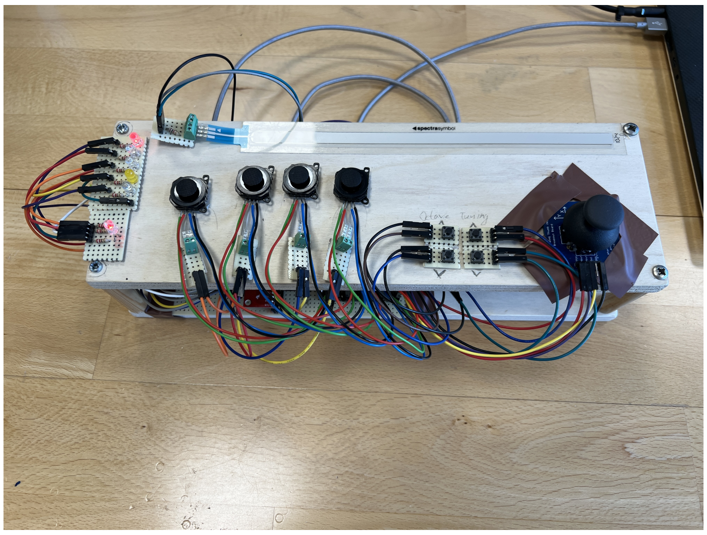
Soundbox extended with MIDI Polyphonic Expression

Master Project Report

Alexander Sela



Aalborg University
SMC

AALBORG UNIVERSITY
STUDENT REPORT

SMC10
Aalborg University
http://www.aau.dk

Title:
Soundbox extended with MIDI Polyphonic Expression

Theme:
Master Thesis

Project Period:
Spring Semester 2025

Project Group:

Participant(s):
Alexander Sela

Supervisor(s):
Dan Overholt

Page Numbers: 55

Date of Completion:
May 25, 2025

Abstract:

To find a possible solution for creating a more mobile and more versatile MIDI Polyphonic Expression Controller, this project aimed at developing a multi-interface MIDI Polyphonic Expression Controller. The prototype combined two interfaces, a ribbon interface and an oblique interface. The oblique interface used joysticks mounted on top of FSR sensors, while the ribbon sensor used a separate joystick and a SoftPod potentiometer. The device used a normal Teensy and was set up as a MIDI Polyphonic Expression Controller to offer a more expressive sound. Mobility was achieved by keeping the physical footprint of the device as small as possible by placing the electronics in a sandwich-like manner. The versatility was achieved by combining multiple interfaces.

Contents

Acknowledgments

I want to start by expressing my gratitude to Dan Overholt, my supervisor, for his support of this project. His support helped to improve certain details of the prototype, and each meeting gave me more motivation for the project. I also want to thank my parents for their continued support, without it this thesis would hardly be possible.

Abbreviations

A list of the abbreviations used in this report, sorted in alphabetical order:

FSR	Force Sensing Resistor
MIDI CI	MIDI Capability Inquiry
MPE	MIDI Polyphonic Expression
UMP	Universal MIDI Package

Table 1: Table with the used abbreviations.

Chapter 1

Introduction

The goal of the master project was to create a MIDI Polyphonic Expression Controller, a device that can generate the sound of multiple instruments and extend the sound with some effects, such as pitch bend or distortion. Additionally, an optional or secondary goal was to create a kind of library for the MIDI Polyphonic Expression commands. The minimum viable product of the project was to create a device that can combine / offer the sound of 2 instruments and the option to apply effects (like pitch bend, delay, or distortion). From a musical perspective, this work aimed to create a MIDI Polyphonic Expression Controller that offered multiple interfaces and increased expressivity. From an academic point of view, the implementation of MIDI Polyphonic Expression on a normal Teensy with the normal usbMIDI libary should help facilitate further development of MIDI Polyphonic Expression, by showing that with a bit of adaption normal MIDI supporting devices can also work with MIDI Polyphonic Expression.
MIDI has been around since 1983 [1] and has since seen some changes, such as the introduction of MIDI Polyphonic Expression in 2018 [2] and the introduction of MIDI 2.0 in 2020 [2]. In addition to MIDI, there were significant efforts to surpass the limitations of MIDI. ZIPI, designed in the mid-1990s, was intended to offer higher resolution control and more flexible messaging than MIDI, including per-note timbral variations and a star network topology, but was never commercially adopted [3, 4, 5]. Around 2000 Yamaha's mLan aimed to transport multichannel digital audio and MIDI over FireWire for synchronized high-speed communication between musical instruments but also had limited adoption and is no longer available [6, 7].
This work consists of multiple parts. The first is supposed to offer an overview of the current availability of MIDI Polyphonic Expression controllers and to introduce MIDI 1.0, MIDI Polyphonic Expression or MPE, and MIDI 2.0. The second is supposed to show how MIDI Polyphonic Expression can be implemented on a Teensy by just using the usbMIDI library, that is included in the Teensy, with the help of a DIY MPE controller. The third and final part is to round everything off and find a conclusion.

Chapter 2

State of the art

2.1 MIDI Polyphonic Expression Controller

The development of MIDI polyphonic expression controllers has produced a variety of different controllers with different interfaces. In the following section, a selection of different MIDI polyphonic expression controllers has been collected and was presented grouped by the type of interface.

Type of Interface	**Product**
Grid	• 203 Electronics Mystrix • Ableton Push 3 • birdkids OffGrid • Galax Synthesizer Co. Galax Balsam • Keith McMillen Instruments QuNeo • Polyend Medusa • unknown devices Topo T16
Guitar	• Artiphon Chorda • Artiphon INSTRUMENT 1 • Sensy Guitar • StretchiI • Zivix Jamstik Classic MIDI Guitar • Zivix Jamstik Deluxe MIDI Guitar • Zivix Jamstik Standard MIDI Guitar • Zivix Jamstik Studio MIDI Guitar
Keyboard	• Ashun Sound Machines Hydrasynth Deluxe • Ashun Sound Machines Hydrasynth Explorer • Ashun Sound Machines Hydrasynth Keyboard • Expressive E Osmose • Keith McMillen Instruments K-Board-C • Keith McMillen Instruments K-Board Pro 4 • Keith McMillen Instruments QuNexus • Korg Keystage 49 • Korg Keystage 61

Table 2.1: Table with various MIDI Polyphonic Expression controllers (Source: [8])

Type of Interface	Product
Keyboard	• ROLI LUMI Keys Studio Edition • ROLI Piano M • ROLI Seaboard 2 • ROLI Seaboard Block • ROLI Seaboard BLOCK M • ROLI Seaboard Block Studio Edition • ROLI Seaboard GRAND Limited First Edition • ROLI Seaboard GRAND Stage • ROLI Seaboard GRAND Studio • ROLI Seaboard M • ROLI Seaboard RISE 2 • ROLI Seaboard RISE 25 • ROLI Seaboard RISE 49
Oblique	• Intuitive Instruments Exquis • Joyst JV-1 • Netz • Shaping the Silence HexBoard • Snyderphonics Manta • SOMA Laboratory TERRA • Striso Musical Instruments Striso board • Striso Musical Instruments Striso duet

Table 2.2: Table with various MIDI Polyphonic Expression controllers (Source: [8])

Type of Interface	**Product**
Ribbon	• Aodyo Instruments Loom • DigiAuxine inc. Kaoline • Haken Audio Continuum Fingerboard 46L2x (Half-size) • Haken Audio Continuum Fingerboard 94L2x (Full-size) • Haken Audio ContinuuMini 29M2x • Haken Audio Slim46 Continuum 46s6x • Haken Audio Slim70 Continuum 70s6x
Spatial	• Artiphon Orba • Artiphon Orba 2 • Artiphon Orba 3 • Audima Labs' Sway • this.is.NOISE.inc MIDI BLASTER • Tirare
Surface	• embodme ERAE Touch • embodme ERAE II • Joué Music Instruments Joué Play Pro Option • Madrona Labs Soundplane Model A • Roger Linn Design LinnStrument • Roger Linn Design LinnStrument 128 • ROLI Lightpad Block • ROLI Lightpad Block M • ROLI Lightpad Block Studio Edition • Sensel Morph

Table 2.3: Table with various MIDI Polyphonic Expression controllers (Source: [8])

Type of Interface	Product
Wind	• Eigenlabs Eigenharp Alpha • Eigenlabs Eigenharp Pico • Eigenlabs Eigenharp Tau • Lekholm Instruments DM48 Digital Chromatic Harmonica • Lekholm Instruments DM48X Digital Chromatic Harmonica
Zones	• Keith McMillen Instruments BopPad RED • Skoogmusic Skwitch

Table 2.4: Table with various MIDI Polyphonic Expression controllers (Source: [8])

Based on the interface, the MIDI Polyphonic Expression controllers can be distinguished into different categories. An overview of the different categories was displayed in the tables 2.1, 2.2, 2.3, and 2.4. The following is a short differentiation of the different interface types that was created by looking up selected devices of each interface categorie and concluding what those devices have in common.

2.1.1 Grid Interface

This category of controllers was designed in that way that the main interaction was structured in a grid-like pattern.

2.1.2 Guitar Interface

Specified a category of controllers, which had an interface layout and playing style that resembled that of an actual guitar. Most controllers of that category support techniques like bends, slides, and vibrato on individual "strings".

2.1.3 Keyboard Interface

Specified a category of controllers, which had an interface similar to that of a real keyboard or piano. Individual keys measure not just velocity but also per-note pressure and sometimes X/Y movement for pitch and timbre control.

2.1.4 Oblique Interface

Oblique interface had interfaces in a special form or the inputs were formed / made in a special way. The devices featured for example joysticks as input options or buttons in non-grid-like patterns. The notes are positioned diagonally or slanted, oftentimes to allow for other scale layouts or more comfortable finger movements.

2.1.5 Ribbon Interface

As a substitute for individual keys or pads, ribbon interfaces frequently use a continuous touch-sensitive strip. This enables microtonal control and smooth pitch glides.

2.1.6 Spatial Interface

The controllers involve 3D gesture tracking using sensors for non-contact or near-surface control. A spatial interface enables or uses the gestures of the performer to create or control sound.

2.1.7 Surface Interface

This interface type is controlled by touch and can have interface shapes similar to other categories. A flat touch surface with an optional physical note separation is a common element of this interface design. It frequently has an adjustable layout and supports multidimensional touch.

2.1.8 Wind Interface

These controllers mimic the expressive control and playing style of wind instruments. Typically, they have breath sensors, and occasionally keys to record more intricate details.

2.1.9 Zones Interface

This type of input interface is marked by having a dedicated zone or area that needs to be interacted with to create a sound, similar to drums. Differences in the strength of an interaction can trigger different effects. The playing surface can be split into different "zones", each assigned to different sounds, instruments, or articulations.

2.2 MIDI

A protocol known as MIDI made it possible for various musical instruments and gadgets to exchange digital messages with one another. MIDI was initially released in 1983 as a result of several manufacturers working together [1]. MIDI 1.0, the original version or standard, is still in use today. Over the years several updates or versions were developed, like MIDI Polyphonic Expression, and MIDI 2.0.

2.2.1 MIDI 1.0

The standard MIDI 1.0 was officially adopted in 1983, and used on the technical side, the standard message sizes of 2-3 bytes, where the data bytes have a precision of 7 bits [2].

There are restrictions with MIDI 1.0. Its keyboard bias is the first thing to notice. There was not much representation of notes that were not part of the western 12-tone equal temperament or music from non-keyboard instruments. The MMA published the MIDI Tuning standard in 1991 for the non-equal temperament application, allowing the user to individually tune the frequencies of the 128 note numbers. However, the manufacturers discontinued adopting this standard [2]. Second, there is a low resolution of controller values. Only seven bits of information are available for data transmission in MIDI 1.0. This reaches its limits when it comes to encoding filters, although it is sufficient for the notes in the 12-tone standard and to encode volume information. This issue was less of an issue with pitch bend because it used 14 bits of resolution by default. Additionally, there was a chance to use the high resolution velocity prefix message to get around the constraint. Generally speaking, the restriction was circumvented or lessened by sending a note on a message using the higher seven bits after sending a regular message using the bottom seven bits. The general CC messages were sent in message pairs and enlarged to 14 bits in the same manner. The corresponding low 7 bits were changed by CC 32–63, and the high 7 bits by CC 0-31. The drawback of this process was that it may potentially produce audio problems because the lower seven bits were presumed to be 0 until the lowest seven bits arrived [2]. Thirdly, the MIDI protocol was a one-way communication system. As a result, the MIDI device is unable to identify the device to which it is transmitting data [2].

2.2.2 MIDI Polyphonic Expression

MIDI Polyphonic Expression was an extension of MIDI 1.0 that enabled the controller to be played polyphonically by the musician by altering the timbre and pitch of individual notes [9]. MIDI Polyphonic Expression was adopted in 2018 [2].

MPE made it possible for electronic instruments, like synthesizers, to express themselves at a level that is normally only achievable with acoustic instruments. Before MPE, all notes were impacted by expressive synthesizer gestures like vibrato or pitch bending. MPE allows a musician to express themselves much more expressively by articulating each note separately. Each note in MPE has its own MIDI Channel, allowing for the application of channel-wide expression messages to each note separately. This is used by music-making devices (like the ROLI Seaboard, Moog's Animoog, and Apple's Logic) to allow musicians to control their fingers in a variety of ways, including left and right, forward and back, downward pressure, and more [10].

There was no per-note expression available in the original MIDI. Every note on the channel was impacted by the conventional pitch bend. Several manufacturers with different systems attempted to overcome this by using MIDI and adding their own protocols. The "Multi-Dimensional Polyphonic Expression" specification was developed by Roli in 2016 and evolved into the MIDI Polyphonic Expression in 2018. Each note was played on a separate channel in the MIDI Polyphonic Expression, which set it apart from standard MIDI. Up to 15 notes of polyphony are possible with this, and one channel was designated as the master channel, which regulates all other channels [2].

On a technical site, one of the biggest changes compared to normal MIDI 1.0 was the change in channel allocation. Like mentioned before, with MIDI Polyphonic Expression each note was now assigned to its own MIDI channel, typically in a 16 MIDI channel range. One channel was regarded as the Master channel for global control [11, 12, 13]. Each note was now able to independently transmit pitch bend, channel pressure or polyphonic aftertouch, control change on channel 74, as well as note on/off and velocity data [11, 12, 13].

Commonly MIDI Polyphonic Expression is described to have multiple actions that each note captures, which are described as "Strike", "Press", "Slide", "Glide", and "Lift" [14, 15, 16]. "Strike" described the key on intensity or attack velocity, "Press" was the term for aftertouch, "Slide" was for the channel control 74, "Glide" was used for pitch bend, and "Lift" was the term used for the key off intensity or release velocity [16].

2.2.3 MIDI 2.0

MIDI 2.0 was the latest addition to MIDI and was announced in 2020 [2]. With MIDI Capability Inquiry (MIDI-CI), MIDI 2.0 introduced a two way communication technique while maintaining backwards compatibilty with MIDI 1.0 and MIDI Polyphonic Expression (MPE). Additionally, MIDI 2.0 introduced a new data format called Universal MIDI Package (UMP), which enables more expressive and high resolution data [2].

MIDI Capability Inquiry

Bidirectional communication between MIDI devices was made possible by MIDI Capability Inquiry, which also included certain additional features like Profile Configuration, Property Exchange, and Process Inquiry [2].

Universal MIDI Package

One, two, three, or four 32 bit words that encode different kinds of messages are called Universal MIDI Package (UMP) words. These messages include 64 bit data messages, 128 bit data messages, MIDI 1.0 channel voice messages, MIDI 2.0 channel voice messages, utility messages, and system real time and common messages [2].

Message Type, Group, and Status fields are present in every Universal MIDI Package (UMP). In each Universal MIDI Package (UMP), the first four bits specify each Message Type [2].

Advancements over MIDI 1.0

The resolution of the data fields was raised by numerous MIDI specification upgrades, enabling more realistic sounds and more information in MIDI performance and creation [2].

Note attributes, per-note control, and new pitch representations were all introduced with MIDI 2.0. Note attributes, such as establishing a synth's waveform, are additional data fields in the Note On and Note Off messages for every played note. There are tow kinds of per note controller messages: assignable per note controller messages and registered per note controller messages. Both kinds function similarly to MIDI 1.0 control change messages, however they only impact one note index rather than a whole channel [2].

Additionally, two new pitch representations, pitch 7.9 and pitch 7.25, were added in MIDI 2.0. Pitch 7.9 was designated as attribute number 3, enabling the note's pitch to be represented in the attribute field using 16 bits. With a resolution of $\frac{1}{512}$ semitones (around 0.2 cents), the final 9 bits represent a fraction of a semitone, while the initial 7 bits represent the semitone like a Note On message without an attribute. This feature allows the note number field to function as a note index without encoding any pitch or scale [2]. It was determined that Pitch 7.25 was Registered Per Note Contoller Number 3. With a resolution of $\frac{1}{33554432}$ semitones (about $2 * 10^-7$ cents), the final 25 bits set the pitch as a fraction of a semitone, while the first 7 bits set the pitch once more as a semitone [2].

Vs MIDI Polyphonic Expression

Compared to MIDI Polyphonic Expression (MPE) in MIDI 1.0, the Registered Per Note Controller and Assignable Per Note Controller messages allow for per-note expressivity in a more streamlined format. Due to MIDI 2.0's backward compatibility, any MIDI device that transmits MIDI Polyphonic Expression (MPE) information can operate in a MIDI 2.0 environment by using a profile that specifies zoning and channeling splitting, just like MIDI Polyphonic Expression (MPE). However, new MIDI devices will probably only use the new Universal MIDI Package (UMP) messages for enhanced expressivity in MIDI devices [2].

Chapter 3

Analysis

3.1 My Features and Design Requirements

The design requirements for this prototype were the following:

- should have multiple interfaces
- should be easy and fun to use
- should offer some additional features, like octave shifting and different tunings
- should have good feedback about octave and tuning settings
- the design and button layout or placement should be comfortable
- should not take up much space

Then these design requirements were then distinguished further between essential features and optional features. Here, the separation showed that the requirement of having multiple interfaces, offering a fun experience, having some additional features, like octave shifting and different tunings, and providing good feedback about octave and tuning settings, all supported the main goal of making a versatile controller. Those were the essential features. The option of having a comfortable design and button layout and that it should not take up much space were more optional features.

Chapter 4

Implementation

4.1 Joystick

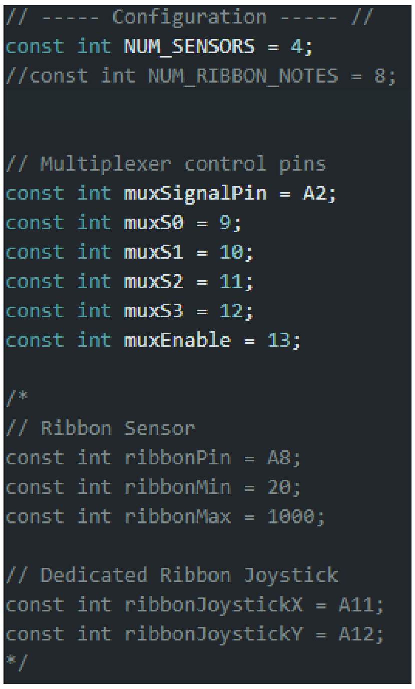
For user input, a group of joysticks was considered. The first choice was the Joystick2 unit [17], based on a recommendation from Daniel Overholt (D. Overholt, personal communication, January 16, 2025). Due to availability, this was replaced with normal "PS2 Arduino Joysticks" [18]. Later on in the prototyping phase these were again replaced by "Thumb Slide Joystick" [19], based on another recommendation from Daniel Overholt (D. Overholt, personal communication, April 10, 2025). Those were then used as the main inputs of the first interface, while the second interface continued to use one of the "PS2 Arduino Joysticks" [18]. Joysticks were chosen as control input, due to the fact that for one a joystick can be controlled by one finger, and second one joystick can control multiple parameters at once.

4.2 First Prototype

This section has the objective of educating the reader about the software solution running on the Teensy. It first starts with the first prototype where both interfaces were separated, and it ends with the software of the final prototype that combined both interfaces.

4.2.1 Interface 1

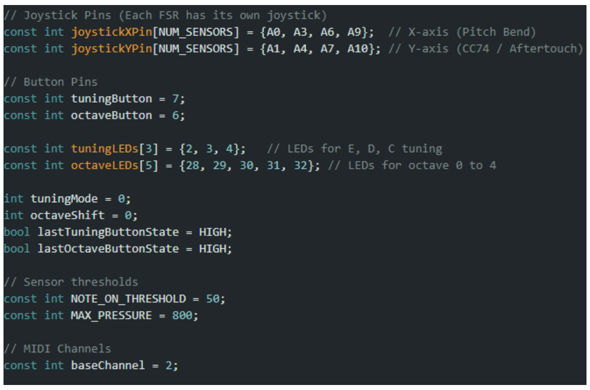
The first interface was designed to be a mixture of surface and oblique interface. The main inputs were measured by FSR sensors and the effects were controlled by joysticks. The main inspiration for the code measuring the touch input was from the "Virus Pad" project from KontinuumLabs [20, 21]. The main inspiration to use joysticks to control the effects was from the "MPE-DIY-Device" from Mathias Brüssel [22, 23, 16]. The pitch bend effect, which was controlled by the joystick, was based on code examples [24, 25, 21]. Primarily the first interface used to measure the different pressure values, measured by the FSRs, if the read value was higher than a specific threshold, the code would send a note on command on a specific channel. Since it is an MPE controller, each FSR was connected to one channel, and therefore each note had their dedicated channel. Each channel would transmit one note depending on which FSR is triggered [11]. The strength of the impulse is also the strength of the impulse. Each FSR was located beneath its own joystick that was programmed to apply the effects (Pitch bend, Modulation, and Aftertouch) to the specific note. The first interface therefore offered a discrete interaction, as one would have when playing a piano. For the code, it started with the normal declaration of variables that would be needed later on (See Figures 4.1, 4.2, and 4.3).



```
// ----- Configuration ----- //
const int NUM_SENSORS = 4;
//const int NUM_RIBBON_NOTES = 8;


// Multiplexer control pins
const int muxSignalPin = A2;
const int muxS0 = 9;
const int muxS1 = 10;
const int muxS2 = 11;
const int muxS3 = 12;
const int muxEnable = 13;

/*
// Ribbon Sensor
const int ribbonPin = A8;
const int ribbonMin = 20;
const int ribbonMax = 1000;

// Dedicated Ribbon Joystick
const int ribbonJoystickX = A11;
const int ribbonJoystickY = A12;
*/
```

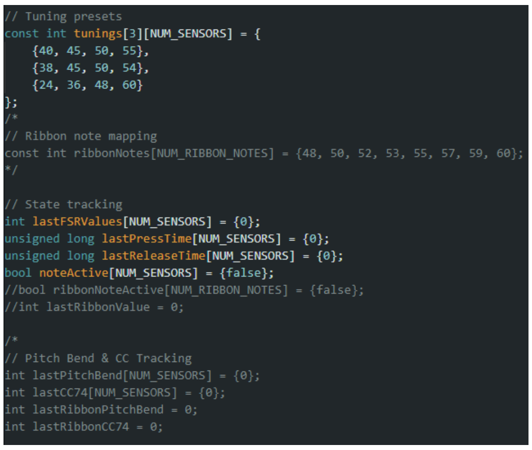
Figure 4.1: Declared Variables

Here, the pin connections for the different components were declared; for example, the multiplexer signal was connected to pin A2, the multiplexer enable pin was connected to pin 13 and the multiplexer pins S0, S1, S2, and S3 were connected to pins 9, 10, 11, 12. It was also declared which analogue pins the joysticks were connected to and to which pins the push buttons for the octave shift and tuning selection were connected, and to which pins the corresponding LEDs were wired to. The sensor threshold for the FSR sensors and the max pressure were defined here as well. Then the base channel was defined from which the channel distribution started, and the MIDI notes for each FSR in each of the 3 tunings were defined. The controller offered the possibility of having 3 different tunings, Tuning E, Tuning D, and Tuning C. It also featured to shift the sound by 4 octaves, from +12 to +48. The tunings and the octave shifts were both inspired by the "MPE-DIY-Device"



```
// Joystick Pins (Each FSR has its own joystick)
const int joystickXPin[NUM_SENSORS] = {A0, A3, A6, A9};  // X-axis (Pitch Bend)
const int joystickYPin[NUM_SENSORS] = {A1, A4, A7, A10}; // Y-axis (CC74 / Aftertouch)

// Button Pins
const int tuningButton = 7;
const int octaveButton = 6;

const int tuningLEDs[3] = {2, 3, 4};   // LEDs for E, D, C tuning
const int octaveLEDs[5] = {28, 29, 30, 31, 32}; // LEDs for octave 0 to 4

int tuningMode = 0;
int octaveShift = 0;
bool lastTuningButtonState = HIGH;
bool lastOctaveButtonState = HIGH;

// Sensor thresholds
const int NOTE_ON_THRESHOLD = 50;
const int MAX_PRESSURE = 800;

// MIDI Channels
const int baseChannel = 2;
```

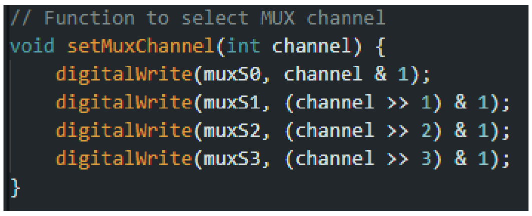
Figure 4.2: Declared Variables



```
// Tuning presets
const int tunings[3][NUM_SENSORS] = {
    {40, 45, 50, 55},
    {38, 45, 50, 54},
    {24, 36, 48, 60}
};
/*
// Ribbon note mapping
const int ribbonNotes[NUM_RIBBON_NOTES] = {48, 50, 52, 53, 55, 57, 59, 60};
*/

// State tracking
int lastFSRValues[NUM_SENSORS] = {0};
unsigned long lastPressTime[NUM_SENSORS] = {0};
unsigned long lastReleaseTime[NUM_SENSORS] = {0};
bool noteActive[NUM_SENSORS] = {false};
//bool ribbonNoteActive[NUM_RIBBON_NOTES] = {false};
//int lastRibbonValue = 0;

/*
// Pitch Bend & CC Tracking
int lastPitchBend[NUM_SENSORS] = {0};
int lastCC74[NUM_SENSORS] = {0};
int lastRibbonPitchBend = 0;
int lastRibbonCC74 = 0;
```

Figure 4.3: Declared Variables

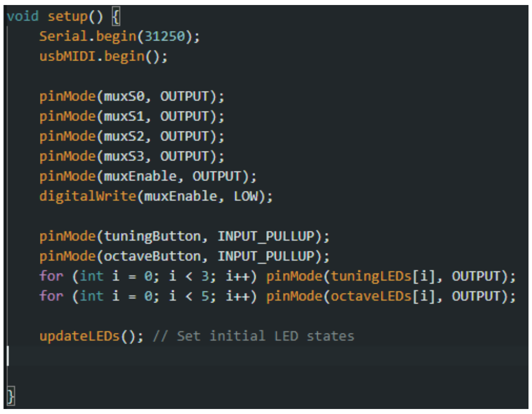
from Mathias Brüssel [22, 23, 16] Before the setup function of, a function was declared to handle the multiplexer (See Fig. 4.4).



```
// Function to select MUX channel
void setMuxChannel(int channel) {
    digitalWrite(muxS0, channel & 1);
    digitalWrite(muxS1, (channel >> 1) & 1);
    digitalWrite(muxS2, (channel >> 2) & 1);
    digitalWrite(muxS3, (channel >> 3) & 1);
}
```

Figure 4.4: Function to handle the multiplexer

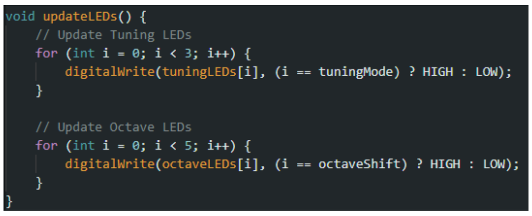
In the setup function (Fig. 4.5), the baud rate was set to 31250 [3], the later used function to handle the LEDs was initialized, and the different pin modes for the buttons, LEDs, and multiplexer pins were declared.



```
void setup() {
    Serial.begin(31250);
    usbMIDI.begin();

    pinMode(muxS0, OUTPUT);
    pinMode(muxS1, OUTPUT);
    pinMode(muxS2, OUTPUT);
    pinMode(muxS3, OUTPUT);
    pinMode(muxEnable, OUTPUT);
    digitalWrite(muxEnable, LOW);

    pinMode(tuningButton, INPUT_PULLUP);
    pinMode(octaveButton, INPUT_PULLUP);
    for (int i = 0; i < 3; i++) pinMode(tuningLEDs[i], OUTPUT);
    for (int i = 0; i < 5; i++) pinMode(octaveLEDs[i], OUTPUT);

    updateLEDs(); // Set initial LED states


}
```

Figure 4.5: Setup section of the code

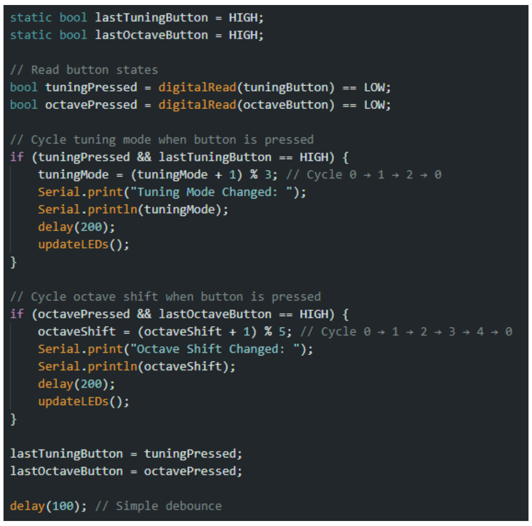
The in the setup initialized "updateLEDs" function (see Fig. 4.6) was positioned at the very end of the code, and was a function that did not take any inputs and did not have any outputs. It consisted of two for loops, the first for loop updating the LEDs responsible to show the tuning the controller is in, and the second for loop updating the LEDs responsible to show the octave the controller is in. The number of available tuning states or octaves, defining the upper limits of the for loops. In both for loops an alternative to an if-else statement was used by comparing the current index with the active octave or tuning mode. The tuning for loop would loop over 3 indexes, the octave for loop would loop over 5 indexes. Each for loop checks if the current index matches the active octave or tuning mode, and if it matched, it would set the corresponding LED to high. Otherwise, to low. At the top of the loop section the first thing was the handling of the buttons to loop through the octaves and tuning states (See Fig. 4.7). For that reason, the first two variables were defined to remember the last state of the octave or tuning button. Then the current button states were read with a "digitalRead()" and set to true when the read pin signals a



```
void updateLEDs() {
    // Update Tuning LEDs
    for (int i = 0; i < 3; i++) {
        digitalWrite(tuningLEDs[i], (i == tuningMode) ? HIGH : LOW);
    }

    // Update Octave LEDs
    for (int i = 0; i < 5; i++) {
        digitalWrite(octaveLEDs[i], (i == octaveShift) ? HIGH : LOW);
    }
}
```

Figure 4.6: Function to handle the LEDs

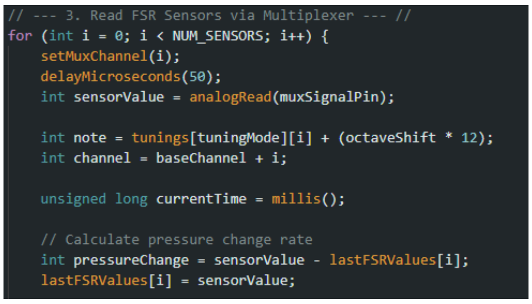
0 (when the button was pressed). Then, two if-statements handled the cycling through the states, one for the tuning modes, and one for the octaves. In short, the if statement would be evaluated as true every time the current button state and the previous button state were high. Whenever the statement was evaluated as true, the tuning mode or octave first incremented by 1, before the modulo operator gave the remainder after the division by 3 or 5, depending on whether looked at the octave statement or the tuning mode statement. This method ensured that the value remained within 0, 1, 2 or 0, 1, 2, 3, 4. Then on the serial monitor it was printed that the tuning mode or octave was changed, followed by a small delay of 200 ms, and the call to the updateLED function (Fig. 4.6) to change the LED accordingly.



```
static bool lastTuningButton = HIGH;
static bool lastOctaveButton = HIGH;

// Read button states
bool tuningPressed = digitalRead(tuningButton) == LOW;
bool octavePressed = digitalRead(octaveButton) == LOW;

// Cycle tuning mode when button is pressed
if (tuningPressed && lastTuningButton == HIGH) {
    tuningMode = (tuningMode + 1) % 3; // Cycle 0 → 1 → 2 → 0
    Serial.print("Tuning Mode Changed: ");
    Serial.println(tuningMode);
    delay(200);
    updateLEDs();
}

// Cycle octave shift when button is pressed
if (octavePressed && lastOctaveButton == HIGH) {
    octaveShift = (octaveShift + 1) % 5; // Cycle 0 → 1 → 2 → 3 → 4 → 0
    Serial.print("Octave Shift Changed: ");
    Serial.println(octaveShift);
    delay(200);
    updateLEDs();
}

lastTuningButton = tuningPressed;
lastOctaveButton = octavePressed;

delay(100); // Simple debounce
```

Figure 4.7: Handling of the buttons in the Loop section

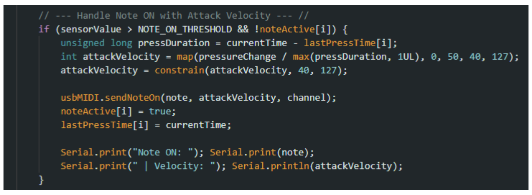
After handling the buttons, a large for loop was declared to read the FSR sensors via the multiplexer. The for loop's number of iterations was defined by the number of active FSR sensors; in the case of this project, it was 4 sensors. The iteration variable was used at the start to set the correct channel of the multiplexer, after which the Teensy would wait 50 microseconds before it read the sensor value. The note associated with the selected sensor was defined by the MIDI number stored in the Tunings matrix and the "octaveShift" variable, which was multiplied by 12. The MIDI number in the "tunings" matrix was set by the "TuningMode" variable and the iteration variable. The row was set by the "TuningMode" variable and the column was set by the iteration variable. The channel, one of the main things to create a proper MPE controller, was set by taking the earlier set base channel plus the current loop iteration. This was to make sure that each sensor and note got its own channel starting from channel 2, the base channel, since channel 1 is and was reserved for global parameters [12, 11, 13]. The current time was then measured, which was later used to define the attack and release velocity. Next was the calculation of how fast the user applied pressure on the FSR sensor to define the attack and release velocity. The pressure change was defined as the current sensor value minus the last read sensor value, which was updated directly in the next line by setting it equal to the currently read sensor value.



```
// --- 3. Read FSR Sensors via Multiplexer --- //
for (int i = 0; i < NUM_SENSORS; i++) {
    setMuxChannel(i);
    delayMicroseconds(50);
    int sensorValue = analogRead(muxSignalPin);

    int note = tunings[tuningMode][i] + (octaveShift * 12);
    int channel = baseChannel + i;

    unsigned long currentTime = millis();

    // Calculate pressure change rate
    int pressureChange = sensorValue - lastFSRValues[i];
    lastFSRValues[i] = sensorValue;
```

Figure 4.8: First part of the loop handling the FSR sensors

The note on handling was done in an if statement to make sure that the note was activated only if the sensor value surpassed a specific threshold and the note was not already active. Here, the duration of the press was measured, the attack velocity was mapped to the range of possible numbers, and then constrained. The "usbMIDI.sendNoteOn" command then took the note, the calculated attack velocity, and the channel. The "noteActive" variable for that note / sensor was set to true, and the "lastPressTime" variable was updated.

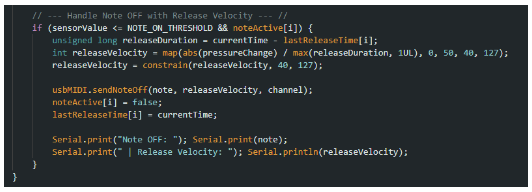


```
    // --- Handle Note ON with Attack Velocity --- //
    if (sensorValue > NOTE_ON_THRESHOLD && !noteActive[i]) {
        unsigned long pressDuration = currentTime - lastPressTime[i];
        int attackVelocity = map(pressureChange / max(pressDuration, 1UL), 0, 50, 40, 127);
        attackVelocity = constrain(attackVelocity, 40, 127);

        usbMIDI.sendNoteOn(note, attackVelocity, channel);
        noteActive[i] = true;
        lastPressTime[i] = currentTime;

        Serial.print("Note ON: "); Serial.print(note);
        Serial.print(" | Velocity: "); Serial.println(attackVelocity);
    }
```

Figure 4.9: Second part of the loop handling the FSR sensors

The third and last part of the FSR sensor handling for loop was another if statement, handling the note off with the release velocity. The statement, to be proven true, was here that the sensor value was below the specified threshold and the current note or sensor was set to active. It started again with calculating the release duration and the release velocity.

The "MIDI.sendNoteOff" command then took the note, the calculated release velocity, and the channel. The "noteActive" variable for the specific note was set back to false, and the "lastReleaseTime" variable was updated with the current time.



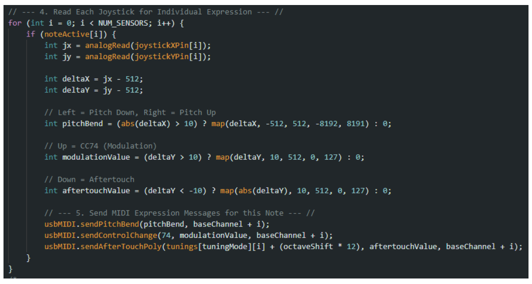
```
    // --- Handle Note OFF with Release Velocity --- //
    if (sensorValue <= NOTE_ON_THRESHOLD && noteActive[i]) {
        unsigned long releaseDuration = currentTime - lastReleaseTime[i];
        int releaseVelocity = map(abs(pressureChange) / max(releaseDuration, 1UL), 0, 50, 40, 127);
        releaseVelocity = constrain(releaseVelocity, 40, 127);

        usbMIDI.sendNoteOff(note, releaseVelocity, channel);
        noteActive[i] = false;
        lastReleaseTime[i] = currentTime;

        Serial.print("Note OFF: "); Serial.print(note);
        Serial.print(" | Release Velocity: "); Serial.println(releaseVelocity);
    }
}
```

Figure 4.10: Third part of the loop handling the FSR sensors

The last part of the first iteration of the first interface was a second for loop, handling the joysticks. This for loop would also just iterate over the number of available FSR sensors. It first checked if the note on the current iteration is active. Then read the x and y axes of the joystick specified for the current iteration / channel. To set the joystick to the center, an intermediate variable was defined, where 512 was subtracted from the currently read joystick value [25]. This was done for both axes. The pitch bend was put on the x-axis, the modulation was put on the upward y-axis and the aftertouch was put on the downward y-axis. For the pitch bend, the code asked if the absolute value of the x-axis intermediate was greater than 10, which was the deadzone value chosen for the joysticks. If that was the case, the code would then map the x-axis intermediate from -512 to 512 to the value range -8192 to 8191, if not the pitch bend value was set to 0. For the modulation, the code checked if the intermediate value was greater than 10. If that was true, the intermediate y-axis value was mapped, in the available range of 10 to 512, to the range of 0 to 127. If not, the modulation value was set to 0. For the aftertouch it was basically similar to the modulation value, but here the if statement looked at if the intermediate value of the y-axis was smaller than 10. If that was the case, the absolute value of the intermediate was mapped, in the available range of 10 to 512, to the range of 0 to 127.

That the pitch bend was mapped to the range of -8192 to 8191 while the aftertouch and modulation were just mapped to the range of 0 to 127, was because the pitch bend uses naturally 14-bit of resolution while modulation and aftertouch remain at the normal 7-bit resolution [25, 2].

In the end, these effects were then sent out by using the standard built-in commands, like "usbMIDI.sendPitchBend", "usbMIDI.sendControlChange",
and "usbMIDI.sendAfterTouchPoly". The control change command needed the modulation channel as well, additionally to the modulation value and midi channel. This was for a typical modulation control channel 74 [16, 13]. For aftertouch, the polyphonic aftertouch function was used, which also needed the currently played note. This was done to ensure that the correct note was affected.



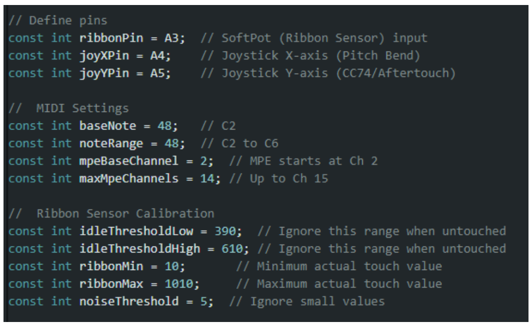
```
// --- 4. Read Each Joystick for Individual Expression --- //
for (int i = 0; i < NUM_SENSORS; i++) {
    if (noteActive[i]) {
        int jx = analogRead(joystickXPin[i]);
        int jy = analogRead(joystickYPin[i]);

        int deltaX = jx - 512;
        int deltaY = jy - 512;

        // Left = Pitch Down, Right = Pitch Up
        int pitchBend = (abs(deltaX) > 10) ? map(deltaX, -512, 512, -8192, 8191) : 0;

        // Up = CC74 (Modulation)
        int modulationValue = (deltaY > 10) ? map(deltaY, 10, 512, 0, 127) : 0;

        // Down = Aftertouch
        int aftertouchValue = (deltaY < -10) ? map(abs(deltaY), 10, 512, 0, 127) : 0;

        // --- 5. Send MIDI Expression Messages for this Note --- //
        usbMIDI.sendPitchBend(pitchBend, baseChannel + i);
        usbMIDI.sendControlChange(74, modulationValue, baseChannel + i);
        usbMIDI.sendAfterTouchPoly(tunings[tuningMode][i] + (octaveShift * 12), aftertouchValue, baseChannel + i);
    }
}
```

Figure 4.11: Part of the loop handling the joystick behaviour

4.2.2 Interface 2

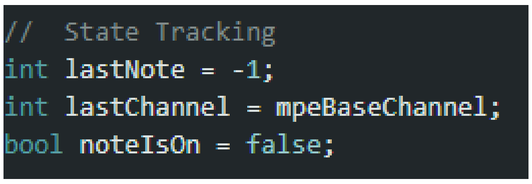
The second interface was designed to offer a more continuous control / interaction of different MIDI notes. Therefore, it used a ribbon sensor that was split into different zones in the code. Each zone was to trigger a different note. An additional joystick was included as well to control again the effects for each note (pitch bend, modulation, and aftertouch).

The very start of the code was again the configuration with the pin declaration and setting up the different variables and limits (See Fig. 4.12, 4.13). Here, for example, was declared at which MIDI note the ribbon sensor should start and what range of MIDI notes it should cover. It was declared on which channel the system should start to send notes on and which was the final channel available. For the ribbon sensor calibration, the min and max values of the sensor were stored, and a noise threshold was declared to ignore smaller input values as noise. This threshold replaced the low and high idle thresholds, that were used, before the noise threshold. The low and high idle thresholds were implemented since in the first internal testing the ribbon sensor would always send values between 400 and 600. This problem was due to the mistake of forgetting to include a voltage divider at the connection of the sensor to the Teensy. After fixing this problem, the low and high idle thresholds were replaced by the noise threshold. The setup loop was in the first iteration quite simple since it only included the begin of the serial communication at the baud rate of 31250 [3] (See Fig. 4.14).



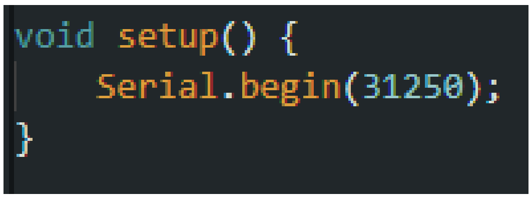
```
// Define pins
const int ribbonPin = A3;  // SoftPot (Ribbon Sensor) input
const int joyXPin = A4;    // Joystick X-axis (Pitch Bend)
const int joyYPin = A5;    // Joystick Y-axis (CC74/Aftertouch)

//  MIDI Settings
const int baseNote = 48;   // C2
const int noteRange = 48;  // C2 to C6
const int mpeBaseChannel = 2;  // MPE starts at Ch 2
const int maxMpeChannels = 14; // Up to Ch 15

//  Ribbon Sensor Calibration
const int idleThresholdLow = 390;  // Ignore this range when untouched
const int idleThresholdHigh = 610; // Ignore this range when untouched
const int ribbonMin = 10;       // Minimum actual touch value
const int ribbonMax = 1010;     // Maximum actual touch value
const int noiseThreshold = 5;  // Ignore small values
```

Figure 4.12: Configuration of the first iteration of the second interface part 1



```
//  State Tracking
int lastNote = -1;
int lastChannel = mpeBaseChannel;
bool noteIsOn = false;
```

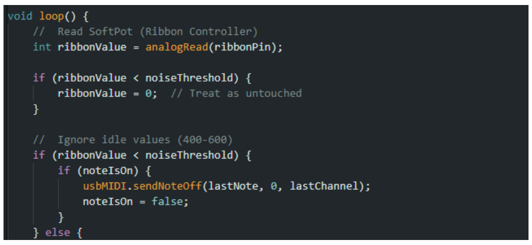
Figure 4.13: Configuration of the first iteration of the second interface part 2



```
void setup() {
    Serial.begin(31250);
}
```

Figure 4.14: Setup loop of the first iteration of the second interface

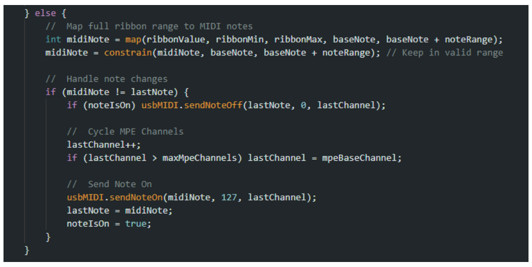
The main loop of the ribbon sensor started with reading the sensor and storing the value as the "ribbonValue". When this value was below the noise threshold, it was assumed to be zero. When the value was below the threshold but the note was played the system was instructed to send the note off command, and the "noteIsOn" variable was set to false (See Fig. 4.15). The noise threshold was introduced to make the system more robust against the small noise the sensor send on its own.



```
void loop() {
    //  Read SoftPot (Ribbon Controller)
    int ribbonValue = analogRead(ribbonPin);

    if (ribbonValue < noiseThreshold) {
        ribbonValue = 0;  // Treat as untouched
    }

    //  Ignore idle values (400-600)
    if (ribbonValue < noiseThreshold) {
        if (noteIsOn) {
            usbMIDI.sendNoteOff(lastNote, 0, lastChannel);
            noteIsOn = false;
        }
    } else {
```

Figure 4.15: First part of the loop of the first iteration of the second interface

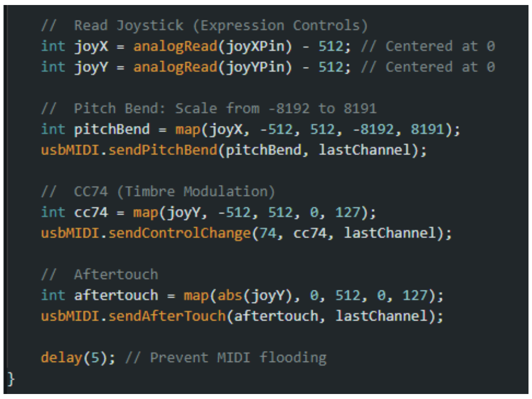
In the case that the input value is over the noise threshold, the ribbon is first mapped to the MIDI notes. The MIDI note is declared by using the mapping function, that mapped the ribbonValue in the range from the minimum ribbon value to the maximum ribbon value to the base note and until the base note plus the note range. Then the constrain function was used to keep the values in the valid range. To handle note changes, another if statement was used. Here the statement was that the current MIDI note is different from the last played MIDI note. When the "noteIsOn" variable was true, the send note off command was used on the last played note on the last used channel. Then the channel was incremented and it was tested if the system had reached the last available channel, in which case the last channel was set back to the base or starting channel. Afterwards, the send note on command was executed with the currently selected MIDI note on the last channel. The last used note was set to the currently selected note and the "noteIsOn" variable was set to true (See Fig. 4.16).



```
} else {
    //  Map full ribbon range to MIDI notes
    int midiNote = map(ribbonValue, ribbonMin, ribbonMax, baseNote, baseNote + noteRange);
    midiNote = constrain(midiNote, baseNote, baseNote + noteRange); // Keep in valid range

    //  Handle note changes
    if (midiNote != lastNote) {
        if (noteIsOn) usbMIDI.sendNoteOff(lastNote, 0, lastChannel);

        //  Cycle MPE Channels
        lastChannel++;
        if (lastChannel > maxMpeChannels) lastChannel = mpeBaseChannel;

        //  Send Note On
        usbMIDI.sendNoteOn(midiNote, 127, lastChannel);
        lastNote = midiNote;
        noteIsOn = true;
    }
}
```

Figure 4.16: Second part of the loop of the first iteration of the second interface

The last part was the handling of the joystick. Here it started again with reading the pins and storing the incoming values on variables. This time the joysticks were also directly centred by directly subtracting 512 from the incoming value [25]. The pitch bend was again mapped onto the x-axis. The x-axis value was mapped in the range from -512 to 512 to the range of -8192 to 8191. Then this pitch bend value was used as the input value of the "usbMIDI.sendPitchBend" function together with the last used channel. Then the modulation was created by mapping the y-axis value in the range from -512 to 512 to the range 0 to 127. This value was then send with the "usbMIDI.sendControlChange" function together with the number of the control channel it should control and the last used channel. In this case the control channel was channel 74, since this is the normal control channel controlling the modulation [16, 13]. The aftertouch value was mapped similarly. It used the absolute value of the y-axis value in the range from 0 to 512, that was mapped to 0 to 127. This was then used as an input function of the "usbMIDI.sendAfterTouch" function together with the last used channel (See Fig. 4.17).



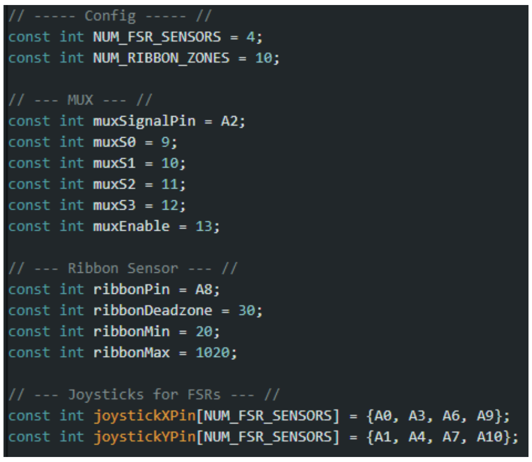
```
    //  Read Joystick (Expression Controls)
    int joyX = analogRead(joyXPin) - 512; // Centered at 0
    int joyY = analogRead(joyYPin) - 512; // Centered at 0

    //  Pitch Bend: Scale from -8192 to 8191
    int pitchBend = map(joyX, -512, 512, -8192, 8191);
    usbMIDI.sendPitchBend(pitchBend, lastChannel);

    //  CC74 (Timbre Modulation)
    int cc74 = map(joyY, -512, 512, 0, 127);
    usbMIDI.sendControlChange(74, cc74, lastChannel);

    //  Aftertouch
    int aftertouch = map(abs(joyY), 0, 512, 0, 127);
    usbMIDI.sendAfterTouch(aftertouch, lastChannel);

    delay(5); // Prevent MIDI flooding
}
```

Figure 4.17: Code to handle the joystick for the ribbon controller

4.3 Final Prototype

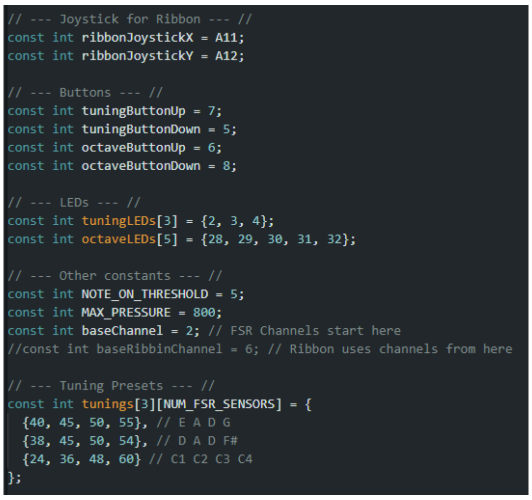
As mentioned above, the main inspiration for using Joysticks as an interface was the "MPE-DIY-Device" from Mathias Brüssel [22, 23, 16]. From this project the tuning modes and octaves and the decision which axis of each joystick controls which effect was inspired by. The final prototype combined both interfaces and added some extras, the first iterations of the separate codes missed. The final version of the code started again with the variable declaration where all the pins and most of the needed variables were configured. The first variables were the number of FSR sensors used and the number of zones in which the ribbon sensor should be divided. To set the number of FSR sensors used and number of zones on the ribbon sensor as variables allowed to make it more open to other people should they ever want to recreate the prototype but want to add more zones or FSR sensors. The multiplexer pins were then declared before the ribbon sensor pins were declared together with the ribbon deadzone and the ribbon min and max values. Then the joystick pins were declared, first for the four joysticks used with the FSR sensors and then for the separate joystick for the ribbon sensor. The pins for the four buttons were set up, before the pins for the LEDs for the tuning and octave. Afterwards, some more variables were declared, like the "NOTE_ON_THRESHOLD" or the "MAX_PRESSURE" or the base channel where the whole system should start from. The "NOTE_ON_THRESHOLD" was used to prevent noise on the FSR sensors to trigger a sound, and the "MAX_PRESSURE" was used to declare the maximum pressure the user could reach on a single FSR sensor. It was later used to scale the FSR input. The base channel was set to 2 since MIDI Polyphonic Expression uses for each note a separate channel starting with channel 2, because channel 1 was or is reserved for global parameters [12, 13]. For that reason, the MIDI Polyphonic Expression controller had channels 2 to 15 to put notes on [12, 11, 13]. Next, the tuning presets were declared. It was set up as a matrix with 3 rows, and the number of coloumns was the number of FSR sensors. In the matrix, each row was a different tuning and each coloumn was one of the FSR sensors. Tuning 1 was E, A, D, G; Tuning 2 was D, A, D, F#; and Tuning 3 was C1, C2, C3, C4. This was inspired by the "MPE-DIY-Device" from Mathias Brüssel [22, 16, 23]. Then the possible notes of the ribbon sensor were declared, which were C3, D3, D#3, F#3, G, G#3, A#, B, C4, and D4. This was inspired by the C minor gypsy tuning [26]. This was followed by several state variables that tracked the state of several things, like, for example, the buttons and in which octave and tuning mode the device was. But it also tracked the value of the previous FSR input or if the current FSR sensor is active.

In the setup loop, serial communication was set to a baud rate of 31250 [3]. Then the pin modes of the different pins were declared, starting with the pins for the multiplexer. The multiplexer pins were all declared as outputs and the enabled pin was set to low to enable the multiplexer. Then the pins for the buttons were declared, for them the built-in pull-up resistors were activated. The pins for the LEDs were all set to be outputs. Lastly, the function to updated the LEDs based on the button presses was declared.



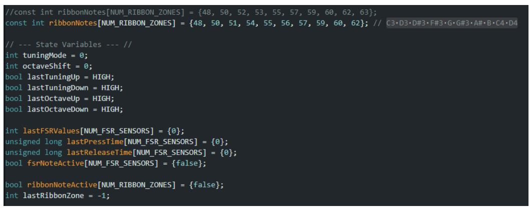
```
// ----- Config ----- //
const int NUM_FSR_SENSORS = 4;
const int NUM_RIBBON_ZONES = 10;

// --- MUX --- //
const int muxSignalPin = A2;
const int muxS0 = 9;
const int muxS1 = 10;
const int muxS2 = 11;
const int muxS3 = 12;
const int muxEnable = 13;

// --- Ribbon Sensor --- //
const int ribbonPin = A8;
const int ribbonDeadzone = 30;
const int ribbonMin = 20;
const int ribbonMax = 1020;

// --- Joysticks for FSRs --- //
const int joystickXPin[NUM_FSR_SENSORS] = {A0, A3, A6, A9};
const int joystickYPin[NUM_FSR_SENSORS] = {A1, A4, A7, A10};
```

Figure 4.18: Variable declaration in the final code.



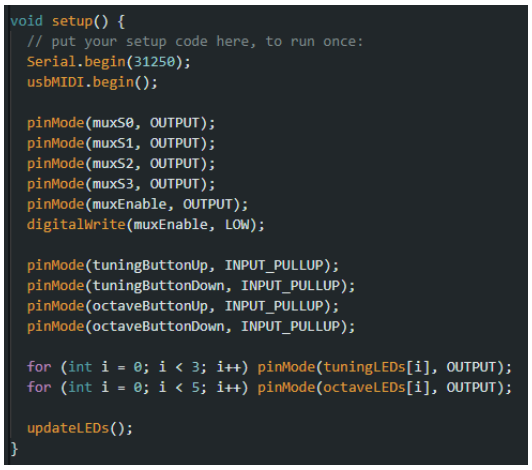
```
// --- Joystick for Ribbon --- //
const int ribbonJoystickX = A11;
const int ribbonJoystickY = A12;

// --- Buttons --- //
const int tuningButtonUp = 7;
const int tuningButtonDown = 5;
const int octaveButtonUp = 6;
const int octaveButtonDown = 8;

// --- LEDs --- //
const int tuningLEDs[3] = {2, 3, 4};
const int octaveLEDs[5] = {28, 29, 30, 31, 32};

// --- Other constants --- //
const int NOTE_ON_THRESHOLD = 5;
const int MAX_PRESSURE = 800;
const int baseChannel = 2; // FSR Channels start here
//const int baseRibbinChannel = 6; // Ribbon uses channels from here

// --- Tuning Presets --- //
const int tunings[3][NUM_FSR_SENSORS] = {
  {40, 45, 50, 55}, // E A D G
  {38, 45, 50, 54}, // D A D F#
  {24, 36, 48, 60} // C1 C2 C3 C4
};
```

Figure 4.19: Variable declaration in the final code.



```
//const int ribbonNotes[NUM_RIBBON_ZONES] = {48, 50, 52, 53, 55, 57, 59, 60, 62, 63};
const int ribbonNotes[NUM_RIBBON_ZONES] = {48, 50, 51, 54, 55, 56, 57, 59, 60, 62}; // C3·D3·D#3·F#3·G·G#3·A#·B·C4·D4

// --- State Variables --- //
int tuningMode = 0;
int octaveShift = 0;
bool lastTuningUp = HIGH;
bool lastTuningDown = HIGH;
bool lastOctaveUp = HIGH;
bool lastOctaveDown = HIGH;

int lastFSRValues[NUM_FSR_SENSORS] = {0};
unsigned long lastPressTime[NUM_FSR_SENSORS] = {0};
unsigned long lastReleaseTime[NUM_FSR_SENSORS] = {0};
bool fsrNoteActive[NUM_FSR_SENSORS] = {false};

bool ribbonNoteActive[NUM_RIBBON_ZONES] = {false};
int lastRibbonZone = -1;
```

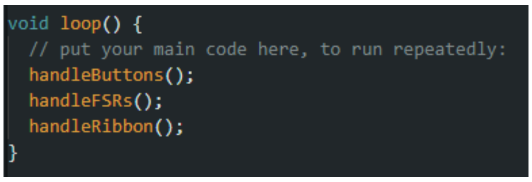
Figure 4.20: Variable declaration in the final code.



```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(31250);
  usbMIDI.begin();

  pinMode(muxS0, OUTPUT);
  pinMode(muxS1, OUTPUT);
  pinMode(muxS2, OUTPUT);
  pinMode(muxS3, OUTPUT);
  pinMode(muxEnable, OUTPUT);
  digitalWrite(muxEnable, LOW);

  pinMode(tuningButtonUp, INPUT_PULLUP);
  pinMode(tuningButtonDown, INPUT_PULLUP);
  pinMode(octaveButtonUp, INPUT_PULLUP);
  pinMode(octaveButtonDown, INPUT_PULLUP);

  for (int i = 0; i < 3; i++) pinMode(tuningLEDs[i], OUTPUT);
  for (int i = 0; i < 5; i++) pinMode(octaveLEDs[i], OUTPUT);

  updateLEDs();
}
```

Figure 4.21: Setup loop in the final code.

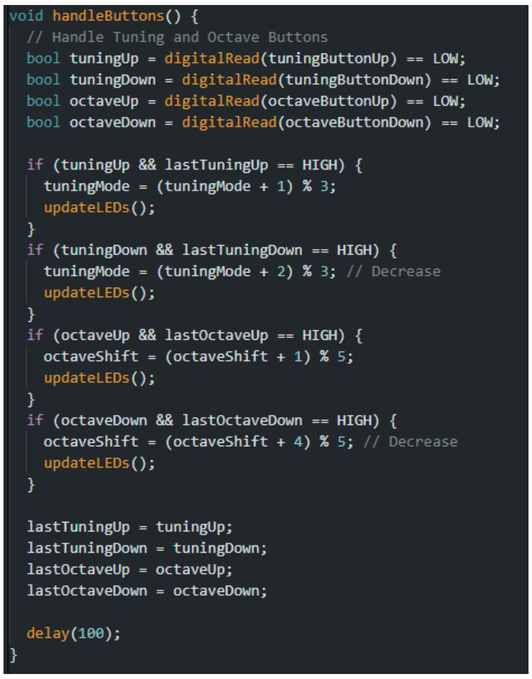
In the final version of the code, it was decided to put the main code handling the different parts into separate functions, which then only needed to be called in the main loop (See Fig. 4.22). For that reason, three functions were created, "handleButtons()", "handleFSRs()", and "handleRibbon()". This was done to make the code more readable and better understandable.



```
void loop() {
  // put your main code here, to run repeatedly:
  handleButtons();
  handleFSRs();
  handleRibbon();
}
```

Figure 4.22: Main loop in the final code.

First, the "handleButtons()" function (See Fig. 4.23). The function first evaluates which button was pressed by reading the button pin with a "digitalRead()" and set the corresponding variable ("tuningUp", "tuningDown", "octaveUp", and "octaveDown") to be true when the pin signal was 0. Then it uses four if statements to control the tuning and octave, two for the tuning mode and two for the octave shifting. The if statement would be evaluated as true every time the current button state and the previous button state were high. For example, if the user pressed the "tuningUp" button, both the current "tuningUp" variable and the "lastTuningUp" variable would be evaluated as high and the first if statement would be true. In this the "tuningMode" variable would be updated with the current tuning mode incremented by 1, before the modulo operator gave the remainder after the division by 3, which ensured that the value remained in the valid range of 0, 1, or 2. With the octaveUp it was the same, just here the modulo operator was 5 since the octave had 5 steps. The decrement of the tuning or octave was similar, here the if statements would be evaluated as true whenever the "tuningDown" and "lastTuningDown" or "octaveDown" and "lastOctaveDown" were both high. For the decrease of the tuning mode, the tuning mode would be updated with the tuning mode incremented by 2 before the modulo operator gave the remainder after the division by 3. And for the decrease of the octave, the "octaveShift" variable was updated with the "octaveShift" variable incremented by 4, before the modulo operator gave the remainder after the division by 5. In each if statement following the update of the variables, the code would call the "updateLEDs()" function to trigger the change of the LEDs. After the if statements, the last button states were updated with the current button states before a short delay was introduced. The usage of four buttons allowed to make quicker adjustments to tuning mode and octave.

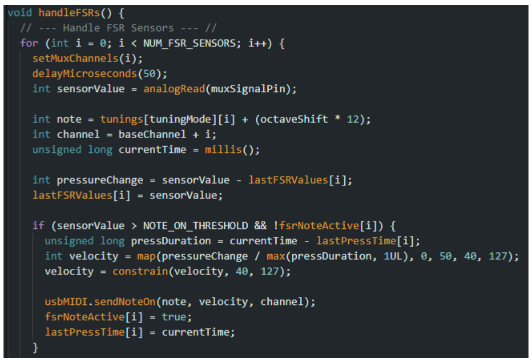


```
void handleButtons() {
  // Handle Tuning and Octave Buttons
  bool tuningUp = digitalRead(tuningButtonUp) == LOW;
  bool tuningDown = digitalRead(tuningButtonDown) == LOW;
  bool octaveUp = digitalRead(octaveButtonUp) == LOW;
  bool octaveDown = digitalRead(octaveButtonDown) == LOW;

  if (tuningUp && lastTuningUp == HIGH) {
    tuningMode = (tuningMode + 1) % 3;
    updateLEDs();
  }
  if (tuningDown && lastTuningDown == HIGH) {
    tuningMode = (tuningMode + 2) % 3; // Decrease
    updateLEDs();
  }
  if (octaveUp && lastOctaveUp == HIGH) {
    octaveShift = (octaveShift + 1) % 5;
    updateLEDs();
  }
  if (octaveDown && lastOctaveDown == HIGH) {
    octaveShift = (octaveShift + 4) % 5; // Decrease
    updateLEDs();
  }

  lastTuningUp = tuningUp;
  lastTuningDown = tuningDown;
  lastOctaveUp = octaveUp;
  lastOctaveDown = octaveDown;

  delay(100);
}
```

Figure 4.23: Handling of the buttons in the final code.

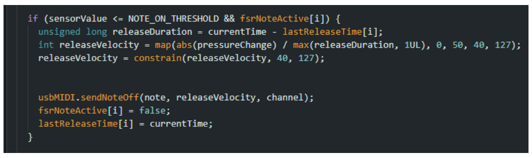
The "handleFSRs()" function included the same lines of code as the first iteration of the first interface (See Fig. 4.24, 4.25, and 4.26).

The "handleRibbon()" function (See Fig. 4.27, 4.28, and 4.29) differed compared to the first iteration of the second interface. It started with reading the ribbon pin and storing the value on a variable. Then in an if-statement, it was evaluated if the value was higher than a threshold or the deadzone. If that was true the width of each zone on the ribbon was calculated with the ribbon maximum minus the ribbon minimum and that divided by the number of zones. Then to identify the zone it took the ribbon value minus the ribbon minimum and that divided by the zone width. Then, this zone was constrained between 0 and the number of maximum zones decremented by 1. Then, this variable was used as the index of the note array. The actual note was set by taking the MIDI number from the note array at the current position, which was calculated just before, and then the octave



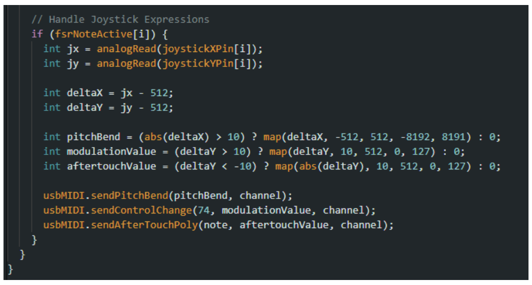
```
void handleFSRs() {
  // --- Handle FSR Sensors --- //
  for (int i = 0; i < NUM_FSR_SENSORS; i++) {
    setMuxChannels(i);
    delayMicroseconds(50);
    int sensorValue = analogRead(muxSignalPin);

    int note = tunings[tuningMode][i] + (octaveShift * 12);
    int channel = baseChannel + i;
    unsigned long currentTime = millis();

    int pressureChange = sensorValue - lastFSRValues[i];
    lastFSRValues[i] = sensorValue;

    if (sensorValue > NOTE_ON_THRESHOLD && !fsrNoteActive[i]) {
      unsigned long pressDuration = currentTime - lastPressTime[i];
      int velocity = map(pressureChange / max(pressDuration, 1UL), 0, 50, 40, 127);
      velocity = constrain(velocity, 40, 127);

      usbMIDI.sendNoteOn(note, velocity, channel);
      fsrNoteActive[i] = true;
      lastPressTime[i] = currentTime;
    }
```

Figure 4.24: Handling of the FSR sensors in the final code.



```
    if (sensorValue <= NOTE_ON_THRESHOLD && fsrNoteActive[i]) {
      unsigned long releaseDuration = currentTime - lastReleaseTime[i];
      int releaseVelocity = map(abs(pressureChange) / max(releaseDuration, 1UL), 0, 50, 40, 127);
      releaseVelocity = constrain(releaseVelocity, 40, 127);


      usbMIDI.sendNoteOff(note, releaseVelocity, channel);
      fsrNoteActive[i] = false;
      lastReleaseTime[i] = currentTime;
    }
```

Figure 4.25: Handling of the FSR sensors in the final code.



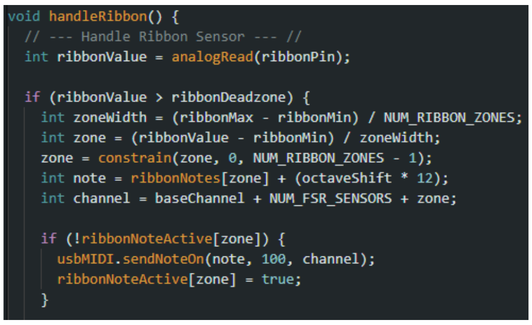
```
    // Handle Joystick Expressions
    if (fsrNoteActive[i]) {
      int jx = analogRead(joystickXPin[i]);
      int jy = analogRead(joystickYPin[i]);

      int deltaX = jx - 512;
      int deltaY = jy - 512;

      int pitchBend = (abs(deltaX) > 10) ? map(deltaX, -512, 512, -8192, 8191) : 0;
      int modulationValue = (deltaY > 10) ? map(deltaY, 10, 512, 0, 127) : 0;
      int aftertouchValue = (deltaY < -10) ? map(abs(deltaY), 10, 512, 0, 127) : 0;

      usbMIDI.sendPitchBend(pitchBend, channel);
      usbMIDI.sendControlChange(74, modulationValue, channel);
      usbMIDI.sendAfterTouchPoly(note, aftertouchValue, channel);
    }
  }
}
```

Figure 4.26: Handling of the FSR sensors in the final code.

shift multiplied by 12 was added. The channel was then set by taking the base channel and adding the number of FSR sensors and the current position to it. Adding the number of FSR sensors to the base channel ensured that the ribbon sensor would not interfere with the channels the first interface used. The next part was another if-statement that evaluated if the current zone or position is currently already active or not. When it was not active, the "usbMIDI.sendNoteOn" function was used with the current note, a velocity of 100, and on the current channel, and the state of the active ribbon note at the current position was updated to true.

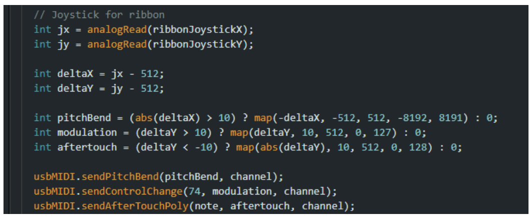
Afterwards, the joystick for the second interface was implemented. The current value of the x- and y-axes was measured. Then an intermediate value was created by centering the joystick by subtracting 512 from the x- and y- value [25]. Then the three effects were implemented. The pitch bend was set on the x-axis, the modulation was set to the upward part of the y-axis, and the aftertouch was set to the downward part of the y-axis. This distribution was inspired by the "MPE-DIY-Device" from Mathias Brüssel [22, 23, 16]. All three effects were implemented similarly. First, the code checked if the previously calculated intermediate value is higher than a given threshold, in this case 10, before the code would map the intermediate value in the possible range of a joystick from -512 to 512 to the range of possible output values. In the case of the pitch bend, the code would test if the absolute value of the intermediate x-axis value was higher than 10. If that was the case, it took the value and on a possible range from -512 to 512 it would map the value to the range -8192 to 8191. If the intermediate value was smaller than 10 the pitch bend would be set to 0. Similarly, the code was checking the y-axis for modulation and aftertouch. When the intermediate y-value was higher than 10 the code would take the value in the range from 10 to 512 and map it to the range from 0 to 127. If not, the modulation would be 0. For the aftertouch, the code would check if the intermediate y-value was less than -10. If true, the code would take the absolute intermediate y-value in the range of 10 to 512 and map it to the range of 0 to 127. If the value was not less than -10, the aftertouch value would be set to 0. Again, the pitch bend was scaled to -8192 to 8191, while the modulation and aftertouch were only scaled to the range of 0 to 127, was due to the fact that the pitch bend naturally uses a 14-bit resolution while modulation and aftertouch remain at the normal 7-bit resolution [25, 2].

The code would then take the built-in usbMIDI function "usbMIDI.sendPitchBend", "usbMIDI.sendControlChange", and "usbMIDI.sendAftertouchPoly" to send the three effects. If the currently read ribbon value was not bigger than the deadzone, the first if would be false, and the code would go to the else part. Here, the code used a for-loop, that iterated from 0 to the number of ribbon zones. In each iteration, if the zone was active, the code would select the note of the zone, by taking the zone as the index of the ribbon note array plus the octaveShift times 12, and the channel as the base channel plus the number of FSR sensors plus the zone. Then it would stop sending the note by sending the "usbMIDI.sendNoteOff" command with the selected note and channel, and the active state of the active zone was set to false.



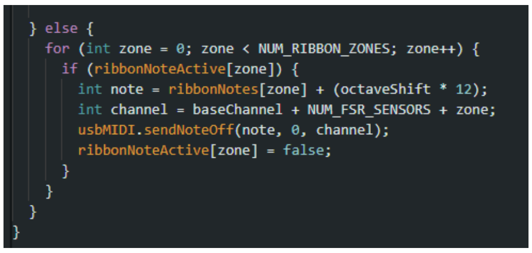
```
void handleRibbon() {
  // --- Handle Ribbon Sensor --- //
  int ribbonValue = analogRead(ribbonPin);

  if (ribbonValue > ribbonDeadzone) {
    int zoneWidth = (ribbonMax - ribbonMin) / NUM_RIBBON_ZONES;
    int zone = (ribbonValue - ribbonMin) / zoneWidth;
    zone = constrain(zone, 0, NUM_RIBBON_ZONES - 1);
    int note = ribbonNotes[zone] + (octaveShift * 12);
    int channel = baseChannel + NUM_FSR_SENSORS + zone;

    if (!ribbonNoteActive[zone]) {
      usbMIDI.sendNoteOn(note, 100, channel);
      ribbonNoteActive[zone] = true;
    }
```

Figure 4.27: Handling of the ribbon sensor / interface in the final code.



```
    // Joystick for ribbon
    int jx = analogRead(ribbonJoystickX);
    int jy = analogRead(ribbonJoystickY);

    int deltaX = jx - 512;
    int deltaY = jy - 512;

    int pitchBend = (abs(deltaX) > 10) ? map(-deltaX, -512, 512, -8192, 8191) : 0;
    int modulation = (deltaY > 10) ? map(deltaY, 10, 512, 0, 127) : 0;
    int aftertouch = (deltaY < -10) ? map(abs(deltaY), 10, 512, 0, 128) : 0;

    usbMIDI.sendPitchBend(pitchBend, channel);
    usbMIDI.sendControlChange(74, modulation, channel);
    usbMIDI.sendAfterTouchPoly(note, aftertouch, channel);
```

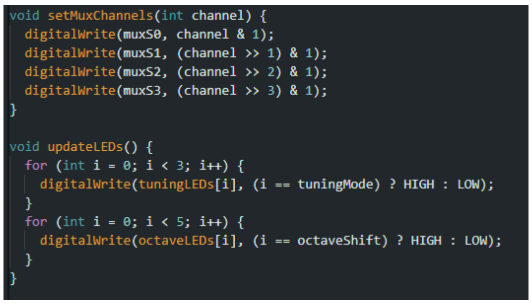
Figure 4.28: Handling of the ribbon sensor / interface in the final code.



```
  } else {
    for (int zone = 0; zone < NUM_RIBBON_ZONES; zone++) {
      if (ribbonNoteActive[zone]) {
        int note = ribbonNotes[zone] + (octaveShift * 12);
        int channel = baseChannel + NUM_FSR_SENSORS + zone;
        usbMIDI.sendNoteOff(note, 0, channel);
        ribbonNoteActive[zone] = false;
      }
    }
  }
}
```

Figure 4.29: Handling of the ribbon sensor / interface in the final code.

Lastly, as the two remaining functions, the function handling the multiplexer channels and the function handling the updating of the LEDs were defined (See Fig. 4.30).



```
void setMuxChannels(int channel) {
  digitalWrite(muxS0, channel & 1);
  digitalWrite(muxS1, (channel >> 1) & 1);
  digitalWrite(muxS2, (channel >> 2) & 1);
  digitalWrite(muxS3, (channel >> 3) & 1);
}

void updateLEDs() {
  for (int i = 0; i < 3; i++) {
    digitalWrite(tuningLEDs[i], (i == tuningMode) ? HIGH : LOW);
  }
  for (int i = 0; i < 5; i++) {
    digitalWrite(octaveLEDs[i], (i == octaveShift) ? HIGH : LOW);
  }
}
```

Figure 4.30: Handling of the multiplexer and LEDs in the final code.

4.4 Design & Ergonomics

The final prototype was designed to have a case, looking like a sandwich of two wooden pieces. This sandwich design allowed the main electronics to be stored between the wooden pieces while the input sensors were placed on the top of one of the wooden pieces. Wood was chosen as the material, since it offered good stability and added a bit more quality to the prototype. It was decided to design it in a way that minimized the physical footprint of the device. On the upper wooden piece, the interfaces were oriented as follows. The ribbon was fixed at the upper edge. In the lower right corner, the separate joystick for the ribbon interface was positioned. Left to it near the lower edge the four buttons were placed. The two buttons increasing the tuning and octave were above the two buttons decreasing the tuning and octave. Again, to the left of the buttons, the first interface was positioned. The four FSRs were moved slightly more to the center of the upper wooden piece. Above each of the FSR sensors the corresponding sliding joystick was positioned with a soft woolen disk to make pushing onto the FSR sensors with the joysticks much more enjoyable and also to make the joysticks sit flat on top of the FSR sensors. The negative aspect of this decision was that the soft disk made the triggering of the notes a bit more difficult. The spacing between the four FSR sensors and joysticks and the slightly different position of them was to make it more ergonomic for the user since the spacing was oriented at the natural spacing between the four main fingers of the left hand. On the left edge of the upper wooden plank, the LED indicators were fixed. The ergonomic positioning, based on the natural position of the fingers of a human hand was inspired by the Thunder from Don Buchla [27].

The placement of the single components or the two interfaces was made this way so that the user had the opportunity to play both interfaces at the same time when using both hands. The right hand could touch the ribbon and control the separate joystick, while the left hand could be placed on the four joysticks that were mounted on top of the FSR sensors

(Interface 1). The four buttons were placed in the middle so that they could be reached with both thumbs.

In the current state of the device or prototype the wires were not hidden beneath some cover. For the main electronics and connection with the multiplexer and the Teensy, a separate PCB board was made. On this board the Teensy could be put into two rails of female headed connectors. The 3V and Ground of the Teensy was extended by using two rows of male headers were the other components were able to be connected to. The multiplexer had a similar setup with two rows of female headed connectors for each pin of the multiplexer. The joysticks had a separate area with pins. The 3V line and ground line for each joystick were combined, and only the x and y signal pins were separated. From these separate pins, the different parts were connected to the Teensy with cables. The signal of the multiplexer was first fed into a voltage divider before it was connected to the Teensy. The ribbon had a separate voltage divider directly soldered to the pin connection. The design of the voltage divider was inspired by the voltage divider from KontinuumLab [28]. Next to the female headed connector rows of the Teensy two separate lines of male headed connectors, one on each side of the Teensy, were mounted to have the possibility to remove the Teensy without having to unplug all the wires from it and also to have a fall back solution if some of the extra male headed connectors had a loose connection.

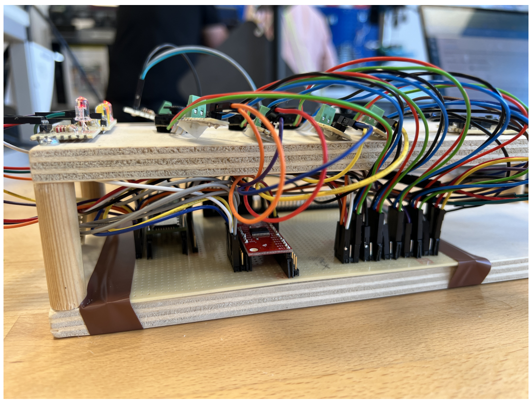


Figure 4.31: Multiplexer and Joystick connections.

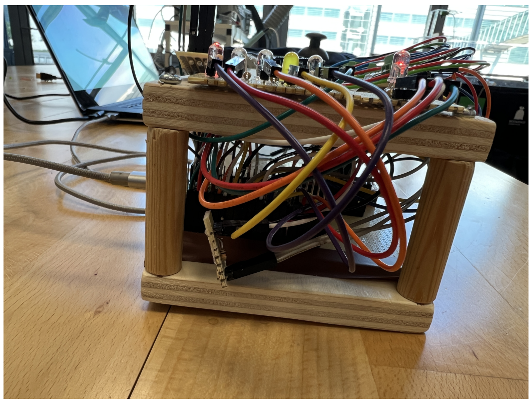


Figure 4.32: View from the side.

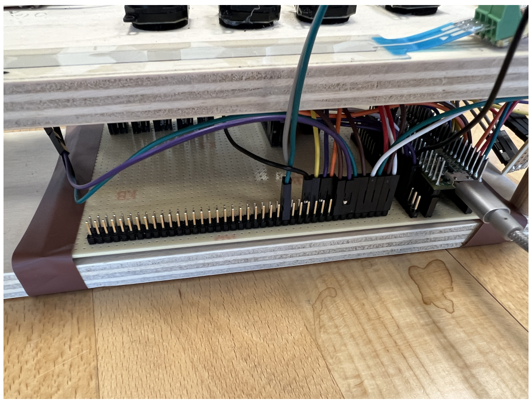


Figure 4.33: Vcc and GND rail.

Chapter 5

Evaluation

5.1 Evaluation Setup

The evaluation consisted of two steps. The first step was a trial period, in which the test participants had time to explore the prototype and try out how the prototype functioned. The second part was an online survey with several questions to different parts of the prototype and about their experience with the prototype. The prototype was briefly introduced to all test participants at the beginning to explain what the prototype was and how it can be used. The test participants were mostly students in medialogy and some students in SMC. They all had mostly some prior experience with MIDI controllers, and therefore were perfect fits for the target group of the device.

The questions were mostly rating questions where different questions or statements were rated on a 7-point Likert scale from most negative (1) over neutral (4) to most positive (7). 1 was the most negative answer and 7 the most positive answer. There were also some open questions where participants were asked to write down their answer.

The survey or online questionnaire was done on Xact. First, the participants were asked about the playability and general usability of the prototype, like, for example, how easy it was to play the controller or to understand how the controller worked or how responsive the controller was to the input. The next sections were musical expression, performance and feedback, aesthetics and design, and the overall satisfaction. The last questions were open-ended questions, like, if the participants had other comments on the prototype. For the playability and general usability, the participants were asked how responsive the controller was to their touch input, how easy it was to play the controller or to understand how the controller worked, how comfortable was the physical layout of the interface, how confident did you feel using the controller after a short period of time, how intuitive did the controls feel, and how enjoyable was it to play the controller. For the musical expression, the participants were asked how expressive the FSR + joystick combo felt when performing, how responsive the ribbon interface was, and how well they could control dynamics, like velocity or aftertouch. For the performance and feedback, the questions were, did the controller responded quickly to your input, were the LED indicators helpful in showing tuning and octave settings, and did you experience any technical glitches or unexpected behaviour. For the aesthetics and design section, the questions were How appealing do you find the controller's design, and How would you rate the perceived quality of the controller's materials and construction. Finally, for the overall satisfaction, the participants were asked how satisfied they were overall. For the open-ended questions, the particiants were asked about what stood out the most about the controller, like what did they like the most, what improvements they would suggest, like what would they improve or change, if they can think of any other useful addition that comes with having multiple interfaces, if they have any additional feedback or thoughts, and if they would consider using this controller in their music production or performance. The participants were also asked if having two interfaces is a good addition or not.

Lastly, it was also asked if the participants had any musical experience and also if they had any experience with MIDI controllers. The questions are also visible in Appendix A.1

5.2 Evaluation Results

The results of the evaluation were the following. More than half of the participants, 56%, felt that the controller was very responsive, so that it responded almost instantly. In general, the answers to the first question ranged from very unresponsive to very responsive. 22% responded with somewhat unresponsive, and the overall average was that it was fairly responsive (See Figures 5.1 and 5.2).

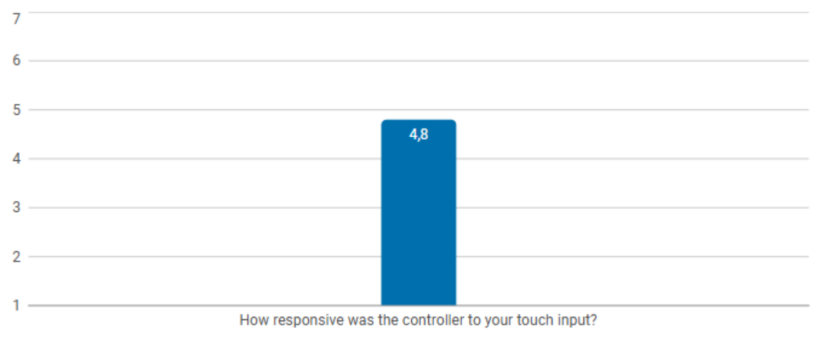


Figure 5.1: Average of how responsive the controller felt to the touch input (1 (most negative) to 7 (most positive)).

Most of the participants found the controller moderately easy to use or to understand how the controller worked. Here, the answers ranged from very difficult to very easy. 22% replied that it was somewhat difficult, it was usable but required noticeable adaptation or effort, and certain features / controls felt awkward. The overall average was the neutral answer, that it was moderately easy to understand and use (See Figures 5.3 and 5.4).

The physical layout was judged mainly to be neutrally comfortable. The responses

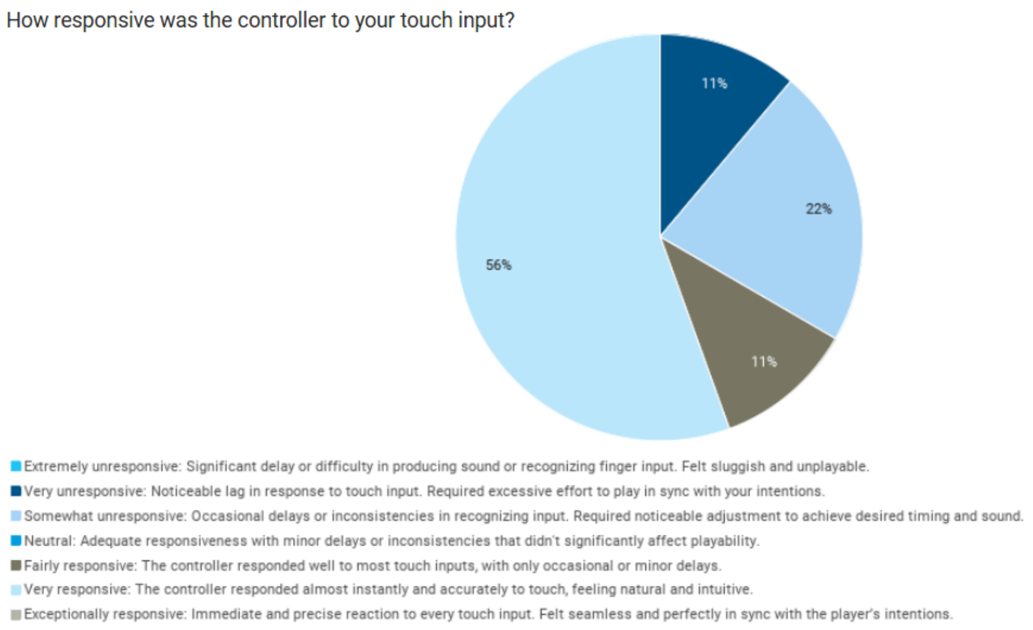


Figure 5.2: Spread of the answers about the responsivness of the controller to touch inputs.

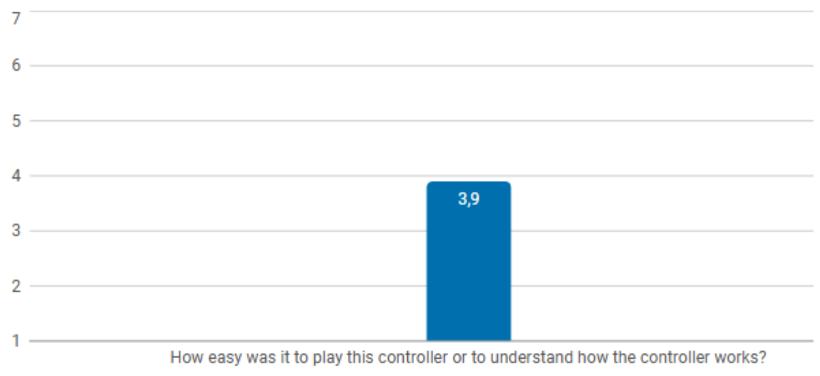


Figure 5.3: Average of how easy it was to understand and play the controller (1 (most negative) to 7 (most positive)).

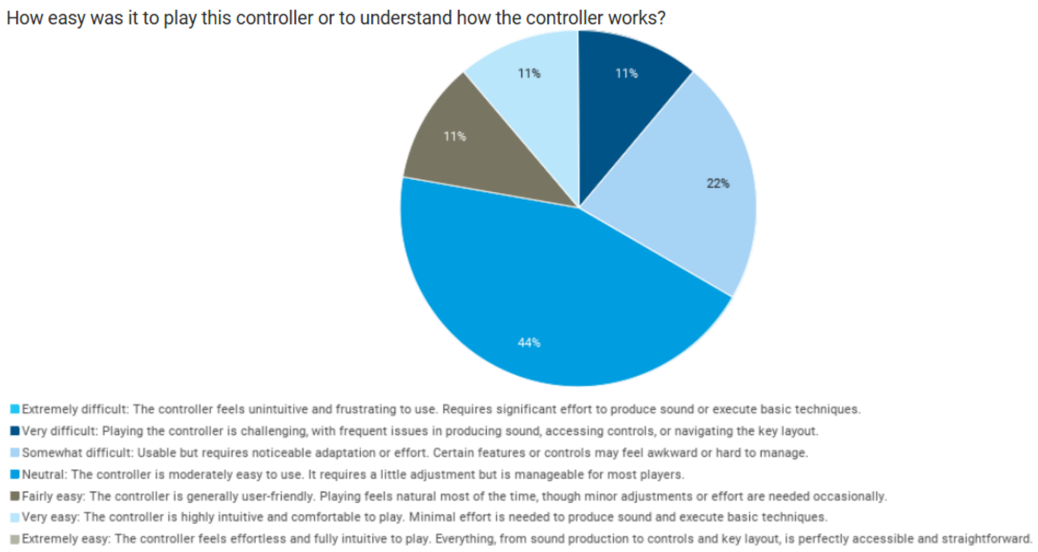


Figure 5.4: Spread of the answers about how easy it was to understand and play the controller.

ranged from somewhat uncomfortable to extremely comfortable (See Figures 5.5 and 5.6).

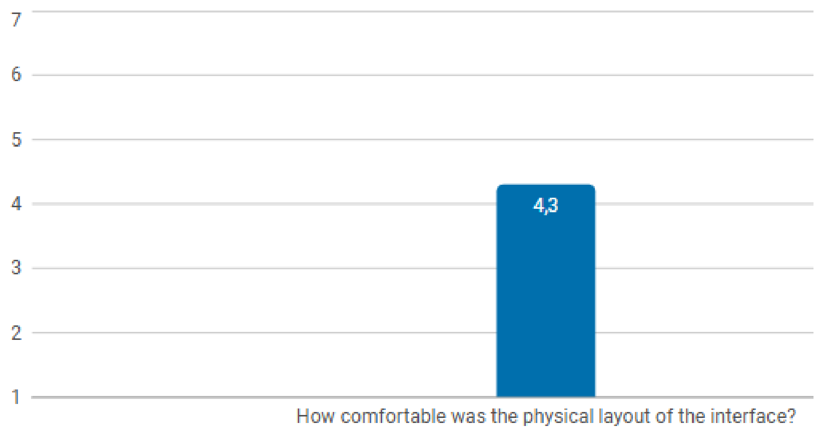


Figure 5.5: Average of how comfortable the layout felt (1 (most negative) to 7 (most positive)).

More than two thirds of the participants felt somewhat confident in using the controller after a short period of time, and some even felt very confident (See Figures 5.7 and 5.8).

The controls felt mostly somewhat intuitive, but the answers ranged from somewhat unintuitive to extremely intuitive (See Figures 5.9 and 5.10).

For the question of how enjoyable it was to play the controller, the majority of the participants, 44%, found it generally enjoyable, with minor drawbacks. The responses ranged from slightly enjoyable, but with significant issues to exceptionally enjoyable, a pleasure to play (See Figures 5.11 and 5.12).

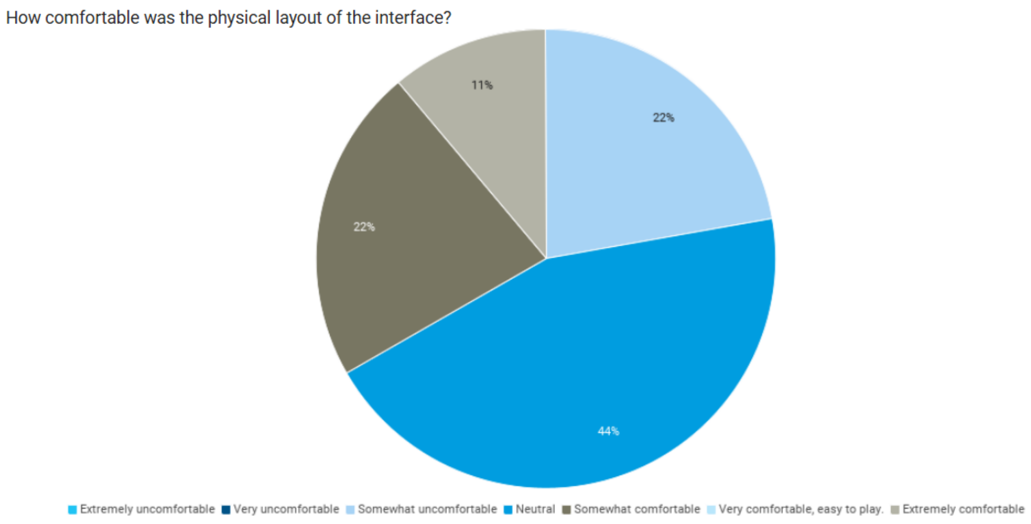


Figure 5.6: Spread of the answers about how comfortable the layout felt.

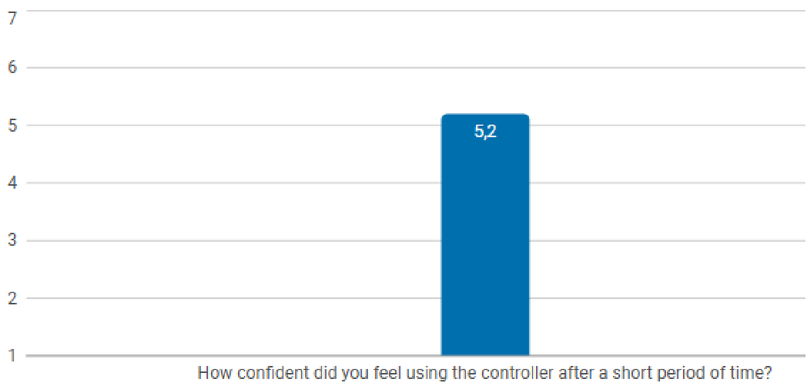


Figure 5.7: Average of how confident the participants felt after a short while (1 (most negative) to 7 (most positive)).

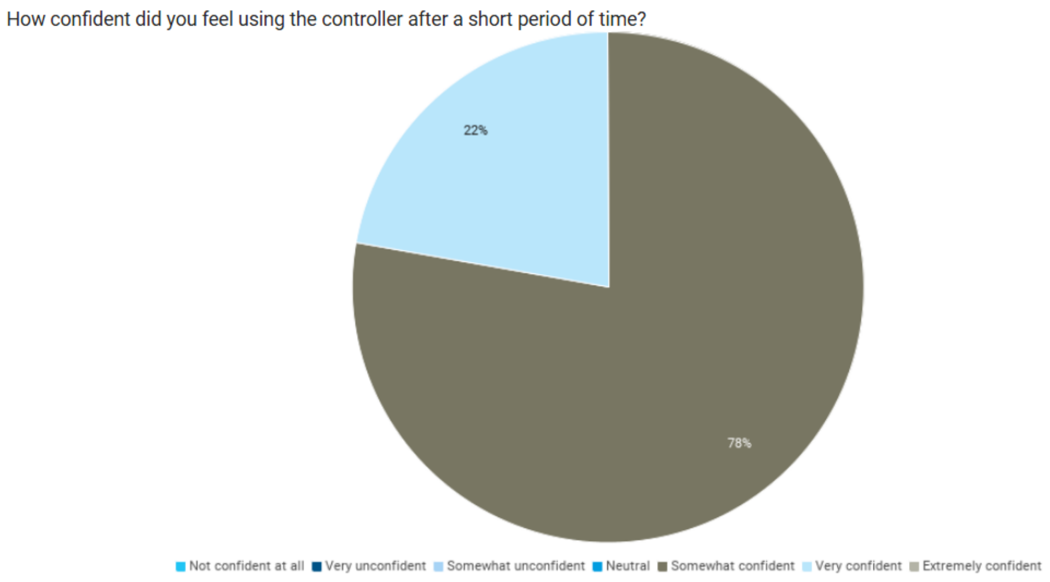


Figure 5.8: Spread of the answers about how confident the participants felt.

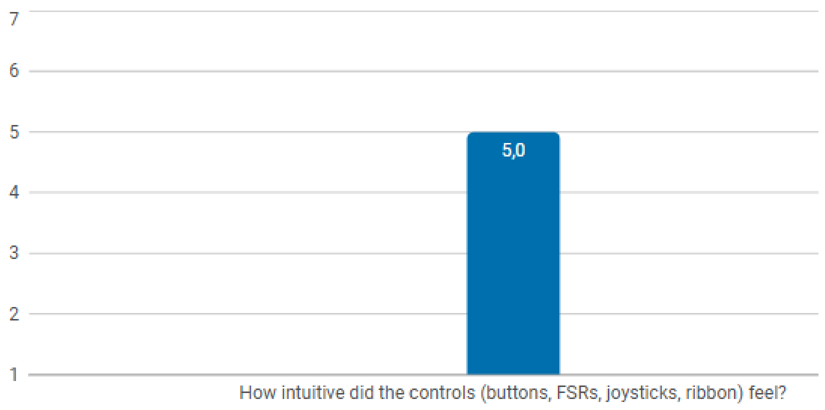


Figure 5.9: Average of how intuitive the controls felt (1 (most negative) to 7 (most positive)).

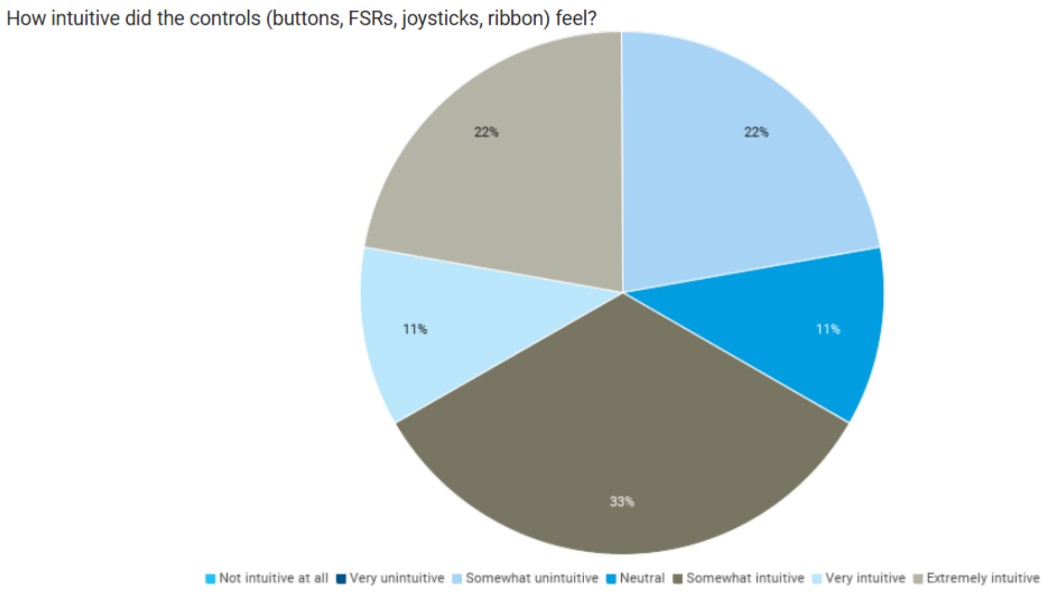


Figure 5.10: Spread of the answers about how intuitive the controls felt.

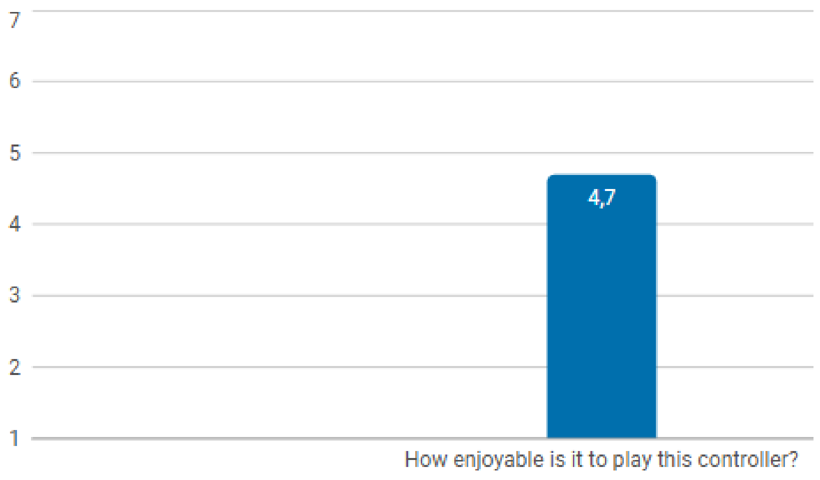


Figure 5.11: Average of how enjoyable it was to play the controller (1 (most negative) to 7 (most positive)).

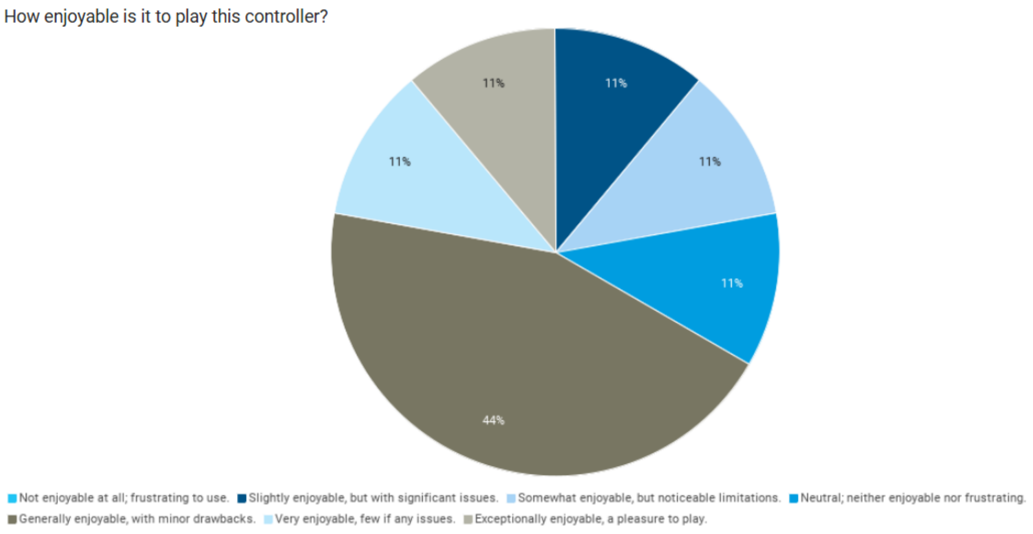


Figure 5.12: Spread of the answers about how enjoyable it was to play the controller.

Interface 1, the FSR and Joystick combo, felt for more than two thirds of the participants somewhat expressive, some found it to be somewhat inexpressive, and one even found it very inexpressive (See Figures 5.13 and 5.14).

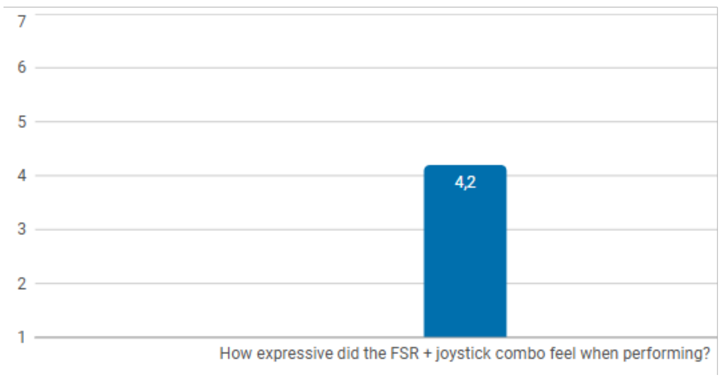


Figure 5.13: Average of how expressive the first interface, the FSR and Joystick combo, felt (1 (most negative) to 7 (most positive)).

Interface 2, the ribbon sensor, was on average rated as somewhat responsive. 22% said it was somewhat unresponsive, another 22% said it was somewhat responsive, another 22% replied with very responsive, and 33% replied with extremely responsive (See Figures 5.15 and 5.16).

On average, the dynamics were neutrally well controllable. The responses ranged from very poorly controllable to very well controllable. Most of the participants found it to be

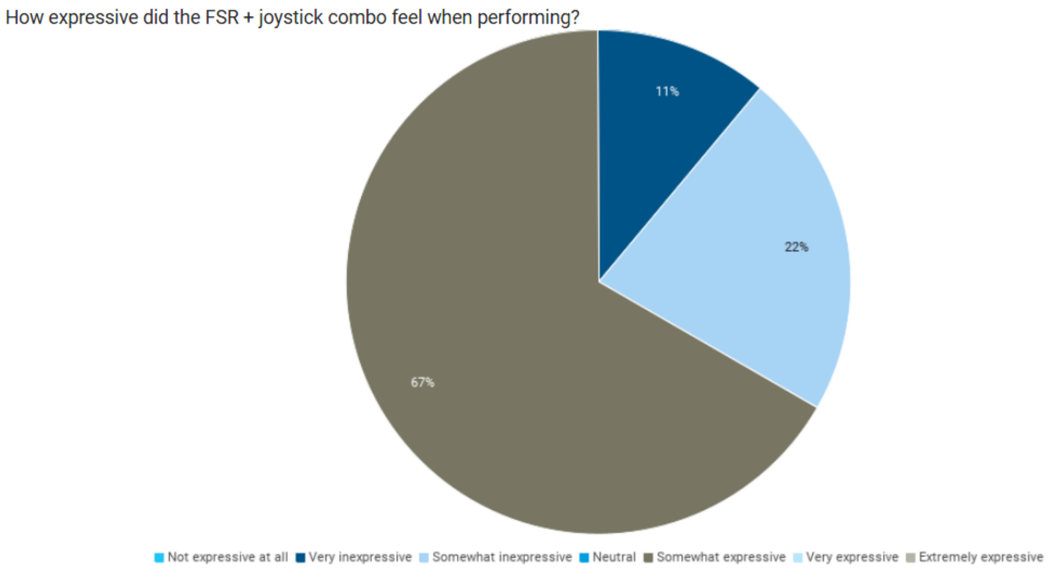


Figure 5.14: Spread of the answers about how expressive interface 1, the FSR and Joystick combo felt.

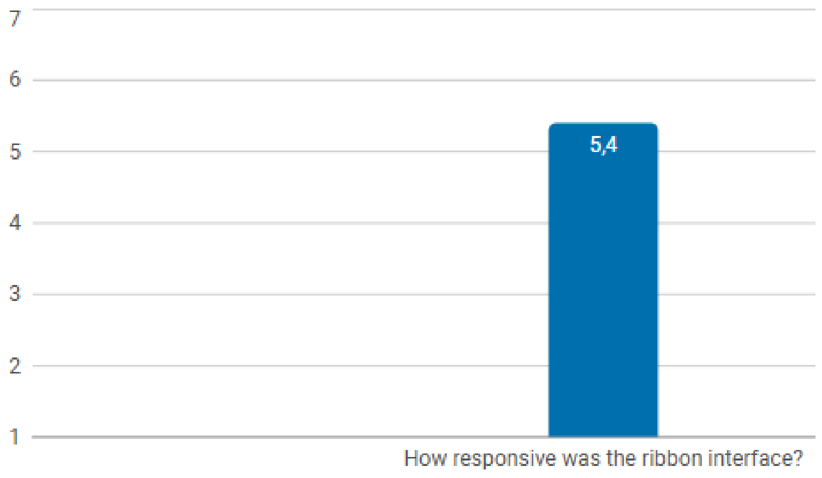


Figure 5.15: Average of how responsive the second interface was (1 (most negative) to 7 (most positive)).

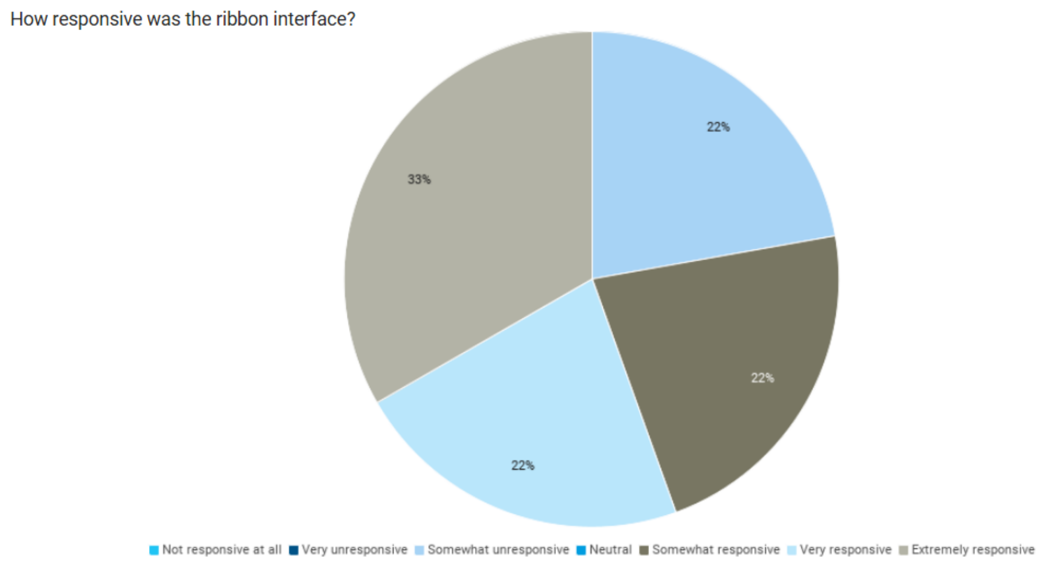


Figure 5.16: Spread of the answers about how responsive the second interface was.

somewhat well controllable (See Figures 5.17 and 5.18).

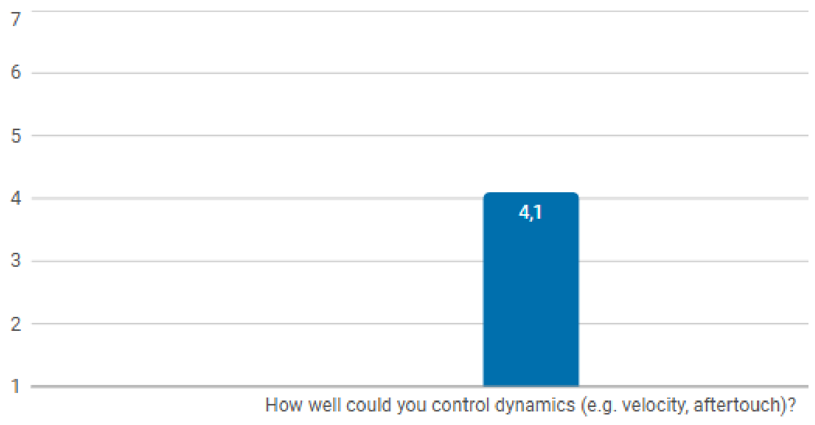


Figure 5.17: Average of how well the dynamics could be controlled (1 (most negative) to 7 (most positive)).

The answers to whether the controller responded quickly to the input ranged from somewhat slow to extremely fast, and most answered that it responded extremely fast (See Figures 5.19 and 5.20).

Next up was the question about the helpfulness of the LED indicators. Most of the participants found the LED indicators for the octave and tuning mode setting to be extremely helpful. The second most picked answers were that it was somewhat helpful and very helpful (See Figures 5.21 and 5.22).

Most of the participants did not encounter any technical glitches or unexpected behaviour while trying out the controller.

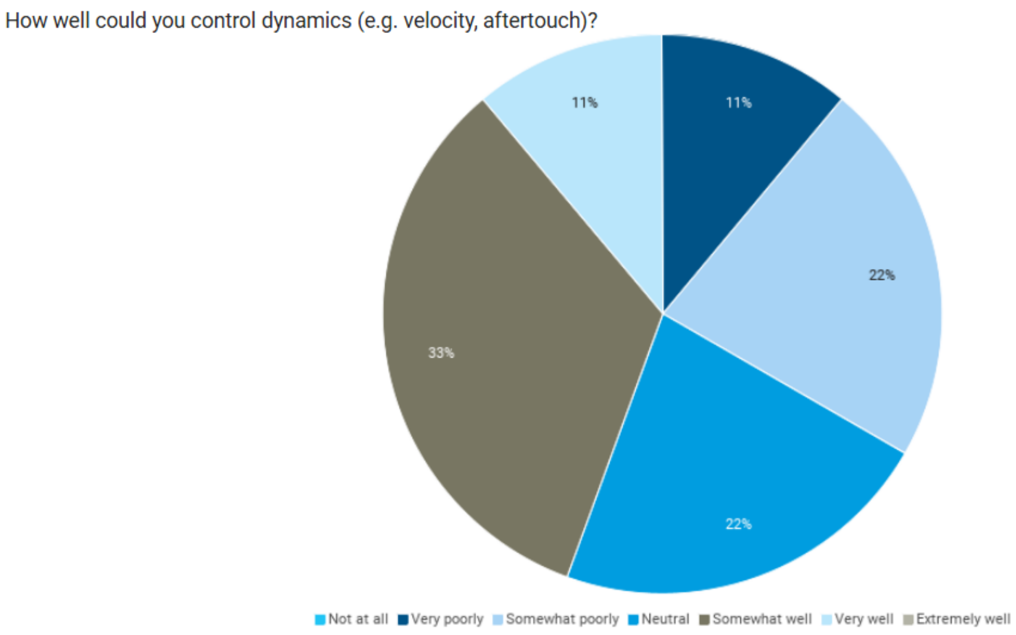


Figure 5.18: Spread of the answers about how well the dynamics could be controlled.

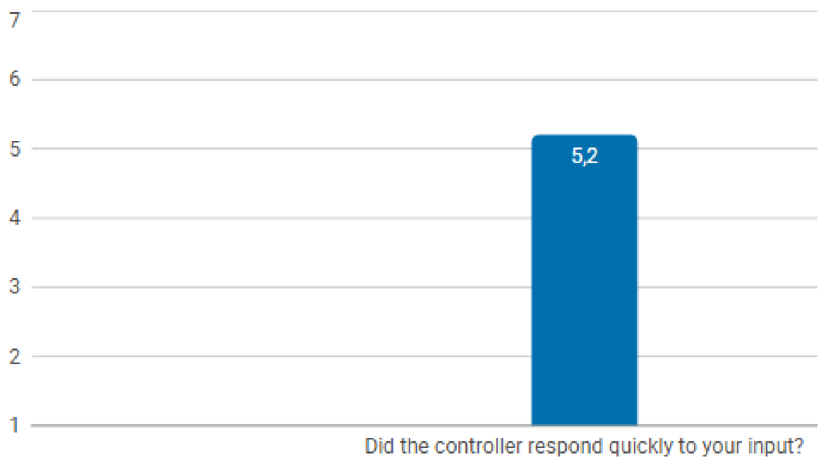


Figure 5.19: Average of how quickly the controller reacted (1 (most negative) to 7 (most positive)).

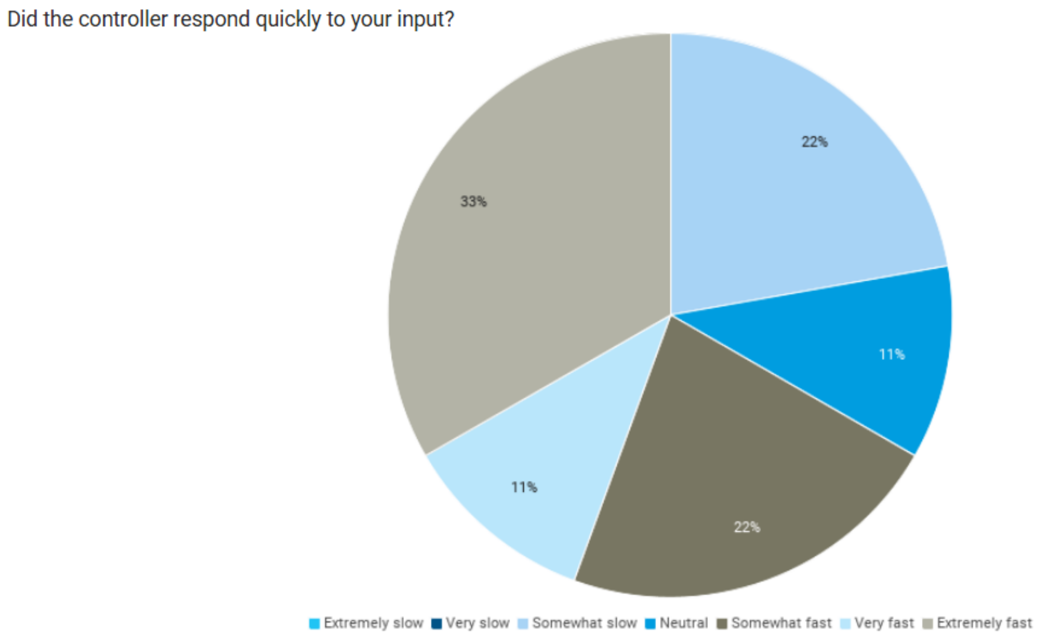


Figure 5.20: Spread of the answers about how quickly the controller reacted.

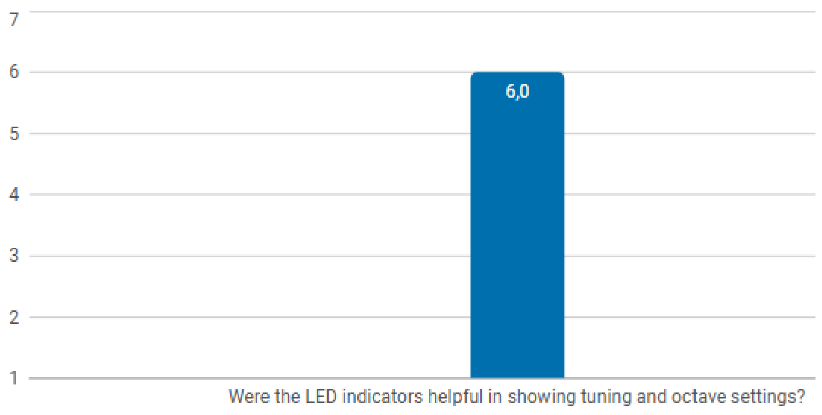


Figure 5.21: Average of how helpful the LED indicators were (1 (most negative) to 7 (most positive)).

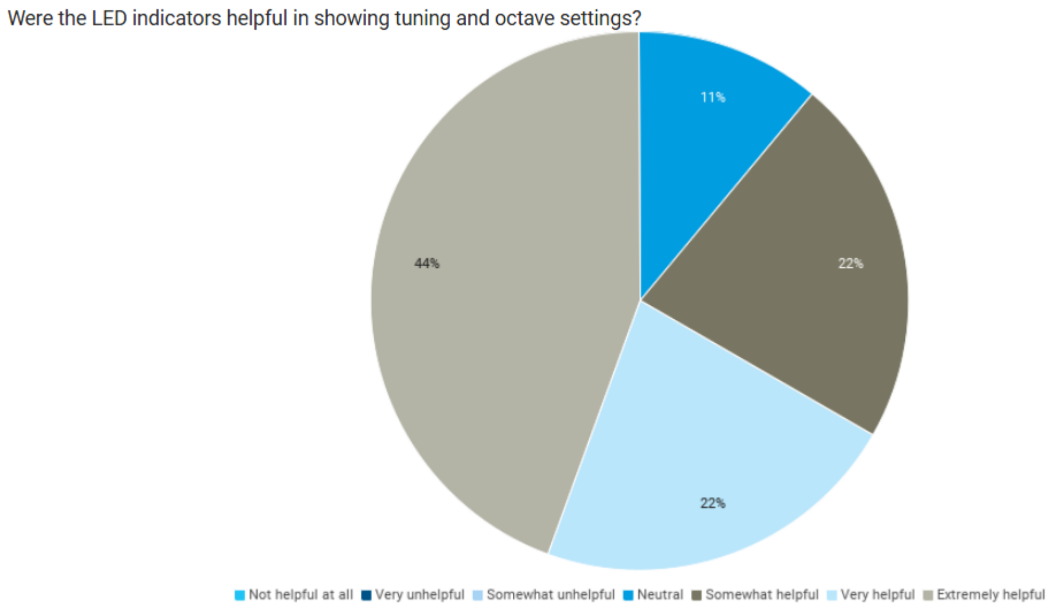


Figure 5.22: Spread of the answers about how helpful the LED indicators were.

For the questions regarding the aesthetics and design of the prototype, the following results were documented. The design was rated as mostly somewhat attractive. The responses ranged from below average design, but acceptable, to very stylish design (See Figures 5.23 and 5.24).

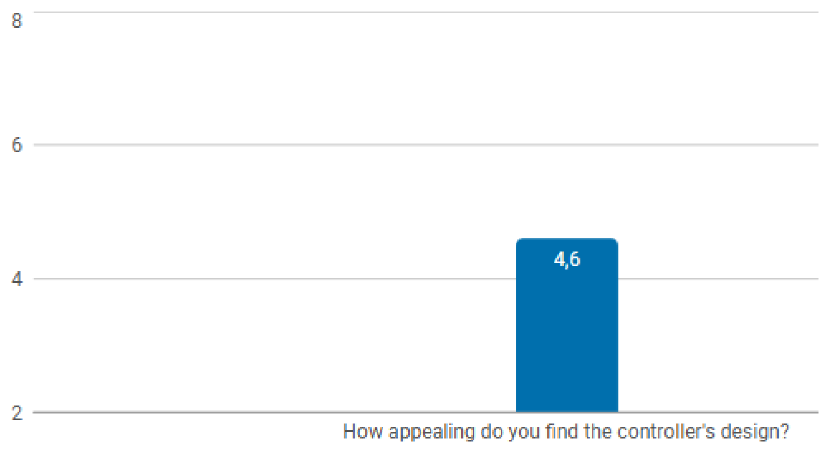


Figure 5.23: Average of how well the design was rated (1 (most negative) to 7 (most positive)).

The perceived quality was rated on average as acceptable. The controller was decently constructed without glaring issues. Many participants also found it as below average, the build quality was passable but uninspiring, and the materials and construction seemed adequate. Some participants found it also good or even very good (See Figures 5.25 and

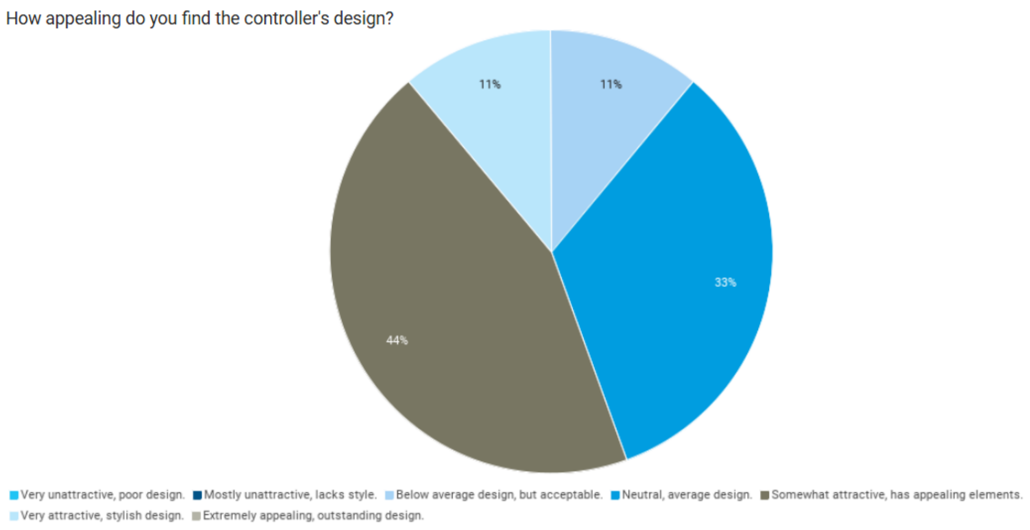


Figure 5.24: Spread of the answers about how well the design was rated.

5.26).

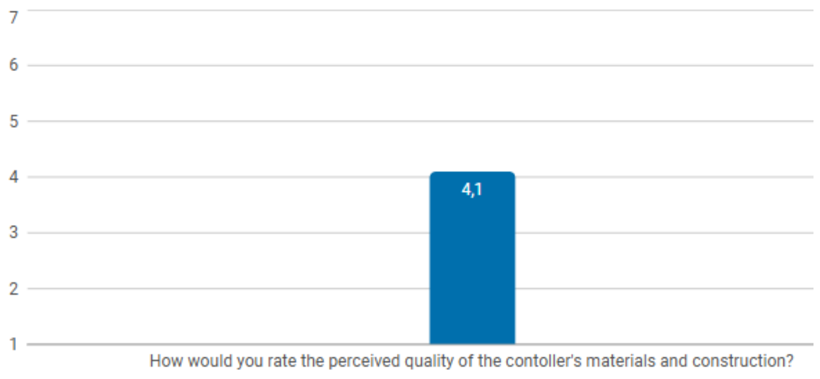


Figure 5.25: Average of how well the quality of the materials and construction was rated (1 (most negative) to 7 (most positive)).

Regarding overall satisfaction, the overall satisfaction was assessed on average as somewhat satisfied, a generally positive experience. Many participants also answered to be very satisfied (See Figures 5.27 and 5.28).

What the participants liked most or what stood out the most to them was the inclusion of octaves and different tunings to extend the range of possible sounds, the ribbon sensor in general, the smoothness of operating the device, the per-note control, the combination of multiple interfaces (the ribbon interface and FSR and Joysticks combo), and the organic / dynamic feel. The various possibilities for sounds were inviting for exploration.

As improvements for the next iterations of the prototype, mainly the reduction of force needed to press the FSR sensors beneath the joysticks was mentioned. Some mentioned

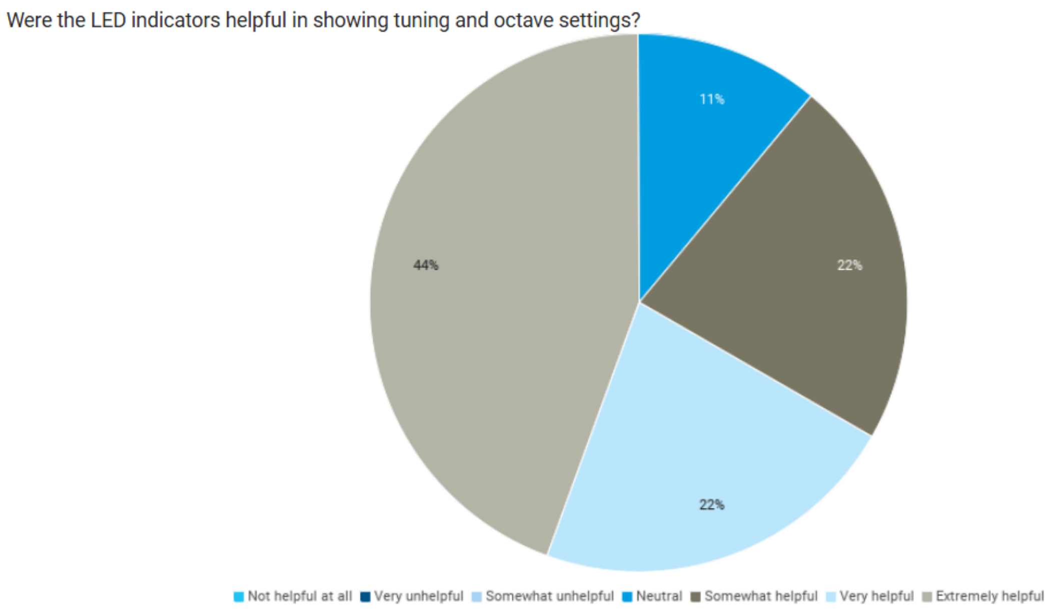


Figure 5.26: Spread of the answers about how well the quality of the materials and construction was rated.

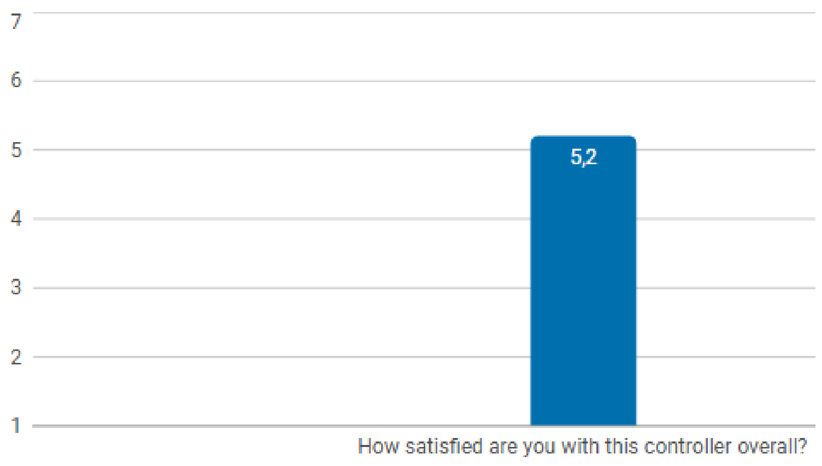


Figure 5.27: Average of how satisfied the participants were (1 (most negative) to 7 (most positive)).

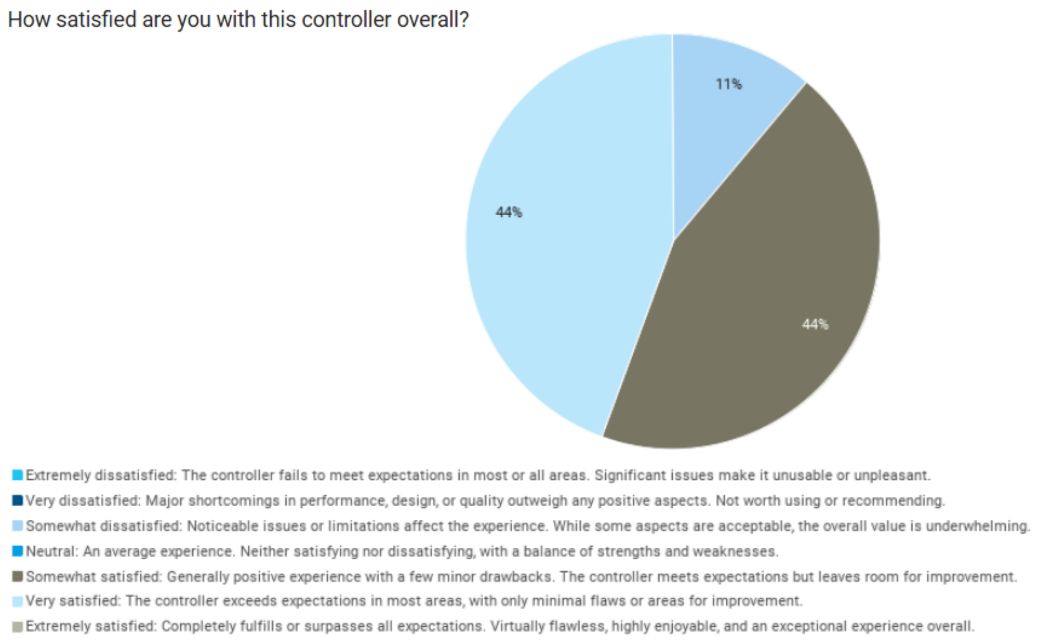


Figure 5.28: Spread of the answers about how satisfied the participants were.

that interface 1 (FSR and Joystick combo) had a noticeable difference in responsiveness compared to interface 2 (ribbon sensor). The ergonomics was also mentioned as a part that would need some improvement, since the device had not much space to rest the hand on. A wish to include a playback functionality, that would hold the active note, was also mentioned. Some of the participants wished that the sound could have some more layers in the next iteration, and one participant mentioned that the ribbon sensor could be replace or the functionality behind it could be changed to allow for multitouch.

All the participants found it a good addition to have multiple, in this case two, interfaces. When asked if the participants could think of any other useful addition that comes with having multiple interfaces, the option for more expressive sounds was mentioned.

Other feedback that was collected was that the joysticks were ergonomically placed and was mentioned as very positive. It was mentioned that the click on the joystick for the second interface could have been used to add more functionality. Adding FM synthesis was mentioned where the joysticks would control different parameters of the synth.

Most of the participants noted that they would use the device in their own music production. Most of the participants had several years of musical experience and some experience with MIDI controllers (See Figures 5.29, 5.30, and 5.31).

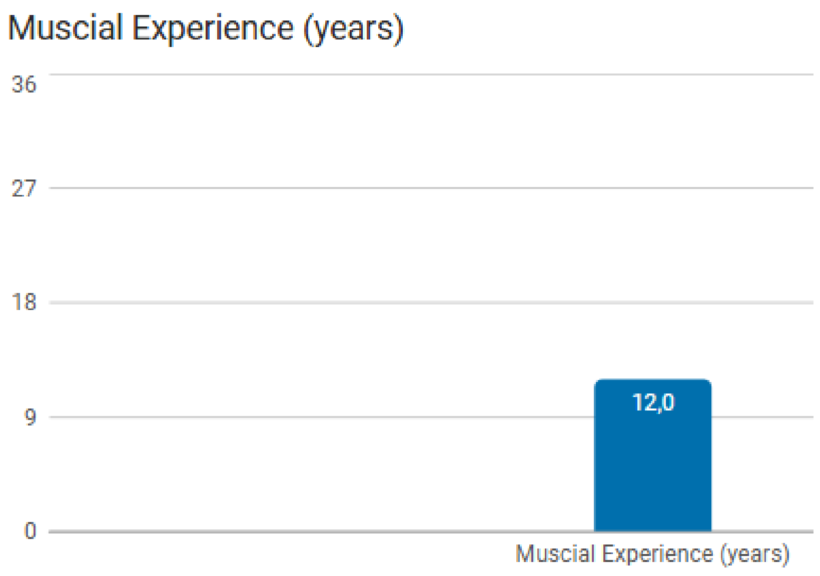


Figure 5.29: Average musical experience in years.

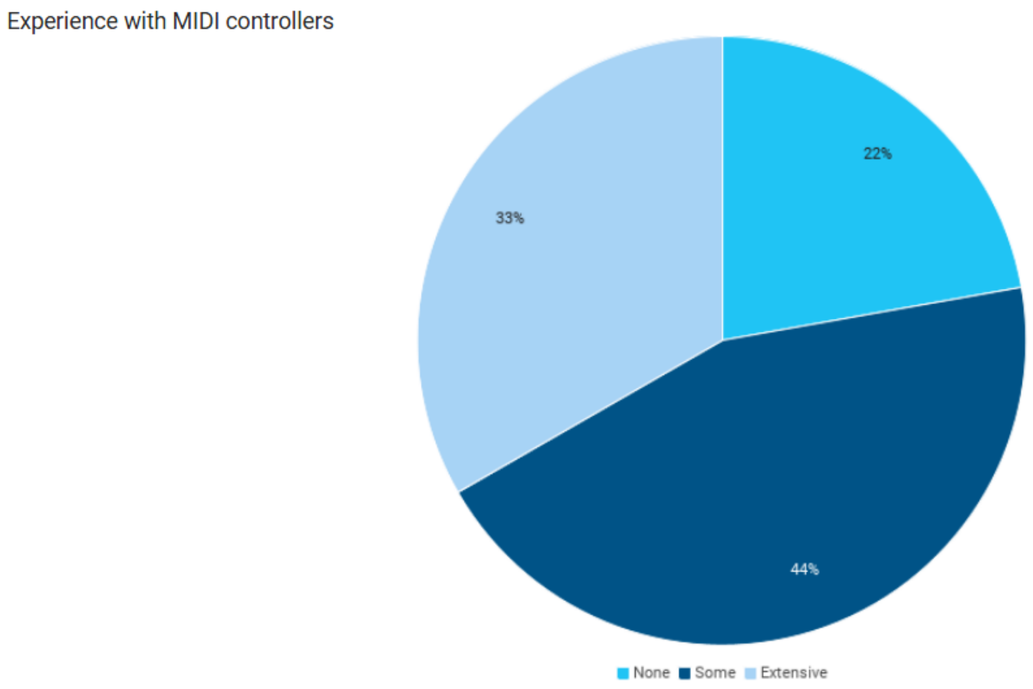


Figure 5.30: Spread of the answers about how experienced the participants were with MIDI controllers.

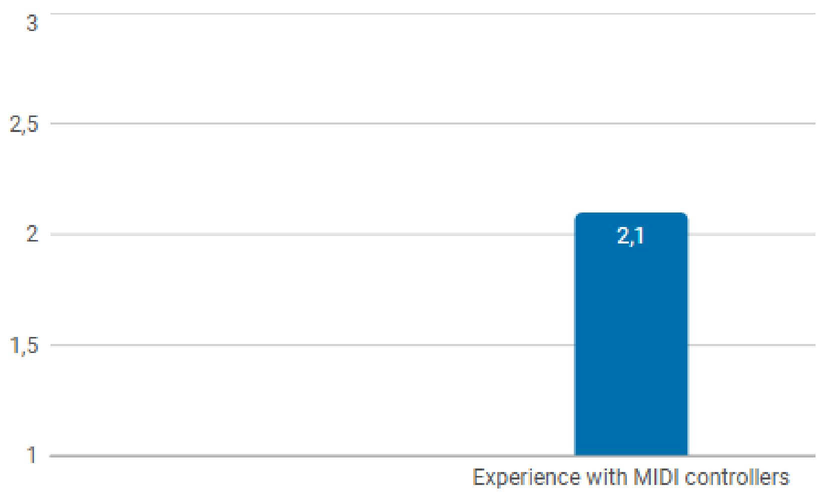


Figure 5.31: Average of how experienced the participants were with MIDI controllers.

Chapter 6

Conclusion

6.1 Future Work

The presented prototype fulfilled all the requirements set out at the beginning. It is a combination of two interfaces, and it had multiple effects. Still, some points of it would benefit from improvements in the future. These were either mentioned by some test participants or discovered during the process of creating the device. For one, better cable management would be beneficial in improving the robustness of the device and making it more usable. Better cable management would also make it nicer to look at. Secondly, making interface 1 (FSR and Joysticks combo) easier to use and more responsive. The reason for the increased difficulty of triggering and holding a sound with the first interface was the use of relatively soft damping discs between the FSR sensors and the joysticks. These damping discs were needed to make the joysticks fit straight on the FSR sensors, but they also negated a lot of the force the user applied onto the joysticks and FSR sensors. Using discs made out of a slightly stiffer material, this issue should be corrected. Third, it would be beneficial to increase the size of the device or think more about the placement of the different parts to allow more space for the hands to rest. For better design and easier understanding, it would be beneficial to put more markings on the front plate of the device that show the different options that the different parts can do.

In the future, it could also be extended to use MIDI 2.0, but since MIDI 2.0 is still not fairly well supported, it was decided to stay for the moment on MIDI Polyphonic Expression.

6.2 Conclusion

The goal of the project was to create an MIDI controller that has extended capabilities by adopting MIDI Polyphonic Expression, and that utilized different interaction interfaces. This was done to offer a different approach to commonly used MIDI Polyphonic Expression controllers, and to help the design of future controllers. This project concluded that it was possible to create a MIDI controller with multiple interfaces that also had the possibility of having different effects. From a musical perspective, it fulfilled the goal of creating a MIDI controller that offered multiple interfaces and increased the expressivity by using MIDI Polyphonic Expression over normal MIDI 1.0. From an academic point of view, it showed the potential that a MIDI Polyphonic Expression controller can have, and offered a new inspiration for further development.

Bibliography

[1] The MIDI Association. *MIDI 1.0*. Apr. 3, 2024. URL: https://midi.org/midi-1-0. (accessed: 05.02.2025).

[2] Julian Hamelberg. "Applications and Implications of MIDI 2.0". MA thesis. Massachusetts Institute of Technology, 2023. URL: https://hdl.handle.net/1721.1/151396.

[3] Matthew Wright. "A Comparison of MIDI and ZIPI". In: *Computer Music Journal* 18.4 (1994), pp. 86–91. ISSN: 01489267, 15315169. URL: http://www.jstor.org/stable/3681361 (visited on 05/21/2025).

[4] Laura's MIDI Heaven. *When was MIDI invented?* URL: https://www.laurasmidiheaven.com/when-was-midi-invented/. (accessed: 22.05.2025).

[5] Natalie Robehmed and Ryan Gaston. *A brief history of MIDI*. Nov. 2019. URL: https://www.perfectcircuit.com/signal/history-of-midi. (accessed: 22.05.2025).

[6] Wikipedia contributors. "MLAN". In: Apr. 2023. URL: https://en.wikipedia.org/wiki/MLAN. (accessed: 22.05.2025).

[7] Paul Wiffen. *An Introduction to MLAN: Part 1*. Aug. 2000. URL: https://www.soundonsound.com/techniques/introduction-mlan-part-1. (accessed: 22.05.2025).

[8] FranklyFlawless. *List of MPE Hardware Controllers (and More)*. Jan. 2025. URL: https://www.kvraudio.com/forum/viewtopic.php?t=594460. (accessed: 03.02.2025).

[9] The MIDI Association. *MPE: MIDI Polyphonic Expression*. Feb. 14, 2024. URL: https://midi.org/mpe-midi-polyphonic-expression. (accessed: 05.02.2025).

[10] The MIDI Association. *MIDI Polyphonic Expression (MPE) specification adopted! MIDI Manufacturers Association (MMA) Adopts New MIDI Polyphonic Expression (MPE) Enhancement to the MIDI Specification*. Aug. 20, 2024. URL: https://midi.org/midi-polyphonic-expression-mpe-specification-adopted. (accessed: 05.02.2025).

[11] Rory Dow. *The ABC of MPE*. June 2021. URL: https://www.soundonsound.com/sound-advice/mpe-midi-polyphonic-expression. (accessed: 16.05.2025).

[12] Erin Barra. *MPE: MIDI Polyphonic Expression explained*. June 2020. URL: https://www.izotope.com/en/learn/midi-polyphonic-expression-explained.html. (accessed: 16.05.2025).

[13] Applied Acoustics Systems DVM Inc. *AAS Multiphonics CV-3 Manual § MPE Implementation Notes*. Version 3.0.0. URL: `https://www.applied-acoustics.com/multiphonics-cv-3/manual/90-mpe-implementation-notes/`. (accessed: 16.05.2025).

[14] Computer Music. *MPE explained using the ROLI Seaboard Block and Equator synth*. Apr. 2020. URL: `https://www.musicradar.com/how-to/mpe-explained-using-the-roli-seaboard-block-and-equator-synth`. (accessed: 16.05.2025).

[15] Ryan Gaston. *What is MPE?* Nov. 2021. URL: `https://www.perfectcircuit.com/signal/what-is-mpe-midi`. (accessed: 16.05.2025).

[16] Mathias [Visuelle-Musik] Brüssel. *MPE-DIY-Device-documentation-20190728*. July 2019. URL: `https://github.com/Visuelle-Musik/MPE-DIY-Device/blob/master/MPE-DIY-Device-documentation-20190728.pdf`. (accessed: 11.03.2025).

[17] M5Stack Technology Co., Ltd. *Unit Joystick2*. 2025. URL: `https://docs.m5stack.com/en/unit/Unit-JoyStick2`. (accessed: 03.02.2025).

[18] Adafruit Industries. *Analog 2-axis Thumb Joystick with Select Button + Breakout Board*. URL: `https://www.adafruit.com/product/512`. (accessed: 11.03.2025).

[19] SparkFun Electronics. *Thumb Slide Joystick*. URL: `https://www.sparkfun.com/thumb-slide-joystick.html`. (accessed: 10.04.2025).

[20] KontinuumLAB. *DIY Pad instrument with pressure sensitive analog soft buttons: The Virus Pad*. Apr. 2020. URL: `https://www.youtube.com/watch?v=X-mxTYgY91Q`. (accessed: 11.03.2025).

[21] KontinuumLab. *KontrolFreak/VirusPad_05_Slider/VirusPad_05_Slider.ino at master · KontinuumLab/KontrolFreak*. Apr. 2020. URL: `https://github.com/KontinuumLab/KontrolFreak/blob/master/VirusPad_05_Slider/VirusPad_05_Slider.ino`. (accessed: 11.03.2025).

[22] Mathias [Visuelle-Musik] Brüssel. *MPE-DIY-Device*. July 2019. URL: `https://github.com/Visuelle-Musik/MPE-DIY-Device`. (accessed: 11.03.2025).

[23] Mathias Brüssel. *MPE-DIY-Device*. July 2019. URL: `https://hackaday.io/project/166803-mpe-diy-device`. (accessed: 11.03.2025).

[24] Jean-Philippe [jphi] Civade. *Pitchbend + Aftertouch + Modulation MIDI à base d'Arduino - Civade.com*. Dec. 2018. URL: `https://civade.com/post/2018/12/16/Pitchbend-Aftertouch-Modulation-MIDI-%c3%a0-base-d-Arduino`. (accessed: 11.03.2025).

[25] J-M-L Jackson. *Send midi pitch bend with a joystick*. May 2021. URL: `https://forum.arduino.cc/t/send-midi-pitch-bend-with-a-joystick/859518/2`. (accessed: 11.03.2025).

[26] Hauke Menges. *Chords and scale notes of C Minor Gypsy*. 2025. URL: `https://feelyoursound.com/scale-chords/c-minor-gypsy/`. (accessed: 28.04.2025).

[27] David Wessel et al. "A force sensitive multi-touch array supporting multiple 2-D musical control structures". In: *Proceedings of the 7th International Conference on New Interfaces for Musical Expression*. NIME '07. New York, New York: Association for Computing Machinery, 2007, pp. 41–45. ISBN: 9781450378376. DOI: `10.1145/1279740.1279745`. URL: `https://doi.org/10.1145/1279740.1279745`.

[28] KontinuumLAB. *Making a DIY analog force sensor under quarantine, with the Kontrol Freak.* Mar. 2020. URL: `https://www.youtube.com/watch?v=gCBbIeI4xTE`. (accessed: 11.03.2025).

Appendix A

Appendix

A.1 Questions used in the evaluation

Table A.1 and table A.2 Questions Used in the Evaluation

Question
1. How responsive is the instrument to your breath and finger input? (Extremely unresponsive: Significant delay or difficulty in producing sound or recognizing finger input. Feels sluggish and unplayable - Exceptionally responsive: Immediate and precise reaction to every breath and finger input. Feels seamless and perfectly in sync with the player's intentions) 2. How easy was it to play this controller or to understand how the controller works? (Extremely difficult: The controller feels unintuitive and frustrating to use. Requires significant effort to produce sound or execute basic techniques - Extremely easy: The controller feels effortless and fully intuitive to play. Everything, from sound production to controls and key layout, perfectly accessible and straightforward) 3. How comfortable was the physical layout of the interface? (Extremely uncomfortable - Extremely comfortable) 4. How confident did you feel using the controller after a short period of time? (Not confident at all - Extremely confident) 5. How intuitive did the controls (buttons, FSRs, joystick, ribbon) feel? (Not intuitive at all - Extremely intuitive) 6. How enjoyable was it to play this controller? (Not enjoyable at all; frustrating to use. - Exceptionally enjoyable, a pleasure to play) 7. How expressive did the FSR + joystick combo feel when performing? (Not expressive at all - Extremely expressive) 8. How responsive was the ribbon interface? (Not responsive at all - Extremely responsive) 9. How well could you control dynamics (e.g. velocity, aftertouch)? (Not at all - Extremely well)

Table A.1: Table with the questions used in the evaluation. Most of the questions allowed for an answer on a rating scale from 1 (most negative) to 7 (most positive).

Question
10. Did the controller respond quickly to your input? (Extremely slow - Extremely fast)
11. Were the LED indicators helpful in showing tuning and octave settings? (Not helpful at all - Extremely helpful)
12. Did you experience any technical glitches or unexpected behaviour? (open)
13. How appealing did you find the controller's design? (Very unattractive, poor design - Extremely appealing, outstanding design)
14. How would you rate the perceived quality of the instrument's materials and construction? (Very poor: Feels extremely cheap and flimsy, with obvious signs of low-quality materials or poor craftsmanship. Likely to break or malfunction quickly - Excellent: Outstanding build quality, with premium materials and flawless construction. Feels highly durable and exudes a sense of luxury and precision)
15. How satisfied are you with this instrument overall? (Extremely dissatisfied: The instrument fails to meet expectations in most or all areas. Significant issues make it unusable or unpleasant - Extremely satisfied: Completely fulfils or surpasses all expectations. Virtually flawless, highly enjoyable, and an exceptional experience overall)
16. What stood out to you the most about the Teensy powered MPE controller? (What did you like the most about using the controller?) (open)
17. What improvements would you suggest for this Teensy powered MPE controller? (What would you improve or change?) (open)
18. The controller also offers two interfaces. Do you think the option to have two interfaces is a useful addition or not? (Yes - No)
19. Can you think of any other useful addition that comes with having multiple interfaces as a wish for future iterations of the prototype? (open)
20. Do you have any additional feedback or thoughts about this controller? (open)
21. Would you consider using this controller in your music production or performance? Why or why not? (open)
22. Muscial Experience (in years)
23. Experience with MIDI controllers (None - Some - Extensive)

Table A.2: Table with the questions used in the evaluation. Most of the questions allowed for an answer on a rating scale from 1 (most negative) to 7 (most positive).