Microcontrollers Final Project

Garage Band

Carissa Lee, Josh Dykas, Ella Stephani

UVM Department of Electrical Engineering

EE3185: Microcontrollers

Instructor: Dr. Tony Barsic

December 12, 2025

Contents

Abstract

This project presents the design, construction, and testing of a fully functional, microcontroller-based instrument controller system for the rhythm game *Clone Hero*. Two independent interfaces—a five-button guitar and a five-pad drum set—were developed using an Arduino Pro Micro with an ATmega32U4 microcontroller, enabling native USB HID functionality for direct keyboard emulation. The guitar controller uses discrete push buttons for fret inputs and a joystick-based strum mechanism, while the drum system employs I2C pressure sensors mounted beneath balloons to detect transient pressure spikes corresponding to drum strikes. Custom firmware was written for each device to map hardware events to Clone Hero keybindings, supported by real-time hit detection algorithms, multiplexer control for five identical I2C sensors, and mechanical enclosures designed through 3D printing and hand-fabricated materials. Initial subsystem tests validated the reliability of keyboard mapping and pressure-based strike detection before integration into the full mechanical assemblies. The final system successfully interfaces with the game environment, demonstrating responsive, playable inputs that emulate the behavior of commercial controllers while offering flexibility for customization and expansion.

Idea and Background

This project aims to create mimetic guitar-hero game controllers using an Arduino pro micro with an ATMega 32U4 chip for key binding. This system will consist of two interfaces, the guitar and a drum set. The five-key guitar will correspond each button press to a note played while a five pad drum set will utilize I2C connections and balloons to detect pressure changes, insignia of a drum strike.

Clone Hero

Clone Hero is a game used on Windows, Linux, or Apple to mimic the traditional, classic Guitar Hero series. The game allows the user to customize the keys used to control the notes within the game. Because Clone Hero accepts keyboard inputs, it also accepts USB HID devices, making it a prime contender for a microelectronic based project.

Guitar Interface

The guitar module is simple and consists of five buttons. Each button is the same color and will serve the same purpose as the keys in the original Guitar Hero game.

Drum Interface

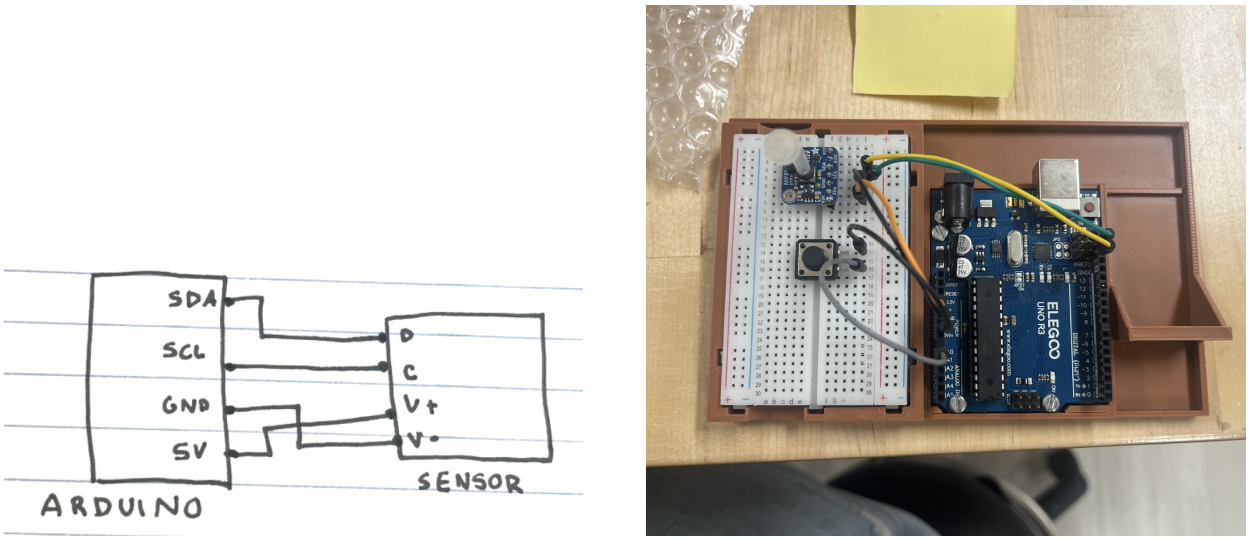
The drum module will utilize five I2C analog pressure sensors, each mounted beneath a balloon. When the balloons surface is tapped, the internal pressure will change, monitored by the sensor. When this value reaches above a threshold, it will determine that a drum strike has been made. This will be mapped onto the Clone Hero game in the drum section.

Initial Testing

To begin the creation process, simple hardware and software compatibility testing ensued for proof of concept.

Drum Pressure Testing

As aforementioned, the drum setup will be created using an I2C analog pressure sensor. For initial testing purposes, only one ”drum” setup will be created (rather than five in the final design). The pressure sensor’s tube was mechanically attached to the bottom of an inflated balloon using a zip tie while the sensor itself had four of its output pins connected to the pro microcontroller: two connected to SRA/SCL on the Arduino, one to power rail, and one to the ground rail. This circuit schematic (a) and experimental setup (b) can be found below in Figure 1.



(a) Circuit schematic

(b) Experimental setup

Figure 1: Single Pressure Sensor Schematic and Experimental Setup

To initiate the software, an initial connection to the pressure sensor was estabished and a pinout was created in correlation to typical I2C devices. Alongside this, a button was also attached to the circuit which would later serve as a PSI zeroing function.

```
// include libraries
#include <Wire.h>
#include "Adafruit_MPRLS.h"

// define pressure sensor pins
#define RESET_PIN -1
#define EOC_PIN   -1
Adafruit_MPRLS mpr = Adafruit_MPRLS(RESET_PIN, EOC_PIN);

// define button pins
```

```
#define BUTTON_PIN A0 // input pullup
```

Following this, hit detection variables were defined. These variables need to establish a delta value that will act as a threshold of pressure change emulating a strike, a release threshold that triggers a rearming of the cycle, and a small cool down between hits.

```
// hit detection variables
#define HIT_DELTA_PSI 0.02f // how much delta psi impulse to trigger
#define RELEASE_THRESHOLD 0.01f // how far below the trigger to re-arm
#define COOLDOWN_MS 150 // minimal time between hits
```

The raw data collected from the mpr pressure sensor is in hPa, to convert this into PSI, this value was divided by a constant represented below in Equation 1.

$$PSI = \frac{hPa}{68.95} \tag{1}$$

This was then implemented into the script to allow for pressure readings in a more identifiable unit format.

```
static inline float readPsi() {
  // convert hPa to PSI
  float hPa = mpr.readPressure();
  return hPa / 68.947572932f;
}
```

After connecting to the pressure sensor, the code continuously searches for a button press. If detected, the code set the current pressure value, at the same time as button press, to be ”zero”.

```
  // button devounce and detection to zero
  bool btn = digitalRead(BUTTON_PIN);
  if (prevBtnState == HIGH && btn == LOW) {
    zeroPsi = readPsi();
    zeroed = true;
    hitLatched = false;
    lastHitMs = 0;
    Serial.print("Zeroed at ");
    Serial.print(zeroPsi, 3);
    Serial.println(" psi");
```

Using this, hit defines were declared.

```
  // hit defines
  float psi = readPsi();
  float trigger = zeroPsi + HIT_DELTA_PSI;
  float release = trigger - RELEASE_THRESHOLD;
```

If a hit was detected, a line was printed to the serial monitor declaring a hit was detected and the readable psi.

```
if (!hitLatched) {
  // detect a hit
  if (psi >= trigger && (now - lastHitMs) >= COOLDOWN_MS) {
    Serial.print("Hit detected at ");
    Serial.print(psi, 3);
    Serial.println(" psi");
    hitLatched = true;
    lastHitMs = now;
  }
```

This effectively created a code that detects a "drum strike" through pressure sensing and reports this data to the serial monitor. The addition of a zeroing/homing button allows for variability in use in different pressure environments. Figure 2 below shows the output monitor, proving efficacy.

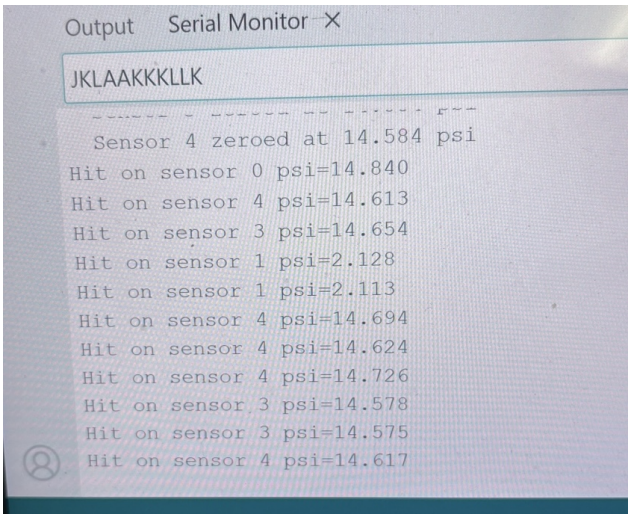


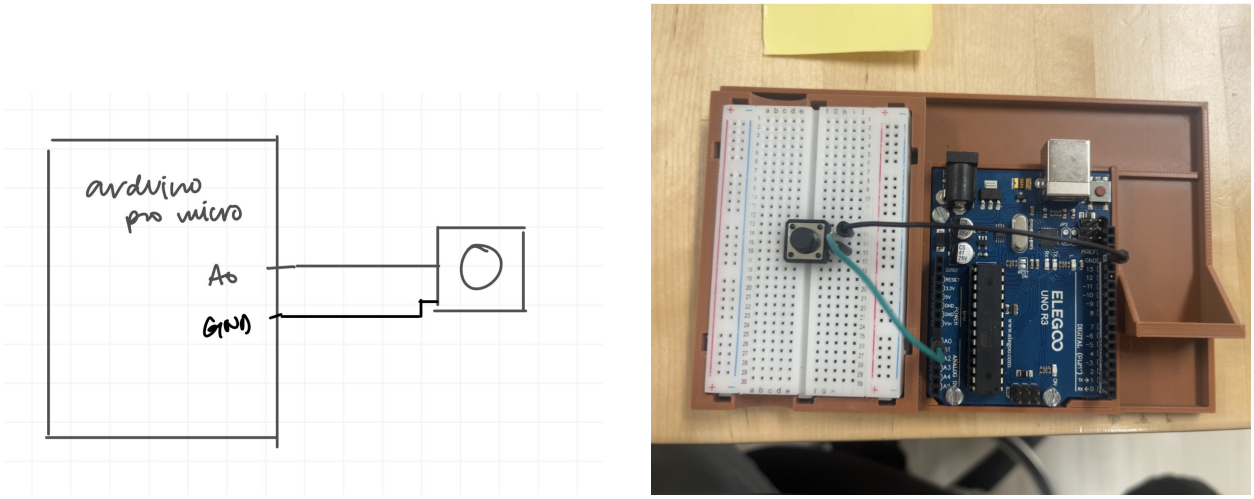
Figure 2: Serial Monitor Output from Drum Testing

Also from this test, the baseline pressure within the balloon was determined to be XYZ. Upon moderate pressure, this value was determined to fluctuate +/- 0.02 PSI, hence why the delta value was established as 0.02PSI. This threshold was set as it corresponds to the pressure change associated with a not too hard, yet intentional hit of the drum. The full code for this section can be found in Appendix A.

Keyboard Mapping (Guitar)

Practicing with key binding is imperative to the success of both systems, the drums and the guitar, as the clone hero game can be controlled through an external device through this method. As aforementioned, the software interface accepts keyboard inputs and hence USB HID devices. The Arduino Pro Micro, which uses the ATmega32U4 microcontroller, supports native USB functionality that allows it to act as a keyboard for binding, ideal for this project. To begin testing, a simple connection between a button press and keyboard typing "A" was established. The library "Keyboard.h" was used to easily establish a logic connection between button mapping and keyboard

letters. A button was connected to pin A0 and ground and a keyboard connection was established. Figure 3 below shows the experimental circuit created (right) and circuit schematic (left). Below that shows the corresponding code for initialization.



Circuit schematic Experimental circuit setup

Figure 3: Circuit schematic (left) and experimental circuit created (right) for Keybinding Tests.

```
#include <Keyboard.h>

#define BUTTON_PIN 2   // connect button to pin 2 and GND

void setup() {
  pinMode(BUTTON_PIN, INPUT_PULLUP); // button active LOW
  Keyboard.begin();
}

void loop() {
  int buttonState = digitalRead(BUTTON_PIN);

  // button pressed when pulled LOW
  if (buttonState == LOW) {
    Keyboard.print("A");
    delay(200); // simple debounce + prevents spamming
  }
}
```

Following this, if a button press was detected (when pulled LOW due to prior input_pullup logic), the keyboard prints the letter A.

```
    void loop() {
  int buttonState = digitalRead(BUTTON_PIN);
```

```
  // button pressed when pulled LOW
  if (buttonState == LOW) {
    Keyboard.print("A");
    delay(200); // simple debounce + prevents spamming
  }
}
```

From this, a blank text file was used to test if a button press connected to the Arduino would successfully print the letter A. The output is below in Figure 4.

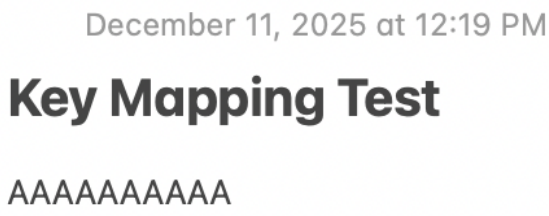
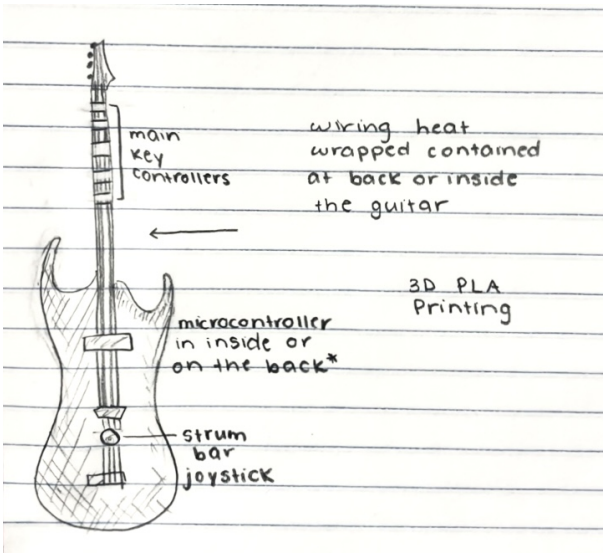


Figure 4: Text File Ouput displaying Button Press printing Letter A

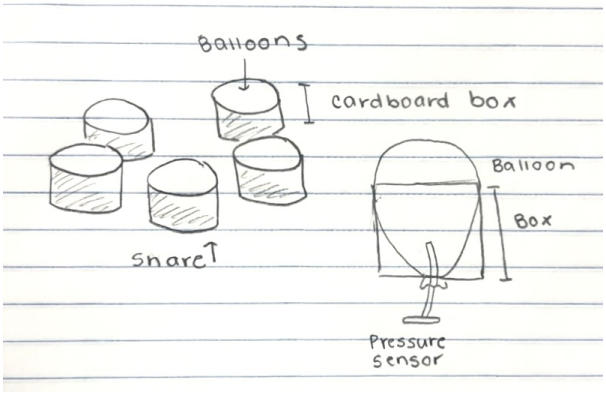
This effectively used a simple hardware output to map keybinds. The full commented code for this section can be found in Appendix B.

Mechanical Creation

Alongside creating the electrical components that will be used to control the clone hero game, this project also utilizes mechanical drafting to create a house for the electrical system to physically appear like the different instruments. Figure 5 below shows rough draft sketches of what the final outcome should look like for the drums (right) and guitar (left).



Guitar sketch

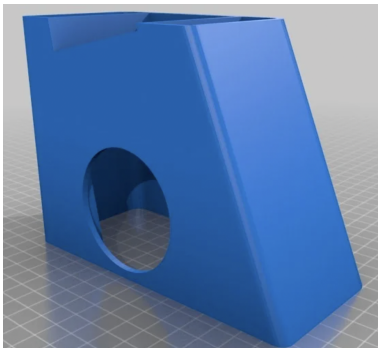


Drum sketch

Figure 5: Rough draft sketches of the expected final outcome for the guitar (left) and drums (right).

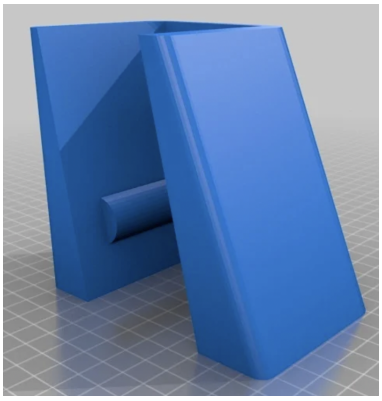
Guitar

For the guitar portion, the website ”thingiverse” was searched to find a guitar that was already 3D rendered [1]. Figure 6 below shows two of the five parts to be later assembled.



Guitar Centerpiece

Guitar Neck



Guitar Bottom

Figure 6: 3D Model of Guitar used for 3D Printing.

Utilizing on-campus resources, the fablab was contacted and used to create the 3D model above out of polylactic acid (PLA). The printed, unassembled pieces are shown below in Figure 7

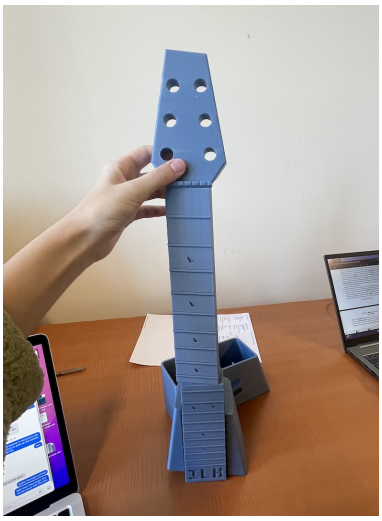
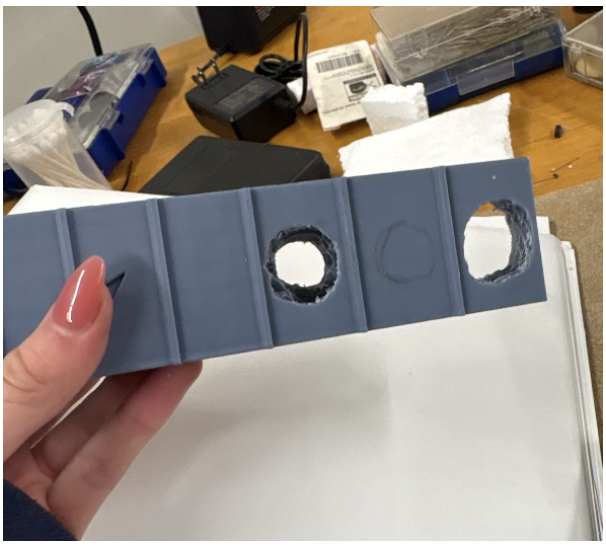
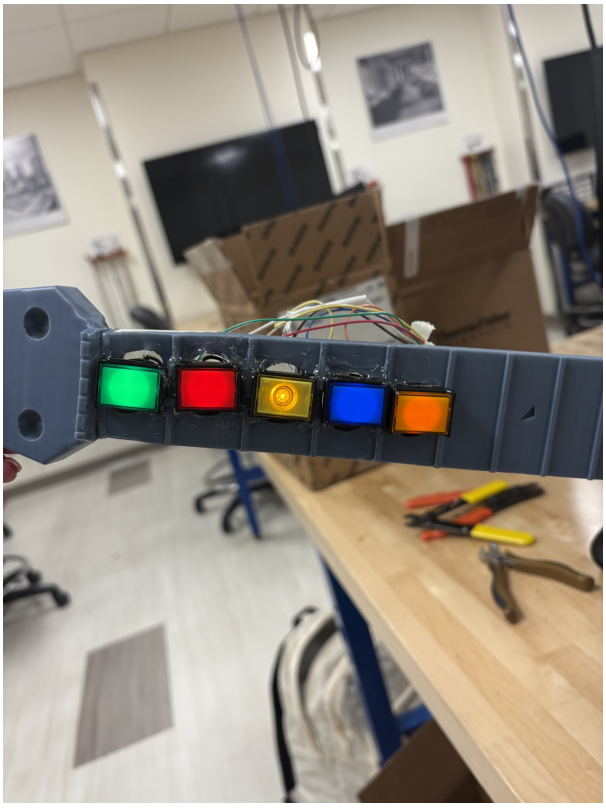


Figure 7: Unassembled 3D printed Guitar Pieces

As previously shown from the design model, five different buttons, serving as the guitar notes, were to be attached to the neck of the guitar. To do this, holes were drilled through the PLA using a dremmel, and the buttons were snuggly glued into place. Figure 8 below shows the holes drilled (left) and the implant of the buttons (right).



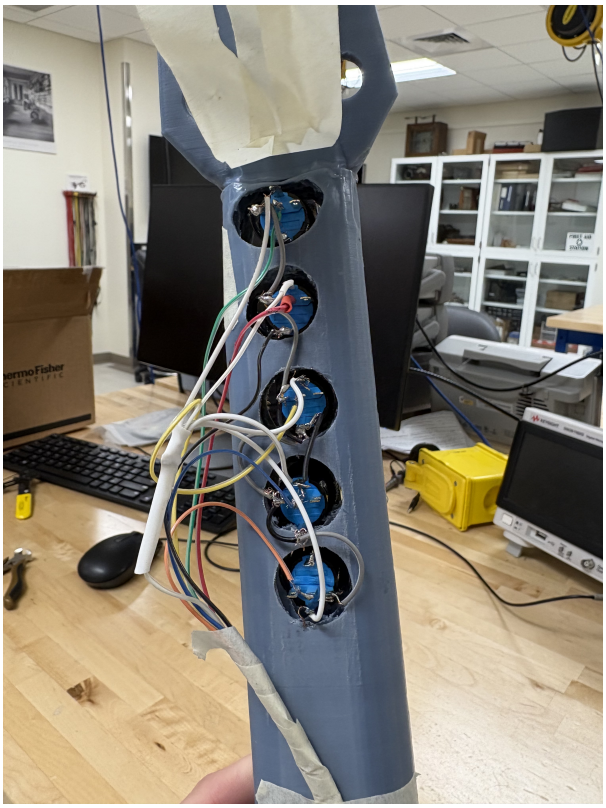
Holes drilled



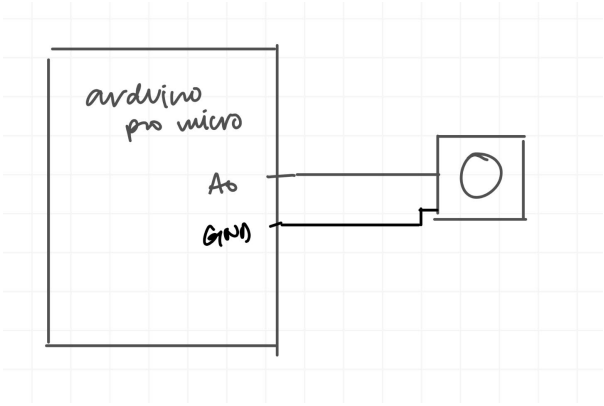
Buttons implanted

Figure 8: Holes drilled (left) and the implant of the buttons (right).

Once the buttons were installed, long wires were attached from each metal prong in the wire in order to connect the buttons to the microcontroller, housed on the back of the base of the guitar. These extension wires were soldered to the buttons while heat shrink wraps were added in order to ensure mechanical stability of the solder joints, and to compartmentalize each cord. Figure 9 shows the attachment of the breadboard, master, and slave devices to the back of the housing unit (left) and the solder joints and extensions cords connected to the buttons on the guitar neck (right).



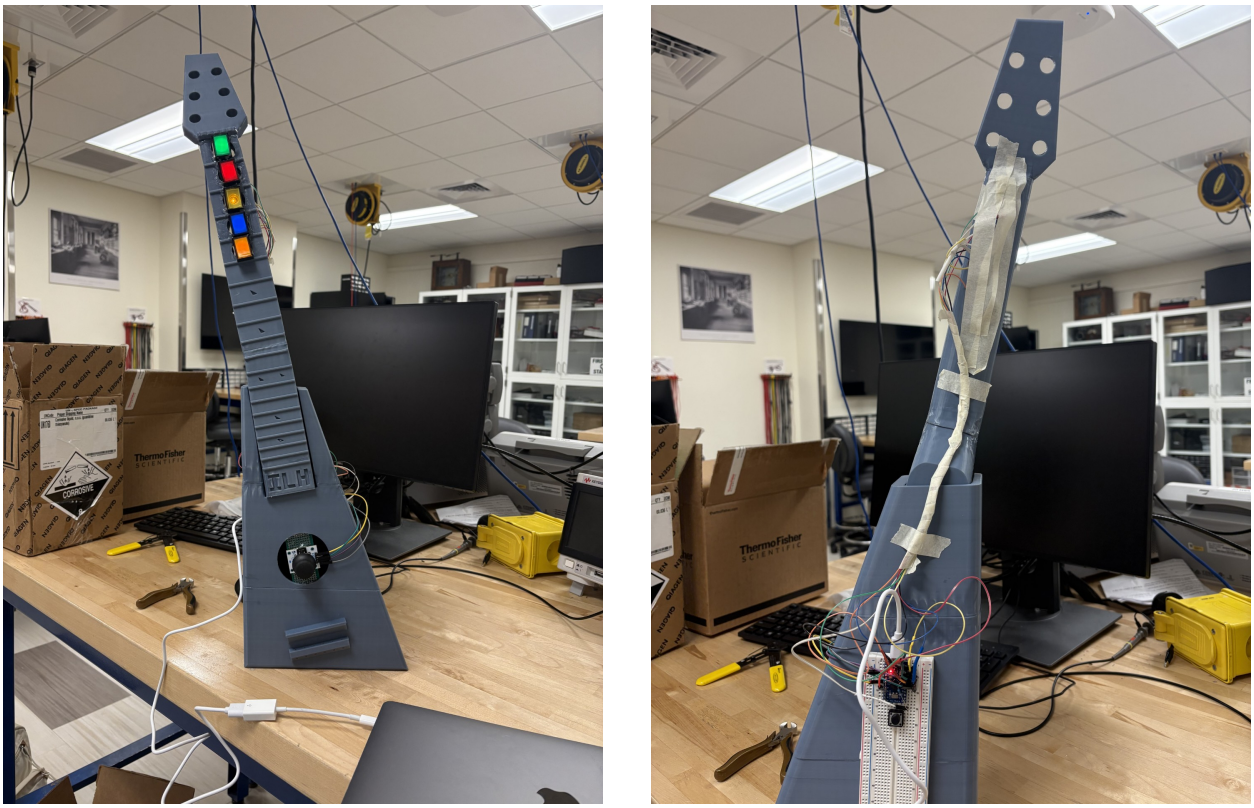
Attachments on back of housing unit



Solder joints and extension cords

Figure 9: Electrical Attachments onto the Mechanical Housing Unit and Soldering Joints.

After having attached all components into the mechanical device, the device was finally glued together and fully assembled, as shown in Figure **??**.



Front view Back view

Figure 10: Front (left) and back (right) views of the final guitar.

Drums

After completing the guitar, the drums were much easier to assemble a mechanical housing unit. Each drum was isolated within its own cardboard box, fitting tightly within its housing unit. The top of the balloon protrudes out of the top of the box, while a hole was cut in the bottom of the box to allow the electrical components to connect, as shown in Figure 11.

Top view with colored drums

Bottom of boxes with wiring holes

Figure 11: Top view of the drum set showing the different colored drums (left), and the bottom of the boxes with holes cut for electrical wiring (right).

The boxes were then connected using cardboard so the unit was able to be more easily transported.

Coding and Keyboard Binding

Following mechanical assembly, two software's were compiled to create the electrical control systems for both the guitar and drum sets. As aforementioned, to create game controls compatible with the Clone Hero interface, key binds needed to be mapped onto user hardware input. To do so, the guitar uses buttons while the drumset uses hit detection. To clarify the key mapping between the game and each user input, Figure 12 below shows the hardware input mapped onto the key presses and in-game color correlations.

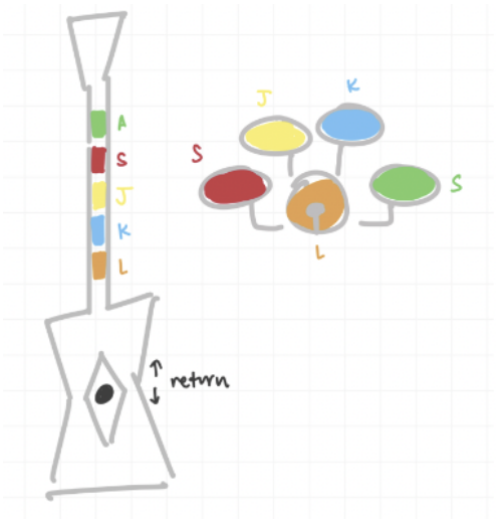


Figure 12: Keybinds Mapped onto Physical Inputs

Using this logic, the two systems were created.

Guitar

This section aims to create an interface that allows a custom guitar prototype to function as a controller within the Clone Hero game environment. To accomplish this, five push buttons mounted along the neck served as the fret buttons, while the strum input was handled by a two-axis joystick module configured to detect up and down strums. Each input was mapped to a specific keyboard key so that pressing a fret or strumming would generate the corresponding in-game actions.

The fret buttons were wired directly to the microcontroller using internal pull-up resistors. Each button pulls its respective pin to ground when pressed. The strum bar—implemented using the vertical axis of a joystick—outputs an analog voltage that varies with displacement. Thresholds were selected so that moving the joystick above or below center triggered a strum-up or strum-down event. Figure 13 below shows the wiring diagram (left) and the experimental setup (right) for the guitar prototype.

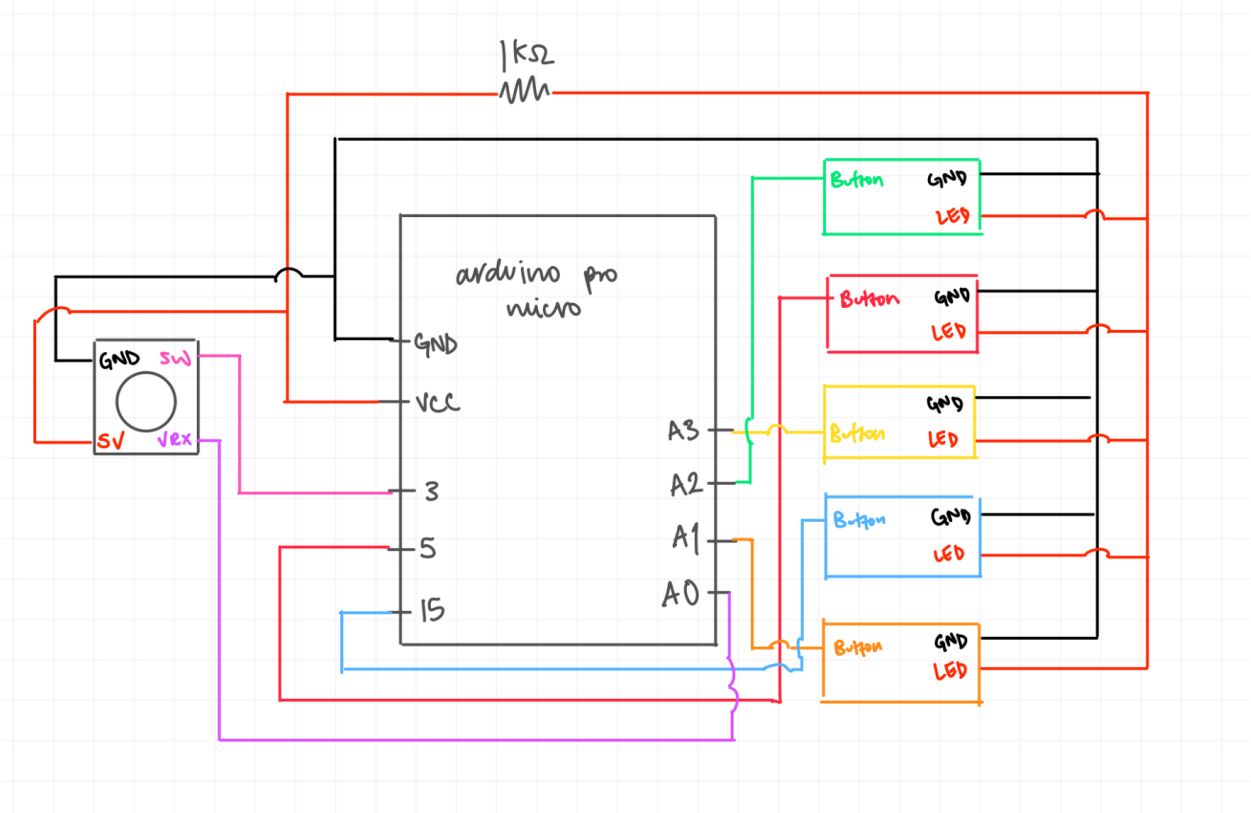


Figure 13: Guitar Electrical Schematic

To organize the wiring and logic, each fret button was assigned a digital input pin and mapped to a keyboard character used by Clone Hero. Table 1 summarizes the pin-level assignments and corresponding keyboard outputs.

```
switch (pin) {
    case green:
        if (isPressed) Keyboard.press('A');
        else Keyboard.release('A');
        break;
```

Table 1: Guitar Key-Mapping Logic

Guitar Key	Keyboard Output
Yellow	J
Green	A
Red	S
Orange	L
Blue	K
Joystick Down	Down Arrow
Joystick Up	Up Arrow
Joystick Button	Return

The joystick's vertical axis was connected to analog input **A0**, and two strum thresholds were defined symmetrically around the center voltage. Crossing the upper threshold produced a "strum-up" action, and crossing the lower threshold produced a "strum-down" action.

```
// Strum bar tracking
const int STRUM_CENTER = 512;
// analog midpoint (0 1023 )
const int STRUM_TOLERANCE = 150;
// how far from center counts as up/down
```

A function is created the reads the analog value of the strum bar, and within a certain tolerance range assigns the movement of the joystick to either the up or down arrow keys.

```
void updateStrum() {
  int val = analogRead(strumBar);

  bool shouldUp = (val > STRUM_CENTER + STRUM_TOLERANCE);
  bool shouldDown = (val < STRUM_CENTER - STRUM_TOLERANCE);

  // Up arrow
  if (shouldUp && !strumUpHeld) {
    Keyboard.press(KEY_UP_ARROW);
    strumUpHeld = true;
  } else if (!shouldUp && strumUpHeld) {
    Keyboard.release(KEY_UP_ARROW);
    strumUpHeld = false;
  }

  // Down arrow
  if (shouldDown && !strumDownHeld) {
    Keyboard.press(KEY_DOWN_ARROW);
    strumDownHeld = true;
  } else if (!shouldDown && strumDownHeld) {
```

```
    Keyboard.release(KEY_DOWN_ARROW);
    strumDownHeld = false;
  }
}
```

Similarly to the fret buttons, it is important to detect if the strum bar is held in either position, whereas another set of conditionals are implemented to continue "pressing" the arrow keys.

The loop combines the fret button and strum bar functions by first reading each of the button states, including the joystick switch (which is used to navigate throughout the Clone Hero game).

If the state of any button changes (from high to low for the first button press), the handleButtonPress( ) function is called and the previous button state is updated.

```
// Fret buttons: press while held, release when let go
if (greenState != lastGreen) {
    handleButtonPress(green, (greenState == LOW));
    lastGreen = greenState;
}
```

A conditional for the joystick switch is impltemented to detect if it is pressed, which is key-binded to the "return" key. The state of the switch is then updated outside the conditional, and the updateStrum( ) function is called, along with a small debounce delay.

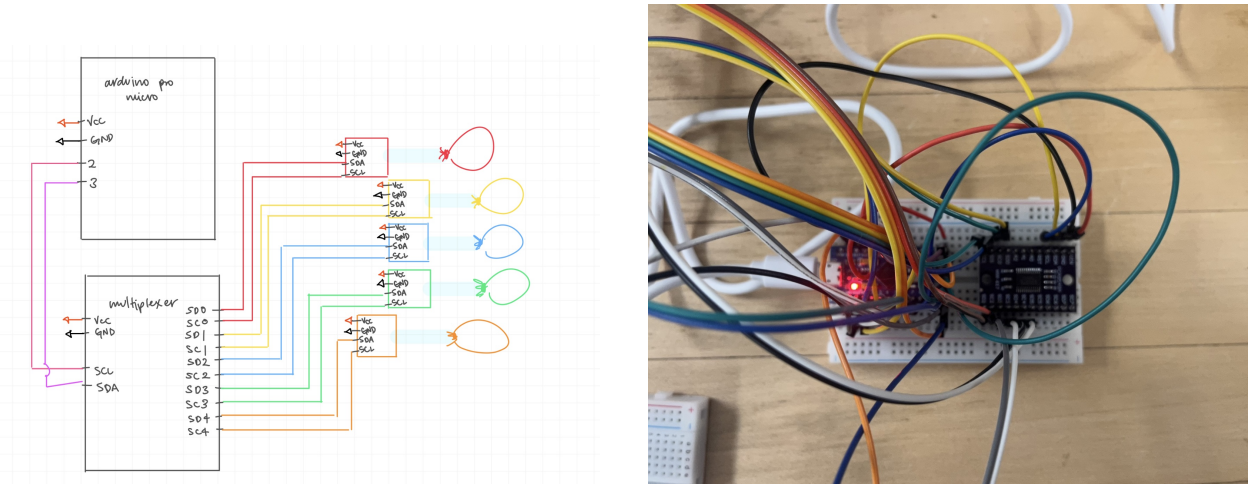
```
// Joystick switch: type Return on press (edge)
  if (lastJoy == HIGH && joyState == LOW) {
    Keyboard.write(KEY_RETURN);
  }
  lastJoy = joyState;

  // Strum bar up/down handling
  updateStrum();

  delay(5);  // small debounce / smoothing
}
```

Drums

This section aims to create a software to connect with the Clone Hero interface to use the mechanical drums prototype as a game controller. To do so, five balloons attached to five pressure sensors were controlled using a multiplexer so when each drum was hit, it would type a different letter on the keyboard. This key binding allowed the drums to work as a makeshift controller for the game interface.

To begin, each balloon was attached to an I2C pressure sensor so that the sensor was reading the pressure from within the balloon. Because each of these sensors was using the same address, as they are the same make and model, an I2C multiplexer was utilized to control the five sensors. The mux used was the TCA9548A. This device connects its power, ground, and I2C serial data and clock to the master device and provides eight controllable outputs. Each pressure sensor had

its I2C pins attached to the multiplexer while its power and ground went ot the shared pins on the master device. The pinout (left) for this and experimental circuit (right) is shown below in Figure 14.



Pinout Experimental circuit

Figure 14: Pinout (left) and corresponding experimental circuit (right) for the drum system.

Table **??** below summarizes this connection scheme. Following the wiring, the testing code was reformatted to account for the multiplexing of the five different pressure sensors, while keeping the basic logic the same. In addition to the test code, the I2C multiplexer settings were established, alongside the amount of pressure sensors.

```
// I2C multiplexer settings
#define TCA_ADDR 0x70

void tcaSelect(uint8_t channel) {
  if (channel > 7) return;
  Wire.beginTransmission(TCA_ADDR);
  Wire.write(1 << channel);
  Wire.endTransmission();
}

// pressure sensor settings
#define NUM_SENSORS 5
```

Keyboard mapping was established based on the channel in which the pressure sensor was wired, the color of the balloon, and the key it is supposed to print. Table 2 below defines this logic.

Table 2: Drum Key-Mapping Logic

Channel	Drum Color	Keyboard Output
0	Yellow	J
1	Green	A
2	Red	S
3	Orange	L
4	Blue	K

Using this, a constant character array was created for later indexing, corresponding key letter to the index of the pressure sensor.

```
const char sensorKeys[NUM_SENSORS] = {'J', 'A', 'S', 'L', 'K'};
```

To begin detecting, the pressure sensors must first be zeroed in order to create a baseline PSI to detect delta hits. To do this, if a button press was detected, a loop files through all the connected sensors to the mux and zeros the current PSI value.

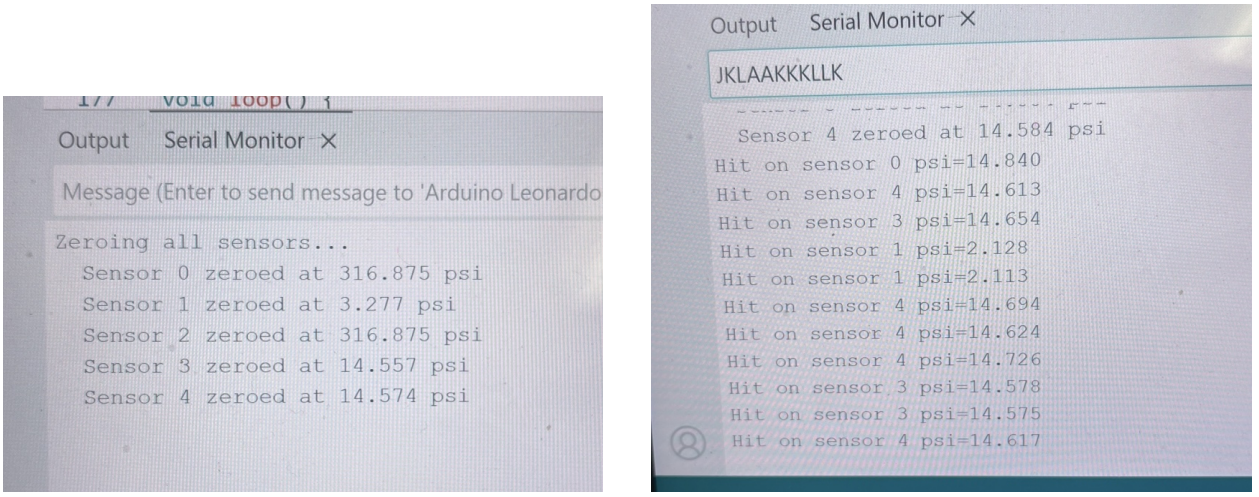
```
// detect button press to zero sensors
void loop() {
  // create a constant boolean to define button state
  bool btn = digitalRead(BUTTON_PIN);
  // if button is pressed, zero sensors
  if (prevBtnState == HIGH && btn == LOW) {
    // print to serial monitor
    Serial.println("Zeroing all Pressure Sensors!");
    // loops through all sensors and zeros
    for (int i = 0; i < NUM_SENSORS; i++) {
      tcaSelect(sensorChannels[i]);
      delay(5);
      zeroPsi[i] = readPsi();
      zeroed[i] = true;
      hitLatched[i] = false;
      lastHitMs[i] = 0;
```

These values are then printed to the serial monitor for verification. Following this, all five sensors are iterated through and the code detects if there is a hit or not based on hit detection logic used within the testing portion of this report. If a hit is detected, these values are printed to the serial monitor, and the key array is indexed into and printed.

```
// loops through each sensor
  for (int i = 0; i < NUM_SENSORS; i++) {
    if (!zeroed[i]) continue;
    // selects the channel currently being indexed
    tcaSelect(sensorChannels[i]);
    delay(2);
```

```
// reads current pressure and establishes hit detection logic
float psi = readPsi();
float trigger = zeroPsi[i] + HIT_DELTA_PSI;
float release = trigger - RELEASE_THRESHOLD;

// if hit was detected, prints vlaues to the serial monitor
// and prints letter
if (!hitLatched[i]) {
  if (psi >= trigger && (now - lastHitMs[i]) >= COOLDOWN_MS) {
    Serial.print("Hit on sensor ");
    Serial.print(i);
    Serial.print(" psi=");
    Serial.println(psi, 3);

    Keyboard.write(sensorKeys[i]);
```

Figure 15 below shows the final serial monitor outputs for both the zeroing function (left) and the hit detection (right).



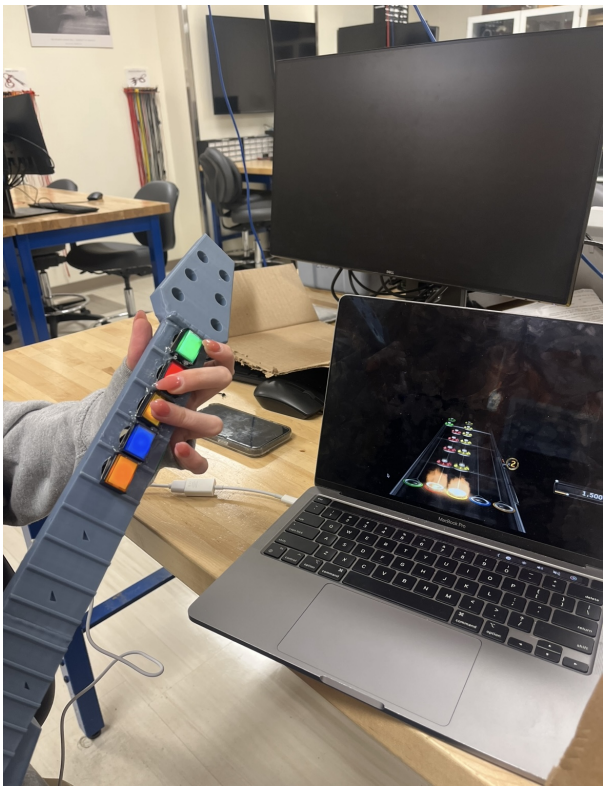
Zeroing function output

Hit detection output

Figure 15: Final serial monitor outputs for both the zeroing function (left) and the hit detection (right).

This effectively created the key-bound controller for the clone hero game. The entire code for this section can be found in Appendix D.

Clone Hero



(a) Guitar practice

(b) Drum practice

Figure 16: Practice setup comparison between guitar and drum performance

Discussion

The results of this project demonstrate that low-cost microcontrollers and accessible fabrication techniques can be used to create functional, customized rhythm-game controllers. Both subsystems—the guitar and the drums—successfully translated physical user interactions into keyboard inputs interpretable by the Clone Hero interface. Achieving this required addressing several design challenges involving electrical integration, sensor reliability, and mechanical housing.

A core success of the project was the use of the ATmega32U4's native USB HID capability. Unlike typical Arduino boards that require additional drivers or serial-to-USB bridges, the Pro Micro allowed the controllers to behave directly as keyboards, greatly simplifying the communication layer with Clone Hero. This enabled the keybinding portions of the design to focus on debouncing, state tracking, and mapping logic rather than protocol translation. The robustness of the keyboard emulation was confirmed early during single-button and text-file testing, later scaling cleanly to five fret inputs and joystick-based strumming without noticeable latency.

The drum subsystem introduced additional complexity due to the need for analog sensing and repeatable hit detection. Using balloons as flexible diaphragms proved to be a lightweight, low-cost, and surprisingly effective method for generating measurable pressure impulses. The incorporation

of the TCA9548A I2C multiplexer was essential, as it enabled five identical sensors—each with the same I2C address—to coexist on the same bus. The hit-detection algorithm, based on PSI deltas, provided reliable performance once appropriate thresholds were identified through experimentation. While the system is sensitive to balloon tension and ambient pressure differences, the inclusion of a zeroing button allowed the user to recalibrate as needed. This highlights the advantage of software-defined thresholds in systems that rely on inexpensive or variable mechanical elements.

Mechanically, the project combined 3D-printed components and improvised structures such as cardboard enclosures for the drums. Although the PLA guitar body required post-processing—drilling, sanding, gluing—it provided a sturdy, aesthetically recognizable form that supported the electronics well. The drum boxes, while simple, allowed fast iteration and easy access for wiring changes during development. Future versions could benefit from more durable materials, improved cable routing, and integrated mounting points for sensors and wiring.

A limitation of the current system is that it does not incorporate dynamic input levels; for example, drum strikes are binary rather than velocity-sensitive. Similarly, the joystick-based strum bar, while functional, could be replaced with a spring-loaded lever mechanism to better emulate the feel of commercial controllers. Additionally, long-term durability of balloon-based sensors remains uncertain, and a redesign using rubber membranes or 3D-printed plates could improve consistency. Despite these considerations, the project successfully met its primary goal: creating a responsive, playable, and fully functional set of Clone Hero controllers.

Overall, this project highlights how microcontrollers, basic sensors, and rapid prototyping tools can be combined into a cohesive human–machine interface. The techniques developed—USB HID keyboard emulation, multiplexer-based sensor expansion, and custom mechanical housing—provide a foundation for future enhancements such as wireless communication, velocity-sensitive drumming, configurable keymaps, or expanded instrument types. The final system showcases the creativity and engineering versatility possible within a microcontroller-based final project.

References

[1] Solstie, “Playable Guitar – Printable Without Supports,” Thingiverse, Thing 486731. [Online]. Available: https://www.thingiverse.com/thing:486731. [Accessed: Dec. 11, 2025]. :contentReference[oaicite:0]index=0

[2] Chat GPT wrote discussion and abstract based on the rest of this report.

Appendix

A.)

```
// Carissa Lee
// CMPE3815 Final Project | Testing Drums

// include libraries
#include <Wire.h>
```

```
#include "Adafruit_MPRLS.h"

// define pressure sensor pins
#define RESET_PIN -1
#define EOC_PIN   -1
Adafruit_MPRLS mpr = Adafruit_MPRLS(RESET_PIN, EOC_PIN);

// define button pins
#define BUTTON_PIN A0 // input pullup

// hit detection variables
#define HIT_DELTA_PSI 0.02f // how much delta psi impulse to
// trigger
#define RELEASE_THRESHOLD 0.01f // how far below the trigger
// to re-arm
#define COOLDOWN_MS 150 // minimal time between hits

bool zeroed = false;
bool prevBtnState = HIGH;
bool hitLatched = false; // true while above threshold; re-arms
// after falling below
float  zeroPsi = 0.0f;
unsigned long lastHitMs = 0;

static inline float readPsi() {
  // convert hPa to PSI
  float hPa = mpr.readPressure();
  return hPa / 68.947572932f;
}

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

  pinMode(BUTTON_PIN, INPUT_PULLUP); // LOW button

  // Initial I2C scan
  Serial.println("I2C scan:");
  for (uint8_t addr = 1; addr < 127; addr++) {
    Wire.beginTransmission(addr);
    if (Wire.endTransmission() == 0) {
      Serial.print("  Device at 0x");
      if (addr < 16) Serial.print('0');
      Serial.println(addr, HEX);
    }
```

```
  }

  if (!mpr.begin()) {
    Serial.println("MPRLS not found. Check wiring.");
    while (1) delay(10);
  }
  Serial.println("Found MPRLS sensor");
}

void loop() {
  // button devounce and detection to zero
  bool btn = digitalRead(BUTTON_PIN);
  if (prevBtnState == HIGH && btn == LOW) {
    zeroPsi = readPsi();
    zeroed = true;
    hitLatched = false;
    lastHitMs = 0;
    Serial.print("Zeroed at ");
    Serial.print(zeroPsi, 3);
    Serial.println(" psi");
  }
  prevBtnState = btn;

  if (!zeroed) {
    delay(10);
    return; // nothing to do until zeroed
  }

  // hit defines
  float psi = readPsi();
  float trigger = zeroPsi + HIT_DELTA_PSI;
  float release = trigger - RELEASE_THRESHOLD;

  unsigned long now = millis();

  if (!hitLatched) {
    // detect a hit
    if (psi >= trigger && (now - lastHitMs) >= COOLDOWN_MS) {
      Serial.print("Hit detected at ");
      Serial.print(psi, 3);
      Serial.println(" psi");
      hitLatched = true;
      lastHitMs = now;
    }
  } else {
```

```
    // wait until psi returns to normlized
    if (psi <= release) {
      hitLatched = false;
    }
  }

  delay(5); // small delay for smoothing
}
```

B.)

```
#include <Keyboard.h>

#define BUTTON_PIN 2   // connect button to pin 2 and GND

void setup() {
  pinMode(BUTTON_PIN, INPUT_PULLUP); // button active LOW
  Keyboard.begin();
}

void loop() {
  int buttonState = digitalRead(BUTTON_PIN);
  // button pressed when pulled LOW
  if (buttonState == LOW) {
    Keyboard.print("A");
    delay(200); // simple debounce + prevents spamming
  }
}


void loop() {
  int buttonState = digitalRead(BUTTON_PIN);

  // if a button press is detected
  if (buttonState == LOW) {
    Keyboard.print("A");
    delay(50); // easy debounce
  }
}
```

C.)

D.)

```
#include <Wire.h>
#include "Adafruit_MPRLS.h"
#include <Keyboard.h>

// I2C multiplexer settings
#define TCA_ADDR 0x70

void tcaSelect(uint8_t channel) {
  if (channel > 7) return;
  Wire.beginTransmission(TCA_ADDR);
  Wire.write(1 << channel);
  Wire.endTransmission();
}

// pressure sensor settings
#define NUM_SENSORS 5

const uint8_t sensorChannels[NUM_SENSORS] = {0, 1, 2, 3, 4};

#define RESET_PIN -1
#define EOC_PIN   -1
Adafruit_MPRLS mpr = Adafruit_MPRLS(RESET_PIN, EOC_PIN);

// button and hit detection constants
#define BUTTON_PIN A0
#define HIT_DELTA_PSI      0.02f
#define RELEASE_THRESHOLD  0.01f
#define COOLDOWN_MS        150

bool   zeroed[NUM_SENSORS]     = {false};
bool   hitLatched[NUM_SENSORS] = {false};
float  zeroPsi[NUM_SENSORS]    = {0.0f};
unsigned long lastHitMs[NUM_SENSORS] = {0};
bool prevBtnState = HIGH;

// keyboard mapping conventions
const char sensorKeys[NUM_SENSORS] = {'J', 'A', 'S', 'L', 'K'};

// pressure to read PSI
static inline float readPsi() {
  float hPa = mpr.readPressure();
  return hPa / 68.947572932f;
}
```

```
void setup() {
  Serial.begin(9600);
  Wire.begin();
  Keyboard.begin();

  // establishes button convention
  pinMode(BUTTON_PIN, INPUT_PULLUP);

  // initializes setup for each sensor
  for (int i = 0; i < NUM_SENSORS; i++) {
    tcaSelect(sensorChannels[i]);
    delay(5);
    if (!mpr.begin()) {
      Serial.print("MPRLS NOT found on channel ");
      Serial.println(sensorChannels[i]);
    } else {
      Serial.print("Found MPRLS on channel ");
      Serial.println(sensorChannels[i]);
    }
  }

}

// detect button press to zero sensors
void loop() {
  // create a constant boolean to define button state
  bool btn = digitalRead(BUTTON_PIN);
  // if button is pressed, zero sensors
  if (prevBtnState == HIGH && btn == LOW) {
    // print to serial monitor
    Serial.println("Zeroing all Pressure Sensors!");
    // loops through all sensors and zeros
    for (int i = 0; i < NUM_SENSORS; i++) {
      tcaSelect(sensorChannels[i]);
      delay(5);
      zeroPsi[i] = readPsi();
      zeroed[i] = true;
      hitLatched[i] = false;
      lastHitMs[i] = 0;

      // print zeroing PSI to serial monitor
      Serial.print("  Sensor ");
      Serial.print(i);
      Serial.print(" zeroed at ");
      Serial.print(zeroPsi[i], 3);
```

```
      Serial.println(" psi");
    }
  }
  prevBtnState = btn;

  unsigned long now = millis();

  // loops through each sensor
  for (int i = 0; i < NUM_SENSORS; i++) {
    if (!zeroed[i]) continue;
    // selects the channel currently being indexed
    tcaSelect(sensorChannels[i]);
    delay(2);

    // reads current pressure and establishes hit detection logic
    float psi = readPsi();
    float trigger = zeroPsi[i] + HIT_DELTA_PSI;
    float release = trigger - RELEASE_THRESHOLD;

    // if hit was detected, prints vlaues to the serial monitor
    // and prints letter
    if (!hitLatched[i]) {
      if (psi >= trigger && (now - lastHitMs[i]) >= COOLDOWN_MS) {
        Serial.print("Hit on sensor ");
        Serial.print(i);
        Serial.print(" psi=");
        Serial.println(psi, 3);

        Keyboard.write(sensorKeys[i]);

        hitLatched[i] = true;
        lastHitMs[i] = now;
      }
    } else {
      if (psi <= release) {
        hitLatched[i] = false;
      }
    }
  }

  delay(5);
}
```