capacitive Touch Sensor in Logic Pro X. The keyboard display shows the keys activated at the time the screenshot was taken and the track in red shows the full MIDI input over time.

2.3 Adafruit APDS 9960 Proximity, Light, RGB, and Gesture Sensor

The APDS 9960 has potential to be used to sense when the bow passes a particular point on the violin or as a controller that can be activated by the violin player to produce pre-set effects such as filters, distortion, or looping.

To test the Gesture Sensor the Adafruit APDS9960 library can be used, in particular the gesture_sensor example sketch shows a function of the sensor that could be useful if the sensor is attached to the body of a violin. The output of this sketch can be seen in Figure 9. The violinist could move their hand, or the bow, across the region above the sensor to add or remove an effect. This code can be found in in Appendix C and has not been modified from the original version.

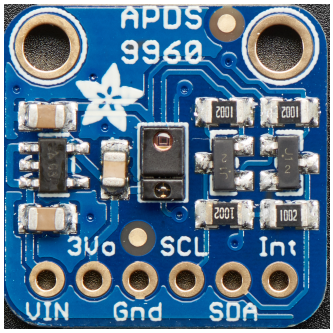

Figure 8. APDS 9960 Board [4].

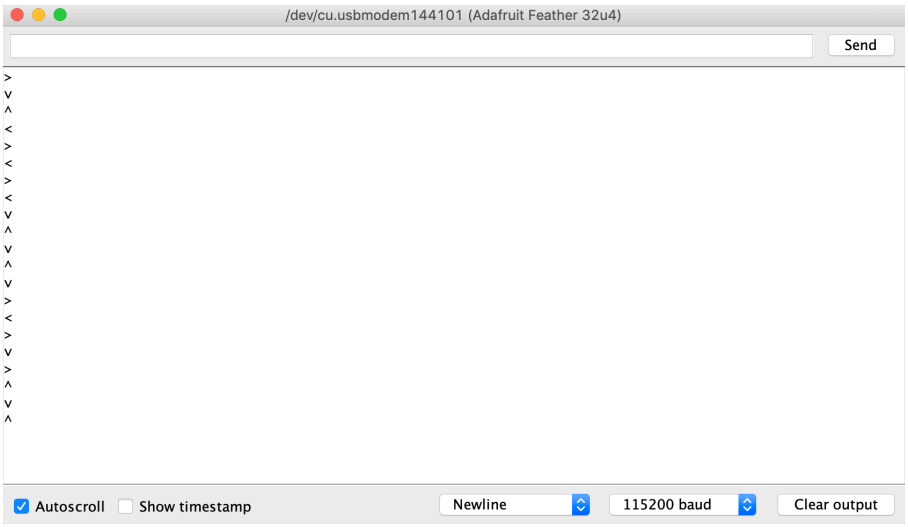

Figure 9. Arduino Serial Monitor Output Showing Gesture Sensor Data.

2.4 Adafruit LSM9DS1 9-DOF Accel/Mag/Gyro+Temp Breakout Board

The LSM9DS1 has several potential uses. First, it could be placed on the violin or bow to capture movement data that could be used to apply active effects to the violin sound, with intensity depending on variables such as how quickly or in which direction the violin or bow is accelerating. Second, it could be placed on the violin or bow to monitor the player's technique in a teaching application.

To test the board the Adafruit LSM9DS1 library can be used, in particular the lsm9ds1 example. The output from running this sketch can be seen in Figure 11. This code can be found in in Appendix D and has not been modified from the original version.

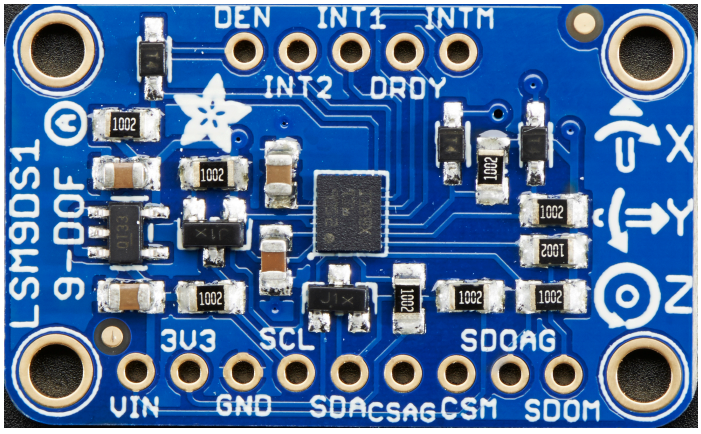

Figure 10. LSM9DS1 Board [5].

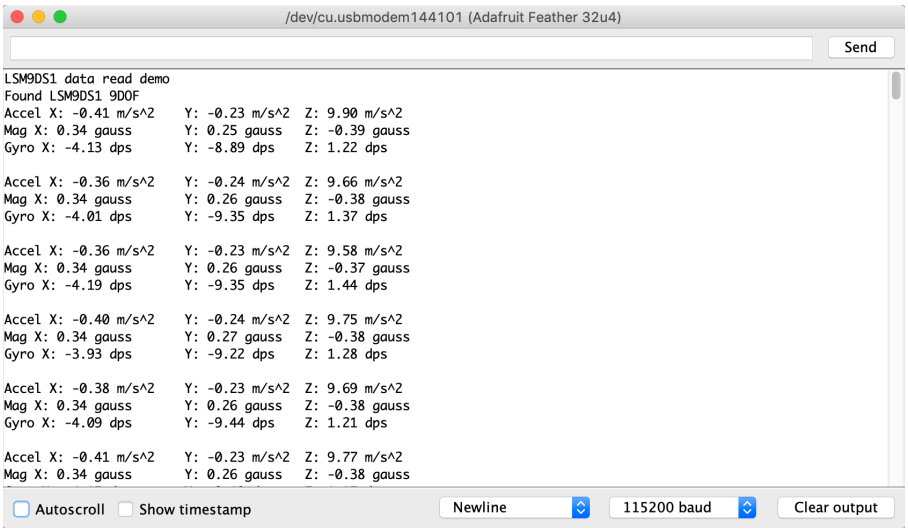

Figure 11. Arduino Serial Monitor Output Showing LSM9DS1 Sensor Data.

2.5 Piezo-Electric Pickup

Piezo-Electric pickups function by capturing the pressure of the vibration from the instrument they are used on. Because of this, they are commonly used on instruments with nylon strings that cannot function with a magnetic pickup.

To make it possible to send the picked-up violin signal over Bluetooth, we are hoping to use basic Piezoelectric Sensors to transmit voltage signals through the Bluefruit 32u4 Microcontroller. These signals can then be processed into a MIDI version of the violin signal in Logic Pro X. If the sampling rate of 8MHz is not high enough an alternate solution is to use a Mono Jack that can be plugged into an analog speaker that can transmit the data to Logic Pro X through USB connection.

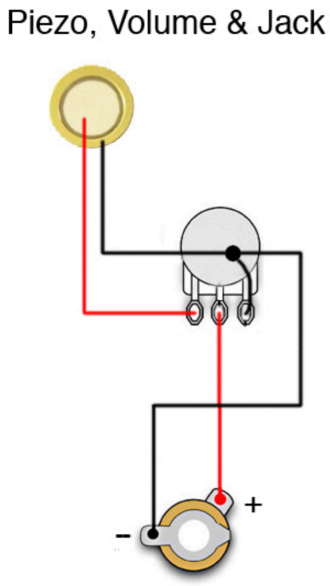

Figure 12. Example Connection of a Piezo-Electric Pickup [6].

3. Design

3.1 PCB Design and Placement on Violin

To test the functionality and potential of the microcontroller and sensors the boards were soldered with pins and attached to a breadboard. Now that functionality and required pin connections are determined a PCB board can be printed. This PCB board will connect the 3V pin on the 32u4 to the Vin pins on the sensors, the SCL and SDA pins to their respective pins on the sensors, and all of the boards to a common ground.

The PCB board will be relatively small and will be placed under the fingerboard of the violin, in the space where the fingerboard hovers above the violin's body.

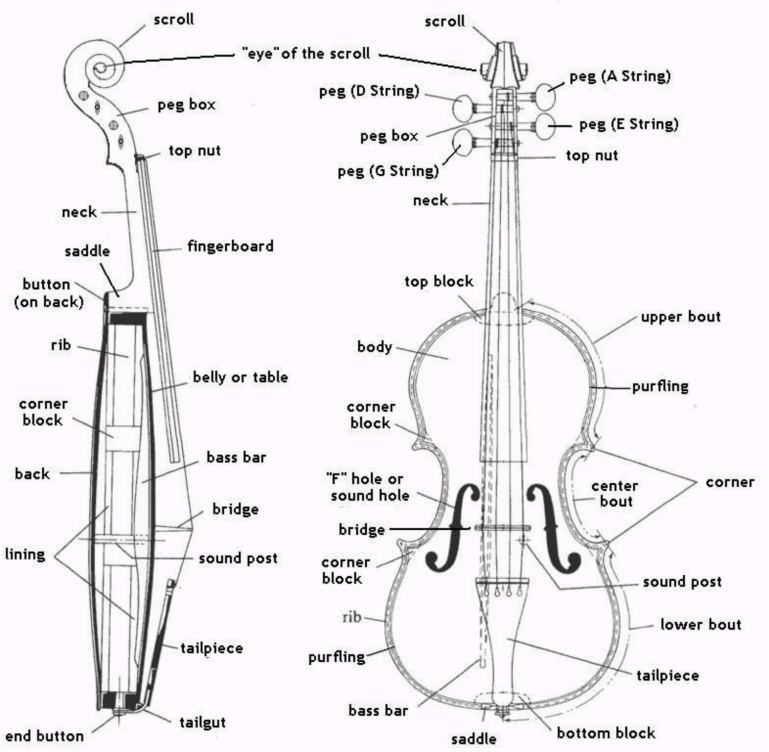

Figure 13. Violin Diagram Displaying Various Parts of the Violin Including the Fingerboard and Body [7].

3.2 Capacitive Touch Sensor Layout

To allow the violin player quick access to effects that will be added by touching the various channels on the Capacitive Touch Sensor, the pins will be connected to conductive tape that will be placed along the lower bout and body of the violin.

Touching one of the sensors should apply the effect programmed to that channel and touching it again should remove the effect. The goal of this is to make effects that would normally be added using foot pedals more accessible by placing them on the body of the violin.

4. Conclusion and Future Work

Over the course of the semester we were able to collect the resources necessary to start building an electronic system to add to a violin and design a rough version of that system. The microcontroller and sensors studied proved to be useful in transmitting and creating sensor and MIDI data.

Next, we plan to create a PCB and install it in a violin along with the capacitive sensors as well as determine which piezo-pickup method is necessary. After this, we will modify the code used to program the 32u4 to use the sensor data to modify the picked-up violin sound.

5. References

[1] Adafruit. Adafruit Feather 32u4 Bluefruit LE: Overview. Adafruit. [Online]. Available: https://learn.adafruit.com/adafruit-feather-32u4-bluefruit-le/overview. Accessed: April 9, 2019.

[2] Adafruit. Capacitive Touch Drum Machine. Adafruit. [Online]. Available: https://learn.adafruit.com/capacitive-touch-drum-machine/overview. Accessed: April 9, 2019.

[3] Adafruit. Adafruit MPR121 12-Key Capacitive Touch Sensor Breakout Tutorial: Overview. Adafruit. [Online]. Available: https://learn.adafruit.com/adafruit-mpr121-12-key-capacitive-touch-sensor-breakout-tutorial. Accessed: April 9, 2019.

[4] Adafruit. Adafruit APDS9960 breakout: Overview. Adafruit. [Online]. Available: https://learn.adafruit.com/adafruit-apds9960-breakout. Accessed: April 9, 2019.

[5] Adafruit. Adafruit LSM9DS1 Accelerometer + Gyro + Magnetometer 9-DOF Breakout: Overview. Adafruit. [Online]. Available: https://learn.adafruit.com/adafruit-lsm9ds1-accelerometer-plus-gyro-plus-magnetometer-9-dof-breakout/overview. Accessed: April 9, 2019.

[6] Ted Crocker. Wiring Diagrams & Schematics. [Online]. Available: http://www.handmademusicclubhouse.com. Accessed: April 9, 2019.

[7] ———. The Anatomy Of A Violin. [Online]. Available: http://www.violinstudent.com/violinmap.html. Accessed: April 9, 2019.

6. Appendice

Appendix A: Bluefruit MIDI Setup Code

```
// ----------------------------------------------------------------------------
//
// ble_neopixel_mpr121.ino
//
// A MIDI sequencer example using two chained NeoPixel sticks, a MPR121
// capacitive touch breakout, and a Bluefruit Feather.
//
// 1x Feather Bluefruit LE 32u4: https://www.adafruit.com/products/2829
// 2x NeoPixel Sticks: https://www.adafruit.com/product/1426
// 1x MPR121 breakout: https://www.adafruit.com/products/1982
//
// Required dependencies:
// Adafruit Bluefruit Library: https://github.
com/adafruit/Adafruit_BluefruitLE_nRF51
// Adafruit NeoPixel Library: https://github.com/adafruit/Adafruit_NeoPixel
// Adafruit MPR121 Library: https://github.com/adafruit/Adafruit_MPR121_Library
//
// Author: Todd Treece <todd@uniontownlabs.org>
// Copyright: (c) 2015-2016 Adafruit Industries
// License: GNU GPLv3
//
// ----------------------------------------------------------------------------
#include "FifteenStep.h"
#include "Adafruit_NeoPixel.h"
#include "Wire.h"
#include "Adafruit_MPR121.h"
#include "Adafruit_BLE.h"
#include "Adafruit_BluefruitLE_SPI.h"
#include "Adafruit_BLEMIDI.h"
#include "BluefruitConfig.h"

#define FACTORYRESET_ENABLE         1
#define MINIMUM_FIRMWARE_VERSION    "0.7.0"

Adafruit_BluefruitLE_SPI ble(BLUEFRUIT_SPI_CS, BLUEFRUIT_SPI_IRQ,
BLUEFRUIT_SPI_RST);

#define NEO_PIN  6
#define LEDS     16
#define TEMPO    60
#define BUTTONS  6
#define IRQ_PIN  A4

// sequencer, neopixel, & mpr121 init
FifteenStep seq = FifteenStep(1024);


Adafruit_NeoPixel pixels = Adafruit_NeoPixel(LEDS, NEO_PIN, NEO_GRB + NEO_KHZ800);
Adafruit_MPR121 cap = Adafruit_MPR121();
Adafruit_BLEMIDI blemidi(ble);

// keep track of touched buttons
uint16_t lasttouched = 0;
uint16_t currtouched = 0;

// start sequencer in record mode
bool record_mode = true;

// set command states to off by default
bool command_mode = false;
bool tempo_mode = false;
bool shuffle_mode = false;
bool pitch_mode = false;
```

```
bool velocity_mode = false;
bool channel_mode = false;
bool step_mode = false;

// keep pointers for selected buttons to operate
// on when in note and velocity mode
int  mode_position = 0;
bool position_selected = false;

// prime dynamic values
int channel = 0;
int pitch[] = {36, 38, 42, 46, 44, 55};
int vel[] = {100, 80, 80, 80, 40, 20};
int steps = 16;

bool isConnected = false;

// A small helper
void error(const __FlashStringHelper*err) {
  Serial.println(err);
  while (1);
}
void connected(void)
{
  isConnected = true;
  Serial.println(F(" CONNECTED!"));
}

void disconnected(void)


{
  Serial.println("disconnected");
  isConnected = false;
}

void setup() {

  Serial.print(F("Initialising the Bluefruit LE module: "));

  if ( !ble.begin(VERBOSE_MODE) )
  {
    error(F("Couldn't find Bluefruit, make sure it's in CoMmanD mode & check
wiring?"));
  }
  Serial.println( F("OK!") );

  if ( FACTORYRESET_ENABLE )
  {
    /* Perform a factory reset to make sure everything is in a known state */
    Serial.println(F("Performing a factory reset: "));
    if ( ! ble.factoryReset() ) {
      error(F("Couldn't factory reset"));
    }
  }

  //ble.sendCommandCheckOK(F("AT+uartflow=off"));
  ble.echo(false);

  Serial.println("Requesting Bluefruit info:");
  /* Print Bluefruit information */
  ble.info();

  /* Set BLE callbacks */
  ble.setConnectCallback(connected);
  ble.setDisconnectCallback(disconnected);
```

```
  Serial.println(F("Enable MIDI: "));
  if ( ! blemidi.begin(true) )
  {
    error(F("Could not enable MIDI"));
  }

  ble.verbose(false);
  Serial.print(F("Waiting for a connection..."));


  // set mpr121 IRQ pin to input
  pinMode(IRQ_PIN, INPUT);

  // bail if the mpr121 init fails
  if (! cap.begin(0x5A))
    while (1);

  // start neopixels
  pixels.begin();
  pixels.setBrightness(80);

  // start sequencer and set callbacks
  seq.begin(TEMPO, steps);
  seq.setMidiHandler(midi);
  seq.setStepHandler(step);

}

void loop() {

  // if mpr121 irq goes low, there have been
  // changes to the button states, so read the values
  if(digitalRead(IRQ_PIN) == LOW)
    readButtons();

  // this is needed to keep the sequencer
  // running. there are other methods for
  // start, stop, and pausing the steps
  seq.run();

}

/////////////////////////////////////////////////////////////////////////////
//                                                                         //
//                       BUTTON PRESS HANDLERS                             //
//                                                                         //
/////////////////////////////////////////////////////////////////////////////

// deal with note on and off presses
void handle_note() {

  for (uint8_t i=0; i < BUTTONS; i++) {

    // note on check
    if ((currtouched & _BV(i)) && !(lasttouched & _BV(i))) {



      // play pressed note
      midi(channel, 0x9, pitch[i], vel[i]);

      // if recording, save note on
      if(record_mode)
        seq.setNote(channel, pitch[i], vel[i]);

    }
```

```
    // note off check
    if ( !(currtouched & _BV(i)) && (lasttouched & _BV(i)) ) {

      // play note off
      midi(channel, 0x8, pitch[i], 0x0);

      // if recording, save note off
      if(record_mode)
        seq.setNote(channel, pitch[i], 0x0);

    }

  }

}

// allow user to select which command mode to enter
void handle_command() {

  // if we aren't in one of the general
  // command modes, we need to select a command
  if(! mode_selected()) {
    select_mode();
    return;
  }

  // tempo and shuffle are handled below
  if(!tempo_mode && !shuffle_mode) {
    handle_change();
    return;
  }

  // increase tempo or shuffle
  if(currtouched == 0x20) {

    if(tempo_mode)
      seq.increaseTempo();
    else if(shuffle_mode)
      seq.increaseShuffle();

  } else if(currtouched == 0x10) {

    if(tempo_mode)
      seq.decreaseTempo();
    else if(shuffle_mode)
      seq.decreaseShuffle();

  }

}

// select which mode to enter
void select_mode() {

  // switch pads 1-6
  switch(currtouched) {

    case 0x1:
      if(lasttouched != 0x3)
        tempo_mode = true;
      break;
    case 0x2:
      if(lasttouched != 0x3)
        shuffle_mode = true;
      break;
    case 0x4:
      position_selected = true;
```

```
      step_mode = true;
      break;
    case 0x8:
      position_selected = true;
      channel_mode = true;
      break;
    case 0x10:
      pitch_mode = true;
      break;
    case 0x20:
      velocity_mode = true;
      break;

  }

}

// pass the appropriate values to the change value helper
void handle_change() {

  byte rgb[] = {0,0,0};
  int display;

  // no button index selected, wait until it has
  // been selected before changing values
  if(! position_selected) {
    get_position();
    return;
  }

  // set up values to increment or decrement
  if(pitch_mode) {

    int last = pitch[mode_position];
    change_value(pitch[mode_position], 127);
    display = map(pitch[mode_position], 0, 127, 0, LEDS);
    rgb[1] = 64;

    if(last != pitch[mode_position])
      midi(channel, 0x9, pitch[mode_position], vel[mode_position]);

  } else if(velocity_mode) {

    int last = vel[mode_position];
    change_value(vel[mode_position], 127);
    display = map(vel[mode_position], 0, 127, 0, LEDS);
    rgb[2] = 64;

    if(last != vel[mode_position])
      midi(channel, 0x9, pitch[mode_position], vel[mode_position]);

  } else if(channel_mode) {
    change_value(channel, 15);
    display = channel + 1;
    rgb[1] = 32;
    rgb[2] = 32;
  } else if(step_mode) {
    change_value(steps, FS_MAX_STEPS);


     seq.setSteps(steps);
     display = steps;
     rgb[0] = 32;
     rgb[2] = 32;
   }
```

```
  if(display % LEDS != 0)
    display = display % LEDS;

  flash(0,0,0);
  show_range(0, display, rgb[0], rgb[1], rgb[2]);

}

// common method for increasing or decreasing values
int change_value(int &current, int max_val) {

  // increasing or decreasing value?
  if(currtouched == 0x10)
    current = current > 0 ? current - 1 : 0;
  else if(currtouched == 0x20)
    current = current < max_val ? current + 1 : max_val;

  return current;

}

////////////////////////////////////////////////////////////////////////////
//                                                                        //
//                          SEQUENCER CALLBACKS                           //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

// called when the step position changes. both the current
// position and last are passed to the callback
void step(int current, int last) {

  // if we are in a command mode, flash command
  if(command_mode) {
    mode_flash(current);
    return;
  }

  note_flash(current);

}

// the callback that will be called by the sequencer when it needs
// to send midi commands over BLE.
void midi(byte channel, byte command, byte arg1, byte arg2) {

  // init combined byte
  byte combined = command;

  // shift if necessary and add MIDI channel
  if(combined < 128) {
    combined <<= 4;
    combined |= channel;
  }

  blemidi.send(combined, arg1, arg2);

}
```

```
////////////////////////////////////////////////////////////////////////////////
//                                                                            //
//                          NEOPIXEL HELPERS                                  //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////

// handles flashing for the different mode states
void mode_flash(int current) {

  // bail if we don't need to flash
  if(position_selected)
    return;

  byte rgb[] = {0,0,0};

  // set colors for modes
  if(pitch_mode) {
    rgb[1] = 64;
  } else if(velocity_mode) {
    rgb[2] = 64;
  } else if(tempo_mode) {
    rgb[0] = 32;
    rgb[1] = 32;
    rgb[2] = 32;
  } else if(shuffle_mode) {
    rgb[0] = 32;
    rgb[1] = 32;
  } else {
    rgb[0] = 64;
  }

  // LEDs on when it's an even step
  if((current % 2) == 0)
    flash(rgb[0],rgb[1],rgb[2]);
  else
    flash(0,0,0);

}

void note_flash(int current) {

  byte rgb[] = {0,0,0};

  // all LEDs off
  flash(0, 0, 0);

  // make sure we stay within the led count
  current = current % LEDS;

  // highlight quarter notes
  if(current % 4 == 0) {
    // red quarter notes in record mode
    // bright blue in play mode
    if(record_mode)
      rgb[0] = 255;
    else
      rgb[2] = 255;
  } else {
    // dim blue sixteenths
    rgb[2] = 64;
  }

  // highlight note
  show_range(current, current + 1, rgb[0], rgb[1], rgb[2]);

}
```

```
// sets all pixels to passed RGB values
void flash(byte r, byte g, byte b) {
  show_range(0, LEDS, r, g, b);
}


// sets range of pixels to RGB values
void show_range(int start, int last, byte r, byte g, byte b) {

  for (; start < last; start++)
    pixels.setPixelColor(start, pixels.Color(r,g,b));

  pixels.show();

}

////////////////////////////////////////////////////////////////////////////
//                                                                        //
//                      GENERAL UTILITY FUNCTIONS                         //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

// read the currently touched buttons and detect any
// pressed patterns that match command states. this
// set of states is set up for 6 pads, but you could
// easily modify the hex values to match the ammount
// pads you are using
void readButtons() {

  // read current values
  currtouched = cap.touched();

  switch(currtouched) {

    case 0x21: // stop all notes. pads 1,6
      seq.panic();
      break;

    case 0x3: // command mode. pads 1,2
      clear_modes();
      command_mode = command_mode ? false : true;
      break;

    case 0x18: // toggle record mode. pads 4,5
      record_mode = record_mode ? false : true;
      break;

    case 0x30: // pause toggle. pads 5,6
      seq.pause();
      break;

    default:
      // if it's not a command pattern, call route
      // it to the proper handler
      if(command_mode)
        handle_command();
      else
        handle_note();

  }

  // save current values to compare against in the next loop
  lasttouched = currtouched;

}
```

```
// get the index of the pressed button. this is used by
// the velocity and note modes to tell which button to change
void get_position() {

  for(int i=0; i < BUTTONS; i++) {
    if ( (currtouched & _BV(i)) && !(lasttouched & _BV(i)) ) {
      mode_position = i;
      position_selected = true;
    }
  }

}

// resets mode display to off
void clear_modes() {

  // turn off all modes
  channel_mode = false;
  velocity_mode = false;
  pitch_mode = false;
  step_mode = false;
  tempo_mode = false;
  shuffle_mode = false;

  // reset mode position
  mode_position = 0;
  position_selected = false;

  // clear pixels

  flash(0,0,0);

}

// check if we have selected a command mode
bool mode_selected() {

  if(! command_mode)
    return false;

  if(channel_mode || velocity_mode || pitch_mode || step_mode || tempo_mode ||
shuffle_mode)
    return true;

  return false;

}
```

Appendix B: MPR121 Test Code

```
/*********************************************************
This is a library for the MPR121 12-channel Capacitive touch sensor

Designed specifically to work with the MPR121 Breakout in the Adafruit shop
  ----> https://www.adafruit.com/products/

These sensors use I2C communicate, at least 2 pins are required
to interface

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

Written by Limor Fried/Ladyada for Adafruit Industries.
BSD license, all text above must be included in any redistribution
*********************************************************/

#include <Wire.h>
#include "Adafruit_MPR121.h"

#ifndef _BV
#define _BV(bit) (1 << (bit))
#endif

// You can have up to 4 on one i2c bus but one is enough for testing!
Adafruit_MPR121 cap = Adafruit_MPR121();

// Keeps track of the last pins touched
// so we know when buttons are 'released'
uint16_t lasttouched = 0;
uint16_t currtouched = 0;

void setup() {
  Serial.begin(9600);

  while (!Serial) { // needed to keep leonardo/micro from starting too fast!
    delay(10);
  }

  Serial.println("Adafruit MPR121 Capacitive Touch sensor test");

  // Default address is 0x5A, if tied to 3.3V its 0x5B
  // If tied to SDA its 0x5C and if SCL then 0x5D
  if (!cap.begin(0x5A)) {
    Serial.println("MPR121 not found, check wiring?");
    while (1);
  }
  Serial.println("MPR121 found!");
}

void loop() {
  // Get the currently touched pads
  currtouched = cap.touched();

  for (uint8_t i=0; i<12; i++) {
    // it if *is* touched and *wasnt* touched before, alert!
    if ((currtouched & _BV(i)) && !(lasttouched & _BV(i)) ) {
      Serial.print(i); Serial.println(" touched");
    }
    // if it *was* touched and now *isnt*, alert!
    if (!(currtouched & _BV(i)) && (lasttouched & _BV(i)) ) {
      Serial.print(i); Serial.println(" released");
    }
  }
```

```
  // reset our state
  lasttouched = currtouched;

  // comment out this line for detailed data from the sensor!
  return;

  // debugging info, what
  Serial.print("\t\t\t\t\t\t\t\t\t\t\t\t 0x"); Serial.println(cap.touched(),
HEX);
  Serial.print("Filt: ");
  for (uint8_t i=0; i<12; i++) {
    Serial.print(cap.filteredData(i)); Serial.print("\t");
  }
  Serial.println();
  Serial.print("Base: ");
  for (uint8_t i=0; i<12; i++) {
    Serial.print(cap.baselineData(i)); Serial.print("\t");
  }
  Serial.println();

  // put a delay so it isn't overwhelming
  delay(100);
}
```

Appendix C: APDS9960 Test Code

```
/*************************************************************************
  This is a library for the APDS9960 digital proximity, ambient light, RGB, and
gesture sensor

  This sketch puts the sensor in gesture mode and decodes gestures.
  To use this, first put your hand close to the sensor to enable gesture mode.
  Then move your hand about 6" from the sensor in the up -> down, down -> up,
  left -> right, or right -> left direction.

  Designed specifically to work with the Adafruit APDS9960 breakout
  ----> http://www.adafruit.com/products/3595

  These sensors use I2C to communicate. The device's I2C address is 0x39

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

  Written by Dean Miller for Adafruit Industries.
  BSD license, all text above must be included in any redistribution
 *************************************************************************/

#include "Adafruit_APDS9960.h"
Adafruit_APDS9960 apds;

// the setup function runs once when you press reset or power the board
void setup() {
  Serial.begin(115200);

  if(!apds.begin()){
    Serial.println("failed to initialize device! Please check your wiring.");
  }
  else Serial.println("Device initialized!");

  //gesture mode will be entered once proximity mode senses something close
  apds.enableProximity(true);
  apds.enableGesture(true);
}

// the loop function runs over and over again forever
void loop() {
  //read a gesture from the device
    uint8_t gesture = apds.readGesture();
    if(gesture == APDS9960_DOWN) Serial.println("v");
    if(gesture == APDS9960_UP) Serial.println("^");
    if(gesture == APDS9960_LEFT) Serial.println("<");
    if(gesture == APDS9960_RIGHT) Serial.println(">");
}
```

Appendix D: LSM9DS1 Test Code

```
#include <Wire.h>
#include <SPI.h>
#include <Adafruit_LSM9DS1.h>
#include <Adafruit_Sensor.h>  // not used in this demo but required!

// i2c
Adafruit_LSM9DS1 lsm = Adafruit_LSM9DS1();

#define LSM9DS1_SCK A5
#define LSM9DS1_MISO 12
#define LSM9DS1_MOSI A4
#define LSM9DS1_XGCS 6
#define LSM9DS1_MCS 5
// You can also use software SPI
//Adafruit_LSM9DS1 lsm = Adafruit_LSM9DS1(LSM9DS1_SCK, LSM9DS1_MISO,
LSM9DS1_MOSI, LSM9DS1_XGCS, LSM9DS1_MCS);
// Or hardware SPI! In this case, only CS pins are passed in
//Adafruit_LSM9DS1 lsm = Adafruit_LSM9DS1(LSM9DS1_XGCS, LSM9DS1_MCS);


void setupSensor()
{
  // 1.) Set the accelerometer range
  lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_2G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_4G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_8G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_16G);

  // 2.) Set the magnetometer sensitivity
  lsm.setupMag(lsm.LSM9DS1_MAGGAIN_4GAUSS);
  //lsm.setupMag(lsm.LSM9DS1_MAGGAIN_8GAUSS);
  //lsm.setupMag(lsm.LSM9DS1_MAGGAIN_12GAUSS);
  //lsm.setupMag(lsm.LSM9DS1_MAGGAIN_16GAUSS);

  // 3.) Setup the gyroscope
  lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_245DPS);
  //lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_500DPS);
  //lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_2000DPS);
}


void setup()
{
  Serial.begin(115200);

  while (!Serial) {
    delay(1); // will pause Zero, Leonardo, etc until serial console opens
  }

  Serial.println("LSM9DS1 data read demo");

  // Try to initialise and warn if we couldn't detect the chip
  if (!lsm.begin())
  {
    Serial.println("Oops ... unable to initialize the LSM9DS1. Check your wiring!
");
    while (1);
  }
  Serial.println("Found LSM9DS1 9DOF");

  // helper to just set the default scaling we want, see above!
  setupSensor();
}
```

```
void loop()
{
  lsm.read();  /* ask it to read in the data */

  /* Get a new sensor event */
  sensors_event_t a, m, g, temp;

  lsm.getEvent(&a, &m, &g, &temp);

  Serial.print("Accel X: "); Serial.print(a.acceleration.x); Serial.print("
m/s^2");
  Serial.print("\tY: "); Serial.print(a.acceleration.y);     Serial.print(" m/s^2
");
  Serial.print("\tZ: "); Serial.print(a.acceleration.z);     Serial.println("
m/s^2 ");

  Serial.print("Mag X: "); Serial.print(m.magnetic.x);   Serial.print(" gauss");
  Serial.print("\tY: "); Serial.print(m.magnetic.y);     Serial.print(" gauss");
  Serial.print("\tZ: "); Serial.print(m.magnetic.z);     Serial.println(" gauss");

  Serial.print("Gyro X: "); Serial.print(g.gyro.x);   Serial.print(" dps");
  Serial.print("\tY: "); Serial.print(g.gyro.y);      Serial.print(" dps");
  Serial.print("\tZ: "); Serial.print(g.gyro.z);      Serial.println(" dps");

  Serial.println();
  delay(200);
}
```